Component styles

Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle *component styles* with components, enabling a more modular design than regular stylesheets.

This page describes how to load and apply these component styles.

Run the in Stackblitz and download the code from there.

Using component styles

For every Angular component you write, you can define not only an HTML template, but also the CSS styles that go with that template, specifying any selectors, rules, and media queries that you need.

One way to do this is to set the `styles` property in the component metadata. The `styles` property takes an array of strings that contain CSS code. Usually you give it one string, as in the following example:

Component styling best practices

See View Encapsulation for information on how Angular scopes styles to specific components.

You should consider the styles of a component to be private implementation details for that component. When consuming a common component, you should not override the component's styles any more than you should access the private members of a TypeScript class. While Angular's default style encapsulation prevents component styles from affecting other components, global styles affect all components on the page. This includes `::ng-deep`, which promotes a component style to a global style.

Authoring a component to support customization

As component author, you can explicitly design a component to accept customization in one of four different ways.

1. Use CSS Custom Properties (recommended)

You can define a supported customization API for your component by defining its styles with CSS Custom Properties, alternatively known as CSS Variables. Anyone using your component can consume this API by defining values for these properties, customizing the final appearance of the component on the rendered page.

While this requires defining a custom property for each customization point, it creates a clear API contract that works in all style encapsulation modes.

2. Declare global CSS with @mixin

While Angular's emulated style encapsulation prevents styles from escaping a component, it does not prevent global CSS from affecting the entire page. While component consumers should avoid directly overwriting the CSS internals of a component, you can offer a supported customization API via a CSS preprocessor like Sass.

For example, a component may offer one or more supported mixins to customize various aspects of the component's appearance. While this approach uses global styles in it's implementation, it allows the component author to keep the mixins up to date with changes to the component's private DOM structure and CSS classes.

3. Customize with CSS ::part

If your component uses Shadow DOM, you can apply the `part` attribute to specify elements in your component's template. This allows consumers of the component to author arbitrary styles targeting those specific elements with the `::part` pseudo-element.

While this lets you limit the elements within your template that consumers can customize, it does not limit which CSS properties are customizable.

4. Provide a TypeScript API

You can define a TypeScript API for customizing styles, using template bindings to update CSS classes and styles. This is not recommended because the additional JavaScript cost of this style API incurs far more performance cost than CSS.

Special selectors

Component styles have a few special *selectors* from the world of shadow DOM style scoping (described in the CSS Scoping Module Level 1 page on the W3C site). The following sections describe these selectors.

:host

Every component is associated within an element that matches the component's selector. This element, into which the template is rendered, is called the *host element*. The `:host` pseudo-class selector may be used to create styles that target the host element itself, as opposed to targeting elements inside the host.

Creating the following style will target the component's host element. Any rule applied to this selector will affect the host element and all its descendants (in this case, italicizing all contained text).

The `:host` selector only targets the host element of a component. Any styles within the `:host` block of a child component will *not* affect parent components.

Use the *function form* to apply host styles conditionally by including another selector inside parentheses after `:host`.

In this example the host's content also becomes bold when the `active` CSS class is applied to the host element.

The `:host` selector can also be combined with other selectors. Add selectors behind the `:host` to select child elements, for example using `:host h2` to target all `<h2>` elements inside a component's view.

You should not add selectors (other than `:host-context`) in front of the `:host` selector to style a component based on the outer context of the component's view. Such selectors are not scoped to a component's view and will select the outer context, but it's not built-in behavior. Use `:host-context` selector for that purpose instead.

:host-context

Sometimes it's useful to apply styles to elements within a component's template based on some condition in an element that is an ancestor of the host element. For example, a CSS theme class could be applied to the document `<body>` element, and you want to change how your component looks based on that.

Use the `:host-context()` pseudo-class selector, which works just like the function form of `:host()`. The `:host-context()` selector looks for a CSS class in any ancestor of the component host element, up to the document root. The `:host-context()` selector is only useful when combined with another selector.

