How Styles Work in Angular

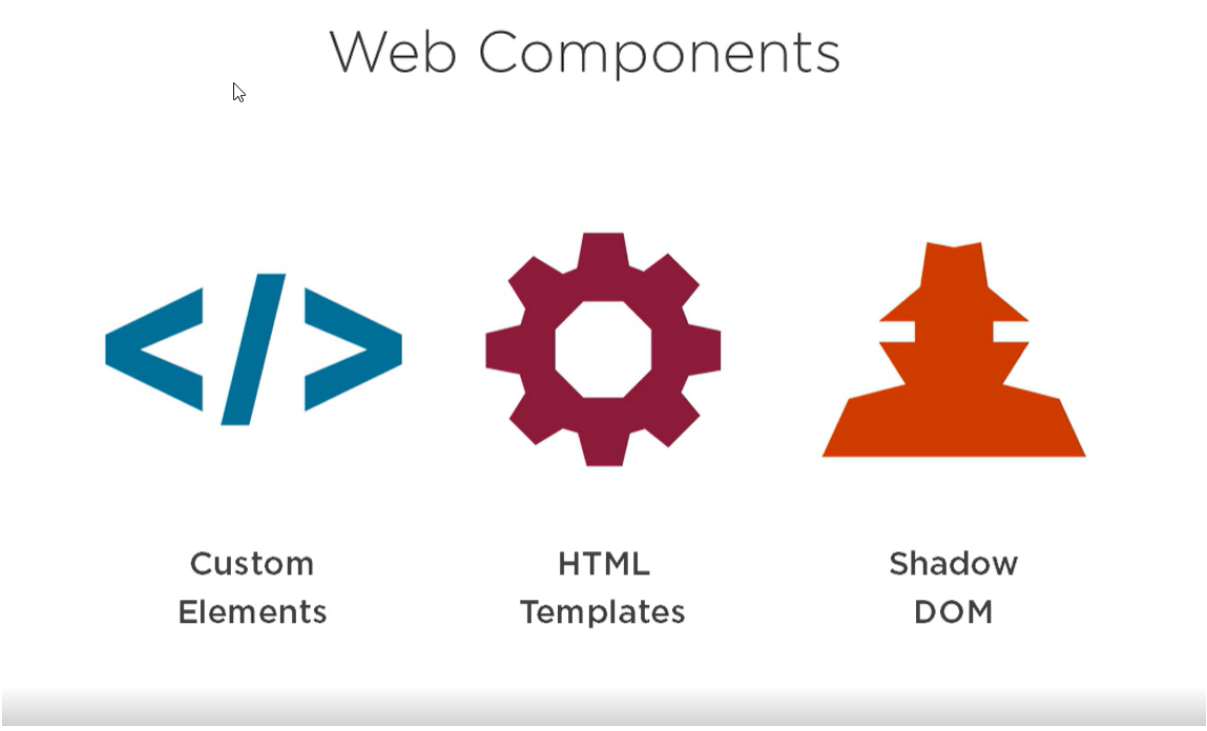
Web Components

A componentized web is where we're at. It's become the modern way to build web applications. Angular is and has been developed with web components in mind since way back in version 2, long before web components were even supported in browsers. Now Angular is a full-scale framework for building web applications, providing much more than web components alone provide out of the box.

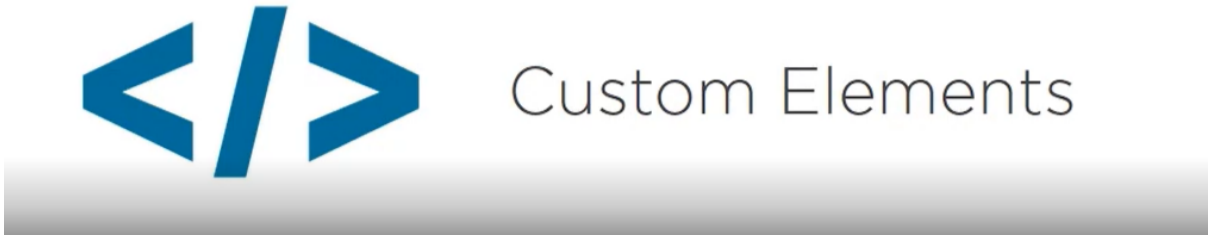
But when using Angular, we are practically building web components. So before we can begin, we should probably understand them a little since they're so closely related. Web components are bundles of modular HTML, JavaScript, and CSS code that represent portions of the UI and the browser.

Think of the HTML5 video player, for example. They provide us the ability to fully encapsulate and isolate logical sections of an application, including all necessary HTML, JavaScript, and CSS, and then they bundle it up for reuse inside of a single application or across multiple different applications.

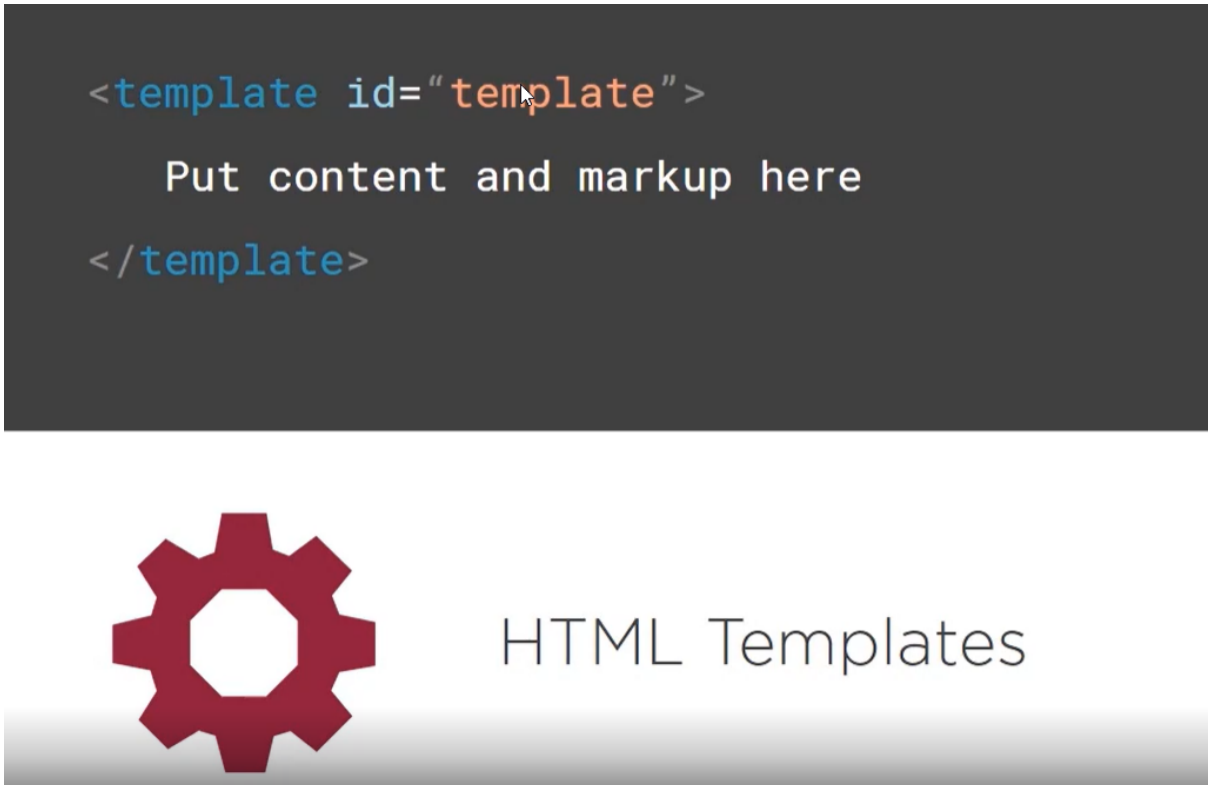
Web components consist of three parts, custom elements, HTML templates, and shadow DOM.



The idea behind custom elements is that they allow us to create and use our own elements. Custom elements work much like the HTML5 video element. This means that we can pretty much create any functionality we need, wrap it up into a web component, and then use it by calling its custom element, just as we would any other HTML helmet.

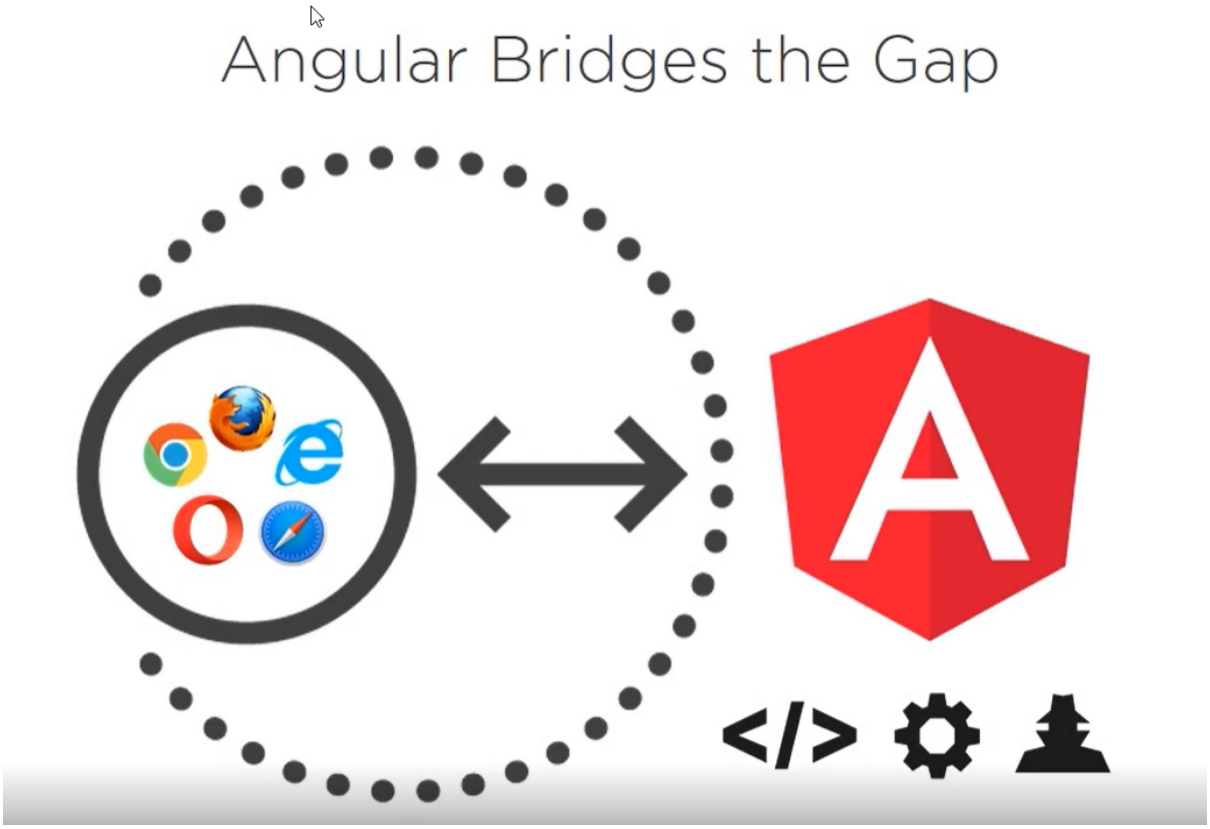


HTML templates are essentially fragments of markup that are not actually rendered to the page. Their sole purpose is to be referenced and then cloned into a specific location. This is where we define the structure of the markup that will be bundled into our component.

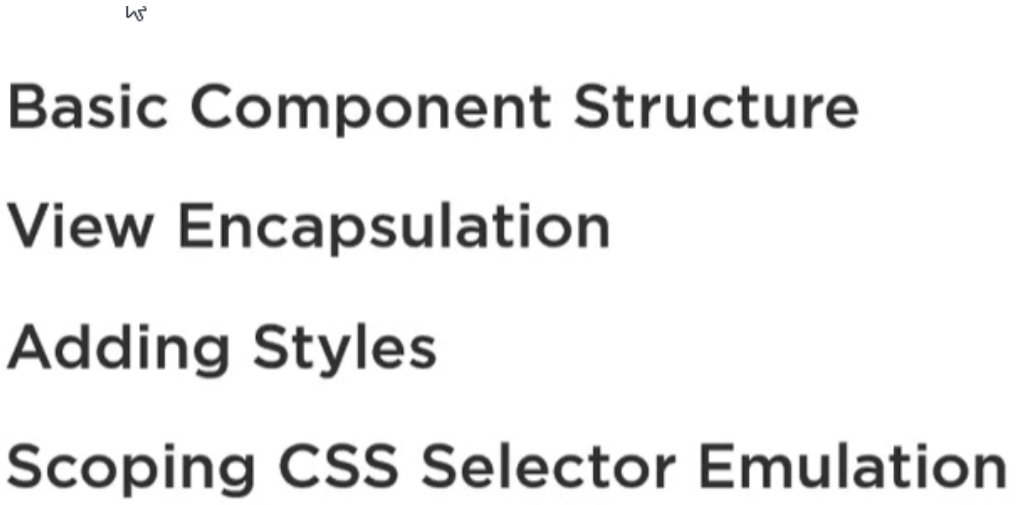


The shadow DOM is a critical aspect for web components. It's the encapsulated DOM that we have access to within our components for styling and scripting against. It's called the shadow DOM because, in theory, it's in the shadow of what's considered the normal, or light, DOM. This means that we can see this markup, but we can't actually access it from the parent styles of JavaScript and vice versa. Our styles in JavaScript from within the component can't leak out and affect the items in the light DOM.

Okay, so that's web components in a nutshell. It's important to note that Angular does fill some gaps between web component features that browsers currently support and those that remain unsupported, most importantly, the host contact pseudo class and backwards compatibility with older browsers. So we'll explore Angular's basic component structure for HTML, CSS, and JavaScript.

