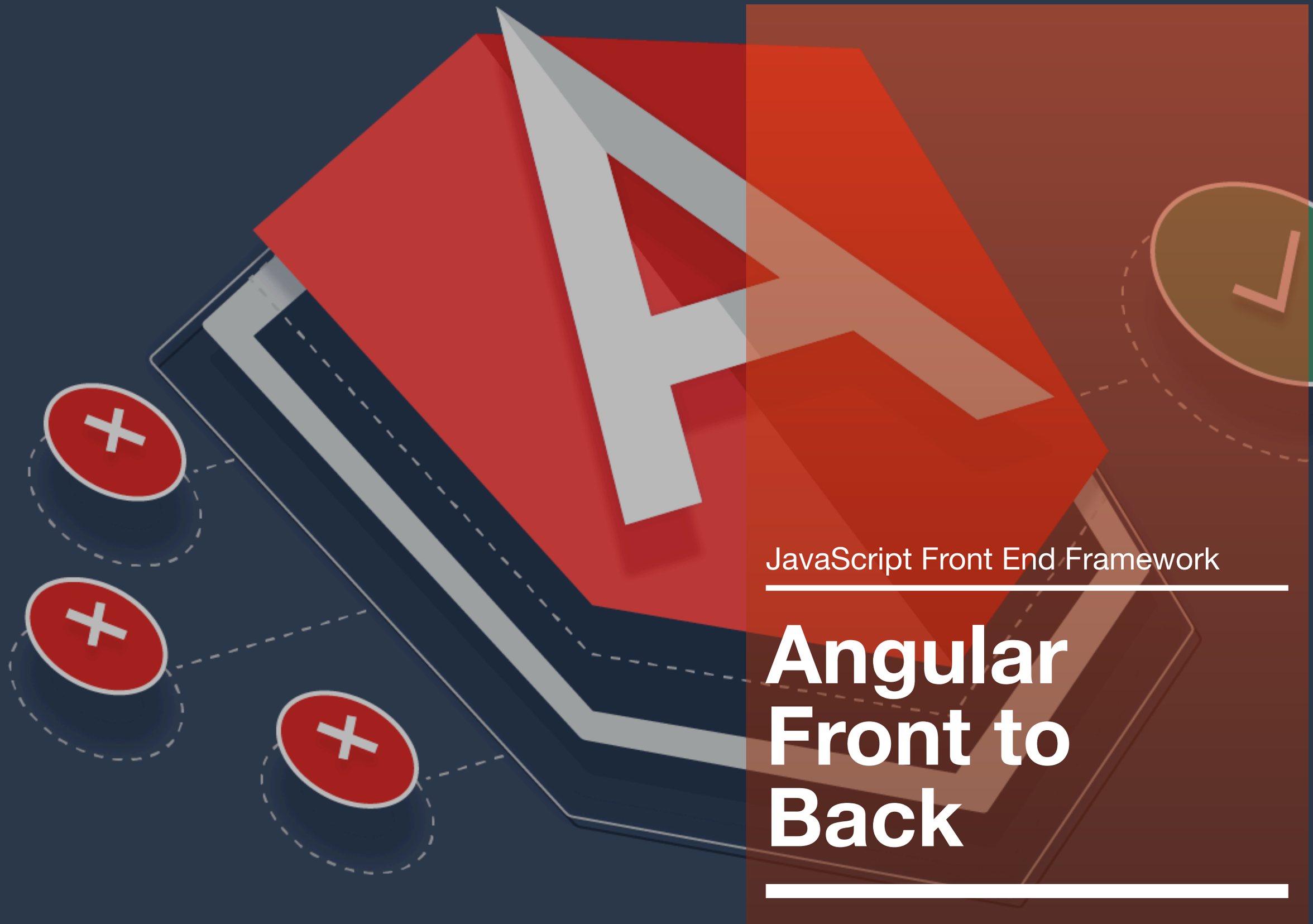


JavaScript Front End Framework

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
```

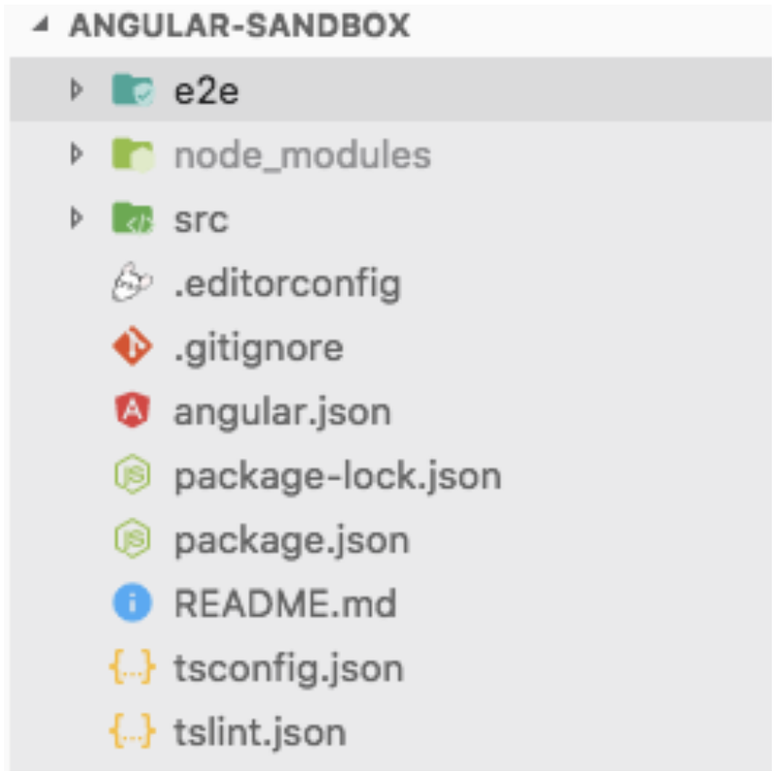
We now have a new boilerplate Angular Application installed on our machine which we can now edit and write our own Angular code to build up our own application.

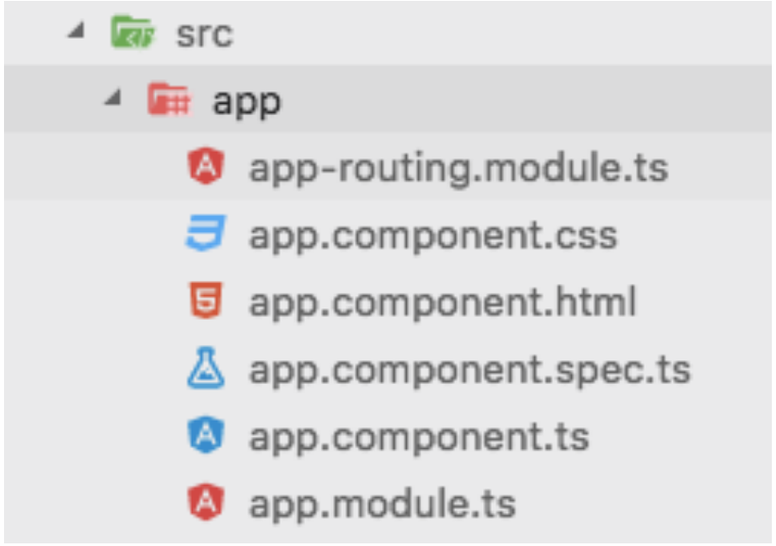
You may also wish to install Augury extension for your chrome browser to view your Angular App component state/ properties within your browser i.e. a dev tool similar to react dev tool but for Angular.
https://chrome.google.com/webstore/detail/augury/elgalmkoelokbchhkhacckoklkejnhcd?hl=en

