Matt Hudson - Exploration 1 Writeup

Links to your StackBlitz sandboxes

Code editing here: https://stackblitz.com/edit/angular-jtdc7h (https://stackblitz.com/edit/angular-jtdc7h)

Front-end site here: https://angular-jtdc7h.stackblitz.io (https://angular-jtdc7h.stackblitz.io)

I used StackBlitz to code along with tutorials and explore the various concepts in Angular. Like the exploration document said, I didn't built a fancy, functional application (this is my first week in Angular after all!) but I made a great demonstration on the various core elements of Angular, and some proof-of-concept components.

What framework did you choose and why?

I went ahead and chose **Angular** since it was the recommended framework for this first exploration, and it seems to be a popular and useful framework for web developement. I skimmed some Angular documentation and learned that Angular was started by an employee of, and now officially supported by Google. Therefore, I determined learning the basics of Angular is definitely a useful skill to have in web development. I was also intrigued by the functionality of Angular when building Single Page Applications (SLAs), as many popular web applications at least partially implement this concept. In addition, I know we will be exploring the MEAN Stack and Angular later in the course, and learning Angular will definitely put me ahead of the game in that sense.

What did you learn about this framework?

[0] History of Angular

Angular started as a JavaScript library **AngularJS**, originally intended to be a proprietory library for easier application development. AngularJS was first released in 2010. In effect, AngularJS works the same way as **JQuery**, in that one would include the library in a `html <script>` tag.

From the AngularJS wikipedia entry on Development History:

> *AngularJS was originally developed in 2009 by Miško Hevery at Brat Tech LLC as the software behind an online JSON storage service, that would have been priced by the megabyte, for easy-to-make applications for the enterprise. This venture was located at the web domain "GetAngular.com", and had a few subscribers, before the two decided to abandon the business idea and release Angular as an open-source library.*

Angular is a complete rewrite of **AngularJS**, which was led by a team at Google and first released in 2016 as Angular 2. This framwork is in active development, and is the industry standard opposed to the now outdated AngularJS. Angular, following in the footsteps of AngularJS, is open-source and freely distributed.

From the Angular wikipedia entry header:

> *Angular (commonly referred to as "Angular 2+" or "Angular v2 and above") is a TypeScript-based open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS.*

While Angular is growing in popularity, AngularJS has been declared as a legacy framework, and is no longer considered the industry standard. AngularJS was updated as versions 1.x.x, with Angular being updated as Angular 2, Angular 3, etc. It is currently on long-term support until 2021, after which it will be officially replaced by Angular.

From the AngularJS wikipedia entry on Development History:

> *In January 2018, a schedule was announced for phasing-out AngularJS: after releasing 1.7.0, the active development on AngularJS will continue till June 30, 2018. Afterwards, 1.7 will be supported till June 30, 2021 as long-term support.*

The latest stable Angular release is Angular 8.2.4, but a preview version of Angular 9 is available.

[1] What is Angular?

Angular is a platform (or *framework*) for developing web applications. While AngularJS is purely front-end, Angular is a framework for building *full-stack* web applications, with additional language features of server-side, object-oriented languages.

Angular is primarily **client-side**, meaning it operates like JavaScript: the code is run on the client application, not on the server. In fact, Angular is essentially an extension of HTML, CSS, and JavaScript, especially in its earlier features.

Angular can run simultaneously on multiple platforms, from web browsers, mobile devices, and native desktop applications.

Angular is used by many popular web sites and applications, including Netflix, PayPal, weather.com, and many other company web sites.

Angular is based on **TypeScript**, which is a *superset* of JavaScript/ECMAScript. This means that TypeScript is an *extension* of JavaScript/ECMAScript, therefore all JavaScript programs are also valid TypeScript programs. This is the same relationship between C/C++ (All C programs are also valid C++ programs).