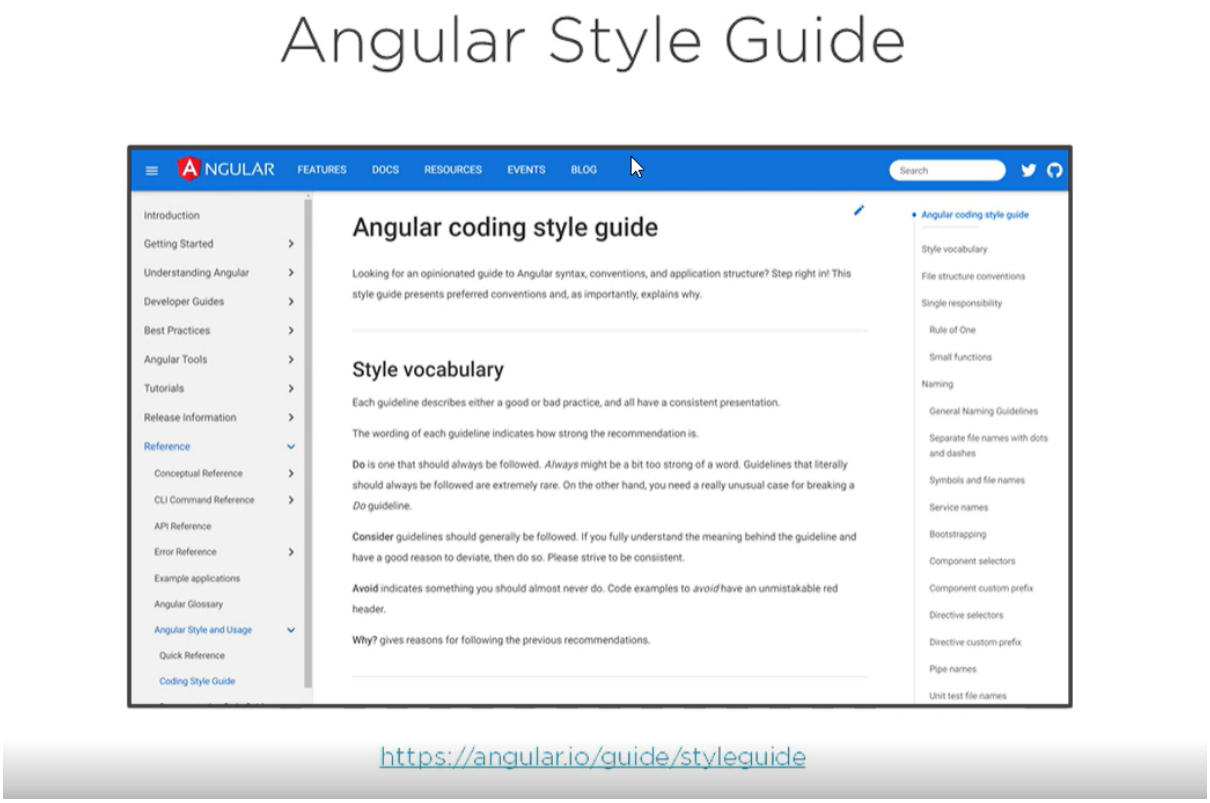


Then we'll move on to Angular's view encapsulation modes, the different ways that we can add style to components, and how Angular emulates some of the CSS scoping module selectors that allow us to effectively work with its emulated shadow DOM.



Angular Component Structure

Angular components are set up to function very much like native web components. We can use custom elements to render markup and styles from a template. Let's begin by familiarizing ourselves with what this looks like. So important to know that we'll be following many of the recommendations from the Angular Style Guide.



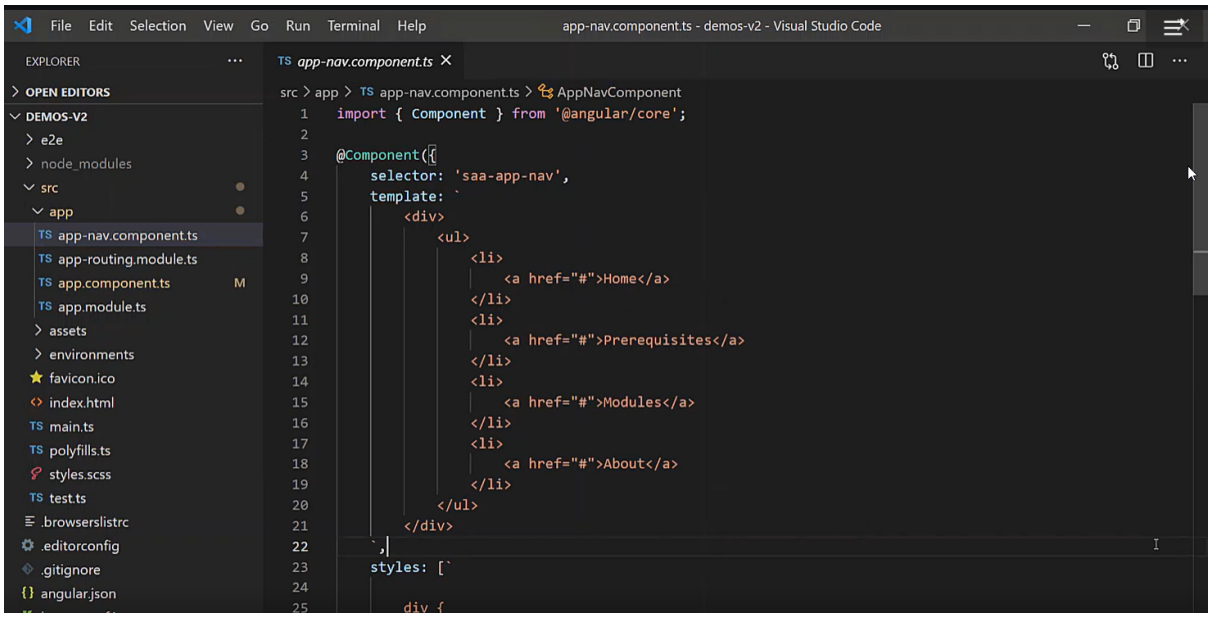
If you'd like to get more information, you should check it out for yourselves by going to the following URL. Okay, let's look at some code.

So since we're using TypeScript, this means that we'll need a TypeScript file to define our component. And since we're following the Style Guide, our file name will use dashes to separate words, followed by a dot and then the type. In this case, it's a component.

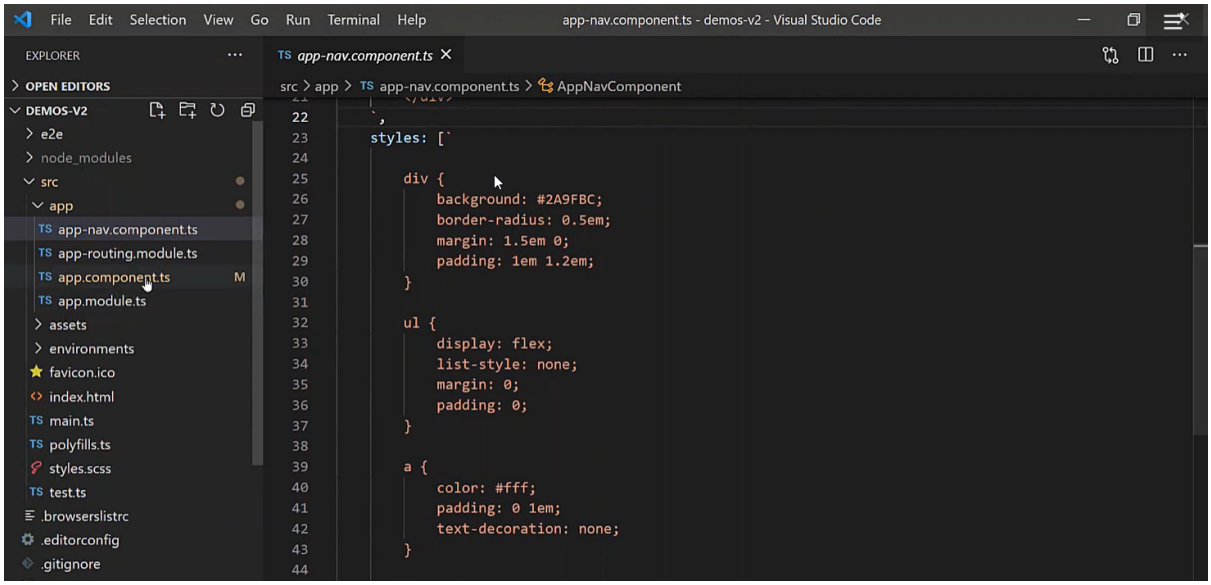
And then, of course, the extension. In this case, it's ts for TypeScript. So this is a component at its most basic level. If we look at the code, we can see that we have a very simple import statement here at the top which imports the Component decorator function from Angular Core.

Then we have the @Component decorator, which contains a set of metadata associated with our component class. This metadata contains the selector for the component, in this case saa-app-nav. The name for this component selector is a custom element of app-nav with the Angular Style Guide recommended app prefix of saa, short for styling Angular applications. And this is the first bit from the web component world, a custom element.

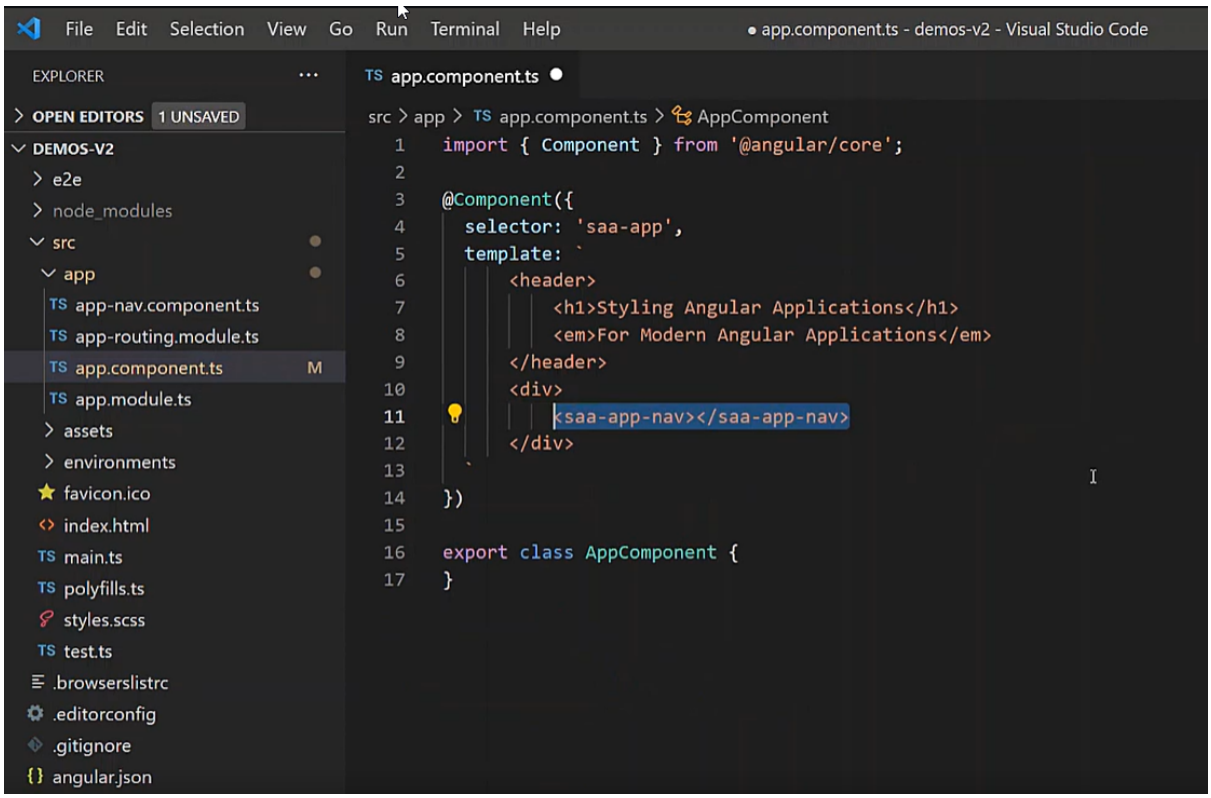
This selector is nothing more than a CSS selector. We could use any element or attribute for that matter, but much of the time we'll find a custom element will be in order. So after the selector, we then have a template. That's right, this is similar to the template concept from web components. All the markup we define here will be inserted every time we use our app-nav custom element. What we also have here is the markup for which our CSS and JavaScript will be scoped to, so essentially, we have our shadow DOM as well.



And finally, we have the styles, which will render the style scope to this component. So now our component is all set up.

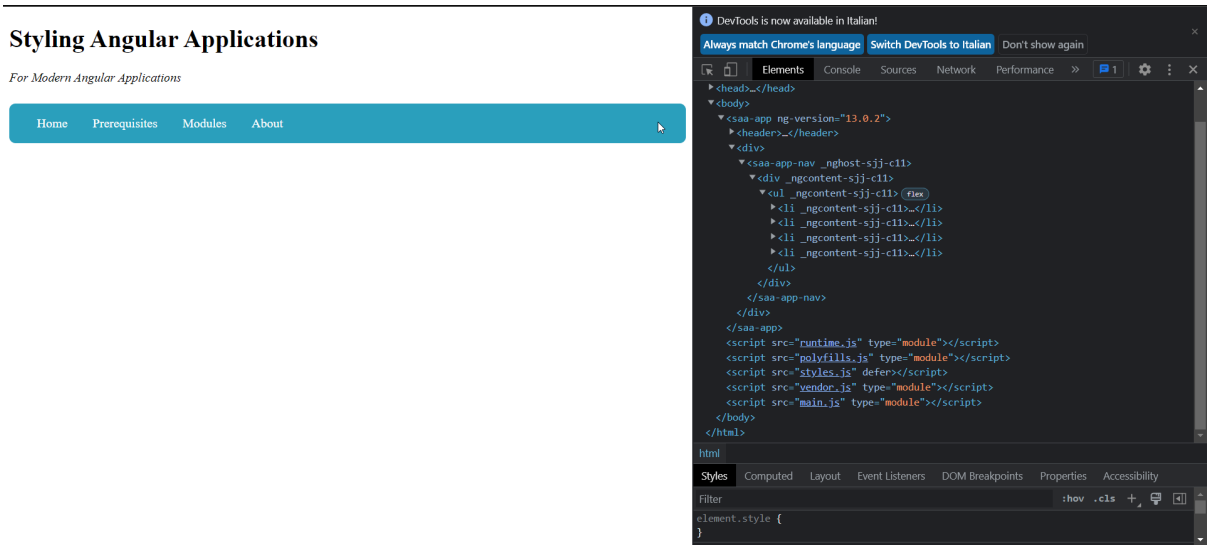


If we add it to our app.component template using its custom element, we can see that it's now rendered to the page. So now we're starting to get somewhere. We've got a basic understanding of the setup required for a basic Angular component.

