Get Started With Angular

An opinionated guide

D. M. Bush

Version 2

GET STARTED WITH ANGULAR

By D. M. Bush

Version 2

Table of Contents

Who is This Book for?

This book is for anyone who wants to get started using Angular quickly but doesn't want to sift through all the possible ways one might create an application using Angular.

You see, most books will show you the multiple ways you could do something and leave the choice to you. That's fine if you have the experience and the desire to know when to pick what, but most programmers I know just want to get going with some way that will work most of the time.

So, in this book, I guide you through the process of building an Angular application as if I were sitting down with you one on one. This will get you started quickly. If you want the multiple ways of doing everything there are plenty of articles and books that do that.

Because this book focuses on getting you going, and I wanted to get this out into the market as soon as possible, there will be things I don't cover. This isn't "Everything You Need to Know about Angular." It is a book that will get you started in the right direction.

I'm also not going to cover anything about HTML or CSS. I'm going to assume that most people either know that, have a web

designer they are working with who can help them with that, or you will go get that information from another source. HTML and CSS would require another book at least the same size as this one.

You won't need to know HTML or CSS for this book though. I will give you all that code in the tutorial section.

While we do focus quite a bit on how-to write Angular, which you can get from just about any Angular book you pick up, this book also focuses on design patterns and architecture. It is no good knowing how to code the individual parts if you don't know how they all fit together.

For this reason, the first section of this book concentrates on a lot of foundational concepts. It goes just deep enough to begin to form a mental picture of how Angular works before we start coding anything "real." Once we've laid that foundation, we move on to a basic CRUD application. Don't skip this part. It isn't just code to illustrate the foundation, it is also code to illustrate additional concepts. While you may not fully grasp concepts in the beginning of the book, once you've done the exercises, maybe a couple of times, you should begin to see how everything fits together.

You may wonder why the whole book isn't just one big tutorial. If I could have written the entire book as a tutorial, I would have. Unfortunately, I felt that starting right off with an application was like building a house with no foundation. Angular introduces a lot of new concepts you probably haven't seen before that would get glossed over if we tried to cover them while working through a tutorial.

This isn't a book about programming. If you are new to programming, this book isn't going to suddenly turn you into a programmer. I'm going to assume you've already been programming before using a language such as C#, Java, or JavaScript.

This isn't to say that you can't learn something by reading this book that is helpful. What it does mean is that I may talk about concepts you've not yet encountered. If you don't know something, look it up.

Finally, this book is for me. I find the act of writing forces me to learn the material even better. I hope it helps you on your journey as well.

How to Use This Book

This book was written the same way I would teach this material, so the best way to use this book is to read it in order.

You should also follow along in a code editor. When you get to the sample application, write the code. The only way you'll really learn this is if you use it.

The code is presented using a `monospaced` font. Places where I've used a monospaced font in line with a paragraph means I'm referring to code related things. Classes, functions, methods, directories, or filenames.

At this point, I need to mention that I've done my best to format the code in a way that work well in both the hard copy of the book and the Kindle version. If you are using a reader, you may want to adjust your settings so you are better able to read the code.

Once you've been through this book, you'll probably want to hang onto it as a quick reference to lookup some of the more complex topics until they stick. I know I plan to keep it handy myself.

Finally, you'll probably want to work through the main exercise multiple times. Angular has a lot of new concepts that may take a few runs through to understand fully.

Any problems, as well as the step by step source for the main exercise, can be found at https://github.com/DaveMBush/get-started-with-angular help your fellow programmer and report any issues using the GitHub issue tracker there.

Why Angular?

With so many JavaScript libraries, frameworks and platforms available, picking a framework, platform, or picking a set of libraries to create your own framework can seem confusing. I'm familiar with several libraries and most of them all have technical merit of some sort. So, technically, you really can't make much of a mistake.

So, here are some considerations I used when I decided to hang my future on Angular.

Who?

First, who is behind this framework? I start here because as an enterprise developer, I want to pick something that I have some reasonable hope will be around several years down the road. Using this measure, the three frameworks that surface to the top are:

- Angular – primarily funded and used by Google and secondarily funded by Microsoft via TypeScript. They also have a huge list of contributing developers and several supporting projects that have been added in a short amount of time.
- React – primarily funded by Facebook
- ExtJS – commercial product

Not only are these the three that make the cut, they are the three I have personal experience with.

Documentation

The next thing I tend to look at when selecting a framework is the documentation. While I was learning React, I found it difficult to find documentation. This is not so much because it didn't exist, but because the library changes so fast, it is hard to know for sure if the explanation for how to do something is accurate for the version you are currently using.

Angular, on the other hand, had superior documentation even during the release candidate cycle which was arguably the most tumultuous time for Angular. Also, since Angular version 2.0 released, the APIs have stayed amazingly consistent.

And then there is ExtJS which has documentation on the same level as Angular but, trying to get help on specific issues tends to be a bit frustrating.

Completeness

The next thing that I find important is completeness. That is, if I choose this platform, how many other decisions will I need to make as a result.

While React has gotten better in this regard, there are still a lot of decisions that need to be made. React just deals with the DOM, and even there, you will need to find your own set of controls once you get past basic HTML controls. You will also need to make decisions about: state management, how to make REST calls, how you will package the code for deployment, as well as many others. I once listened to a podcast that stated you typically need to make forty different decisions before you ever start programming an application using React.

On the other end of the spectrum, is ExtJS. This gives you everything you need. No decisions to make.

And then there is Angular. The only thing you could argue that you don't get with Angular is a set of controls. Out of the box, everything else you might need is there.

Ecosystem

Another consideration is the overall ecosystem. If you need something that isn't provided by the library or framework, what are the chances you'll be able to get it from someplace else? If you have a question, can you find the answer easily?

It is here that ExtJS falls flat on its face. There are a few places that provide additional controls. But in general, getting help, finding additional controls and even finding programmers who are willing to write code using ExtJS is difficult.

Contrast this situation to either React or Angular and either of these two become clear winners. However, I give a slight edge to Angular because finding the correct answer is a little easier with Angular compared to React.

Flexibility

How much control will you have over the layout of the HTML? If all you are using the framework for is filling out regular forms, ExtJS might be all you need. However, in one shop I worked at, the UX team came up with requirements that required us to manipulate the HTML and CSS. Something that, while possible, was also more difficult than it should have been. This is because all layout in ExtJS is done using JavaScript. I think they were

aiming at implementing something like Java Swing using JavaScript.

If you want to plug in an alternate library, is that possible? Since React is a library, this is a given. Angular, while more complete also allows you to use alternate libraries. But, while adding in additional libraries is theoretically possible, tends to be difficult at best.

Design Patterns

Does the library or framework come with recommended design patterns that are established and recognized by the community. Are these design patterns complete?

Here again, React and Angular pass with flying colors, while ExtJS fails miserably. The problem with ExtJS is that it implements "MVC" and "MVVM" but doesn't implement either in a form that is recognizable as what most people mean by either of those terms, nor are they complete implementations.

Enterprise Appropriate

All three frameworks can be considered "Enterprise Appropriate" in the sense that you can use them to build enterprise level applications. But being Enterprise Appropriate goes further than that. First, are the organization's current developers going to be able to learn this framework? Will they understand the design patterns well enough to use them appropriately? Are there use-cases that won't work well with the library or framework? Can you find other programmers who have experience with this framework or library? If they know it, will you need to train them to use it your way or are they likely to know it your way already?

That last one is something you need to consider. Just because someone knows React, for example, doesn't mean they necessarily know the way you use React. On the other hand, you can be pretty sure that anyone who has built an Angular application before is going to be able to adapt to your environment with little to no additional training.

In fact, as I've visited other organizations using Angular, I've been pleasantly surprised at how similar everyone is using Angular given that at this point we've all essentially blazed our own trail with a small amount of guidance from what we can find online.

For example, it seems everyone has committed to the Angular CLI and using NgRX for state management rather than using the older MVVM framework from AngularJS or working from some pre-existing seed project.

Summary

If you can't tell, I'm not a fan of ExtJS. But between React and Angular, I feel the edge goes to Angular.

Not the Old Angular

Angular 1, officially referred to as AngularJS, was a tool developed using what could be referred to as “Old School” JavaScript development practices. This was a time when JavaScript was seen more as a secondary add-on. Because of this, we just included our JavaScript directly in our HTML files and didn’t worry too much about performance. The server rendered the bulk of our code and that is where we spent most of our time optimizing our code.

With the introduction of frameworks such as jQuery, knockout.js and others, our web applications quickly shifted from being primarily about HTML that was rendered from the server to being more and more about JavaScript and less about HTML rendered on the server.

AngularJS was developed in during this time. As the popularity of AngularJS and other similar Single Page Application (SPA) frameworks took off, it became clear that we needed to start thinking about our JavaScript a little more intentionally.

It was during this time that Node became a popular tool for managing our JavaScript builds. Various ways of bundling and minifying JavaScript, HTML and CSS were developed. Many are still available and have heavily influenced many of the more recent tools.

But this is where the trouble for many developers began. So many choices! If you are new to this kind of development, all the choices can be overwhelming. Even to those of us who

"grew up" during the maturation of JavaScript often have trouble keeping track of what the latest favorite tooling is.

The new Angular, officially referred to as Angular, takes a new look at this problem. While many existing frameworks leverage what is available today, Angular, can be considered a bleeding edge framework while leveraging many of the tools and concepts that we've learned while making the transition to the world of SPAs.

One place where the Angular community has learned from its predecessors. The Ember team had developed a command line interface for their framework that allows a user to type in a command and generate a JavaScript file that had the boilerplate code already written and ready to be modified. It also can scaffold out a new project, run a development server, and run build tools that generate the files needed to place on the server, already intelligently bundled and minified.

Now, while the Angular team was working on this great new version, another project started up to make getting started with Angular much easier than doing everything by hand. They forked the Ember project and started tweaking it to support Angular. The result is a development environment that is almost as easy to use, if not easier to use, than what we were doing in the past with support for many Angular features most beginners would never even attempt to implement if they had to implement them by hand simply because they wouldn't even know about the option.

However, to have the benefit of using the Angular CLI, you are going to need to prepare your environment with Node, NPM, and the Angular CLI. You might want to consider changing editors. Many of the editors you are familiar with, like Eclipse and

Visual Studio, are not as powerful as some of the newer editors that have been developed recently for programming using TypeScript and JavaScript.

Community

Later in this book, we will create a project. The source code will be hosted on GitHub. The good news is, I can keep that up to date. The bad news is, books can't be updated without putting out a new version.

My intent is to use the issue management system associated with the tutorial project to also report problems with the book and keep the book as up to date as possible.

https://github.com/DaveMBush/get-started-with-angular

Setting Up Your Environment

Node/NPM

To use the Angular CLI, you are going to need to have Node and NPM installed on your computer. The current recommended version is 6.9.0 or higher.

Now, before you install Node, you'll want to install the Node Version Manager.

You see, eventually you are going to end up in a situation where you are going to need two different version of node installed on your computer because one project was built and depends on one version of node and another version was built and depends on another. Ideally, you could just upgrade the lower version to the higher version. But what if you can't?

If you are running Windows, you can get NVM for Windows from https://github.com/coreybutler/nvm-windows/releases. If you are running anything else, you can grab it from https://github.com/creationix/nvm.

Once you have NVM installed, you should be able to run

```
nvm install version.number.here
```

At the command line. Where '`version.number.here`' is the version of node you want to install.

Or, you can just run

```
nvm install node
```

To install the latest version of node.

You can run `nvm install` for each version you want installed on your computer. To see which versions are already installed, you can run

```
nvm ls
```

And to switch to a specific version you can run

```
nvm use version.number.here
```

`nvm use` is sticky so it will be the version you are using the next time you reboot your computer.

NPM

When you install Node, you also get the Node Package Manager (NPM). It is not a separate install. There are only a few commands you need to know to get started, but if you are really interested in the various commands that you can run, check out https://docs.npmjs.com/.

The three main commands you will use are:

```
npm install -g packageName
npm install --save packageName
npm install --save-dev packageName
```

The first line installs the package into your system globally. That's what the -g is all about. You would use this command for packages that you always want to have available from the command line. Depending on how your computer is configured, you may need to use `npm install -g` with elevated privileges.

The second and third lines install `packageName` locally and registers it in `package.json` so that when someone starts

working on the project, they can get all the packages they need to have installed to work with the application installed. The difference between the `--save` and `--save-dev` switch is that `--save` is for modules that will need to be installed when the Node application is released and `--save-dev` is for modules that are only needed for development. Typically for the build process.

The safest thing is to just always install using `--save`. If we ever implement server-side rendering, most of our Angular related code will need to be installed using `--save`. Build tools can be installed using `--save-dev`.

Angular CLI

Next, you will need to install the Angular CLI.

Now, you may have already used some other tutorial that encouraged you to code everything by hand. They aren't wrong and one could argue you learned some of the inner workings of Angular better that way. But, if you want to be productive with Angular, make fewer mistakes, conform to the Angular style guide, and generate Angular code the way you are likely to see it implemented in most organizations, you'll follow my advice and use the Angular CLI.

If after finishing this book, you've decided it isn't for you, I guess you can go back to doing things "the hard way."

You'll want to install the Angular CLI globally so that you will be able to execute the `ng` command to create the application, or create new files in the application.

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

Recommended Angular Editors

Now, you can keep using the editor you are currently using if you want to, assuming that is supports TypeScript. But, my experience is that the editor you typically use for server-side development is less than satisfactory for developing client-side code. Yes, I understand that you are already comfortable with it. However, I think the improved syntax helps that are available in the two editors I recommend are well worth the investment in the time it will take to learn them.

VS Code

Visual Studio Code. Don't let the "Visual Studio" part fool you. While VS Code was developed by Microsoft and can edit many different file types. It is not Visual Studio for Windows that is primarily aimed at Microsoft .NET. In fact, VS Code is written using TypeScript and runs on Windows, Linux or Mac. It is also Open Source and FREE!

It is also not, technically speaking, an Integrated Development Environment (IDE) though you'll find as you install the various plugins that are available, the line tends to look very blurry.

VS Code is a code editor that was initially built primarily for JavaScript and TypeScript, so it handles those languages very well. But because it has a rich Application Programming Interface (API) there is a rich eco-system of plugins available that allow it to support all kinds of languages including C#, Java, PHP, F# and others.

The downside to using VS Code is that it will take some time to get configured correctly.

Some plugins that I've started using in my own environment that you might want to consider using are:

- Angular v4 TypeScript Snippets
- Bookmarks - Allows you to set a bookmark in your code so you can go to another page and come back to where you left off.
- Bootstrap 3 Snippets
- Color Picker - Great if you are using colors in CSS.
- Debugger for Chrome - Allows you to debug the application using VS Code instead of web Developer Tools.
- Git Lens - Shows you who worked on the code last by "inserting" text into your code that you can't edit and blends into the background.
- Sort Typescript Imports
- Stylelint - Helps make sure your CSS is valid and follows specific rules
- TSLint - Validates your TypeScript
- TypeScript Hero - Helpers for working with TypeScript. Auto imports is one of my favorites.

WebStorm

The other editor I like is WebStorm. Unlike VS Code, WebStorm is a commercial product. But, instead of installing a bunch of plugins to get started, it does everything you need once it is installed.

I've used both and have things I like about each and miss from the other. The main thing I miss from VS Code is the Git Lens plugin. The main thing I miss from WebStorm is that it integrates with GitHub better.

Starting a New Project

Using the Angular CLI

Now, you've installed all the tools that you need. The next step is to create your first project.

While there may be integrations with the Angular CLI in your editor, to be sure that everyone can use this material, we are going to do this all from the command line.

The first thing we are going to need to do with our editor is to create a new project so we have a place to create our new Angular application using the CLI. Then on the command line move into the directory that is the top of your project and run the following line:

```
ng new projectName --directory=.
```

Don't worry. This will run for a while. You may even think it has hung. It hasn't. It is still working.

Once you have everything installed, you should be able to run

```
npm start
```

This will build the project and run the development server on port 4200. To see that everything is working, start up your browser and navigate to http://localhost:4200

Directory Structure

Now, you may have already been tempted to look at the directory structure and either got lost looking at all the files or

just started opening things randomly to try to figure out what everything is. I understand that. But, let's step back and try to unravel what we are looking at here.

.angular-cli.json

First, at the top-level directory, we have a set of project related files. The first one you may not recognize is the `.angular-cli.json` file. Most of the stuff in this file you will never change but there are a few things we need to point out here.

In the `apps` section, there are three subsections that you need to be aware of. In the order they appear, `assets`, `styles`, and `scripts`.

You'll notice the `assets` section already has two entries. `"assets"` and `"favicon.ico"`. `"assets"` is where we will place any files that our app needs to access directly from our templates. Generally, I try to avoid putting files in `assets` because these files don't go through the bundling and minification process. Any files or directories you place in this section are relative to the `src` directory when you access them from your html templates.

`styles` is where we include global CSS files we want our application to use. Once again, there is already an entry here. You can add others if you need to because these files do end up getting bundled and minified. You'll see later that this is where we add other CSS files that allow our applications to use Bootstrap and vendor specific CSS. Once again, any files you put in here are relative to the `src` directory.

Finally, `scripts` is where we add any additional JavaScript files we want to include. Once again, this should be avoided.

One common mistake is to include jQuery here. But, because jQuery and Angular manipulate the DOM in two very different ways, they have a reputation of not playing nice together. Don't do that.

In general, you will be `require`ing external JavaScript and TypeScript code into your application and you should avoid putting any files in the `scripts` section at all.

Accessing the DOM

While we are talking about DOM manipulation, this is a good place to mention that you don't want to use any alternate method of manipulating the DOM. Don't use jQuery, don't use standard JavaScript APIs. Only use what has been provided by Angular.

There are two good reasons for this. The first, which I've already mentioned, is that you can confuse Angular by manipulating the DOM in a way it doesn't know about. That's reason enough.

But, there are two ways you can run your Angular code where the DOM won't be accessible. One is by running Angular Universal, which is a fancy name for being able to render Angular on the server. You would want to implement this so that the first page renders faster and so that more of the search engine spiders are able to see the content of your pages.

Another way of rendering you might be interested in allows you to run your entire Angular application in a Web Worker. The Webwork will fire messages up to the main application to tell it how to render into the DOM, but it doesn't have direct access to the DOM.

If you do access the DOM directly, you should ensure that you are doing so for very good reasons and you should try to do it in a way that will continue to work using Angular Universal or Web Workers.

CSS Animations

One way we've started enhancing the user experience of our web applications is by using CSS animations. If you are used to using these, you are going to want to use the Angular Animations library. Something we won't be covering in this book.

In Angular, instead of using CSS to make an element visible or not, we'll be using other code to put the element in the DOM or take it out of the DOM. The problem then becomes, since the element didn't previously exist, you won't be able to animate it using CSS animation. Using the Angular Animation API solves this problem.

package.json

`package.json` is central to every application that runs in Node. It tracks the current version of the project, where it is hosted, what packages it depends on, and more.

We've already kind of described two of the sections in this file. The `dependencies` section is where all the dependencies we install with the `--save` flag get registered. If we install something with the `--save-dev` flag, they are registered in `devDependencies`.

But, the place I want to focus on next is the `scripts` section. You'll see that there are several scripts that the CLI has placed in this section for us.

Remember I had you run `npm start`? How did NPM know that is was supposed to build and run our code? Because it knows that when we said, "start" what we really wanted it to do was to run `ng server`.

The full syntax for starting any one of these scripts is `npm run scriptName`. But, start, and test are so common that we don't need to specify "run".

A cool little trick that you'll see us use later is that you can separate multiple lines with `&&` so you can run multiple lines with one script as well as having one script call another.

tslint.json

One of the things we can make heavy use of in Angular is linters. Linters ensure that our code all follows style recommendations. Where should curly braces start? When do we use space and when do we not? The `tslint.json` file specifies the rules that the Angular community suggest that we use. For the most part, I agree with these rules. But, there are a few places where I don't and I've overwritten them with my own preferences in this file. We will discuss this in more depth during the code walk through.

tsconfig.json

This is a file you won't need to change. At least not right away. This just tells TypeScript how to compile the code.

karma.conf.json

The last two files we want to discuss are files related to testing our code.

The `karma.conf.js` file is how we configure our unit tests. There is nothing in here you need to change. It will run the tests in Chrome and every time you change your code, it will rerun the tests.

protractor.conf.js

Similarly, `protractor.conf.js` is where we configure our application level tests. Protractor uses selenium under the hood to drive the web browser. Using this, we can write tests that verify that the application works the way we expect it to.

node_modules

Next, let's walk through the various directories that show up in our application. If you've been using node for even a week, you should recognize that `node_modules` is the directory where all our node packages get installed. Anytime you run `npm install --save` or `npm install --save-dev` the code for the package you are installing shows up under `node_modules`.

One of the great features of this is that we don't have to put the `node_modules` directory into version control. In fact, you really shouldn't do that. It would put way more under version control than is reasonable.

In fact, try this. Right now, delete the `node_modules` directory from your project and once it is gone, run `npm install`. This will bring back all the files we just deleted. To

verify that this worked, now rerun `npm start` and browse to your application at http://localhost:4200

src

The `src` directory is where we write our Angular application. We'll come back to this in a while and walk through what we have so far.

e2e

While all our unit tests are written parallel to the code they are testing, your application level tests, the ones that use protractor, are placed in this directory.

dist

You don't see the `dist` directory yet because we haven't built the application yet. But if you run `npm run build`, you will see that this directory shows up.

Everything you need to run your Angular application is in this directory. To deploy it onto the server, all you need to do is to copy and paste this into the root directory for the application.

src Directory

Next, let's open that `src` directory and look around.

There are several files right under the `src` directory. Several are just necessary to making Angular work and are files you should never need to modify. Rather than detailing each one, I'm just going to focus on the ones need to know about now.

index.html

This is the page that holds your app. If you open it, you'll see it is basically a shell. In general, you don't want to touch this file. I know the temptation might be to put some HTML around `<app-root>`, or worse add in JavaScript tags. But, using Angular, there are better ways of handling both.

polyfills.ts

What's a polyfill?

As JavaScript has matured, various capabilities have been added to the language. The newer browsers implement these new capabilities. But, the older browsers are still using the old stuff. Most of the new capabilities are just additions that can be tacked on to the old language. A polyfill is code that we can ship with our application that will "patch" the older browsers so that we can write code to the newer JavaScript specification.

You might think, "but, I'm always going to be working with the newer browsers!" Lucky you!

style.css

As we saw when we looked at the `.angular-cli.json` file, `styles.css` is where we put our application specific but component general style information.

main.ts

This file is our basic bootstrap file. It puts us in production mode if we've compiled for production and it loads the first module.

This file is important because it is possible to start the application in browser mode, or in other modes as various

modules become available. This how Angular is able to render code on the server and will be able to render in a Web Worker.

You may wonder how angular knows to kick off the application using `main.ts`. Well, remember the `.angular-cli.json` file? All the information is in that file.

Angular Basics

Before we move too far along, we need to step back and get a big picture view of how Angular works. There are three major types of Angular code you will be working with rather consistently. Modules, Components, and Services.

Modules

Modules are a relatively new feature of Angular. Modules existed in AngularJS, but the team tried hard to keep them out of the new Angular framework. Eventually, they decided we really need modules to write cleaner code.

You might be tempted to think of modules as a kind of namespace. They really do much more than group our code together like namespaces; a better metaphor might be micro-libraries.

Here are some of the key benefits of modules:

- Define what components are available to be used by the application that uses the module.
- Define what services and other injectables (we'll get to those soon) are available.
- In the case of Lazy-Loading and the application module, defines the entry component.
- Allows us to hide internal components from external use.

Components

Components are the visible part of our application. Components use HTML like templates along with global and component

isolated CSS to specify what they look like. What may feel quite a bit different to how you've dealt with similar frameworks in the past is that the HTML and the TypeScript are two parts of the same whole. At no point should you consider the TypeScript code that is part of your component as either a Controller or a Model in an MV* framework.

Don't worry, we still are going to use a design pattern that separates our View from our Business logic, we just won't be doing that here.

The HTML for our view is referred to as the "template" and this template code has a specific syntax that is going to looks like HTML but isn't entirely HTML. Think of it as "HTML with syntax sugar."

Attributes

We are normally used to assigning values to attributes in our code to make the attribute behave a particular way. For example, you would assign a value to the style attribute to make an element look a certain way.

But what if what we want the control to look like depends on some other condition. Now we need to make the attribute look at some variable or function. But how is the code supposed to know that the string is a variable instead of the actual value? In Angular, we wrap the attribute in square brackets.

```
<div [style]="someVariable"></div>
```

This simple syntax tells the Angular compiler that `someVariable` is a value in our class rather than the literal that will never change.

Similarly, if we have an event that needs to call a function in our code, say the click event, we wrap the event name in parenthesis.

```
<div (click)="myclickHandler($event)">
</div>
```

All events work like this. You'll notice I also passed in $event. This tells the compiler to pass the event payload to the function. If you don't include it, the function won't use it. You can also pass other variables, but if you need the event to be passed in, this is how you do it.

Now, there is one additional twist to this syntax. Occasionally, you'll have an attribute that can function as both a property, like we used with style, and as an event. The most common scenario is when the value of a control changes. In this case, we wrap the attribute with both.

```
<div [(ngModel)]="someVariable"></div>
```

The truth is that this is just a convenience syntax. Under the hood you have `[ngModel]` and `(ngModelChange)`.

The other odd-looking syntax you might see are `*ngFor` and `*ngIf`. These allow you to loop through data to create a list using one block of HTML code, or in the case of `*ngIf` selectively include or exclude HTML fragments.

If you are accustomed to using CSS show and hide to manipulate what gets displayed in the DOM, you will want to ditch that habit in favor of `*ngIf`.

We cover these and other template syntax in more detail in the "Template Syntax" chapter.

Services

Services are code that do things for us. Most of the time, they make a request to the server to get data or send data back. But this is not the only way they can be used.

In some implementations, you'll find that Services are used to share data between components.

The chief thing you need to know about a service is that it is `@Injectable` into other classes using Dependency Injection. Because of this, they need to be registered with a module that defines the class we want to inject the service into prior to when that class needs access to the feature.

Unlike AngularJS, Services are not Singletons in the true sense of the word. If you define a Service in a module, then the same instance will be available everywhere. But, if you define a service in a component, it will be available to that component and any of its children.

You make a service available by defining it in the component's or module's providers array as part of the decorator:

```
@NgModule({
  providers: [
    SomeService,
    SomeOtherService
  ],
  bootstrap: [AppComponent]
})
export class SomeModule { }
```

Then, in the class that needs the instance of the object you inject it into the constructor like this:

```
export class MyClass {
  constructor(someService: SomeService,
      someOtherService: SomeOtherService)
{
  }
}
```

This makes the services available to the constructor. If you prefix the parameters with `private`, `protected` or `public`, the parameter names become members of the class.

```
export class MyClass {
  constructor(
    private someService: SomeService
  ) {
  }
}
```

TypeScript Basics

For the most part, TypeScript feels a lot like JavaScript. Most people pick it up without having any formal training. But, there are a few things that will make you much more productive if I just let you know them up front.

Variable Declaration

There are three ways of declaring a variable in TypeScript. You can either use the JavaScript `var` keyword like you've always done or you can use the `let` keyword or the `const` keyword.

But first, what problem are we trying to solve?

In the old JavaScript world, we would declare variables in a block of code, but where-ever we declared that variable, the actual declaration was "hoisted" to the top of the function. In fact, there never was anything like block scope in JavaScript. Just function scope.

This caused one problem that was rather common. If I create a for/next loop that called an asynchronous function, the asynchronous function will use the last value of the for/next incrementer when it finally runs the asynchronous functions.

The primary benefit of using `let` or `const` is that they effectively provide for block level scope so that we can write code like I described above and it will behave in the way we would expect from other languages.

Unless you explicitly want to avoid block level scope, you should never use the `var` keyword to declare a variable in

TypeScript. This falls under the "just because you can, doesn't mean you should" rule.

In my experience, you will use `const` more often than `let`. Here's the difference.

If you are declaring a variable that will only ever be assigned one value, you declare it using `const` and immediately assign it the value. What isn't obvious is that changing the contents of an object does not change the value of an object. So, doing something like this:

```
let myArray = [];
myArray.push({});
```

Would be more valid as:

```
const myArray = [];
myArray.push({});
```

Because pushing something into the array doesn't change the value, or the pointer, of `myArray`. It only changes the content. `myArray` still points to the same object it did prior to calling `push()`.

Types

The main thing that makes TypeScript what it is, is that it allows us to type-check our code. You don't have to. In fact, there are times when this might get in your way. But, you have a choice.

By default, TypeScript uses inference when it can to figure out the type of a variable. This is important because I can bet one of the first errors you are going to see is a type mismatch error.

For example, you might try to do something like this:

```
let v = 'abc';
// some other code, and then ...
v = 20;
```

But that won't compile because TypeScript saw that v is pointing to a string and is now assuming v will always be a string. When it sees we are trying to assign a number to v, it complains that the types don't match.

However, you could do something like this:

```
let v: any = 'abc';
// some other code, and then ...
v = 20;
```

This tells the TypeScript compiler that we are OK with the variable v being any type which is how JavaScript works by default.

We can type a variables as any of the `Classes` and `Interfaces` that are either part of JavaScript or that we, or a third party, create. We can also type them as `boolean`, `number`, `string`, `array`, `enum`, `any`, `void`, `tuple`, `never`, `null`, and `undefined`.

I'm going to assume that most of the above types are self-explanatory except for `tuple`, `never`, `null` and `undefined`. So, let's dig a little deeper on those last few.

Tuples

A tuple is a highly controversial type. But here is what it lets you do. It allows you to return a highly defined array. That is, I can specify that a function returns an array that has a specific

number of elements and each element is a specific type. It has its uses, but it is probably one of the types that you want to reserve for special cases.

Never

The never type allows you to specify that a function never returns. There are two reasons why this would be true. First, you've entered an infinite loop or second, you've thrown an exception. Again, not something you are likely to use.

Null and Undefined

You can also explicitly specify that a type can only handle null or undefined. But what is much more likely is that you specify that you don't want to use these.

