ANGULAR STYLING JUMPSTART

EVERYTHING NEEDED TO STYLE ANGULAR COMPONENTS IN ONE PLACE

ANGULAR UNIVERSITY

Table Of Contents

Note: see details at the end of this book for getting the Free Angular For Beginners Course, plus a bonus content Typescript Video List.

- How does Angular encapsulate styles?
- The :host pseudo-class selector
- Combining the :host selector with other selectors
- The ::ng-deep pseudo-class selector
- The :host-context pseudo-class selector

Angular CLI Sass Support

- Angular CLI - Sass, Less and Stylus support
- Demo of some the things we can do with Sass

Final Thoughts & Bonus Content

- Conclusions and Recommendations
- Bonus Content - Typescript - A Video List
- Bonus Content - Angular For Beginners Course

Angular Styling Jumpstart

Introduction

Welcome to the Angular Styling Jumpstart Book, thank you for joining!

This book is meant to be a one-stop shop for everything that relates to Angular component styling: it contains everything that you are likely to need to style your components, in a single comprehensive reference.

So without further ado, let's get started!

I hope you will enjoy this book, please send me your feedback at admin@angular-university.io.

Section 1 - `ngClass` and `ngStyle`

Component Styling using `ngClass` - when to use it?

Most of the styles that we need to apply are always present, and can be simply be applied as standard HTML in our templates, like this:

```
1
2  <p>A Bootstrap Primary Button:</p>
3  <button class="btn btn-primary">Button</button>
4
```

```
5
```

But there are often styles that are applied conditionally to our templates - they are added to an element only if a certain programmatic condition is met.

This is, for example, the case of **state styles** (if we adopt the SMACSS terminology).

For these cases, is `ngClass` needed?

Note that many state styles can be natively implemented using browser CSS pseudo-classes, such as for example:

- styles for identifying an element with the focus, via the `:focus` pseudo class
- hover styles and on-click active state styles (using `:hover` and `:active`)

For these type of state styles natively supported by the browser, it's better to use the CSS pseudo classes whenever possible. So for these very common cases we won't need `ngClass`.

Some good examples for the use of `ngClass`

But there are many other state styles that are not natively supported by the browser. These styles could for example include:

- styles for identifying the currently selected elements of a list
- styles for identifying the currently active menu entry in a navigation menu

- styles to identify a certain feature of a element; for example to identify a new element in an e-commerce site

If the element that we are styling only has one of those state styles, we can even apply it simply by using the plain input property template syntax, without any extra directive:

```
<p>Default Button:</p>
<button class="btn btn-primary" type="submit">Button</button>

<p>Equivalent example using Button:</p>
<button class="btn"
        [class.btn-primary]="true"
        type="submit">Button
</button>
```

Notice the syntax `[class.btn-primary]` that is activating the `btn-primary` CSS class, effectively adding it to the button.

This expression will add or not the class to the element depending on the truthiness of the expression, which in this case is always true.

> *But more often than not, an element ends up having multiple state styles, and that is when the `ngClass` directive comes in handy!*

The `ngClass` directive will take an expression that will be used to determine which state styles to apply at a given time to the styled element.

The expression passed on to `ngClass` can be:

- an object
- an array
- a string

Let's go over each one of these 3 cases with examples, and then see how we can make sure that we can still keep our templates light and readable.

Passing an Array of CSS classes to `ngClass`

One way of defining what classes should be active at a given moment is to pass an array of strings to the `ngClass` directive.

For example, the following expression contains an array of classes:

```
<p>Passing an Array of classes:</p>
<button [ngClass]="['btn', 'btn-primary']">Button</button>
```

Angular will then take the array passed to `ngClass`, and apply the CSS classes that it contains to the HTML button element. This is the resulting HTML:

```
<button class="btn btn-primary">Button</button>
```

Notice that the CSS classes don't have to be hard-coded in the template using this syntax (its just an example), more on this later.

