Custom Directives

Overview

By this point you should be able to create your own custom component as well as use a set of *built-in* Angular directives.

Now it's time to learn how to build your own *custom* directives.

But you might be surprised to hear that you've *already* created a custom directive. That's because *Components are Directives*.

Components have all the features of Directives but also have a view, that is to say they have a template and some HTML that is injected into the DOM when we use it.

Another difference is that a single HTML element can only have a *single component* associated with it. However a single element can have *multiple directives* associated with it.

Lets continue with our joke example app and create a directive which shows the punchline of the joke when the user hovers over the card.

In this section you will learn:

- How to create custom directives using the `@Directive` decorator.
- How directives can both listen to events and change properties of the host element they are associated with.
- How we can configure a directive so that it can take `inputs` when it's defined on an element, like `[aDirective]={config:'value'}`

Creating a custom directive

In this lecture we are going to create our very own custom directive.

Learning Objectives

- Know how to create a basic directive using the `@Directive` decorator.
- Know how to use selectors to associate an element with a directive based on an *attribute*.
- Know how to interact with the raw DOM element of the associated element from the directive.

Directive decorator

We'll call our directive `ccCardHover` and we'll attach it to the card block like so:

```
<div class="card card-block" ccCardHover>...</div>
```

The Angular team recommends using directives as attributes, prefixed with a namespace. We've prefixed our directive with the namespace 'cc'.

We create directives by annotating a class with the `@Directive` decorator.

Lets create a class called `CardHoverDirective` and use the `@Directive` dectorator to associate this class with our attribute `ccCardHover`, like so:

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
  selector:"[ccCardHover]"
})
class CardHoverDirective { }
```

Attribute selector

The above code is very similar to what we would write if this was a component, the first striking difference is that *the selector is wrapped with `[]`*.

To understand why we do this we first need to understand that the selector attribute uses *CSS matching rules* to match a component/directive to a HTML element.

In CSS to match to a specific element we would just type in the name of the element, so `` input {⋯ }`or `p {⋯} ``.

This is why previously when we defined the selector in the `@Component` directive we just wrote the *name* of the element, which matches onto an element of the same name.

If we wrote the selector as `.ccCardHover`, like so:

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
  selector:".ccCardHover"
})
class CardHoverDirective { }
```

Then this would associate the directive with any element that has a *class* of ccCardHover, like so:

```
<div class="card card-block ccCardHover">...</div>
```

We want to associate the directive to an element which has a certain attribute.

To do that in CSS we wrap the name of the attribute with `[]`, and this is why the selector is called `[ccCardHover]`.

Directive constructor

The next thing we do is add a constructor to our directive, like so:

```
import { ElementRef } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef) {
  }
}
```

When the directive gets created Angular can inject an instance of something called `ElementRef` into its constructor.

> How this works is called *Dependency Injection*, it's a really important aspect of Angular and we discuss this in detail in a later section.

The `ElementRef` gives the directive *direct access* to the DOM element upon which it's attached.

Let's use it to change the background color of our card to gray.

`ElementRef` itself is a wrapper for the actual DOM element which we can access via the property `nativeElement`, like so:

```
el.nativeElement.style.backgroundColor = "gray";
```

This however assumes that our application will always be running in the environment of a browser.

Angular has been built from the ground up to work in a number of different environments, including server side via node and on a native mobile device. So the Angular team has provided a *platform independent* way of setting properties on our elements via something called a `Renderer`.