Combined Types

So, let's say you have a parameter or a variable that accepts multiple types. You could just use any and go on your merry way. But, wouldn't it be nice if you could say, "I want this type to be either a string or a number."? Well, you can. Simply by using the pipe operator between types.

```
let v: string | number = 'abc';
```

But we can take this even further. Let's say we want to make sure that the variable is a particular class type that we also want to be sure implements a specific interface. For that, we use the ampersand.

```
let v: Person & Manager;
```

And while we are at it, what if we want to make sure that a variable only accepts string types that are not null or undefined?

By default, the compiler allows null and undefined to be assigned to anything, but there is a compiler switch that turns that feature off. If you use the compiler switch and you want to allow null or undefined, you'll need to use the pipe operator to include them.

Interface

For the most part, a TypeScript Interface looks a lot like an Interface in other languages, but there are some differences that you need to know about.

First, you don't need to create a class that implements an Interface and then instantiate an object from that class to have an object of an interface type. If you stop to think about it, this makes sense. The problem is, there are a lot of people teaching TypeScript who are still using interfaces this way.

In JavaScript, you can create an Object Literal. TypeScript adds to JavaScript. So, it only makes sense that TypeScript also allows you to create an Object Literal.

Let's say you have a parameter that takes an interface of type Name. If the object we pass in conforms to the interface definition, the code will compile.

```
interface Name {
  firstName: string;
  lastName: string;
}

// function that takes a Name
// as a parameter
foo(name: Name) {
}
```

```
// call the function with an object
literal
foo({firstName: 'Dave', lastName:
'Bush'});
```

Optional

We've been talking a lot about Parameters and Interfaces. In both cases, we often want to define a parameter or a property as optional.

For example, most people have a middle name, but our Name interface doesn't account for that. If we added it, we'd want to make it optional since it is possible for that to not be included. On the other hand, we don't want people adding whatever they want.

The way we make sure a parameter is optional is by placing a question mark after the property, but before the colon.

```
interface Name {
  firstName: string;
  // middleName is optional
  middleName?: string;
  lastName: string;
}

// foo() now takes an optional
```

```
// name parameter
foo(name?: Name) {}
```

this

The `this` keyword in JavaScript is probably the hardest concept to fully understand. And while recent advances in the language have helped tame it, it still doesn't fully conform to the model most people have in their mind of how an Object-Oriented language should behave. This is because JavaScript isn't an Object-Oriented language, it is a Prototypal language. There are similarities, but they aren't the same.

TypeScript, on the other hand, is more object oriented. I say "more" because it is only object-oriented in the places where you are taking advantage of TypeScript specific features, such as a TypeScript Class. If you create an object literal that has an inline function, you are back in JavaScript land.

In a class, if you have a method that calls another method in the same class, you must use the `this` keyword to go after it.

```
class SomeClass {
  someFunction() {
  }
  someOtherFunction() {
    this.someFunction();
  }
}
```

This may take some getting used to if you are coming from JavaScript where you can call any function that is in scope without using the `this` keyword. But, I can assure you that having this rule imposed on the language solves a lot of bugs

caused by the `this` side effects, so it is well worth the adjustment.

Arrow Functions

Fat Arrow functions, Arrow Functions, or Lambda Expressions all refer to the same concept. They are probably one of my favorite features of the latest version of TypeScript and JavaScript both because they allow us to write code with fewer characters and because they solve a very real problem that has confused JavaScript developers for years.

If you've written any serious application using JavaScript, one of the following scenarios will be familiar to you.

Any time you create an event handler, when the function gets called, the 'this' keyword isn't pointing to the object you are in, it is pointing to the context of the event that fired it. This could be null, a windows object, or something else. We often get around this problem by using the `bind()` function to wrap the context of the function.

What the arrow functions in TypeScript do is that they form a closure around the current 'this' context by taking advantage of how TypeScript is compiled into JavaScript.

You see, when your TypeScript code is compiled, every place you referred to 'this' it refers to a variable named '_this'. Inside the arrow function, they refer to this same _this instead of creating a new one or looking at the context the function was called from.

The main difference between a regular anonymous function and an arrow function is that we leave out a lot of junk.

```
let newFunc = function(x) {
  // do something with x
}

Compared to:
let newFunc = (x) => {
  // do something with x
}
```

But wait! There's more. If you only have one line of code in your function body, you can remove the curly braces.

Let's say you want to create an arrow function that returns the square of some number. You could write this as:

```
let newFunct = (x) => x * x;
```

Fat arrow functions return the value from the function automatically.

Decorators

I don't want to spend a lot of time on decorators. If you've been using .NET, you'll recognized decorators as "Attributes". Java programmers are probably used to calling them annotations.

Effectively, what a decorator does is that it adds additional information to a function, field, or class that marks it for special use. While you can create your own decorators, we will only concentrate on implementing decorators that have already been defined for us.

You'll know that something is a decorator because it is a symbol prefixed with the @ symbol.

Import vs Export

Last in our discussion of TypeScript are the keywords import and export.

But first, why do we need these keywords?

Well, if you are familiar with other languages such as C#, VB.NET or Java, you will recognize the concept of Import as the keyword that says, "Tell this file I'm going to reference code from that other file over there in here." And then when we compile our code, the compiler makes sure that we are using that other code correctly.

A similar thing happens in TypeScript, but in Angular we get the added benefit of also being able to use this information so that we only include the code we are using.

You see, in the old days, we would suck in entire JavaScript libraries just because we were using a few functions. But now with concepts like "Tree Shaking" that we will cover later, we can look at the actual code we are referencing and only include that code. This reduces the size and number of files that our customer must download to use our applications.

The export keyword, on the other hand, tells the compiler what functions, classes, and interfaces external code can reference. If it isn't exported, it is only available to code in the file it was declared in.

Access Modifiers

One of the problems that JavaScript has always had is that we are unable to specify that a member of a class is public, private or protected. Up until recently, we have had to simulate classes

using Functions. Even today, all the class keyword does is setup the keyword as a Function under the covers.

Typescript allows us to specify that our members are public, private or protected just like other object-oriented languages you may be used to. And just like other languages, public allows another class' function to access the member. Private only lets the class itself access the member. Protected lets the class or any child class access the function.

If you don't provide a modifier, TypeScript assumes that the member is public by default.

Constructors

In TypeScript, you specify a constructor using the constructor keyword.

```
export class Foo {
  constructor() {}
}
```

A common use case in object oriented programming is to pass parameters into your constructor and immediately assign them to member variables in your class.

```
export class Foo {
  firstName: string;
  lastName: string;
  constructor(
    firstName: string, lastName: string) {
      this.firstName = firstName;
      this.lastName = lastName;
```

```
    }
}
```

In TypeScript we can use shorthand notation for this. If you prefix the constructor parameter with public, private, or protected it will automatically turn the parameter into a member variable with the designated scope.

The following results in code that is functionally equivalent to the previous example.

```
export class Foo {
  constructor(
      public firstName: string,
      public lastName: string) {}
}
```

More

There is a lot more available in the language than what I've introduced to you here and knowing the parts I've left out will ultimately make you a better programmer and make your code more stable. But the parts I have introduced will get you going and will make you familiar with the parts you will see most often.

If you want more information, I invite you to visit the TypeScript site at https://www.typescriptlang.org/ where you will find the language is very well documented.

A Simple Application

Now that we have a minimal background on Angular and TypeScript, we can continue looking at the files that were generated by the Angular-CLI.

app.module.ts

The first file of importance is the `app.module.ts` file. This holds the `AppModule` class which is decorated by `@NgModule`.

The next question you should be asking yourself is, "What makes this code run?" And the answer to that is that it is loaded by `main.ts`. Remember? We looked at that already.

The `@NgModule()` decorator takes an object literal as a parameter. As you can see here, the literal has four properties: declarations, imports, providers, and bootstrap. One that isn't here that you will see a lot is exports. There are a few others that I am not covering here because they aren't needed for you to get some basic work done.

declarations

The declarations property takes an array of components that are part of this module. If you don't include a component here and the component is part of this module, you won't be able to use the tag in any of the templates. Think of this as the module saying, "I own this component."

imports

The imports property allows for you to specify the modules that this module will also be using. When you import a module, the components the child module exports and the components any modules it imports has exported are available to the parent module.

exports

The exports property allows you to specify the components in the module that you want to make available to modules that would import this module directly or indirectly. This is different from declarations because declarations define the components the module can use while exports define the components we will let other modules use. It is a way of making components private or public.

providers

The providers section allows us to specify any services that we want to have available to this module or any children of this module. What makes a service a service is that it is a class that has the @Injectable decorator. If an object isn't defined in the providers section, you won't be able to inject it into your code.

There is a temptation to define all our providers, services, @Injectables in the app.module.ts file. Sometimes we must. But you should attempt to define them as far down the module dependency tree as is possible. Keep related code as close together as is possible.

bootstrap

Finally, we come to bootstrap. This is where we list components that we want to have be immediately available when this module is loaded.

Most of the time what this is going to look like in our code is the top-level component of our application because that component needs to be available right away.

The bootstrap directive is only necessary in our top-level module and typically is not found in any other modules that might be defined.

app.component.ts

In the `app.module.ts` file, we specified that `AppComponent` is a component we want to use. This component is defined in `app.component.ts`.

Once again, you will notice that our class is decorated. This time it is with @Component and the string literal has three properties: `selector`, `templateUrl`, and `styleUrls`.

selector

The selector property allows us to specify what HTML tag we will use to specify where our template will go inside of other templates. You'll notice that the selector here is "app-root". We've seen this before. If you open the index.html file, you'll see that this is the tag we've included inside of our body tag.

By default, the Angular-CLI makes "app" the prefix to all our application selectors. If you want to change that to something else, you can open the `.angular-cli.json` file and change the apps.prefix property to the prefix that you want to use.

You'll also want to change the "app" string in the "directive-selector" and "component-selector" for your tslint.json rules. "app" could be an array of valid prefixes as well as a single string.

templatUrl

The `templateUrl` property points to the HTML file that holds the HTML fragment that will render where this.

You could also use the `template` property and put your html in line with the TypeScript. But I prefer to keep my HTML separate from my TypeScript code. Since this is the format that the Angular-CLI uses, I think it would be more work to change to all inline.

You can only use `templateUrl` OR `template`. Not both in the same class.

styleUrls

`styleUrls` points to an array of CSS files that this component should use to control layout of the component. While it allows you to specify multiple files, most of the time you will only have one CSS file that you will point to. Like templates, you could also use the `styles` property to provide style information in line with your TypeScript.

Sometimes you have no need for additional CSS for your component. The temptation is to remove the file. But, since we are bundling and minifying the code, there is no cost to keeping the file around, even if it is empty. You should just keep the file as part of the codebase because having all your components structured similarly will make your code easier to maintain and you never know when you might need the file.

Other Properties

There are other properties but these are the three that you will see repeatedly. We'll pick up others as we move through this training.

app.component.html

The `app.component.html` file is going to look a lot like any other HTML file. This file has a lot of information about other sites you can visit. But the main point of interest is how they get data from the `AppComponent` class to show up in our output.

The `{{}}` marker is basically an indicator that Angular should evaluate the expression between the braces and print the output where the braces are. In this case, we are looking for a public field or property in `AppComponent` named "title" and outputting the value where the braces are.

There are other ways of displaying data and sending data back into the TypeScript class that we will look at later.

app.component.css

There isn't anything here of interest. In fact, the file is empty. But there are some tips and tricks we will learn along the way.

app.component.spec.ts

Angular provides a way of testing components. This allows us to verify that our components are doing what we expect.

However, the one thing that is most likely to change in your application is the presentation layer. So, even though we can test the View, the question remains, "Should we?"

We will cover testing later on, for now, just know that all the files that end in `.spec.ts` are tests files that are automatically run for us when we use the command `npm test`.

NPM Scripts

Start

```
npm start
```

This starts the application locally in debug mode. That is, if you use Developer Tools to look at the source, it will understand how your TypeScript files relate to the resulting JavaScript files and throws additional warnings to the console.

You can try running this now and navigating to localhost:4200 where you should, eventually, see "Welcome to App!!" along with the previously mentioned links.

If you open Developer Tools and look at the "Source" you should see the resulting JS files under the "localhost:4200" branch. These are the files you are actually running.

Further down, you will see the `webpack://` branch. Within that, you will see several directories. One will be a directory that looks like a directory on your file system that you recognize, the others will be something that references "webpack". We don't care about those.

Open the file system directory. You will see a `src` directory. This will have all the files that you are familiar with. If you open the TS files, you'll see the original source. If you open HTML or CSS files though, you won't see the original templates or CSS. This is because all our HTML and CSS gets turned into JavaScript that gets written into the DOM at runtime. This can be particularly frustrating when you are trying to debug your CSS.

To fix this, we are going to change our start script so that our CSS gets extracted during our debug builds.

Open up package.json and change the start script to look like this:

```
"start": "ng serve --extract-css"
```

Now when you run npm start and look at the source, your CSS for your component is available to you and MUCH easier to adjust in the browser tools so you know what needs to be adjusted in your source code.

Build

```
"build": "ng build"
```

As I mentioned previously, the build script bundles and minifies all the files that you need and puts them in the dist directory. That's all pretty cool, but we can make it do even more.

In previous versions of the CLI, we had to add the –aot optionto force Ahead of Time Compile (AoT). But now, any time we compile for production, --aot is assumed.

What is AoT?

Without an AoT compile, when you build your application, and even when you use the "start" script, there is a compile step that needs to happen to all our templates. This happens on the client side before we see the page.

By compiling with AoT, we can make all that compile process happen at build time. This does two things for us. First, and most obvious, we can skip that step on the client, which makes our code run sooner. But, less obvious and just as important, it

eliminates the need for the compiler code to be downloaded from the server. This gives us an additional performance boost.

But, there is a side effect to using AoT mode. It forces the compiler to be stricter. So, things that were working when you compiled using the "start" script may not even compile when you use the "build" script with the AoT flag turned on.

So, what I recommend is that we use AoT for our "start" script as well.

```
"start": "ng serve --aot"
```

We get the same benefit of running the code using the development server, but we have the added benefit of not having any surprises when we go to build.

Of course, we will still need to be able to debug occassionally, and for that we use the script that was our "start" script and create a new script for that configuration.

```
"startDebug": "ng serve --extract-css
    --aot"
```

The line above should be all on one line. But because of text wrapping, it is displayed on two lines.

You'll notice that I've added –aot to this as well because I've found times where the code just doesn't work without it. Eventually, the plan is to make aot the default for all builds.

Lint

We can verify that we've written solid TypeScript by using the built-in linter. With recent updates, you may need to add the `--type-check` flag to the linting script so that it can validate that

there are no type errors in our code.

```
"lint": "ng lint --type-check --fix"
```

I also add the –fix option so that the linter will fix any problems it finds automatically.

Enforcing Rules

Now, having lint rules in place is no good if they are never enforced. But, there is good reason to make sure these rules are enforced, not least of which is that some of these rules ensure the code will do what we had in mind when we wrote it. But having rules also ensures that the code is easier to maintain.

To solve this, we are going to use a pre-commit hook in GIT to ensure that we never commit files that don't conform to our rules.

To make this work, we are going to install a new package called "pre-commit"

```
npm install --save-dev pre-commit
```

Once it has installed, you can create a pre-commit section that holds an array of script names that you want to run prior to committing your files to GIT. I normally add this section just under the scripts.

So far, we want to run the lint script, and we can assume we will want to run our tests.

```
"pre-commit": [
  "lint",
  "test"
],
```

There is one other linter that is also beneficial, stylelint. Stylelint allows you to set up rules for your CSS so that all your CSS files look the same regardless of who coded them.

Stylelint needs to be installed globally.

```
npm install -g stylelint
```

If you want to make sure this gets installed when you run npm install for your project, you should add this to a `postinstall` script in the scripts section of your `package.json` file.

```
"postinstall": "npm install -g stylelint"
```

You will also need to configure the CSS rules you want to follow. To do this, add a `.stylelintrc` file to the root of your project. My file looks like this:

```
{
  "extends": "stylelint-config-standard",
  "rules": {
    "indentation": 4,
    "no-empty-source": null,
    "selector-type-no-unknown": null,
    "color-hex-case": ["lower", {
      "message": "lowercase letters are
easier to distinguish from numbers"
    }]
  }
}
```

That is, I use the standard configuration, but I make sure indentation uses 4 spaces and all my hex color codes are in lowercase.

Now, to make this work, we need to install the standard rules.

```
npm install --save-dev
    stylelint-config-standard
```

Again, that should all be on one line.

And finally, add `csslint` as a script in your script section and a `csslint` in `pre-commit`.

```
"csslint": "stylelint src/**/*.css --fix",
```

And

```
"pre-commit": [
  "csslint",
  "lint",
  "build"
],
```

Third Party CSS

Often, we will want to install CSS from the outside world. It is common to install BootStrap and FontAwesome for example. If you use components from a third-party vendor, they will often have additional CSS files they will want you to use.

Bootstrap

For the moment, we will install BootStrap and FontAwesome. The process is similar for other CSS files you may need to install.

So, start by installing the two libraries:

```
npm install --save-dev bootstrap@3
  font-awesome
```

Then include a reference to the CSS in the styles section of `.angular-cli.json`

```
"styles": [
"../node_modules/bootstrap/dist/css/bootst
rap.css",
"../node_modules/bootstrap/dist/css/bootst
rap-theme.css",
"../node_modules/font-awesome/css/font-
awesome.css"
],
```

Do NOT include the JS files. The jQuery in the Bootstrap files does not play nice with Angular. We will only be using bootstrap for styling. Not for any of the controls.

Notice! Because we are bundling and minifying the code ourselves, it doesn't matter if you use the *.min.css files or not. The result will be about the same.

Material Design and Others

While we are using Bootstrap 3 here, this is not the only option. We use it here because it is readily available and because most people are already familiar with it. Instead of trying to teach Angular AND some other CSS library, we use Bootstrap. If you have some other theme system you want to use with your code, you are free to do so.

Routing

In the old world where all our pages were on the server and every change on the client side required a full round trip to the server, each page was a unique URL on the server. In the Single Page Application (SPA) world, we only load one "Page" from the server and the client takes care of making it look like we have moved from one page to another.

When done well, we can create pages that reuse existing content on the screen causing a minimal screen refresh while still allowing the user to link to a specific "Page" in our application.

These "Pages" are called "Routes" As in, here is the route to some code I want to execute.

There is a down side that shows up every time someone tries to do this for the first time. You won't see this problem until the first time you try to deploy your code because the development server handles this issue for you.

The problem is this. When a server receives a request from the browser, it tries to find that file on the server. If it doesn't exist, the server returns a 404 error. File not found.

Most servers provide ways of circumventing this issue by providing rules. Essentially, you write a server rule that says, "If the browser asks for a file that doesn't exist, send them back index.html instead." You may need to provide exceptions or otherwise refine the rule if your server is also rerouting other traffic.

Assuming you have that end of things working correctly, here are the steps to get basic routing working in your Angular application.

Define Your Routes

While we could easily define our routes in `app.module.ts`, the code we write will be much easier to maintain if we create a separate module file named `app-routes.module.ts`. So to start, create an `app-routes.module.ts` file right next to your `app.module.ts` file. You can do this with the Angular CLI by typing the following in the command line from within the src/app directory:

```
ng g module app-routes --flat
```

When you create a module with the Angular CLI, it will put it in a subdirectory. By using the `--flat` option we are telling the Angular CLI to put the app-routes module in the current or, in this case, the root of the application.

Open the file, it has some stuff in it that we don't need. Remove the `CommonModule` references and the declarations section of the `@NgModule` decorator.

In this new file, you will create an empty Routes array, called routes and decorate the class with @NgModule

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

```
@NgModule({})
export class AppRoutesModule {}
```

You need to also import `Routes` and while you are doing that, you might as well import `RouterModule` because you are going to need that soon too.

```
import { Routes, RouterModule }
    from '@angular/router';
```

Next, in your `app.modules.ts` file, import `AppRoutesModule` using both the TypeScript import

```
import { AppRoutesModule }
    from './app-routes.module';
```

And as part of the imports section of the @NgModule decorator

```
@NgModule({
  …
  imports: [
    …
    AppRoutesModule
  ],
  …
})
export class AppModule { }
```

We really haven't done anything useful yet, we've just setup some boilerplate code that will compile so we won't have to think about it anymore.

Now, back to the `app-routes.module.ts` file.

Each element in our Routes array defines a specific route in our system relative to the parent route it is a part of. At the top level the parent route would be the root of the application.

Here are the properties that are available to us:

path

The path property allows us to specify what path, or URL, will load this route. If you want the component to load for any path, use `'**'` as the value. If you want the component to load for the root element, use '' for the path and specify `pathMatch: 'full'` as another property. You can also use the value `'**'` to mean, "match anything." We typically use `'**'` to match what would typically be thought of as 404 errors. For this to work correctly, it should be the last element in your top most route definition.

pathMatch

As we've already mentioned, `pathMatch` should be `'full'` to match `''` as the exact path. But you can also give this value `'prefix'` to tell it to match any path that starts with the value. You only need to specify this value if you want to use `'full'`.

It should also be noted that this value only evaluates the part of the path you are in. If you use this in a child path, it won't match the whole path, but only the part that is in the child.

component

Component specifies what component should get loaded when the path is matched.

children

The children property allows us to specify an array of child paths.

Route Components

Since our routes will need components, let's start by creating several components so that we can illustrate routing.

But first, a short word about how we organize our code.

In many demos online, the tendency is to put all our components right under the app directory. But, in larger applications, I've found that it makes a lot more sense to create a route directory under the app directory that we place each of our routes in.

Now, you might think that we would want to place our child routes as child directories under the routes they are a part of, but the problem with this is that we often have child components in our routes. How do we know which directory represents a child route and which represents a child component?

No. What we really want to do is place even the components that represent child routes right under our routes directory.

Say we have a `Page1` route and there is a `SubPage` route that is a child of `Page1`. To make it clear, we put `SubPage` in a directory named `page1.sub-page`.

As for components that are common to multiple pages, we place those in a `shared` directory which is right under the app directory. This keeps our directories well organized and the code neatly organized as well.

So, the next obvious thing that we need to do is that we need to create a `routes` directory. Do that now.

Next, at the command line, inside the new routes directory, execute the following Angular CLI command

```
ng g component page1
ng g component page2
ng g component page1-subpage
```

As you executed each command, it should have created a directory for each component with the corresponding css, html, ts and spec files. Then it updated the app.module.ts file for you so that the components are available for use in the system.

You may also notice that we created the component as `page1-subpage` instead of `page1.subpage`. The reason for this is that the CLI doesn't like period separation of file names. So, the next thing we are going to do is that we will change the directory name to `page1.subpage`. You will also need to change the TypeScript import line that references this directory in your `app.module.ts` file.

app-routes.module

So, now that we have components to page to, let's create our route definitions. Back to the `Routes` array in our `app-routes.module.ts` file.

```
export const routes: Routes = [
  {
    path: 'page1',
    children: [
      {
```

```
        path: '',
        pathMatch: 'full',
        component: 'Page1Component'
      },
      {
        path: 'subpage',
        component: Page1SubpageComponent
      }
    ]
  }, {
    path: 'page2',
    component: Page2Component
  }
];
```

The first definition may look a bit odd. We are setting up a route to page1, but the route is the children. Then in the children we define a route to `''`. This is where the Page1Component is specified as the component we want to load.

You will also note that we specified `pathMatch: 'full'` for the `Page1Component`. This is because we only want this component to be loaded when the child path is empty.

Using this definition, everything loads into the top-level router-outlet. If we placed `Page1Component` at the same level as we defined the `page1` path, then Angular would expect to have a router-outlet in `Page1Component` where `Page1SubpageComponent` would be loaded.

You need to be careful how you define your routes.

Next, you will need to import the three components using the TypeScript import statement.

```
import {Page1Component} from
  './routes/page1/page1.component';
import {Page2Component} from
  './routes/page2/page2.component';
import {Page1SubpageComponent}
  from './routes/page1.subpage/page1-
subpage.component';
```

Now that everything is defined, we just need to tell Angular where we want these components to show up. For right now, open the app.component.html file and remove everything that is there and add the router-outlet component.

```
<router-outlet></router-outlet>
```

Now, `router-outlet` is a component that is defined in the `RouterModule`, so we need to import that in the imports section of our `AppRoutesModule`. But we don't just import the `RouterModule`, we use `RouterModule.forRoot()` and pass in the route array we just defined into `forRoot()`.

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutesModule {
}
```

There is one final tweak that we need to make to our route definition. Right now, if you go to the root of the application, there isn't a component defined for it. To fix this, we are going to add the following definition at the top of our routes:

```
{
  path: '',
  redirectTo: 'page1',
  pathMatch: 'full'
},
```

You need pathMatch: 'full' to tell the router to only match this rule when the path is `''` exactly, just like we did with the Page1Component in the children list. Otherwise, it will match everything.

The `redirectTo: 'page1'` part tells it to redirect to the page1 path when this rule is true.

Lazy-Loading

I realize that I still need to show you how to navigate to routes in your application, but first let's look at Lazy-Loading the routes.

In the past, when building a Single Page Application, the custom was to load all the JavaScript code we needed for the page up front. If you have multiple pages in your application, some of those may never be needed by your user. What are we doing loading stuff that will never get used?

Instead, it is better to load only what we need when we need it. While it might take longer if you totaled up each load, the user perceives the experience as faster. Therefore, what we want to do is to make each of our pages load as we need them.

To do this, we need to create a module for each of the components that represent our top-level routes. Remember I said we want to import modules and services into the module that is closest to where we need them? By only importing things where

we need them, Angular can create the smallest package necessary all up and down the dependency tree.

To make things easy and to do them the way you would have done them if you had done it this way to begin with, let's delete all the subdirectories under routes. It's OK. We aren't losing any work that we can't quickly get back. We haven't added any code to these routes yet.

At the command line, navigate to the routes directory and then type in the following Angular-CLI commands:

```
ng g module page1
ng g component page1
ng g module page1-subpage
ng g component page1-subpage
ng g module page2
ng g component page2
```

And then, just like we did the first time, rename the `page1-subpage` directory to `page1.subpage`.

Next, go to the `app.modules.ts` file and remove the references to the `Page1`, `Page2`, and `Page1Subpage` components anywhere you see them.

Do the same thing in the `app-routes.module.ts` file.

Now, the way we define our routes changes slightly. We will still need the redirect route, but everything else changes.

The key to making this work is the property `loadChildren`, which is a string in the format of:

```
'pathToModule#ModuleClassName'
```

We'll work from the top down. Change the routes array in `app.module.ts` to look like this:

```
export const routes: Routes = [
  {
    path: '',
    redirectTo: 'page1',
    pathMatch: 'full'
  },
  {
    path: 'page1',
    loadChildren:
'./routes/page1/page1.module#Page1Module'
  }, {
    path: 'page2',
    loadChildren:
'./routes/page2/page2.module#Page2Module'
  }
]
```

When we try to access something from page1, it will load the `Page1Module` and try to resolve it from there. When we try to access something from page2, it will load the `Page2Module`. Both happen during runtime.

Next, go to `page1.module.ts` and, import `RouterModule` and add the following to the imports section of `@NgModule`

```
RouterModule.forChild([
  {
    path: '',
    pathMatch: 'full',
    component: Page1Component
```

```
  }, {
    path: 'subpage',
    loadChildren: '../page1.subpage/page1-
subpage.module#Page1SubpageModule'
  }
]),
```

Next move over to page2 and do something similar. Since page2 doesn't have a sub-route, you only need one route.

```
RouterModule.forChild([
  {
    path: '',
    pathMatch: 'full',
    component: Page2Component
  }
]),
```

And again, similarly for page1-subpage

```
RouterModule.forChild([
  {
    path: '',
    pathMatch: 'full',
    component: Page1SubpageComponent
  }
]),
```

If you haven't already, move your command-line prompt back to the root of the project and type

```
npm start
```

To start the server and compile your code. If everything compiles, you should see 3 chunk files along with the other files

we saw when we compiled the code without Lazy-Loading. One each for each of the routes.

Run the app now to make sure it works correctly.

See how easy that was? It isn't that much harder than specifying the routes like we did the first time, but we get huge benefits in perceived performance.

Passing Parameters

The last thing you need to know about routing is how to pass parameters. You would normally do this when you are coming from an existing list of items. Each item has some sort of unique identifier. We click some link and that takes us to another page to show details or to edit the content. For our purposes here, it doesn't matter.

To specify that a route takes a parameter, use colon notation:

```
path: 'detail/:id'
```

Angular knows it is a parameter when you use a URL to get to it because of the location.

Retrieving Parameters

Let's say you have a component that represents a route with a parameter. For that to be useful, you'll need to pull the parameter out of the route information.

To do this, you'll need to inject `ActivatedRoute` into the constructor of the component. Then when you need the parameter(s) you can use:

```
export class SomeComponent {
  constructor(route: ActivatedRoute) {
    route.params.take(1).subscribe(params
=>
      // Use params['id'], where 'id'
      // is the name we gave the
      // parameter in the path.
    );
  }
}
```

Route Navigation

Now that we have routes in place, we need to discuss how to navigate from one route to another. The temptation, having just used URLs to go from one to the other, would be to use hyperlinks and put the information in the `href` attribute.

No doubt, you could probably get that to work, but the main problem with using that method is that there are no safeguards to make sure that the URL you use to navigate when you are developing will work when you move the site to another environment.

The reason for this is that we must set the base `href` for the site. During development this is normally `'/'`. But when you go to production, it will probably be some sub directory.

Also, because of this base `href`, every page/route we land on is still relative to that base. This means that every route we want to navigate to would have to be hard wired to the base of the site, and again, that's assuming the site will always be in the same relative location when it is deployed.

Now, if we can't use a regular URL to navigate, what do we use instead?

routerLink

You use the `routeLink` directive added to your anchor tag.

```
<a [routerLink]="['/page1']">go here</a>
```

This may look a little different from what you expected, so let's break this down.

The `routeLink` directive takes an array. Since we can't pass an array as a string, the only way we can pass it is by evaluating it at runtime. Remember, the square bracket syntax is an indication to the Angular compiler that what we are assigning is something that should be evaluated. Typically, this would be pointing to a function or variable in our TypeScript code. In this case, we are pointing to a literal array. Everything between the opening and closing quotes is JavaScript.

As for the actual parameter, the string in the single element array works much like you would use a URL. The forward slash says to start at the root of the web application (instead of the root of the domain like a URL would.) And the page1 is the route we've already defined. If you leave the forward slash off, it is relative to the current route.

But what about passing parameters?

