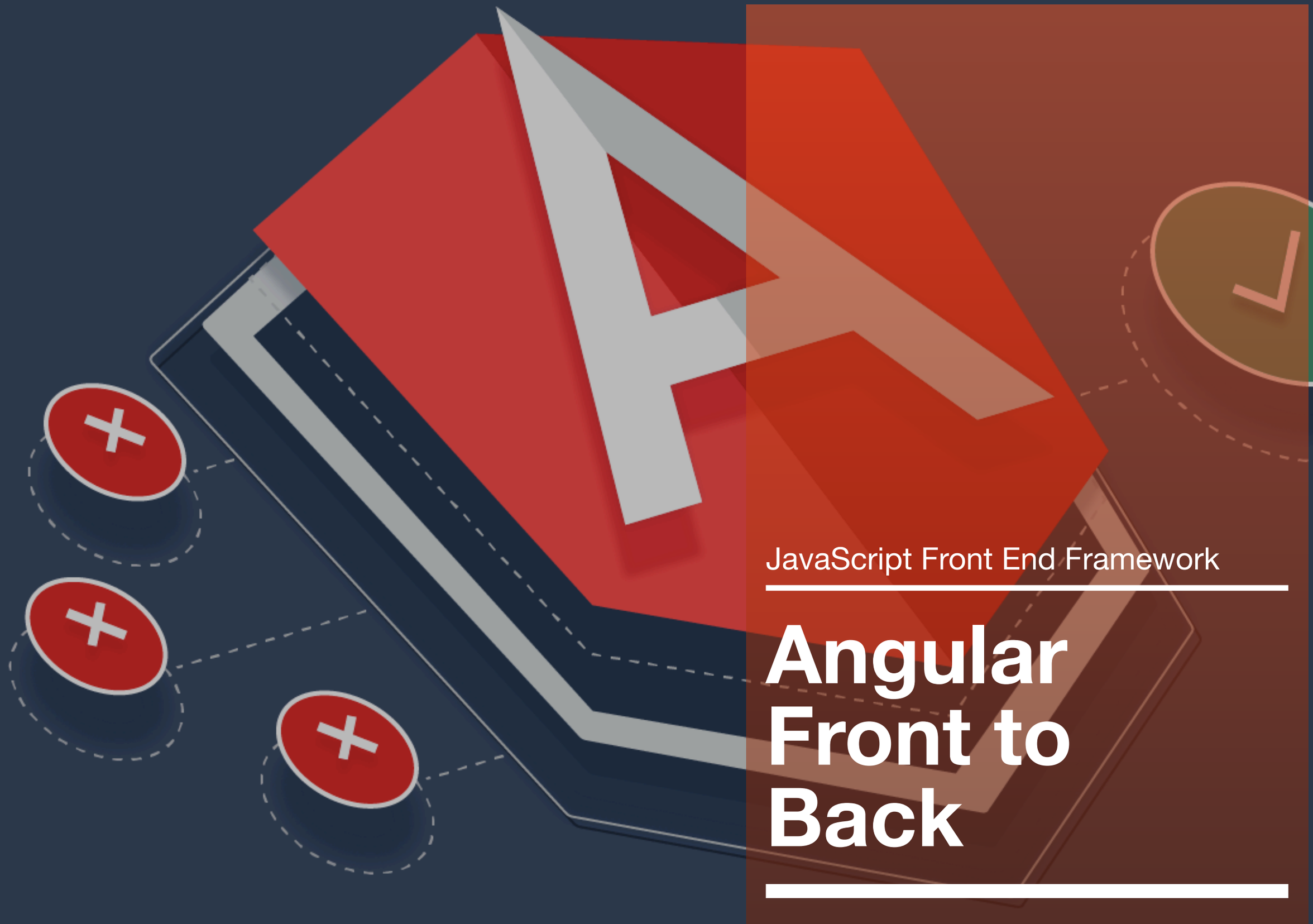


JavaScript Front End Framework

Angular Front to Back

Introduction

What is Angular? Why use Angular?

Angular is a front-end JavaScript framework that was created by and is maintained by Google. This is a framework used to create powerful front-end web applications (*front-end = running on the client side browser*). Note: it is possible to run Angular on the server side, however, at its core it is a front-end JavaScript framework. Angular is also part of a very popular full stack web development called MEAN - MogoDB, Express, Angular and NodeJs.

Angular and AngularJS are NOT the same and are two completely different frameworks.

Angular is great for rapid development & code generation compared to using just vanilla JavaScript (such as routing, HTTP request, form validations and many more). Angular also organises your code more neatly and breaks functionality up into individual components e.g. a navigation bar as one component and the search bar as another. Angular allows us to create dynamic content as opposed to static webpages. Angular is also cross platform and does not matter whether your application is viewed on a Windows, Mac or Linux or regardless of the browser viewed on such as Explorer, Safari, Chrome or Firefox. Finally, Angular has unit testing baked into the framework which makes it easy to create and run unit tests within your Angular applications.

Core Features:
Components, Services, Routing, Testing, Build Tools, Data Binding, Templating, HTTP Module, Observables, Forms Module, Directives, Pipes, Dependency Injection, Animation and TypeScript.

All of these things listed above come packaged into the core framework by default. Contrast this with the React framework where you would have to install libraries separately such as routing, http client, testing etc. However, Angular is going to be a bit more bloated compared to frameworks such as React due to all the core features.

However, with each new version of Angular, the file size are becoming more compact.

Ways to install Angular:

There are a few ways to get started with Angular and these are…

- Angular CLI
- Quickstart Seeds (*boilerplate applications - don't have all the tools the CLI provides*)
- Absolute Scratch (*not recommended*)

The Angular CLI (command line interface) requires Node.js and NPM (a JavaScript runtime and package manager) as a dependency in order to run. The CLI is the standard to quickly build an Angular project. The CLI creates a complete development environment with a dev server, build tools and everything else that you would need.

Version History:

- AngularJS / Angular 1 (*2010*)
- Angular 2 (*2015*)
- Angular 4 (*2017*)
- Angular 5 (*2017*)
- Angular 6 (*2018*)
- Angular 7 (*2018*)

Angular 3 was skipped due to the misalignment of the router package from the rest of the framework packages (there may have been some other issues as well for the skipping of version 3). AngularJS is different from Angular because AngularJS uses controllers & scopes whereas the later frameworks which now uses components. There are some developers that continue to use AngularJS (Angular 1) and this is completely different from Angular (i.e. angular 2 and above). Since Angular 4, there is now a 6 month release cycle for each new version of the framework.

Setup & File Structure

Environment Setup

NodeJS and Node Package Manager (NPM) can both be downloaded onto your machine by following the link below:
https://nodejs.org/en/

Once installed, if you are on a windows, you may wish to also install git bash (*this is not mandatory but is highly recommended terminal over the default windows powershell terminal*) from the link below:
https://git-scm.com/downloads

Finally, you would want to install a code editor such as Visual Studio Code, Atom or Sublime Editor (*the preference is yours*). Some useful extensions for Visual Studio Code to install are:

- Angular v7 Snippets by John Papa
- Bracket Pair Colorizer by CoenraadS

To toggle the display of the terminal within VS Code press both the **ctrl** + **~** (~ = *tilda*) keys on your keyboard.

We should now have the necessary tools installed in order to write code within our environment and we should be ready to start creating our very first Angular application projects using the Angular CLI tool.

Setup & File Structure

Angular CLI

Below is the webpage and GitHub page for the Angular CLI documentation on the things that you can do using the Angular CLI tool:

https://cli.angular.io/

https://github.com/angular/angular-cli

The first step is to install the Angular CLI tool globally within the terminal. To check that we have node and npm installed on our machines we can use the following commands:

```
$ node -v
$ npm -v
```

To install the Angular CLI tool globally run the following command within the terminal:

```
$ npm install -g @anglular/cli
```

Once installed we can now run the Angular CLI tool commands anywhere within the terminal as this should be now installed globally on your machine. You can run the following command to see if it installed correctly:

```
$ ng version
```

Navigate within the terminal to the folder/directory that you wish to create your Angular App using the Angular CLI tool we installed (alternatively create the folder and open it in terminal to quickly navigate to the folder/directory path). Run the following command replacing the AngularAppName with the name of your Angular App you wish to create:

```
$ ng new AngularAppName
```

The Angular CLI will ask a few questions before installing the various packages and setting up your angular application such as:

- Would you like to add Angular routing? (y/N) — answer y
- Which stylesheet format would you like to use? (use arrow keys) — choice is your's CSS, SCSS or SASS.

After answering these questions the CLI should install all the packages/dependencies required and setup your application files and folders/directories.

If we open this project folder in VS Code we can open up the integrated terminal and run some Angular CLI commands within our project directory such as:

- To view all the commands and flags in the CLI tool:

```
$ ng help
```

- To create the production/build assets of your application:

```
$ ng build
```

- Run end to end (e2e) testing:

```
$ ng e2e
```

- Builds your code on local server with auto-reload to view your application in the browser on localhost:4200:

```
$ ng serve --open
```

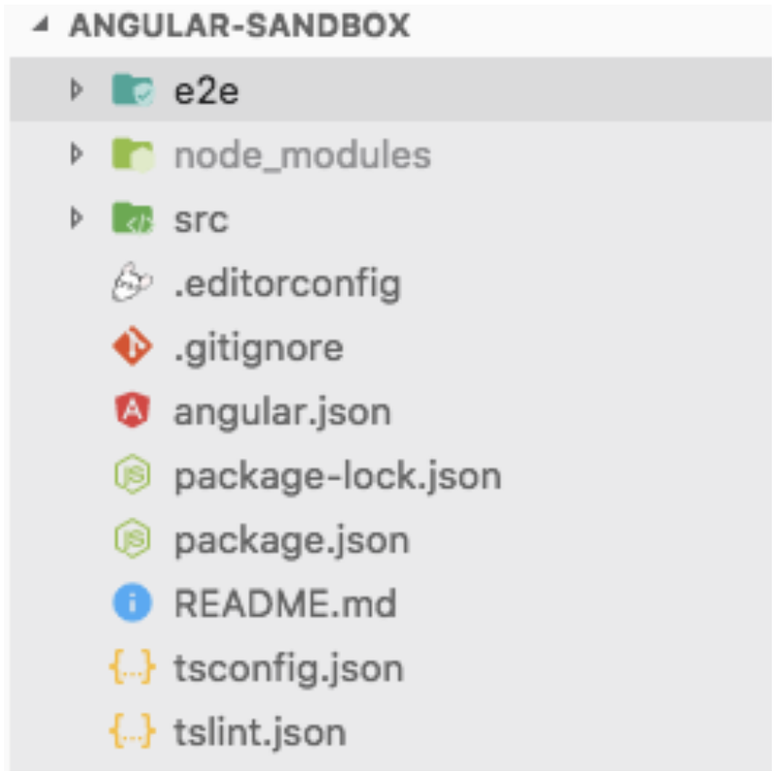
We now have a new boilerplate Angular Application installed on our machine which we can now edit and write our own Angular code to build up our own application.

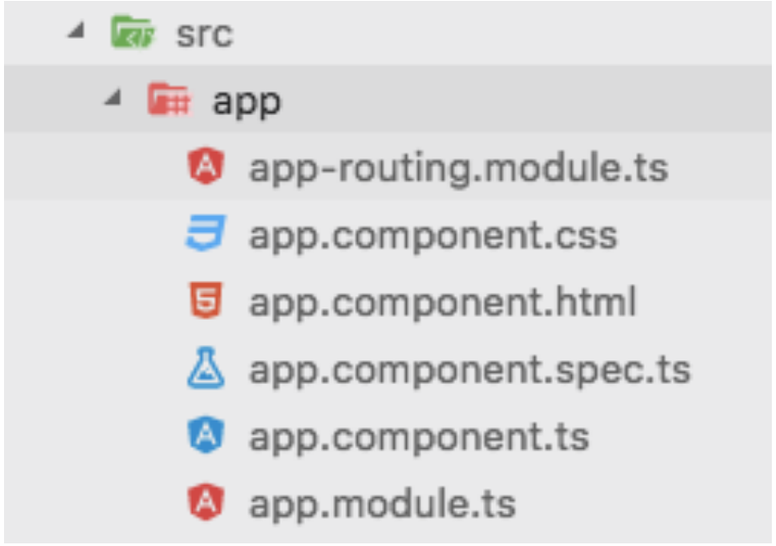
You may also wish to install Augury extension for your chrome browser to view your Angular App component state/ properties within your browser i.e. a dev tool similar to react dev tool but for Angular.
https://chrome.google.com/webstore/detail/augury/elgalmkoelokbchhkhacckoklkejnhcd?hl=en

Setup & File Structure

Angular File Structure

Now that we have created the base application file using the Angular CLI, we are now going to dive into the framework and directory structure of our Angular Application.





