Quickstart

ES6 JavaScript & TypeScript

Angular CLI

Components

- Overview (/courses/angular/components/overview/)
- Architecting with Components (/courses/angular/components/architecting/)
- Templates, Styles & View Encapsulation (/courses/angular/components/templates-styles-view-encapsulation/)
- Content Projection (/courses/angular/components/content-projection/)
- Lifecycle Hooks (/courses/angular/components/lifecycle-hooks/)
- ViewChildren & ContentChildren (/courses/angular/components/viewchildren-and-contentchildren/)
- Wrapping Up (/courses/angular/components/summary/)
- Activity (/courses/angular/components/activity/)

Built-in Directives

Custom Directives

Reactive Programming with RxJS

Pipes

Forms

Dependency Injection & Providers

HTTP

Routing

Unit Testing

Advanced Topics

ViewChildren & ContentChildren

AUTHOR: Asim

In this video I'm using an online editor called Plunker (https://plnkr.co/) to write and run Angular code. The book and code has since been updated to use StackBlitz (https://stackblitz.com) instead. To understand more about why and the differences between read this (/courses/angular/quickstart/overview/#_plunker_vs_stackblitz).

‹ LIFECYCLE HOOKS (/COURSES/ANGULAR/COMPONENTS/LIFECYCLE-HOOKS/)

WRAPPING UP ›

(https://github.com/codecraft-tv/angular-course/tree/current/4.com...) (https://stackblitz.com/github/codecraft-tv/angular-course/tre...)

TABLE OF CONTENT

- Learning Objectives
- Example application
- ViewChild
- ViewChildren
- ViewChild referencing a template local variable
- ContentChild & ContentChildren
- Summary
- Listing

Learning Objectives

- Understand the difference between view children and content children of a component.

- Know how to get references to child components in host components.

Example application

The view children of a given component are the elements used *within* its template, its view.

We can get a reference to these view children in our component class by using the `@ViewChild` decorator.

We'll explain how all this works using the joke application we've been working with so far in this course.

We've changed the application so that the `JokeListComponent` shows two jokes in it's own view and one joke which is content projected from it's host `AppComponent`, like so:

Listing 1. JokeListComponent

TypeScript

```
@Component({
  selector: 'joke-list',
  template: `
<h4>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j"> (1)
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>

<h4>Content Jokes</h4>
<ng-content></ng-content>  (2)
`
})
class JokeListComponent  {
  jokes: Joke[] = [
    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)")
  ];
}
```

1. The component renders jokes in it's *own* view.
2. It also projects some content from it's host component, in our example the other content is going to be a third joke.

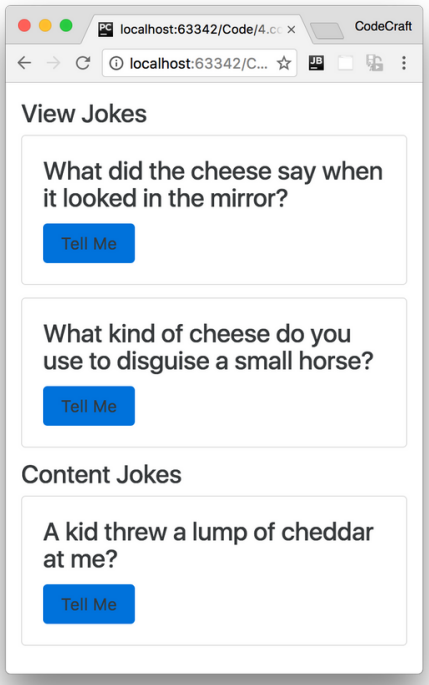
Listing 2. AppComponent

TypeScript

```
@Component({
  selector: 'app',
  template: `
<joke-list>
  <joke [joke]="joke"> (1)
    <span class="setup">{{ joke.setup }}?</span>
    <h1 class="punchline">{{ joke.punchline }}</h1>
  </joke>
</joke-list>
`
})
class AppComponent {
  joke: Joke = new Joke("A kid threw a lump of cheddar at me", "I thought 'That's not very mature'");
}
```

1. Use content projection to inject a third joke into the `JokeListComponent`.

When we view this app now we see 3 jokes, two of which are from the JokeListComponent and the third is projected in from the AppComponent.



ViewChild

In our `JokeListComponent` lets add a reference to the child `JokeComponents` that exists in it's view.