```
<a [routerLink]=
"['/page1',someVariable]">
  go here</a>
```

Each comma delimited value represents a segment of your route.

Router.navigate()

The other way you might want to cause navigation to a page to occur is by using the `navigate()` method hanging off the `Router` class. Using dependency injection, you inject the Router into the class that needs to use it and then use that instance to call `navigate()`. The parameter you pass in looks very similar to what you used for `routerLink`.

```
router.navigate('/page1',someVariable);
```

Both `routerLink` and `Router.navigate()` both support URL like references using `'./path'` or `'../path'`.

Optional Parameters

There will come a day when you want to be able to pass parameters some of the time, but not all the time. Angular calls these "Optional Parameters."

You can pass these in an anchor tag or in the navigate() method using the queryParams property.

```
<a [routerLink]= "['/page1',someVariable]"
   [queryParams]="{param1: value1,
      param2: value2}">
  go here</a>
```

Or

```
router.navigate('/page1',{queryParams:
   {param1: value1, param2: value2}});
```

To retrieve them, subscribe to the `queryParams` array hanging off the `ActivateRoute` object like we subscribe to the `params` array for fixed parameters.

Where Am I Now?

The last part of routing you will commonly need to know about is detecting what the current route is. This will require you to inject the Router object into the component that needs the information. Once you have the router object, you can use code like this:

```
constructor(router: Router) {
  router.routerState.snapshot.url
}
```

This will get the current route `url`. If I need this as soon as I've navigated to a route, I grab this as part of listening for the router's `NavigationEnd`.

```
router.events
  .filter(arg =>
      arg instanceof NavigationEnd)
  .subscribe(arg => {
    this.selectedTab = router
      .routerState
      .snapshot.url.split('/')[1];
    return this.selectedTab;
});
```

Guards

Guards control access to our routes. What happens if you have a route that only certain people should have access to. Like an admin page. Sure, you could leave the link off so no one can click the link to get to the page, but that doesn't prevent someone from pasting the link to the forbidden page into the address bar of the browser and getting there anyhow.

In Angular, we have four kinds of guards and two ways of creating them.

The four types of guards are:

- CanActivate
- CanActivateChild
- CanDeactivate
- CanLoad

If you follow my advice and always lazy load your routes, then the two you will most often use are `CanLoad` and `CanDeactivate`. `CanLoad` provides rules for Lazy-Loading a module. `CanDeactive` provides rules for leaving a route.

If you decide to bundle routes together, then you may also need `CanActivate` and `CanActivateChild`. `CanActivate` is exactly what it sounds like. Can I activate this route? `CanActivateChild` would go on a route definition that has a children's collection. This rule determines if I can activate the children.

To use Guards in our application, the first thing we need to do is to define them. The easiest way to define them is as a function that returns a `boolean` value, a `boolean` Observable, or a `boolean` Promise. For our purposes here, we will just return a `boolean` value. But when you have some asynchronous call you need to make to determine if we should return true or false, you'll want to return an Observable or a Promise. I favor Observables.

The definition for a Guard rule looks like this:

```
@NgModule({
  …
  providers: [{
    provide: 'ruleNameHere',
    useValue: () => {
      return true;
    }
  }
])
```

Then, to use the rule, you assign the appropriate rule the name of the rule.

```
{
  path: 'page1',
  canLoad: ['ruleNameHere'],
  loadChildren:

'./routes/page1/page1.module#Page1Module'
}
```

Notice that canLoad, as well as the other guard properties, takes an array. This allows you to apply multiple rules to a route.

The other way of defining a guard is as a class that implements an `interface`, or multiple interfaces that include `CanActivate`, `CanActivateChild`, `CanDeactivate`, and `CanLoad`. You then implement the corresponding functions in your class.

Now, to include the rule you use the Class rather than a string:

```
{
  path: 'page1',
  canLoad: [RuleClassHere],
```

```
  loadChildren:

'./routes/page1/page1.module#Page1Module'
}
```

Multiple Outlets

The last thing about routing that is helpful to know about is that you are not limited to one and only one router-outlet on the page. You can place additional router-outlets if you give the additional router-outlets a unique name attribute.

Then, to tell your code to put something in that attribute, you use the name attribute. This is like named containers which we will look at later.

```
<router-outlet name="name1">
</router-outlet>
```

Now, how do we get a component to show up there?

Remember our `routes` array? You can register components with routes by adding an outlet property to your route item that matches the name you gave the outlet:

```
{
  path: 'somePath',
  component: SomeComponent,
  outlet: 'name1'
}
```

And here is where things get just a little funny. At least they seem that way to me.

You see, while it may seem like that would be enough to get the component to show up, it isn't. You need to specify it when you link to it.

```
<a [routerLink]="['/somePath', {outlets: {'name1': ['somePath']}}]">linkme</a>
```

If you have multiple named outlets on the page, you can specify each of them as elements of the outlets object that you pass to `routeLink`.

Now, let's think about what we've done here. We've told Angular that when we specify `somePath` as the route, we want the outlet named `name1` to use the component `SomeComponent`. This means that we can create another path that uses the same named `router-outlet` but is associated with an entirely different path.

Components, Modules, and Services

Components, Modules and Services are the key components of any Angular application. A clear understanding of why they exist, what they are, and how they relate to each other is critical to being able to write clear, maintainable, Angular code.

Unfortunately, because they are so inter-related, covering each of these in an order that doesn't mention the other is nearly impossible. This next section is full of open loops. Rest assured, these open loops will be closed by the time we finish this section.

Components

There are two main types of components in Angular. Smart components and dumb components. No! Really! That's what they are called.

This is a unique architectural distinction that will become critically important to how you develop your applications.

If you think about how you may have written client-side code in the past. You had components that you considered application components that you would let do anything any other application component code do, including calling services to get data. Then you had components that you wanted to be able to reuse throughout your code that you made sure only rendered the data they were given and fired events out to tell the component's

container that something important happened in the component. Maybe data changed, or a button was clicked. The important thing is, the component has no idea what the application is doing.

Continuing to think back on this way of programming, you probably also ran into a situation where you've created a component that knew about the application that you wanted to be able to use in multiple places within your application. But, you didn't figure this out until after you had tied it to the application's functionality. Too bad. Now you need to do a bit of refactoring just so you can reuse that component.

Now, what if you made all your components application agnostic?

In that case, every component would serve two primary functions.

First, they would display whatever data they were given. They would not retrieve their own data. Nor would they be given a pointer to how to retrieve the data. You must always be thinking, "if the framework I am using changes, or a better framework comes along and I want to change, how might the code I am writing need to change to support it?"

Second, the component should fire events out when something important happens so that the containing component can respond appropriately.

This isn't to say that the component itself will not do some processing. Particularly in the case of dumb components, if the

rules you are coding are component specific, you can have rules. In fact, you should.

Now, you might wonder how an application is going to achieve its purpose if the components never know anything about the application. This is where smart components come in.

A smart component is a top-level component that knows how to retrieve the data it needs from where the data is stored and it knows how to get the data back to where it is stored.

But, once again, this is all that it is responsible for. No real logic should appear in your smart component. In fact, if you've written your smart component correctly, all your methods should have a cyclomatic complexity of less than two. This means you'll end up with code that doesn't need to be unit tested.

There is one other super critical reason why you'll want to make sure none of your application logic shows up in any of your components. You can change how you present information to the user without worrying too much with how that will impact the way the application works.

For example, let's say you start out writing a web application. Then one day, you decide this would work well as a native phone or tablet app. There are ways you could do this where everything remained as JavaScript or HTML, but if you want to use native controls, you'll need to use some other presentation framework. Ionic and Native Script are two such frameworks that work with Angular. But neither of them use HTML. If you've stuck all your application logic in your components, you'll have to rewrite your logic as well as the presentation. Not to mention that you'll have to retest everything and possibly fix

bugs in multiple places when you are maintaining the application later.

So, you may be asking yourself, “If I don’t put my application logic in my controls, where do I put it?”

Services

You put your logic in non-view specific classes. In the case of Angular, this is going to be a class decorated with @Injectable() and because it has been decorated with @Injectable(), you are going to be able to use Angular’s Dependency Injection container to make an instance of that class available to other classes in your application, including your smart components.

```
import {Injectable} from '@angular/core';

@Injectable()
export class SomeService {
    private accumulator = 0;

    add(x: number): number {
        return x + this.accumulator;
    }
}
```

The question you should be asking now is, “How do I inject the object into my other code?”

As it turns out, this is easy. In your class constructor, you provide parameters for the various objects you want to inject. For the Injectable we created above, you would inject that into another class using

```
constructor(private srv: SomeService) { }
```

You don't have to prefix the injected variable name with private, protected, or public. You only do that to make it a member variable. If you are only going to use the injected object in the constructor, you can leave those modifiers out.

There are service that come with Angular, like the `Http`, or `HttpClient` services that allows you to make HTTP calls to your server, but the bulk of the services you will use will be ones that you've created.

If you are coming from the old AngularJS world, you are probably already familiar with the multitude of "Service" types. It was quite confusing. In Angular, we just have Services. However, there is one small improvement that Angular makes to how Services behave.

In AngularJS, Services and their variants were essentially Singletons. Once the Dependency Injection container created a service for you, that Service instance was used anywhere you injected the service.

This is still true in the Angular world if you "provide" the Service in a Module. If, instead of providing the service in a module, you decide to "provide" the Service in a Component, the instance of the Module will be local to the component and any of the component's children.

Which leads to the obvious question, "what is a module?"

Modules

When talking to programmers who are familiar with .NET or Java, I typically refer to Modules as "kind of like a namespace."

But, they are not much like a namespace at all. That's just a way of bridging a cognitive gap between something we are familiar with and something we are not.

We will start with the namespace concept and then branch out from there.

Just like with a namespace, modules are a way of grouping code. When you load a module into your code, you are saying, "I want to make everything that module provides, declares, exports, and imports available to my application."

Unlike a namespace it isn't just about where your code lives. It is more about what code gets loaded and is available for use. This is one of the reasons why simply importing a module with the TypeScript import statement isn't sufficient for telling your application that the module is available. Besides, Angular modules aren't a TypeScript thing. So, a module is simply a way of telling Angular that Services and Components are available for use in our application.

Now that we have this background, we can move back to some code.

Continuing on from our Service example above, if we want to make that service available to our code globally, we register the Service in our Module in the providers section of our @NgModule() decorator.

```
import { SomeService }
    from './some.service';
import { NgModule } from '@angular/core';

@NgModule({
  ...
```

```
  providers: [SomeService],
  ...
})
export class AppModule { }
```

I've used the AppModule class here for simplicity. In practice, you should create modules for groups of code in your application. You'll see why as we move along.

The next thing we'll want to make available to our module are components. The thing about components is, we have components we want to make available to components in our module and we have components we want to make available to other modules that might import us in the future.

This might sound odd, but it is a good thing. For simple components, you might want the component to be available in both situations. But in more complex components, you might have sub-components that only exist to support the components you want to make available to the world. You don't want to expose those for multiple reasons. One of those is, the more components your expose, the more you must document and the more lock-in you create for yourself in terms of API. Even if it is a closed system that you have some sort of control over, these are issues you'll want to consider.

To make a component available to just the components that are a part of the module, you specify them in the declarations section. If you open the `AppModule` we created using the CLI, you'll see that `AppComponent` is declared in the declarations section.

You'll also see that the `AppComponent` has a selector of `app-root` and that `app-root` appears in the index.html file. The only reason that Angular knows to put the html template from

`app.component.html` where `app-root` shows up in `index.html` is because we specified `AppComponent` in our declarations section.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  …
})
export class AppModule { }
```

Now, in the case of the `AppModule`, anything we declare is also publicly available because the `AppModule` is the top level of our module hierarchy. But if you were to import another module into this module, you'll want to expose other components so they can be used throughout your application. You do that by placing them in the exports section.

```
...
exports: [SomeComponent],
...
```

Dependency Injection

Before we get too carried away with how to use Angular's Dependency Injection system, we need to spend a little time focusing on what Dependency Injection is and why we would want to use it.

In an object-oriented world, a lot of the objects we create in our system are dependent on other objects. Even something as

trivial as using a Date sets up a dependency on an external object.

The chief problem with coding with dependencies is that it makes any given method we create that is dependent on some other object incredibly difficult to test. This alone should be a strong enough reason to use dependency injection. But, I'm not naive enough to believe that everyone who reads this will be implementing automated testing in their applications. So, why else might we want to use dependency injection?

Probably the next greatest reason would be that it allows us to swap out functionality easily. A couple of places where this might be of benefit are with the global error handling system that global errors that occur while Angular is processing our code and the `Http` or `HttpClient` services.

In the case of error handling. Angular has a service already defined called `ErrorHandler`. It doesn't do a whole lot. But, we can create our own `ErrorHandler` class that does exactly what we want. All we need to do is create a class that looks just like the `ErrorHandler` class and then, when we "provide" it in our module, we use a slightly different syntax.

```
export class MyErrorHandler
  implements ErrorHandler {
  handleError(error) {
    // do something with the exception
  }
}


@NgModule({
```

```
  providers: [
    {
      provide: ErrorHandler,
      useClass: MyErrorHandler
    }]
})
class MyModule {}
```

The providers section is saying, “Any place we said we want to use the `ErrorHandler` class in our code, use the `MyErrorHandler` code instead.”

Now, what is particularly great about this is that this will now be used not just for code I write, but for ANY Angular code that gets executed after this “providers” section has been executed.

Similarly, as you’ll find out later, using the `Http` service, or the `HttpClient` service in Angular 4.3 or later, we tend to write a lot of boilerplate code repeatedly. If you encapsulate all that code in your own implementation of `Http` or `HttpClient`, you can use it as a drop-in replacement for `Http`. In this case, since you are the only one using the code, you may just want to specify the new class as its own service, but you have the option of making it replace `Http` if you want to or there is code you are running that you can’t replace the `Http` service on.

In the same way, maybe you have code you’ve written that you want to refactor. You want to continue to use the old code until you’ve got the new code working. By setting up a `useClass` replacement on the old code, you can switch between the old code and the new code simply by changing that one line.

There is another option you might consider using as well. In some cases, particularly with testing, it may make more sense to

use the `useValue` property in place of the `useClass`. If you do this, you are saying, "I've already created the object you should use and here it is." You might use during testing so that you can set the state of the object before it gets used in the code that you are testing.

That's dependency injection and some different ways it might be useful to you. However, to use Angular you really don't have much of an option. Oh sure, you might be able to create some trivial object using the "new" keyword. But for the most part, any service you might decide to use probably has other services it uses. Just try `new`ing up an `Http` service sometime. You'll see what I mean!

As we've already seen, we can either make objects available globally by specifying them in the providers section of a module, or we can localize them to a component and the component's children by specifying them in the providers section of our component.

To use our object, we simply specify the typed parameter in in our constructor, as we've already seen.

Now a word here about the temptation to bypass dependency injection.

As I've already mentioned, you can't really use a lot of the features of Angular without using dependency injection. But you might be tempted to create a trivial class and instantiate it without using dependency injection.

This may, or may not, be appropriate. What you need to consider before you make this choice is why dependency injection exist, which we've already covered, and how what you

are about to do is going to impact your ability to leverage the advantages of dependency injection in the future.

In general, you should error on the side of using dependency injection.

Observables

If you've been programming JavaScript based applications for any length of time, you've probably already made the progression from callback hell to promises, but just to recap. Whenever we make any kind of asynchronous call in JavaScript, we need to provide a callback function to the call so that, when the call completes, the function can be called with any resulting data. Function calls you may typically make that need this kind of feature are `setTimeout()`, `setInterval()`, and AJAX calls using the various libraries that support this.

The problem with using callbacks is that you can end up with "Callback Hell" where you have callbacks inside of other callbacks. Our code becomes messy and difficult to reason about.

To try to flatten this situation out, promises were created. Instead of creating a callback function and passing it into the asynchronous function, the asynchronous function returns a promise that has a function we can pass our function into. This function can return yet another promise. The result is that instead of having nested callbacks, all our callbacks live at the same level.

However, in the process, we lost the ability to cancel an asynchronous function using callbacks. Most of the time, this was not a huge concern, but in the case of AJAX calls, we did end up making more request than we really needed to. Most people never even recognized this as an issue. But if you go and look at some of your older code, you will see that you have several places where the code would work more efficiently if

you were able to cancel a call that was being superseded by a new call.

Meanwhile, some additional functions were added to JavaScript Arrays. Maybe you've seen some of them? `map()`, `reduce()`, and `filter()` are three of the most used functions.

What? You haven't seen these? If you have and you know how they work, you can skip this next section. But, if you haven't, pay careful attention because this next section is critical to understanding how Observables work.

Array Functions

map()

Let's say you have a list of objects that you need to transform into another form. In the example below, we want to transform our list of objects into a list that can be used in a dropdown list using `fullName` for the display and id for the value.

Without using map(), your code might look something like this:

```
var someList = [
  {
    id: 1,
    firstName: 'Dave',
    lastName: 'Bush'},{
    id: 2,
    firstName: 'John',
    lastName: 'Doe'},
  …
```

```
];
var i;
var newArray = [];
for(i = 0; i < someList.length; i++) {
  var item = someList[i];
  newArray.push({fullName: item.firstName
+
    ' ' + item.lastName, id: item.id});
}
```

The thing is, we do most of the code from `var newArray = []` on down throughout out code and between projects. It is only the code in the `push()` that changes.

What if we were to make the code into a function? That's what `map()` does. It takes a function as a parameter that takes an item as a parameter. Inside the function, we use that item to specify how we want to transform the item and the whole `map()` function returns the new array.

The code above, turns into this.

```
var someList = [{
  id: 1,
  firstName: 'Dave',
  lastName: 'Bush'
},{
  id: 2,
  firstName: 'John',
  lastName: 'Doe'
},
  …
];
var newArray =
```

```
someList.map(function(item){
  return {fullName: item.firstName + ' ' +
      item.lastName, id: item.id};
});
```

filter()

Now, let's say that for some reason, you only want to include items in the new array that include a last name that starts with 'B'.

Our old style code would look something like this:

```
var someList = [{
  id: 1,
  firstName: 'Dave',
  lastName: 'Bush'
},{
  id: 2,
  firstName: 'John',
  lastName: 'Doe'
},
  …
];
var i;
var newArray = [];
for(i = 0; i < someList.length; i++) {
  var item = someList[i];
  if(item.lastName.startsWith('B') {
      newArray.push(item);
```

```
    }
}
```

And once again, this is code we tend to write quite a bit. So, what if we had a function that did this for us?

This is exactly what the filter function is for. So, rewriting the code above using filter() would look like this.

```
var someList = [{
  id: 1,
  firstName: 'Dave',
  lastName: 'Bush'
},{
  id: 2,
  firstName: 'John',
  lastName: 'Doe'},
  …
];
newArray = someList.filter(function(item)
{
  return item.lastName.startsWith('B');
};
```

Once again, you can see that we passed in a function that takes the current item as a parameter. The function returns true or false. If it returns true, the item gets included in the new array.

Chaining

What if we want to filter AND transform the data?

The temptation for programmers new to this model is to use the map and push the item into an array that was declared outside of the map.

```
var someList = [{
  id: 1,
  firstName: 'Dave',
  lastName: 'Bush'
}, {
  id: 2,
  firstName: 'John',
  lastName: 'Doe'
},
  …
];
var newArray = [];
someList.map(function(item) {
  if(item.lastName.startsWith('B')) {
      newArray.push({fullName:
          item.firstName + ' ' +
          item.lastName, id: item.id});
  }
});
```

But that really isn't all that much better than if we were just using a for/next loop like we've been doing. Old habits die hard.

Instead, we can take advantage of function chaining. What this allows us to do is to filter and then map.

```
var someList = [{
  id: 1,
  firstName: 'Dave',
  lastName: 'Bush'
}, {id: 2,
  firstName: 'John',
  lastName: 'Doe'},
```

```
  …
];
newArray = someList.filter(function(item)
{
  return item.lastName.startsWith('B');
}.map(function(item) {
  return {fullName: item.firstName + ' ' +
      item.lastName, id: item.id};
});
```

So much cleaner.

reduce()

The final useful function we have available to us for dealing with common array loops is `reduce()`. `reduce()` allows to loop through an array and accumulate the items in an array into another array, an object, or a value.

The `reduce()` function take two parameters. The first parameter is a function. The second parameter is the starting value for the accumulator.

The function that we pass in takes three parameters. The current value of the accumulator, the current item, and the current item index. Most people only use the first two parameters in their function. The function returns the new accumulator value that then gets passed into the next call to the function.

So, a simple example would be, given an array of numbers, add them all up.

```
var numbers = [1, 2, 3];
var total = numbers.reduce(
    function(sum, item) {
```

```
        return sum + item;
}, 0);
```

Events as Arrays

Now, imagine that events that fire are part of one long continuous array. An array that never ends. If this were true, listening to events would be as familiar as processing an array.

This is all an observable is. It treats everything as though it were an array, adds several other functions that give us even greater functionality, and several functions that allow us to deal with the fact that events are not only sequential, but also time based.

And because events aren't really arrays, we call this series of items a "stream." So, when you read about "streams" while working with Observables think, "list of items."

Button Click

For example, let's say you have a button on your screen and you want to know when it is pressed. Let's say your button is represented by a member variable name "myButton". In your code, you would listen to a button click by writing code that looks something like:

```
this.myButton.subscribe((event) =>
    /* do something in response
    to the click here */
);
```

You will notice that we used the `subscribe()` function instead of `map()`. We still have a `map()` function. But, `subscribe()` is how we tell the application, "we want to start

listening to the stream now." Otherwise, `subscribe()` works just like `map()` does.

Yes, I know what you're thinking. "How is that better than just having the template call an event handler?"

Well, the fact of the matter is, it really isn't all that much better. But, here is where it does make more sense.

Debouncing Keystrokes

If you've been writing application in JavaScript for a while, I'm sure you've written classic debounce handlers. You know. "Don't fire this event until you are no longer receiving change events from the input field."

I won't write out the old code here. It is relatively long, hard to follow, and therefore somewhat complicated.

But here is how we handle it using Observables.

```
this.myInput
  .debounceTime(250)
  .subscribe((event) =>
    /* do something in response to
       the input field change here */
  );
```

debounceTime(250) tells the Observable to wait for 250 milliseconds to see if there is some other event that comes in and use that event instead. That is much easier than the old way.

AJAX

While you could handle button clicks and debounce logic using old school JavaScript tricks, in Angular, it is practically

impossible to make an AJAX call without using Observables. This is because the `Http` service and the `HttpClient` service use Observables instead of callbacks or promises to manage dealing with the data that eventually is returned from the AJAX call.

Since `Http` and `HttpClient` are similar, we will continue our discussion of handling AJAX calls using `HttpClient`. The main advantage to using `HttpClient` is that it handles parsing the response into a JavaScript object we can use. `Http` just returns the raw Response object and parsing it out is up to us.

`HttpClient`, on the other hand, returns the object we would have parsed out with `Http`.

So, assuming you've injected `HttpClient` into the class that is going to use it, a typical get might looks something like this:

```
this.httpClient
  .get<TypeInfo>('/api/get-data')
  .subscribe((x: TypeInfo) =>
    /* do something with the data */
  );
```

So, walking through this, you may notice that some things look very similar to Promises and then, there are some other things that aren't so much. But trust me this gets much better. We are just starting out small.

First, what is that `TypeInfo` thing?

You see, our get call is what is normally referred to as a "templated method." In simple terms, get doesn't know what type it returns until you tell it. So, we are telling it that it returns a TypeInfo type. TypeInfo is just a name I made up. You would

create an interface that is relevant to the type of information that your AJAX code is returning.

Other than that, we subscribe to the observable that `get()` returns and process the data.

But what if our get call fails?

Oh, we have a method for that.

First, we can trap failures with a `catch()` call.

```
this.httpClient
  .get<TypeInfo>('/api/get-data')
  .catch(err: Error =>
    /* do something with the error */
  )
  .subscribe((x: TypeInfo) =>
    /* do something with the data */
  );
```

But, maybe you want to retry the failed call before you give up.

```
this.httpClient
  .get<TypeInfo>('/api/get-data')
  .retry(3)
  .catch(err: Error =>
    /* do something with the error */
  )
  .subscribe((x: TypeInfo) =>
    /* do something with the data */
  );
```

And because we have a `catch()`, with must have a finally too, right?

Try doing all that with a Promise or a Callback.

Oh, and did I mention you can cancel AJAX calls using Observables? Yep. It's true. In fact, my experience has been that if you make the same call from the same service two times in a row, it will cancel the first call before it makes the second.

The final thing that tends to trip people up who are learning about Observables is that nothing in the observable chain executes until you subscribe to the observable and an event happens.

Once you start getting comfortable with all the methods you have available to you, you'll begin to see the power of using Observables over using Promises or Callbacks, even if there are similarities.

Template Syntax

Before we continue too much further, we need to discuss template syntax. For the most part, templates are simply HTML files with some syntax sugar that lets our template talk to our Component class. In fact, when everything is compiled, everything ends up being essentially the same class, although the template is a child class of the class in our TypeScript file. At least for now.

Interpolation

If you look back at the application we created using the Angular CLI, you'll notice that the app.component.html file has content that looks like this:

```
<h1>
  {{title}}
</h1>
```

That {{}} around title is the "Interpolation Operator". It tells Angular to display the contents of the public member variable, "title" at that location. What you need to know about this operator is that it is how we can flip our templates into "JavaScript mode" where we can put any legal JavaScript expression. You will end up using this operator quite a bit.

Property Assignment

The next most common thing you will need to do is to assign values to properties in other components. This may be components you write, which we'll talk about later, or

components you use from some third party, or even just plain HTML components.

Properties are different from attributes, although there is some overlap. For our purposes here, we will assume we have an attribute named bar in an element named foo. We want to assign the value of 'xyz' to our bar property. This is how you would do that.

```
<foo bar="xyz"></foo>
```

Pretty straight forward. However, what if the value we want to assign is a variable in our component class? Let's say we assigned "xyz" to a variable "abc". To make this work, we need to put brackets around bar.

```
<foo [bar]="abc"></foo>
```

Now, just like our Interpolation example, you can use any expression to the right of a bracketed property. So, this would be perfectly legal:

```
<foo [bar]="'xyz'"></foo>
```

Which would assign "xyz" to "bar".

And this is also valid:

```
<foo [bar]="'xyz' + abc"></foo>
```

Which would concatenate whatever was assigned to abc to "xyz" and assign the whole concatenated value to "bar".

I've seen examples from commercial component vendors who don't seem to understand how this works. They put brackets around everything. Hopefully, this has cleared it up so you don't make the same mistake.

This whole idea of interpolation is commonly referred to as "data binding" because we are binding our template to data in our class.

Events

Next in our list of special template syntax is how we handle events. Typically, in standard JavaScript, we would hook up an event in our HTML to a method using `onEventName`. For example, the click event would be wired up using `onClick="method()"`.

In Angular, events are represented by parenthesis.

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

If your event handler is expecting the event object, you can pass it like this:

```
<button (click)="clickMethod($event)">
```

It is important that you refer to `$event`, or it won't do what you expect.

One common mistake I tend to see is that if your method takes a parameter and you never send it from the template, you'll get a compiler error.

Another common error is forgetting the parenthesis when the method doesn't take a parameter.

Bananas in a Box

In Angular.JS, we had a concept of "Two Way Data Binding". Any time the form changed data it pushed that data down into

the class that was supporting the form. And, any time the data in the class changed, the form updated.

In Angular, we have One Way Data Binding that looks like Two Way Data Binding. Let me explain.

Any time we want to both update the form when the data changes in the class and update the data in the class when the form changes, we use Bananas in a Box syntax. This would most commonly be used with the ngModel property.

```
<input type="text"
  [(ngModel)]="classProperty" >
```

I don't want to get too deep into ngModel right now other than to say that this is how we get at the value property of the component we are using. You can use this with any property that allows for setting and retrieving a field or property in your class.

Under the hood, what is really going on is that an event handler is being created by the compiler and it is the event handler that is setting the value. From our perspective, it is two-way data binding. We will dive deeper into this topic when we learn how to create our own custom components.

ng-template

`ng-template` works like any other DOM element, except you use it when you want to use some feature that requires a DOM element, but you don't really want another DOM element to display. You'll see it in use soon. I just wanted to make sure we introduced it so it wasn't a complete surprise to you.

Here is one of many use cases. In Angular, you can't use `*ngFor` (a loop) in the same element as an `*ngIf`. I've had cases where I wanted one or another block of code to show up with loops inside each. The only way to make that happen intelligently is to use `ng-template` for the IF part of the code while putting regular HTML elements inside.

Typical usage looks something like this:

```
<ng-template
   ... appropriate directives here ...>
    ... template html here ...
</ng-template>
```

Template Variables

There are times when we want to create a variable in our templates so we can reference components in our template from other components in our template as well as being able to name components so we can reference them from within our TypeScript class file. In Angular, we do this by prefixing the variable name with a pound sign. This syntax will show up frequently as we explore Angular.

```
<input #firstName type="text">
<button
  (click)="clickHandler(firstName.value)">
  Click Me</button>
```

You'll see we created a client side variable named "firstName" which later gets used in our `clickHandler()` to pass the value of the field.

ngIf

`*ngIf` can be added to any DOM element to control when it shows up in the DOM. Unlike the old days when we controlled if a DOM element was visible or not using CSS, in Angular, you just don't put it into the DOM if you don't want it there.

A simple use case would look like this:

```
<div *ngIf="foo === 'xyz'">
```

If foo equals 'xyz' the DIV element will show up.

Note: the most common mistake using `*ngIf` is forgetting to include the asterik. If your code isn't working, that should be the first thing you look for.

Recently, the `*ngIf` syntax has been embellished to allow for THEN and ELSE blocks. When you use this syntax, you can tell it to use the content from another template by naming the template with a template variable.

```
<div *ngIf="show; else elseBlock">
  Text to show
</div>
<ng-template #elseBlock>
    Alternate text
</ng-template>
```

In the code above, "show" is a boolean variable in your class. "Text to show" displays if "show" is true. "Alternate text" will display when "show" is false.