- The e2e folder contains the end to end testing files.
- The node_modules folder contains all the node packages/dependencies required by the Angular app.
- Within the src folder there are many subfolders. The app folder is basically your Angular application and will contain all the components, services, models and anything else to do for the application. The src folder also holds other files such as the index.html, styles.css, main.ts etc (*main.ts is the same as main.js but a TypeScript file*).
- Most of the other files are for development, testing and compiling etc. these are things that are not directly related to writing your Angular application. They are more related to your development environment in general.
- The package.json is the most important file which holds the manifest of all your application information and dependencies and npm scripts.
- The tsconfig.json files is the configuration for TypeScript which compiles our TypeScript down to ES5 JavaScript syntax which is supported by all browsers.
- The karma.config.js is the configuration file for testing.
- The .angular-cli.json file is the configuration file for the CLI.

Note: by default any new files created using the Angular CLI will have an indentation of 2 spaces. If we want to make changes to this for example, we would want 3 space indentation, we would need to update the .editorconfig file which is located at the root of the project directory.

.editconfig File:

```
# Editor configuration, see https://editorconfig.org
root = true

[*] charset = utf-8
indent_style = space
indent_size = 2
insert_final_newline = true

[*.md]
max_line_lenght = off
trim_trailing_whitespace = false
```

We could update the indent_size to 3 and inert_final_newline to false. This will ensure any new files created using the Angular CLI will follow the editor config rules setup (this is all optional). Install the EditorConfig for VS Code by EditorConfig extension in visual studio code. This will ensure the edtorconfig rules will be applied to all new files created within the project. Note: files created using the Angular CLI (*or any CLI tool*) will ignore the editorcongif rules but we could go into the files created and press ctrl + shift + p on the keyboard to use the Reindent Lines function within VS Code to reindent the code to the editor config rules.

Setup & File Structure

Setting up SSH & Github

SSH stands for Secure Shell and allows a secure way for two machines to communicate with each other. In order to make a secure connection, we would need to setup a SSH keys. In order to set this up, we would need to go into the terminal (*Windows would need to use git bash*) and enter the following commands. We can learn more on SSH Setup with GutHub on the link below:

https://help.github.com/enterprise/2.15/user/articles/connecting-to-github-with-ssh/

To check for existing keys on your machine enter the command:

$ ls -a ~/.ssh

This will check the /users/username/.ssh to find any such file or directory. If the terminal returns “No such file or directory” then this would mean that we do not have any keys on our machine and would need to create some. If we see id_rsa or id_rsa.pub in the terminal then this would mean we already have existing keys.

To create a SSH key we would use the following command:

$ ssh-keygen -t rsa -b 4096 -C “email@email.com”

This will create a private key which we will keep on our machine and a public key that we will give out to third party services such as GitHub. The rsa is the most popular SSH key and the bigger the bits the better the security (4096 is the recommended size for most services). The email is associated with the key pair. Once we run the command it is going to ask for some information such as:

- What we would like to call the file — it is recommended to stick with the default of id_rsa
- Enter passphrase & Confirm passphrase — optional.

Both the private and public keys will be saved within the directory /user/username/.ssh/ and we can start using them in a meaningful way.

If we were to run the previous command to check for existing keys on the machine, it should return the two files created i.e. the id_rsa (private key) and the id_rsa.pub (public key) as the output.

The private key should be kept private and never given out to any third party services. We should treat this file as a password. If someone was to access the private key, this would allow others to steal our machine identity and trick another machine into thinking that they were us.

The next command we would run is going to make sure that when we try to communicate with another service, such as GitHub, it will know which SSH key to use.

$ eval “$(ssh-agent -s)”

This will check whether ssh-agent is running, if it is not running, it will start things up and will display the Agent process id (pid).

The last command is to add the private key to the Agent:

$ ssh-add ~/.ssh/id_rsa

Once the identity has been added, we are now ready to actually take the public key file and provide that to the third party services such as GitHub. The following command will copy the public key to the clipboard (this is specific to the OS you are running - below is for mac):

$ pbcopy < ~/.ssh/id_rsa.pub

For the other operating system view the guide on:

https://help.github.com/enterprise/2.15/user/articles/adding-a-new-ssh-key-to-your-github-account/

Once copied you can go into your GitHub account into settings and within the SSH and GPG keys tab you can add a new SSH Key. You can add a title and the key and you should be able to use SSH with GitHub.

We can run the following command to check if the SSH key was set up correctly with GitHub:
$ ssh -T git@github.com
It will ask a question if we are sure that we want to continue connecting with the server and we can answer yes and let the command run. This will make a very basic ssh connection to the server and this is either going to work if the key was setup correctly or fail is something did not get setup correctly.

You should see the message if correctly setup, returned in the terminal:
Hi GitHubUsername! You've successfully authenticated, but GitHub does not provide shell access.

We can now push our files up to GitHub using a secure SSH connection compared to the standard http requests.

Setup & File Structure

An Intro to TypeScript

TypeScript is not mandatory in order to create Angular applications, however, there are many benefits for using it. TypeScript is a superset of JavaScript created by Microsoft and it provides the same functionality of JavaScript plus additional benefits, similar to how SASS is for CSS. TypeScript is compiled down to regular JavaScript code.

Some of the features it provides are:

- Static Typing
- Object Classes
- Modules
- let/const scoping
- Other ES6 features

TypeScript files use the .ts extension and are compiled to regular .js extension files using the TSC (TypeScript compiler) which is included in the Angular CLI.

What is Static Type Checking?

Vanilla JavaScript is dynamically typed meaning we do not need to declare the variable types or define the function return values and we can change variable types for example string to numbers etc. This can be seen as both a good and bad thing i.e. good because you have less code to write but bad because the code is less organised and more prone to errors.

TypeScript allows us to define the type of variables or the values, however, this is completely optional and you do not need to use static typing with TypeScript or Angular as it is completely up to the developer.

Below is a quick example of what is possible with TypeScript:

```
let name: string = ‘John Doe’;

function addNumbers(num1: number, num2: number): number {
      return num1 + num2;
}
```

We use the colon (:) followed by the type to declare the variable or value type. What this allows for, if we try to pass in a value that is not the same type for example we declared a string and passed in a number, this will give us an error when we try to run/compile our code. This makes the code more robust and less prone to errors.

What are the available Types?

String, Number, Boolean, Array, Void, Null, Tuple (ordered list/array) and Any.

Object Based Classes:

Below is a basic example of a Class Object using TypeScript.

```
Class Greeter {
      greeting: string;
      constructor(message: string) {
            this.greeting = message;
      }
      greet( ) {
```

```
            return “Hello, ” + this.greeting;
        }
}

let greeter = new Greeter(“world”);
```

In the above we have a class called Greeter which has a property (*attribute*) called greeting which has a type of string. We have two methods within the class (*a method is a function within a class*) called constructor and greet. The constructor is a special type of method which runs when the object is instantiated/initialised. The message value passed into the constructor is assigned to the greeting property of the class using the this. keyword to set the greeting value. The greet method returns Hello followed by whatever the greeting variable value is.
Finally, the last line of code initialise/instantiates the object using the greeter class, passing in the string world which gets passed into the constructor to set the greeting variable value (the constructor runs as soon as we instantiate the class object). This will print out Hello world if we ran the .greet( ) method on the greeter object.

As you can see, this is very similar to a ES6 JavaScript classes and it works just like a class in any other programming languages such as Java, C#, Python, PHP etc.

This is a very brief introduction to TypeScript and what is possible using it. We will explore more of TypeScript in the following sections as we learn the basics of Angular and writing our own Angular code.

Diving Into Components

Components Explained

Components are the foundation of most Angular applications, therefore, it is very important to understand how components work and how they are structured.

Angular apps have a common structure which are made up of a few different entities including modules and services, however, components make up the bulk of the application. When we look at applications user interfaces for any applications, we can see pieces of the UI as their own individual component with their own functionality, properties and methods for example a navigation bar, search bar, input boxes etc. This is why framework such as Angular (React and Vue) are great for because they organise the code and the application itself.

Every regular component includes a class which can have properties and methods, method being a function inside of a class. Components also have a template associated with them which is what the user actually sees in the browser. We can bind data from the component class to the template and vice versa. This is what makes angular so dynamic.

Most Angular applications will have something called a root app component, and all the components we create will be nested inside of that root component.

Why use components?

It provides code organisation and allows us to break up the user interface (UI) so that we can encapsulate the functionality with the properties and methods of each component. Components promotes reusability and stops us from having to repeat ourselves within our code. We are also able to reuse components across multiple different applications in the future without having to re-write the code again. Finally, it also helps with better teamwork because it allows other programmers to read our code and know exactly what is going on within our code.

Structure of a Component:

Below is an example code of the Root App Component.

TypeSript Component:

```
Import { Component } from '@angular/core';

@Component ( {
      selector: 'my-app',
      template: ` <h1> Hello { {name} } <h1> ` ,
} )

export class AppComponent { name = 'Angular'; }
```

HTM Template :

```
<body>
      <my-app>Loading AppComponent content here… </my-app>
</body>
```

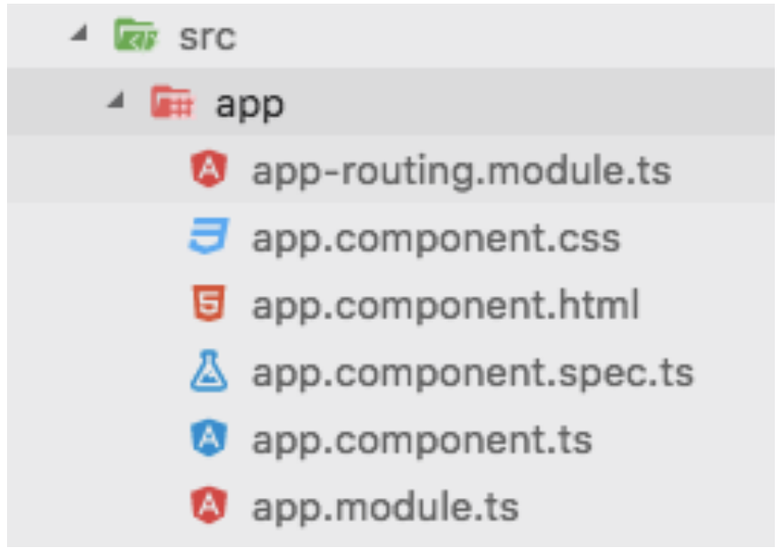
The first line is importing in the Component module from the angular/core package. A component is really just a class with properties and methods. We have a class called AppComponent which has one property called name which is set to a string called Angular. In addition to the class we have something called a decorator and this is what the @Component code is. This is a decorator that can add meta data or special information to the component which is part of TypeScript. The selector is part of the decorator and is responsible for what is going to be displayed inside the HTML tag/template to display the component. This is how the template knows which component to display.

Within the decorator we also have the template which has a template literal for containing <h1> tags followed by Hello and {{name}}. The name in the template is accessing the name property within the class using something called String interpellation. We can access properties and methods from our class and bind data and all sorts of things that relate the class to the template decorator.

If we were to look at this component within the browser, we would see a <h1> element of Hello Angular.

Generating Components with Angular CLI:

If we look at the Angular App created using the Angular CLI tool we would notice 4 files: component.html, component.css, component.spec.ts and component.ts



The .hmtl is the template, the .css is the styling, the spec.ts is the testing and the .ts is the main component file. We can either create these files on our own or use the Angular CLI tool to generate components for us using the following command in the terminal:

$ ng g component components/component-name

$ ng g c components/component-name

We can specify the folder to place the files in and the name of the component file by replacing the the component/component-name with our desired location/name (*it is recommended to place all components within a components folder for better organisation*). The component is then added to the @ngModule declarations so that Angular knows where to actually look for that component.

Diving Into Components

Creating a Basic Component

Now that we have an overview of Components i.e. what they are and why we use them, we are going to explore creating a custom component in Angular.

The first thing we would want to do is to create a components directory within our app folder. Within this new components directory we can create separate component folders (named after the component) that will be nested inside the components folder. This folder will contain the files that would make up our component.

Example setup:

src > app > components > componentName (e.g. user) >

- componentName.component.ts
- componentName.component.html
- componentName.component.css
- componentName.component.spec.ts

This is the general format you would want to use for the naming convention of your component files for example if we are making a user component we would name it user.component.ts as the file name.