From the TypeScript wikipedia entry header:

> *As TypeScript is a superset of JavaScript, existing JavaScript programs are also valid TypeScript programs.*

The main purpose of TypeScript is to add static typing to the language. JavaScript is ***dynamically typed***, meaning variables declared can change type dynamically, and type checking is performed at runtime, not compile time like C++ and Java. TypeScript adds the option to enable static typing in JavaScript programs. The biggest features of TypeScript are that it:

```
1) Adds the ability to denote variables as a specific type, and to perform static
type checking on the program.
2) Advanced types such as <Generics>, (tup,les), and enum types
3) Type inferencing
```

It makes sense that Angular is based upon TypeScript, as Angular originated as a JavaScript library.

Angular supports ***Single-Page Applications***, or **SPA**s. Single Page Applications function like desktop applications, and work by adjusting the current page's content asyncronously rather than navigating to other pages. This allows a single URL to access the entire application or portion of the application.

Lastly, Angular is based on the Model-View-Controller (MVC) architectural design. Specifically, it helps in creating web pages that dynamically update the View (web page) when the Model (data) is updated. Its architectural design is very similar to other languages that utilize this architecture such as Java (think: JavaFXML applications such as those we designed in CS3330).

[2] Components

Angular ***Components*** are reusable segments of code that perform a specific function. Components represent a single part of the application view. Components are complied together into ***Modules*** to form the hierarchial structure of an Angular application. This is similar to ***Classes*** in Object-Oriented languages such as C++ or Java, and ***Namespaces*** or ***Packages***, which are collections of Classes in C++ and Java, respectively.

An Angular Application has at least one Component called the ***Root Component*** named **AppComponent**. This acts as the 'entry point' to bootstrap the application and that connects each component in the hierarchy together. This is equivilant to the

```
public static void main(string[] args) {
    // run actual FXML Application through a method call here
}
```

method in a JavaFXML application, for instance.

Each component is comprised of:

```
1) A Class that contains the component's logic and data (the MC in MVC)
2) An associated html Template that defines how to display the component's data (th
e V in MVC)
```

Components are designed by the

```
@Component()
```

decorator. In between `()` you define a JSON object containing metadata about the component, such as `styleURLs` to specify .css files to include, `templateURL` to specify the .html file to use as a Template, and `selector` to designate the name of the html tag associated with that component. (More on Decorators later on)

```
// [2] Example component
@Component({
    name: "myComponent",
    templateURL: "templ.html",
    styleURLs: [ "./myapp.css" ]
})

export class MyComponent {
    // data and methods go here
    classcode = "CS4830";
}
```

Components are specified in a `.component.ts` file, with associated `.html` and `.css` files linked into the component. You can use

```
import
```

and

```
export
```

to import data and export the typescript class associated with the component.

Components can also have **Services**, which are classes that handle business logic (that is, the rules and logic of handling data for the ***business*** of the application).

[3] Modules

Angular Modules bundle many Angular Components together. Modules work as namespaces or packages do in C++/Java, but they also provide a context in which its components are compiled. In a sense, a module also acts like a POM file in Maven by specifying what needs to be compiled and to link dependencies.

Modules are used as means to separate concerns of the Application. Each Module should be associated with a particular ***Feature*** of the application, such as displaying a list of items in a store, displaying a navigation bar, etc. Ultimately, an Angular Application is a collection of ***Modules***, which are themselves collections of ***Components***.

Similar to components, an application has one or more than one module. The module that always exists is the *Root Module*, named **AppModule**. This serves as the bootstrapper for the rest of the application, and the central module where all other modules ultimately converge.

Modules are declared in a `.module.ts` file. Modules import classes from Components, and export a module Class. This class is modified by the `ngModule` decorator, which specifies properties of the module.

```
    import // all your modules
    @ngModule({
        imports: "yadayadayada",
        declarations: "Some class names go here",
        bootstrap: "The class to bootstrap the application"
    }
    export class AppModule { }
`
```

One neat feature of Modules is called ***lazy loading***. I have personally come across the term ***lazy evaluation*** in a functional programming language such as Haskell. Lazy loading is the principle that the application should only load Modules, Components and Services **when they are needed**. So instead of loading the entire application when the user visits the site, Angular only loads specific items when they are needed. This helps to spread the loading time over a longer period, and improves performance. This is similar to ***lazy evaluation*** in Haskell - the principle that a function should only evaluate an expression when its return value is needed.