You can make everything use templates by specifying a then template.

```
<div *ngIf=
  "show; then thenBlock; else elseBlock">
    Never shows
</div>
<ng-template #thenBlock>
    Text to show for then
</ng-template>
<ng-template #elseBlock>
    Text to show for else
</ng-template>
```

In this case, the DIV is completely ignored and the content that is displayed is from the templates.

ngSwitch

If you had a lot of conditions, you could use `*ngIf` this, `*ngIf` that, `*ngIf` something else. But just like in JavaScript, TypeScript or just about any other language a switch statement handles this in a much more elegant way.

To make this work in an Angular template, you'll need to specify an outer container that holds the `ngSwitch` directive. This can be an `ng-template` or any other valid element.

```
<div [ngSwitch]="memberVariable">
 ... </div>
```

Notice that we use the bracket notation because we are assigning the value to `ngSwitch`.

Then we test the value in `ngSwitch` against a series of values using `*ngSwitchCase`

```
<div *ngSwitchCase="value1" >
  ... </div>
```

Now, if the value of `memberVariable` is "value1" this div block would show and the other div blocks with other `*ngSwitchCase` values would not appear. If you have multiple `*ngSwitchCase` values that match the value of `ngSwitch`, each will be displayed.

ngFor

Now, `*ngFor` is probably the single most used conditional directive available in the Angular ecosystem. What this simple little directive allows us to do is to specify a block of html that we want to iterate over for every value in a list. Notice that I said "list" here and not an Array. This is because `*ngFor` works with anything that implements the `NgIterable` interface. This allows us great flexibility because we are not limited to arrays. `NgIterable` is defined as either an `Array` or an `Iterable` type. An `Iterable` type is anything that has implemented the `Symbol.iterator` property.

Huh?

You are probably already familiar with the `for...in` syntax from JavaScript. This iterates through the keys of an object. `for...of` iterates through the values of those keys. `*ngFor` will iterate through either an array, or a list of property values.

In addition, ES2015 supports additional types that also support `Symbol.interator`. `Map` and `Set` are two such types. If you are not yet familiar with these, you should look them up.

Pipes

Often the source of data we want to display in a component is in a different form than how we want to display it. The most common situation is dates. You could control how the date is displayed by binding to a property in your class that formats the date for you. Or, you could use the Date Pipe.

All Pipes work the same way. The basic syntax is:

```
{{ data | Pipe:parameter1:parameter2 }}
```

All Pipes take the data variable as a parameter, but they can also take other parameters. These show up after the Pipe name and are separated by the colon operator.

In the case of the Date, there is one parameter that is a string that controls how you want to format the date.

```
{{ dateVar | date:"MM/dd/yy" }}
```

Now, here is a tip for you. It is possible for the output of one pipe to be the input for another pipe.

```
{{ dateVar | date:"fullDate" | upperCase
}}
```

Safe Reference

Let's say you need to access data in your class that is embedded in another object. Typically, you would access that data using simple dot notation.

```
{{ mainObject.childValue }}
```

But often, the parent object doesn't exist when the view is being rendered the first time. Or it may not ever exist. If you simply use dot notation to access the child, the application will throw an exception, just like it would if you were accessing an undefined or null variable in your TypeScript code. In TypeScript, we could test for this condition with a simple if statement. And we could use a ternary operator in our template to achieve the same effect. But, the template syntax allows us to shortcut that syntax by prefixing the dot with a question mark.

```
{{ mainObject?.childValue }}
```

If `mainObject` exist, then `childValue` will be displayed. Otherwise, nothing will display at that location until `mainObject` shows up.

Non Null Assertion !.

In TypeScript 2, you can enforce strict null checking with the compiler option --strictNullChecks. But if you do this, you may run into a condition where you know a variable in your template will never be unintentionally null, but the compiler doesn't know. To tell the compiler, that you are sure it won't be null you use the non-null assertion operator

```
{{ mainObject!.childValue }}
```

The effect of this operator is that it turns off strictNullChecks for this code.

Forms

All that observable stuff we talked about carries over into forms. From my experience, this is one of the most powerful and overlooked features of Angular.

Templated Forms

If you are coming from the old AngularJS world, you are familiar with binding form items to variables in your JavaScript, or in our case, TypeScript files. This method of getting the data out of the form is called "Template Driven Forms." The syntax for this is pretty simple

```
<form ngForm >
  <input [(ngModel)]="memberVariable"
    name="elementName" type="text">
  <button (click)="onSave($event)">
    Save
  </button>
</form>
```

The onSave() method in our TypeScript class will then retrieve the data from the member variable(s) and send them on to wherever they need to go next.

I am purposely not giving you a lot of detail on how to use this approach because I don't want you to use this as your primary means of developing forms. My basic issue with this method is that it does not scale well for more complex forms. This isn't to say you can't create more complex forms using templated forms, we've been doing it for years, but compared to the alternative,

any method we might develop to create complex forms using templated form is going to look like a hack.

There are still times where you might consider using this approach to forms, but there is a better way. Reactive Forms.

Reactive Forms

Reactive Forms gets the name from the fact that they rely on everything I just told you about Observables. Instead of having the template push and pull data from the TypeScript class, we set up an observable on the form so that every time the form changes, we are notified. When we want to change the form data, we push the data up to the form in the template.

So far, this might just sound like two ways of doing the same thing. But, because we are working with Observables and not just data, the debounce logic I showed you previously can now easily be applied to the entire form. This, combined with the Redux design pattern that I will show you later using NgRX, allows us to solve several otherwise awkward programming problems we have historically had to face.

Forms

To create a reactive form, our form in our templates look just a bit different from their counterpart template based forms.

```
<form [formGroup]="form">
  <input type="text"
    formControlName="myControl">
</form>
```

And then in our TypeScript file, the relevant code would look like this:

```
export class MyFormClass {
  form: FormGroup
  constructor(formBuilder: FormBuilder) {
    this.form = formBuilder.group({
      myControl: ['',
        Validators.required]
    });
  }
}
```

The first thing you'll notice is that we have created a member variable in our class named "form" that is of type `FormGroup`. This corresponds to the form we assigned to `[formGroup]` in our template.

The second thing you'll notice is that we assigned the `formControlName` property the value of "myControl" which corresponds to the `myControl` property in the object we passed to `formBuilder.group()`.

Third, you'll notice that we've attached our validations as part of the `formBuilder.group()` in our class. This is where we start to see some of the additional power of using Reactive Forms. We'll talk more about this later.

Finally, for all this to work, we've injected a `FormBuilder` into our class so that we can do all this. `FormBuilder` is part of `ReactiveFormsModule`, so you'll want to make sure that has been imported into one of your modules. I typically include it in `app.module` because all my forms in the application are going to need it.

Now, on the surface this may not appear too earth shaking. But, let's pick up the conversation about validation. Using templated

forms, if you wanted to add custom validation, this would require that you write a validation as a directive that could be added to a form element. Not impossible, but more work than is required with Reactive Forms. You see, using Reactive Forms, you create a static function that takes a `FormControl` as a parameter. You can put that validation function in the class the component is a part of, or someplace else more globally available. Where you put it will depend on architectural decisions you'll need to make on a case by case basis.

Here is a sample validation function I wrote to validate that a date is valid. It's a sample. You'll probably want to embellish it.

```
static isDate(c: FormControl){
  if(!c.value
    .match(
      /^\d{1,2}\/\d{1,2}\/(\d{2}|\d{4})$/
    )
  )
  return {invalidDate:true};
}
```

To include this validation, you would place it as a parameter in your `FormGroup` creation.

```
export class MyFormClass {
  form: FormGroup
  constructor(formBuilder: FormBuilder) {
    this.form = formBuilder.group({
      myControl: ['', MyFormClass.isDate]
      });
  }
  static isDate(c: FormControl){
```

```
    …
  }
}
```

You can also combine validations by using `Validators.compose()`.

```
export class MyFormClass {
  form: FormGroup
  constructor(formBuilder: FormBuilder) {
    this.form = formBuilder.group({
      myControl: ['', Validators.compose([
            Validators.require,
            MyFormClass.isDate]
      )]
    }
  }
...
}
```

Displaying Errors

If all we are validating is that the control has a value, our ability to display an error is simple. One way you can write your template would be like this:

```
<div *ngIf="form.controls['name'].errors
&& form.controls['name'].touched">
  Name is required
</div>
```

Where 'name' is the formControlName we gave our control. What this is saying is, "if there are any errors and the user has made changes to the control, display this block of HTML."

In the case where you only have one validation, you can also use:

```
<div *ngIf="form.controls['name'].valid &&
form.controls['name'].touched">
  Name is required
</div>
```

When you are using multiple validations and you want to display a different message for each error, you will need to use the `hasError()` method. You pass in a string that matches the name of the error property that represents the validation. In the case above, "invalidDate" is the property we are looking for.

For the validators that come with Angular, the strings you'll need to use are:

- Validator.min - "min"
- Validator.max - "max"
- Validator.require - "required"
- Validator.requireTrue - "required"
- Validator.email - "email"
- Validator.minLength - "minLength"
- Validator.maxLength - "maxLength"
- Validator.pattern - "pattern"

Using this information, we can display validation errors in our templates when the user has made a change and a validation fails.

```
<div *ngIf="form.controls['name']
  .hasError('required') &&
   form.controls['name'].touched">
```

```
  Name is required
</div>
```

You may also want to enable or disable buttons based on the valid state of the form. You can use the `valid` property on the `formGroup` and set the disabled property.

```
<button [disabled]="!form.valid">
  Save
</button>
```

Where "form" is the name of the `formGroup`. Notice that here we don't care if the form has been touched or not because we don't want them saving if the form is invalid even if they haven't touched the form.

Retrieving Data

While template based forms update the variables in your class file as you change the data in the form, Reactive Forms work by turning the form into an observable. Every time a change is made to the form, the observable places a new element in the que with the current state of the form. While this may seem awkward and counterintuitive at first, this gives us some flexibility that we don't have with Templated Forms. One of those advantages is that we can write debounce logic for our forms so that we only deal with the changes when the user has stopped making them.

Given that we have a formControl named "form" our reactive code might look something like this:

```
this.form.valueChanges
  .debounceTime(500)
  .subscribe(value) => {
   // store value someplace
```

```
    // so you can get at it later
  });
```

Right now, you are probably wondering how this is more effective than using a Templated Form. But before we dive further into why you want to write your code in this way, we need to show how to update a Reactive Form with new data.

Updating Form Values

Since we have a way of getting data out of our forms, there must also be a way to get data back into our forms. And the way we do that is with the `patchValue()` method. The patch method takes an object with properties matching the names of the controls on our form and values that the controls should be set to. That much is straight forward. Where things get interesting is that if you just do that much it will trigger a form change event that will then cause our subscription to fire. Probably not what you had in mind.

So, instead, what you want to do is to pass in an option as the second parameter to `patchValue()`.

```
this.form.patchValue(dataChanges, {
  emitEvent: false
});
```

`emitEvent: false` tells `patchValue()` to not fire off that `valueChanges` event in our form when it makes the change.

Listening for Changes to One Control at a Time

At some point, you are going to want to know when one of the controls have changed so you can do something unique.

The easy way to do this is to subscribe to the control like you subscribe to the form.

```
this.form.controls['controlName']
    .valueChanges.subscribe(...);
```

1-Way vs 2-Way Data Binding

Working with Templated Forms feels like the old 2-Way Data Binding that we used in AngularJS, even though it is still just 1-Way Data Binding that feels like 2-Way Data Binding. But Reactive Forms both feels like and IS 1-Way Data Binding.

The real difference between the two is that Reactive Forms tends to force us to think about our forms in terms of a group, rather than as individual parts. It also gives us the flexibility to store the state of our forms somewhere else other than our forms. From my point of view, this is the most important reason for moving toward a Reactive Forms model.

You see, the danger of storing our form state in our form is that we then start writing logic in our form that really belongs elsewhere in our code. This is not unique to Angular. In the 30 years that I've been programming I've seen this mistake made multiple times regardless of the design pattern used.

One of the first places I saw this was back when I was writing C++ code using Microsoft's MFC framework. There was this design pattern they used called, "Document/View." If you aren't familiar with Document/View, all you really need to know about it is that it is MVC without the Controller. Programmers had no real idea where to put logic. Some would put it in the View. Some would put it in the document. Personally, I put it in the

Document unless I could be sure that the code I was writing was only ever going to be used by the View.

Then, in the multitude of MV* frameworks we had a similar problem. Even though we had a Controller where we might put things, people would still put logic in the View.

Now, my issue with putting logic in the View is that the part of our application that is most likely to change is the View. And so, you must ask yourself, every time you put logic in your View, "If I had to rewrite this View using something else would I need to duplicate this code?" or said another way "Is this code only required to make this presentation render the data it is given?"

In the Angular world, you might think, "If this Web application suddenly became a Mobile application and I were no longer using HTML, would I still need this code and would I want it available further down the stack than where I currently have it now?" Or even more extreme. If you suddenly decided to make your web application a desktop application with an entirely different display mechanism, would you need to duplicate this code?" Or, to take it even further. Let's say you don't even have a View. Maybe you want to automate everything using some sort of AI engine. Could you do it? Or would you need some sort of View for your code to work?

And so, my basic problem with 2-Way Data Binding is that it does not make it easy to get the data outside of the component. While 1-Way Data Binding not only makes it easy because we are working with a set of data at a time, but it encourages it.

The View should only do two things. First, it should render the data it is given. Second it should tell the outside world when something happened that might be interesting.

Because it only does these two things, you should never need to write a Unit Test for View code.

Think about it. If I only render what I've got and fire events there shouldn't be any conditional code that needs to be tested.

Yes, there are conditions about when to display things. But that is code that should not be all that complex. Even the error message code I wrote above either works or it doesn't. Something that should be easily verified during integration or application level testing.

Often people reach out to me on the Angular Slack channel and ask me the question, "How do I access that component from this component?" If the answer isn't, "use ViewChild()." You are probably violating the principles I've been talking about.

External State

There are many advantages to getting your state information outside of your components beyond being forced to think about the state as something that isn't necessarily a view thing.

First, if we put the state information outside of the components we are working on, that state will persist for the life of the application.

Using one of the older models of keeping the state with the component, if the component is destroyed and then we come back, we start off with a clean form unless we retrieve the information from the database. Think of this as a "Clean by

default" model. But, by placing the state external to the component and making that state information globally available, when we come back to the component, we can retrieve the state from the last time we had the component loaded.

Another advantage to making the state globally available is that now we don't have to have a component to access the state. This means that all our business logic is view agnostic. This makes it much easier to write tests for our code. This means you are more likely to write tests for your code.

But all this advantage does come at a slight cost. It requires you to start thinking about your code in terms of data and processes instead of how it is going to look. And this is the most difficult hurdle to jump over as you move toward this new paradigm. And yet, given all you should gain, it is a hurdle worth learning how to jump over.

Of course, we aren't going to leave you to create your own framework for this. One has already been created for you.

NgRX

History

If you used AngularJS you are probably familiar with the MV* design pattern that has proliferated ever since knockout.js was introduced. This pattern, which started with the Model View Controller pattern and was made famous by the Gang of Four book "Design Patterns" was well suited for writing desktop applications where it was easier to have the View, Controller, and Model separated and distinct. MVC never really worked well on the browser and so, the MVVM pattern which also started on the desktop, was the pattern that most of us used during those early days.

The problem with MVVM is that to implement it well, you have to determine if your data has changed or any data it is looking at has changed. This becomes problematic because, you are forced between two choices. Slow or complete. Pick one. Because you can't have both.

Now, an additional problem with design patterns as with just about anything that becomes a buzzword, is that many people think they understand how to use MVVM or MVC or even Redux or NgRX because they can write code that results in a working application. So, before we move forward with an explanation of Redux or more specifically NgRX, let's look at design goals that a good design pattern should try to achieve.

Separation of Concerns

A place for everything and everything in its place. That's the mantra for good organization. And organized code is easy to follow. By placing our data in one place, our business rules in another place and our view in yet another place, it becomes easy to switch each out as needed.

As I've helped organization with JavaScript design patterns, I've noticed that most programmers tend to want to put all their logic in the view. Let me tell you why that's a bad idea.

Let's say you are developing a web site. You put it together in such a way that the presentation layer asks for the data it needs directly and then displays it on the screen. While you are at it, you place all your logic in the same view code. You manage to get it all working and shipped. But six months go by and finally someone ask for a change. Because you've placed all your code in basically one file, the code that seemed easy to maintain because it was all in one place suddenly feels like a monster. How do I get from here to there? What makes this display at the right time? If I change this code here, will everything still work?

Maybe you've had to maintain systems like this.

Now, if that isn't bad enough. Let's say that the change is a little more drastic. Maybe we want to move to mobile. We can do that in two ways. First, we could just trim down our HTML template file so that it only displays for certain resolutions, or we could move to a mobile framework that doesn't use HTML at all, while still conforming to the templated markup we typically use for HTML. In either case, our template is going to change.

If you've put all your logic, or even some of your logic, in the View layer you'll need to repeat that logic in your new View. Awkward at best.

Finally, by placing all your logic in code that isn't your view, you don't need the view available to write your unit tests. While I understand that you may not write unit tests. I bet the reason you don't is because writing tests are difficult. And they are difficult because your code is structured poorly.

There are similar benefits to putting your logic in its own location and the state information, or data, in another location that is well known. But in my experience, if you can keep the logic out of the View, you are likely to put it where it rightfully belongs.

Performance

Another critical metric to evaluate design patterns against is performance. Performance at scale. Something the AngularJS community is more than familiar with. The MVVM model we typically used with AngularJS worked well for small applications, but as our applications grew it became obvious that MVVM wasn't really the best tool for the job.

And while there are features in Angular that look a lot like MVVM, it really isn't. Even so, if you use Angular like you used AngularJS, you are going to end up with similar performance issues. The performance is better, but then we are responsible for more of the change detection as well.

Completeness

Finally, we want whatever model we implement to be complete. I've worked with other frameworks that have improved

performance of MVVM by leaving out critical features. Can you get stuff done? Sure. But not without pain.

Redux using NgRX

The Redux design pattern has its roots in the React library. While not part of React, it is a morph of a design pattern, called Flux, created by Facebook to handle problems they were seeing in their own code using the older MV* pattern.

Weather it is Flux, Redux, or NgRX, in every case there are some basic principles at play.

First, we want to get away from managing state in our components. For that matter, we want to eliminate application logic from our components. All our presentation layer should do is reflect data it has been given and tell the application to do something (that may or may not change the data) when it is appropriate.

Now, you are probably asking. If I don't store my state in my presentation layer. In my components. Where do I store it?

Managing State

You store state in an application store. In Angular, this Store is injectable, so it is available globally. This is not such a strange concept. Think of your store as a client side, nosql like, database that manages the state of your application.

The next obvious question may be, "How do I tell my presentation layer that something in the state has changed?" Well, you don't. At least, not like that. You set up code that "listens" for changes to state and updates the presentation layer.

The whole system uses a series of publish/subscribe mechanisms, or you may think of it as a message based system. For those of you old enough to have programmed Windows 3.x, the pattern will feel somewhat reminiscent of the message queue we worked with when we wrote programs for Windows desktop applications.

The whole mechanism works in a circular fashion, so it is difficult to pick a starting point when describing how this all works. But, since the Store seems to be the central mechanism, let's start there.

Assume we have a store, and in that store is a "wait" entity that holds an integer.

In our presentation layer, we have a "wait" component that displays a wait spinner and grays out the screen when some long running process happens.

To show the "wait" component, we add an `*ngIf` that shows the component when the wait integer is greater than zero. If it is zero, we don't show the component. To do this, we select the wait entity in our component that contains the wait component and the component looks at the variable we assign the wait observable to. Now, every time the wait counter is greater than zero, our wait component displays.

But how do we increment the wait counter?

To increment the counter, we dispatch an action that is picked up by a reducer. In our action, we send two bits of information typically. The action we want to perform, and any data that action may need to do the task.

Our reducer listens for these actions and when the start action fires, it will increment the wait counter.

Similarly, we would create an end action that would decrement the counter.

Whenever the counter changes, the observable we setup will notice and the wait component will show or hide appropriately.

Everything is Functional

Now, in NgRX (and Redux) everything that happens in a reducer happens in a very Functional way. If you aren't familiar with functional programming, there are only a few things you really need to understand about Functional Programming that will help you create great reducers.

Data in Data Out

First. If a function is given the same parameters each time, it will always return the same value. You may think this is obvious. But, let me show you why it is not.

In just about every other programming model, and especially in Object Oriented programming, you may have a function that takes two parameters. But the function itself calls the system clock either directly or indirectly and uses that as part of the computation for the return value. There is no way this function is going to return the same value every time it gets called.

Another similar example is that you may have an object that has a member variable. Another function that takes N parameters also looks at the state of the member variable when determining the return value.

In both scenarios, being able to test these functions becomes troublesome. And it is because of these issues that we've created work arounds like Dependency Injection so we can control these hidden parameters in our tests.

In a functional world, this problem doesn't exist. The same parameters give us the same return value every time. And so, we also have a side benefit of being able to store the computed value in a lookup table so that the next time we pass those same parameters, we can just return the value and skip the computation. We don't do this all the time, but it is an option for a function that is only going to respond to a limited number of parameter values.

Immutability

The second rule of functional programming is that all data is immutable. What this means to us is if we return an object from a function that is based on a parameter of that function, we return a new object. You might think this would make the system slow. But, it makes it faster.

Remember I said that our observable gets notified every time our data changes? If our data is an object, it is much more efficient to determine that we have a new object than to try to detect if one of the values in the object changed. In fact, trying to figure out if a value in an object changed is one of the reasons that AngularJS was incredibly slow. By ensuring that our view only changes when an object it is looking at changes, we can create incredibly performant web applications.

Full Cycle

So, let's put this together. We have an entity in our store called wait that our store is listening to. When a long running process starts, we send a "start" message to our reducer and our reducer increment the wait counter. Our presentation layer sees that our wait counter is now greater than zero and displays the wait component. When the long running process completes, an "end" message is sent to the reducer which decrements the counter. The presentation layer now sees that the counter is zero and the wait component is removed from the DOM.

Of course, this is a simple example. What about something more complex?

What About Data?

This time, we want to create an edit screen that loads data from the server when it displays and sends the data back to the server when a "Save" button is clicked.

The first thing you should notice about these two scenarios is that both immediately violate functional programming principles. In the first case, we may pass in an ID to retrieve the record we want to edit. There is nothing that ensures that every time we pass in that ID that we will get the same set of data back. It may have changed since the previous time we made that same call.

In the case of saving the data, when everything works correctly, we might get the same value back. But what if it doesn't? Maybe the server is having trouble or is down.

Fortunately, all Functional programming implementations recognize that real programming can't happen without side

effects, and NgRX is no different. Therefore, NgRX has created Effects. Effects are like reducers in the fact that they respond to Actions that have been dispatched. But, unlike Reducers, Effects are member variables in a class that are Observable streams that return other actions.

Now, don't worry so much about what this looks like in code just yet. I want to make sure you have a solid mental image of what this looks like first.

Now, back to our example.

You load up your component that is looking for a record to edit from the server. So, the first thing you will probably do is you'll setup a listener to the entity that will hold the state information you want to edit. Once that is in place, the next thing you'll do is that you'll send an Action out telling your Effect to load the data. Once the data has been loaded, your effect will return an Action, that the system will dispatch for you, telling your reducer "here is the new data" at which point your entity will get updated with the data which will cause the listener you set up in your component to recognize there is new data and will display the data on the screen.

Implementation

Now that we understand the theory, how do we write code to implement this?

I've hinted at three types of code we will need to implement to make NgRX work in our code. Actions, Reducers, and Effects. There is a third type, the Store, which is provided by NgRX which we will need to register our Reducers and Effects with.

Before we get started, make sure you have TypeScript 2.4.x or above installed in your local project. The CLI may complain, depending on what version of it you are using. But, NgRX 4 requires us to use TypeScript 2.4.x. You should also have RxJS 5.4.x or above installed.

You will also need to install @ngrx. You can do this using the following NPM command:

```
npm install --save @ngrx/store
  @ngrx/effects
```

All on one line.

Actions

An Action is an object that contains a type variable and optionally, a payload. Depending on how you code your action, the payload may or may not have "payload" as the variable name. In NgRX version 2, payload was an optional variable. To improve type checking, payload was removed from the Action interface.

The official documentation for NgRX version 4 encourages us to create a class for each action we want to dispatch.

Using our Wait example from above, you might want a Start wait action and an End wait action. So, you would create a wait.action.ts file that has two classes in it. A Start action and an End action.

```
 import {Action} from '@ngrx/store';

export const START = 'Wait.Start';
export class Start implements Action
{
```

```
    readonly type = START;
}

export const END = 'Wait.End';
export class End implements Action {
    readonly type = END;
}
```

To use these actions in our code, we would import them as a bundle:

```
import * as Wait from './wait.actions';
```

And then we would dispatch the action using the store we've injected into our code. We'll cover that later. But for now, the dispatch basically looks like this:

```
this.store.dispatch(new Wait.Start());
```

What makes this method of creating actions useful is that when we need to pass additional data along with our action, we can do that simply by adding parameters to our constructor.

```
export const ADD = 'Wait.Add';
export class Add implements Action {
    readonly type = ADD;
    constructor(public payload: number){}
}
```

And if you need more than one payload item, you can pass in multiple parameters.

Reducers

Reducers are functions that allow us to change the state of our entity within our application Store. Reducers respond to the

actions that have been dispatched and return a new object based on the changes requested.

To continue on with our Wait example, `wait.reducer.ts` might look like this:

```
import {ActionReducer} from '@ngrx/store';
import * as Wait from './wait.actions';

// This could go in wait.actions.ts
export type Action =
    Wait.Add | Wait.End | Wait.Start;

export function waitReducer(
  state = 0, action: Action): number {
  switch(action.type) {
    case Wait.START:
      return state + 1;
    case Wait.END:
        return state - 1;
    case Wait.ADD:
        return state + action.payload;
    default:
      return state;
  }
};

export const WaitReducer:
    ActionReducer<number> = waitReducer;
```

The reason we write the waitReducer function and then assign it to the WaitReducer constant rather than putting the code all in

one line is because the compiler will complain about our code otherwise.

Notice that we have a 'default' in our switch statement. Don't forget to add this. It isn't an accident. You see, when we register our Reducers and Effects with the store and then subsequently dispatch actions to them, all the reducers and effects get called. In the case of reducers, if you don't return something for each one of them, you'll end up with a store that doesn't know what kind of state it is in. So, always return the current state as your default.

The one thing that may not be obvious here is that the Reducer is a function. Not a class. This means you can't inject other classes into it. This is one of the reasons that the file is named `wait.reducer.ts` and not `wait.reducers.ts`.

But, Effects are different.

Effects

Effects are classes. Each Effect within the Effects class is a member variable. So, we name our Effects file for Wait, `wait.effects.ts`.

Now, remember I said that Effects were for things that caused side effects? You may wonder, why kind of side effect does our Wait stuff need?

Under normal circumstances, not any. However, you can get into a situation in development mode where one of the safe guards detects that you've changed the state of wait multiple times in a change detection loop. Angular expects changes to be relatively static.

One way we can deal with this is to delay incrementing and decrementing to another change detection loop using setTimeout() indirectly.

To do this, we are going to rip out the START and END case statements from our Reducer and add them into our Effects class.

And then in `wait.effects.ts`

```
import { Observable } from
    'rxjs/Observable';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';
import 'rxjs/add/observable/timer';
import { Actions, Effect } from
    '@ngrx/effects';
import { Injectable } from
'@angular/core';
import * as Wait from './wait.actions';

@Injectable()
export class WaitEffects {
    @Effect()
    start$: Observable<Wait.Add> =
            this.actions$
        .ofType(Wait.START)
        .switchMap(() =>
            Observable.timer(1).take(1)
        )
        .map(():Wait.Add =>
            ({type: Wait.ADD, payload:
1}));
```

```
    @Effect()
    end$: Observable<Wait.Add> =
            this.actions$
        .ofType(Wait.END)
        .switchMap((action: Wait.End) =>
            Observable.timer(1).take(1)
        )
        .map((): Wait.Add =>
          ({type: Wait.ADD, payload: -
1}));

    constructor(private actions$:
Actions){
    }
}
```

You can name the member variables whatever you want. They never get used. `ofType()` is a filter that makes the observable stuff only fire for that particular type.

The magic happens in the `Observable.timer(1).take(1)`. Here we wait for one millisecond, take the first item out of the stream and immediately close the observable. Once that completes, the `map()` returns a new action.

Every Effect must return an action or you must tell the Effect that it won't return an action. Notice that we don't dispatch the action. That is done by NgRX for us. We just return the action.

To tell an Effect that no action will be returned, pass `{dispatch: false}` into `@Effect()`.

```
@Effect({dispatch: false}) …
```

You might wonder why we import each observable thing we need individually. By doing this, we only import exactly what we need, which causes the resulting bundled code to be smaller. Smaller code means we get a faster load time.

Returning Multiple Actions from Effects

At the other end of possibilities, you may need to return multiple Actions from an Effect. One way to do this is by using `mergeMap()`.

```
.mergeMap((/* previous value here */) =>
{
    return [
        new ActionGroup.Action1(),
        new ActionGroup.Action2()
    ];
})
```

Registration

None of this is going to work if we don't register our Reducers and Effects with the NgRX Store.

If you look at a lot of the literature on how to write NgRX code, you'll often see that they recommend that you put the code in `AppModule`. That will work, but how much more effective to put the code for your Store in its own module. So, I create a Module class called `AppStores` and put it in a file named `app.stores.ts`. I know, technically it is a module and it should be `app-stores.module.ts`. Using `app.stores.ts` isolates it from a normal module. All I want to put in here is Store stuff.