Taking user.component.ts as our example, the first thing we would want to do within this file is to import Component from angular/core. We then want to create a decorator (decorator is part of TypeScript which allows us to add extra information onto our class/component) and then finally create our class and export it. The component syntax/code will look something like the below example:

```
Import { Component } from ‘@angular/core’;

@Component( {
      selector: ‘app-user’,
      template: ‘ <h2> John Doe </h2> ’
} )

export class UserComponent {

}
```

In the selector we use app- in front of the component’s name to follow the convention of the Angular CLI. We use template in our decorator to add the template within the .ts file rather than creating a separate template file (*this is not good practice and we should use templateURL property to like to a separate component.html file that will store our HTML template*).

The class name uses the pascal case syntax i.e. the first letter is uppercase and then every other word first letter is uppercase. We use export to make sure that our class is accessible outside this file.

When we create a component, we must add it to the app.module.ts file which acts as the meeting place for all the application components, services and modules etc. Within this file we would import it and add to the declarations:

```
Import { UserComponet } from ‘./components/user/user.component’;
@ngModule ( { declarations: [ AppComponent, UserComponent ], … } )
```

Finally, we need to embed the component within the app.component.html file using the custom html tag for our component:

```
<h1>Angular Sandbox</h1>
<app-user></app-user>
```

If we open up the browser for the application we should see John Doe displayed on the screen. We have created our first basic custom angular component.

Important Note: It is not good practice to put the html template directly into the component.ts file. We would usually want to have a separate component.html file for your html template. Within the component.ts file decorator we would use the templateUrl property to link the separate html template to our component.

```
templateUrl: './user.component.html'
```

If we wanted custom css we could also add this to the component.ts file within the decorator using the styles property and inlining the css style, however, again this is not good practice. Instead we should create a separate component.css file and then within our component.ts file decorator we would use the StyleUrls property to link our css file. This property takes in an array because we could have multiple css style files that we would want to link.

```
styleUrls: ['./user.component.css']
```

We have now analysed how to create a basic Angular component with a template and css. We are now ready to look at data binding and string interpolation with our components as well as creating properties and methods for our class component.

Diving Into Components

Properties & Methods

A **property** is like an attribute of a component while a **method** is a function within a component. **String interpolation** allows us to place component properties (*i.e. variables*) inside of a html template that are read as strings.

Example:

```
export class UserComponent {
    firstName = ‘John’;
    lastName = ‘Doe’;

    constructor( ) {
        console.log(‘Hello User…’);
    }
    sayHello( ) {
        console.log(`Hello ${this.firstName} `);
    }
}
```

Note: constructor in object oriented programming (OOP) are a special function that runs automatically when the object is instantiated. In the above case it is a component and so the constructor automatically runs when the component initialises.

The constructor is usually used to inject dependencies e.g. bringing in a data-service we would inject it within the brackets ( ) as arguments/parameters.

We can also create our own methods. Using back ticks we can use template literals to inject the component properties within the method as seen above. To access the component properties we need to use the .this keyword followed by the property name that we would like to access. The `${ }` syntax is from ES6 which is pure vanilla JavaScript and has nothing to do with Angular and is a much simple and clean syntax compared to ES5 concatenation and quotes syntax — Example of the difference demonstrated below:

```
`Hello ${this.firstName}, welcome back!`
'Hello ' + this.firstName + ', welcome back!'
```

To use the custom method inside of the constructor we would also need to use the this keyword followed by the method name as seen below:

```
constructor ( ) {
      this.sayHello( );
}
```

We should now have a basic understanding of properties and methods within an Angular Component and can now look at data-binding and string interpolation.

Diving Into Components

String Interpolation

There are different ways to bind the data from our component to the html template; **string interpolation** being one of such methods. To create a string interpolation we would use the double curly brackets and pass in any JavaScript expressions that can evaluate to a string within the curly brackets. We can also use numbers and math because numbers can also evaluate (*convert*) into a string.

String interpolation syntax example within a component.html file:

```
{{ 'Hello' }}
{{ 20 }}
{{ 1 + 1 }}
```

Generally, string interpolation is used to output properties from components i.e. it allows us to have variables inside of our html templates which makes our html more dynamic and powerful. For example:

```
<h1> {{ firstName }} {{ lastName }} </h1>
```

Important Note: even if we use TypeScript to type a variable as a number, we can still use the number variable value inside of string interpolation.

We can also create object literals as properties within our component and output these within our html template using string interpolation as seen in the example below:

.component.ts File:

```
export class UserComponent {
    address = {
        street: '1 Abbey Street',
        city: 'Birmingham',
        country: 'UK'
    };
}
```

.component.html File:

```
<li>Address: { { address.street } } { { address.city } } { { address.country } }</li>
```

Again object literals are basic vanilla JavaScript and has nothing to do with Angular.

We can also use string interpolation with method return values as seen in the below example:

.component.ts File:

```
showAge( ) {
    return this.age
}
```

.component.html File:

```
{{ showAge( ) }}
```

Diving Into Components

Using Types

Types is a big part of TypeScript and the reason why Angular uses TypeScript. Types makes the code more robust and less prone to errors. So far we have seen properties defined by the values assigned, but this is not always the case i.e. sometimes we would declare properties without any assigned values.

In the below example the firstName variable is assigned a string value and therefore it is seen as a type of string. If we were to re-assign the firstName variable to a number (*for example 4*) the VS Code console compiler would give us an error of: *Type '4' is not assignable to type 'string'* — this would still render the component to the screen.

```
firstName = 'John';
```

However, if we were not to assign firstName a value then this would be seen as a type of any and we would not see the VS Code compiler error of the above when we assign the number to the variable.

```
firstName;
```

Usually, you would want to assign a type to the property. In order to do this we would use the colon followed by the type we want the property to be — for example:

```
firstName: string;
```

If we now try to assign the firstName property with a number value such as 4, this will provide us the VS Code compiler error to indicate that we have tried to use a number for a variable that was expecting a string.

Important Note: JavaScript is not a type language and allows the assigning of any values to variables. TypeScript allows us to have the benefit of using Type with our variables and be notified of errors in our code where a variable receives a different type value than what the variable is expected to have.

Example Type Code:

```
export class UserComponent {
    firstName: string;
    lastName: string;
    age: number;
    address;

    constructor( ) {
        this.firstName = ‘John’,
        this.lastName = ‘Doe’,
        this.age = 28,
        this.address = {
            street: ‘1 Abbey Street’,
            city: ‘Birmingham’,
            country: ‘UK’
        }
    }
}
```

There are different types within TypeScript which we will analyse within the below table which will provide the type, description and an example code to go along with it.

Type	Description	Example Code
Any	As the name suggest this allows for any of the types below to be assigned to the variable.	foo: any; foo = true;
Boolean	These variables can only be assigned boolean values which are true or false	hasKids: boolean; hasKids = false;
Array	The variable will hold an array. We can indicate the type of array using the type name.	mixedArray: any[ ]; mixedArray = [1, ‘a’, true, null] numArray: number[ ]; numArray = [1, 2, 3]; strArray: string[ ]; strArray = [‘a’, ‘b’, ‘c’]
Tuple	This will hold a defined/mapped array. Anything after the defined arrays can be of any type within the tuple array.	tuple: [string, number, boolean]; tuple = [‘a’, 1, false]
Void	As the name suggest the variable can only be assigned void; however, this will throw an error and the only thing you can assign it is undefined.	v: void; v = undefined;
Undefined	As the name suggest the variable can only be assigned undefined.	u: undefined; u = undefined;
Null	As the name suggest the variable can only be assigned null.	n: null; n = null;

We can assign type to function/method parameters and return values. In the below example we have assigned type for each parameter as well as returned value.

```
addNumber(num1: number, num2: number): number {
    return num1 + num2
}
```

In the above function we can only assign numbers and if we try to assign anything other than a number type this will throw an error in the compiler console when we come to compile our Angular application. This will notify us of any errors in the code where we were expecting a type but received a different type which helps ensuring our code is robust and less prone to errors.

We can assign type to function/method parameters and return values. In the below example we have assigned type for each parameter as well as returned value.

```
addNumber(num1: number, num2: number): number {
    return num1 + num2
}
```

In the above function we can only assign numbers and if we try to assign anything other than a number type this will throw an error in the compiler console when we come to compile our Angular application. This will notify us of any errors in the code where we were expecting a type but received a different type which helps ensuring our code is robust and less prone to errors.

Diving Into Components

Interfaces

Interfaces are like models for your data. If we take the below User class component and follow the convention we have come to know of setting up the properties and types, our code will look something like the below:

```
export class User {
    user:  {
        firstName: string,
        lastName: string,
        age: number,
        address: {
            city: string,
            country: string
        }
    }
    constructor( ) {
        this.user = {
            firstName = ‘John’, lastName: ‘Doe’, age: 28, …
        }
    }
}
```

The class component code looks rather complicated and messy. We are able to use interfaces to model the data and

tidy up the above class component. To do this we would use the interface keyword followed by the interface name. We can model the data and then use that interface within the component. Below is an example of how the User data can be modelled using an interface:

.component.ts File:

```
export class User {
    user: User;
    constructor( ) {
        this.user = { … }
    }
}

Interface User {
    firstName: string,
    lastName: string,
    age?: number,
    address?: {
        city?: string,
        country?: string
    }
}
```

As you can see we have created an interface for the User data and modelled the properties and type for each of the properties. We can make some properties optional by adding the ? next to the optional property name(s).

We can then reference the interface within the component class which tidies us the above code. In most cases, we would use the the interface data (*User or any other interface data*) within other components, services or somewhere else and therefore, instead of repeating the interface over and over again, we would want to hold the interface in a separate file and reference it within the component, service or any other files that require the interface data model.

Within the App folder we would want to create a new directory called models. This folder will hold all of our application interface files.

File Structure Example:

src > app > models >

- InterfaceName.ts (*e.g. User.ts*)

We would need to export the interface file in order for the interface file to be accessible to other files within our application. We can make this a default or named export by using either export default to export keywords in front of the interface keyword:

export interface User { … }

We would then need to import the interface (*named export since we did not use default export above*) into our component.ts file in order to use the interface file within our component file as we did above:

import { User } from '../../models/User';

This allows us to break up our code for better organisation, readability, maintainability and reusability of our application code. To conclude, we can see that interfaces are very simple as they are simply a mapped object of the data model.

Diving Into Components

Generating Components & OnInit

So far we have seen how we could generate the components files manually; however, we could easily generate the component files using the Angular CLI. We can use either the integrated VS Code terminal or the OS terminal to navigate to our project directory and run the following command:

$ ng generate component components/componentName --spec=false

Abbreviated version:

$ ng g c components/componentName --spec=false

To save the new component files within the components directory (*or any other directory should your file structure differ*) we can specify the folder/path name, in the above we specified components/ as our path. Replace componentsName for the name of the new component e.g. users.
This will create a new users folder within the components directory and this users folder will hold the 4 component files (*component.ts, component.html, component.css and component.spec.ts*). Adding the --spec=false flag at the end of the command will not create the .spec.ts test file.

This command also updates the app.module.ts file for us automatically by importing the new component file and adding it to the @ngModule declaration array. Therefore, the Angular CLI tool has done all the setup for us and there is nothing else required for us to setup. If we open the component,html file we would see a simple <p> element as the template with the componentName works! The component.css is an empty file and finally, the component.ts file has a basic templated/shell for us to work with which contains all the code we would have wrote manually.

One thing to notice with the component.ts file is that we now have implements OnInit after the class component name

and a new ngOnInit method within our component class as seen in the example below:

```
export class UsersComponent implements OnInit {
        constructor( ) {
        }
        ngOnInit( ) {
        }
}
```

The OnInit is a lifecycle method which runs automatically when the component is initialised just like how the constructor does when the component initialises. This is the lifecycle method and we would commonly use it if we were making an AJAX call through a service and we wanted to fill/assign the properties with the data requested. The constructor method on the other hand is more commonly used for dependencies injection (*i.e. to bring in services etc.*). We could use the constructor method to assign property values; however, using the OnInit method is the recommended common standard practice because that is the intended purpose of the OnInit lifecycle method.

Finally, within the **app.component.html** file we would need to manually add the new users component element tag for it to display within our application, because the CLI tools does not add this for us:
`<app-users></app-users>`

We now have a component created for us within seconds using the Angular CLI tool which is much easier than creating all the files ourselves manually.

To conclude, we have analysed and gone through all of the basics of Angular components and we should be comfortable in creating new components for our Angular Applications whether manually or using the Angular CLI tool.

Template Syntax

Loops with ngFor

The ngFor allows us to loop through an array and dynamically output the values to the template. The below is an example of using the ngFor loop syntax:

.component.ts File:

```
import { User } from '../../models/Users';
    export class UsersComponent implements OnInit {
    users: Users[ ];
    ngOnInit( ) {
        this.users = [
            {
                firstName: 'John',
                lastName: 'Doe'
            },
            {
                firstName: 'Caroline',
                lastName: 'Dova'
            }
        ]
    }
}
```

We need a way to loop through the users array and output the values within the template file. This is where we would use ngFor to loop to output each user within the users array within our template file.