What's nice about Modules and Components is that applications like VSCode and StackBlitz do most of this initial setup for us by creating the root Module/Component and creating a basic framework to expand upon.

Having the correct directory structure is also important for organizing Modules and Components! So under `src`, create directories for each module. In each module directory, create component directories for each component. Organizing files is important for readability and maintainability!

[4] Templates

Templates are a part of each Angular Component that define the view of that component. Templates are written as **HTML** and are usually contained in a separate file in the project directory.

Recall that each ***Component*** has a selector property designated in its @Component decorator. This specifies the name of the HTML tag that represents this element. Templates are used to specify the inner HTML of that tag. For example, if my template for a GroceryList component with the `grocery-list` selector was:

```
<ul>
    <li>Milk</li>
    <li>Eggs</li>
    <li>5 Pounds of Cookies</li>
</ul>
```

then if I were to specify `<grocerylist></grocerylist>` in another component (for example, my

AppComponent), it would render as such:

```
<grocery-list>
    <ul>
        <li>Milk</li>
        <li>Eggs</li>
        <li>5 Pounds of Cookies</li>
    </ul>
</grocery-list>
```

Essentially, this is like taking the output of the GroceryList component and setting the innerHTML of the parent `grocery-list` component to that output.

Templates should ideally be specified in a separate `.html` file, and linked to the component in the `@Component` decorator as the `templateURL` attribute. Alternatively, one can specify inline html using the `template` attribute, specifying the html in the `.ts` file. I wouldn't use this approach in practice, however, since templates can get large and complex, and separating the parts of the component improves readability.

[5] Interpolation

Interpolation allows the content of a component's template (view) to update according to the values in the component's class (model). To do this, one can specify a value with a name and value, then use `{{ double curly braces }}` in the template, specifying the name of the data. Whenever the data is changed, the value will update in the view.

So for example, in my grocery component, I can specify a string value in the class:

```
    public storeToGoTo = "HyVee";
```

and specify to put this data into the template:

```
    <strong>Go to {{ storeToGoTo }}!!!</strong>
```

The view will interpolate the value, and place "HyVee" in the place specified to read "Go to HyVee!!!" If I then update the value in the class to "Price Chopper", the view immediately updates asyncronously to this new value.

Interpolation is, in essence, the evaluation of an ***expression*** inside the `{{ double curly braces }}`. So, `{{ 2 + 3 }}` will evaluate to `5`. `{{ storeToGoTo }}` evaulates to HyVee because it pulls that value from the class, and the expression doesn't have more to it. But if I add `+ " in Columbia"` to my expression, Angular will concat that string as part of an expression to "HyVee in Columbia".

In this sense, you can also use method calls built in JavaScript and methods defined in the component's class. For example, call this method:

```
    happy() {
        return ":)";
    }
```

> `

using this expression:

```
    {{ happy() }}
```

The big limitation of interpolation is that they only evaluate set expressions. That is to say, you can't create local variables in an interpolation expression. You also can't access global JS objects such as window.location from an interpolation expression. You can solve these issues by adding methods in your class, and binding global JS values to data in your class. The other big issue is that interpolations ***only return strings***. So interpolation like this works for inner HTML and such, but we need other methods if we need to set something like a number, boolean value, or even some type of object.

[6] Decorators

Decorators, designated by the `@` symbol, are functions that modify classes like ***annotations*** in Java, such as this one to designate a functional interface:

```
@FunctionalInterface
public interface IFunction {
    public int doIt();
}

public class MyImpl {
    public static void main(String args[]) {
        IFunction impl = (int n)->n * 2 + 3;
        System.out.println(impl.doIt(5));
        // 13
    }
}
```

Here, the `@FunctionalInterface` notation performs a function on the `IFunction` interface, to create metadata that the interface is a functional one. Similarly, in Angular, **Decorators** are used to perform functions on TypeScript classes. For example, the `@Component` directive designates a class as belonging to a component, and binds metadata to it as a JSON Object:

```
@Component({
    name: "myComponent",
    templateURL: "templ.html",
    styleURLs: [ "./myapp.css" ]
})

export class MyComponent {
    // data and methods go here
}
```

There are many decorators in Angular used to designate the purpose of typescript classes, from components and modules to more. For example, `@Input()` designates a data as inputted from the parent component (this is done by setting the HTML attribute of the same name).

