BUILDING MOBILE APPLICATIONS WITH IONIC & ANGULAR

JOSHUA MORONY

Lesson 4: Decorators

Each class (which we will talk about in the next section) you see in an Ionic application will have a **decorator**. A decorator looks like this:

```
@Component({
    someThing: 'somevalue',
    someOtherThing: [Some, Other, Values]
})
```

They definitely look a little weird, but they play an important role. Their role in an Ionic application is to provide some *metadata* about the class you are defining, and they always sit directly above your class definition (again, we'll get to that shortly) like this:

```
@Decorator({
    /*meta data goes here*/
})
export class MyClass {
    /*class stuff goes here*/
}
```

The `@Component` is just one type of decorator, but there are other types as well. I have

used `@Decorator` above as a generic example, but keep in mind that there isn't a decorator literally called `@Decorator` (I'm just using it as a placeholder).

This is the only place you will see a decorator, they are used purely to add some extra information to a class (i.e. they "decorate" the class). So, let's talk about exactly how and why we would want to use these decorators in an Ionic application.

The decorator name itself is quite useful, here are a few you might see in an Ionic application:

- @Component
- @Pipe
- @Directive
- @Injectable

Just the decorator itself is enough to prove extra information about the class:

```
@Injectable()
export class MyService {

}
```

But we can also supply an object to the decorator to provide more information about the class. Here's the most common example you'll see in your applications:

```
@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

}
```

With the help of the decorator, this class is identified as a component and that:

- The tag for the component will look like this: `<app-page-home></app-page-home>`
- It needs to fetch its template from `home.page.html`, which will determine what the user will actually see on the screen (we'll be getting into that later as well).
- It needs to fetch the styles for this component from `home.page.scss`

These are the most commonly used properties for the `@Component` decorator, but there are more. If you've got a super simple template, maybe you don't even want to have an external template file, and instead define your template like this:

```
@Component({
    selector: 'app-page-home',
    template: `<p>Howdy!</p>`,
    styleUrls: ['home.page.scss']
```

```
})
export class HomePage {

}
```

Some people even like to define large templates using the `template` property. Since ES6 supports using backticks (the things surrounding the template above) to define multi-line strings, it makes defining large templates like this a viable option if you prefer (rather than doing something ugly like concatenating a bunch of strings).

Now that we've covered the basics of what a decorator is and what it does, let's take a look at some specifics.

Common Decorators in Ionic Applications

There are quite a few different decorators that we can use. In the end, their main purpose is simply to describe *what* the class we are creating *is* so that it knows what needs to be imported to make it work.

Let's discuss the main decorators you are likely to use, and what the role of each one is. We're just going to be focusing on the role of the decorator, for now, we will get into how to actually build something useable by defining the class in the next section.

@Component

I think the terminology of a *component* can be a little confusing in Ionic. As I mentioned,

our application is made up of a bunch of components that are all tied together. These components are contained within folders inside of our main **src** folder, and they look like this:

home

- home.page.ts
- home.page.html
- home.page.scss

NOTE: Depending on how your application is set up, your pages may consist of more files than just these three. However, these three files are the key parts of creating a component.

A **@Component** is not specific to Ionic, it is used generally in Angular. A lot of the functionality provided by Ionic is done through using components. In Ionic, for example, you might want to create a search bar, which you could do by using one of the components that Ionic provides like this:

```
<ion-searchbar></ion-searchbar>
```

You simply add this custom tag to your template. Ionic provides a lot of components but you can also create your own custom components, and the decorator for that might look something like this:

```
@Component({
```

```
    selector: 'my-cool-component'
})
```

which would then allow you to use it in your templates like this:

```
<my-cool-component></my-cool-component>
```

The `@Component` decorator will likely be the one you use the most. Even if we aren't creating our own custom components like lists and buttons, each page in your application is still its own component. Always keep in mind that your whole application is just components within components within components. Your entire application is a component, and inside of that component you have **pages** for your application which are also components, and inside of those components, you might use even more components like Ionic's buttons and lists.

NOTE: Technically speaking a component should have a class definition and a template. Things like pipes and services/providers aren't viewed on the screen so have no associated template, they just provide some additional functionality. Even though these are not technically components you may often see them referred to as such, or they may also be referred to as services or providers.

@Directive

The **@Directive** decorator allows you to create your own custom directives. We've

touched on the concept of a directive briefly, but basically, it allows you to attach some behaviour to a particular component/element. Typically, the decorator would look something like this:

```
@Directive({
    selector: '[my-selector]'
})
```

Then in your template, you could use that selector to trigger the behaviour of the directive you have created by adding it to an element:

```
<some-element my-selector></some-element>
```

It might be a little confusing as to when to use **@Component** and **@Directive**, as they are both quite similar. The easiest thing to remember is that if you want to modify the behaviour of an existing component use a **directive**, if you want to create a completely new element/component use a **component**. As I mentioned before, technically speaking, a component is still considered to be a directive (it is just a directive with its own template, rather than a directive that attaches to another component).

@Pipe

@Pipe allows you to create your own custom pipes to filter data that is displayed to the

user, which can be very useful. The decorator might look something like this:

```
@Pipe({
  name: 'myPipe'
})
```

which would then allow you to implement it in your templates like this:

```
<p>{{someString | myPipe}}</p>
```

Now `someString` would be run through your custom `myPipe` before the value is output to the user. The main role of a pipe is to take some data in (`someString` in this case) and then return that data in a different format. A common example would be doing things like converting the format of currencies or dates to display in a more friendly manner (e.g. 13th of April, 2018 rather than 2018/04/13).

@Injectable

An **@Injectable** allows you to create a service for a class to use. A common example of a service created using the **@Injectable** decorator, and one we will be using a lot when we get into actually building the apps is a **Data Service** that is responsible for fetching and saving data. Rather than doing this manually in the classes for your pages, you can inject your data service into any number of classes you want, and call helper functions from that

Data Service. Of course, this isn't all you can do, you can create a service to do anything you like.

An **@Injectable** will often just look like a normal class with the **@Injectable** decorator tacked on at the top:

```
@Injectable({
  providedIn: 'root'
})
export class DataService {

}
```

In older versions of Angular, we would need to add our `@Injectable` classes to the `providers` array in the root module of the application. We can now use this `providedIn` property to make it available to use in our application. By providing the service in `root` it will be provided to the root module of the application, which will allow it to be used as a "singleton" throughout the entire application. A singleton means that one instance of the class will be instantiated for use through the entire application - so if you access the service from two different pages, you are going to be accessing the same data. If you set some variable in the service on `PageOne`, you could then grab that variable from `PageTwo` since it is a singleton. This is typically how you would use services in your application.

Summary

The important thing to remember about decorators is: *there's not that much to remember*.

Decorators are powerful, and you can certainly come up with some complex looking configurations. Your decorators may become complex as you learn more about Ionic, but in the beginning, the vast majority of your decorators will probably just look like this:

```
@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

}
```

I think a lot of people find decorators off-putting because at a glance they look pretty weird, but they look way scarier than they actually are. In the next lesson, we'll be looking at the decorator's partner in crime: the class. The class definition is where we will do all the actual work, remember that the decorator just sits at the top and provides a little extra information.