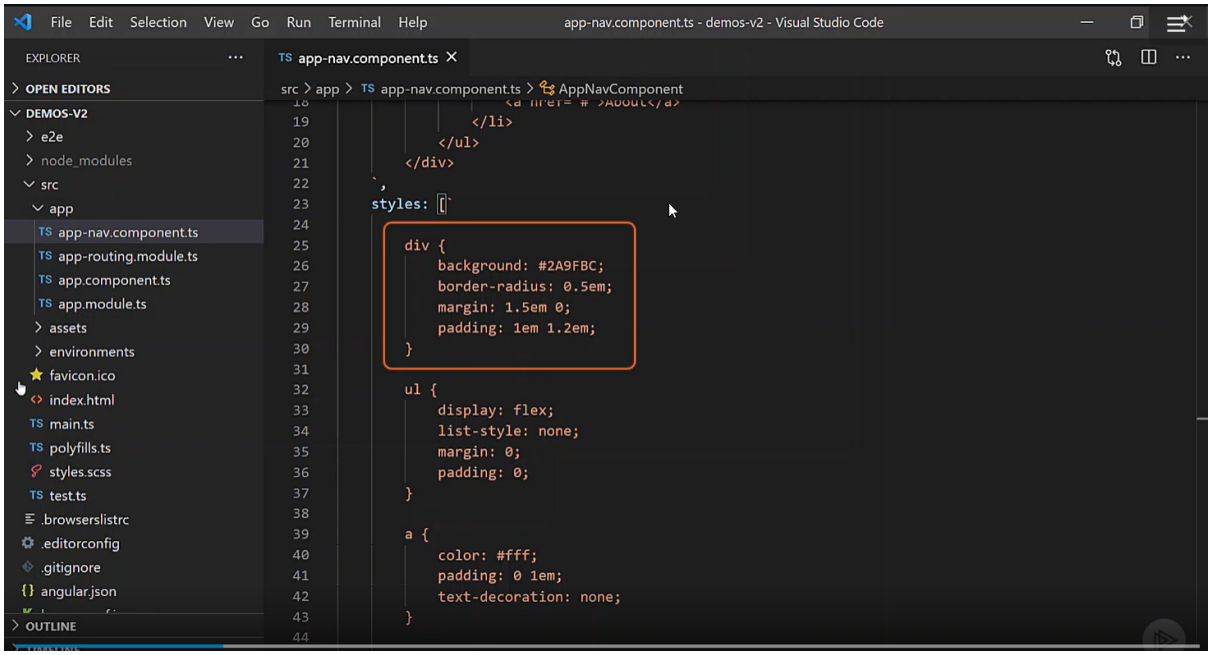


View Encapsulation

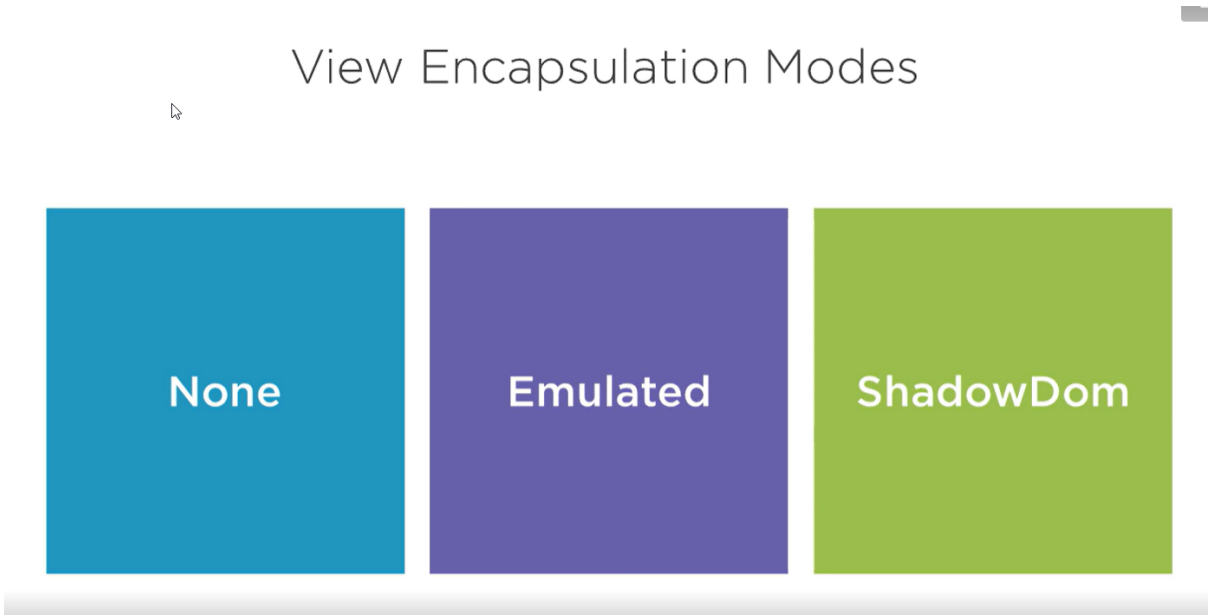
All right, so we just created a very simple Angular component and added it to our application, so now it's time to tear into the code and see what's really going on. If we start by inspecting the markup for our component, we'll see some pretty crazy stuff going on here. We see here that we have all of the markup from our template, and it's wrapped inside of our custom element. We can see that it's just markup rendered right within the same HTML page for application. So Angular components work by inserting the contents of their template directly within the element that matches the selector that we provided within the component.ts file.



But some of the styles we added within our component are tied generically to a div selector. But our app component also contains a div, and these styles aren't applying to it. Why?



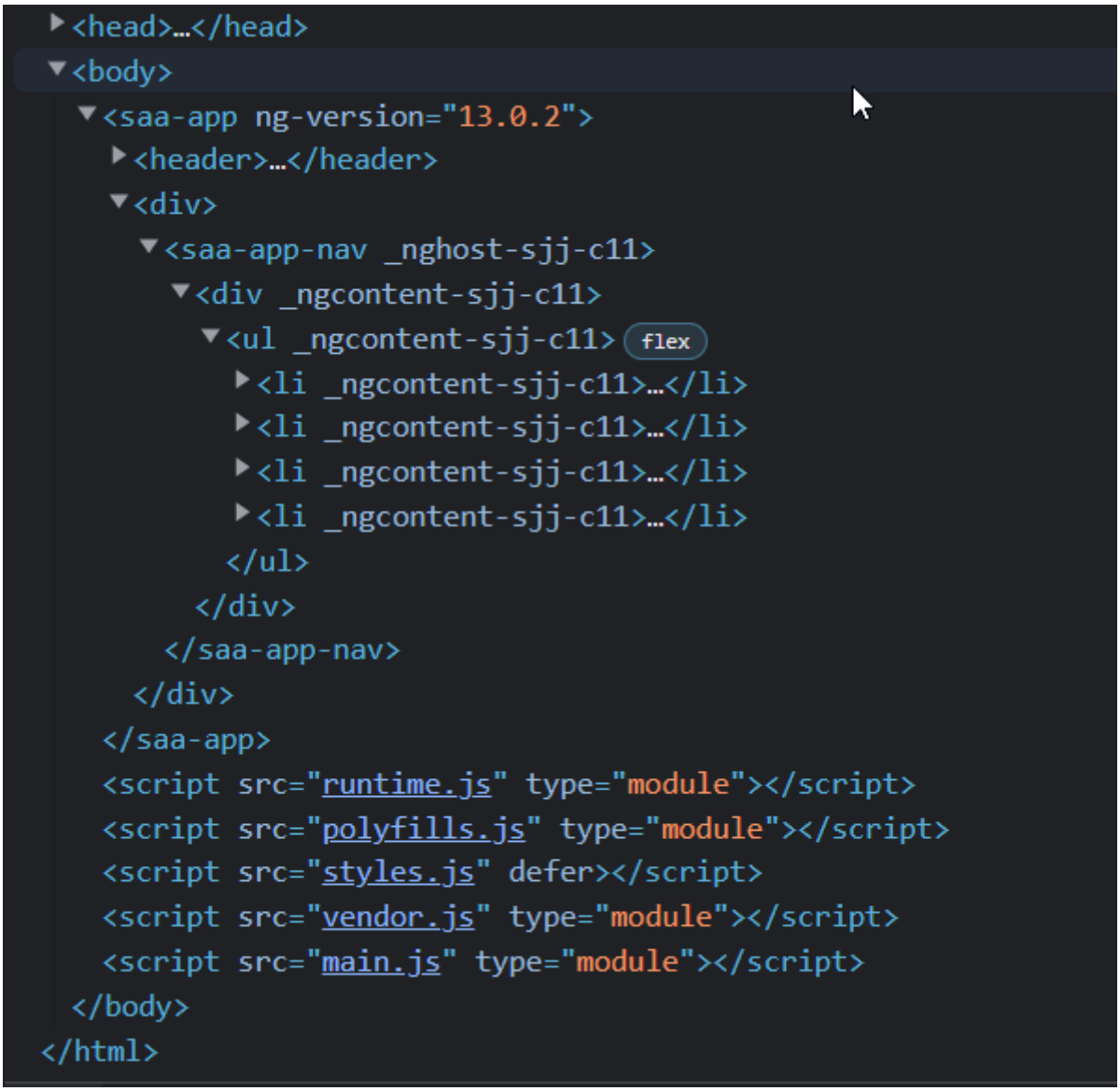
Well, it has to do with Angular's view encapsulation modes. View encapsulation in Angular is how we control whether we emulate, use shadow DOM, or don't use any style scoping behavior. There are three different modes, None, Emulated, and ShadowDom.



We'll take a look at each one of these to see how they're treated, but let's start with the default, Emulated mode.

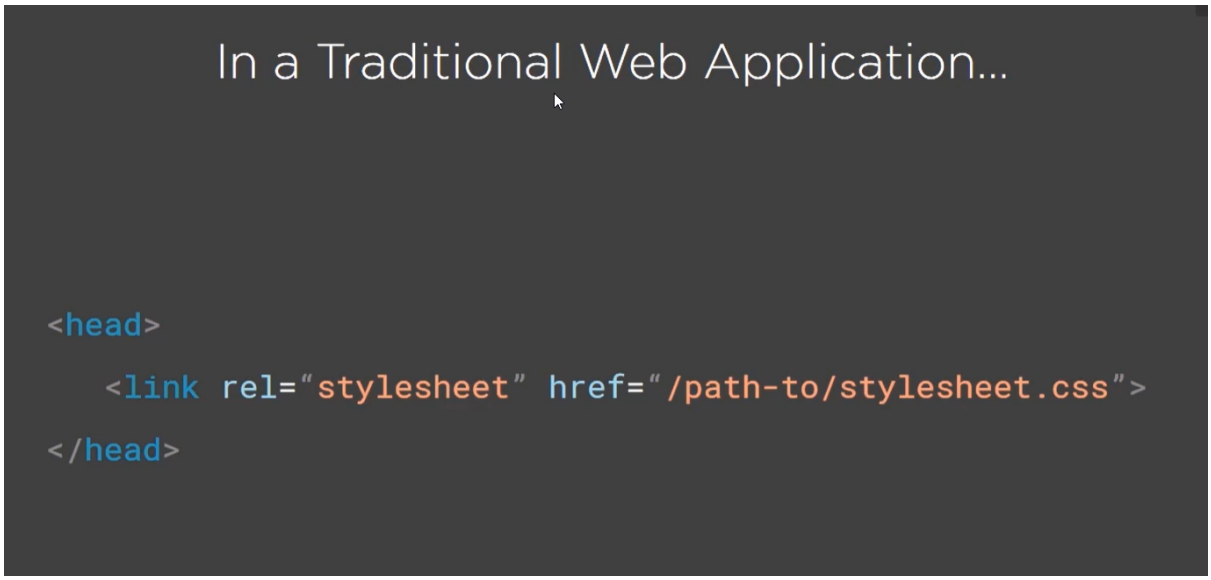
Out of the box, Angular will add some scoping attributes that we can see here on the nodes within this component. The custom element gets this funny looking _nghost attribute, and its content from the template also gets another funny looking _ngcontent attribute. These attributes are dynamically added to all nodes within our components. In this case, the custom element gets an attribute prefixed with _nghost because it's considered the host of our component.

The nodes within this host element get attributes prefixed with ngcontent because they're considered content nodes within the component.

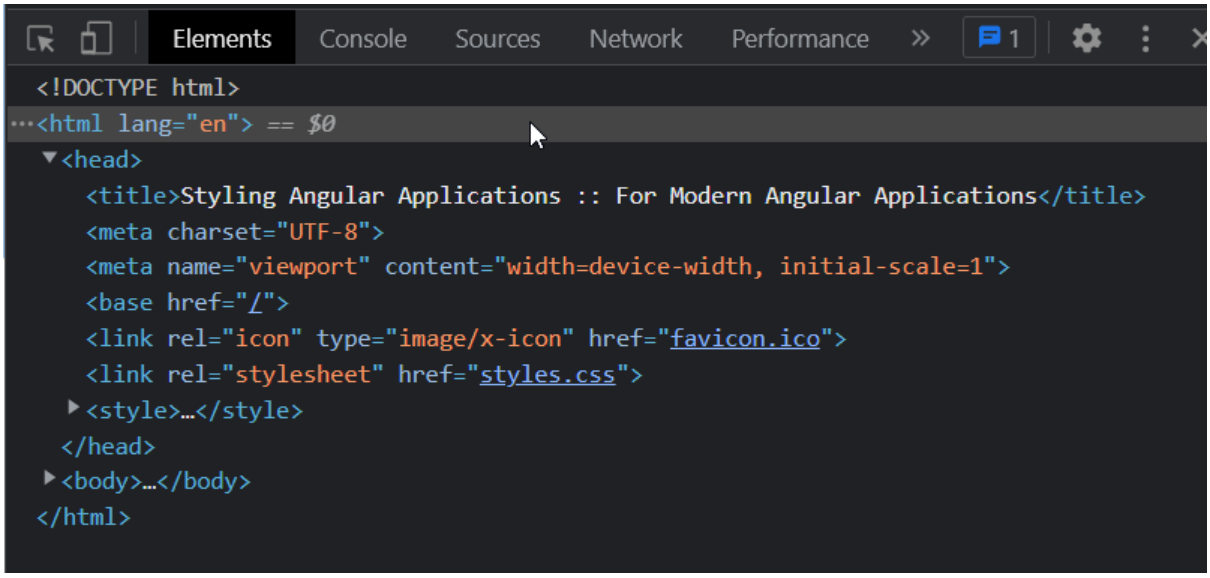


Okay, great, but this still doesn't explain why our styles aren't applying to all divs, right? Well, to fully understand how style scoping works in Angular, we need to take

a look at how the framework processes the CSS that we provide for our components. If we consider a traditional web application, we generally tend to link to an external style sheet using a link tag in the head of our document.