We do this by using the `@ViewChild` decorator like so:

TypeScript

```
import { ViewChild } from '@angular/core';
.
.
.
@Component({
  selector: 'joke-list',
  template: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent {

  jokes: Joke[] = [
    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)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent; (1)

  constructor() {
    console.log(`new - jokeViewChild is ${this.jokeViewChild}`);
  }
}
```

1 We are storing a reference to the child `JokeComponent` in a property called `jokeViewChild`.

> Note
>
> `jokeViewChild` isn't an instance of a `Joke` class, it is the actual instance of the child `JokeComponent` that exists inside *this* components view.

We create a new property called `jokeViewChild` and we pre-pend this with a decorator of `@ViewChild`. This decorator tells Angular *how* to find the child component that we want to bind to this property.

A `@ViewChild` decorator means, search inside this components template, it's view, for this child component.

The parameter we pass as the first argument to `@ViewChild` is the *type* of the component we want to search for, if it finds more than one it will just give us the first one it finds.

If we try to print out the reference in the constructor, like the code sample above, `undefined` will be printed out.

That's because by the time the constructor is called we haven't rendered the children yet. We render in a *tree down approach* so when a parent component is getting constructed it means the children are not yet created.

We can however hook into the lifecycle of the component at the point the view children have been created and that's with the `ngAfterViewInit` hook.

To use this we need to make our component implement the interface `AfterViewInit`.

TypeScript

```
@Component({
  selector: 'joke-list',
  template: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent implements AfterViewInit {

  jokes: Joke[] = [
    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)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;

  constructor() {
    console.log(`new - jokeViewChild is ${this.jokeViewChild}`);
  }

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);
  }
}
```

In the `ngAfterViewInit` function `jokeViewChild` has been initialised and we can see it logged in the console.

ViewChildren

The above isn't so useful in our case since we have *multiple* joke children components. We can solve that by using the alternative `@ViewChildren` decorator along side the `QueryList` generic type.

TypeScript

```
import { ViewChildren, QueryList } from '@angular/core';
.
.
.
class JokeListComponent implements AfterViewInit {

  jokes: Joke[] = [
    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)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>; (1)

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);
    let jokes: JokeComponent[] = this.jokeViewChildren.toArray(); (2)
    console.log(jokes);
  }
}
```

1. We use the `@ViewChildren` decorator which matches *all* `JokeComponent`s and stores them in a `QueryList` called jokeViewChildren.

2 We can convert our `QueryList` of `JokeComponent`s into an array by calling `toArray()`

When we run the above application we see two `JokeComponents` printed to the console, like so:

```
Array[2]
> 0: JokeComponent
> 1: JokeComponent
```

> **Important**
>
> The reason we see 2 jokes printed out and 3 is because only two of the jokes are *view children* the other joke is a *content child*. We cover content children in the end of ths lecture.

ViewChild referencing a template local variable

One practical application of `@ViewChild` is to get access to template local variables in our component class.

In the past we've said that template local variables are just that, *local* to the template.

But as the first param to the `@ViewChild` decorator we can also pass the name of a template local variable and have Angular store a reference to that variable on our component, like so:

TypeScript

```
@Component({
  selector: 'joke-list',
  template: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent implements AfterViewInit {

  jokes: Joke[] = [
    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)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>;
  @ViewChild("header") headerEl: ElementRef; (1)

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);

    let jokes: JokeComponent[] = this.jokeViewChildren.toArray();
    console.log(jokes);

    console.log(`ngAfterViewInit - headerEl is ${this.headerEl}`);
    //noinspection TypeScriptUnresolvedVariable
    this.headerEl.nativeElement.textContent = "Best Joke Machine"; (2)
  }
}
```

1 The type of our template variable is an `ElementRef`, which is a low level reference to any element in the DOM. We are requesting a reference to the header template variable which points to the first `<h4>` element in the template.

2 Since `headerEl` is an `ElementRef` we can interact with the DOM directly and change the title of our header to *Best Joke Machine*.

> Note
>
> It's not recommended to interact with the DOM directly with an `ElementRef` since that results in code that's not very portable.

ContentChild & ContentChildren

The concept of a *content child* is similar to that of a *view child* but the content children of the given component are the child elements that are *projected* into the component from the host component.

In our example application we are projecting one joke in from the host `AppComponent`.

To get a reference to that child we can use either the `@ContentChild` or the `@ContentChildren` decorators. They work in similar ways to the view child counterparts, `@ContentChild` returns one child and `@ContentChildren` returns a `QueryList`.