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.

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>
```

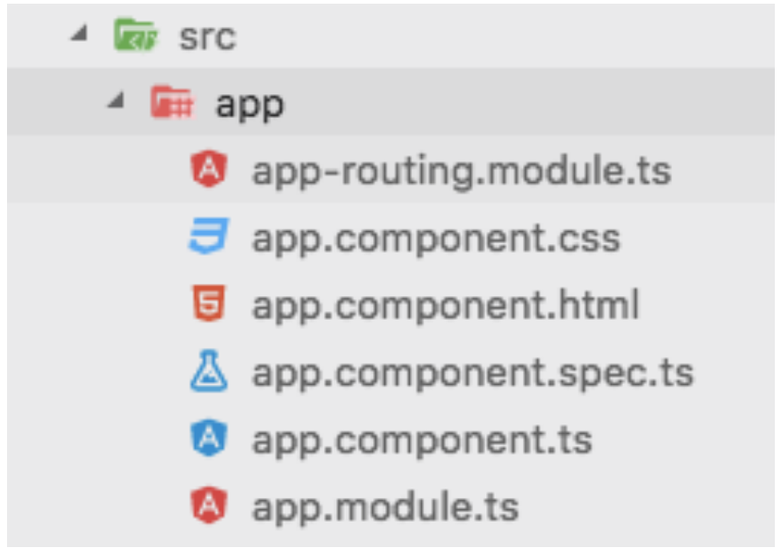
The first line is importing in the Component module from the angular/core package. A component is really just a class with properties and methods. We have a class called AppComponent which has one property called name which is set to a string called Angular. In addition to the class we have something 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:

users.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.

components.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>
```