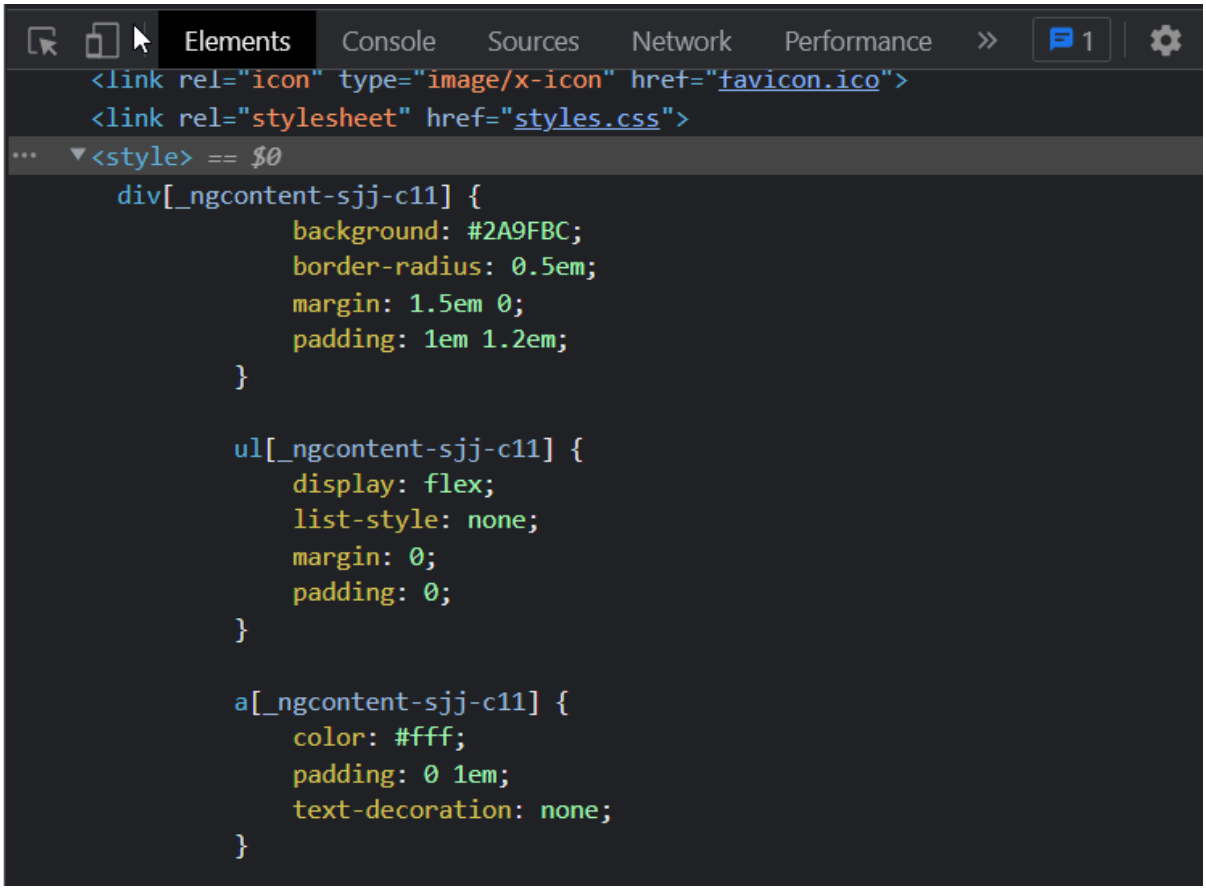
So let's take a look at the head. While we can still add external style sheets if we need to, when leveraging Angular's view encapsulation within our components, these components styles will be inserted as style blocks in the head.



See this style block here? Right below the `<link rel="stylesheet" href="styles.css">` ?

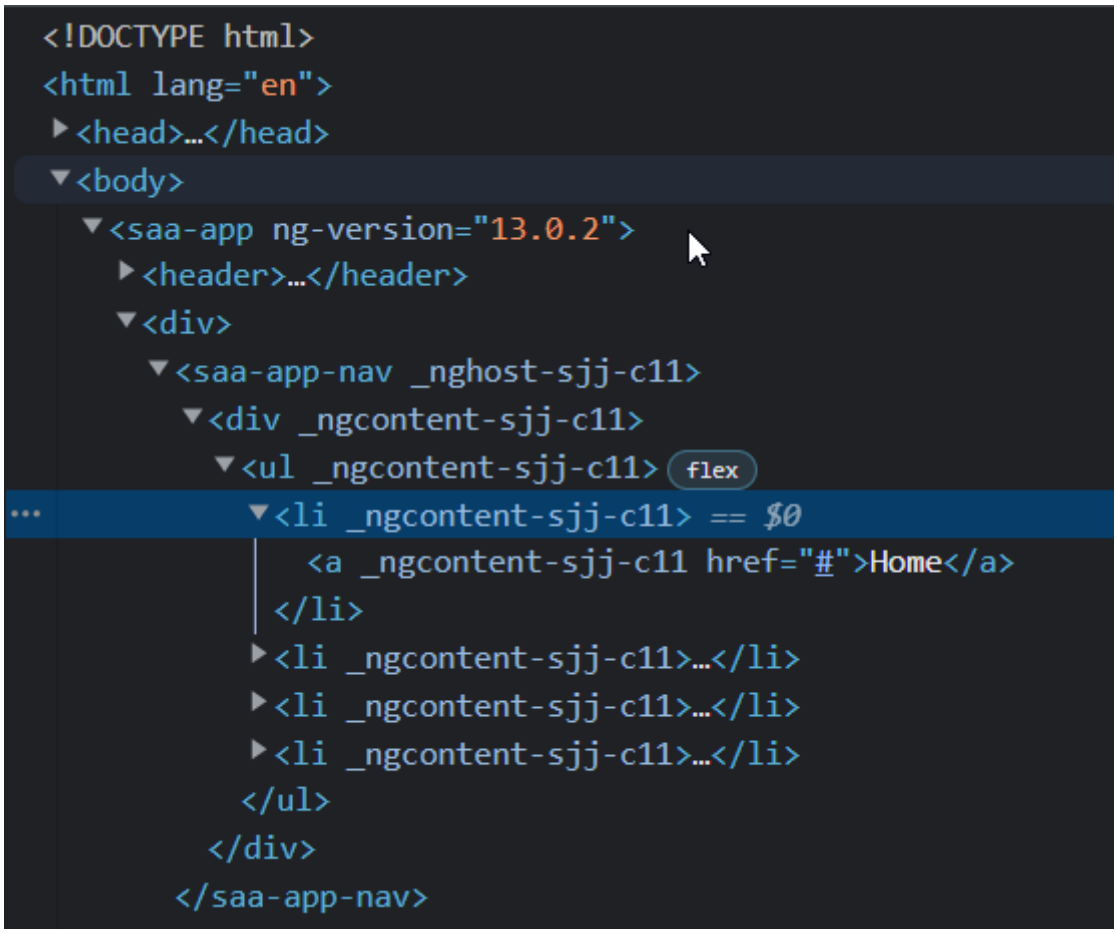
This is actually where Angular inserts the style for our component. If we expand it, we can see the styles that we added in the styles property within our component metadata.

We can also see that our div selector has been rewritten to include an attribute selector appended to it, and these attributes match those that were added to our component markup by Angular. These scoping attributes are completely unique to each component, which in turn means that these styles will only apply to the markup elements with matching attributes. And this is what provides CSS that's scoped uniquely to the set of markup. It's really a clever approach to take our styles out of the global space.

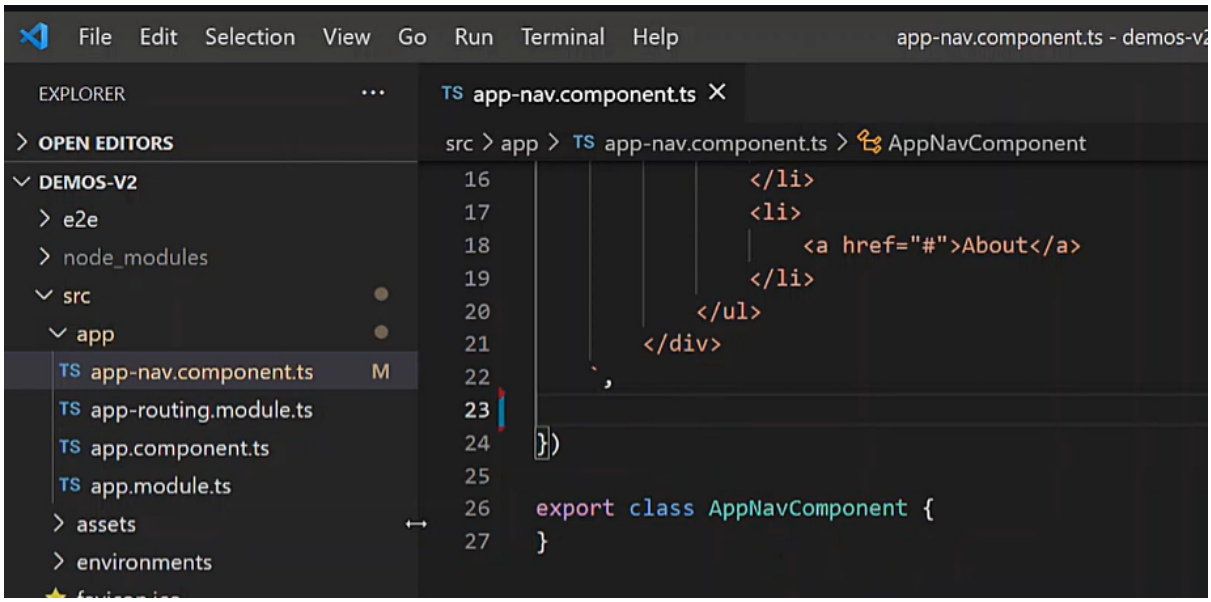


Now I can already imagine the thoughts swirling around in your heads, but before you even try, do not add styles into your components based off these attribute names. It won't work. The attributes are different every time the app is rerendered.

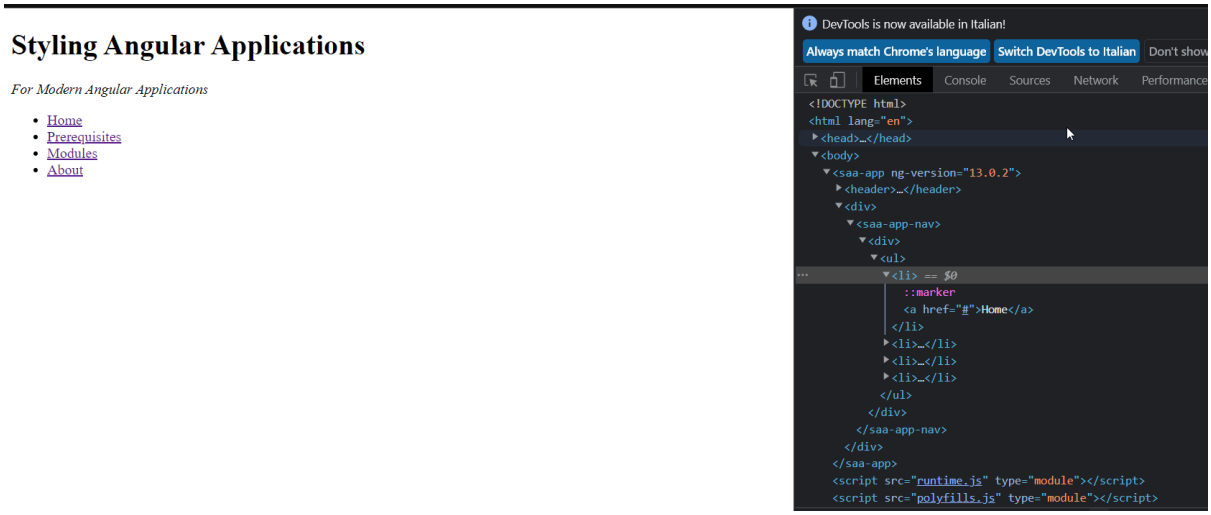
So if we refresh and let the app reload, we can see that the attributes for our component will have a slightly different name, so definitely don't do that.



Something else to point out, if we were to remove the styles property from our component like so,

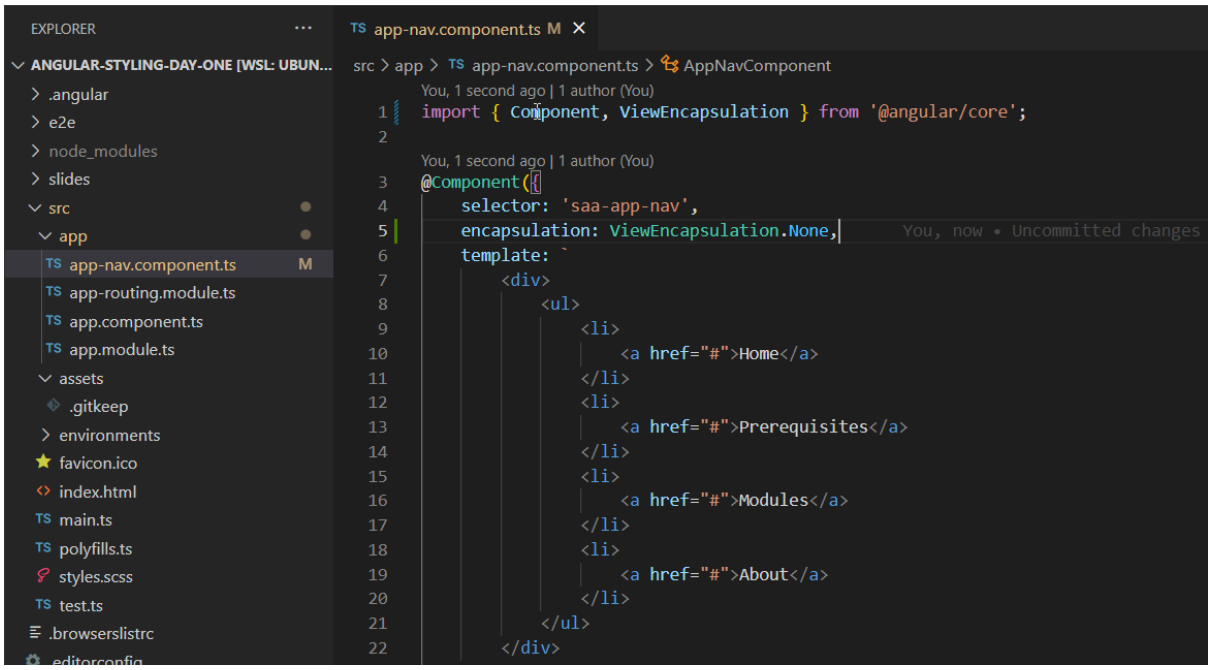


Angular will actually not add the scope name attributes to the component since they're not needed. So if we inspect this now, we'll just see plain old HTML with no attributes. Okay, so that's the default emulated view encapsulation mode.