So, in my `app.stores.ts` file, I put code that looks like this:

```
import { WaitEffects }
    from './wait.effects';
import { AppState } from './app.state';
import { WaitReducer }
    from './wait.reducer';
import { NgModule } from '@angular/core';
import { StoreModule, ActionReducerMap }
    from '@ngrx/store';
import { EffectsModule }
    from '@ngrx/effects';

const reducers: ActionReducerMap<AppState>
    = {
        wait: WaitReducer
    }

@NgModule({
    imports: [
      StoreModule.forRoot(reducers),
      EffectsModule.forRoot([WaitEffects])
    ]
  })
export class AppStore { }
```

And the AppState interface that I used for my reducer map looks like this:

```
export interface AppState {
    wait: number
}
```

A couple of things you should notice about this code. First, the reducers object we create at the top of the file describes what our store looks like. So far, we have an entity named "wait" that is controlled by our `WaitReducer`.

Then to register the reducers with our application, we pass them on to `StoreModule.forRoot()` in the imports section of our module definition.

Similarly, we must register our Effects with the application. We do this by passing an array of Effects to `EffectsModule.forRoot()` in the imports section of our module definition.

Of course, none of this code will even get included in your project unless you import this module into your `AppModule` class.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppStore
  ],
  providers: [],
  bootstrap: [AppComponent]
```

```
})
export class AppModule { }
```

Retrieving Data

Retrieving an entity from a store is quite simple. Leaving out the imports that a good editor should help you with, here is the relevant code:

```
export class AppComponent {
  wait: Observable<number>;
  constructor(store: Store<AppState>) {
    this.wait = store.select((s: AppState)
      => s.wait);
  }
}
```

And now you can subscribe to `wait` and whenever the value changes, you'll get a notification and can do something with it.

Lazy-Loading NgRX

Prior to NgRX version 4, all our state stuff had to live in the root of the application. This was problematic because it also meant we were unable to easily locate our Actions, Reducers and Effects with the Routes they belonged to. So, we had feature level components but everything else was more function based. That is, all our State stuff lived together in one directory separate from the feature they supported. Ugly!

But now, we can create Reducers and Effects that live with the feature we support.

Always Import forRoot()

The first thing you'll need to be aware of is that even if you aren't storing any state at the application level, you will need to call `StoreModule.forRoot()` and `EffectsModule.forRoot()` with an empty object and array.

```
imports: [
  StoreModule.forRoot({}),
  EffectsModule.forRoot([])
]
```

AppState vs FeatureState

You'll remember that you'll typically use an interface called `AppState` to define the structure of the Store for your application. For reasons we'll get to later, you are going to want to create a separate interface for each set of feature reducers you are loading.

Feature Name as AppState Property

When you add a feature reducer, you'll need to supply a name as the first parameter.

```
imports: [
    StoreModule.forFeature('featureName',
        featureReducers),
    ...
]
```

Where `featureReducers` is the map of reducers for the feature. Like what we did for the root level reducers.

This “featureName” becomes the name of the store entity you’ll need to select to get at the store entities in your feature reducers.

Imagine you have a feature named “featureName” as we’ve coded above and your `featureReducer` object has a feature property named “sub”. Not super original, but it will do for an example.

To select “sub” from your store, you would use code that would look something like this:

```
store.select(s =>
  s.featureName.sub);
```

This means that if you want to strongly type your selections using AppState, you will need to define a field in your AppState interface as “featureName” that is typed as the State interface for that feature. Let’s call that FeatureState.

```
export interface FeatureState {
    sub: SubModel;
}
```

And then our AppState would look like this:

```
export interface AppState {
    featureName: FeatureState
}
```

And now we can write our select code like this:

```
store.select((x: AppState) =>
    x.featureName.sub);
```

If you have Effects that go with your reducers, you’ll also need to import them with

```
imports: [
  EffectsModule.forFeature([
    …list of effects here
  ])
]
```

Features Without AppState

It is possible to access the feature state without putting it in your AppState. I'm not a big fan of this method generally, because you lose a bit of type safety in the process. But, there are time when you may have feature code that can't know about the AppState. For example, you may have code that lives in a library (separate package that you've imported) that needs to access the state slice. Since it doesn't know, and can't know, about the AppState of the application, this alternate method may be the only way you can access the code.

To implement the access method, you use the `createFeatureSelector()` and `createSelector()` methods which you can import from `@ngrx/store`.

The above code using these methods would look something like:

```
const featureSelector =
  createFeatureSelector<FeatureState>(
    'featureName');
const subSelector =
  createSelector(featureSelector,
    (x: FeatureState) => x.sub));
store.select(subSelector);
```

Where store is injected using:

```
constructor(store: Store<FeatureState>) {}
```

Note: I'm not suggesting that you write your code like this. If you are repeating the `createFeatureSelector()` `createSelector()` code in multiple places, you should look for a way of not repeating yourself. I've put it all in one place here, so you can see how the methods tie together in the bigger picture.

Features Without Lazy Loading

One misconception that you may have about features is that they must be lazy loaded. This could not be further from the truth. Features exist so that we can lazy load our NgRX code. But, you could implement all of your NgRX code as features as long as you have included empty forRoot() calls for the Store and the Effect

Retrieving Data

Every application we write needs to access some external data. Some more than others. There are two ways you might see in existing Angular code for retrieving and updating data. There is an `Http` Service and an `HttpClient` service.

`HttpClient` adds a lot of additional functionality. But the core concept you need to understand about both mechanism is that they use Observables under the hood. Get familiar with Observables because they are a core part of Angular.

There are a couple of advantages to using Observables to retrieve data from the server instead of using call-backs or promises as we have in the past.

First, we can cancel an Observable. This is built into `Http` and `HttpClient`. Make the same call two times in a row on the same Observable and you'll see the first call gets canceled.

Second, retrying a failed call is as easy as tacking on `.retry(2)` into the stream.

```
http.get('/api/getData').retry(2)
    .subscribe();
```

That's not real code. But that's basically how retry is done.

Additionally, there is both a `catch()` function and a `finally()` function that you can put in the chain. This is brilliant, and they would do exactly what you would expect. When an exception is thrown retrieving data from the server,

`catch()` is called. While, `finally()` is called regardless of what happens.

`Http` is the older way. When it requested data from the server, whatever was returned was returned as a string. You'll know you are working with `Http` code when you see some code that looks like this:

```
http.get(…).map((x) => x.json())…
```

That bit of `x.json()` code turns the string into a JavaScript object.

By switching your code to use `HttpClient` instead of `Http`, you immediately gain the benefit of eliminating this step while all your other code remains the same. But, you also have the additional benefit of using Interceptors.

Interceptors allow you to access data as it is being sent to and returned from the server. This is an advanced topic that we are going to ignore in this book.

`HttpClient` also allows us to receive event notifications as progress is being made uploading or downloading data.

GET

To make GET request from the server, the process is simple. But, you might be tempted to concatenate your own string when you are passing parameters. The problem with doing this, other than it is a lot more work than you need to do, is that you might end up passing data that can't be parsed correctly. The correct way to pass parameters is to create an `HttpParams` object and then use the `set()` or `append()` methods on that to set the

name/value pairs. Finally, you pass it into the optional second parameter of your get call.

```
const parameters = new HttpParams()
    .set('aaa',valueAaa)
    .set('bbb', valueBbb);

return this.httpClient.get(
    getUrl, { params: parameters });
```

Note that we are chaining using `.set(…)`

An alternate way to set parameters is to use `.append()`, which doesn't require chaining like we did above.

```
const parameters = new HttpParams();
parameters.append('aaa',valueAaa);
parameters.append('bbb', valueBbb);

return this.httpClient.get(
    getUrl, { params: parameters });
```

Using `.append()` you can conditionally add parameters to the list.

POST

Posting data is straight forward. You pass the object you want to POST to the server as the second parameter to the `post()` method. The first parameter is the URL.

Finally, each method is a templated method. This means, you can specify the type you expect the method to return.

```
httpClient.get<MyType>(url, …);
```

Which helps us to know our code is going to do mostly what we expect before we ever run it.

Import Http or HttpClient

To make `Http` or `HttpClient` available in your code, you need to import the `HttpModule` or the `HttpClientModule`, depending on which you want to use, in your `app.module.ts` file. Don't try to add `Http` or `HttpClient` to your "provides" section. You'll find out fast that this doesn't work if you do.

For more information on how to access data, you should check the Angular documentation on the Angular.io site.

Architecture

So far, we've spent a lot of time looking at the details of each part that will compose an Angular application. It is a lot like picking up each individual piece of a puzzle and examining it prior to putting a puzzle together. Or, to use another analogy, it is like becoming familiar with all the components involved in building a house before you set out to build your own house.

And now, like building a house, we are going to sketch out generally how we are going to assemble the parts when we build our application. We will see how all the parts fit together.

State

Since we just finished discussing NgRX, let's start there. Aside from the fact that we will be using NgRX to manage state for our application, what needs to be emphasized here is that we won't be managing state anywhere else in our application.

One of the mistakes I see developers new to Flux/Redux and now NgRX make repeatedly is that they somehow think that NgRX is only for managing the state we will be persisting. Nothing could be further from the truth. All our data should end up in our Store with very few exceptions.

Components

On a related note, our smart and dumb components will only be responsible for displaying the current state and sending notifications out to change state. Our components will not have any logic in them. There are two reasons for this. First,

components are difficult to test. Not impossible. But difficult. One reason for this is that the presentation template code is the one thing that is most likely to change over time. This means that every time you make a change in the presentation you'll need to update your tests to reflect this change. This is not an effective use of our time if we can avoid it.

Second, we don't want to test code that is library code. Only our code should have tests. So, if you follow my suggestion and only display what is in the current state and send messages to it, you will end up with methods and properties in your code that have zero complexity and are therefore not worth testing.

We'll cover testing later. You'll see that writing tests for components tends to be complex and you'll want to avoid it if possible.

Component Services

But, to say that components don't have any logic at all is insane. In recognition of this, we will be creating Component Services that we can inject into our components. Any component that has excessive logic will use the Component Service and will use the Façade design pattern to "proxy" calls from the component down into the service and reflect properties from the service back up into the component. By doing this, we can test the service without jumping through a lot of hoops while still allowing logic to be available to our component.

The added advantage to this strategy is that if we need to rearrange, or replace the components, we can do so without having to change a lot of the logic. This isn't always possible, but it is more likely if we design our code this way from the beginning.

Data Retrieval

One of the questions that came up while I was learning NgRX and writing my first application was, “Should we just access the data directly from the Effects or should we use a Service?” We decided that it is best to use a service since there is a lot of code that gets repeated within crud operations that would be useful to treat using Object-Oriented principles instead of Functional principles.

Building A Basic Application

Now, you may be tempted to skip this section. But, I'll warn you, we haven't covered everything you need to know yet about Angular. We've just covered enough to get started with an application. The remainder we can cover along the way.

Or, maybe you skipped right to this section. That's OK, I guess, if all you want is a step by step to building your first application, but you don't care why. However, I strongly recommend you read from the beginning or you may find some of what we do strange. You might even think I've done it wrong.

The application we are about to build is going to be a relatively simple contacts database. One screen will list our contacts. Another screen will allow us to add and edit contacts. Along the way, we'll use what we've learned so far and learn a few more tricks along the way.

Just to make life easy for everyone, I'm going to assume you are using VS Code. My suggestion is that you use VS Code while you are following along. When you are working on your own project, you can move back to whatever editor you want to use.

Here are the versions that I'm using for this tutorial:

- VS Code 1.18.0
- Angular CLI 1.5.2
- TypeScript 2.4.2

Some of the plugins I'm using that you might find useful

- Sort Typescript Imports

- StyleLint
- TSLint
- TypeScript Hero

I use other plugins, these are the ones that will specifically help you write the following code.

I'm going to assume that you've already installed NVM, Node, VS Code, Angular CLI and TypeScript.

Just one more tip before we get going. Angular development, as well as many other SPA frameworks, tends to be very disk intensive. If you don't already have an SSD drive for development, you should invest in one now.

All the code for this tutorial is available on GitHub at https://github.com/DaveMBush/get-started-with-angular. For this version of the book, I've created a branch named "master-CLI-1.5.2". The steps along the way will be branches off that. The previous version of this book is branched off "master". You can follow along and compare your work to the branches if you get stuck along the way.

Help! It doesn't work!

Along the way, you may put in some code, run it, and it doesn't do what you expect. Here are things to look for.

- Did you save all the files?
- Are you missing an import statement? I won't always provide the imports you need for two reasons. First, you need to get used to putting these in yourself. Second, a good editor like VS Code with the plugins I suggested, will do the work for you.

- Sometimes, stopping the server that you started with `npm start` or `npm run startDebug` will solve an issue.

1 - Getting Started

Create a directory where you want your project to live. To keep us all on the same page, let's call it angular-book. Open your copy of VS Code and select "File" -> "Open Folder" and then select `angular-book`. Once the folder has loaded, open the integrated terminal window using the `CTRL + ` ` shortcut. (Windows or Linux) or `CMD + ` ` (Mac).

At the command line, type in:

```
ng new angular-book --directory=.
  --routing=true
```

All on one line.

The `--directory` option tells the CLI to put the code in the current directory, otherwise it would create an angular-book directory inside the directory we are already in. The `--routing=true` option will stub out the top-level routing module for us. Since we want to use routing, we include this option. No reason to write more code than we need to.

Let this run for a bit. It may look like it has stalled, but it is still going. It just can't show what it is doing while it is installing the NPM packages you will need.

Once you are done, you should be able to type:

```
npm start
```

in your terminal window which will compile your code and start the development server. Once that completes, you should point your browser to `http://localhost:4200`

The code for this section is in the branch named "step-1-CLI-1.5.2"

2 - Enforcing Best Practices

Previously, I mentioned that I adjust my environment with several tweaks all aimed at enforcing best practices I've discovered along the way. I understand you may not agree with me and I certainly can't force you to comply. However, if you have a problem later, you may wish you had gone ahead and implemented this step.

Let's go ahead and put the pre-commit hooks in that I described earlier. Two commands should install all the packages we need.

```
npm install -g stylelint
npm install --save-dev pre-commit
    stylelint-config-standard
    tslint-immutable
    stylelint
```

By now, you should realize that second through fifth lines are all one line. I'll stop repeating that now.

Next, you'll need to add the `postinstall` script to the scripts section of your `package.json` file to make sure `stylelint` gets installed after we `npm install` the packages. We would typically do this after downloading a fresh copy from version control, or after deleting the `node_modules` directory.

```
"postinstall": "npm install -g stylelint"
```

Adjust the existing scripts and add the “startDebug” script

```
"start": "ng serve --aot",
"startDebug": "ng serve --extract-css
   --aot",
"build": "ng build",
"test": "ng test",
"lint": "ng lint --type-check --fix",
```

Put the following rules in your `tslint.json` file or copy it out of the “step-2-CLI-1.5.2” branch.

```
{
  "extends": ["tslint-immutable"],
  "rulesDirectory": [
      "node_modules/codelyzer"
  ],
  "rules": {
      "typedef": [true,
          "call-signature",
          "arrow-call-signature",
          "parameter",
          "arrow-parameter",
          "property-declaration",
          "variable-declaration",
          "member-variable-declaration",
          "object-destructuring",
          "array-destructuring"
      ],
      "array-type": [true, "generic"],
      "readonly-keyword": false,
      "readonly-array":
        [true, "ignore-local"],
```

```
        "no-let": false,
        "no-any": true,
        "cyclomatic-complexity": [true, 10],
        "no-unused-variable": true,
        "arrow-return-shorthand": true,
        "callable-types": true,
        "class-name": true,
        "comment-format": [
            true,
            "check-space"
        ],
        "curly": true,
        "eofline": true,
        "forin": true,
        "import-blacklist": [
             true,
             "rxjs",
             "rxjs/Rx"
        ],
        "import-spacing": true,
        "indent": [
            true,
            "spaces",
            4
        ],
        "interface-over-type-literal": true,
        "label-position": true,
        "max-line-length": [
            true,
            140
        ],
        "member-access": false,
```

```
        "member-ordering": [
            true,
            "static-before-instance",
            "variables-before-functions"
        ],
        "no-arg": true,
        "no-bitwise": false,
        "no-console": [
            true,
            "debug",
            "info",
            "time",
            "timeEnd",
            "trace"
        ],
        "no-construct": true,
        "no-debugger": true,
        "no-duplicate-super": true,
        "no-duplicate-variable": true,
        "no-empty": false,
        "no-empty-interface": true,
        "no-eval": true,
        "no-inferrable-types": true,
        "no-misused-new": true,
        "no-non-null-assertion": true,
        "no-shadowed-variable": true,
        "no-string-literal": false,
        "no-string-throw": true,
        "no-switch-case-fall-through": true,
        "no-unnecessary-initializer": true,
        "no-trailing-whitespace": true,
        "no-unused-expression": true,
```

```
        "no-use-before-declare": true,
        "no-var-keyword": true,
        "object-literal-sort-keys": false,
        "one-line": [
            true,
            "check-open-brace",
            "check-catch",
            "check-else",
            "check-whitespace"
        ],
        "prefer-const": true,
        "quotemark": [
            true,
            "single"
        ],
        "radix": true,
        "semicolon": [
            "always"
        ],
        "triple-equals": [
            true,
            "allow-null-check"
        ],
        "typedef-whitespace": [
            true,
            {
                "call-signature": "nospace",
                "index-signature":
"nospace",
                "parameter": "nospace",
                "property-declaration":
                    "nospace",
```

```
                "variable-declaration":
                    "nospace"
            }
        ],
        "typeof-compare": true,
        "unified-signatures": true,
        "variable-name": false,
        "whitespace": [
            true,
            "check-decl",
            "check-operator",
            "check-separator",
            "check-type"
        ],
        "directive-selector": [
            true,
            "attribute",
            "app",
            "camelCase"],
        "component-selector": [
            true,
            "element",
            "app",
            "kebab-case"],
        "use-input-property-decorator":
true,
        "use-output-property-decorator":
            true,
        "use-host-property-decorator": true,
        "no-input-rename": true,
        "no-output-rename": true,
        "use-life-cycle-interface": true,
```

```
      "use-pipe-transform-interface":
true,
      "component-class-suffix": true,
      "directive-class-suffix": true
  }
}
```

As of the Angular CLI version 1.5.2, you'll also need to update the version of codelyzer that gets used. Open the package.json file and change it from the version that is there (3.x) to ~4.0.1. I hope this gets fixed in later versions of the CLI.

```
"codelyzer": "~4.0.1",
```

Once you've changed that line, you'll need to run

```
npm install
```

to install the newer version of codelyzer.

Next, add a `.stylelintrc` file (note the filename starts with a dot) and add the following content to it.

```
{
  "extends": "stylelint-config-standard",
  "rules": {
    "indentation": 4,
    "no-empty-source": null,
    "selector-type-no-unknown": null,
    "color-hex-case": ["lower", {
      "message": "lowercase letters are
easier to distinguish from numbers"
    }]
```

```
  }
}
```

And finally, back into our `package.json` file to make a few more adjustments.

Add a csslint script to the script section:

```
"csslint": "stylelint src/**/*.css --fix",
```

And add the precommit section

```
"pre-commit": [
  "csslint",
  "lint",
  "test",
  "build"
],
```

At this point, you should run each of the scripts in the pre-commit section to verify that they all work before you commit.

Since we haven't added any CSS yet, that script should run without errors. However, we will need to fix some problems that the tslint process picks up.

In my code, as I write this, the first complaint is that type of the title variable isn't declared in app-component.ts. But if we define it as a string, we get another error stating that it is trivially inferred. This is a bug with the linter right now. So, we'll just disable the typedef check for this using:

```
// tslint:disable-next-line:typedef
```

The next place we see a similar problem is in `environment.ts` is expecting a typedef. I just disable that check with:

```
// tslint:disable-next-line:typedef
```

This isn't our code and we won't be touching it, so we don't care about it. While you are turning off typedef for `environment.ts`, also turn it off for `environment.prod.ts`.

Another place where you can just turn off typedef is in `main.ts`. Once again, this is not our code and not code we will touch.

You'll see there is a type error on line 20 of `test.ts`. This can be corrected.

Change line 16 so that it defines __karma__ as having a loaded member and a start member which lint complains about later on in the list of errors:

```
declare const __karma__:
    {loaded: Function, start: Function };
```

And then on line 20 give the noop function a void return.

```
__karma__.loaded = function (): void {};
```

This leaves line 28 which can be fixed by specifying that require has a context function on line 17

```
declare const require: {context:
Function};
```

And specifying that context is an object with a keys function on line 28.

```
const context: {keys: Function} =
    require.context('./', true,
        /\.spec\.ts$/);
```

Now we can move on to the problems in `app.component.spec.ts`. This is a test file. The first error we see is on line 16. Once again, it wants a typedef. If you hover over `createComponent(AppComponent)` you will see that it returns `ComponentFixture<AppComponent>` so this is an easy fix. Type the `fixture` variable as `ComponentFixture<AppComponent>` and import `ComponentFixture` from `@angular/core/testing`.

```
const fixture:
    ComponentFixture<AppComponent> =
    TestBed.createComponent(AppComponent);
```

We can't really do anything about the next line so you can just disable the typedef. All the other lines are the same except for line 28 which has the same problem as the app variable, so we will just disable typedef for it too.

The final code we need to work on is the `app.po.ts` file which has two errors.

By hovering over the final return value, you can see what each returns. Typing `navigateTo()` to return `promise.Promise<any>` will still cause an error. What it really returns is an object so, we will type is as promise.Promise<object> instead.

`getParagraphText()` can be typed with `promise.Promise<string>`

You will need to import `promise` from `selenium-webdriver`. Once again, you can check the step-2-CLI-1.5.2 branch if this is unclear.

You might think that disabling all those type checks would make this more work than it is worth. You'll see that this is the only time you need to go to this much work.

When you go to run the unit test, you'll see that it will run them and then the browser window will stay up. If you want it to close once it has run the tests, you can change the test script to:

```
"test": "ng test --single-run",
```

The code for this section is in the branch named "step-2-CLI-1.5.2"

3 - Adding CSS

So that we can make our sample app look respectable, we are going to add Bootstrap 3 into the mix.

The first thing we will want to do is to install the bootstrap and font-awesome packages.

```
npm install --save-dev bootstrap@3
  font-awesome
```

And then include them in the styles section of `.angular-cli.json`.

```
"styles": [
"../node_modules/bootstrap/dist/css/bootst
rap.css",
"../node_modules/bootstrap/dist/css/bootst
rap-theme.css",
```

```
"../node_modules/font-awesome/css/font-
awesome.css"
],
```

Don't overwrite the `styles.css` file that is already in this section which should appear last!

The code for this section is in branch "step-3-CLI-1.5.2"

4 - Support for NgRX

Next, we want to add support for NgRX.

Run

```
npm install --save @ngrx/store
  @ngrx/effects
```

to install the @ngrx packages.

This code is in branch step-4-CLI-1.5.2

5 - Routes

Next, we will add a couple routes. According to the Angular Style Guide, our directory structure should be relatively flat and organized by feature. So, each route would exist right under the app directory. But there are two other directories that exist right under the app directory. `core` and `shared`. The way I prefer to structure my applications is to create a `routes` directory right under the app directory and then I place each of my routes under that directory.

I also place sub-routes right under the routes directory instead of under the route it is a sub-route of as you might be tempted to do. The reason for this is that I like to use sub directories under

each of my routes for the dumb components that will support that route. Placing sub-route directories in that same location makes the differentiation between routes and dumb components more difficult to see.

Finally, while it is possible to use routing without Lazy-Loading, I always use Lazy-Loading. There are multiple reasons for this. One of the main ones is that by implementing routing the same way every time, everyone on my team understands how routing is implemented, even if they don't fully understand why. But there are other reasons. First, if you use Lazy-Loading, you'll be forced to create Angular Modules at a finer grain of detail, rather than stuffing everything into `AppModule`. This is because the compiler tends to complain if you haven't imported modules that the Lazy-Loaded module requires. Second, by using Lazy-Loading, the app will load faster.

For this app, we are going to create two routes. List and Edit.

First, create a `routes` directory under the `app` directory.

For each route, we will need a module, so in the terminal window, navigate into the `routes` directory and then execute these two ng commands:

```
ng g module list
ng g module edit
```

This should create two directories under routes with module TypeScript files inside.

Now, to create the components that will be the smart components for our routes. Stay in the `routes` directory and run these two ng commands.

```
ng g component list
ng g component edit
```

You will notice that the response you get back from each creates all the component files and then registers the component with the list or edit module.

Before we forget, in the terminal window, navigate back up to the root directory for your application.

Next, we want to register these components and modules with Angular as routes that should be lazy loaded.

Open `app-routing.module.ts` and fill in the `Routes` array so that it looks like this:

```
const routes: Routes = [{
  path: '',
  redirectTo: 'list',
  pathMatch: 'full'
}, {
  path: 'list',
  loadChildren:
    './routes/list/list.module#ListModule'
}, {
  path: 'edit',
  loadChildren:
    './routes/edit/edit.module#EditModule'
}];
```

You should remember from our previous discussion on Lazy-Loading that we also need to register the child routes with each of the child modules.

Open `edit.module.ts` and add this `RouterModule.forChild()` to the imports section:

```
RouterModule.forChild([{
  path: '',
  pathMatch: 'full',
  component: EditComponent
}])
```

Don't forget to import `RouterModule`. Your editor should be able to assist you with this step.

Similarly, open `list.module.ts` and add a similar `RouterModule.forChild()` to the imports section:

```
RouterModule.forChild([{
  path: '',
  pathMatch: 'full',
  component: ListComponent
}])
```

If you run the application now, you'll still see the original "Welcome to App" page with "List Works" at the bottom. Not quite what we had in mind. Let's go back into `app.component.html` and take out everything that isn't:

```
<router-outlet></router-outlet>
```

If you run the application now, you should see "List Works!" when you load the app in your browser. The URL should point to `http://localhost:4200/list`. If you change this to `http://localhost:4200/edit`, you should see "Edit Works!"

Note: you may need to break out of the previous build and restart `npm start` to see the changes. The file watcher and rebuild process isn't entirely reliable.

If you run `npm run lint` on this code, you will see that we have two issues. It is the same issue in both components we just added. `ngOnInit()` doesn't specify a return type. Change them both to return void.

```
ngOnInit(): void {
}
```

One other problem you will see is that some of the unit tests will fail. This is because the template for the `AppComponent` has changed. Remember I said that we don't really want to test our HTML, just functionality? Now, you know why. Open `app.component.spec.ts` and remove all the test except for the one that says "should create the app." Once you've fixed this, your tests should succeed.

This code is in branch "step-5-CLI-1.5.2"

6 - Wait Component

Now it is time to build our first component. This component will display a wait spinner over the application when it is visible. We will make it visible whenever a long running process is occurring. For example, when we are retrieving data.

Because this is a component that isn't related to a specific route, we will place it under our shared directory. Since we have not created that yet, the first command we want to run is:

```
ng g module shared
```

Since we are creating this directory right under the `app` directory, you can run this at the root of your application and the Angular CLI will figure out where it needs to go.

Next, we want to create our Wait component. For this, you'll need to navigate into the shared directory in your terminal window and then run:

```
ng g component wait
```

This will register wait with `SharedModule` in the `declarations` section. But that won't make it available to our entire application. To make it available to the module that imports it, you also need to export it. So, open `shared.module.ts`, add an exports section and export WaitComponent.

```
  exports: [WaitComponent]
```

You will also need to import the shared module into `app.module.ts` by adding it to the imports section of that module.

```
imports: [
  BrowserModule,
  AppRoutingModule,
  SharedModule
],
```

Now, like all components, it should display "Wait Works!" if it is working, so let's add it to our `app.component.html` file right above the router-outlet.

```
<app-wait></app-wait>
<router-outlet></router-outlet>
```

Rebuild the application and load it in your browser. You should see “Wait Works!” and then right under it “List Works!”.

Next, we need to make the `AppComponent` allow for positioning of its child components. To do this, we will adjust the app.component.css

```
:host {
    position: absolute;
    height: 100%;
    width: 100%;
    display: block;
}
```

`:host` is a special selector that allows us to control the CSS of the tag we are using for our control. In this case, it is the same as if we had added this same definition in our `styles.css` file that defined `app-root`. Many people new to Angular create a container element in the template and style that instead. My issue with that method is that you end up creating more html than you need to.

Now, we can add the HTML for our Wait component template that has some meaning.

```
<div class="overlay"></div>
<div class="fa fa-spinner fa-spin"></div>
```

And put this code in the `wait.component.css` file

```
.overlay {
    position: absolute;
```

```
    width: 100%;
    height: 100%;
    background-color: gray;
    z-index: 1;
    opacity: 0.5;
}

.fa-spin {
    color: black;
    z-index: 2;
    font-size: 50px;
    left: 50%;
    top: 50%;
    position: absolute;
    opacity: 1;
}
```

And when you compile and run the application, you get a spinner that shows up over the list or edit routes.

Now, having a spinner that shows up all the time is of no use to us. We want to control it. And the way we will control it is by using NgRX.

Since we can register reducers and effects at the feature level, and Wait is kind of a feature, we are going to create a root with nothing in it and rely on the features to supply what they need.

So, open `app.module.ts` and add these two lines to the top of your imports section.