[7] Binding

Binding is the critical advantage offered by Angular: the updating of the view to reflect changes in the model. Interpolation is actually a form of binding! We bind data from the model to item(s) in the interpolation expression, and the view will update if we update the model. We *bind* the two elements together, almost like quantum entanglement, where two particles reflect changes in the other even across light-year distances! (this segment brought to you by my hobby in particle physics)

One-Way Binding is binding that only occurs one way: from model->view or vice versa. The following sections explore the different types of one-way binding and what they are used for.

Property Binding

Property binding is the binding of class data to *properties* of HTML elements, or vice versa.

It is important to distinguish between ***atrributes*** and ***properties***. Attributes are specified in the template's HTML and are the values used to initalize an element. Properties exist in the DOM (Document Object Model) and can change through user interaction. For example, the ***attribute*** value of this tag: `<input value="yeet">` is "yeet". This tells the browser to initialize the input's ***property*** value to "yeet". But if the user types in something else in the input box, the property value changes to that value, for example "whomst". The HTML ***attribute*** value remains unchanged, but the property value changes.

In property binding, we are binding to the ***property*** value, not the ***attribute*** value. To do this, we specify this in our template syntax:

```
<tag [property]="myData" etc. etc. etc.></tag>
```

> ```
> or
>
> <tag bind-property="myData" etc. etc. etc.></tag>
> ```

where ***property*** defines the HTML property we want to bind the data to, and "myData" references the data in the class to pull in. You can also use interpolation:

```
<tag property={{ expression }} etc. etc. etc.></tag>
```

I like the second approach, since it seems more readable to me, and interpolation is a very powerful tool! However, recall that interpolation only returns strings. This means we can't set properties that aren't strings, such as whether an input field is disabled or not. For those types of properties, we need to use the first option.

Class Binding

We can use *Property Binding* to bind data to an element's **class** property:

```
<tag [class]="myClasses"></tag>
```

However, this doesn't work when we pre-specify the class attribute in our template as well. Therefore, we can't use Property Binding by itself to allow an element to change classes based on some data change in the model. We use ***class binding*** to solve this problem. To specify class binding, use the following syntax:

```
<tag [class.classname]="someTest"></tag>
```

where classname is the name of a class that could be enabled or not enabled, and someTest refers to an expression that evaluates to true or false. The class is enabled when the expression evaluates to `true`, and vice versa. We can use this test to control if the class is enabled, and we can also specify other classes that are always enabled.

The best way to use class binding is with the `ngClass` directive (more on ***directives*** later). We specify an object that maps string keys (names of the classes to enable/disable) to expressions that evaluate to true/false. For example:

```
<p [ngClass]="classes" class="centered">My classes are determined by binding!</p>
```

uses this `classes` object to determine if these classes are enabled or not:

```
public classes = {
    "red": this.shouldBeRed,
    "blue": this.shouldBeBlue
}
```

Note here, however, that the "centered" class is ***always*** enabled, since it is specified in the `class` attribute! Therefore, we can use the `ngClass` directive to set/unset ***optional*** classes based on changing data.

Style Binding

Style Binding works very similar to the way the first type of Class Binding works: Specify a style property to bind to, and specify an expression that evaluates to the value of that style property. For example:

```
<tag [style.display]="shouldDisplay">
```

will read the shouldDisplay value and set that value as this element's display style element. However, I see this being less useful, as this type of inline styling should be avoided in favor of making classes and using normal class binding instead.