script.ts

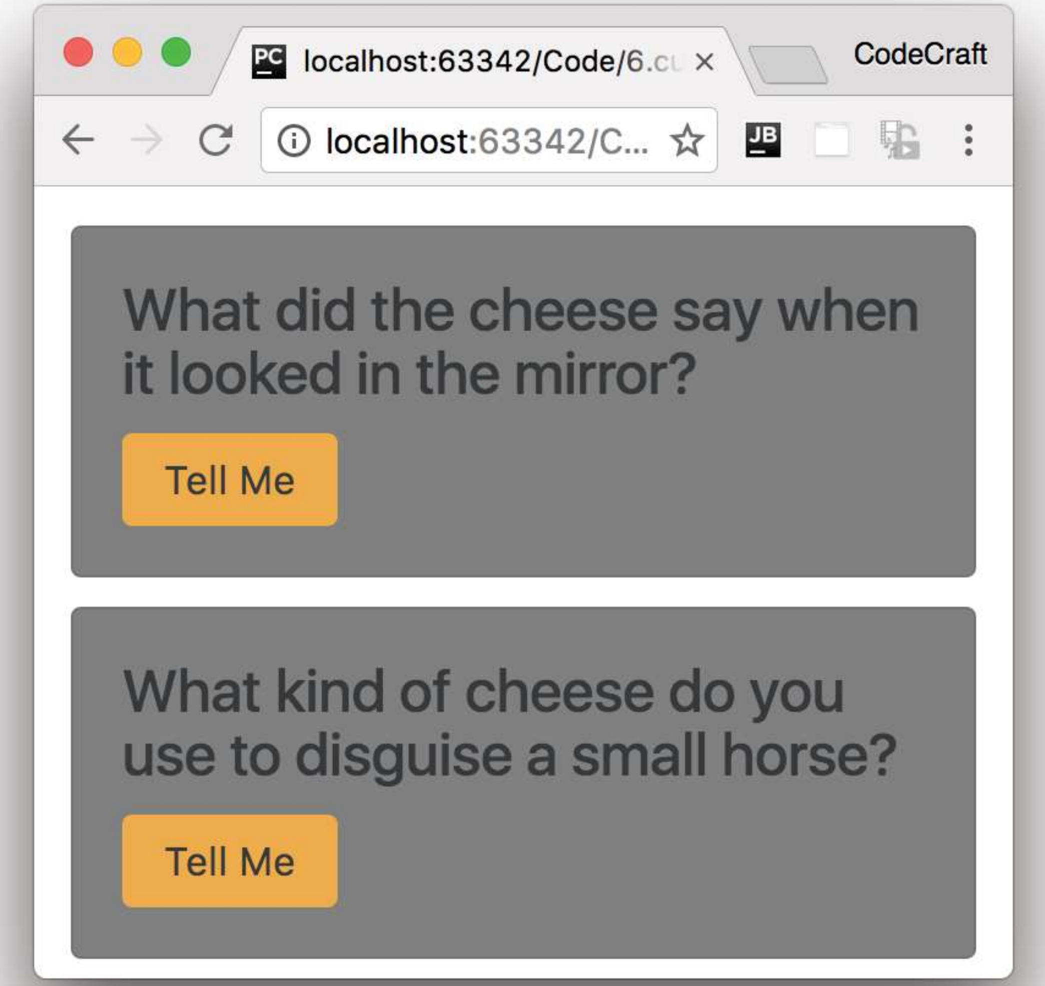
```
import { Renderer } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef,
              private renderer: Renderer) { ①
    renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray'); ②
  }
}
```

① We use *Dependency Injection* (DI) to inject the `renderer` into our directives constructor.

② Instead of setting the background color directly via the DOM element we do it by going through the `renderer`.

❗ In the future if we wanted to render our application on a platform other than a web browser then the `Renderer` calls the appropriate functions to change the background color on that platform. We are not limited to only being run in a web browser with a DOM.

Running the application now show this:



What did the cheese say when it looked in the mirror?

What kind of cheese do you use to disguise a small horse?

Summary

We create a directive by decorating a class with the `@Directive` decorator.

The convention is to associate a directive to an element via an *attribute selector*, that is the name of the attribute wrapped in `[]`.

We can inject a reference to the element the directive is associated with to the constructor of the directive. Then via a `renderer` we can interact with and change certain properties of that element.

The above is a very basic example of a custom directive, in the next lecture we'll show you how you can detect when the user hovers over the card and a *better* way of interacting with the host element.