```
StoreModule.forRoot({}),
EffectsModule.forRoot([]),
```

The reason we put these first is so that any modules we import after this which use `forFeature()` will have the `forRoot()` stuff already defined.

Next, we will add the action, reducer and effects files that we developed previously. We will put this under the `shared/wait` directory

```
// wait.actions.ts

import {Action} from '@ngrx/store';
// tslint:disable:typedef
export const START = 'Wait.Start';
export class Start implements Action {
    readonly type = START;
}

export const END = 'Wait.End';
export class End implements Action {
    readonly type = END;
}

export const ADD = 'Wait.Add';
export class Add implements Action {
    readonly type = ADD;
    constructor(public payload: number) {}
}
```

```
//wait.reducer.ts
import {ActionReducer} from '@ngrx/store';
import * as Wait from './wait.actions';

// This could go in wait.actions.ts
export type Action =
    Wait.Add | Wait.End | Wait.Start;

export function waitReducer(
  // tslint:disable-next-line:typedef
  state = 0, action: Action): number {
  switch(action.type) {
    case Wait.ADD:
        return state + action.payload;
    default:
      return state;
  }
};

export const WaitReducer:
    ActionReducer<number> = waitReducer;
```

```
// wait.effects.ts
import { Observable } from
    'rxjs/Observable';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';
import 'rxjs/add/observable/timer';
import { Actions, Effect } from
    '@ngrx/effects';
import { Injectable } from
```

```
'@angular/core';
import * as Wait from './wait.actions';

@Injectable()
export class WaitEffects {
    @Effect()
    start$: Observable<Wait.Add> =
        this.actions$
            .ofType(Wait.START)
            .switchMap(
                (action: Wait.Add) =>

Observable.timer(1).take(1)
            )
            .map((): Wait.Add =>
                ({ type: Wait.ADD,
                   payload: 1 }));

    @Effect()
    end$: Observable<Wait.Add> =
        this.actions$
            .ofType(Wait.END)
            .switchMap(
                (action: Wait.End) =>

Observable.timer(1).take(1)
            )
            .map((): Wait.Add =>
                ({ type: Wait.ADD,
                   payload: -1 }));
    constructor(private actions$:
Actions){
```

```
    }
}
```

Next, we'll need to create a Shared state interface. Right now, all it will have in it is the wait. Put `shared-state.ts` directly under the shared directory:

```
export interface SharedState {
    wait: number;
}
```

Then update your `shared.module.ts` file to look like this:

```
const reducers:
  ActionReducerMap<SharedState> = {
    wait: WaitReducer
}

@NgModule({
  imports: [
    CommonModule,
    StoreModule.forFeature('shared',
      reducers),

EffectsModule.forFeature([WaitEffects])
  ],
  declarations: [WaitComponent],
  exports: [WaitComponent]
})

export class SharedModule { }
```

I've left out the import statements for brevity.

And now, we need an `AppState` that will help us retrieve the `WaitState`. Put `app-state.ts` directly under the app directory.

```
import { SharedState }
    from './shared/shared-state';

export interface AppState {
    shared: SharedState
}
```

I know, that seems like a lot of setup. The more you do this, the easier it gets. It takes longer to describe than it does to do the work.

Next, we want to have our Wait component listen to our wait state and display every time the counter is greater than zero.

In the `app.component.ts` file, we will select the wait state and assign it to an observable variable.

```
export class AppComponent {
  wait: Observable<number>;
  constructor(store: Store<AppState>) {
    this.wait = store.select((x: AppState)
      => x.shared.wait);
  }
}
```

Then, in our `app.component.html` template file, we add an `*ngIf` to our `app-wait` element.

```
<app-wait *ngIf="(wait | async) > 0">
</app-wait>
```

And now, just to prove that it works as expected, let's add a button to our List component that dispatches a Start when the button is clicked, waits for 5 seconds, and then dispatches an end.

First, add the button.

```
<p>list works
  <button (click)="start()">Start</button>
</p>
```

Inject the Store into the ListComponent:

```
constructor(private store:
Store<AppState>)
{
}
```

Then add the `start()` method in the `ListComponent` class.

```
start(): void {
  this.store.dispatch(new Wait.Start());
  Observable.timer(4000)
    .take(1)
    .subscribe((): void =>
      this.store.dispatch(new
Wait.End()));
}
```

If you click the button, the wait component will display for four seconds and then it will be removed.

Run lint to clean up any issues you may have.

There is one last thing we are going to need to fix up before we can commit our code. Our `AppComponent` test fails because it doesn't know about `app-wait`, the `StoreModule` or the `EffectsModule`.

Also, the `ListComponent` test doesn't know about the `StoreModule` or the `EffectsModule`.

So, add this code to the imports section of the `app.component.spec.ts` file:

```
StoreModule.forRoot({}),
EffectsModule.forRoot([]),
RouterTestingModule,
SharedModule
```

And add an imports section to the `TestBed.configureTestingModule()` in `list.component.spec.ts`

```
imports: [
  StoreModule.forRoot({}),
  EffectsModule.forRoot([]),
  SharedModule
]
```

This should fix your tests so they will run.

The code so far is in the branch "step-6-CLI-1.5.2"

7 - Dealing with Errors

Errors. Exceptions. Whatever you want to call them. They happen. The problem is, most of the time, we never see them.

Has this scenario ever happened to you? Someone is using, or testing the application you wrote. They report some "bug" but when you go to investigate the problem, you can't because when you isolate for just that bug, everything works as expected. If you investigate further, you find out that some exception got thrown along the way that caused JavaScript to stop working. The "bug" that got reported was caused by a bug that happened several steps prior to when the "bug" occurred. The original issue did get reported… on the console window in Developer Tools but no one saw it because no one was looking at the console window.

Now, what if every error that occurred was displayed on the screen? That's what we are going to achieve here.

There is a class in `@angular/core` that we can inherit from and provide to our application. Whenever an error occurs in Angular, the `handleError()` method on this code will get called passing in the error that occurred.

All we need to do is to inherit from this class and provide it with special syntax that tells Angular to use our code instead of the original class.

This code belongs in our `shared` directory.

Inside of the `shared` directory, add a directory named `errors`. Under the `errors` directory add a file named `error-handler.ts` and place the following code in it:

```
import { AppState } from '../../app-
state';
import { Store } from '@ngrx/store';
import { ErrorHandler as _ErrorHandler }
```

```
    from '@angular/core';

export class ErrorHandler
    extends _ErrorHandler {
    constructor(private store:
        Store<AppState>) {
            super();
    }
    // tslint:disable-next-line:no-any
    handleError(error: any): void
    {
        super.handleError(error);
    }
}
```

A couple of things to notice. Right now, we are injecting Store but not using it. We will. We need to write some more code first.

Second, we are extending from the original error handler provided by Angular. This is because we want to take advantage of some of the features the original class provides. Because we are extending, right now we are just using this class as a pass-through to the original class. This will change as we continue.

Third, we have to pass error as type `any` because the `super.handleError()` is expecting a type any. So, we've disabled our no-any type check on our `handleError()` definition.

It is probably obvious at this point that we will be using NgRX to assist us with our Error handling. The next code fragments will add the supporting NgRX code in our `errors` directory.

Every time there is an error, we will add the error to our errors store. Our component will display a dialog showing every error in the errors store and will have an "OK" button that will clear the errors from the store.

While our component will mask the screen so you can't do anything other than close the dialog, there are times when multiple errors occur while the dialog is displaying. This dialog will be smart enough to continue displaying errors.

The first thing we'll want to do is that we'll want to create the Actions that we'll need.

Create an `errors.actions.ts` file in the `errors` directory and give it the following content.

```
import {Action} from '@ngrx/store';

// tslint:disable:typedef
export const ADD = 'Errors.Add';
export class Add implements Action {
    readonly type = ADD;
    constructor(public message: string) {
    }
}

export const CLEAR = 'Errors.Clear';
export class Clear implements Action {
    readonly type = CLEAR;
}
```

Next, we code the reducer. Add the file `errors.reducer.ts` in your errors directory with the following content.

```
import {ActionReducer} from '@ngrx/store';
import * as Errors from
'./errors.actions';

export type Action =
    Errors.Add | Errors.Clear;

export function errorsReducer(
  // tslint:disable-next-line:typedef
  state: ReadonlyArray<string> = [],
    action: Action): ReadonlyArray<string>
  {
  switch(action.type) {
    case Errors.ADD:
        return [...state, action.message];
    case Errors.CLEAR:
        return [];
    default:
        return state;
  }
};

export const ErrorsReducer:
    ActionReducer<ReadonlyArray<string>> =
        errorsReducer;
```

There may be some code here that you've never seen before. First, you'll see that we are using `ReadonlyArray` here. This is just like an array, but it makes sure that we never change the content of the array. Instead, we must create a new array and copy the old content into it. You'll remember from our

discussion about NgRX and Functional programming that this is a requirement. If you skip this step, you are likely to end up with code that doesn't function as you expect it to.

The next bit of unfamiliar code you will see is the `[...state, action.message]` code. The three dots are called the "spread operator." It takes the current content of state and puts the same items as the first items in the new array. The last item is the message we are adding. It does the same thing as `state.push(action.message)` except for the fact that it creates a new array in the process to fulfill our requirement that the return value be a new object.

Now that we have the reducer defined, we can register it with our shared module. First, open `shared-state.ts` and add this definition to the interface:

```
errors: ReadonlyArray<string>;
```

Next, open `shared.module.ts` and add this to the reducers constant:

```
errors: ErrorsReducer
```

Now we need to define the Errors component. In the terminal window, navigate into the shared folder and type

```
ng g component errors
```

Once again, we need to make `ErrorsComponent` available to the module that imports `SharedModule`, so we need to export it. Add ErrorsComponent to the exports of SharedModule.

```
exports: [WaitComponent, ErrorsComponent]
```

Now that we have a component, let's add it to our app.component.html file. For now, we just want to see that it displays the default content.

```
<app-wait *ngIf="(wait | async) > 0">
</app-wait>
<app-errors></app-errors>
<router-outlet></router-outlet>
```

Compile and run to verify that the app-errors component displays.

Now, let's turn our `app-errors` into a dialog using Bootstrap.

Open `errors.component.html` and add the following code:

```
<div class="overlay"></div>
<div class="modal" tabindex="-1"
    role="dialog">
  <div class="modal-dialog"
    role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h4 class="modal-
title">Errors</h4>
      </div>
      <div class="modal-body">
        <p>Error messages go here</p>
      </div>
      <div class="modal-footer">
        <button type="button"
        class="btn btn-
```

```
default">OK</button>
      </div>
    </div>
  </div>
</div>
```

And put the following code in errors.component.css

```
.overlay {
    position: absolute;
    width: 100%;
    height: 100%;
    background-color: gray;
    z-index: 1;
    opacity: 0.5;
}

.modal {
    display: block;
    top: 25%;
}
```

You'll notice that we didn't put in a `:host` definition because we don't want the container to have any impact on this control.

Now, if you run the application, an Error dialog will display.

Next, we need to pass in an array of strings that we will display where we currently display "Error messages go here."

To do this, we need to specify an `@Input()` field in our `ErrorComponent` class.

Open up `error.component.ts` and add a field named "data" with a decorator `@Input()`

```
import { Component, Input, OnInit }
   from '@angular/core';

@Component({
  selector: 'app-errors',
  templateUrl: './errors.component.html',
  styleUrls: ['./errors.component.css']
})
export class ErrorsComponent
    implements OnInit {

  @Input() data: ReadonlyArray<string> =
    [];
  constructor() { }

  ngOnInit(): void {
  }

}
```

Now, back to errors.component.html, find this line:

```
<p>Error messages go here</p>
```

And change it to

```
<p *ngFor="let item of data">{{item}}</p>
```

You'll see, if you give the data array a list of strings, they will show up in the dialog when you run the code.

Next, we need to select out the errors from our store in `app.component.ts` similar to how we selected out wait.

```
export class AppComponent {
  wait: Observable<number>;
```

```
  errors:
    Observable<ReadonlyArray<string>>;

  constructor(store: Store<AppState>) {
    this.wait = store.select(
      (x: AppState) => x.shared.wait);
    this.errors = store.select(
      (x: AppState) => x.shared.errors);
  }
}
```

And now we can 1) display the dialog only when there are errors and 2) pass the errors into the Errors component so they can be displayed.

```
<app-errors
  *ngIf="(errors | async)?.length > 0"
  [data]="(errors | async)">
</app-errors>
```

Now, in the `ListComponent` class, change the `start()` method so that it dispatches a message to the error instead of dispatching to Wait.

Make sure your import uses the Add method from the Errors actions and not the Wait actions.

```
import * as Errors from
  '../../shared/errors/errors.actions';
…
  start(): void {
    this.store.dispatch(
```

```
        new Errors.Add('Here is an error'));
    }
```

And in the error.component.html, add a click handler to the OK button.

```
<button (click)="clear()" type="button"
    class="btn btn-default">OK</button>
```

And a corresponding method to the `ErrorsComponent` class that dispatches the Clear action.

```
export class ErrorsComponent
  implements OnInit {
  @Input() data: ReadonlyArray<string> =
    [];

  constructor(
    private store: Store<AppState>) { }

  ngOnInit(): void {
  }

  clear(): void {
    this.store.dispatch(
        new Errors.Clear()
    );
  }
}
```

Now, you should be able to build the application and run it. If you click the “Start” button, the dialog will display with the

message in it. If you click the OK button in the dialog, the dialog should close.

Now, back to the Error handler code.

As you may remember, the code we have so far just passes through to the default handler. We want this to log errors to our Errors entity. This is simply a matter of dispatching the error.

```
// tslint:disable-next-line:no-any
handleError(error: any): void {
    super.handleError(error);
    this.store.dispatch(
        new Errors.Add(error.message)
    )
}
```

Then, we need to provide this in our `app.module.ts` file.

```
import {
  ErrorHandler as NgErrorHandler,
  NgModule } from '@angular/core';
import { ErrorHandler }
  from './shared/errors/error-handler';

// other existing code here

  providers: [
    {
      provide: NgErrorHandler,
      useClass: ErrorHandler
```

```
    }
  ],
```

The last thing we need to do is to fixup one of the tests with a missing provider. The `errors.component.spect.ts` file needs a provider for store. The temptation would be to add that in the provides section, but the `StoreModule` is what loads Store for us when we call `StoreModule.forRoot()`.

Since we aren't going to run any code that needs to access the Errors entity in our store, adding this line in our `TestBed.configureTestingModule()` method will be fine:

```
imports: [StoreModule.forRoot({})],
```

This code is in branch "step-7-CLI-1.5.2"

8 - Styling the App

You'll remember that we added some Bootstrap CSS to our project, but we haven't used it much yet.

Since we are about to get to the part of this tutorial where we are going to display some data, let's add in some basic Bootstrap styles now.

Change `app.component.html` to look like this:

```
<app-wait *ngIf="(wait | async) > 0">
</app-wait>
<app-errors
  *ngIf="(errors | async)?.length > 0"
  [data]="(errors | async)">
</app-errors>
```

```
<nav class="navbar navbar-default">
  <div class="container">
    <div class="navbar-header">
      <div class="navbar-brand">
        Angular Tutorial App
      </div>
    </div>
  </div>
</nav>
<div class="container">
  <router-outlet></router-outlet>
</div>
```

And, for now, change list.component.html to look like this again:

```
<p>
  list works!
</p>
```

And while we are at it, let's remove the `start()` method from the `ListComponent` class. This means you'll also need to remove the `Store` injection from the constructor.

Run lint to clean everything up.

This code is in branch "step-8-CLI-1.5.2"

9 - List Route

I know that's a lot of foundational code we just wrote. The good thing is you can use what we've done so far as a seed for any future projects.

Now, let's focus on the list route. What we want to do here is to display a list of contacts. We are going to keep this simple. We want to display a list of contacts with the following fields:

- ID
- FirstName
- LastName
- DateOfBirth

For now, we'll put our list of contacts in our `ListComponent` class. Later we'll simulate getting it from the server via NgRX.

The first thing we will want to do is to create an interface that represents the contacts. Navigate to the list directory and execute the following CLI command

```
ng g interface contact
```

This should place a `contact.ts` file in your list directory. Open this file and fill it out with the following definition:

```
export interface Contact {
    id: number;
    firstName: string;
    lastName: string;
    dateOfBirth: Date
}
```

And now, in your `list.component.ts` file, add a member variable named "contacts" that has a type of `ReadonlyArray<Contact>` and assign it a list of contact records.

```
export class ListComponent
  implements OnInit {
```

```
  contacts: ReadonlyArray<Contact> = [
    {
      id: 1,
      firstName: 'Dave',
      lastName: 'Bush',
      dateOfBirth: new Date(2000, 0, 15)
    },
    {
      id: 2,
      firstName: 'John',
      lastName: 'Dough',
      dateOfBirth: new Date(1990, 5, 15)
    }
  ];
  constructor() { }

  ngOnInit(): void {
  }
}
```

Now, to display this we need to go into our template and add some HTML. Remove the HTML that is already there and add this in its place.

```
<table class="table table-hover">
  <thead>
  <tr>
      <td>First Name</td>
      <td>Last Name</td>
      <td>DOB</td>
  </tr>
  </thead>
  <tbody>
```

```
  <tr *ngFor="let contact of contacts">
      <td>{{contact.firstName}}</td>
      <td>{{contact.lastName}}</td>
      <td>
 {{contact.dateOfBirth |
date:'shortDate'}}
      </td>
  </tr>
  </tbody>
</table>
```

This is just a basic table styled using Bootstrap. The key part of the code is the `*ngFor` code which we've described previously.

This code is in branch "step-9-CLI-1.5.2"

10 - List NgRX

Next, we will use NgRX and a Service to retrieve our data and get it to the view where we can display it.

For our NgRX code, we will require two Actions. `List` and `ListResult`. This is because we are going to ask for the List from the server via the `Effect` using the `List` action and once we have the list, we are going to send it to the Reducer with the `ListResult` Action.

The first thing we need to do is create a `list.actions.ts` file in our route/list directory and make it look like this:

```
import { Contact } from './contact';
import { Action } from '@ngrx/store';

// tslint:disable:typedef
export const LIST = 'List.List';
```

```
export class List implements Action {
    readonly type = LIST;
}

export const LIST_RESULT = 'List.Result';
export class ListResult implements Action
{
    readonly type = LIST_RESULT;
    constructor(public rows:
        ReadonlyArray<Contact>) {}
}
```

By now, you should be familiar with NgRX code. Nothing fancy here.

Next, we will need a Reducer that will handle our `ListResult` Action.

```
// list.reducer.ts
import { ActionReducer }
  from '@ngrx/store';
import { Contact } from './contact';
import * as List from './list.actions';

export function listReducer(
    state: ReadonlyArray<Contact> = [],
    action: List.ListResult):
        ReadonlyArray<Contact> {
        switch(action.type) {
            case List.LIST_RESULT:
                return action.rows;
            default:
                return state;
```

```
        }
    }

export const ListReducer:
   ActionReducer<ReadonlyArray<Contact>> =
     listReducer;
```

Notice how I typed the action parameter as `List.ListResult` to get access to `action.rows`.

Now, we will create an Effect to retrieve our list of contacts.

```
// list.effects.ts
import { Observable } from 'rxjs/Rx';
import { Actions, Effect }
  from '@ngrx/effects';
import { Injectable } from
'@angular/core';
import * as List from './list.actions';

Injectable()
export class ListEffects {
    @Effect()
    list$: Observable<List.ListResult> =
        this.actions$
        .ofType(List.LIST)
        .map(():List.ListResult =>
          new List.ListResult(
            [
                {
                    id: 1,
                    firstName: 'Dave',
                    lastName: 'Bush',
                    dateOfBirth:
```

```
                        new Date(2000, 0,
15)
                },
                {
                    id: 2,
                    firstName: 'John',
                    lastName: 'Dough',
                    dateOfBirth:
                      new Date(1990, 5,
15)
                }
            ]
        )
    );

    constructor(private actions$: Actions)
      {}
}
```

Again, you should notice that we are taking advantage of typing everything to ensure that we are returning what we intend. We are still working with fake data, but this is the basic pattern for using NgRX to get data into the view. We will put this in a service in the next step, but for now, let's get what we have so far wired into our list module and list component.

You see, none of the NgRX code we just wrote will be seen by any of our code if we do not register it with our list module. To do this, first we will need to create a `ListState` interface. In your terminal window, navigate into the list directory. Then execute the following CLI command:

```
ng g interface list-state
```

Inside the resulting `ListState` file, place the following content:

```
import { Contact } from './contact';

export interface ListState {

    list: ReadonlyArray<Contact>

}
```

Next, in `app-state.ts`, update the interface to have a list field.

```
export interface AppState {
    shared: SharedState;
    list: ListState;
}
```

Next, in `list.module.ts`, update the file to have a reducers constant that has a list element and imports the `forFeature()` for `StoreModule` and `EffectsModule`

```
import { EffectsModule }
  from '@ngrx/effects';
import { ActionReducerMap, StoreModule }
  from '@ngrx/store';
import { ListEffects }
  from './list.effects';
import { ListReducer }
  from './list.reducer';
import { ListState } from './list-state';
import { RouterModule }
  from '@angular/router';
```

```
import { NgModule } from '@angular/core';
import { CommonModule }
  from '@angular/common';
import { ListComponent }
  from './list.component';

const reducers:
ActionReducerMap<ListState>
  = {
    list: ListReducer
}

@NgModule({
  imports: [
    StoreModule.forFeature(
      'list', reducers),
    EffectsModule.forFeature(
        [ListEffects]),
    CommonModule,
    RouterModule.forChild([{
        path: '',
        pathMatch: 'full',
        component: ListComponent
      }])
  ],
  declarations: [ListComponent]
})
export class ListModule { }
```

With that code written, we can now select our entity out of the store and listen to it. To do this, go to `list.component.ts`.

Remove the contacts code we added before and re-inject the store.

Next add a contacts member variable that is typed as `Observable<ReadonlyArray<Contact>>` and in the contructor, select out the list and assign it to the new contacts member.

```
import { Store } from '@ngrx/store';
import { AppState } from '../../app-
state';
import { Observable } from
    'rxjs/Observable';
import 'rxjs/add/operator/map';
import { Contact } from './contact';
import { Component, OnInit }
  from '@angular/core';

@Component({
  selector: 'app-list',
  templateUrl: './list.component.html',
  styleUrls: ['./list.component.css']
})
export class ListComponent
  implements OnInit {
    contacts:
      Observable<ReadonlyArray<Contact>>;
    constructor(
        private store: Store<AppState>) {
      this.contacts = store.select(
        (x: AppState) => x.list.list);
  }
```

```
  ngOnInit(): void {
  }
}
```

Now, in the `list.component.html` template file, change your `*ngIf` to use the new contacts variable.

```
<tr *ngFor="let contact of (contacts |
   async)">
```

(All on one line).

The `async` pipe subscribes to contacts and pulls the data out of it for us.

The final step is to tell NgRX to retrieve the data when we load the route. To do this, we add a dispatch in our `ngOnInit()` method.

```
import * as List from './list.actions';

…
  ngOnInit(): void {
    this.store.dispatch(new List.List());
  }
```

You may wonder why we don't dispatch in the constructor instead. The constructor is only called the first time a component is created. Once for each instance. But once it has been created, the constructor will never be called again. This is why `ngOnInit()` exist. Anything you want to run every time the component is rendered goes into `ngOnInit()`. The only code that should go in your constructor would be code that only needs to ever run once. Therefore, we put selects in the constructor. Once we've selected an entity out of the Store, we

can subscribe and unsubscribe to it directly, or indirectly via `async`, as the component is rendered and taken off the screen, which Angular refers to as "destroy."

`npm start` to make sure it runs.

Next we need to add the list NgRX stuff to our `list.component.spec.ts`. The easiest thing to do is add a `StoreModule.forFeature()` after the `StoreModule.forRoot()`

```
imports: [
  StoreModule.forRoot({}),
  StoreModule.forFeature('list',
    {list: ListReducer}),
  EffectsModule.forRoot([]),
  SharedModule
]
```

Then lint and test to make sure you can commit.

This code can be found in branch "step-10-CLI-1.5.2"

11 - Contacts Service

OK. So now we add a service to retrieve our data. Once again, navigate to the list directory and run this CLI command:

```
ng g service contacts
```

I was tempted to call it a List service, but it is really a service to get contacts that will have a `list()` method to get a list of contacts for me.

Because we want to be able to access `HttpClient`, make sure the `HttpClientModule` is imported in `app.module.ts`

and then inject `HttpClient` in the constructor of `ContactsService`.

```
constructor(private httpClient:
HttpClient)
 { }
```

Now, create a list method in it that returns an Observable of Arrays of Contacts or Observable of an empty object.

```
list():
  Observable<{} | ReadonlyArray<Contact>>
{
    return this
        .httpClient
        .get<ReadonlyArray<Contact>>
            ('/api/contacts/list')
      .retry(2)
      .catch((e: Error) =>
            /* handle errors here */);

}
```

You will also need to import '`rxjs/add/operator/retry`' and '`rxjs/add/operator/catch`' for this code to compile.

Just a warning here. When you retrieve the data from the server, everything will be a string, so you'll need to convert the date string that comes back to a date object. It will be up to you as to how you want to store the date on the server which means all I can tell you about how to convert it is you'll want to add a `map()` after the `catch()` that takes the array of contacts and iterates through them with an inner `map()`.

Now, since we don't have a server, we are going to comment out this code and replace it with a fake that does the same thing.

```
return Observable.from([[
  {
      id: 1,
      firstName: 'Dave',
      lastName: 'Bush',
      dateOfBirth: new Date(2000, 0, 15)
  },
  {
      id: 2,
      firstName: 'John',
      lastName: 'Dough',
      dateOfBirth: new Date(1990, 5, 15)
  }
]]);
```

You'll need to import '`rxjs/add/observable/from`' to get the `Observable.from()` to work. You might be recognizing a pattern here. If the method is a static method, you import from `rxjs/add/observable/methodName`. If it is an instance method, you import from `rxjs/add/operator/methodName.`

In the code above, you can see it is the same array we've been using all along. Notice too that we have an array of records inside an array. This is not a mistake. The way `Observable.from()` works, it takes an array and each element in the array represents an observable item. Since we want the two records to come back as one observable item, you pass the array of records in the observable array.

Next, we want to replace the array in `ListEffects` with a call to this service. Open the `list.effects.ts` file and start by injecting ContactsService in the constructor.

```
constructor(private actions$: Actions,
  private contactsService:
ContactsService)
  {}
```

Then change the list$ effect to look like this:

```
@Effect()
list$: Observable<List.ListResult> =
    this.actions$
    .ofType(List.LIST)
    .switchMap(():
 Observable<{} | ReadonlyArray<Contact>>
=>
 this.contactsService.list())
    .map((x: ReadonlyArray<Contact>):
    List.ListResult =>
   new List.ListResult(x)
);
```

And then, provide the service in the `ListModule`.

```
  providers: [ContactsService],
```

You'll also no longer need the httpClient injection in the service so you should remove that.

Here is the key that makes this work. You use `switchMap()` when you want another observable to take the place of the observable that started the chain. So, `switchMap()` returns the contents of the observable that is returned inside of it.

Another way of saying this is that the `switchMap()` subscribes to the Observable, the contents are pulled out and sent to the `map()` that comes right after it.

The last thing we need to do before we move on is to fix the `contacts.service.spec.ts` file so it will run our tests again.

Add an imports section to the `TestBed.configureTestingModule()` that imports `HttpClientModule`.

This code is available in branch "step-11-CLI-1.5.2"

12 - Add/Edit/Delete Navigation

In this section we are going to add buttons that will let us Edit and Delete existing records as well as Add new records. When we are done, we will be able to click the buttons and the Edit/Add will take us to the Edit route. Delete will, ultimately call the delete method on our Contacts Service.

Start by opening `list.component.html` and add a column to the table. The heading won't have anything in it, but the rows will have an Edit and Delete button for each record.

While we are at it, we will attach an Edit and Delete method to each click event and pass it the ID for the row.

```
<table class="table table-hover">
  <thead>
  <tr>
    <td>First Name</td>
    <td>Last Name</td>
```

```
    <td>DOB</td>
    <td> </td>
  </tr>
  </thead>
  <tbody>
<!-- all one line -->
  <tr *ngFor="let contact of (contacts |
     async)">
    <td>{{contact.firstName}}</td>
    <td>{{contact.lastName}}</td>
<!-- all one line -->
    <td>{{contact.dateOfBirth |
         date:'shortDate'}}</td>
    <td>
      <button (click)="edit(contact.id)"
        class="btn btn-default">
        Edit</button>
      <button (click)="delete(contact.id)"
        class="btn btn-default">
        Delete</button>
    </td>
  </tr>
  </tbody>
</table>
```

Now, in the `ListComponent` class, provide the `delete()` and `edit()` methods:

```
delete(id: number): void {
}
```

```
edit(id: number): void {
}
```

That's all for now. Run the code and verify that the buttons show up.

Now, the delete button needs to fire a Delete action so we need to create one. By now, you should be able to create your own Actions without being told how. You can use a previous Action as a template if you need to. So, in the `list.actions.ts` file, add a Delete action that takes the ID as a parameter. Once you've got that in place, go ahead and dispatch the action from the delete method in the `ListComponent` class.

Next, we'll need to setup a listener in our `ListEffects` class, but before we do that, we should add a delete service to our `ContactsService` class. Open it now and add a delete method that looks like this:

```
delete(id: number): Observable<{}> {
  return this
    .httpClient
    .delete('/api/contacts/' + id);
}
```

Since we don't have a real back end, we are going to substitute this with a noop.

```
delete(id: number): Observable<object> {
  return Observable.of({});
}
```

Now we can add the Effect.

```
@Effect()
delete$: Observable<List.List> =
    this.actions$
    .ofType(List.DELETE)
    .switchMap(
      (action: List.Delete):
Observable<{}>
       => this
         .contactsService
         .delete(action.id)
    )
    .map((): List.List => new
List.List());
```

Notice what we are doing here. We are listening for a `Delete` Action, calling the `delete()` method on the Contacts Service and then returning a `List` Action. The `List` Action will get picked up by the `list$` Effect which will call the `list()` method on the `Contacts` service and will return the `ListResults` Action that the reducer will pick up and put in our store.

If we really could delete a record, the result will be that our record will be removed from the list.

The one mistake I made when I started learning NgRX was putting too much in an Effect. Each Effect should do a tiny slice of what needs to be done. There are times when you may not want your Effect to return anything, but these will be rare. If you don't want an Effect to return an Action, you need to make sure you tell it so by passing `{dispatch: false}` into `@Effect()`.

```
@Effect({dispatch: false}) …
```

Next, let's implement the `edit()` method in our `ListComponent` class.

Before we do, we need to update our edit route to recognize when we pass an id as a parameter. Navigate to your `AppRoutingModule` class and update the Routes to look like:

```
const routes: Routes = [
  {
    path: '',
    redirectTo: 'list',
    pathMatch: 'full'
  }, {
    path: 'list',
    loadChildren:

'./routes/list/list.module#ListModule'
  }, {
    path: 'edit/:id',
    loadChildren:

'./routes/edit/edit.module#EditModule'
  }, {
    path: 'add',
    loadChildren:

'./routes/edit/edit.module#EditModule'
  }
];
```

You'll notice that we've changed our edit route to allow us to pass a parameter (/:id). But we've also added an "add" route that

resolves to the same module. We'll use this new "add" route soon.

If you run the code now, you should be able to access the edit route using either

`http://localhost:4200/edit/1` (or any number)

Or by using

```
http://localhost:4200/add
```

Now we can go back to the `edit()` method in our `ListComponent` class. And put in the code to make the edit button take us to the edit route.

First, we will need to inject a Router into our `ListComponent` constructor.