.component.html File:

```
<h2>Users</h2>
<ul>
    <li *ngFor="let user of users">
        {{ user.firstName }} {{ user.lastName }}
    </li>
</ul>
```

Within our element we use what is called a ng directive which is like a html attribute. Adding the asterisk before the directive is what is known as a structural directive. Anything that has to do with the structure and control of the output such as ngFor, ngIf, ngSwitch, etc. requires an asterisk before it.

In the above ngFor structural directive we assign a let variable called user and we loop through the users array from the component.ts file and assign the values in each iteration to the let user variable. Within the list item element we can use the double curly brackets to string interpolate the list item data we wish to dynamically add to the element, in our case the firstName and lastName of the user.

Note: we could create a method within our component.ts file which can push data onto an existing array using the JavaScript push method:

```
addUser(user: User) {
    this.users.push(user);
}
```

Template Syntax

Conditionals with ngIf

The ngIf structural directive allows us to add conditionals to our html templates. In the below example, we can follow on from the previous ngFor users list and add a conditional using the ngIf structural directive to demonstrate the syntax.

.component.ts File:

```
export class UsersComponent implements OnInit {
    users: Users[ ];
    showExtended: boolean = true;
```

.component.html File:

```
<ul>
    <li *ngFor="let user of users">
        { { user.firstName } } { { user.lastName } }
        <ul *ngIf="showExtended">
            <li>Age { { user.age } }</li>
            <li>Address { { user.address.street } } { { user.address.city } } { { user.address.country } }</li>
        </ul>
    </li>
</ul>
```

This conditional will display the list items for Age and Address if the showExtended evaluates to true. In the above

case this is true, however, if we changed the showExtended value from true to false, this will result in the list items for Age and Address not being displayed within the template.

Note: we can use many ngIf directives within our templates to dynamically display different html elements based on where the if statement evaluates to true. For example, we can look at the users array and if the users array is above 0 we can display a certain html elements and have another elements displayed if the users array evaluates to 0:

```
<ul *ngIf="users.length >0">
    <li *ngFor="let user of users">
        { { user.firstName } } { { user.lastName } }
        <ul *ngIf="showExtended">
            <li>Age { { user.age } }</li>
            <li>Address { { user.address.street } } { { user.address.city } } { { user.address.country } }</li>
        </ul>
    </li>
</ul>

<h4 *ngIf="users.length == 0">No Users Found</h4>
```

In the above if there were no users within the users array then only the <h4> element will be displayed within the browser and if users did exist then the <h4> will not be displayed but the above list elements will be displayed as the conditional if evaluated to true.

Angular 4 introduced the ng-template which is like using an if else statement within our template. Example below:

```
<ul *ngIf="users.length >0; else noUsers">
      <li *ngFor="let user of users">
            { { user.firstName } } { { user.lastName } }
      </li>
</ul>

<ng-template #noUsers>No Users Found</ng-template>
```

The noUsers relates to a template name (*this can be called anything we want*) and this template will display for the else statement when the if statement evaluates as false. We would use a ng-template tag using the hash # to give an id name. This id name would be the same name as defined in the else statement.

Either way of using the ngIf statements would work and it is of personal preference whether you us the ng-template along with else statements or you create each scenario case elements ad add an ngIf statement to display the elements if it evaluates to true.

We can also use JavaScript logical operators within our if statements as we would do normally (*such as +, -, &&, | |*) for example:

```
<ul *ngIf="loaded && users.length >0"> … </ul>
```

In the above example we have two conditional statements where both need to evaluate to true in order to display the unordered list items. We could use setTimeout( ) to mimic a AJAX request for the users. If this is the case we would need to use the users? because the users would not exists but we are checking users until it has been fetched or not.

```
<ul *ngIf="loaded && users?.length >0"> … </ul>
<h4 *ngIf="users? == 0">No Users Found</h4>
```

Template Syntax

Adding Bootstrap

Bootstrap is a front end css framework which makes the styling of applications look better. We could have added the CDN within the index.html file, however, this is not the most elegant way of doing things within Angular.

The first step would be to install Bootstrap using the terminal command. Note: we would need to install its dependencies, jQuery and popper.js as well:

```
$ npm install bootstrap@4.1.3 jquerry popper.js
```

To prevent any updates to any later versions, you could go into the package.json file and remove the carrot ^ symbol before the version number. This will ensure the installed version will never be updated to a later version:

```
“bootstrap”: ”^4.1.3”
“bootstrap”: ”4.1.3”
```

We would then want to open up the angular.json file and add our style scripts to the existing styles.css:

```
“styles”: [
      “src/styles.css”,
      “node_modules/bootstrap/dist/css/bootstrap.min.css”
]
```

Within the same file (angular.json) below the styles area add to the scripts array:

```
“scripts”: [
      “node_modules/jquery/dist/jquery.min.js”
      “node_modules/popper.js/dist/umd/popper.min.js”
      “node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

Once you have made the changes and saved the angular.json file, you would need to restart the server incase it was still running when the changes were made. This will ensure the changes take affect after we restart the server with ng serve —open command. We can now use bootstrap within our Angular application.

When the application starts up, we would notice that the application would look slightly different. This is because Bootstrap is now being taken into account and the default Bootstrap styling are now being applied (*the font styling being the noticeable change*).

We can now use Bootstrap classes to style the angular application. Below is an example Bootstrap classes creating a simple navigation bar element:

```
<nav class=“navbar navbar-dark bg-primary mb-4”>
      <div class=“container”>
            <a href=“#” class=“navbar-brand”>Angular Sandbox</a>
      </div>
</nav>
```

Template Syntax

Property Binding

Property binding allows us to have dynamic attributes within our templates and component files. To add property binding we use an opening and closing square brackets [ ] and the property/attribute we want to bind from our component. Below is an example of property binding to a button.

.component.ts File:

```
export class UsersComponent implements OnInit {
    enableAdd: boolean = true;
}
```

.component.html File:

```
<button [disabled]="!enableAdd">Add New User</button>
```

In this example we are disabling a button based on the property from the component. If the enableAdd attribute (property) is set to false, this will disable the button else the button will be enabled if the value is true. This demonstrates that we are able to manipulate our template element attributes using the component properties values which makes our templates more dynamic.

Below is another example of property binding using the square brackets. In this example we are binding a user image url from within the component file and displaying the output within our template file as an image element with a src attribute.

.component.ts File:

```
export class UsersComponent implements OnInit {
    ...
    ngOnInit( ) {
        this.user = {
            image: ‘http://lorempixel.com/600/600/people/1'
        }
    }
}
```

.component.html File:

```
<img [src]=“user.image” alt=“user profile”></img>
```

We have indicated that the src attribute is a property binding attribute and it gets its value from the component.ts file.

There are other ways to bind property to templates, however, using the square bracket is the main syntax. An alternative way to bind property is to use the two curly brackets (string interpolation) as demonstrated below:

```
<img src=“{ { user.image } }” alt=“user profile”></img>
```

The final alternative method for property binding is to use the bind— keyword followed by the attribute you wish to bind and then set the value to the component property, as demonstrated below:

```
<img bind-src=“user.image” alt=“user profile”></img>
```

To conclude there are three methods for property binding, although the first method is recommended.

Template Syntax

Class Binding & ngClass

Class binding is where we add or use certain classes depending on certain properties, values or certain aspects of the component. We can dynamically set a class on our template as seen in the example below:

User.ts File:

```
export interface User {
    isActive?: boolean
}
```

.component.ts File:

```
export class UsersComponent implements OnInit {
    ngOnInit( ) {
        this.users = { …, isActive: true }
    }
}
```

.component.html File:

```
<li [class.bg-light]="user.isActive"></li>
```

In the above example we are setting a class of bg-light (a bootstrap class) if the property isActive is true i.e. the user is active. Therefore, within our users.component.ts file we can add the isActive property to an object which will dynamically style the html element within the template by applying the bg-light class.

This is very powerful because it allows us to connect the style of the UI to the values and the functionality or the state of the application/component. The ngClass on the other hand allows us to bind one or more classes to an element. For example:

.component.html File:

```
<button [ngClass]="currentClasses">Add New User</button>
```

.component.ts File:

```
export class UsersComponent implements OnInit {
    enableAdd: boolean = true;
    currentClasses = { };

    ngOnInit( ) { this.setCurrentClasses( ) }

    setCurrentClasses( ) {
        this.currentClasses = {
            "btn-success": this.enableAdd
        }
    }
}
```

The button will dynamically have the class of "btn-sucess" applied to it if the this.enableAdd property is true. The function must be called and this can be done via the constructor( ) or the ngOnInit( ) methods. Note: we can add more classes if we want and are not limited to one class. We can also target the class via our css to custom class styles.

Template Syntax

Style Binding & ngStyle

Style binding and ngStyle is very similar to class binding and ngClass. This allows us to add styles to our template elements. Below is an example code:

.component.html File:

```
<li [style.border-color]="user.isActive ? 'green' : '' "></li>
```

In this example we are adding an inline css styling to the template element using the style binding. We use the style. keyword followed by the css style within square brackets and then use a ternary operator to check whether the component property evaluates to true (*in this case isActive*). If true then the inline style will be set with a attribute of green else there will be no value applied to the border-color. This allows us to dynamically add inline styles to our templates.

The ngStyle (similar to the ngClass) allows us to add one or more css styles to our templates. Example below:

.component.html File:

```
<h3 [ngStyle]="currentStyles">{ { user.firstName } }</h3>
```

.component.ts File:

```
export class UsersComponent implements OnInit {
    showExtended = false;
    currentStyles = { };
```

```
        ngOnInit( ) {
            this.setCurrentStyles( )
        }

        setCurrentStyles( ) {
            this.currentStyles = {
                “font-weight”: this.showExtended ? ‘bold’ : ‘’,
                “font-color”: this.showExtended ? ‘’ : ‘red’
            }
        }
}
```

The principal to the syntax is the same as the ngClass. Within the template file we use the keyword [ngStyle] and add the attribute to a component property. Within the component we add the property and set it to an empty object. We then create a method that would add the css style(s) to the empty component property object we created. We can use the ternary operator to add conditionals for the styles to be applied. Finally, we need to instantiate the method either via the constructor( ) or ngOnInit( ) methods. This will add CSS styling dynamically to the template based on the conditions matched. Note: leaving a style as an empty string will use the default style.

To conclude, the Style binding and ngStyle is very similar to the class binding and ngClass syntax and this allows us to dynamically bind inline css style attributes to our templates based on our component properties.

Template Syntax

Pipes & ngNonBindable

Pipes are a way to format data within our templates. There are some pre-made pipes available within Angular by default; however, we can also make our own pipes. To use a pipe we would go into the .component.html file and after bringing in a data from our component we would use the pipe | character followed by the pipe name we wish to format our data. Below are examples of the pre-made pipes available to use:

.component.html File:

```
<h1>Name: { { user.firstName | uppercase } } { { user.lastName | lowercase } } </h1>
<li>Balance: { { user.balance | currency:“GBP” } } </li>
<li> Joined: { { user.registered | date: “dd/MM/yyyy” } } { { user.registered | date: “shortTime” } } </li>
```

The uppercase pipe converts the text data to all upper case characters while the lowercase pipe converts text data to all lower case characters.

The currency pipe formats the data to look like a currency i.e. it adds two decimal places and the dollar sign $ by default. To add a property we use the colon : followed by the property, in our case we formatted it to be GB Pounds £ rather than the American dollar. Note: if we add another property of code this will display the code rather than the symbol or we could add the property symbol to show the symbol (*e.g. currency:“GBP”:“symbol”*).

The date pipe will format the JavaScript date data into a more presentable date format (*by default this is in the American format of “mmm dd yyyy”*). To format the date we would use the colon followed by the format type property (*e.g. dd/mm/yyyy, yyyy, mmm, shortDate, longDate, fullDate etc.*). We can also format the time e.g. shortTime, longTime etc. Note you cannot format date and time using just the one property binding. You would need to do the

above example and use two property binding and format one for the date and the other for the time, in order to display the date and time within the template using the pipe to format the data.

There is a pipe for numbers, for example we can use a pipe for formatting a number to two decimal places using the number pipe, as demonstrated the below:

```
<h4>{ { 5 |  number:“1.2” } } </h4>
```

There is also a pipe for percentages. This will take the number and convert it to a number for example 1 will become 100% while .5 will be formatted to 50% and so on (*percent will multiply the number by 100 to get the percentage number and add the percentage % sign after the number to display as a percentage*).