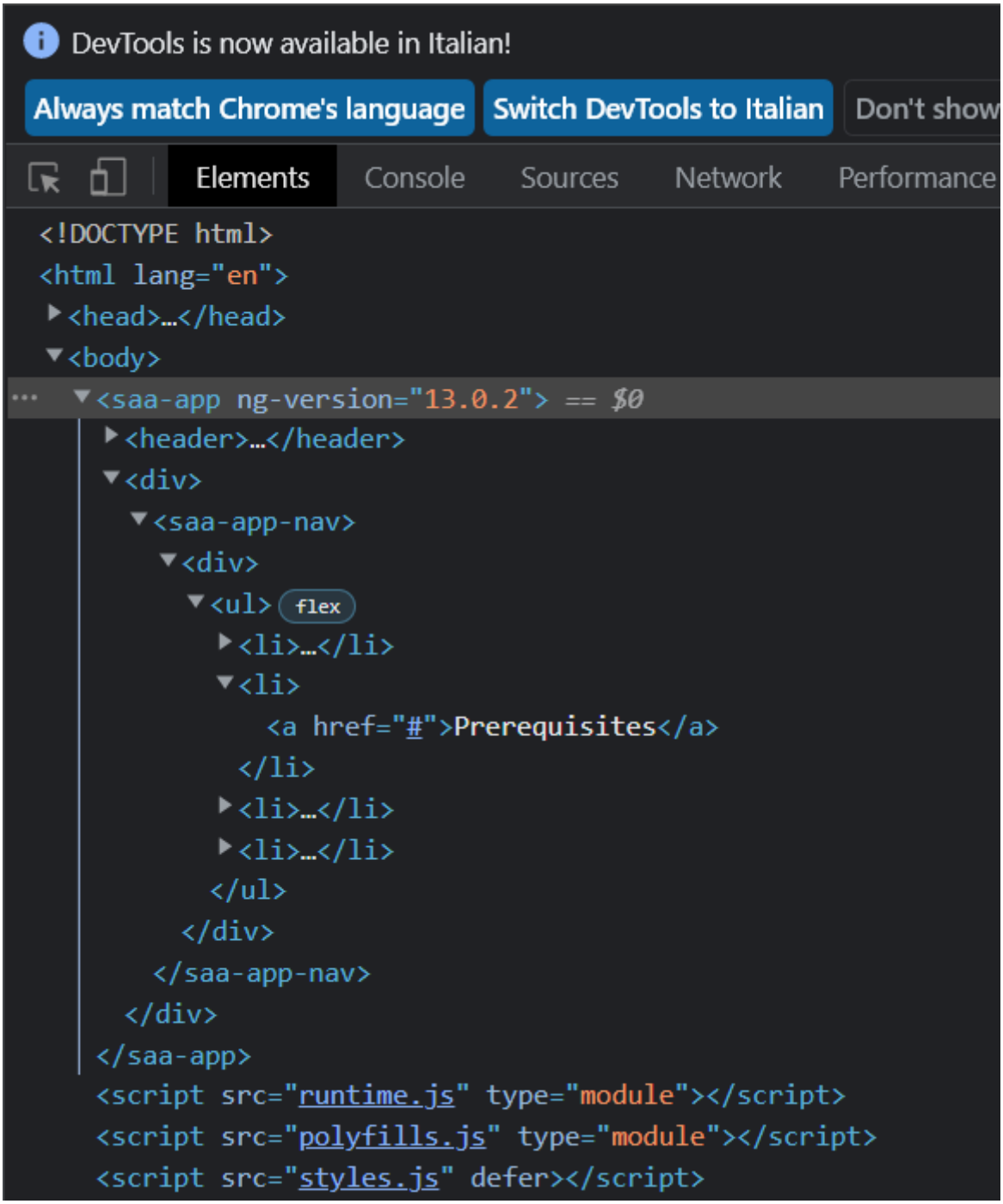
View Encapsulation Modes: None & Native

Currently, the bulk of the power of Angular's view encapsulation modes is found within the default Emulated mode. This is because it provides something very spectacular, scope and encapsulation, well, at least emulated scope and encapsulation. But modes None and ShadowDom essentially just turn it off and let the browser handle things. They're both very important to know, however, so let's take a look at each of them individually to find out why.

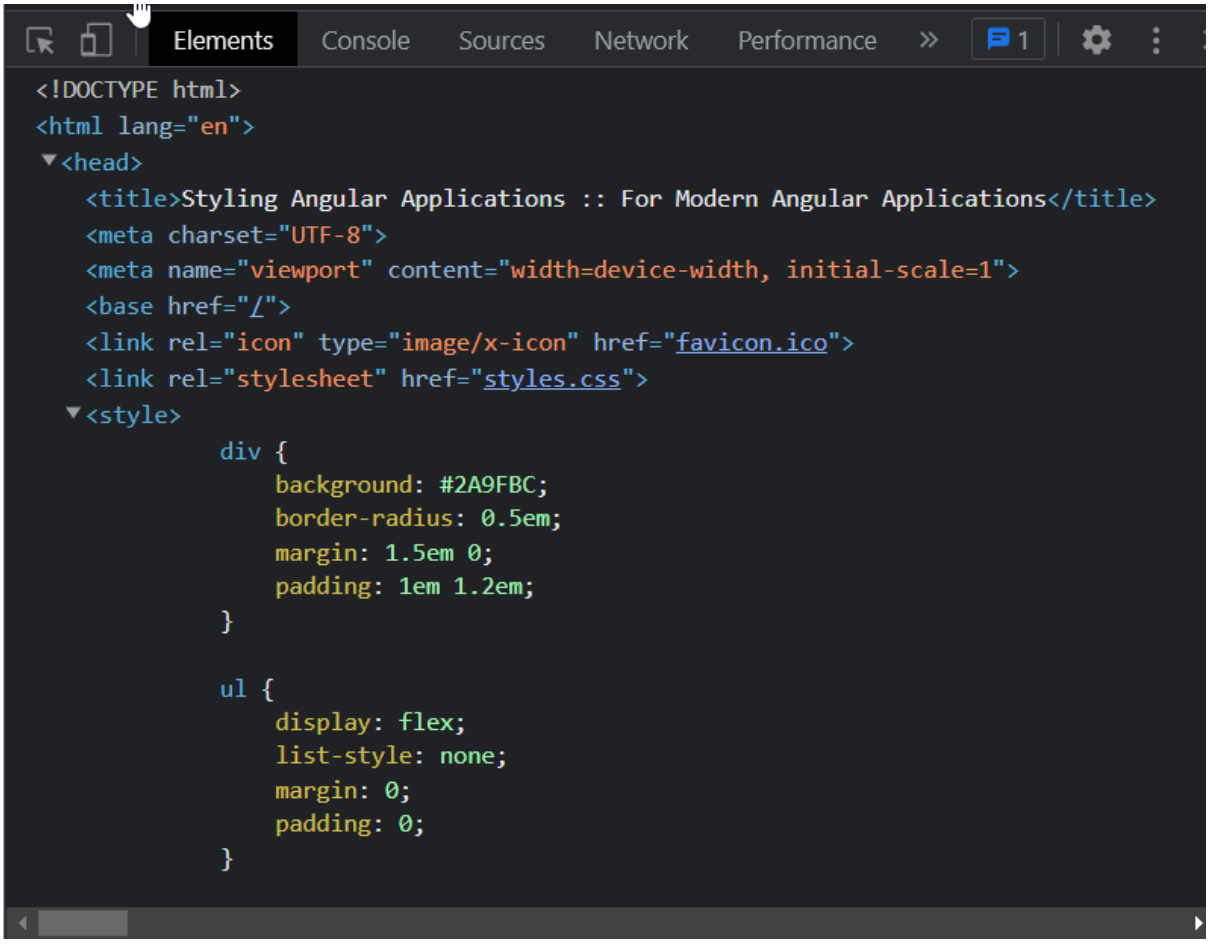


First off, to change the view encapsulation mode, we can only do it on a per-component basis right within the Component decorator with the encapsulation property. Before we can use this property though, we'll need to import ViewEncapsulation from Angular Core. Now let's start by setting our ViewEncapsulation to None. So as you may have expected, this completely turns off Angular's shadow DOM emulation and simply inserts our component styles unscoped into the head of the document. This means that they will now apply everywhere, and as we can see, in this case, they are now applying to all divs.

So if we inspect our component markup, we see that there are no longer any nghost or ngcontent attributes added.

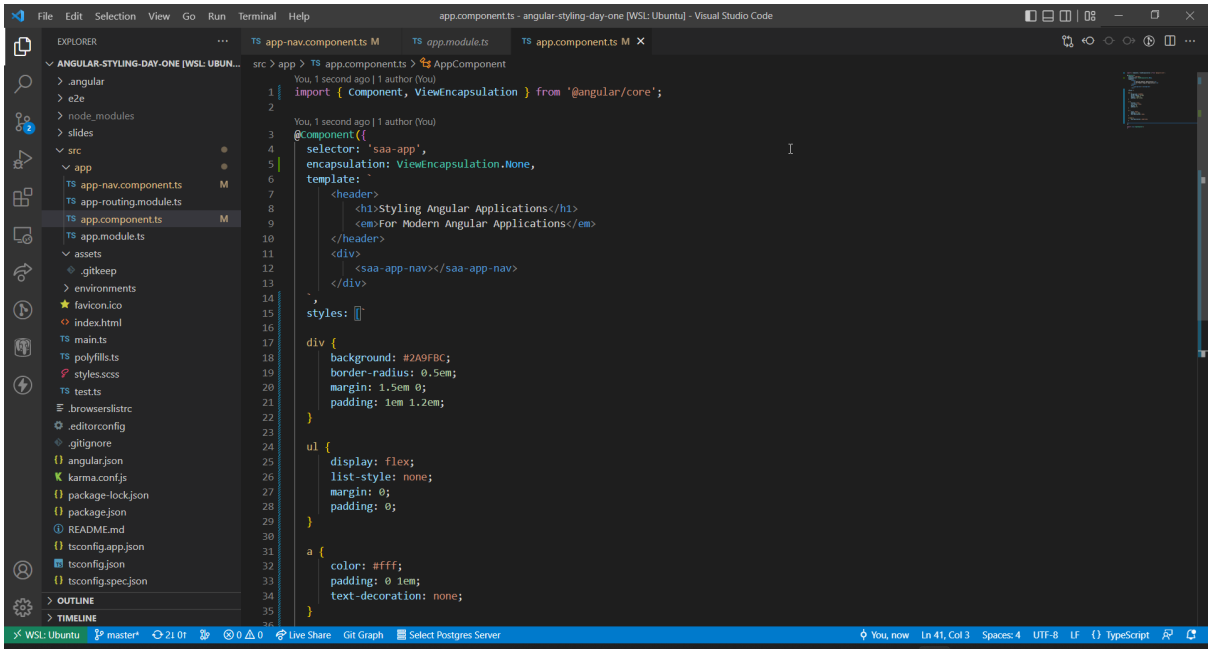


And if we look at our styles in the head, we'll see that they are added here just as we wrote them, without the addition of the custom scope name attributes.



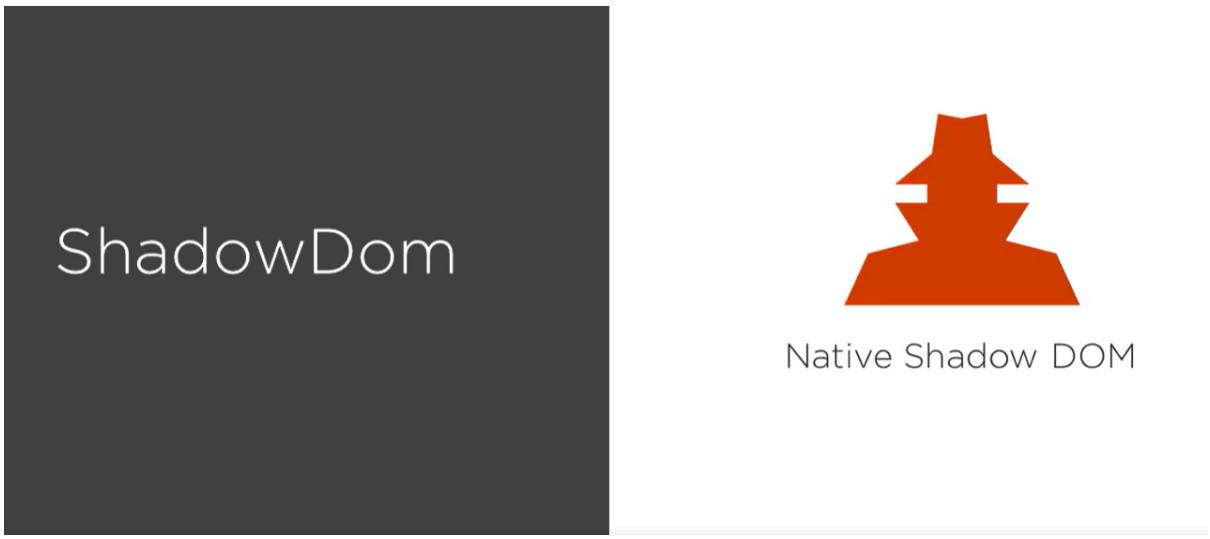
After seeing firsthand how Angular's Emulated mode emulates shadow DOM for style scoping, it just doesn't seem very great. In fact, it almost feels wrong to do it this way, but it does have some use cases. For example, we might want to apply some global styles in the root of our app.