Event Binding

Event binding works in the opposite direction as the previous types of binding: it binds based on changes on the ***client*** side and responds to ***events*** that occur such as clicking a button, hovering the mouse over something, etc. Each event needs an ***Event Handler*** method that is invoked upon detecting the event. To specify event binding, use this syntax:

```
<tag (event)="eventHandler()"></tag>
```

where ***event*** is the event to listen for. This event is located on the DOM, and can be a standard event such as ***click***, ***hover***, etc. or a custom event (though this requres defining what that event is and is a complex topic ommitted in this exploration).

You can also specify to set data to a particular value in this expression, for example to set the value of `textToDisplay` to "You clicked my button!":

```
<button (click)="textToDisplay='You clicked my button!'">Click me!</button>
```

Combining Event Binding with other types of 1-way binding (Interpolation, Property, Class, Style) allows a psuedo-two-way interaction between the front-end view and back-end class model. The user can interact with the web page to alter the data, and the changed data will be updated in the view at the same time!

Template Reference Variables

TRVs are a way of sending additional information to an EventHandler when using Event Binding. For instance, you might want to send the value of an input field when a user trigger an on-click event by clicking a button. To do this, specify a TRV by using a pound sign plus a name:

```
<tag #myVar .....>
```

You can then access to the properties of that element and send these values as parameters to your event handler:

```
<button (click)="myEventHandler(myVar.property)">Click me to do something</button>
```

Two-Way Binding (Banana in a Boat)

pic unrelated

Two-Way Binding is important in situations where the view and model need to be always in sync. For example, when you fill out a form, you want the values of the inputs to match the model, so that the application can give feedback on the input, for example.

You can specify two-way binding using the `ngModel` directive. You surround ngModel with `[(A banana in a boat)]` - paranthesis inside square brackets. This combines property binding [] with event binding (). You then identify the property to bind to. Using this, you can keep the value of a property in sync in an input field and elsewhere, automatically updating the value!

```
<input [(ngModel)]="myTwoWayBind" value="Banana in a Boat">
```

> ```
> <p>{{ "The value is: " + myTwoWayBind }}</p>
> ```

[8] ***Structural*** Directives

Directives are specifications on how we **direct** Angular should alter the DOM. We have seen *Attribute* Directives before in Binding; i.e. changing the styling classes and content of HTML elements based on the content of the model. The directives in this section (also known as *Structural* Directives) direct how Angular should ***add and remove*** HTML elements entirely, i.e. alter the ***structure*** of the view.

There are three main structural directives:

```
1) *ngIf
2) *ngSwitch
3) *ngFor
```

These directives act similar to the control structures found in most high-level programming languages: if statements, switch statements, and for-each loops. ngIf and ngSwitch are used to render elements depending on some boolean expressions/conditions, and ngFor is used to create a series of HTML elements through iteration.

ngIf

The `*ngIf` directive is used to only render an HTML element if an expression is evaulated to true.

```
<tag *ngIf="someExpression"></tag>
```

The element displays normally if `someExpression` evaluates to `true`. If it evaluates to `false`, the HTML element is never created and omitted from the view. This is different than just togging `display: none` or similar; the HTML element ***doesn't even exist***.

With ngIf, you can also specify an HTML to render ***instead*** if the expression evaluates to false.

```
<tag *ngIf="someExpression; else elseElement"></tag>
```

We then designate another HTML element with a Template Reference Variable:

```
<anothertag #elseElement></anothertag>
```

You can also specify `then` as a means of creating a `if-then-else` structure.

```
<tag *ngIf="someExpression; then thisElement; else elseElement"></tag>
```

ngSwitch

The `*ngSwitch` directive works the same way as a `switch` statement in any popular imperative programming language. Angular looks at some data, then selects what to render based on that data. You can and should also specify a default case to display if no patterns match.

```
<tag [ngSwitch]="data">
```

```
    <item *ngSwitchCase="option1">Option 1</item>
    <item *ngSwitchCase="option2">Option 2</item>
    <item *ngSwitchCase="option3">Option 3</item>
    <item *ngSwitchDefault>This will be selected if data is not any of the above op
tions</item>
</tag>
```

The DOM will display the element that matches the correct case, or the default if there is no match.

ngFor

The `*ngFor` directive works the same way a **for-each** loop (not a FOR loop!) works in any other programming language. It essentially iterates over a collection such as a list and constructs HTML elements containing that data. For example, this `*ngFor` constructs an ordered list of all letters in the alphabet:

```
<ol>
    <li *ngFor="let letter of letters">{{letter + " "}}</li>
</ol>
```

where `letters` is an array of alphabet characters defined as data in the class:

```
public letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'];
```

This will create a `<li>` element for each element in the list, with its content determined through interpolation as the current iterative element `letter`. This can be incredibly useful for rendering lists from the model to the view.