```
<h4>{ { 1 |  percent } } </h4>
```

There are many default pipes within Angular and we can also create our own custom pipes.

The ngNonBindable directive allows us to attach the directive to an element and anything within the element will become non-bindable i.e. it will not bind the data to the string interpolation or any other binding methods. To make an element non-bindable we simply add the ngNonBindable as an attribute to an element. We do not need to pass any values to this directive/attribute:

```
<h4 ngNonBindable><pre>{ { 5 |  number:“1.2” } }<pre></h4>
```

This will display the data code { { 5 | number:“1.2” } } within the browser because the ngNonBindable prevents any binding of data. This is useful for blog posts where you would want to show/preview the code rather than the code being processed via the browser.

Events & Forms

Mouse Events & Manipulating State

There are different types of mouse events that are available to us and we can use these events to manipulate the application state. To add an event in angular we write the name of the event type within brackets and set the attribute to the name of the event (function name) that we want to trigger:

.component.html File:

```
<button (click)="eventName($event)"></button>
```

.component.ts File:

```
export class UsersComponent implements OnInit {
    ...
    eventName(e) {
        console.log('Event Type:' + e.type);
        console.log(e);
    }
}
```

In this example we are using the mouse click event to trigger off an event (function) called eventName. We can pass in the event within our function using the $event argument. We can then create the event within the .components.ts file and can also pass in the event object as a argument within the function (*in the above example we used e*) which will allow us to have access to the event object within the function (*i.e. everything we would normally have access to within an event object, within vanilla JavaScript*).

The other vanilla JavaScript mouse event types we have in addition to the click event are:

`<button (mouseover)="eName($event)"></button>`	When we hover over the button element.
`<button (mouseout)="eName($event)"></button>`	When we leave the button element.
`<button (mousedown)="eName($event)"></button>`	The mouse button pressed down, it immediately triggers
`<button (mouseup)="eName($event)"></button>`	The mouse button released up, it immediately triggers.
`<button (dblclick)="eName($event)"></button>`	The mouse button double clicked.
`<button (drag)="eName($event)"></button>`	When we drag/move the button element.
`<button (dragover)="eName($event)"></button>`	When we drag/move the button element over itself.

We can use the event to manipulate data for example we could call a addUser event that will push a user to an array and this will display in the template:

.component.html File:

```
<button (click)="addUser({ firstName: 'Elena', lastName: 'Gilbert', isActive: true })"></button>
```

.component.ts File:

```
addUser(user: User) {
    this.users.push(user);
}
```

The button click arguments will pass into the addUser function as the user argument which would then be used to add/push the new user to the users array object.

Events & Forms

Toggling Values with an Event

In this section we are extending our knowledge with events and how we can toggle state values within the application. The below examples demonstrates the toggle of showing/hiding user information.

Users.ts File:

```
export interface User {
    hide?: boolean
}
```

.component.ts File:

```
export class UsersComponent implements OnInit {
    ngOnInit( ) {
        this.users = [
            { ..., hide: true }
        ]
    }

    toggleHide(user: User ){
        user.hide = !user.hide;
    }
}
```

.component.html File:

```
<button (click)=“toggleHide">Toggle<button>
<ul *ngIf=“!user.hide”><li>{ { user.firstName } } { { user.lastName } }</li></ul>
```

We would only want to display the content if the user.hide value is false, hence we used the exclamation mark in front. The function is set to !user.hide to toggle the hide attribute from true to false and vice versa (i.e. opposite of the current property value).

Note: we do not need to create the function within the component.ts file for this function, as we can add the function expression directly within the template file click event:

```
<button (click)=“user.hide = !user.hide;”>Toggle<button>
```

We could use a library called font-awesome to add icons to the application. This would be a perfect case scenario to use ngClass to switch the class of the icon element to switch between a + and a — symbol when we toggle the show/hide attribute.

```
<button (click)=“user.hide = !user.hide;”><i [ngClass]=“ user.hide ? ‘fa fa-plus’ : ‘fa fa-minus’ ”]></i><button>
```

We are using a ternary operator inside of the ngClass attribute to toggle between the two classes. This will determine whether to display the + icon to toggle show or to display the — icon to toggle hide depending on the user.hide attribute is true (user info is hidden) or false (user info is displayed).

To conclude, we can add functionality to the application such as a toggle button to display or hide elements within the application with the use of events. This allows the application to be more dynamic.

Events & Forms

Form Input Events

In this section we are going to analyse the various keyboard & input events that we can utilise within the application. These events are particularly useful for web forms.

Within forms we have a submit event where we can call a function (*usually called onSubmit( ) but can be named anything we want*) which will be called when a form is submitted.

```
<form (submit)="onSubmit($event)"></form>
```

When you submit a form, this will automatically refresh the webpage by default. To prevent this behaviour we would use the vanilla JavaScript.preventDefault( ) method on the event object as demonstrated below.

```
onSubmit(e) {
    e.preventDefault( );
    console.log('Submitted form!');
}
```

Form inputs have a few events that we can use. For example we can trigger an event on the press of a keyboard key to fire off an event.

```
<input (keydown)="eventName($event)" type="text"></input>
```

We can view/display the keyboard key that was pressed by **console.log(e.target.value)** within our event function.

There are more form input events such as:

<input (keyup)="eName($event)"></input>	When a key is released up (*the opposite of keydown*).
<input (keypress)="eName($event)"></input>	When a key is pressed (*similar to keydown*).
<input (focus)="eName($event)"></input>	When we click in a form input field (*i.e. focus*).
<input (blur)="eName($event)"></input>	When we click out of a form input field (*the opposite of focus*).
<input (cut)="eName($event)"></input>	When we cut from the input field.
<input (copy)="eName($event)"></input>	When we copy from the input field.
<input (paste)="eName($event)"></input>	When we paste inside the input field.

As you can see, there are many useful and powerful form input events that we can utilise within the application for some interesting functionality for example we could use the paste event to prevent a user from pasting in a password which would force the user to enter the password within the application.

To conclude, we have now gone through the various events (*mouse and keyboard*) that we can utilise within our Angular application to trigger off event functions which allows us to either manipulate the component properties or the template itself. This adds a lot of interactivity ad dynamic components to our applications.

Events & Forms

ngModel & Two Way Binding

The ngModel directive allows the application to have two way data binding between a component and an input field. Below is an example of the ngModel directive and creating two way binding:

Firstly, we need to go to the app.module.ts file and import the Forms Module because the ngModel comes from the Form Module by default but is not imported in the app.module.ts module by default. This will remove any errors we would get in the browser JavaScript console.

app.module.ts File:

```
import { FormsModule } from '@angular/Forms';
@ngModule( {
    imports: [ ..., FormsModule ]
} )
```

.component.ts File:

```
export class UsersComponent implements OnInit {
    user: User = {
        firstName: '';
        lastName: '';
        ageName: null;
    }
}
```

We would want to create a new property that will store the form input fields. Note: we can assign the interface to this property, in the above we gave it a interface type of User. When dealing with forms and binding data, we need to have some default values (*generally we want the default values to be blank as seen in the above example*). Note: we cannot assign numbers as an empty string and therefore must use null to avoid any JavaScript console errors. Once we have this property in place, we would want to bind this data to each form input fields within the template file.

To bind a component data to a template form input field, we need to use the ngModel directive on the input element.

.component.html File:

```
<input type="text" [ ( ngModel ) ]="user.firstName">
```

When using the ngModel directive we want to assign the value of component property we wish to bind the input field to (*in the above example we bound the input field to the component user.firstName property*).

Important Note: the square bracket [ ] syntax in angular signifies binding from the component to the template such as property binding, class binding, etc. and the regular brackets ( ) signifies events which bind from the template to the component/class. Therefore, the ngModel directive is for two way binding and so it uses both the square and regular brackets.

We can now use a form button to submit the two way binded fields between the component and template input fields using a click event and passing in data from these fields which is stored in the user class property.

```
<button (click)='addUser( )' [disabled]="user.firstName == '' || user.lastName == '' ">Submit</button>
```

Note: we can add some sort of form validation as seen above whereby we are setting the disabled attribute to the

<button> element if the user.firstName and user.lastName property is empty. We are using the ngModel directive to two way bind the properties of the input field with the component property. This is a basic form of validation but there are other validation methods available such as template driven forms and reactive forms. We are also create our own custom validation when we submit a form.

.component.ts File:

```
export class UsersComponent implements OnInit {
    user: User = {
        firstName: '';
        lastName: '';
        ageName: null;
    }
    users: User[ ]

    addUser( ) {
        this.users.unshift( this.user);
    }
}
```

We no longer require the (user: User) argument within our event function because we have now binded the form input data to the component property. We can now access the user property using this.user and either push this object data to the end of the existing array object using the JavaScript .push( ) method or alternatively use the JavaScript .unshift( ) method to add the new object to the beginning of the array.

Events & Forms

Template Driven Form Validation

To create a template driven form validation we use the ngSubmit directive. The ngSubmit directive works similar to the submit event, except we do not need to prevent the default form submit behaviour from occurring because the ngSubmit is not going to submit the form.

.component.html File:

```
<form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)"></form>
```

We no longer need to pass in the $event object in the onSubmit event function however we need to add a variable that is equal to ngForm because the ngForm directive is included within the forms module by default but is hidden. We created this variable as #userForm and passed this value with onSubmit event function as an argument. This now makes the userForm the global variable for our template driven form.

We can now use this template driven form to add validation to the input fields within the form. Below is an example:

```
<input type="text" name="firstName" #userFirstName="ngModel" required minlength="2">
```

The validation attributes required makes the input field mandatory while minlength attribute also adds a minimum length validation to the input field above. We also add a variable for this input which we called #userFirstName and then set the value to ngForm. This adds the input to the global form validation and also creates a shortcut for referencing the input field rather than writing userForm.firstName. followed by whatever else.

We now have an input with validation that can now be used to validate form submission as seen below:

```
<button [disabled]="!userForm.form.valid">Add New User</button>
```

The disable class will only be applied to the button if the global form (we named as variable #userForm) form input fields are not valid. Therefore, if the above firstName input field is not valid then the button will have the class of disabled. If there were more input fields, if any one of the input fields are not valid then the button will be disabled. Therefore, we can add validation to the input field and then have this affect the global validation of form itself.

We can use bootstrap classes to add styling for error input by using bootstrap class of "is-invalid" - this will add a red border around the input field. To add text feedback we would need to add within the "form-group" a new with the class of "invalid-feedback". This will link the is-invalid with the invalid-feedback elements and the feedback will only display if the input has the "is-invalid" bootstrap class. We can dynamically add the is-invalid class using Angular ngClass directive.

```
<div class="form-group">
    <label>First Name</label>
    <input class="form-control" type="text" name="firstName" #userFirstName="ngModel" required
    minlength="2" [ngClass]="{ 'is-invalid' :userFirstName.errors} && userFirstName.touched">
    <div class="invalid-feedback" [hidden]="!userFirstName.errors?.required">
        First Name is required.
    </div>
    <div class="invalid-feedback"  [hidden]="!userFirstName.errors?.minlength">
        Must be at least 2 characters long.
    </div>
</div>
```

The colon is used to add conditionals for the ngClass. Notice that the above makes reference to the userFirstName input variable and how shorthand the syntax is compared to writing something like fuserForm.firstName.errors etc. The touched event triggers when a user clicks inside of a input field. This ensures the error is displayed when user clicks inside of the input field and enters invalid data.

We added property binding for the hidden attributes on the invalid-feedback elements, where the required or minlength is not in effect. (*we add the ? as an optional because the value could be null and we want the property binding to continue to work, this is known as the Elvis operator*). This will switch between what error message is displayed.

We can add a pattern attribute to validate the input field using regular expression, as seen below with the email input field example:

```
<input type="email" name="email" #userEmail="ngModel" required pattern="[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.
[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?"