The following example italicizes all text inside a component, but only if some *ancestor* element of the host element has the CSS class `active`.

Note that only the host element and its descendants will be affected, not the ancestor with the assigned `active` class.

(deprecated) `/deep/`, `>>>`, and `::ng-deep`

Component styles normally apply only to the HTML in the component's own template.

Applying the `::ng-deep` pseudo-class to any CSS rule completely disables view-encapsulation for that rule. Any style with `::ng-deep` applied becomes a global style. In order to scope the specified style to the current component and all its descendants, be sure to include the `:host` selector before `::ng-deep`. If the `::ng-deep` combinator is used without the `:host` pseudo-class selector, the style can bleed into other components.

The following example targets all `<h3>` elements, from the host element down through this component to all of its child elements in the DOM.

The `/deep/` combinator also has the aliases `>>>`, and `::ng-deep`.

Use `/deep/`, `>>>` and `::ng-deep` only with *emulated* view encapsulation. Emulated is the default and most commonly used view encapsulation. For more information, see the View Encapsulation section.

The shadow-piercing descendant combinator is deprecated and support is being removed from major browsers and tools. As such we plan to drop support in Angular (for all 3 of `/deep/`, `>>>` and `::ng-deep`). Until then `::ng-deep` should be preferred for a broader compatibility with the tools.

{@a loading-styles}

Loading component styles

There are several ways to add styles to a component:

- By setting `styles` or `styleUrls` metadata.
- Inline in the template HTML.
- With CSS imports.

The scoping rules outlined earlier apply to each of these loading patterns.

Styles in component metadata

Add a `styles` array property to the `@Component` decorator.

Each string in the array defines some CSS for this component.

Reminder: these styles apply *only to this component*. They are *not inherited* by any components nested within the template nor by any content projected into the component.

The Angular CLI command `ng generate component` defines an empty `styles` array when you create the component with the `--inline-style` flag.

ng generate component hero-app --inline-style

Style files in component metadata

Load styles from external CSS files by adding a `styleUrls` property to a component's `@Component` decorator:

Reminder: the styles in the style file apply *only to this component*. They are *not inherited* by any components nested within the template nor by any content projected into the component.

You can specify more than one styles file or even a combination of `styles` and `styleUrls`.

When you use the Angular CLI command `ng generate component` without the `--inline-style` flag, it creates an empty styles file for you and references that file in the component's generated `styleUrls`.

ng generate component hero-app

Template inline styles

Embed CSS styles directly into the HTML template by putting them inside `<style>` tags.

Template link tags

You can also write `<link>` tags into the component's HTML template.

When building with the CLI, be sure to include the linked style file among the assets to be copied to the server as described in the Assets configuration guide.

Once included, the CLI includes the stylesheet, whether the link tag's href URL is relative to the application root or the component file.

CSS @imports

Import CSS files into the CSS files using the standard CSS `@import` rule. For details, see `@import` on the MDN site.

In this case, the URL is relative to the CSS file into which you're importing.

External and global style files

When building with the CLI, you must configure the `angular.json` to include *all external assets*, including external style files.

Register **global** style files in the `styles` section which, by default, is pre-configured with the global `styles.css` file.

See the Styles configuration guide to learn more.

Non-CSS style files

If you're building with the CLI, you can write style files in sass, or less, and specify those files in the `@Component.styleUrls` metadata with the appropriate extensions (`.scss` , `.less`) as in the following example:

@Component({ selector: 'app-root', templateUrl: './app.component.html', styleUrls: ['./app.component.scss'] }) ...
The CLI build process runs the pertinent CSS preprocessor.

When generating a component file with `ng generate component`, the CLI emits an empty CSS styles file (`.css`) by default. Configure the CLI to default to your preferred CSS preprocessor as explained in the Workspace configuration guide.

Style strings added to the `@Component.styles` array *must be written in CSS* because the CLI cannot apply a preprocessor to inline styles.

@reviewed 2021-09-17