## Content Projection

**Goals**

Change our application so that a developer who is re-using our JokeComponent can also configure *how* a joke will be rendered on the screen.

**Learning Objectives**

* What is content projection and why might we want to use it.
* How to project content using the ng-content tag.
* How to project multiple pieces of content using css selectors.

**Motivation**

Lets say someone else wanted to use our JokeComponent but instead of displaying the punchline in a <p> tag they wanted to display it in a larger <h1> tag.

Our component right now doesn’t let itself be *reused* like that.

However we can design our component with something called *content projection* to enable it to be customised by the component or developer who is using it.

**Content projection**

If we add the tag <ng-content></ng-content> anywhere in our template HTML for our component. The inner content of the tags that define our component are then *projected* into this space.

So if we changed the template for our JokeComponent to be something like:

<

div class="card card-block"

>

<h4 class="card-title">{{ data.setup }}</h4>

<p class="card-text"

[hidden]="data.hide">

<ng-content></ng-content>

①

</p>

<a class="btn btn-primary"

(click)="data.toggle()">Tell Me

</a>

<

/div

>

① We’ve replaced {{data.punchline}} with <ng-content></ng-content>

Then if we changed our JokeListComponent template from this:

<

joke \*ngFor="let j of jokes" [joke]="j"></joke

>

To this:

<

joke \*ngFor="let j of jokes" [joke]="j"

>

<h1>{{ j.punchline }}</h1>

①

<

/joke

>

① In-between the opening and closing joke tags we’ve added some HTML to describe *how* we want the punchline to be presented on the screen, with a <h1> tag.

The <h1>{{ j.punchline }}</h1> defined in the parent JokeListComponent, replaces the <ngcontent></ng-content> tag in the JokeComponent.

This is called *Content Projection* we *project* content from the parent Component to our Component.

If we create our components to support content projection then it enables the consumer of our component to configure *exactly* how they want the component to be rendered.

The downside of content projection is that the JokeListComponent doesn’t have access to the properties or methods on the JokeComponent.

So the content we are projecting we can’t bind to properties or methods of our JokeComponent, only the JokeListComponent.

**Multi-content projection**

What if we wanted to define *multiple* content areas, we’ve got a setup and a punchline lets make both of those content projectable.

Specifically want the setup line to always end with a ? character.

We are using this example for demonstration purposes only. This problem could

easily be solved in a number of other ways, all of them easier than using content projection.

We might change our JokeListComponent template to be something like:

<

joke \*ngFor="let j of jokes" [joke]="j"

>

<span>{{ j.setup }} ?</span>

<h1>{{ j.punchline }}</h1>

①

<

/joke

>

But in our JokeComponent template we *can’t* simply add two <ng-content></ng-content> tags, like so:

<

div class="card card-block"

>

<h4 class="card-title">

<ng-content></ng-content>

</h4>

<p class="card-text"

[hidden]="data.hide">

<ng-content></ng-content>

</p>

<a class="btn btn-primary"

(click)="data.toggle()">Tell Me

</a>

<

/div

>

Angular doesn’t know *which* content from the parent JokeListComponent to project into which ngcontent tag in JokeComponent.

To solve this ng-content has another attribute called select.

If you pass to select a css matching selector, it will extract *only* the elements matching the selector from the passed in content to be projected.

Let’s explain with an example:

<

div class="card card-block"

>

<h4 class="card-title">

<ng-content select="span"></ng-content>

①

</h4>

<p class="card-text"

[hidden]="data.hide">

<ng-content select="h1"></ng-content>

②

</p>

<a class="btn btn-primary"

(click)="data.toggle()">Tell Me

</a>

<

/div

>

①<ng-content select="span"></ng-content> will match <span>{{ j.setup }} ?</span>.

②<ng-content select="h1"></ng-content> will match <h1>{{ j.punchline }}</h1>.

That however can be a bit tricky to manage, lets use some more meaningful rules matching perhaps by class name.

To support this lets change our parent components content to identify the different elements by classnames, like so:

<

joke \*ngFor="let j of jokes" [joke]="j"

>

<span class="setup">{{ j.setup }} ?</span>

<h1 class="punchline">{{ j.punchline }}</h1>

①

<

/joke

>

**Summary**

Sometimes the nature of a component means that the consumer would like to customise the view, the presentation of data, in a unique way for each use case.

Rather than trying to predict all the different configuration properties to support all the use cases we can instead use content projection. Giving the consumer the power to configure the presentation of the component as they want.

<

div class="card card-block"

>

<h4 class="card-title">

<ng-content select=".setup"></ng-content>

①

</h4>

<p class="card-text"

[hidden]="data.hide">

<ng-content select=".punchline"></ng-content>

②

</p>

<a class="btn btn-primary"

(click)="data.toggle()">Tell Me

</a>

<

/div

>

**Listing**

<http://plnkr.co/edit/g5nuoD8m4e1tTp7gDW4N?p=preview>

*script.ts*

import {platformBrowserDynamic} from '@angular/platform-browser-dynamic'; import {Component, NgModule, Input, Output, EventEmitter, ViewEncapsulation} from

'@angular/core';

import {BrowserModule} from '@angular/platform-browser';

class Joke {

public setup: string;

public punchline: string;

public hide: boolean;

constructor(setup: string, punchline: string) {

this.setup = setup;

this.punchline = punchline;

this.hide = true;

}

toggle() {

this.hide = !this.hide;

}

}

@Component({

selector: 'joke-form',

templateUrl: 'joke-form-component.html',

styleUrls: [

'joke-form-component.css'

],

encapsulation: ViewEncapsulation.Emulated

// encapsulation: ViewEncapsulation.Native

// encapsulation: ViewEncapsulation.None

})

class JokeFormComponent {

@Output() jokeCreated = new EventEmitter<Joke>();

createJoke(setup: string, punchline: string) {

this.jokeCreated.emit(new Joke(setup, punchline));

}

}

@Component({

selector: 'joke',

template: `

<

div class="card card-block"

>

<h4 class="card-title">

<ng-content select=".setup"></ng-content>

</h4>

<p class="card-text"

[hidden]="data.hide">

<ng-content select=".punchline"></ng-content>

</p>

<a class="btn btn-primary"

(click)="data.toggle()">Tell Me

</a>

<

/div

>

`

})

class JokeComponent {

@Input('joke') data: Joke;

}

@Component({

selector: 'joke-list',

template: `

<

joke-form (jokeCreated)="addJoke($event)"></joke-form

>

<

joke \*ngFor="let j of jokes" [joke]="j"

>

<span class="setup">{{ j.setup }}?</span>

<h1 class="punchline">{{ j.punchline }}</h1>

<

/joke

>

`

})

class JokeListComponent {

jokes: Joke[];

constructor() {

this.jokes = [

new Joke("What did the cheese say when it looked in the mirror", "Hello-me

(

Halloumi)"),

new Joke("What kind of cheese do you use to disguise a small horse", "Mask-a-

pony (Mascarpone)"),

new Joke("A kid threw a lump of cheddar at me", "I thought

‘

That

’

s not very

mature

’

"),

];

}

addJoke(joke) {

this.jokes.unshift(joke);

}

}

@Component({

selector: 'app',

template: `

<

joke-list></joke-list

>

`

})

class AppComponent {

}

@NgModule({

imports: [BrowserModule],

declarations: [

AppComponent,

JokeComponent,

JokeListComponent,

JokeFormComponent

],

bootstrap: [AppComponent]

})

export class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);