You can also specify data about the collection to use in your output, i.e. `index as i` will bind the current index of the element to the data `i`. There is also `first` and `last`, which refer to the first and last elements of the array respectively, and even functions such as `odd` and `even`.

[9] Routing

Routing is what empowers Angular when developing ***Single Page Applications***, since routing allows us to update the currently rendered view without fully navigating away from the current page. Routing is useful when we want multiple views in an application, and to navigate to different pages from a single overarching view.

Routing can either look at the URL when first entered to determine which view to render, or Angular can switch views by responding to the user, such as when a user clicks a link. To add routing to your application, first add this tag to your index.html's `<head>` tag:

```
<base href="/">
```

This tells Angular how to format URLs when routing in the application. We also need to create a routing Module named `app-routing.module.ts` to configure our routing. There's a bit of configuration here, which I'll save the details for in the code. The most important thing is to declare in the routes list JSON objects that specify the path to look for, and the component to load when that path is detected. Then, add the `<router-outlet>` tag into some view. Angular will see if an entered URL matches a route, and if so, plugs the appropriate view into the router-outlet tag!

Note that if no path match was found, the application automatically redirects to the base URL we defined in `index.html`.

This works by itself if you type the route in the browser, but we also want to allow navigation through buttons and interaction on the webpage itself. This also enables seamless transitions between different views, as other parts of the view aren't reloaded. To do this, we can enclose links (`<a>` tags) in a `<nav>` tag. You can then specify the `routerLink` property to designate the path to route to when the button is clicked. You can also specify the `routerLinkActive` property to enable a style class when this particular route is active.

```
<nav>
    <a routerLink="/routing" routerLinkActive="activeLink">Route 1</a>
    <a routerLink="/routing2" routerLinkActive="activeLink">Route 2</a>
</nav>
```

This is **incredibly** useful to create a navigation bar, which the user can use to navigate throughout a single page application. I implemented something like this in CS2830 for my final project using PHP, but it still required navigating away and doing a whole reload! This is ***wayyyy*** better, and I'm excited to use this concept in the future!

There's also a lot more about Routes such as wildcard routing and redirecting that I want to look into later, but I feel like I get the basics of routing here.

[10] Forms

Forms are a huge concept in Angular. Just owing to what I've already seen in Angular, the framework is extremely useful when designing a web application that uses forms. Being able to keep the model and view in sync, and to edit the content of a page dynamically based on input values in forms, allows us to create forms that respond quickly to user input, which is a great tool to have! I wouldn't be surprised if Google Forms itself is implemented in Angular.

The big feature of forms in Angular seems to be ***feedback***. Forms such as those we've used with HTML/PHP are not ideal, since the data in the view is separate from the model. The model only recieves these values when the user submits, and the data is sent through a GET/POST request to a PHP action. However, since Angular allows us to bind the two together, we can create apps that immediately react to input we enter. We can use this to validate input, and display visual feedback to the user ***while*** they are filling out a form!

There are two types of forms in Angular: ***Reactive*** and ***Template-Driven***. ***Reactive*** forms are based more in the component's TS class, whereas template-driven forms primarily live in the template of the component. Which one to use mostly relies on what you're looking for out of the form.

Reactive Forms

Reactive forms are mostly comprised of code in the component TS class. Reactive forms mostly involve using methods and data in the class to edit the form data, whereas template-driven forms use HTML and other directives in the template file.