<div class="invalid-feedback"  [hidden]="!userFirstName.errors?.pattern">
        Email is not valid.
</div>
```

Note that there are many regular expression we could have used for email. Regular expression is a huge topic on its own and is not covered within this guide material.

To conclude, we can use template driven forms to validate the form input feels and buttons.

Events & Forms

Template Driven Form Submission

We now know how to create a template driven form and validate the form input on the component.html file using the ngForm and ngSubmit directives. We now need to handle the form validation on the component.ts file via the onSubmit( ) event when the form submits.

component.ts File:

```
import { …, ViewChild } from @angular/core;

export class UserComponent implements OnInit {
    …
    @ViewChild(‘userForm’) form: any;

    onSubmit( { value, valid } : { value: User, valid: boolean } ) {
        if( !valid) {
            console.log(‘Form is not valid!’)
        } else {
            value.isActive = true;
            this.users.unshift(value);
            this.form.reset( );
        }
    }
}
```

The first thing is to import something called ViewChild from angular/core. This gives us access to a child component or directive in which we are going to utilise the ngForm directive. We then need to add a new property on the component class which uses the ViewChild decorator and pass in whatever we called the form variable within the component.html file (*i.e. #userForm="ngForm"*). This becomes the identifier for our form property.

Within the component.html file we passed in the form variable as an argument within the onSubmit argument. This form object provides us the form input values as well as whether the input values are valid/invalid. We can access this object within the onSubmit( ) function within the component.ts file as value and valid properties. We are using destructuring to place the values from the form into variables that we can make reference to within our conditional if statement.

Using the if else statement we can check if the form is valid (*note if you added a validation on the button to be enabled only when the fields are valid, the form object will always return true for valid*) and perform some action. In the above example, if valid is not true then it would log a message to the console else it would push the form values to the beginning of the users array as a new users object. We can also add properties onto the value before pushing it onto the array by taking the values object and adding new properties.

Finally, we can also reset the form fields by using the component form property that we added to the class component using the @ViewChild( ) decorative. This saves us from the hassle of clearing all the form fields ourselves when the form is submitted.

To conclude, we have seen how easy it is to validated and submit forms using the template driven forms.

Services, HTTP, Input & Output

What is a Service?

What is a Service?

A Service is basically a class that can use to create and send functionality and data across multiple components. This effectively allows us to keep our components lean and not bloated (*with Ajax calls etc.*) and more importantly, it allows us to keep the application DRY (*don't repeat yourself*). It is important aspect of programming in general to prevent repeating things unnecessarily.

An example of a service and DRY methodology: If we have a server that we are fetching users from (*or whatever it may be*) and we need those users in three different Angular components, we do not want to code the http request in all three of the component files. Instead, we would want to create a service file and place the request code once and then call on that service class method from each of the three components. This prevents us from repeating ourselves.

Services are ideal for making Ajax calls. Angular is a front-end framework and so a lot of the time we are going to be communicating with a back end using HTTP request and Angular has its own HTTP module. We would usually make these request inside of a service.

How do we create a service?

We need to create a new .services.ts file which is usually stored within its own services directory, for example:
src > app > services >
- serviceName.services.ts

Within this .services.ts file we need to import @Injectablt decorator which allows us to use dependency injection for our service. Dependency injection means that we are able to inject our services or whatever it may be into a component.ts file constructor( ) lifecycle method (*remember components have classes with a constructor( ) method*). Inside of the constructor brackets, this is where we would inject the service/dependencies. Once we inject the dependencies, this provides access to all the properties and methods within that service. We create a service class and this is where all the properties and methods are defined.

Once we have created a service class, we need to add/import the service as a provider in the app.module.ts root component file within the @ngModule decorator provider array, the same as we would do for components. We can then call the service into a component(s) by importing it. The component can then use all the properties and methods from within that service.

We can do all this either manually or using the Angular CLI. The Angular CLI commands is:

```
$ ng generate service services/serviceName --module=app        or
$ ng g service services/serviceName --module=app                    or
$ ng g s services/serviceName --module=app
```

The services is the directory name, while the serviceName is the name you wish to call the service file. This will also generate the service file and the class. Adding the module=app.module flag, this will automatically add/import the service file as a provider within the app.module.ts @ngModule decorator provider array for us. Note: we can have many modules within an Angular app and therefore when using the flag we need to specify the main app.module file.

Below is a very simple example code snippet of a Service Class file structure looks like:

getUsers.services.ts File:

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

import { User } from './user';
import { USERS } from './mock-users'

@Injectable( )
export class UserServices {
    getUsers( ): User [ ] {
        return USERS;
    }
}
```

We import the Injectable named export from the @angular/core module. In the above case we are also bringing in a User model (interface) and mock back-end JSON file of all the users array (*usually we would deal with some kind of API or backend database to fetch the existing users array using HTTP*).

We then need to define @Injectable( ) as a decorator and we then have our service class which usually has the naming convention of Name followed by Service (*i.e. NameService*). In the above example there is one method in this class called getUsers which is type to the User module (interface) as an array and it then returns the USERS within an array object. Whichever component file we inject this service, the component will have access to the getUsers( ) service method and is able to pull all the users data from the service.

Services, HTTP, Input & Output

Creating a Service

To create a service within our project, enter the following code within the Angular CLI:

```
$ ng g s services/serviceName —module=app.module --spec=false
```

This will generate the serviceName file (*e.g. data.service.ts*) within the services directory and will also add the service to the app.module @ngModule decorator. If the services folder does not exist, the CLI will create the folder. If we omit the —spec=false flag, the CLI would create a .service.spect.ts test file.

If we open the .service.ts file we would see the service class created for us including an empty constructor:

.service.ts File:

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

@Injectable( )
export class NameService {
    constructor( ) { }
}
```

app.module.ts File:

```
import { NameService } from './services/name.service';
@ngModule( { declarations: [...], imports: [...], providers: [ NameService ] } )
```

A Service is a provider and it provides the application and components with different things usually using an outside API. Below is an example code of a data service which provides users data:

.services.ts File:

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

@Injectable( )
export class NameService {
    users: User[ ]

    constructor( ) {
        this.user = [
            firstName: 'John',
            lastName: 'Doe';
        ]
    }
    getUsers( ): User[ ] {
        return this.users
    }
}
```

In the example above, this service imports the Users array model (interface) and sets users to the type of User array object. Within the constructor method, the users data property is assigned an array object of the users data. Finally,

the getUsers method has a type of User array and simply returns the users array property. We now have a service that stores the users data which any component can tap into this service properties and methods to retrieve the users array data and any other service methods. Below is an example of how we can bring in the service into the component and inject the dependency.

.component.ts File:

```
import { DataService } from '../../services/data.service';
export class UsersComponent implements OnInit {
    users = User[ ];

    constructor( private _dataService: DataService) { }

    ngOnInit( ) {
        this.users = this.dataService.getUsers( )
    }
}
```

The Dependency Injection within the constructor can be either public or private. Private means that this variable that we are connecting to the service can only be used within this class and cannot be used anywhere else because it is private. This private variable can be called anything we want (in the above it is called dataService) and then followed by a colon : to set the variable to the data service we want (*in the above we setting the variable to the DataService class that we imported in*). Note: sometimes we see variables leading with an underscore _ to indicate the variable name as a private variable. We are now able to use this.dataService followed by any property or method within the service class. In the above example we used the getUser( ) method on the ngOnInit method to set the component users property to the dataService users array object.

Services, HTTP, Input & Output

Working with Observables

So far we have seen code snippets for synchronous execution whereby the code is executed line by line, one line at a time. Each time a function (or method) is called the program execution waits until the function returns before continuing to the next line of code. However, the problem with synchronous programming is that suppose a function starts a time consuming process (*usually fetching data from an API or server*), the application is stuck waiting fo the process to end before it can execute the next line of code. Asynchronous execution avoids this bottleneck.

Observables are like data streams that are open and we can subscribe to them. We are able to create our own observables as well as return something else as an observable. Observable is not from an Angular package but it comes from an extension called reactive extensions, also known as RxJS. Below is an example of creating our own observable:

.service.ts File:

```
import { Observable } from ‘rxjs’;

@Injectable( )
export class DataService {
    data: Observable <Array <any> >;

    getData( ) {
        this.data = new Observable(observer => {
            setTimeout( ( ) => {
```

```
                observer.next( [1] );
            }, 1000);
            setTimeout( ( ) => {
                observer.next( [2] );
            }, 2000);
            setTimeout( ( ) => {
                observer.next( [3] );
            }, 3000);
        });
        return this.data;
    }
}
```

We created a data property and set it to an Observable with a type of array and the array object type of any (*we could have set this to any by itself*). We then can create a method called getData and assigned the data property to a new Observable which takes in an observer as the argument.

Using an arrow function we can use the observer.next( ) method to publish to anything that has subscribed to the observable. We used the JavaScript setTimeout method to demonstrate the observable open data stream whereby every second it will fetch whatever is within the .next( ) brackets (*in the above case these are array between 1 to 3. This could have been anything like an object for example {name: 'Andy'}*).

.component.ts File:

```
import { DataService } from '../../services/data.service';
```

```
export class UserComponent implements OnInit {
    data = any;

    constructor( private _dataService: DataService) { }

    ngOnInit( ) {
        this._dataService.getData( ).subscribe(data => {
            console.log(data);
        });
    }
}
```

Within the component.ts file, we need to import the service in order to use the observable and within the ngOnInit method we subscribed to the observable by using the .subscribe( ) method. The subscribe method takes in a value which could be called anything (*in the above we used data*) and we can use an arrow function to return the data (*in the above we console.log( ) the data*).

If we were to look at the browser console, we can see the observable as an open data stream and this is proven by the setTimeout method by waiting each second as the next data is being released. The data can be anything.

Observables is a useful feature which makes it easy to work with asynchronous data e.g. where we are fetching data from an API or HTTP. Note: creating data streams is outside of Angular and you may wish to look into RxJS separately to learn how to create data streams.

Within the .service.ts file, we would import along with the Observable operator called of which also comes from RxJs.

```
import { Observable } from ‘rxjs’;
import { of } from ‘rxjs’;
import {User} from ‘../../models/User’
```

The of operator allows us to return an array as an observable for example we could return a users array. Taking the getUsers( ) method from the last section we could create an observable to return an observable array of users.

```
getUsers( ): Observable<User[ ]> {
      return this.user
}
```

The getUsers method is set to an Observable which is given the type of the User array model (interface). To return this.users array as an observable we would wrap the property within the of( ) operator method. We can now subscribe to the getUsers observable method.

Within the .component.ts file we can no longer call **this.users = this._dataService.getUser( )**; instead we would just subscribe to the service which will give the users and we can then set it to the component this.users property. The application should continue to work with an asynchronous API.

```
this._dataService.getUser( ).subscribe(users => {
      this.users = users;
});
```

Services, HTTP, Input & Output

HTTP Client Setup & GET Request

In this section we are going to be analyse the Angular HTTP module and using a fake REST API from a website called JSON placeholder which provides fake REST API data for development:

https://jsonplaceholder.typicode.com/

In the below example we are going to be fetching using the GET HTTP method user posts from JSON placeholder. Within the terminal we would create a new component and services file for posts by running the following Angular CLI command:

```
$ ng g c components/posts --spec=false --module=app
$ ng g s services/posts --spec=false --module=app
```

Once we have created the posts component and the post service, within the posts.component.ts file we would create a property for posts and set it to the type of posts array (*which will be a model (interface) - we would create this file within the models directory called Posts.ts and import it into the component.ts file*).

.component.ts File:

```
import { Post } from '../../models/Post';

export class PostsComponent implements OnInit{
    posts: Post[ ];
}
```

The model (interface) for the post will be modelled from the JSON placeholder posts data which has different data we could pull from the JSON file.
https://jsonplaceholder.typicode.com/posts

Posts.ts File:

```
export interface Post {
    id: number,
    title: string,
    body: string
}
```

In order to use the HTTP module we would need to import this into the app.module.ts file.

app.module.ts File:

```
import { HttpClientModule} from '@angular/common/http';

@ngModule: ( { …, imports[…, HttpClientModule ], … } )
```

The posts data from JSON placeholder will come from the the service. Within the service.ts file we will import the HTTP Client and HTTP Headers from the HTTP module. We would also need to bring in observable from rxjs because the HttpClient returns an observable.