Passing a String of CSS classes to `ngClass`

Its also possible to pass to `ngClass` a string, that contains all the CSS classes that we want to apply to a given element:

```
<p>Passing a string:</p>
<button [ngClass]="'btn btn-primary'"
        type="submit"
        (click)="submit()">
  Button
</button>
```

This syntax would give the same results as before, meaning that the two CSS classes `btn` and `btn-primary` would still be applied.

Passing a configuration object to `ngClass`

The last and most commonly used way that we can configure `ngClass` is by passing it an object:

- the keys of that object are the names of the CSS classes that we want to apply (or not)
- and the values of the configuration object should be booleans (or an expression that evaluates to a boolean) that indicate if the CSS class should be applied

Let's have a look at an example of how to use this syntax:

```
<p>Passing a configuration object:</p>
<button [ngClass]="{ btn:true, 'btn-primary':true }">
  Button
</button>
```

This example would give the same results as before: the two CSS classes would still get applied.

But if for example start using longer expressions to calculate our boolean values, or have several state classes, this syntax could quickly become hard to read, overloading the template and putting too much logic in it.

Let's then see what we can if we run into that case!

Delegating to the component which styles should be applied

One of the roles of the component class is to:

- coordinate the link between the View definition (the template), and the data passed to the component (the Model)
- as well as to keep track of any other type of visual component state that is tied uniquely to the component and is transient in nature (like a flag saying if a collapsible panel is open or not)

If our `ngClass` expressions start to get too cumbersome and hard to read, what we can do is pass to `ngClass` the output of a component method:

```
1
2  @Component({
3      selector: 'app-root',
4      template: `
5          <button (click)="toggleState()">Toggle State</button>
6
7          <p>Obtaining the CSS classes from the
8                  component method:</p>
```

```
        <button [ngClass]="calculateClasses()"
                        (click)="submit()">Button</button>
 `})
export class AppComponent {

    stateFlag = false;

    toggleState() {
        this.stateFlag = !this.stateFlag;
    }

    submit() {
        console.log('Button submitted');
    }

    calculateClasses() {
        return {
            btn: true,
            'btn-primary': true,
            'btn-extra-class': this.stateFlag
        };
    }
}

```

Notice that we could pass parameters to this method if needed. Let's then break down what is going on in this example:

- The component now has a member variable `stateFlag`, which will identify if a given component state is active or not.
- This could also have been an `enum`, or a calculation derived from the input data
- The method `calculateClasses` will now return a configuration object equivalent to the one we just saw above
- the CSS class `btn-extra-class` will be added or not to the HTML button depending on the value of the `stateFlag`

variable

But this time around the calculation of the configuration object is done in a component method, and the template becomes a bit more readable.

This function could have also returned an array or string containing multiple CSS classes, and the example would still work!

As we can see, between the native browser functionality and `ngClass`, we will be able to do most of the styling for our components.

But are there use cases where we would like to apply styles directly to an element?

How to use `ngStyle` to add embedded styles to our templates

Just like in the case of plain CSS, sometimes there are valid use cases for applying directly styles to an HTML element, although in general this is to be avoided.

This is because this type of embedded styles takes precedence over any CSS styles except styles that are marked with `!important`.

To give an example of when we would like to use this: Imagine a color picker element, that sets the color of a sample rectangle based on a handle that gets dragged by the user.

The varying color of the element needs an embedded HTML style, as its not known upfront. If we run into such an use case using Angular, we can implement it using the `ngStyle` built-in core directive:

```
<p>Passing an object to ngStyle:</p>
<button [ngStyle]="{background: 'red'}">Button</button>
```

And this would be the resulting HTML:

```
<button style="background: red">Button</button>
```

Just like the case of `ngClass`, if our `ngStyle` expression starts to get too large, we can always call a component method to calculate the configuration object:

```
<p>Obtaining styles from a component method:</p>
<button [ngStyle]="calculateStyles()">Button</button>
```

And with this, we can now add both CSS classes and embedded styles conditionally to our components!

But another key feature of Angular that we have not covered yet, is the ability to isolate a component style so that it does not interfere with other elements on the page.