A good way to do this is to set ViewEncapsulation to None on our app component and then add these global styles right within this component.



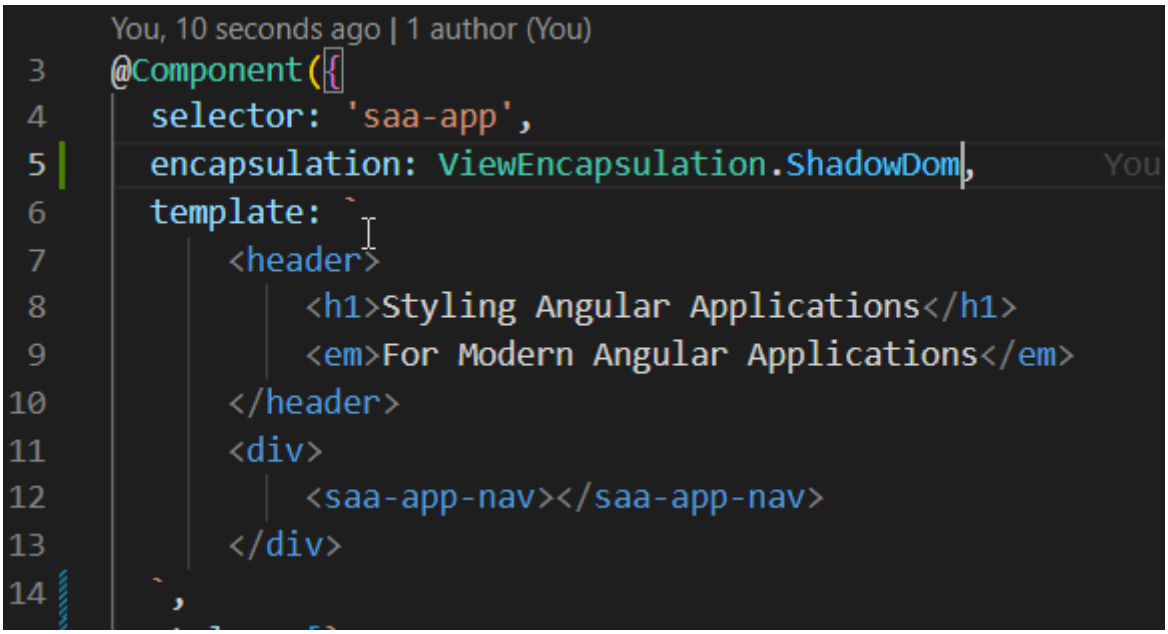
Since the app component serves as the root for our app, this is a great place for these styles to live. Any styles here will live in the global scope and apply to all elements. And since this is the first component in the tree, it's styles will be declared first, meaning that all lower-level styles from the components below will be able to override them. So even though it's not as cool as the others, you may still need to use it from time to time.

Next we have ShadowDom mode.

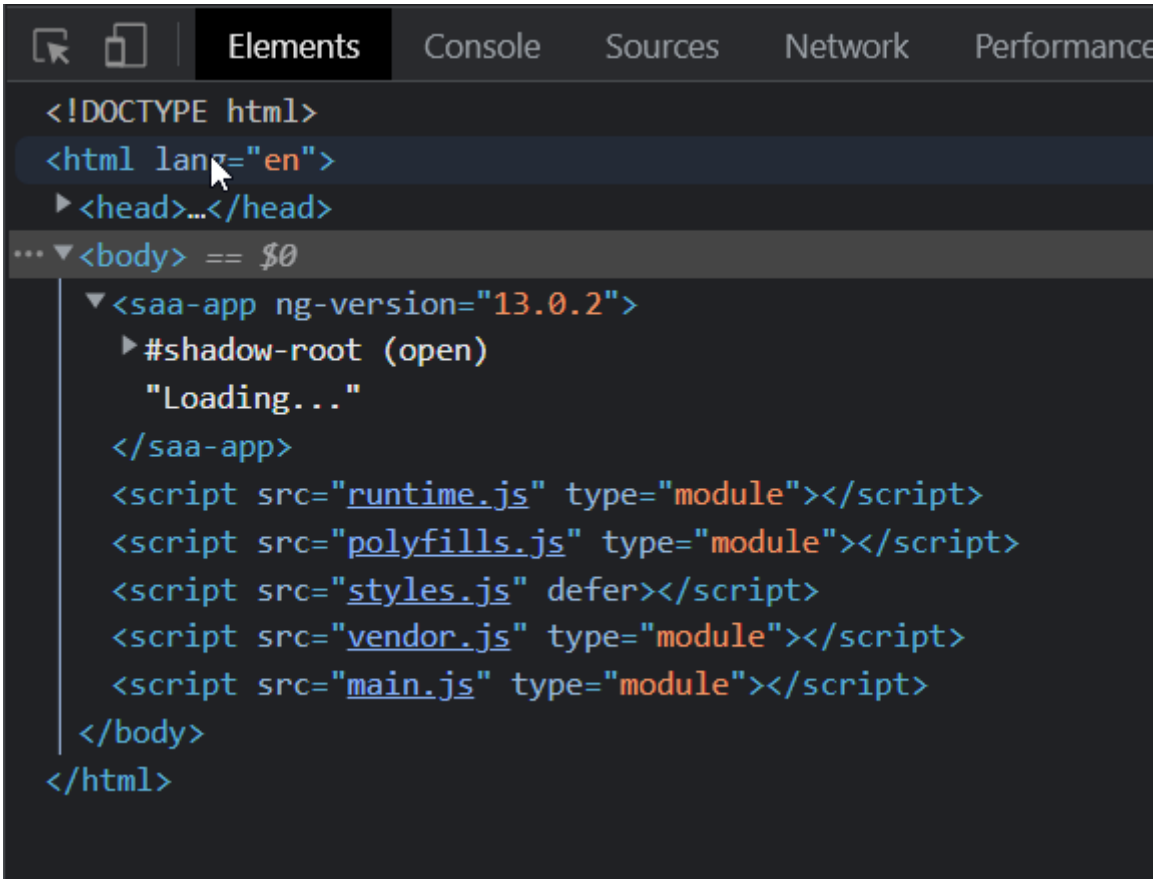


This will use the actual native shadow DOM for browsers that support it. There will be no emulation, but instead an actual shadow root will be created, which will isolate the markup and CSS outside of the scope of the parent document.

So, if we set ViewEncapsulation to ShadowDom, when the app reloads, we can inspect our markup.



We now see that this component has an actual shadow root we can expand. And when we look inside, we see that we now have actual native CSS encapsulation and that we do not have the Angular scope name attributes on our DOM nodes. We are able to see this because we are using Chrome, which has support for shadow DOM, but it would be broken in older browsers.



Shadow DOM is really okay to use if you're not too concerned with supporting older browsers and if you're not planning on using the host contact pseudo class, which we'll look at later. But you may just want to stick with the default emulated mode for the widest range of support.

Pretty much, for the rest of this section, we'll be focusing on components using the default Emulated mode. And with that in mind, let's move on to exploring the different ways that we can add styles to components and the different ways that Angular will handle them.

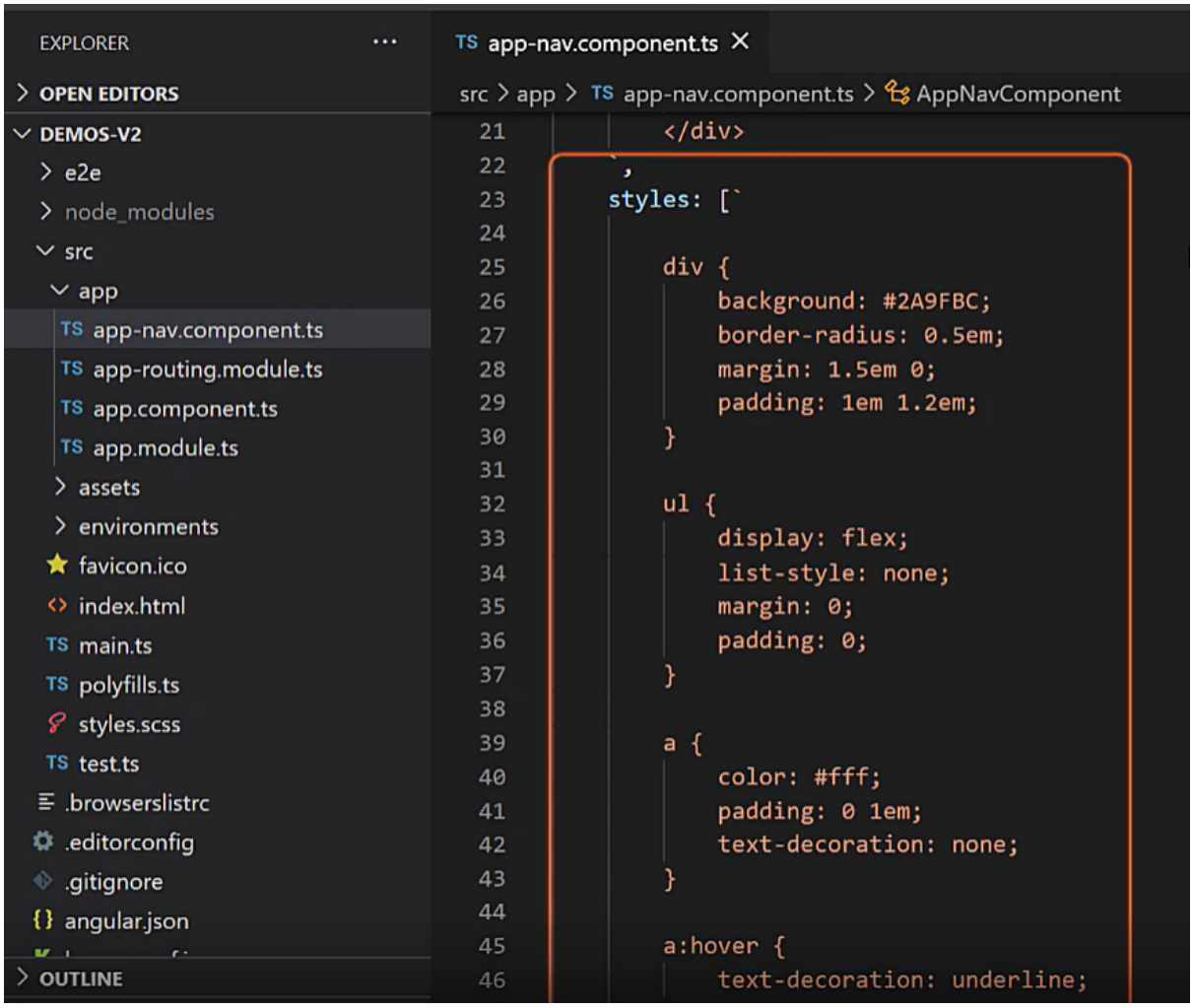
Component Styles

There are several ways that we can add CSS in Angular. So far, we've seen that we can add styles right in the component metadata with the styles property, so that's one way we can do it. Other ways we can include styles in components include adding embedded style sheets right within the component template, linking to external files from the component template with the link tag, using CSS imports, and

adding them in line in the markup themselves, and then my personal favorite, using an external style sheet with the styleUrls property.

So let's take a closer look at each of these different methods.

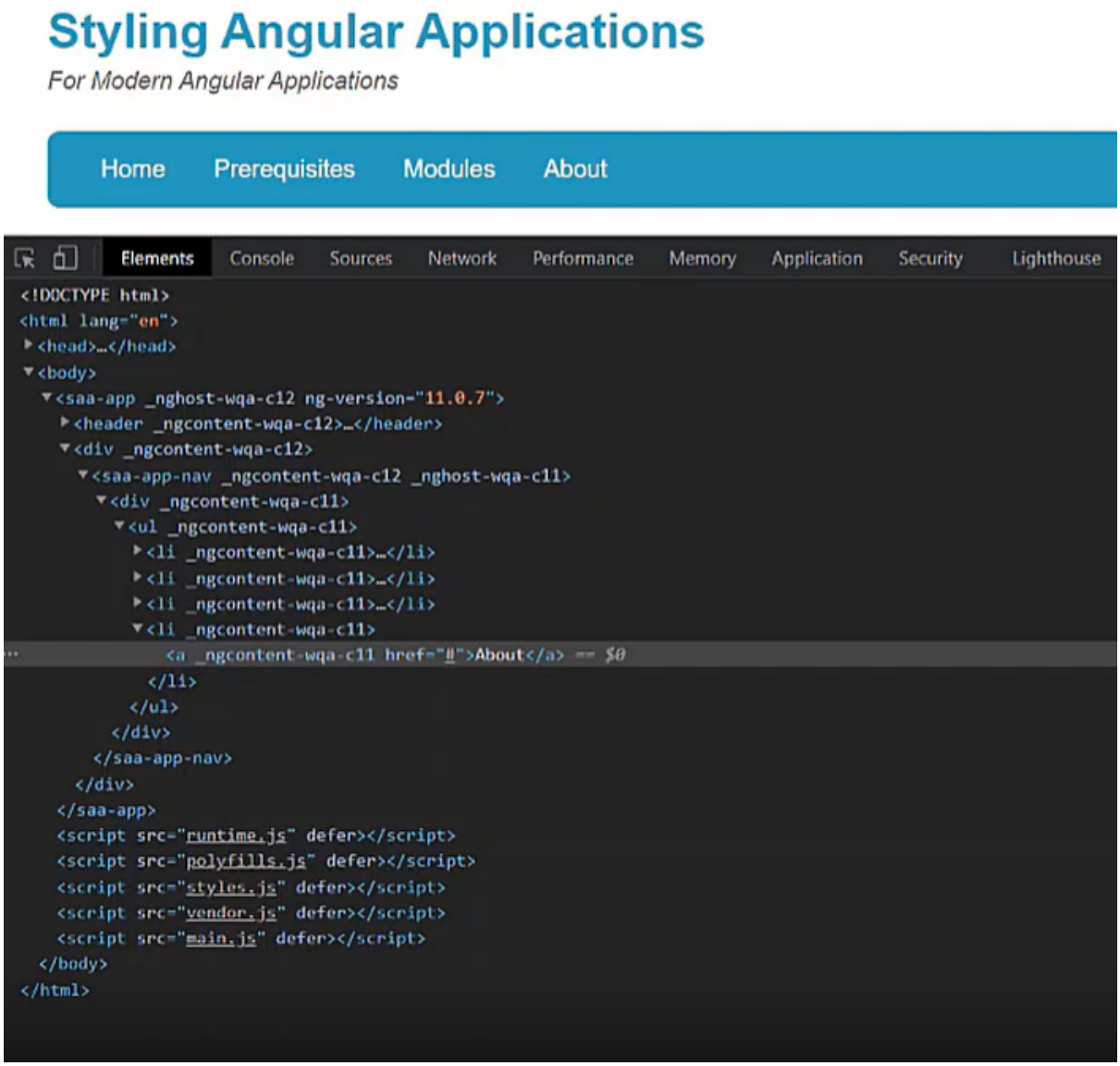
First step, we have the method that we've been using so far, the styles property. So with the styles property, we simply add styles as we would within an external style sheet, but we're just adding them here within the styles block. And we've already looked at how Angular processes these styles with the custom scope name attributes and inserting them in the head, so we won't spend any more time on them here.