```
constructor(private store:
Store<AppState>,
    private router: Router) {
    this.contacts = store.select(
        (x: AppState) => x.list.list);
}
```

Then all we need to do in our `edit()` method is make it look like this:

```
edit(id: number): void {
  this.router.navigate(['/edit', id]);
}
```

To round this all out, let's add an Add button to the `List` route. Open list.component.html and update the template by placing the following HTML fragment at the bottom of the file:

```
<button (click)="add()"
  class="btn btn-primary pull-right">
  Add</button>
```

And placing an `add()` method in the `ListComponent` class that redirects to the "add" route.

```
add(): void {
  this.router.navigate(['/add']);
}
```

To fix the tests, add a provider for the router in the `list.component.spec.ts` file:

```
imports: [
  StoreModule.forRoot({}),
  StoreModule.forFeature(
    'list', {list: ListReducer}),
  EffectsModule.forRoot([]),
  RouterModule.forRoot([]),
  SharedModule
]
```

Since we aren't testing routing, this should be sufficient to get the tests running.

You'll also need to set the `APP_BASE_HREF`. This is needed for routing and happens automatically in the browser. But the tests don't know anything about it so we need to set it to something. Add the following providers section in the TestBed configuration in `list.component.spec.ts`

```
providers: [
  {
    provide: APP_BASE_HREF,
```

```
    useValue: '/'
  }],
```

Since the import for `APP_BASE_HREF` tends to be a problem, I'll warn you to import it from `@angular/common`.

This code can be found in the branch "step-12-CLI-1.5.2"

One thing you should notice while looking at the `ListComponent` class. There isn't a lot of code. The methods we have only do one thing. So, when we test this code, all we really care about is that the component gets created as expected.

13 - Edit Route

Now it is time to focus on the Edit component. Since we will be using Reactive Forms, we need to import the `ReactiveFormsModule` into our Edit module.

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

```
})
export class EditModule { }
```

Then in our `EditComponent` class, we define our form in the constructor.

```
form: FormGroup;
constructor(
  private formBuilder: FormBuilder) {
    this.form = this.formBuilder.group({
      firstName: ['',
Validators.required],
      lastName: ['', Validators.required],
      dateOfBirth: ['',
          Validators.required]
    })
}
```

Remember our discussion about `constructors` vs `ngOnInit`? The `FormGroup` only needs to be defined once, so we place the code in our constructor.

Now we can add the HTML in our template to support the form.

You might want to just grab this from the branch and copy and paste it in.

```
<div class="row">
  <div class="col-lg-offeset-3 col-lg-6">
    <form class="form-horizontal"
      [formGroup]="form">
      <div class="form-group">
        <label class="col-lg-3"
          for="firstName">
          First Name:</label>
```

```
        <div class="col-lg-9">
          <input type="text"
            class="form-control"
            placeholder="first name"
            id="firstName"
            formControlName="firstName">
        </div>
      </div>
      <div class="form-group">
        <label class="col-lg-3"
          for="lastName">Last
Name:</label>
        <div class="col-lg-9">
          <input type="text"
            class="form-control"
            placeholder="last name"
            id="lastName"
            formControlName="lastName">
        </div>
      </div>
      <div class="form-group">
          <label class="col-lg-3"
            for="dateOfBirth">
            Birth Date:</label>
          <div class="col-lg-9">
            <input type="text"
              class="form-control"
              placeholder="birth date"
              id="dateOfBirth"

formControlName="dateOfBirth">
          </div>
```

```
        </div>
      </form>
    <div class="col-lg-12">
        <button
         class="btn btn-primary pull-
right"
        [disabled]="!form.valid">
         Save</button>
        <button
        class="btn btn-default pull-
right">
          Cancel</button>
    </div>
  </div>
</div>
```

If you build the application now and then navigate to the edit screen you should see the form fields and the "Save" button should be disabled until all the fields have data in them.

The one thing you will notice about this form is that it will accept anything for a date. Let's add date validation. The following is a very simple date validation method. You may want to embellish it. Add this right before the constructor in your `EditComponent` class because our lint rules expect static methods to come before instance methods.

```
static isDate(c: FormControl): object {
  if(!c.value.match(
    /^\d{1,2}\/\d{1,2}\/(\d{2}|\d{4})$/))
{
      return {invalidDate: true};
```

```
    }
}
```

Now change your `dateOfBirth` form definition to compose both this date validation and the required validation.

```
dateOfBirth: ['', Validators.compose(
  [ Validators.required,
    EditComponent.isDate]
)]
```

Compose allows us to attach multiple validations, in the order we want them to validate, to the control.

Before we continue, we need to fix up the `edit.component.spec.ts` file so the tests will run. The thing that is missing is that we need to import the `ReactiveFormsModule` in the `TestBed` configuration. If you've been following along, you should be able to add this on your own.

Using your knowledge from the section above "Displaying Errors". You can now display an error message for "required" and another for "invalid date".

The code for this section is in the branch "step-13-CLI-1.5.2"

14 – Edit State Management

Now that we have a basic functioning form, it is time to wire up the state management. The first thing we want to do is to setup a listener on the form so that when it changes, we know about it.

To keep everything strongly typed, we are going to need to add an interface that represents our form. Using the CLI, add a new

interface in the edit directory called `EditForm` and give it the following definition:

```
export interface EditForm {
    firstName: string;
    lastName: string;
    dateOfBirth: string;
}
```

In the `EditComponent` class, find the `ngOnInit()` method and add a listener on the form. For right now, we are just going to `console.log()` the contents of the form so you can see it change.

```
ngOnInit(): void {
    this.formSubscription =
      this.form.valueChanges.subscribe(
        (x: EditForm) =>
        console.log(JSON.stringify(x))
  )
}
```

Once again, we add the listener in ngOnInit() because we want this to occur every time the control is rendered.

Notice that we assigned the subscription to the subscription variable `formSubscription`. This is so we can unsubscribe from it when the component is removed from the DOM.

Make sure you include the `formSubscription` member variable at the top of the class and type is as `Subscription`.

Add an `OnDestroy` implements to the `EditComponent` class and add the `ngOnDestroy()` method to the class so that we can unsubscribe from the form listener.

```
export class EditComponent
  implements OnInit, OnDestroy {
...

  public ngOnDestroy(): void {
    this.formSubscription.unsubscribe();
  }
```

Now, run the code and open Developer Tools on your browser. You will see that every time we change one of the fields, the `console.log()` writes out the current values of each of the fields.

Next, we need our NgRX files for Edit. First, we will need an Update Action that takes an `EditForm` type as a parameter. Create that next.

Then, we will need a reducer that listens for the update action and updates the state to represent what was just passed in. Go ahead and write that now.

Now, we should be able to dispatch an Update Action where we currently have the `console.log()`. Inject a Store into the constructor of `EditComponent`.

```
  constructor(
    private formBuilder: FormBuilder,
    private store: Store<AppState>) {
```

And then add the dispatch:

```
ngOnInit(): void {
  this.formSubscription =
    this.form.valueChanges.subscribe(
      (x: EditForm) =>
       this.store.dispatch(
```

```
            new Edit.Update(x)))
}
```

The last thing we need to do for this to work correctly is that we need to create an `EditState` interface, register the reducer, and register edit with `AppState`.

This should be very similar to work we did in the List, so I'll leave you on your own for this step. If you get stuck, you can refer to the branch where this code is stored.

Now that we have state getting pushed into the Store, we now need to make sure that every time the edit state changes, the form gets updated.

The first step in making this work is to select the entity out of the Store in our constructor.

Add a member variable to EditComponent

```
  editEntity: Store<EditForm>;
```

Then assign it the selection in the constructor.

```
this.editEntity = this.store.select(
    (x: AppState) => x.edit.form);
```

Now, in the `ngOnInit()` method, subscribe to editEntity and whenever it changes, update the form.

```
this.editEntitySubscription =
  this.editEntity.subscribe(
    (x: EditForm) =>
```

```
    this.form.patchValue(
      x, {emitEvent: false}));
```

You'll notice we saved the subscription off to `editEntitySubscription` so that we can unsubscribe from it in `ngOnDestroy()`. Make sure you add it as a member variable, type it as `Subscription` and unsubscribe in `ngOnDestroy()`.

`patchValue()` allows us to send data into the form. The `emitEvent` thing is telling `patchValue()` to not fire a change event. Otherwise, you'll end up in an infinite loop. That is, the observer on our form will change the state via the reducer which will fire the listener on the state, which will call `patchValue()` which will fire the form listener. And on and on it goes.

To fix the test, you'll need to add an empty `StoreModule.forRoot()` to the imports section of `edit.component.spec.ts` and then add a `StoreModule.forFeature()` right after it to include the edit feature store.

```
const reducers:
  ActionReducerMap<EditState> = {
  form: EditReducer
}
...
    TestBed.configureTestingModule({
      imports: [ReactiveFormsModule,
      StoreModule.forRoot({}),
      StoreModule.forFeature(
```

```
            'edit', reducers)
    ],
```

This code is available in branch “step-14-CLI-1.5.2”

15 - Refactor ContactsService

We are going to continue to use the Contacts service that we originally created in our List route. Since we are going to use it in multiple locations, we need to move it to our shared directory. This is a normal part of development. Sometimes you don’t know you will need something to be shared until later in the development process.

So, the first thing you will need to do is that you’ll need to move `contacts.service.ts` and `contacts.service.spec.ts` to the shared directory. In VS Code, this is a simple drag and drop operation.

Next, we need to remove `ContactsService` from the providers section of the List module and add it to the providers section of the Shared module.

You should also move `contact.ts` since it is used by `ContactsService` and will also be shared between routes. It is just an interface so it doesn’t have any module fixes that are needed.

Now, you need to find all the other places where you are referencing `ContactsService` and update the import statement to load it from the `shared` directory instead of the `list` directory. You can either do this using VS Codes ability to search the directory or you can just use `npm start` to build the application and fix the compile errors.

Next, because we are going to want to reuse the list of contacts, let's move them out of the `ContactService.list()` method and make them a constant outside of the class.

After you've made the changes, run the application to make sure it works. Don't forget to run the tests and fix any issues that reveals as well.

The code for this step can be found on branch "step-15-CLI-1.5.2"

16 - ContactsService Pseudo Database

Up until now, we've been happy with our data being static. But we can do better. Now that we have our data available to all our ContactsService methods, let's use that as our database.

Change `const contacts: …` to `let contacts:` so that we can change the data it is pointing to.

Now, change the code in delete to filter out the item we are deleting and assign it back to the contacts variable.

```
delete(id: number): Observable<object> {
  contacts = contacts
    .filter((x: Contact) =>
        x.id !== id)
    .reduce(
      (
   previousContact:
ReadonlyArray<Contact>,
   currentContact: Contact ) =>
       [...previousContact,
currentContact]
    , []);
```

```
    return Observable.of({});
  }
```

Notice that we are assigning a new array back to contacts. This is on purpose. If you modify the existing array, your screen will not update correctly, especially once we add in change detection optimizations later.

While we are here, let's add in three other methods we are going to need soon. `get()`, `update()` and `add()`.

```
get(id: number):
  Observable<{} | ReadonlyArray<Contact>>
{
  return Observable.from(
    [[contacts.find((x: Contact) =>
      x.id === id)]])
}

update(contact: Contact):
Observable<number> {
    const c: Contact =
      contacts.find((x: Contact) =>
          x.id === contact.id);
    c.dateOfBirth = contact.dateOfBirth;
    c.firstName = contact.firstName;
    c.lastName = contact.lastName;
    contacts = [...contacts];
    return Observable.of(contact.id);
}


add(contact: Contact): Observable<number>
{
```

```
    const maxId: number =
      contacts.reduce(
        (max: number, c: Contact) =>
          {
            if(max < c.id) {
              return c.id;
      }
      return max;
    }
    , 0)
    contact.id = maxId;
    contacts = [...contacts, contact];
    return Observable.of(maxId);
  }
```

With this in place, we can now retrieve a "record" from Contacts when we access the edit route.

You can use this same kind of code in your own applications. In fact, I do this all the time when I don't have control over the server-side code but I know what I should be getting back from the server. All I need to do is to mock out the data in my service and I can continue coding. When the server is finally ready, I can replace the code with calls to the database.

The code so far is in branch "step-16-CLI-1.5.2"

17 - Load a Contact

Now that we have the code in our service in place, we can work on getting that data into our Edit component. We are going to need an action for this that we will call Get to match the method in our service. It takes the ID as a parameter. Write this code now. It belongs in `edit.actions.ts`.

Once we retrieve the data using an Effect that will respond to the Get action, we'll need to send the data to our reducer. We can use the Update action we've already created for that.

Now, add an `Effect` that responds to the `Get` action, retrieves the `Contact` from the service and returns an `Update` action that will be picked up by the reducer. Since we haven't created an Edit Effects file, you'll need to create this first.

```
@Injectable()
export class EditEffects {
    @Effect()
    get$: Observable<Edit.Update> =
        this.actions$
        .ofType(Edit.GET)
        .switchMap((action: Edit.Get):
 Observable<{}|ReadonlyArray<Contact>> =>
          this.contactsService
            .get(action.id))
        .map((x: ReadonlyArray<Contact>):
ReadonlyArray<EditForm> => x.map((c:
Contact) =>
            ({
                firstName: c.firstName,
                lastName: c.lastName,
                dateOfBirth:
                  c.dateOfBirth
                    .toLocaleDateString()
            })
        ))
        .map((x: ReadonlyArray<EditForm>):
           Edit.Update =>
             new Edit.Update(x[0]));
```

```
 constructor(
   private actions$: Actions,
   private contactsService:
ContactsService
  ) {}
}
```

The key bit of code that may not be obvious just from looking at it is that we are transforming the data from a `Contact` type to an `EditForm` type and then passing the `EditForm` item to `Update()`. `toLocaleString()` converts our Date object into a string using the current locale settings.

Now that we have this in place, we can go back to our `EditComponent` class and send off a notification in our `ngOnInit()` method to load the record for the ID that was passed as part of the route.

First, you'll need to inject an `ActivatedRoute` into the constructor.

```
constructor(
  private formBuilder: FormBuilder,
  private store: Store<AppState>,
  private route: ActivatedRoute) { ...
```

Then as the last line of `ngOnInit()` add:

```
this
  .route.params.first()
  .subscribe(
    (params: Map<string, string>) =>
    {
```

```
        this.store.dispatch(
          new Edit.Get(parseInt(
              params['id'] ?
            params['id'] : '-1', 10)));
    });
```

What this is doing is subscribing to the current route and grabbing the current id parameter from it. The else -1 is for the add route which won't have an ID. We are using `parseInt` because Get is expecting an integer but `params` only holds strings.

The other thing that may look odd is the `Map<>` type definition. Here we are telling TypeScript that we are passing an object with properties that are strings and values that are strings. This is always true of route parameters.

Finally, don't forget to register the Effect with the Edit module or you'll be wondering why no data comes back when you load the Edit route. Do that now.

Now, if you run this code, you should see an edit screen that has information in it for the ID that was selected.

To get the tests to work, you'll need to add the provider `APP_BASE_HREF` and import the empty `RouterModule.forRoot()` to the `edit.component.spec.ts` file.

The code so far is in branch "step-17-CLI-1.5.2".

18 - Save a Contact

Now that we can load a record from our pseudo database into our editor, the next logical thing we should be able to do is to save

the data back into our database. And here is one of the places that is going to seem slightly different from what you may be used to doing.

You see, as we've been updating the database, we've been updating our Edit Store entity with the changes. There is no need to try to retrieve the data from the Edit component. It is already in the store.

Instead, all we need to do is to create a Save action in our Edit Actions which will get picked up by an Effect.

Since we didn't save the ID as part of the state information, we will need to make a few changes now. Change the `EditForm` interface to require an ID.

```
export interface EditForm {
    id: number;
    firstName: string;
    lastName: string;
    dateOfBirth: string;
}
```

Now build your application and fix the errors that popup so that `EditForm` types have an id.

This is one of the reasons I recommend enforcing strong type checking. Imagine making this change if you couldn't rely on the compiler to tell you all the places that are impacted by the change.

Now that we are storing the ID, we'll also need to make ID part of the form. Add a hidden field in the form to hold the ID and add it to the form group, otherwise id will be undefined when we go to save the form later.

```
this.form = this.formBuilder.group({
  id: ['', Validators.nullValidator],
  firstName: ['', Validators.required],
  lastName: ['', Validators.required],
  dateOfBirth: ['', Validators.compose(
    [ Validators.required,
      EditComponent.isDate]
  )]
});
```

The `Validators.nullValidator` is what you use when you don't need any validations.

And in the template:

```
<form class="form-horizontal"
     [formGroup]="form">
      <input type="hidden"
          formControlName="id">
```

Next, create a Save action. Now that we've updated our `EditForm` interface, we won't need to pass any data.

And now, we create an Effect that responds to the Save action. Remember, we need to get the state information from the store. To do this, we need to inject the Store into the `EditEffects` constructor.

Do this now. It looks the same as every other time we've injected the Store.

Once you have the store injected, your Effect will look like this:

```
@Effect()
save$: Observable<Edit.Get> =
this.actions$
```

```
  .ofType(Edit.SAVE)
  .switchMap(
    (action: Edit.Save):
       Observable<EditForm>  =>
         this.store.select(
           (x: AppState) => x.edit.form)
           .first()
     )
  .switchMap(
    (form: EditForm):
      Observable<{} | number> =>
        this
          .contactsService
          .update({
            id: form.id,
            firstName: form.firstName,
            lastName: form.lastName,
            dateOfBirth: new Date(
              form.dateOfBirth
            )
          })
)
.map((id: number) => new Edit.Get(id));
```

The most complicated Effect we've written yet. And yet, not really all that complicated at all.

The first `switchMap()` allows us to grab the form data from the store. One of the critical parts of this script is `.first()` which ensures we only get the data once. Otherwise, you'll end up in an infinite loop caused by multiple changes to the form.

The second `switchMap()` saves the data into the service.

This returns an `id` which we then dispatch with the `Get` action to reload the data.

The only thing left is to do is to wire up the Save button.

Put a click handler on the button in the `edit.component.html` template and a `save()` method in the `EditComponent` class. In the `save()` method, dispatch an `Edit.Save()` action.

This should be familiar. You can find code where you've done this in the past and copy/paste/modify.

While we are here, let's also wire up the "Cancel" button to take us back to the list.

This will use the Router so we'll need to inject that into the constructor.

```
constructor(
    private formBuilder: FormBuilder,
    private store: Store<AppState>,
    private route: ActivatedRoute,
    private router: Router) {
```

And then a cancel method that routes to the list route.

```
cancel(): void {
  this.router.navigate(['/list']);
}
```

Also, add a click handler to the Cancel button that calls this `cancel()` method.

Now, we can save data, and cancel to get back to the list and see the change there and then come back into the edit screen and see that the data is there as well.

This code is in branch “step-18-CLI-1.5.2”

19 - Add Contact

This is the last step before we have a full working application!

We have just about everything we need in place. However, when we add a contact, the add route gets triggered and it is going to request a record of id: -1, which doesn’t exist.

To fix this, we are going to change our Get Effect to handle when no data comes back. This happens when the record that comes back from the contactsService is undefined.

Now our Effect looks like this:

```
@Effect()
get$: Observable<Edit.Update> =
    this.actions$
    .ofType(Edit.GET)
    .switchMap((action: Edit.Get):
 Observable<{}|ReadonlyArray<Contact>> =>
    this.contactsService.get(action.id))
    // no record means we retrieved id -1
    .map((records:
ReadonlyArray<Contact>):
        Contact => records[0] || {
            id: -1,
            firstName: '',
            lastName: '',
            dateOfBirth: null
```

```
        }
    )
    .map((x: Contact): EditForm =>
        ({
            id: x.id,
            firstName: x.firstName,
            lastName: x.lastName,
            dateOfBirth: x.dateOfBirth ?
      x.dateOfBirth.toLocaleDateString() :
      ''
        })
    )
    .map((x: EditForm): Edit.Update =>
        new Edit.Update(x));
```

Now, when save$ ends up calling `contactsService.update()` it will get passed a -1. The easiest way of dealing with this is to adjust the `update()` method to handle either adding a record or saving an existing record. Open `ContactService` and change `update()` to look like this:

```
update(contact: Contact):
    Observable<number> {
    if(contact.id < 0) {
      return this.add(contact);
    }
    const c: Contact = contacts.find(
     (x: Contact) => x.id === contact.id);
    c.dateOfBirth = contact.dateOfBirth;
    c.firstName = contact.firstName;
    c.lastName = contact.lastName;
    contacts = [...contacts];
```

```
    return Observable.of(contact.id);
  }
```

And change `add()` to return a number with the new id value

```
add(contact: Contact): Observable<number>
{
    const maxId: number =
    contacts.reduce(
      (max: number, c: Contact) => {
        if(max < c.id) {
          return c.id;
        }
        return max;
      }
    , 0)
    contact.id = maxId + 1;
    contacts = [...contacts, contact];
    return Observable.of(contact.id);
}
```

And now we have a fully functioning CRUD application.

The code is in branch “step-19-CLI-1.5.2”

20 - Change Detection

There are two objections I normally hear when I first introduce NgRX to programmers. The first is, “isn’t that going to make my applications slow?” And the second is, “that’s a whole lot of extra code, scattered all over the place, just to get clean separation!”

The first objection is easy to handle.

When people are first exposed to the functional nature of NgRX and realize that every time they change the value of an object, they'll need to create a new object, they immediately assume the code will be slow. The truth is, the code they write will be slower.

What they aren't seeing is the code the framework must run. There are two places the framework code is running that will be optimized by using new objects every time the state changes.

The first is change detection. If Angular knows that the data has changed because the object has changed, that is much easier to detect than checking every value in the object. And if the object has child objects, it is even slower.

Second, if we can tell the framework to only perform change detection when an object changes, we reduce the number of change detections that must take place.

Third, and indirectly, because we are rewriting less of the DOM, we save even more time. Manipulating the DOM using JavaScript is one of the slowest operations in JavaScript.

And all we need to do to make sure Angular knows we are using this more Functional approach is to implement `changeDetection: ChangeDetectionStrategy.OnPush` inside each of our @Component() directives.

Our EditComponent class ends up looking like this:

```
@Component({
  selector: 'app-edit',
  templateUrl: './edit.component.html',
  styleUrls: ['./edit.component.css'],
```

```
  changeDetection:
      ChangeDetectionStrategy.OnPush
})
```

While you won't be able to tell the difference in the application we wrote, you could add this now and the code should continue to work. When you write larger applications, this will matter more.

You can force the CLI to create components with OnPush change detection by default by opening the .angular-cli.json file and navigating to the bottom of the file where the "defaults" section is and changing "component" to look like this:

```
"defaults": {
  "styleExt": "css",
  "component": {
    "changeDetection": "OnPush"
  }
}
```

At this point, add the `OnPush` change detection strategy to all your components.

But, how does Angular know to look for changes at all?

One of the libraries that Angular depends on is a library called Zone.JS. One of the features of Zone.JS is that it detects when any event has occurred. Angular uses that as an indication that something must have changed. If you need to, you can run your code outside of Zone.JS, but you'll hardly ever need to.

While we are talking about Zone.JS, another nice feature is that it keeps track of all the context you are running in so that it can provide a better stack trace. What does that mean?

If you've been programming using JavaScript for any length of time, I'm sure you've run into situations where you have some exception that got thrown. The stack trace takes you all the back to the beginning of the callback that caused it. But what called the callback?

Zones keeps track of who called the callbacks and provides you a stack trace that includes those calls too.

Now, the side effect of using Zones is that there is going to come a time when you implement `OnPush` notification and your code no longer updates the screen correctly. Or it doesn't update correctly unless you click on something.

This is an indication that you've made a change and no event occurred, or occurred at the right time, to let Angular know it should perform change detection. It doesn't happen often, but it does happen often enough that you need to know how to fix the problem.

The standard fix for this is to inject `ChangeDetectorRef` into the constructor of your component that isn't updating correctly and call `markForCheck()` on the object when the data that isn't being reflected correctly in your component is changed.

The code so far is in branch "step-20-CLI-1.5.2"

21 - "404" Route

Normally, when a page can't be found on the server, a 404 error is returned. But, once we are running an Angular application, all our "pages" are running on the client. And even if we ask for the

wrong page the first time, the server should dutifully serve up the application and let Angular figure out where the page is.

Which all means, now Angular is responsible for displaying “file not found” pages.

The simple fix for this is to create a wildcard route in your top-level route that says, “anything else should load this file not found route.”

In the application we created above, this route information would go into `app-routing.module.ts`.

To start, we need to create a route for our 404 page, just like every other route we’ve created. The secret sauce to making this load every time a page can’t be found is that we add a wild card route at the bottom of our route definitions that redirects to our 404 route whenever a route can’t be found.

```
const routes: Routes = [{
  path: '',
  redirectTo: 'list',
  pathMatch: 'full'
}, {
  path: 'list',
  loadChildren:
    './routes/list/list.module#ListModule'
}, {
  path: 'edit/:id',
  loadChildren:
    './routes/edit/edit.module#EditModule'
}, {
  path: 'add',
  loadChildren:
```

```
    './routes/edit/edit.module#EditModule'
}, {
  path: 'file-not-found',
  // next two lines are all one line
  loadChildren: './routes/file-not-found/file-
not-found.module#FileNotFoundModule'

}, {
  path: '**',
  redirectTo: 'file-not-found'
}];
```

Using what you've learned so far, add the "file not found" route now.

This code is in branch "step-21-CLI-1.5.2"

22- Custom Form Controls

Up until now, we've used components as a way of displaying data, but we've yet to create anything that looks like a control. This isn't because the concept is difficult. We just haven't had a need for it. We still don't have a need, but we are going to convert our label/input pair in our Edit route to a custom control so you can see just how easy this is.

If you are just creating controls that will be used for your application, you can create your controls under the shared directory. But, if you are going to create controls and components that will be used between projects, you might find it useful to create each control in a separate project. This prevents you from having to do an "all or nothing" upgrade when one of your components requires a dependency that you may not be willing to take. Yet, you need a bug fix that is in some other

component. Keeping each component isolated and independent will save you headaches down the road.

Now, on with our control. The first thing we need to do is that we need to create a new component. A custom control is just a component with some interfaces implemented.

Navigate to the shared directory and use the `ng` command to create an TextInput component.

You'll remember that this declares the component in the module, but it doesn't export it. Add the export statement to the shared module now.

In your Edit route, you already have the HTML you are going to need for a template. We are going to pull out one of the `<div class="form-group">` sets and work with that. This will allow us to encapsulate a lot of what we've been doing to layout the form, all in one place. In a normal control, we wouldn't want to encapsulate the Bootstrap classes so much, but for our purposes, this will work to illustrate how you would make your own custom control.

If you copy out the fragment and put it in the template for the control, that will be a great start.

Then, replace the input fragments in the Edit control with references to `<app-text-input></app-text-input>` while keeping the `formControlName` associated with each input control, associated with each `app-text-input` control for all three inputs on the form.

You'll see that making this change significantly simplifies our Edit template.

Now, to make our component a control, you'll need to implement the `ControlValueAccessor` interface. Add this interface to your implements clause in the `TextInputComponent` class and implement the functions for that interface.

```
export class TextInputComponent
  implements OnInit, ControlValueAccessor
{
  constructor() { }

  ngOnInit(): void {
  }

  writeValue(obj: any): void {
    throw new Error(
      'Not implemented yet.');
  }

  registerOnChange(fn: any): void {
    throw new Error(
      'Not implemented yet.');
  }

  registerOnTouched(fn: any): void {
    throw new Error(
      'Not implemented yet.');
  }

  setDisabledState(
    isDisabled: boolean): void {
    throw new Error(
```

```
        'Not implemented yet.');
    }
}
```

We will fill these methods in later. This interface tells Angular how to communicate with our component.

The next bit of code we need to implement finishes the ability for Angular to communicate with our code.

Place this fragment right above the `@Component` decorator.

```
const CUSTOM_INPUT_CONTROL_VALUE_ACCESSOR:
  ExistingProvider = {
  provide: NG_VALUE_ACCESSOR,
  useExisting: forwardRef(
    () => TextInputComponent),
  multi: true
};
```

And next, we register this provider with the control in the providers section.

```
providers:
  [CUSTOM_INPUT_CONTROL_VALUE_ACCESSOR]
```

This is boiler plate code that you will add for every control you create.

But, since our linting rules won't let us use a Class, or Function before it is declared, the code above won't pass our linting rules. But, it can't be helped. So, add a comment above "useExisting" to disable the rule temporarily.

```
// tslint:disable-next-line:no-use-before-
declare
```

The first method we need to implement is `writeValue()` in this case, the value will always be a string so you can change the typing to string. We will then assign the value to an internal member variable named _value.

But, how does our input field get updated with this new information? For that, we create a value property and bind to it from our template. We will implement this once we have the other methods in place.

`registerOnChange()` is a function that Angular calls to register a change listener. Whenever the value changes, we'll call the function that `registerOnChange()` passed to us.

`registerOnTouched()` is a function that Angular calls to register the function it wants us to call whenever the blur event occurs.

`setDisabledState()` sets the control's disabled state to on or off.

From this it is clear that any time the value property changes, we need to call the `onChange` method. We also need to detect when the blur event has fired on our input field.

The code for the interior of our component class will end up looking like this:

```
  _value: string;
  constructor() { }
  _onChange: Function = () => {};
  _onBlur: Function = () => {};
```

```
ngOnInit(): void {
}

writeValue(obj: string): void {
  this._value = obj;
}

set value(v: string) {
  if(v !== this._value) {
    this._value = v;
    this._onChange(v);
  }
}

get value(): string {
  return this._value;
}

registerOnChange(fn: Function):
  void {
  this._onChange = fn;
}

registerOnTouched(fn: Function):
  void {
  this._onBlur = fn;
}
```

Next, we need to get the value into the `input` element. To do this, we will use binding. This is a place where using templated forms works better than reactive forms. But, to do that, we'll

need to import `FormsModule` into the Shared module. Also, since we are using this component is a lazy-loaded module, we need to import the Shared module into the Edit module.

Do that now.

Find the `input` element in `the text-input.component.html` template and add an `ngModel` binding. While you are here, also take out the `formControlName` property.