I didn't delve into much of what Reactive Forms are, but I understand they are mostly based in the class logic rather than in the HTML. This is a topic I want to explore later on; I simply didn't have enough time to cover all of this in a week, however.

Template-Driven Forms

Template-Driven forms are mostly comprised of HTML in the template file. TD forms track the input automatically using [(two-way binding)] in order to perform input validation and generate visual feedback. These can be great for many cases, but since this lives in HTML, it is hard to unit test and can be increasingly and overwhelmingly verbose to read. Therefore, TD Forms are great for simple forms, but not complex ones.

One creates form elements the same way one makes HTML forms; however, you can use the features of Angular to implement binding, and to create the HTML based on the data you provide. For example, you can use `*ngFor` to create `<option>` tags for each of your options you define in a list!

Angular also automatically creates an `ngForm` directive in each `form`. You can reference its properties using a Template Reference Variable you define for the form.

Angular also automatically updates classes of form elements based on certain properties:

```
- Has the control been visited? ng-touched / ng-untouched
- Has the control's value been changed? ng-dirty / ng-pristine
- Is the control's value valid? ng-valid / ng-invalid
```

You can manipulate the form styling to display feedback based on which of these classes an element has. For example, you can combine this with an `*ngIf` directive to show an error message only if the name isn't valid. You can add validators in the HTML of the `<input>` tag.

Lastly, the `(ngSubmit)` directive allows you to link an event handler that runs when the form is submitted.

Overall, I didn't have much time to delve into all the idiosyncrocies of Angular Forms; however, I feel as though I have a good basic understanding, and I was able to create a simple responsive form on my StackBlitz playground!

[11] Pipes

Pipes, previously known as *filters* in Angular 1.x, work by "piping" input through a "filter" function that modifies the data before displaying it. I have been previously familiar with piping in terms of the Linux control operator, and this works similarly, though not exactly.

Piping is specified in the .html file, enclosed in interpolations `{{ }}`. You can read this syntax as:

```
{{ inputvalue | pipefunction }}
```

There are a lot of pipe functions to explore, but some include `currency` to turn a number into currency representation, `date:[format]` to format dates, `json` to represent JSON objects, string manipulation, and more!

[12] Dependencies / Dependency Injection

Dependencies are a rough beast to handle, especially on the level of enterprise-level development (where we're all headed, in some form or another, probably). I dealt with this during my internship at Cerner, and it can be quite the nightmare if managed poorly. Many of the blocks I had to get through were due to new, incompatible versions of dependencies that I had no idea about. Making dependement management easy is hard, though. That's why *Dependency Injection* (DI) is so important.

Dependencies can either be ***tighly coupled*** or ***loosely coupled***. Consider the following demonstratative images:

Tighly coupled:

Loosely coupled:

The difference here is that the first power adaptor is hard-wired; you can't remove or replace the brick in the middle! Which means the system is entirely dependent on that brick being and staying that exact same brick. The seocnd adaptor has a changeable plug. Meaning, you can change the brick you use (as long as it uses the same plug, or ***interface***) and reuse other components like the cables and wall adaptor.

This demonstrates the difference between ***tightly*** and ***loosely*** coupled dependencies. For example, if you write a component that utilizes a very specific class by referencing it directly, including invoking its specific constructor/methods by name, this is bad! The dependency is tightly coupled; if someone changes that class by editing its method signatures, you break your code. If your dependences are ***loosely*** coupled, you reference a class that implements some abstract ***interface***. Therefore, you can count on the class to implement those exact methods, and you don't need to worry about those private implementation details. Plus, you can use other classes that implement the same interface just by calling that class instead! It's pretty sweet!

I only touched upon dependency injection because it's a very complex topic that warrants its own week of work. But, it's a great principle to consider going forward.

What resources did you utilize?

- Angular's main web site (https://angular.io)

Evidently, the best place to learn a new technology is from its official documentation. This site provided technical details for the actual Angular platform, which was important for me to develop my examples.