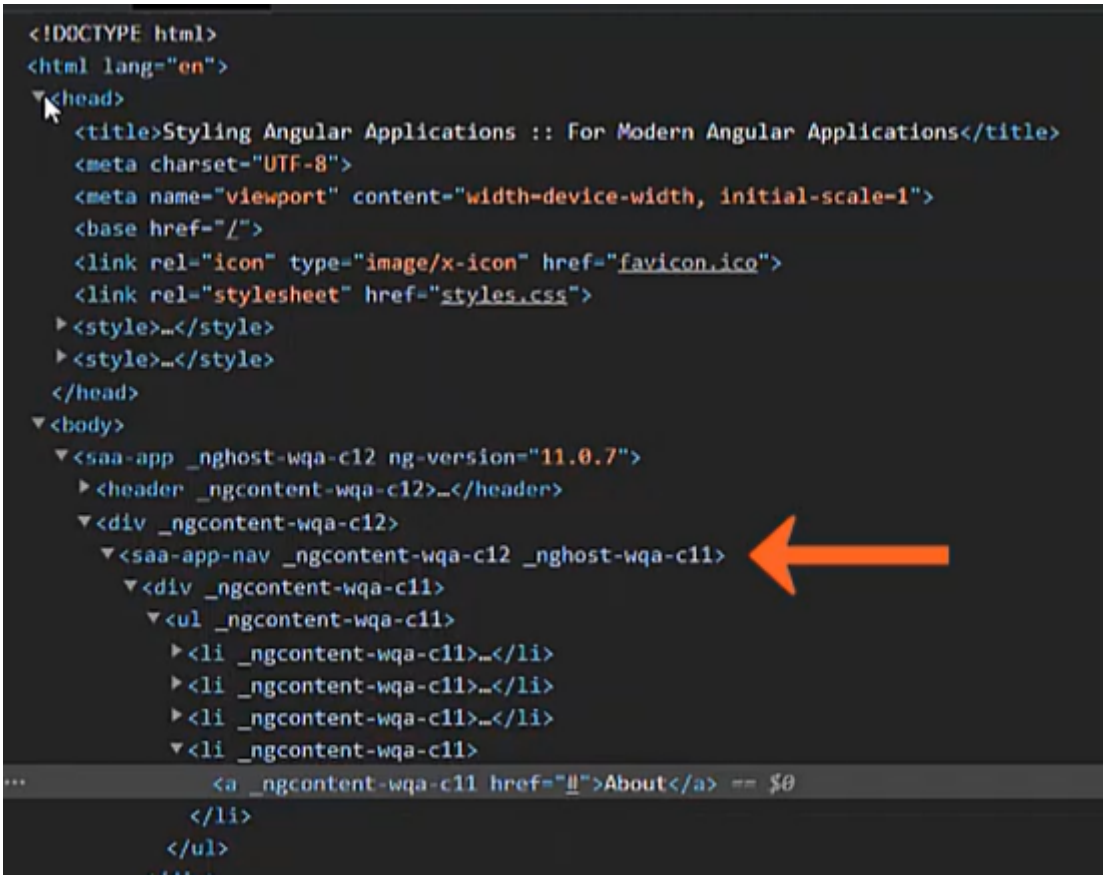
Instead, let's move on and look at adding styles directly within the template itself.

To add styles directly into the component template, we just need to add the style element and then add styles within it.

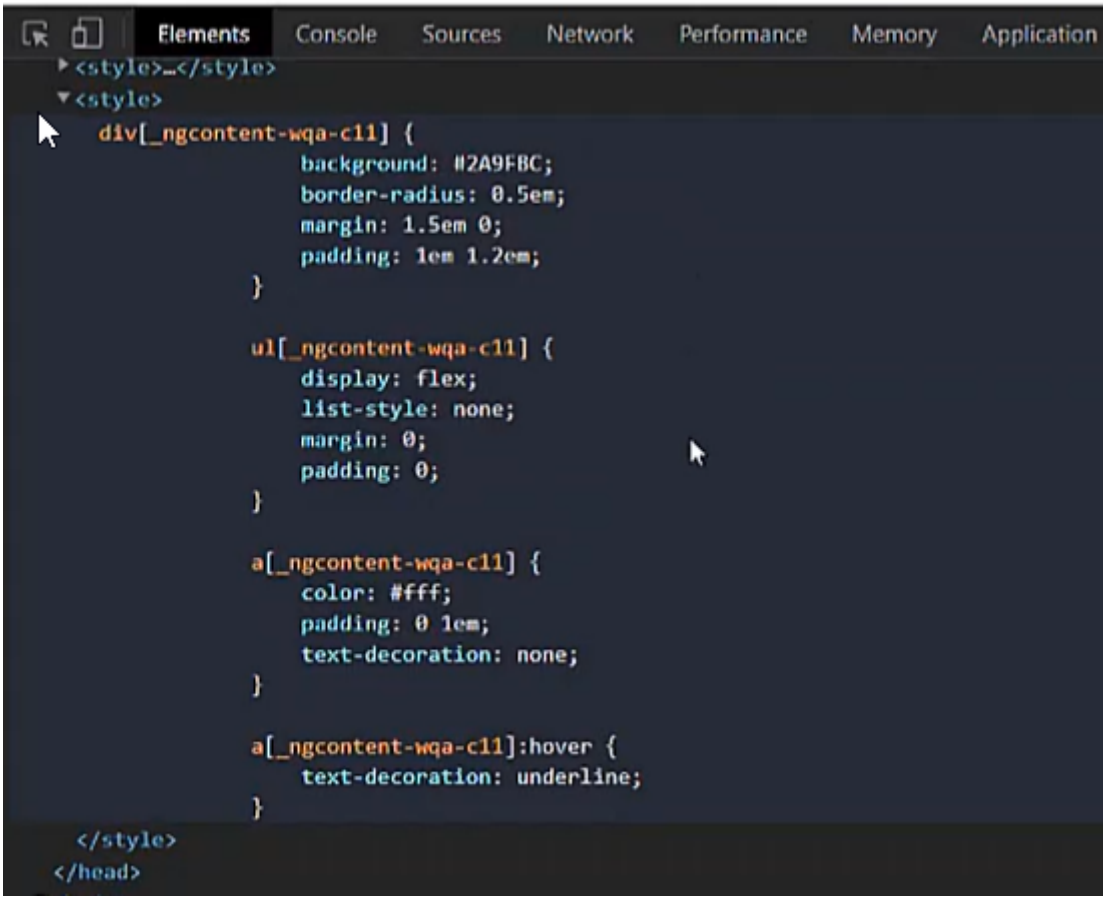
When we look at this in the browser, we can see that our styles are all good.



This thing looks as we'd expect it to. What's crazy, though, is that these styles will be treated just as those using the style property in component metadata, meaning that once rendered, they will have been moved from their original location in the template and inserted into the head. They will also get all of the crazy Angular scope name attributes as well. Let's take a closer look by inspecting this.



So here we can see that Angular has added the scoping attributes, and we can see that there is no style block here. So when we look at the head, we can now see our style block has been inserted here instead.

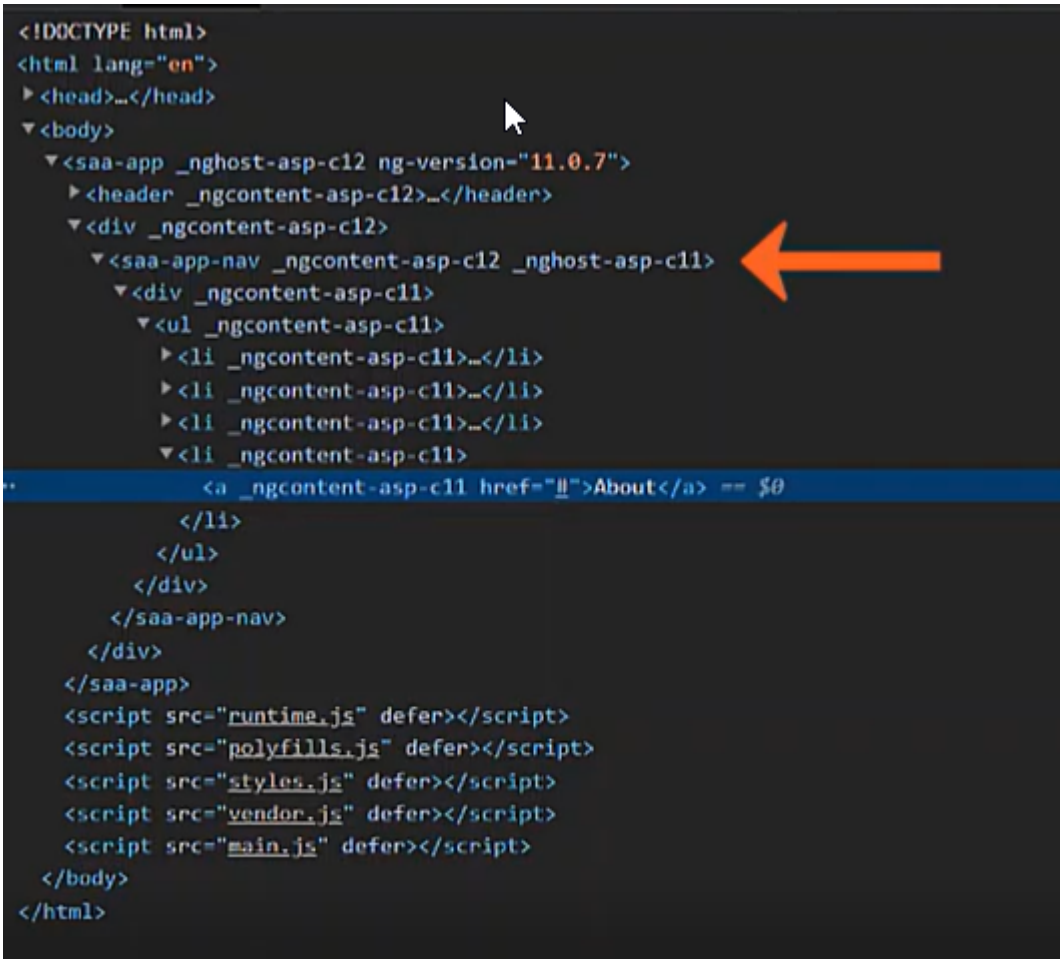


So this method works basically the same as when we add them using styles property; no real difference other than where we add the styles.

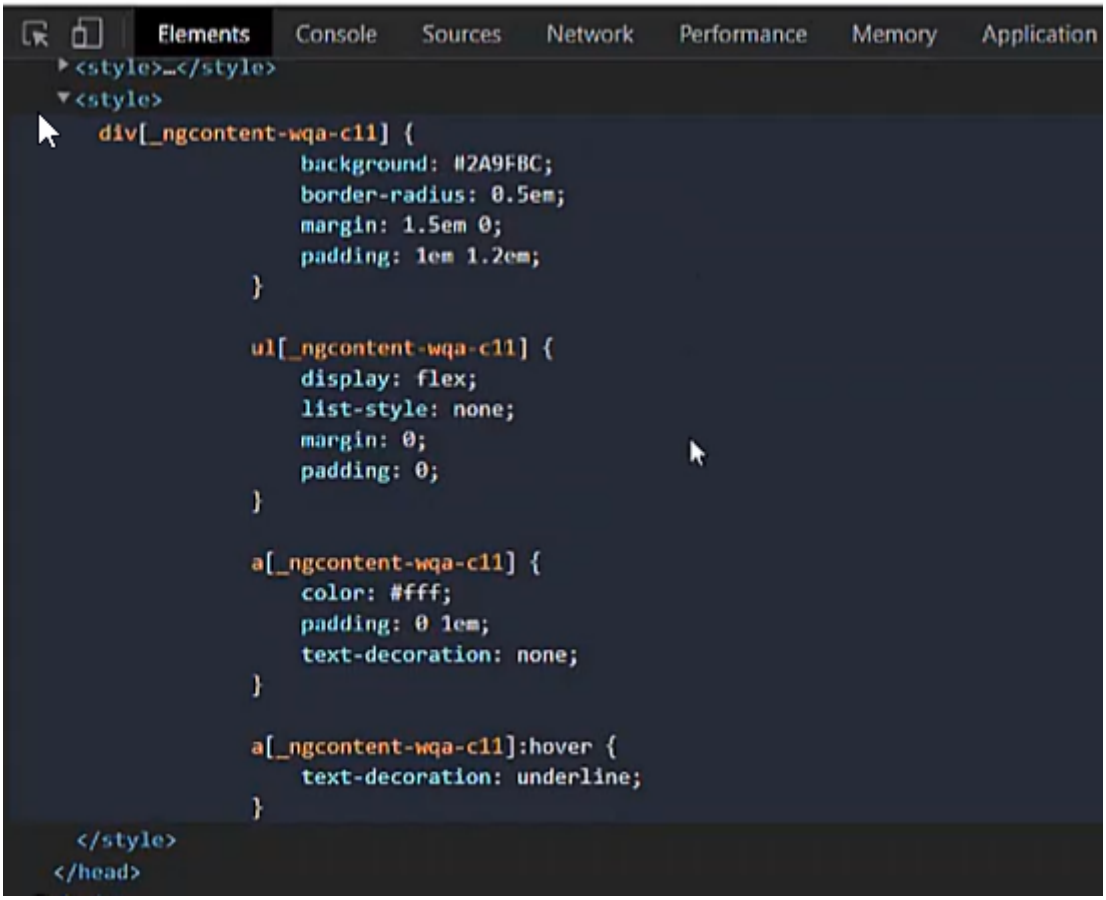
Now, if we don't want to use the styles property or an embedded style sheet, we can add them externally and then add a link tag to reference them. This is very similar to how we add styles in traditional web applications.