```
<input type="text"
  class="form-control"
  placeholder="first name"
  id="firstName"
  [(ngModel)]="value">
```

Now, we need to bind the blur event in the input field to an `onBlur` method that will call the event handler passed into `registerOnTouched()`

First, in the template:

```
<input type="text"
  class="form-control"
  placeholder="first name"
  id="firstName"
  [(ngModel)]="value"
  (blur)="onBlur()">
```

Then in our class:

```
onBlur(): void {
  this._onBlur();
}
```

The only thing left is taking care of `setDisabledState()`. While we don't need it for this code, it is still useful to implement it. The thing is, we need to pass the disabled state down to the input field. But, how do we gain access to the input field?

There are two ways we could implement this. We could just bind `disabled` to a member variable in the input field. This would probably be the easiest way to implement it. But, there is another way that will also teach you about another feature of Angular.

At some point in your Angular journey, you'll need to access a child component. For this, you need the `@ViewChild()` decorator.

`@ViewChild` takes one parameter. Either a string that represents a template variable, or a class name (not a string) that represents the component we are trying to retrieve.

In this case, we will try to access the input field using the template variable method. To do this, we need to assign it a template variable name. The most obvious choice is `#input`

```
<input #input type="text"
  class="form-control"
  placeholder="first name"
  id="firstName"
```

```
  [(ngModel)]="value"
  (blur)="onBlur()">
```

Now, in our class file, we can access it by creating a member variable:

```
export class TextInputComponent
  implements OnInit, ControlValueAccessor
{
  @ViewChild('input') input:
     HTMLInputElement;
  ...
```

Now, in our `setDisabledState()` method we can set the disabled state of the input element:

```
setDisabledState(isDisabled:
  boolean): void {
    this.input.disabled = isDisabled;
}
```

And that's the basics of creating controls. But, we aren't done with ours yet.

First, our control has a label that references the input field by ID. We can give our control an ID and then use that value to create another ID for the input field. To do this, we will need to inject ElementRef into our constructor, and then assign the new ID to a member variable that we will bind to from the template.

To make the variable something we can bind to we need to decorate it with the @Input() decorator.

```
@Input('id') inputId;
```

This is going to cause a linting error because normally, we don't want the input name and the variable name to be different. I our case here, we do want them to be different, so you'll want to disable the check with:

```
// tslint:disable-next-line:no-input-
rename;
```

And now we can use it in our constructor.

```
constructor(elementRef: ElementRef) {
  this.inputId =
    elementRef.nativeElement.id + '-
input';
  }
```

And in the Edit template

```
<app-text-input
formControlName="firstName"
  id="fn"></app-text-input>
<app-text-input formControlName="lastName"
  id="ln"></app-text-input>
<app-text-input
  formControlName="dateOfBirth"
  id="dob"></app-text-input>
```

And now clean up the template for the `TextInput` component to use the `inputId` value.

```
<div class="form-group">
  <label class="col-lg-3"
    [for]="inputId">
    First Name:</label>
```

```
  <div class="col-lg-9">
    <input #input type="text"
      class="form-control"
      placeholder="first name"
      [id]="inputId"
      [(ngModel)]="value"
      (blur)="onBlur()">
  </div>
</div>
```

Now, if you run the code now, you'll see that all the labels are the same. We could fix this by just setting a property, but there is another way of solving this problem that will teach you something new.

We can make use of `ng-content`. By putting an `ng-content` tag where we want the label to appear, any content we put between the open and closing `app-text-input` tag will appear in our label.

```
<label class="col-lg-3"
  [for]="inputId">
  <ng-content>
  </ng-content>
</label>
```

And now, our Edit route template looks like this:

```
<app-text-input
  formControlName="firstName" id="fn">
    First Name:
</app-text-input>
<app-text-input formControlName="lastName"
  id="ln">
```

```
    Last Name:
</app-text-input>
<app-text-input
  formControlName="dateOfBirth"
  id="dob">
    Date of Birth:
</app-text-input>
```

I will admit it is a slightly contrived example, but it does show something of the flexibility you have available.

`ng-content` works in any component. Not just controls. If you want to have multiple areas where you can insert content, you can assign a selector to `ng-content`.

```
<ng-content select=".errors"></ng-content>
```

And then from the Edit route:

```
<app-text-input
  formControlName="dateOfBirth"
   id="dob">
      Date of Birth:
      <div class="errors">Errors go
here!</div>
</app-text-input>
```

“Date of Birth:” will show up as the label and “Errors go here!” will show up under the input field.

Selector can be any valid CSS selector “.class”, “element” or “[attribute]” or any combination.

Now, to fix up the tests.

We will need to import `FormsModule` and `SharedModule` in `edit.component.spec.ts` by placing them in the imports section of the `TestBed` configuaration. Make sure `SharedModule` comes after `StoreModule.forRoot()`. You will also need to add an empty `EffectsModule.forRoot()` before `SharedModule` since `SharedModule` imports `@Effects` code.

You will also need to add `FormsModule` to the imports section of `text-input.component.spec.ts`

The code so far is in branch "step-22-CLI-1.5.2"

23 – What About Wait?

You may remember that when we started this tutorial, we created a Wait component. I showed you that it worked, but we haven't used it since. We really don't need it because we aren't waiting for anything slow. But, if we were using `HttpClient` to go after real data, there is a possibility that there would be some lag time and we might want to display the wait.

Since we don't have a real server, we are going to simulate wait time in our `ContactsService.list()` method. Change the return statement to look like this:

```
return Observable
  .timer(2000).first()
  .switchMap(() =>
    Observable.from([contacts])
  );
```

This will cause the `list()` method to wait two seconds before returning the contacts.

Now that we have some latency, we can introduce how we would handle displaying the Wait component. Remember, all we must do is dispatch `Wait.start()`. So, we are going to inject the Store into the constructor of `ContactsService` which will allow us to call dispatch.

```
constructor(private store:
Store<AppState>)
{}
```

And then we can dispatch start before our return statement.

```
this.store.dispatch(new Wait.Start());
```

But how do we end it? Easy! Observables have a `finally()` method that will get triggered regardless of how the Observable ends. We just need to tack a `finally()` onto the end of the return and dispatch the end.

```
return Observable
  .timer(2000)
  .first()
  .switchMap(() =>
    Observable
      .from([contacts])
  ).finally(() =>
     this.store.dispatch(new Wait.End()));
```

This pattern would work the same way if we were observing an `HttpClient` call.

If you run this code, and then go add a record, and then come back to the list, you'll notice that while we are waiting for the new records to show, the old records will already be on the page. This happens because it is displaying the data that is currently in

the Store. Once the NgRX code updates the state, the new information is displayed.

You'll need to import `StoreModule.forRoot({})` in the TestBed configuration for the tests to pass on the service.

This code is in branch "step-23-CLI-1.5.2"

Using Angular with Backends

We haven't talked at all about how to use Angular with your backend server because, I don't know what your backend server is. I'm trying to keep this book as generic as possible.

But, there are a few things I can point out that may enable you to develop more efficiently.

Development

While you are developing, you can set your backend server up, however you do that, and then you can setup a proxy on the Angular Dev Server that you start with `npm start` or `npm run startDebug`, to go get the data on your server.

There are two steps to this. First create a file in the root of your project named, `proxy.conf.json`. Inside this file, define your proxies.

A simple example would look something like:

```
{
  "/api": {
    "target": "http://localhost:3000",
    "secure": false,
    "pathRewrite": {
      "^/api": ""
    }
```

```
  }
}
```

Which tells ng serve that anytime a relative request is made to "/api" it should be redirected to http://localhost:3000/api instead.

If you need to access a server that isn't on localhost, you'll need to use the "changeOrigin" options and set it to true otherwise you'll end up with a Cross Origin Request (CORS) issue.

```
{
  "/api": {
    "target": "http://npmjs.org",
    "secure": false,
    "pathRewrite": {
      "^/api": ""
    },
    "changeOrigin": true
  }
}
```

It isn't that hard to figure out once you know it exist.

The second thing you'll need to do is you'll need to tell `ng serve` to use the proxy file.

```
ng serve --proxy proxy.conf.json
```

You just need to update the `package.json` start scripts to include this option and you'll be all set.

If you need more information, you can read the current Angular CLI documentation.

Once you are ready to deploy your application, you'll take the files from your build directory and put them in the root of your

application directory. If you are deploying to a subdirectory off the root of your domain, you'll want to build your application using the `--base-href` option. This will throw in the `base` tag in the header of your index.html page.

Another place you may get stuck when you move the code to a real server is that when you ask for a route directly, you may get a 404 error. This is because the server doesn't know anything about your route.

To fix this, you'll need to use something like mod_rewrite, or your server's equivalent, to write rules that tell the server to always return the index.html file for paths that look like routes.

Unit Testing

Throughout the course of this book, we've not spent much time dealing with the topic of testing other than to fix tests that were created for us so that they would succeed. Through that process, you've probably already caught on that our `*.component.spec.ts` files have a `TestBed.configureTestingModule` method that must be updated to include everything that typically goes in our module definitions.

And, I've already mentioned that I'm not a huge fan of testing component logic, so I avoid putting logic in my components.

But, why test at all?

Why Test?

If you are anything like me, you have at least been convinced of the need to learn unit testing, but you probably have little to no idea how unit testing fits into the overall development life cycle. You may think you do, but trust me, if you are just starting out, you have no idea.

You see, most people think that unit testing is about testing. I understand the confusion. After all, it is called unit TESTING. As if TESTING is what this is all about. But unit tests, and their cousins systems testing, and application testing, are more about development than testing.

In fact, when people first get started with testing, they tend to write tests after they've written the application. There are several problems with this.

First, if you wait until you are done writing the code, you probably won't write any test. We all know that if we follow the classic "waterfall" life-cycle, testing always shows up at the end. Deadlines slip. Everything takes longer than we thought when we started the project and we end up with a deadline and no formal, repeatable tests. If you wait until the end, the chances that you'll write tests are very small.

Testing Influences Design

But this isn't the only problem with waiting until the end of the development process. Let's assume that you can write some test. What you'll find is that because you were not thinking about testing when you were writing your code, you wrote your code in a way that makes your code very hard to test.

Let me illustrate with a real-life example.

At one of my previous contracts, our manager decided to have one of our guys write a set of classes that determined if a user could access a screen based on four criteria. Each of those criteria were developed as separate classes because each could be used independently elsewhere in the system. Those separate components were easy to create unit tests for. However, when it got to the point where they wanted to test the class that wrapped them all, it became obvious that the wrapper couldn't be tested as a distinct unit because it had a pretty heavy dependency on the other classes.

As we thought about the problem, it looked like the only way to solve the issue was to use mocking and dependency injection (if you don't know what these are, don't worry, we'll get into those later). But then I realized all the wrapper class really needed was the results from the other classes. If we did that, we could easily

just pass results in and get the result of that. To make that all usable without having to know anything, we just needed to wrap that all in a class that hides everything and forget about testing it since the cyclomatic complexity of that method was probably going to be about three. Low enough that the method could be verified through inspection rather than by testing.

I tell you that story to illustrate what for me is the main reason for testing. It influences your design. Influencing your design helps you write more maintainable code.

Testing Clarifies the Requirements

By writing tests first, you are forced to develop clearer specifications. I've run into this recently on a project that I'm currently working on. I can't write a test for the code I'm about to implement because I don't clearly understand how this is supposed to interact with the rest of the application. Until I do, I can't move forward. If I were not writing a test, this would not be as clear now as it is. Oone could argue that eventually it would become clear. But it is more likely I would leave the feature out completely because I forgot about it entirely. Something I've been known to do in the past.

The Tests Create the Specification

By writing tests in advance, you have the specification coded. This forces you to keep the specification up to date if it changes because your tests won't run unless you do. How many other programming methods are there that force the specifications to be kept up to date? I can't think of any.

Several years ago, long before the community was actively talking about Test Driven Development, I worked for a short

time at a company as a "bug fixer." That was my role. They had hired me because they had some software that was "basically done" but "had some issues." It should only take a few weeks.

The first thing they needed me to do was to get their web site to send out email. It turns out it was a configuration problem. But they were so impressed ("the last guy we had in here spent two weeks on that problem and still hadn't solved the problem.") that they gave me more and more bugs.

This Is the Job That Never Ends

The gig that was supposed to be a couple of weeks long was quickly turning into a perpetual job. Soon I learned that what I was working with was a system that had a lot of bugs. Unfortunately, no one else seemed willing to admit that.

Eventually, frustrated by the fact that this system seemed to have a new bug every day, I asked for the specifications so that I could create a test plan. That's when I found out the worse news of all about this system.

Lost Specifications

They had lost the specs. No, really! And it gets worse. Not only had they lost the specs, but they were unwilling to admit this to the client and instead they were relying on the process of fixing the bugs to eventually squash all the bugs so they could end up with a stable system.

Since I was not yet familiar with the concepts of unit testing or Test-Driven Development, I accepted this as the best we could do. At least I was getting paid well.

The Plot Thickens

About three months into this gig, the manager of my project went on vacation which left the project in HIS manager's hands. That's when the poop hit the fan.

The Oracle consultant that was working with me and I were called into the office.

"Why does this system still have bugs?!!" Oh, he was angry! That should be all bold and all caps and all underlined. He was out of control.

I responded, “Well sir, several weeks ago I asked for the requirements document so that I could write a test plan and I was told the requirements were lost. If we can't write a test plan, we never will be able to ensure that the system is working the way that it should.”

To which he responded, “I want you to write a test plan.”

I guess he heard “test plan” but not much else of what I said.

So, I repeated my need for a requirements document.

I went back and forth with him insisting that I write a test plan and me stating that it could not be done without requirements until I finally said, "I think I've done all I can do here." Walked out of the office, packed up my stuff, went home.

But, it didn't have to end like that.

A Better Way

Had I known about test driven development, every time a new bug came in, I could have written a new test, even if it was only a unit test and not acceptance test. Eventually, I would have

created not only the test plan, but I would have created the specification, or at least the parts that tended to break, and we would have ended up with a stable system like they thought they were going to get.

A Safety Net

Have you ever looked at bad code and thought, “I bet I could make this better” only to realize that making a change might break something? You'll really appreciate TDD. If there is a good test suite for the code you want to re-factor, you can be sure that any changes you make won't break something it should do. I've left a lot of code alone for fear of breaking something else.

But, when I have a test suite, I've could quickly re-factor my code and know that it still works at least as well as it did prior to my re-factoring.

I experienced this first hand on one rather complicated system. I received the requirements in bits and pieces. Even when I started, I realized that the only way I could be sure this was working correctly would be to write tests as I went.

It is a good thing I did because one day, I received another piece of the puzzle that significantly complicated how I needed to write the code. Soon, I realized that the only way I was going to be able to accommodate this new requirement would also require me to significantly rewrite the existing code.

It is only because I had tests in place that I could do this with any amount of confidence that the code still fulfilled all the requirements I’d already coded for.

Prevent Feature Creep

Another way that creating tests first helps your development process is that it prevents feature creep on the part of developers. How many times have you added a feature into the system that no one asked for? By coding to the test, you reduce this urge.

Better Design

Many people start unit testing by writing test after the fact and wonder how this can possibly be helpful. This is because they've written them after they've written the code and they've completely bypass 80% of the benefits. Sure, you've got some test. But that is all you have and creating the tests probably took you twice as long as it should have simply because you wrote them after the fact.

Faster Testing

The final reason you should write test is that it is much faster to test using automation and it speeds development when you do it right.

Let me try to explain this because most people think the opposite is true. They think that by writing test, you'll be adding time to the development process. Part of the reason for this is because they've always added testing on to the end of the development process and, as I've already explained, adding test at the end takes much longer than adding them during the creating of the main code.

But let me ask you this: Don't you already test the code you write? I'm not talking automated testing. But most of us at least run the code we've written periodically to see if what we've

written does what we thought it was going to do. Some of us test more frequently than others, but we all do it.

Let's take a regular web page as an example. Something simple. We are working on a simple page that collects information from a user and puts it into a database.

Doing this manually, we must load the web page, fill in the information, click the submit button, wait for the page to reload. And then we must check to make sure the data got saved as we expected.

And we must do that until we get the page right for all our happy path conditions (the user gave us data we expected.) And all our sad path conditions (the user gave us various forms of invalid data on the form.)

But if you write a test for each of those conditions, you can just click a button and wait for the results to come back.

In addition, as you continue working on the page, you can be ensured that adding a new feature didn't break what was already working. This normally doesn't get found until we've given the finished product to the user when we normally say something like: "Odd, it WAS working at one time. I remember testing it." Yes, but have you tested it recently? How many other changes have you made since you tested it?

Excuses for Not Testing

As I started my own journey into unit testing, I slowly began to realize that it is easy to come up with reasons to NOT test my code as I was writing it, even once I understood what that was supposed to look like. The reason I think most programmers don't unit test code, once they understand what it is they are

supposed to be doing is that they don't feel like they have permission.

To this I also answer, "How much permission do you need?"

Do you ask for permission to compile and link the code?

Do you ask for permission to write every line of code to make the system does what it should do?

Do you ask for permission to run your code periodically to make sure it does what you had in mind when you wrote the code?

Do you periodically add code that makes you feel good but is not directly related to the task at hand? Admit it, I don't think I know of any programmer who doesn't.

Then why do we feel like we need permission to write unit test for our code?

We don't write test code because we aren't convinced it is necessary to do the job we've been given. We complain that our managers don't want us to write unit test. But the problem is that you asked for permission in the first place. And, by asking for permission, you've basically told your manager that unit testing is optional. Your manager has said “no” because YOU think it is optional.

It isn’t his job to understand that not testing will produce technical debt. He's not even interested in understanding what technical debt is. All he cares about is this. When will this project be done? When you say it is done, will it work as expected or will it have a lot of bugs that need to be fixed yet?

Most of the managers I've worked for in the past will accept whatever number I give them once they understand that when I deliver the software to them, it is going to work.

Now, I will admit, in some cases there are places where you've been explicitly told to not create unit test. But even here I will assert it is because someone asked management the question.

And so, we need to evaluate why it is we think creating unit test are optional. Probably because what we've been doing for so long seems to be working and, when we try to incorporate unit test, the process seems slower. And it is. Initially, writing unit test is slower just like writing using a new language or a new framework or anything else new is slower than what we know. But the ultimate efficiency that writing unit test as we code provides has been proven to more than offset the learning curve involved.

There is one other valid reason for not testing and that is, we simply don't know how. This is almost as big of a reason as not believing it is worthwhile. But, I think if we thought testing was worthwhile, we'd start testing and figure it out as we went along.

If you think about your career, I bet there are a lot of things you know now that you didn't know when you started out. The fact of the matter is, most of us learn on the job. We start out with basic skills, but it is the day to day implementations that improve those skills.

What I hope to do in this section is to give you enough of the skills and some strategies for how to use those skills, that you can begin to use them in your day to day coding practice. Don't let good enough be the enemy of the perfect. Your first set of tests will be garbage. As you stick with it, you'll wonder what

you were thinking when you wrote your first test. But this should not deter you. This is what happened when you first started coding. Maybe it is still happening. It is the practice that will make you better able to write tests and better able to write code that is testable.

There is another reason we don't test. Most of the code you are currently writing simply isn't testable. I'll give you some tips along the way to help make your code more testable, but it is also in the writing of tests that your code will become more testable.

What not To Test

With all that I've written about testing so far, you would think I would be a proponent of testing every line of code in your system. Not so. Here are some items you might consider not testing.

Generated Code

Generated code includes any code that uses some automated process to create code your system is using. This assumes, of course, that you have tests for the code generators that test both that they generate the expected code and the code that was generated works as expected. But to write a test for every instance of the code the generator writes is overkill.

Low Code Complexity

Another place you don’t need to test is code that has low code complexity. This includes methods that have one or two lines.

For example, a getter or setter that simply passes a value to a member variable or retrieves the data from a member variable probably doesn't need to be tested.

In a lot of the code we wrote above, all we did was dispatch an Action. That code isn't worth testing.

However, once your code introduces a condition of some sort, you need to write some tests.

Third Party Libraries

While I know it isn't true, you should assume that the third-party libraries you are using work. This is one reason I find testing components from the presentation perspective useless. If you write the component with testing in mind, the only code you should have left should be code that uses Angular directly.

You're Thinking About It Wrong

I would argue that if you are thinking about what you should write a test for, you are probably still thinking of Test Driven Development as something you do for the sake of testing rather than for the sake of design, maintenance, and problem solving.

When you write code, you should be thinking, "What problem am I trying to solve?" Or better yet, "How can I state the problem in terms of a 'When/Then' statement?"

When you think about the problem this way, what you test becomes that When/Then statement.

When you do this, the question no longer is about how much code you should test, but instead becomes "Have I written a test for every reasonable condition this code may encounter?"

Jasmine

The Angular Unit Testing framework is built on top of Jasmine. If you've used Jasmine in the past, this should be very familiar to you. Angular adds some additional methods to enable the more asynchronous nature of Angular.

Each TypeScript file should have a corresponding spec file. When you use the CLI to generate a file, it adds the corresponding spec file for you. You'll notice that modules don't have spec files because they typically have no logic worth testing.

Inside of each spec file, you'll find your test arranged in combinations of `describe`, `beforeEach`, `afterEach` and `it` functions. `describe` functions can have other describe functions inside of them as well as one or more `beforeEach`, `afterEach` or `it` functions.

`it` functions are where we test to verify something is true. This is done using an `expect` function which returns a Matcher object that has additional functions we use to test for true.

`beforeEach` functions run prior to any `it` or `it` functions within a `describe`. What is unique about `beforeEach` is that each `beforeEach` runs multiple times. Once for each `it` within the `describe` or nested `describe`. Each `it` is testing a new run of code. In this way, you are not ending up with left over state from a previous run when you run any individual test.

Similarly, `afterEach` functions run after each `it` function.

For a full description of how Jasmine works, you can visit the Jasmine documentation page.

Now, if you open `app.component.spec.ts`, you'll see within our first `beforeEach` a new function called `async`. Within that is the `TestBed` configuration method we've become so familiar with. However, if you open the `contacts.service.spec.ts` file, you'll see the same `TestBed` configuration within a `beforeEach` but no `async` method wrapping it.

What's the difference?

You'll notice that at the end of the `TestBed` configuration method in `app.component.spec.ts`, there is another method chained on to it called `compileComponents`. `compileComponents` is an asynchronous method. The `async` method wraps this so that the tests don't continue until the component has been compiled. Since we have no components to tests in a service, this `async` method is not needed in `contacts.service.spec.ts`.

Continuing our look at `app.component.spec.ts`, you'll see that our only it method creates our component using `TestBed.createComponent(AppComponent)` and assigns it to a fixture variable. This is the standard way of creating a component for testing. If it were me, I'd put this code in a `beforeEach` method so that I'm not repeating the code every time. Once you have the fixture, you can get at the `componentInstance` using `fixture.debugElement.componentInstance`. From there you can access any public field, property, or method you need.

You can also access the host element of the component using `fixture.debugElement.nativeElement` and once you have that, you can access child element using `…nativeElement.querySelector(css selector here)`. As I've already mentioned, I don't find testing that my html is structured correctly or that the data binding worked as expected very useful.

Karma

As you've probably noticed, when we run the tests, all that gets reported on the command line is how many tests were run, how many failed, and how many succeeded. If an exception or compile error occurs while running the tests, buried in the stack trace you can figure out which test failed. But I find having the tests listed out for me makes it easier to track down what needs to be fixed.

To do this, we are going to add a package to our project that will give us this additional information. Start by installing `karma-spec-reporter` as a dev dependency.

```
npm install --save-dev karma-spec-reporter
```

And add it to the list of plugins in your `karma.conf.js` file:

```
plugins: [
  require('karma-spec-reporter'),
  require('karma-jasmine'),
  require('karma-chrome-launcher'),
  require('karma-jasmine-html-reporter'),
  require('karma-coverage-istanbul-
reporter'),
```

```
    require('@angular/cli/plugins/karma')
],
```

Finally, in the reporters: section, replace ‘progess’ with ‘spec’

```
angularCli: {
  environment: 'dev'
},
reporters: ['spec', 'kjhtml'],
port: 9876,
```

Now we have a report that tell us exactly what’s happening.

Code Coverage

Code Coverage is a way of telling if the tests you’ve written exercise all the code in your application. Unfortunately, it won’t tell you if you’ve written all the test you should write. I use it as a way of determining if I’ve missed anything obvious.

To implement code coverage, you can change your test script in `package.json` to include the `--code-coverage` option. What I typically do is create a `testWithCoverage` script that includes this option. That way I can choose if I want code coverage or not when I run the tests.

I also make the code coverage script the one I use for my pre-commit hook to help enforce some code coverage prior to committing my code.

Once you’ve run the tests with the code-coverage option turned on, a new directory will be created called `coverage`. There is an index.html file directly inside this folder. If you open it with your browser, you will be able to see the code coverage summary for each directory. If you drill into the directories,

you'll see the summary for each file. Drill into the file and you'll see the detail of the coverage.

If you drill into `app.component.ts`, you'll see that we have pretty good coverage, but we never actually reach a point where we've retrieved the wait or error stores.

Since this is trivial code, we can just tell code coverage to ignore it by using the `/* istanbul ignore next */` comment in our source code. You may wonder why we use "istanbul." It is because Istanbul is the code coverage engine.

Simplify Testing

Looking at the Edit component, you'll see that we have several methods that probably don't need to be tested at all. For example, `save()` and `cancel()`. But, there are other functions that might make sense to test. For example, the `isDate()` static method that we are using for date validation. It would be helpful to write tests that ensure it really is validating our dates correctly.

One could argue that we should write a test that verifies that we can retrieve the ID parameter when it is passed. On the other hand, if that doesn't work, won't that become obvious quickly?

Now the main problem with writing tests for these now is that we must create a new component to do so. If we put this code in a service, that will allow us to write tests without having to deal with the fact that we have a component.

If you haven't already, load up the project and add in an Edit service to the `routes/edit` directory using the CLI.

Copy in all the methods, interfaces and member variables from the component.

Now, back in the component, add a `providers` section in the @Component decorator and put `EditService` in it.

```
@Component({
  selector: 'app-edit',
  templateUrl: './edit.component.html',
  styleUrls: ['./edit.component.css'],
  changeDetection:
    ChangeDetectionStrategy.OnPush,
  providers: [EditService]
})
```

And now, change your constructor so that the only thing that is injected into the component is the Edit service.

Remove all the code from within the constructor.

```
constructor(
  private editService: EditService) {
  }
```

Remove the `isDate` method.

Now, in the `ngOnInit` method, just call the `ngOnInit` method in the service and do the same with `ngOnDestroy`, `save` and `cancel`.

We no longer need anything other than `form` for member variables. So, remove all the variables except for `form`.

And now, change form to a read only property so that the template will always see the form.

If you had any other variables in the template that accessed variables in the class, you would need to implement a similar pattern.

```
get form(): FormGroup {
  return this.editService.form;
}
```

Since we moved `isDate` into `EditService`, we need to make a change in `EditService`. If you haven't already, change `EditComponent.isDate` to `EditService.isDate` in the validations.

Rebuild the application. You'll see everything still works and the code is essentially the same. One might argue it isn't as direct. But, what did this give us in exchange for this extra file?

First, we no longer need a test file for our component.

Second, we can more easily test our `isDate` function. We just need to mock a `FormControl` object and pass it in.

Because we aren't using a component for our tests, our test should run faster since we can skip the `compile` and `createComponent` steps for each test.

I've added the changes so far, including the changes I made to karma.conf.js and package.json as branch "step-24-CLI-1.5.2".

Application Testing

Just as I couldn't tell you everything about how to write Unit Test, I'm not going to tell you all you need to know about writing End to End Test. However, I can get you started and guide you past the rough patches.

The first thing you are going to want to do is to add the `--aot` flag to the e2e script in `package.json`. I've found that if you use `--aot` everywhere, you code will work the same way everywhere. End to End testing is no different

```
"e2e": "ng e2e --aot",
```

Now, you will notice an `e2e` directory in your project. This is where your End to End tests go.

Inside this directory, you will find two files. One, named `app.e2e-spec.ts`, is a jasmine test file for the application level. The second, `app.po.ts` is a file that holds a "Page Object" class that represents the application.

We'll describe page objects soon, but first, let's make some changes so that our tests will run correctly.

First, in `app.po.ts`, change `getParagraphText()` to `getHeaderText()` and change the content of the method to look for the banner.

```
getHeaderText(): promise.Promise<string> {
  return element(
    by.css('app-root .navbar-brand'))
```

```
        .getText();
}
```

You'll notice that I've added return value type of `promise.Promise`. This is in the `selenium-webdriver` package.

```
import { promise }
  from 'selenium-webdriver';
```

Now we can run our tests with

```
npm run e2e
```

This should build the project and run the tests.

Page Objects

If you are new to writing End to End tests, your first instinct is going to be to write all the code you need to test your application in the e2e-spec file. This would be a huge mistake.

Here's the problem. While writing your application, your HTML is going to move around. Remember, that is why we don't write unit test for our components. Yet, we still need something that is going to verify our pages. Just as an example, let's say you reference just one of your fields 500 times. This is not unheard of. As you develop your application, you realize that you need to move the field to another logical part of the screen so that your CSS selector to access that component changes. If you accessed it directly all 500 times, that's 500 changes you need to make. Yeah, I know search and replace can help with that, but you know as well as I do that search and replace isn't fool proof.

Instead of accessing our elements directly, we can create a page object that provides a kind of alias for that field. Either as a get method, as we did with `getHeaderText`, or as a property.

Now, we can access the field through the method we created in our page object.

And page objects aren't limited to the page. You might create page object looking things for your components as well.

Finally, you need to know that End to End tests aren't meant to test everything. They are meant to test what you can't test with Unit Tests. As a rule, do as much testing as you can with Unit Test and do enough testing with End to End testing to know that the application works well when you put all the parts together.

For the full documentation on how to use Protractor, visit http://www.protractortest.org

The code changes for this section are in branch "step-25-CLI-1.5.2"

The End

Once again, if you found errors of any kind, please report them in the issue tracker for the sample code on GitHub.

If you liked this book, please leave a favorable rating on Amazon and let your friends and co-workers know about this book.