Lets use `@ContentChild` to get a reference to the third joke that is projected in, like so:

TypeScript

```
import { ContentChildren, ContentChild } from '@angular/core';
.
.
.
class JokeListComponent implements AfterContentInit, AfterViewInit { (1)

  jokes: Joke[] = [
    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)")
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>;
  @ViewChild("header") headerEl: ElementRef;

  @ContentChild(JokeComponent) jokeContentChild: JokeComponent; (2)

  ngAfterContentInit() { (3)
    console.log(`ngAfterContentInit - jokeContentChild is ${this.jokeContentChild}`);
  }

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);

    let jokes: JokeComponent[] = this.jokeViewChildren.toArray();
    console.log(jokes);

    console.log(`ngAfterViewInit - headerEl is ${this.headerEl}`);
    this.headerEl.nativeElement.textContent = "Best Joke Machine";
  }
}
```

1 Just like before we need to tap into one of the component lifecycle hooks, this time it's `AfterContentInit`

2 We create a `jokeContentChild` property and bind it to the content child by using the `@ContentChild` decorator.

3 By the time the `ngAfterContentInit` hook is run the `jokeContentChild` property is set to the content child.

> **Tip**
>
> You can implement multiple interfaces just by separating them with a `,`.

If we logged `jokeContentChild` in our constructor it would again log undefined, since it's not actually initialised at that point.

Content children are only visible by the time the `AfterContentInit` lifecycle hook has run.

Summary

An Angular application is composed from a number of components nested together.

These components can nest in two ways, as view children, in the template for that component. Or they can nest as content children, via content projection from a host component.

As developers of our components we can get access to these child components via the `@ViewChild` and `@ContentChild` (and `@ViewChildren` and `@ContentChildren`) decorators.

View children of a component are the components and elements in *this* components view.

Content children of a component are the components and elements that are *projected* into *this* components view by a host component.

View children are only initialised by the time the `AfterViewInit` lifecycle phase has been run.

Content children are only initialised by the time the `AfterContentInit` lifecycle phase has been run.

Listing

Listing 3. main.ts

TypeScript

```
import { platformBrowserDynamic } from "@angular/platform-browser-dynamic";
import {
  Component,
  NgModule,
  Input,
  Output,
  EventEmitter,
  ViewEncapsulation,
  SimpleChanges,
  OnChanges,
  OnInit,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy,
  ViewChild,
  ViewChildren,
  ContentChild,
  ContentChildren,
  ElementRef,
  QueryList
} 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",
  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: `
<h4 #header>View Jokes</h4>
<joke *ngFor="let j of jokes" [joke]="j">
  <span class="setup">{{ j.setup }}?</span>
```

```
  <h1 class="punchline">{{ j.punchline }}</h1>
</joke>
<h4>Content Jokes</h4>
<ng-content></ng-content>
`
})
class JokeListComponent implements OnInit, AfterContentInit, AfterViewInit {
  jokes: Joke[] = [
    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)"
    )
  ];

  @ViewChild(JokeComponent) jokeViewChild: JokeComponent;
  @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>;
  @ViewChild("header") headerEl: ElementRef;
  @ContentChild(JokeComponent) jokeContentChild: JokeComponent;

  constructor() {
    console.log(`new - jokeViewChild is ${this.jokeViewChild}`);
    console.log(`new - jokeContentChild is ${this.jokeContentChild}`);
  }

  ngOnInit() {
  }

  ngAfterContentInit() {
    console.log(
      `ngAfterContentInit - jokeContentChild is ${this.jokeContentChild}`
    );
  }

  ngAfterViewInit() {
    console.log(`ngAfterViewInit - jokeViewChild is ${this.jokeViewChild}`);

    let jokes: JokeComponent[] = this.jokeViewChildren.toArray();
    console.log(jokes);

    console.log(`ngAfterViewInit - headerEl is ${this.headerEl}`);
    this.headerEl.nativeElement.textContent = "Best Joke Machine";
  }
}

@Component({
  selector: "app",
  template: `
<joke-list>
  <joke [joke]="joke">
    <span class="setup">{{ joke.setup }}?</span>
    <h1 class="punchline">{{ joke.punchline }}</h1>
  </joke>
</joke-list>
`
})
class AppComponent {
  joke: Joke = new Joke(
    "A kid threw a lump of cheddar at me",
    "I thought ‘That’s not very mature’"
  );
}

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, JokeComponent, JokeListComponent],
  bootstrap: [AppComponent]
```

```
})
export class AppModule {}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Caught a mistake or want to contribute to the book? Edit this page on GitHub! (https://github.com/codecraft-tv/angular-course/tree/current/4.components/6.viewchildren-and-contentchildren/index.adoc)

‹ LIFECYCLE HOOKS (/COURSES/ANGULAR/COMPONENTS/LIFECYCLE-HOOKS/) WRAPPING UP ›