.services.ts File:

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

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import Post from '../../models/Posts'

@Injectable( )
export class PostServices {
      postsUrl: string = 'https://jsonplaceholder.typicode.com/posts'

      constructor(private _http: HttpClient ) { }

      getPosts( ) : Observable<Post[ ]> {
            return this._http.get<Post[ ]>(this.postsUrl);
      }
}
```

We would create a property for the JSON placeholder posts URL (*ensure the url is https, otherwise we may run into some errors*). We would then inject the http client (*the posts URL*) as a dependency within the constructor method. We would create it as a private property and set it to the HttpClient.

Finally, we would create a method which will have a type of observable and a sub type of the Post array model (interface). The method will return the private _http variable and we would want to call the .get method to make a GET request to JSON placeholder to return the JSON list of posts data. We can also give the get a type of a Post array model (interface). We would pass in the postsUrl variable within the .get method because we are making a request to the JSON placeholder posts url.

We can now go back to the component.ts file to implement the service within the ngOnInit lifecycle method so that the posts are fetched when the component is initialised. We therefore need to import the postService and inject it within the constructor method and then use it within the ngOnInit method.

.component.ts File:

```
…
import { PostService } from '../../services/post.service';
constructor (private _postService: PostService)

ngOnInit( ) {
    this._postService.getPost( ).subscribe(posts => {
        console.log(posts);
        this.posts = posts;
    });
}
```

The private postService variable could have been named anything we wanted and is set to the PostServices observable. We can then use the getPost method from the post service which is an observable and subscribe to it. We can return the data (*which we named as the variable called posts*) and do whatever we want with the data. In the example above we could console.log(posts) or set the posts array property to the posts data fetched.

Within the console we would have an array with 100 posts data values from the JSON placeholder API. To conclude we are successfully using the HTTPClient module to make a GET request to the JSON placeholder url to fetch 100 user posts data.

To wrap the whole component up, we could add the data within the .component.html template file to display the data within the application as seen in the below example:

.component.html File:

```
<h2>Posts</h2>
<div class="card" *ngFor="let post of posts">
    <div class="card-body">
        <h3> { { post.title } } </h3>
        <p> { { post.body } } </p>
    </div>
</div>
```

We can use the *ngFor structural directive to loop through the posts component property and create a div wrapper with a <h3> element for the post title and a <p> element for the post body for each posts data within the post array. We can use bootstrap framework classes of card and card-body to style the component.html elements.

To conclude, we have now learned how to setup the HTTPClient module within the Angular application and how to use the GET method to fetch data from an external API and then display the data in the browser. We will now continue to explore the other HTTP methods of POST, UPDATE and DELETE which would make up a CRUD web application.

Services, HTTP, Input & Output

HTTP Client POST Request

In this section we are going to look at the POST HTTP method using the Angular HTTP Client Module. We are also going to analyse how to break components into separate component files and have them communicate with each other. For example, we may want a post component which displays all posts and another post form component on its own that allows the user to use the form to post a new post.

We could use the Angular CLI to create a new post-form component using the following command:

```
$ ng g c components/post-form --spec=false --module=app
```

We could create a HTML form within the post-form.component.html template file and then embed the app-post-form element within the posts component template file which will display both the post component and post-form component templates within the browser.

post.component.html File:

```
<app-post-form></app-post-form>
```

We can import the PostsService and Post model (interface) into the post-form component and inject the PostService dependency within the post-form constructor method.

post-form.component.ts File:

```
import { PostService } from '../../services/post.services';
```

```
import { Post } from '../../models/Post';

export class PostFormComponent implements OnInit {
    constructor(private _postService: PostService ) { }
    addPost(title, body) {
        this.postService.savePost( { title, body } as Post ).subscribe(post => {
            console.log(post)
        });
    };
}
```

We can then use the PostService on a method within the component class for example addPost as seen above. The addPost method will take in a title and body properties from the post form submitted by the user as arguments and call on the .savePost method passing in an object with the title and body and setting the object as the Post model (interface). Finally, the method will subscribe to the observable which will return the new post.

We would then need to create the savePost method within the post.service.ts file as this does not exist.

post.service.ts File:

```
const httpOptions = {
    headers: new HttpHeaders( { 'Content-Type': 'application/json' } )
}

@Injectactable( )
```

```
export class PostService {
    postsUrl: string = 'https://jsonplaceholder.typicode.com/posts'
    constructor(private _http: HttpClient) { }

    savePost(post, Post): Observable<Post> {
        return this._http.post<Post>(this.postUrl, post, httpOptions);
    };
}
```

When we send a POST HTTP request, we would want to send a Header with the content type (*the content type will be application/JSON*). We would create a variable called httpOptions (*or anything we want*) which will store any Header values that we would want to send. We set the header values using HttpHeader from the HTTP Client Module.

The savePost takes in the post as an argument which comes from the form and we can set it to the type of Post array model (interface). The savePost method will return a type of Observable which has a subtype of a Post model (interface) - this will be a single post and not an array of Post.

Finally, on the savePost method we will return a post method on the HttpClient which we assigned the type of Post model (interface) and passed in three arguments that the post method takes:

1) the JSON Placeholder URL which we stored in a variable called postUrl
2) the post object submitted from the post-form which was passed into the savePost method as the post argument
3) the HTTP Header which we stored in a variable called httpOptions

This will now post the post-form data submission to the API and return the post JSON object within the console.

Services, HTTP, Input & Output

Event Emitter & Output

In the last section we analysed how we could create separate component files within an application (example being post and a post-form component). We also analysed how to use the HTTP POST method and the arguments required for the post method. However, there is currently an issue of the components talking with one another — the post-form will post a new post but this component will not update the post component to display the new post submitted using the post-form component. In this section we are going to analyse how event emitters can be used to communicate between multiple component files by emitting events.

Within the post-form.component.ts file, we will need to import Event Emitter and Output from @angular/core module.

post-form.component.ts File:

```
import { …, EventEmitter, Output } from '@angular/core';

export class PostFormComponent implements OnInit {
    @Output( ) newPost: EventEmitter<Post> = new EventEmitter;
}
```

Within the properties of the class component we would want to create an @Output property which we called newPost and newPost is a type of EventEmitter which a subtype of the Post model (interface). We assigned newPost as a new Event Emitter.

When a post request is made using the PostService savePost method which returns an observable that we can subscribe to within the addPost method which then returns a post request json object, we can now (*instead of*

console.log(post)) use the newPost event emitter and use the .emit method, passing in the post object that comes from the subscribe method. In effect, this will now emit a new event from within the post-form.component.ts file with the new post object.

```
addPost(title, body) {
    this.postService.savePost( { title, body } as Post ).subscribe(post => {
        this.newPost.emit(post);
    });
}
```

We can now go into the post.component.html template file where we embedded the app-post-form element to pass in the newPost event emitter.

post.component.html File:

```
<app-post-form (newPost)="onNewPost($event)"></app-post-form>
```

Within the component element we use brackets around the event emitter name (*in the above example the event emitter name was called newPost using the @Output( ) decorator*) and assign the value of the method name we are going to run within the post.component.ts file passing in the $event object as an argument.

Within the post.component.ts file we would now have to create the event method (*this is called this onNewPost in the above example*).

post.component.ts File:

```
export class PostFormComponent implements OnInit {
    posts: Post[ ];

    onNewPost(post: Post) {
        this.posts.unshift(post);
    }
}
```

The onNewPost will take the event object which we will pass down into the method as post and set it to the type of Post model (interface). We then want to add the new post onto the existing posts array property within this component file. The .unshift method will add the new post to the front of the array while .push method will add the new post to the end of the array.

To conclude, we now have separate components that are able to communicate with each other by using event emitters and the @Output decorator. This allows one component to pass data along to another component by emitting an event passing in the data as the argument which will trigger a event in another component which will then be able to utilise the data that has been passed down from the emitter event as the event object. This allows application UI/UX components to be broken down further into smaller component files.

Services, HTTP, Input & Output

Input & Edit State

To continue from the last section, we analysed both event emitters and the @Output decorator. In this section we are going to analyse the @Input decorator to input data from one component into another component, something similar to a edit form state.

In the post.component.html file we could create a edit button which has a click event that will call a editPost method and pass in the post data (*which comes from the *ngFor structural decorator*) from the selected post.

```
<app-post-form (newPost)="onNewPost($event)"></app-post-form>
<div *ngFor="let post of posts">
    <h3> { { post.title } } </h3>
    <p> { { post.body } } </p>
    <button (click)="editPost(post)">Edit</button>
</div>
```

Within the post.component.ts file the component can have a new property for the current post and the editPost method for the click event from the template file.

```
export class PostComponent implements OnInit {
    currentPost: Post = {
        id: 0, title: '', body: ''
    }
```

```
    editPost(post: Post) {
        this.currentPost = post;
    }
}
```

The currentPost property has a type of the Post model (interface) and we need to set the current post to the default values of an empty form which matches the Post model (interface) data structure. The editPost method takes in the post from the template click event which is set to the type of the Post model (interface) and the currentPost property is then set to the passed in post argument as its new value.

Within the post.component.html file we can now update the app-post-form element to now take in an input value.

```
<app-post-form [currentPost]="currenPost" (newPost)="onNewPost($event)"></app-post-form>
```

In the previous section we saw an output coming from the component element which was signified by the regular brackets (*i.e. (newPost) was the output*). Input values are signified by square brackets. In the above example we are inputting the currentPost property from the post component into the post-form component. When we pass an input within a component element we need to define it as an @Input.

Within the post-form.component.ts file, we would need to import in the Import named export from @angular/core.

```
import { …, EventEmitter, Output } from '@angular/core';
export class PostFormComponent implements OnInit {
    @Output( ) newPost: EventEmitter<Post> = new EventEmitter( );
    @Input( ) currentPost: Post;
```

We defined the currentPost coming in from the post.component.html file as an @Input( ) decorator property within the post-form component file and gave it a type of the Post model (interface). The currentPost is now a property of post-form just as if the post-form component was within the post component itself. As we can see we can separate the components out rather than having one component file with a lot of code for different UI/UX elements.

We can now bind the currentPost property within the post-form.component.html template file using ngModel.

```
<input type="text" name="title" [ ( ngModel ) ]="currentPost.title" />
<input type="text" name="body" [ ( ngModel ) ]="currentPost.body" />
<input type="hidden" name="id" [ ( ngModel ) ]="currentPost.id" />
```

We are two was binding using ngModel the currentPost title data to the title input and the currentPost body to the body input element. We would also want to create a hidden field within the form to store the currentPost id value. This value would be used to help edit the data in a HTTP PUT request or delete a data using the HTTP DELETE request which would need to take in the unique id of a post to delete from a database.

*Note: we could change the form button by creating a isEdit property which stores a boolean value and then use the @Input( ) decorator to pass the property into the post-form to display/hide buttons (one for Add Post and another for Edit Post) using the *ngIf structural decorator.*

We are now able to click on an edit button from the post component template file which will input the data into the post-form component template file using the @Input decorator allowing the two component to remain separate but continue to communicate with each other.

Services, HTTP, Input & Output

HTTP Client PUT Request

So far we have analysed how to use the Angular HTTP Client module GET and POST request. In this section we are going to look at the HTTP PUT request method. The PUT request allows data to be sent to an API or backend database to update an existing data record with the new values passed with the request.

As we have seen previously we would need to create a button within the .component.html template file that will call a updatePost method from the .component.ts file and this file method will call the PUT request method from a service passing data along. The syntax/setup is very similar to both the GET and POST method and is demonstrated below:

post-form.component.html File:

```
<button (click)="updatePost( )" *ngIf="isEdit"> Update Post </button>
```

post-form.component.ts File:

```
@Input( ) currentPost: Post;

updatePost( ) {
    this._postService.updatePost(this.currentPost).subscribe( );
}
```

The currentPost is coming from the @Input( ) decorator which is getting the value from the post.component.ts and post.component.html template file.We would use the currentPost property data which has the id, title and body of the post that we would want to pass along to the updatePost service class method.

Note: we do not need to return anything from the subscribe method because the UI will automatically update due to the two way binding. However, if we wanted to test the update method worked we could have added the below:

```
.subscribe(post => {
      console.log(post)
} );
```

post.service.ts File:

```
      const httpOptions = { headers: new HttpHeaders( { 'Content-Type': 'application/json' } ) }
      export class PostService {
            postUrl: string = 'https://jsonplaceholder.typicode.com/posts'

      updatePost(post: Post): Observable<Post> {
            const url = ` ${this.postUrl} / {post.id} `;
            return this._http.put<Post>(url, post, httpOptions);
      }
```

The updatePost service method will take in post which is set to the type of Post model (interface) and the method return is set to the type of Observable with a subtype of Post model (interface). The put request is sent to a different URL which contains the post id we wish to edit. This is stored in a variable that uses ES6 template string to dynamically concatenate the URL and post id variable as a single URL string value.
We then return the post method passing in the url, post object and the httpOptions header (*which contains the content type of json*) as the put method arguments.

Services, HTTP, Input & Output

HTTP Client DELETE Request

The HTTP Client DELETE request method syntax is exactly the same as the PUT request. This will delete a data record from the API/database using the id as the identifier. Again we would need a button in the template file to call an event in the component file which will call on a service method to send a http delete request passing in the relevant data object and id.

post.component.html File:

```
<button (click)="removePost(post)"> Remove Post </button>
```

post.component.ts File:

```
removePost(post: Post) {
    if(confirm('Are you sure?') ) {
        this._postService.removePost(post.id).subscribe(( ) => {
            this.posts.forEach((curr, index) => {
                if(post.id === curr.id) {
                    this.posts.splice(index, 1);
                };
            });
        });
    };
}
```

We are using a JavaScript confirm message to confirm the user wishes to delete and if they continue this will send a http delete request passing in the post id as an argument to the service method. The delete request does not give an object back and so the subscribe method arrow function will have an empty brackets. Within the arrow function we will loop through the posts array using a JavaScript forEach loop to check if the post data passed into the method matches the current iteration and if so we are going to .splice (remove) the post from the posts array using the index. Note: the forEach method takes in two arguments, one is the current iterator property and the other is the index value. Finally, we can set the currentPosts component property back to the default to reset the form in the browser.

post.service.ts File:

```
    const httpOptions = { headers: new HttpHeaders( { 'Content-Type': 'application/json' } ) };

    removePost(post: Post | number) {
        const id = typeof post === 'number';
        const url = ` ${this.postUrl} / {id} `;
        return this._http.delete<Post>(url, httpOption);
    }
```

The removePost takes in the post data which is set to a type of either the Post model (interface) or a number which returns an Observable with the subtype of Post model (interface). The ternary operator is checking whether the post argument passed in is a number or an object, if number we will use post else we will take the post.id number and store it in a variable called id. In the url variable we use a template string to concatenate the jsonplaceholder url and the post id we wish to delete. Finally, we return from the HttpClient the delete method which takes in two arguments of: url and the Http Header (*content-type set to applicaition/json*) which we stored as a variable called httpOption.

To conclude, we now know how to use the HTTP Client module to make CRUD (GET, POST, PUT, DELETE) requests.

Angular Router

Angular Router Overview

We have learnt how to create an application on a single page with a single/embedded components all on a single URL. Angular Router allows us to assign separate components to separate URLs so that it can appear as though there are multiple pages on the web application or website.

What does the Router Do?

- It handles navigation from one component/view to another.
- It mimics loading separate pages via browser.
- It includes back button functionality.
- Most Angular apps are Single Page Applications (SPA). We have the main root index.html files and use the router to make it appear as though there are multiple pages.

Basic steps to creating Routes:

- Create a new app-routing module (similar to app.module.ts file).
- Import the @angular/core Router Module.
- Create Routes.
- Create Router Outlet which is a special tag that will show the current component for the route.
- Add Links so that we can switch back and further between different routes.

Additional Overview:

The <base href="/"> is the root url of the application. If we deploy the application to a hosting site and put the application in a subfolder, we would want to put the subfolder in the href (*e.g. /blogs*).

The app-routing module snippet:

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

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

Creating routes snippet:

```
Const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'about', component: AboutComponent },
    { path: ' ** ', component: PageNotFoundComponent }
];
```

Creating routerLink snippet:

```
<a class="nav-link" href="#" routerLink="/"> Home </a>
```

Creating router-outlet to app.component snippet:

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

Angular Router

Creating App Routing Module

The recommended way of setting up the Angular Router (based on the Angular documentation) is to setup a separate module for the application router. Just like the app.module.ts file, there is the option of creating more modules and use those within the application.

Important Note: if you answered Yes to the question “Would you like to add Angular routing? (y/N)” when setting up the application project files for the first time, this would have already created the app-routing.module.ts file and linked it to the app.module.ts file for you. The below command is to install the app-routing files if you answered No to the setup question.

To create a new app-router module we would enter the following command in the terminal using Angular CLI:

```
$ ng generate module app-routing --flat --module=app
```

Alternative shorthand command line code:

```
$ ng g m app-routing --flat --module=app
```

We add the module flag to connect the app-routing module to the root app.module file. This will create the app-routing.module.ts file and update the app.module.ts file within the Angular application.

If we open the app-routing.module.ts file, you will notice the Angular CLI has created a basic format for a module file which brings in NgModule from @angular/core and CommonModule from @angular/common which are both required for module files. This also has the @NgModule decorator to import the CommonModule.

app-routing.module.ts File:

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

@NgModule( {
    declarations: [ ],
    imports: [ CommonModule ]
} )
export class AppRoutingModule { }
```

app.module.ts File:

```
import { AppRoutingModule } from './app-routing.module';
@NgModule( {
    declarations: [...],
    imports: [..., AppRoutingModule ]
    ...
) }
export class AppModule { }
```

Note: we could also manually create these files ourselves without the need of the Angular CLI.

We would need to update the app-routing.module.ts File to remove the CommonModule as it is not required for the routing module; however, we would need to import the RouterModule and Routes from @angular/router module. The updated app-routing.module.ts file will look something like the below:

app-routing.module.ts File:

```
import { NgModule } from '@angular/core';
import { RoutingModule, Routes } from '@angular/router';

const routes = [ ]

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

We would create a variable called routes and set it to the type of Routes model (interface) which we imported from the @angular/router module and set it to an empty array. This is where we would map the URL paths to whichever component we would want to load. Within the @ngModule decorator we would need to get rid of declarations because this is not needed in the app-routing module. We would update the imports to include RoutingModule and use .forRoot( ) method and pass in the routes variable containing all the routes we would create for the application. Finally, we would need to export the RouterModule itself so that we can bring it into the app.module.ts file as an import.

Finally, we need to create the router-outlet within the **app.component.html file:**

```
<div>
    <router-outlet></router-outlet>
</dv>
```

Angular Router

Creating & Mapping Routes

To add a new Route to the Angular app-routing module we would import the component into the app-routing.module.ts file and add the an object within the routes variable array.

app-routing.module.ts File:

```
import { HomeComponent } from ‘./components/home/home.component’;
import { UsersComponent } from ‘./components/users/users.component’;
import { PostsComponent } from ‘./components/posts/posts.component’;

const routes: Routes = [
    { path: ‘’, component: HomeComponent },
    { path: ‘users’, component: UsersComponent },
    { path: ‘posts’, component: PostsComponent }
];
```

The object within the routes variable array should have two properties: one for the route path and the other for the component to display at the route path (only one component can be named). Note: an empty path value refers to the root application URL i.e. the root route for the Angular application is localhost:4200 or whatever the domain name if hosted on a web hosting site.

Navigating to the URL for example localhost:4200/users will display the users component while localhost:4200/posts will display the posts component in the users browser.

Angular Router

Linking to Routes

Now that we have the app-routes module setup and have created some routes within the application to display different components on different URL routes, we can now add links within template files to provide navigation links to various routes within the application.

Angular has a special binding called routerLink that we can use replacing the HTML href attribute. Example below:

.component.html File:

```
<button routerLink="/">Home</button>
<button routerLink="/user">Users</button>
<button routerLink="/posts">Posts</button>
```

These button can now act as links to the various pages within the website/web application. Note the "/" will route the user back to the root URL.

To add an active state to link, Angular has special directives that we can use called routerLinkActive and routerLinkActiveOptions (*the exact: true will check if the route URL is an exact match which is useful for root routes*) for example:

```
<li [routerLinkActive]="[ 'active' ]" [routerLinkActiveOptions]="{exact: true}">
    <a routerLink="/"> Home </a>
</li>
```

Angular Router

Params & ActivatedRoute

We can add dynamic routes that require parameters to display a certain component for example if we wanted a route for an individual post details page.

app-routing.module.ts File:

```
import { PostComponent } from ‘./components/post/post.component’;

const routes: Routes = [
    { path: ‘post/:id’ component: PostComponent }
]
```

The :id parameter within the route path is required so that we know which post to display within the post component when a user follows the certain path and provides a post id parameter within the URL route.

post.component.ts File:

```
import { Component, OnInit } from ‘@angular/core’;
import { ActivatedRoute } from ‘@angular/router’;
import { Location } from ‘@angular/common’;
import { PostService } from ‘../../services/post.service’;
import { Post } from ‘../../models/Post’;
```

We require to import the ActivatedRoute from router module, Location from the angular common module and the

PostService because we want to make a GET request to get the single post data from the API.

```
export class PostComponent implements OnInit {
    post: Post;

    constructor(
        private _route: ActivatedRoute
        private _postService: PostService
        private _location: Location
    ) { }

    ngOnInit( ) {
        const id = +this._route.snapshot.paramMap.get('id');

        this._postService.getPost(id).subscribe(post => this.post = post);
    }

}
```

We created a proper of post and set it to a type of Post model (interface) and then injected all of the components dependencies (i.e. the PostService, ActivatedRoute and Location) into the constructor as private variables.

On the ngOnInit lifecycle method we are creating variable called id and grabbing the value from the URL router using +this._route.snapshot.paramMap.get( ) method syntax and passing in whatever we want to get from the active URL route — in the above we want the id number which we called id within the app-routing.module.ts route path *'post/:id' (is we used 'post/:idNo' then we would need to pass idNo in the get( ) method to return the id from the active URL route*).

We can use this parameter to make a get request to the API to fetch the individual post. Note that we are using the ES6 arrow function which allows us to use a short form syntax to return component post property implicitly. The explicit return syntax would have been:

```
.subscribe(post => {
      return this.post = post
} );
```

Within the postService.service.ts file we would create the getPost GET request service class method.

postService.service.ts File:

```
    getPost(id: number): Oberervable<Post> {
        const url=`${this.postsUrl}/${id}`;
        return this._http.get<Post>(url);
    }
```

We should now be able to display the post details within the post.component.html template file. We would use the *ngIf structural directive to check if there is a post and then display the post title and body.

post.component.html File:

```
<div *ngIf="post">
    <h3> { { post.title } } </h3>
    <p> { { post.body } } </p>
</div>
```

We can now add a link to the posts.component.html file so that the <h3> titles would link to the individual post component page.

posts.component.html File:

```
    <h3><a routerLink="/post/{ { post.id } }">{ { post.id } }</a></h3>
```

To conclude, we can now set route parameters and use these within our component files to create dynamic routes within the Angular Application and have the ability to grab these values from the active route URL within the browser.

Angular Router

Page Not Found Route

Now that we understand how to create routes and link to routes within an Angular application, we would want to create a component for the 404 Page Not Found which will display when a user tries entering an invalid non-existing route.

The Page Not Found component will have a very simple template file to display the Page Not Found message to the user.

Within the app-routing.module.ts file we would import the page-not-found component and add a route path within the routes variable.

```
Import { PageNotFoundComponent } from ‘./components/page-not-found/page-not-found.component’;

const routes: Routes = [
      …, { path: ‘**’, component: PageNotFoundComponent }
]
```

We would need to make sure the page not found route objects is added at the bottom of the array list objects. The path of double asterisks (**) is a wild card and regardless of where the user navigates to, the router will display the PageNotFound component if the route does not match any of the existing routes created within the routes array.

To conclude, we now have a base understanding of setting up routes within an Angular application and wraps up the fundamentals of creating multipage Angular applications using the Angular Router module.

Angular Router

Guard for Routes

In Angular we can create something called a Guard which protects certain routes from being accessed if a condition is not met for example Authentication. Guards are services. We would create a new folder within src/app called guards to separate the application guard services from other services files which are located within the services folder.

To create a guard file we would create a new file with a name followed by .guard.ts file extension. Below is an example of a Auth Guard:

src > app > guard
> auth.guard.ts

We can create as many .guard.ts file as we want with different criteria's to protect the application routes.
Note: we could use the Angular CLI to create a service file within the guard folder or manually create the file.

auth.guard.ts File:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
Import { Observable } from 'rxjs';

@Injectable( )
export class AuthGuard implements CanActivate {
    constructor(
```

```
            private _router: Router,
        ) { }

        canActivate( ): Observable<boolean> {
            if(criteriaIsTrue) {
                return true;
            } else {
                this._router.navigate( [ '/' ] );
                return false;
            }
        }
}
```

We need to import CanActivate from the core @angular/router module. The @Injectable decorator is required to inject the service as a dependency and then export the class implements the CanActivate. Within the constructor we would need to pass in all the dependencies for this guard service. In the above we created a private variable called _router and set it to the Router import.

In order to use CanActivate we need a method called canActivate else this would provide an error. The canActivate returns an Observable which we set to return a boolean. We can add any criteria for the return code block and create a re-route for when the criteria returns false. We must also return true or false within the code block as the Observable is expecting a boolean. In the above we are navigating back to the root of the application and returning false when the criteria has not matched.

app-routing.module.ts File:

```
…
import { AuthGuard } from ‘./guards/auth.gaurd’;

const routes: Routes = [
      { path: ‘’, component: DashboardComponent, canActivate: [ AuthGuard ] }
];

@NgModule({
      exports: [RouterModule],
      imports: [RouterModule.forRoot(routes)],
      provider: [AuthGuard]
});
```

We need to import the Guard service into the app-routing module and because a Guard is a service we must also add it as a provider within the @NgModule. Whichever, routes we want to protect using the Guard we would simply add an extra parameter of canActivate followed by which guard we would want to use (*we are able to use more than one Guard to protect routes*).

To conclude we are now able to protect an application route by using Guards and this concludes setting up routes within an Angular application.