Section 2 - Style Isolation

Why Style Isolation?

Why would we want to isolate the styles of our components? There are a couple of reasons for that, and one key reason is CSS maintainability.

As we develop a component style suite for an application, we tend to run into situations where the styles from one feature start interfering with the styles of another feature.

This is because browsers do not have yet widespread support for style isolation, where we can constrain one style to be applied only to one particular part of the page.

If we are not careful and systematically organize our styles to prevent this issue (for example using a methodology like SMACSS), we will quickly run into CSS maintenance issues.

Wouldn't it be great to be able to style our components with just short, simple and easy to read selectors, without having to worry about all the scenarios where those styles could be accidentally overridden?

Another benefit of style isolation

Here is another scenario: how many times did we try to use a third-party component, add it to our application just to find out that the component is completely broken due to styling issues?

Style isolation would allow us to ship our components knowing that the styles of the component will (most likely) not be overridden by other styles in target applications.

This makes the component effectively much more reusable, because the component will now in most cases simply just work, styles included.

Angular View Encapsulation brings us all of these advantages, so let's learn how it works!

A Demo of Angular Emulated Encapsulation

In this section, we will see how Angular component styling works under the hood, as this is the best way to understand it. This will also allow us to debug the mechanism if needed.

In order to benefit from the default view encapsulation mechanism of Angular, all we need to do is to add our CSS classes to an external CSS file:

```
1
2  .red-button {
3      background:red;
4  }
5
```

But then, instead of adding this file to our `index.html` as a `link` tag, we will instead associate it with our component using the `styleUrls` property:

```
1
2  @Component({
3      selector: 'app-root',
4      styleUrls:['./app.component.css'],
5      template: `
6          <button class="red-button">Button</button>
7      `})
8  export class AppComponent {
9
10     ...
11 }
```

The color red would then be applied to this button, as expected. But what if now we have another button, for example directly at the level of our `index.html`?

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>StylesApp</title>
  <base href="/">
   ...
 </head>
<body>

  <app-root></app-root>

  <button class="red-button">index.html Button</button>

</body>
</html>

```

If you didn't know that there was some sort of style isolation mechanism in action, you might be surprised to find out that this button does **NOT** get a red background!

So what is going on here? Let's see how this mechanism works, because knowing that is what is going to allow us to debug it if needed.

How does Angular Style Isolation work? Emulated View Encapsulation

To better understand how default view encapsulation works, let's see what the `app-root` custom HTML element will look like at runtime:

```
```

```
<app-root _nghost-c0="">
    <h2 _ngcontent-c0="">Component Style Isolation example</h2>
    <button _ngcontent-c0="" class="red-button">Button</button>
</app-root>
```

Several things are going on at the level of the runtime HTML:

- a strange looking property was added to the `ap-root` custom element: the `_nghost-c0` property
- Each of the HTML elements inside the application root component got another strange looking but different property: `_ngcontent-c0`

What are these properties?

So how do these properties work? To better understand these properties and how they enable style isolation, we are going to create a second separate component, that just contains a button with the blue color.

For simplicity, we will define the styles for this component inline next to the template:

```
@Component({
  selector: 'blue-button',
  template: `
      <h2>Blue button component</h2>

      <button class="blue-button">Button</button>
  `,
  styles: [`
```

```
        .blue-button {
            background:blue;
        }
    `]
})
export class BlueButtonComponent  {
}

```

And using this newly defined component, we are going to add it to the template of the application root component:

```

@Component({
    selector: 'app-root',
    styleUrls:['./app.component.css'],
    template: `
        <button class="red-button">Button</button>

        <blue-button></blue-button>
    `})
export class AppComponent {

   ...
}
```

> *Try to guess at this stage what the HTML at runtime would look like, and what happened to those strangely named properties!*

The host element and template element style isolation properties

With this second component in place, let's have a second look at the HTML. The way that these two properties work will now become much more apparent:

```

```

```
<app-root _nghost-c0="">

    <h2 _ngcontent-c0="">Component Style Isolation example</h2>

    <button _ngcontent-c0="" class="red-button">Button</button>

    <blue-button _nghost-c1="" _ngcontent-c0="">

        <h2 _ngcontent-c1="">Blue button component</h2>

        <button _ngcontent-c1="" class="blue-button">Button</button>

  </blue-button>

</app-root>

```

Notice the `blue-button` element, we have now a new host property called `_nghost-c1`.

The `blue-button` element is still tagged with the `_ngcontent-c0` property which is applied to all template elements on the application root component.

But now, the elements inside the `blue-button` template now get applied the `_ngcontent-c1` property instead!

Summary of how the host and template element properties work

Let's then summarize how these special HTML properties work, and then see how they enable style isolation:

- upon application startup (or at build-time with AOT), each component will have a unique property attached to the host element,

depending on the order in which the components are processed: `_nghost-c0`, `_nghost-c1`, etc.

- together with that, each element inside each component template will also get applied a property that is unique to that particular component: `_ngcontent-c0`, `_ngcontent-c1`, etc.

This is all transparent and done under the hood for us.

How do these properties enable view encapsulation?

The presence of these properties could allow us to write manually CSS styles which are much more targetted than just the simple styles that we have on our template.

For example, if we want to scope the blue color to the `blue-button` component only, we could write manually the following style:

```
1
2  /* Style 1 - a simple CSS style,
3     with low specificy and easilly overridable */
4  .blue-button {
5      background: blue;
6  }
7
8  /* Style 2 - a similar style, with a much higher
9    specificity and much harder to override */
10 .blue-button[_ngcontent-c1] {
11     background: blue;
12 }
13
```

While style 1 was applicable to any element with the `blue-button` class anywhere on the page, style 2 will only work for elements that have that strangely named property!

So this means that style 2 is effectively scoped to only elements of the `blue-button` component template, and will not affect any other elements of the page.

So we now can see how those two special properties do enable some sort of style isolation, but it would be cumbersome to have to use those properties manually in our CSS (and in fact, we should **not**).

> *But luckily, we don't have to. Angular will do that automatically for us.*

How does Angular encapsulate styles?

At startup time (or at build time if using AOT), Angular will see what styles are associated with which components, via the `styles` or `styleUrls` component properties.

Angular will then take those styles and apply them transparently the corresponding isolating properties, and will then inject the styles directly into the page header as a `style` tag:

```
<style>
  .blue-button[_ngcontent-c1] {
          background:blue;
   }
</style>

```

The `_ngcontent-c1` property is unique to elements of the `blue-button` template, so the style will be scoped to those elements only.

And that is how the Angular default view encapsulation mechanism works!

This mechanism is not 100% bullet-proof as it does not guarantee perfect isolation, but in practice, it will nearly always work.

The mechanism it's not based on the shadow DOM but instead in these special HTML properties, so if we really wanted to we could still override these styles.

But given that the native Shadow Dom isolation mechanism is currently available only in Chrome and Opera, we cannot yet rely on it.

This mechanism is very useful because it enables us to write simple styles that will not break easily, but we might want to break this isolation selectively from time to time.

Let's learn a couple of ways of doing that, and why we would want to do that.

The `:host` pseudo-class selector

Sometimes we want to style the component custom HTML element itself, and not something inside its template.

Let's say for example that we want to style the `app-root` component itself, by adding it, for example, an extra border.

We cannot do that using styles inside its `app.component.css` associated file, right?

This is because all styles inside that file will be scoped to elements of the template, and not the outer `app-root` element itself.

If we want to style the host element of the component itself, we need the special `:host` pseudo-class selector. This is the new version of our `app.component.css` that uses it:

```
1
2  /* other styles on app.component.css */
3  ...
4
5  /* styles applied directly to the ap-root element only */
6  :host {
7      border: 2px solid dimgray;
8      display: block;
9      padding: 20px;
10 }
11
```

This selector will ensure those styles are only targeting the `app-root` element. Remember that `_nghost-c0` property that we talked about before? This is how it's used to implement the `:host` selector at runtime:

```
1
2  <style>
3    [_nghost-c0] {
4      border: 2px solid dimgray;
5      display: block;
6      padding: 20px;
7    }
8  </style>
```

```
9
```

The use of the special `_nghost-c0` will ensure that those styles are scope only to the `app-root` element, because `app-root` gets added that property at runtime:

```
1
2  <app-root _nghost-c0="">
3    ...
4  </app-root>
5
```

Combining the `:host` selector with other selectors

Notice that the can combine this selector with other selectors, which is something that we have not yet talked about.

This is not specific to this selector, but have a look for example at this selector, where we are styling `h2` elements inside the host element:

```
1
2  /* let's add another style to app.conmponent.css */
3  :host h2 {
4      color: red;
5  }
6
```

You could be surprised to find out that this style only applies to the `h2` elements inside the `app-root` template, but **not** to the h2 inside the `blue-button` component.

To see why, let's have a look at the styles that were generated at runtime:

```
<style>
   ....

  [_nghost-c0]   h2[_ngcontent-c0] {
    color: red;
  }

</style>

```

So we can see that the special scoping property gets applied also to nested selectors, to ensure the style is always scoped to that particular template.

But if we did want to override the styles of all the `h2` elements, there is still a way.

The `::ng-deep` pseudo-class selector

If we want our component styles to cascade to all child elements of a component, but not to any other element on the page, we can currently do so using by combining the `:host` with the `::ng-deep` selector:

```

:host /deep/ h2 {
    color: red;
}

```

This will generate at runtime a style that looks like this:

```

<style>
[_nghost-c0]  h2 {
    color: red;
```

```
}
</style>

```

So this style will be applied to all `h2` elements inside `app-root`, but not outside of it as expected.

This combination of selectors is useful for example for applying styles to elements that were passed to the template using `ng-content`.

The `:host-context` pseudo-class selector

Sometimes, we also want to have a component apply a style to some element outside of it. This does not happen often, but one possible common use case is for theme enabling classes.

For example, let's say that we would like to ship a component with multiple alternative themes. Each theme can be enabled via adding a CSS class to a parent element of the component.

Here is how we could implement this use case using the `:host-context` selector:

```

@Component({
  selector: 'themeable-button',
  template: `
        <button class="btn btn-theme">Themeable Button</button>
  `,
  styles: [`
      :host-context(.red-theme) .btn-theme {
        background: red;
      }
      :host-context(.blue-theme) .btn-theme {
          background: blue;
```

```
        }
    `]
})
export class ThemeableButtonComponent {

}

```

These themed styles are deactivated by default. In order to activate one theme, we need to add to any parent element of this component one of the theme-activating classes.

For example, this is how we would activate the blue theme:

```

<div class="blue-theme">
    <themeable-button></themeable-button>
</div>

```

All of this functionality that we saw so far was using plain CSS.

But especially in the case of themes, it would be great to be able to extend the CSS language and for example define the primary color of a theme in a variable, to avoid repetition like we would do in Javascript.

That is one of the many use cases that we can support using a CSS preprocessor.

Section 3 - Angular CLI Sass Support

Angular CLI - Sass, Less and Stylus support

A CSS pre-processor is a program that takes an extended version of CSS, and compiles it down to plain CSS.

The Angular CLI supports all major pre-processors, but the one that seems most commonly used in Angular related projects (such as for example Angular Material) is Sass.

In order to use a Sass file instead of a CSS file, we just need to pass such file to the `styleUrls` property of a component:

```
1
2  @Component({
3      selector: 'app-root',
4      styleUrls:['./app.component.scss'],
5      template: `...`
6  })
7  export class AppComponent {
8     ...
9  }
10
```

The CLI will then take this Sass file and convert it to plain CSS on the fly. Actually, we can generate new components using Sass files using this command:

```
ng new cli-test-project --style=sass
```

We can also set a global property, so that Sass files are used by default:

```
ng set defaults.styleExt scss
```

Demo of some the things we can do with Sass

A pre-processor is a great thing to add to our project, to help us write more maintainable styles. Let's have a look at some of the things that we can do with Sass:

```
1
2  $border-color: red;
3
4  :host-context(.au-fa-input-red-theme) {
5
6    border-color: $border-color;
7
8
9    &.input-focus {
10     -webkit-box-shadow: 0px 0px 5px  $border-color;
11     box-shadow: 0px 0px 5px  $border-color;
12   }
13
14 }
```

If you have never seen this syntax before, it could look a bit surprising! But here is what is going on, line by line:

- on line 2, we have actually defined a CSS variable! This is a huge feature missing from CSS
- we can define not only colors but numbers or event shorthand combined properties such as: `$my-border: 1px solid red`
- on lines 6, 10 and 11 we are using the variable that we just created
- on line 9 we are using a nested style, and making a reference to the parent style using the `&` syntax

And this is just a small sample of what we can do with Sass, just a few very commonly used features. The Angular CLI also has support for global styles, that we can combine with our component-specific view encapsulated styles.

We can add global styles not only for the supported pre-processors, but for plain CSS as well.

Conclusions and Recommendations

There are a ton of options to style our components, so its important to know which one to use when and why. Here is a short summary:

- sometimes we want global styles, that are applied to all elements of a page - we can add those to our `angular-cli.json` config
- most styles can be added simply to the HTML directly, for those we can simply add them to the class property, and no special Angular functionality is needed
- Many state styles can be used using browser-supported pseudo-class selectors such as `:focus`, we should prefer those conventional solutions for those cases
- for state styles that don't have a pseudo-class selector linked to it, its best to go with `ngClass`
- if the `ngClass` expressions get too big, it's a good idea to move the calculation of the styles to the component class
- only for situations where we have a dynamically calculated embedded style should we use `ngStyle`, this should be rarely needed

- the Angular View encapsulation mechanism allows us to write simpler styles, that are simpler to read and won't interfere with other styles
- The default view encapsulation mechanism will bring the long-term benefit of having much less CSS-related bugs. Using it, we will rarely fall into the situation when we add some CSS that fixes one screen but accidentally break something else - this is a huge plus!
- If we are writing a lot of CSS in our project, we probably want to adopt a methodology for structuring our styles from the beginning, such as for example SMACSS
- At a given point we could consider introducing a pre-processor and use some of its features, for example for defining CSS variables

I hope this helps with all the many options that we have available for styling our components! If you have any questions about the book, any issues or comments I would love to hear from you at admin@angunar-university.io

I invite you to have a look at the bonus material below, I hope that you enjoyed this book and I will talk to you soon.

Kind Regards,
Vasco
Angular University

Typescript - A Video List

In this section, we are going to present a series of videos that cover some very commonly used Typescript features.

Click Here to View The Typescript Video List

These are the videos available on this list:

- Video 1 - Top 4 Advantages of Typescript - Why Typescript?
- Video 2 - ES6 / Typescript let vs const vs var When To Use Each? Const and Immutability
- Video 3 - Learn ES6 Object Destructuring (in Typescript), Shorthand Object Creation and How They Are Related
- Video 4 - Debugging Typescript in the Browser and a Node Server - Step By Step Instructions
- Video 5 - Build Type Safe Programs Without Classes Using Typescript
- Video 6 - The Typescript Any Type - How Does It Really Work?

- Video 7 - Typescript @types - Installing Type Definitions For 3rd Party Libraries
- Video 8 - Typescript Non-Nullable Types - Avoiding null and undefined Bugs
- Video 9 - Typescript Union and Intersection Types- Interface vs Type Aliases
- Video 10 - Typescript Tuple Types and Arrays Strong Typing

Angular For Beginners Course

If you are looking to learn Angular, have a look at this free 2h introductory course.

The Angular Tutorial for Beginners is an over 2 hours course that is aimed at getting a complete beginner in Angular comfortable with the main notions of the core parts of the framework:

Click Here To View The Angular For The Beginners Course

The other Angular courses on the same website have about 25% of its content free as well, have a look and enjoy the videos.