- **Codevolution Angular Tutorial Playlist on YouTube (https://www.youtube.com/watch?v=0eWrpsCLMJQ&list=PLC3y8-rFHvwhBRAgFinJR8KHIrCdTkZcZ)**

A video series on YouTube going over the basics of Angular. I found this to be a great watch to understand the basics and help write this documentation. I primarily used this source, following along the playlist and coding as I went.

- Angular's wikipedia page (https://en.wikipedia.org/wiki/Angular_(web_framework))

The wikipedia page was useful for learning about the ***history*** of Angular, and the evolution of the technology from its conception as a JavaScript library to today. I did not use this documentation for technical details, but as a way to familiarize myself with ***what*** Angular is.

- AngularJS's wikipedia page (https://en.wikipedia.org/wiki/AngularJS)

Once I learned that ***Angular*** is a rewrite of ***AngularJS***, I wanted to research on the history of AngularJS and how it relates to Angular itself.

- TypeScript's wikipedia page (https://en.wikipedia.org/wiki/TypeScript)

Useful for learning what TypeScript is and how Angular relates to TypeScript's static typing principles.

- SitePoint Angular Introduction (https://www.sitepoint.com/angular-introduction/)

Useful introduction to what exactly Angular ***is*** and the advantages it brings to the table of web development.

- JavaTPoint Angular 7 Tutorial (https://www.javatpoint.com/angular-7-tutorial)

A very useful, W3Schools-esque tutorial with examples and documentation on the concepts of Angular.

- StackBlitz (https://stackblitz.io)

A wonderful tool for building and testing angular applications without the trouble (and cost) of actually running it.

What do you still want to learn about this framework?

1) Services! I didn't get to touch on the subject during my exploration, but the concept definitely seems important, especially from the point of view of developing a full web application. It seems like services are important when handling persistant data, instead of the instance data used in this exploration, which only persists during the viewing of the web page.

2) More info about Forms! There's an overload of info about creating Forms in Angular, and I only scratched the surface! I feel as though I'd need to study services first, since forms send data to services, which actually store and process the data we send through the form.

3) Unit Testing with Karma and Jasmine. I've been exposed to unit testing before using jUnit, jBehave, Hamcrest, etc. in Java, but this was only last summer.

4) Developing automated tests is crucial for software maintainability, and it's underdeveloped in most undergraduate CS students. Since I'm familiar with unit testing in general, it probably wouldn't take that long to learn a new framework, but it would be an important step to take when developing a large application.

5) Looking into using the Angular CLI for development. I used StackBlitz to develop my code for this exploration, but this abstracts a lot of the finer details away in the process. Ultimately, knowing the Angular CLI commands would be immensely helpful if I was developing/deploying an Angular app on, say, my EC2 instance.

6) How Angular works with node/nodeJS. I have only dabbled in using node for writing a basic Discord bot service over the summer. I still have a lot to learn about nodeJS, however, and from what I've seen, Angular as it is today heavily utilizes node for data storage and back-end tools. I feel like what I've learned is beneficial on the front-end, but I would need to look into how Angular uses Node (and MongoDB/Express.js for that matter) to build a full MEAN stack app.

7) Some more study in Javascript/Typescript. I haven't really used the object-oriented functionality of Javascript -> Typescript in Angular, since most of my JS has been relatively simple and used jQuery. While I understand well the concepts of OO design, and I could follow along very well, taking a step back from Angular to just JS/TS could help me better work in Angular.

8) Ultimately, just working more in this framework. Obviously a week isn't enough time to really develop something spectacular. Simply playing around in StackBlitz and doing some further research into what Angular has to offer will lead me to be better at Angular.

What problems did you run into?

The largest issue for me was getting the environment set up right. At first, diving right into an Angular application looked like a huge mess. I think this is because the default Angular app already contains a lot of files: html, css, ts, json, etc. I used the codevolution tutorials, which went through what each of these files were, and I understood the environment much better!

I also had a small problem at first with adding components and functionality to my application. I quickly realized, however, that Stackblitz (and VSCode, I presume) have commands that will generate things like new components and modules for you! I also liked that Stackblitz updated existing files to add import statements, so I could focus on the application logic rather than the nitty-gritty details.