We add a link tag with the relationship attribute of style sheet, followed by an href with the location of our external style sheet. And to follow the Angular style guide, we'll name this file with the same name as our component, but we'll give it a CSS extension.

Angular will handle this method in the same manner as the embedded style sheet. So if we examine our component, here we can see that Angular has added the scope name attributes.

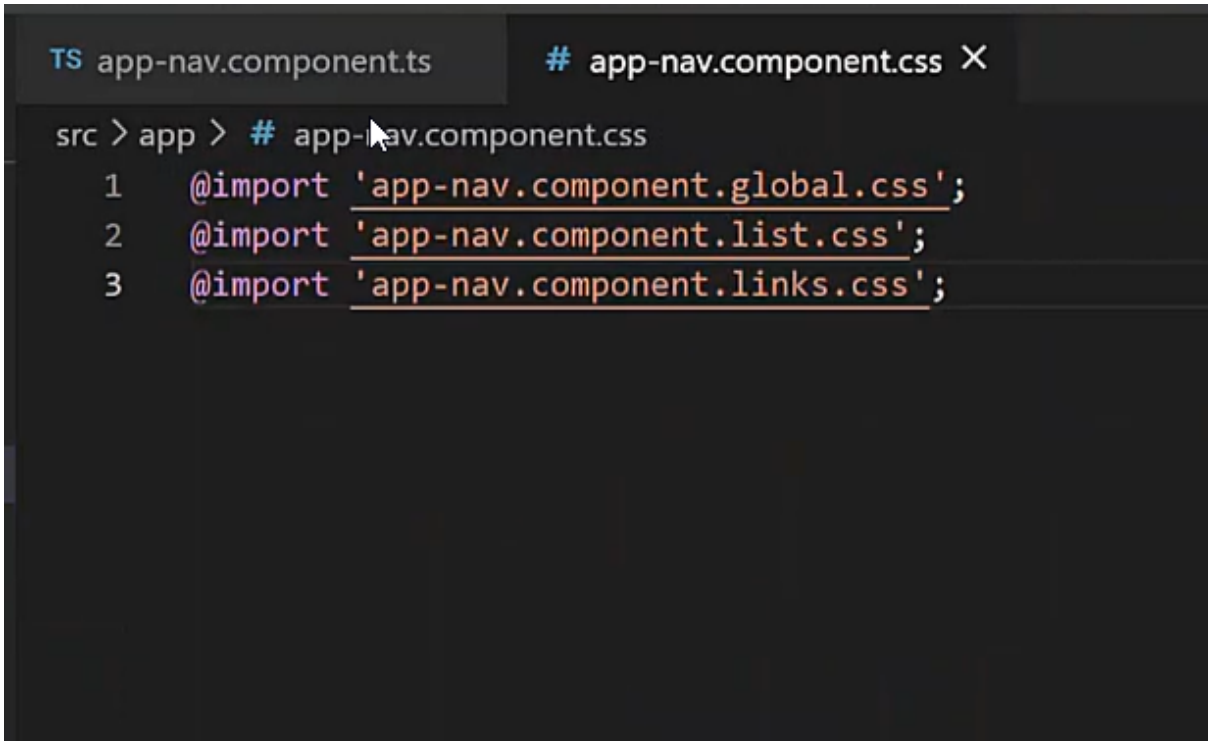


Also, we can see that our link to the style sheet has been removed. Well, if we take a look at the head again, we can see that Angular has properly scoped our styles and moved them here once again.



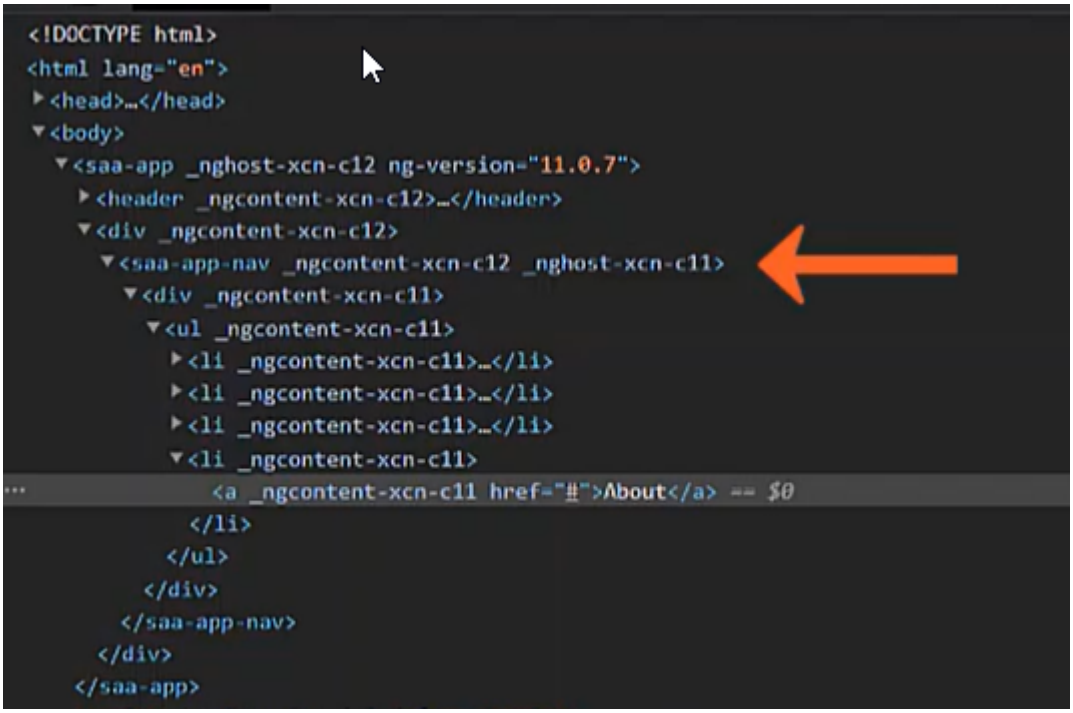
This method is nice because it separates out our styles into their own file and puts it in a familiar CSS editing environment with the same end result of properly scoping the component styles and inserting them in the head.

Now what if we want to bring in multiple files, but don't want to add a link for all of them? Well, we can use CSS imports. We just add imports as we would a normal CSS file.



So how does Angular handle these imports? Let's take a look.

Here we can see that Angular has once again added the scope name attributes, and if we examine the head, again we'll see that Angular has added our styles there.



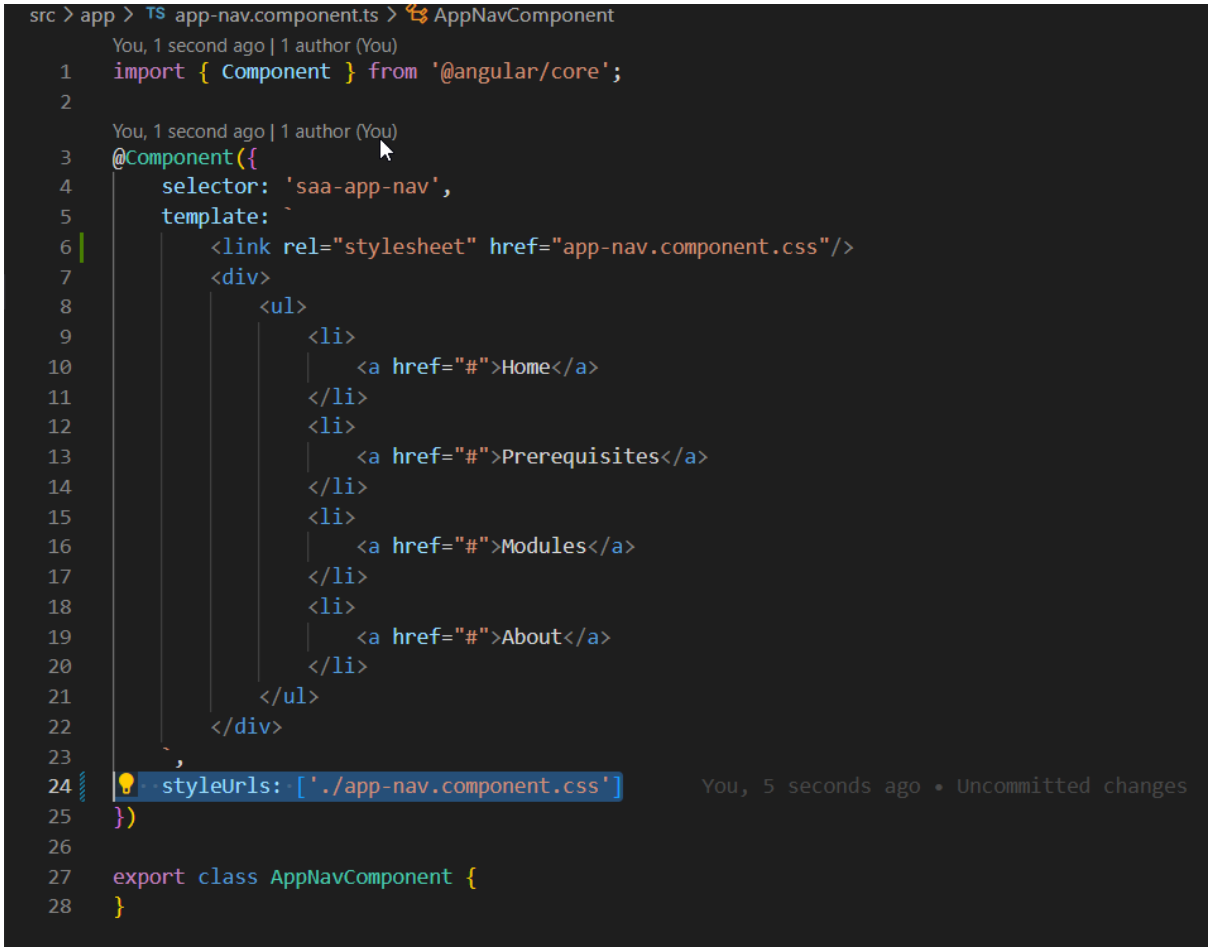
So this is pretty nice. It allows us to potentially break up larger style sheets into much smaller, more organized files. Another way that we can add styles to our components is to add them directly inline on the markup elements themselves.

This is probably not how we want to do it, for obvious reasons, but it's good to know all the different ways that we can add styles so that we can make the best decision for any given situation that may arise. So if we add styles inline, Angular will actually not do anything with these styles, they'll just render as expected.

Okay, so those are all great, but my personal favorite method is, again, using external style sheets like we saw earlier, but this time using Angular styleUrls property to link to them, as opposed to a traditional link tag. I like working in this way because, for one, it's easier to set up. It doesn't require a link tag with a relationship attribute and the path to our file. Instead, it just needs a property with a file name.

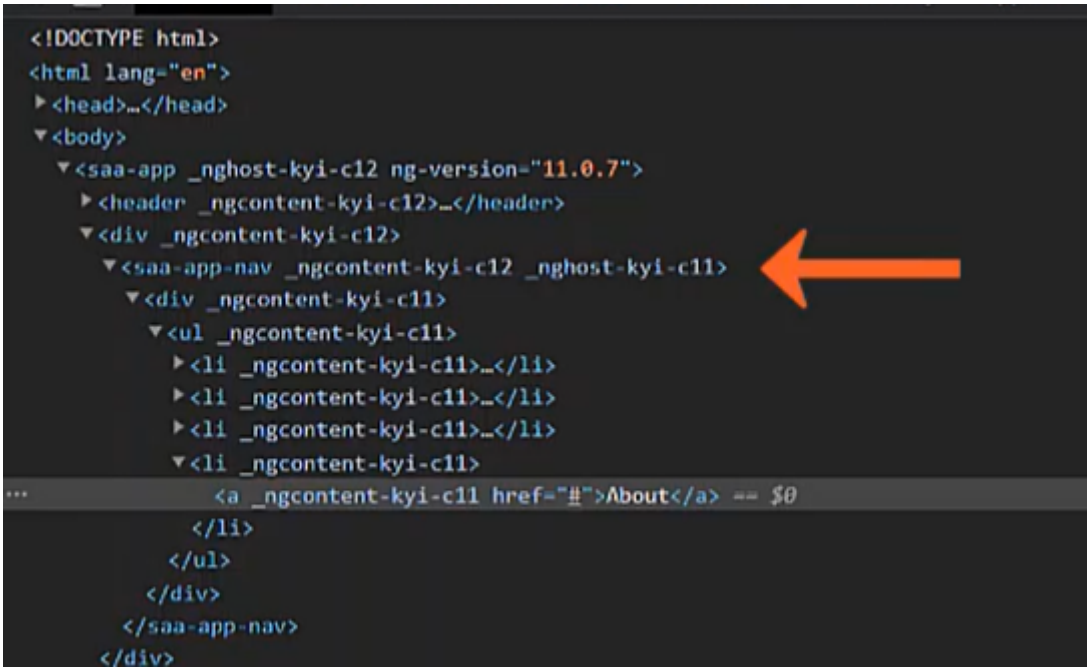
I also like it because it's how most of us are used to working with CSS. One of the biggest drawbacks to the other methods is that we're writing CSS in TypeScript, so depending upon our editor, there is a lack of syntax highlighting and code completion for CSS. That's what's great about using external files is that they have a css extension, meaning that our editor will open these files in their proper CSS editor.

To use this method, we need to first add a CSS file for our component, just as we did earlier. Next we need to tell our component to use this file, so we'll add it to the metadata with the styleUrls property. This property will take an array of URLs to existing style sheets, but we currently only have a single file, so we add our file here.

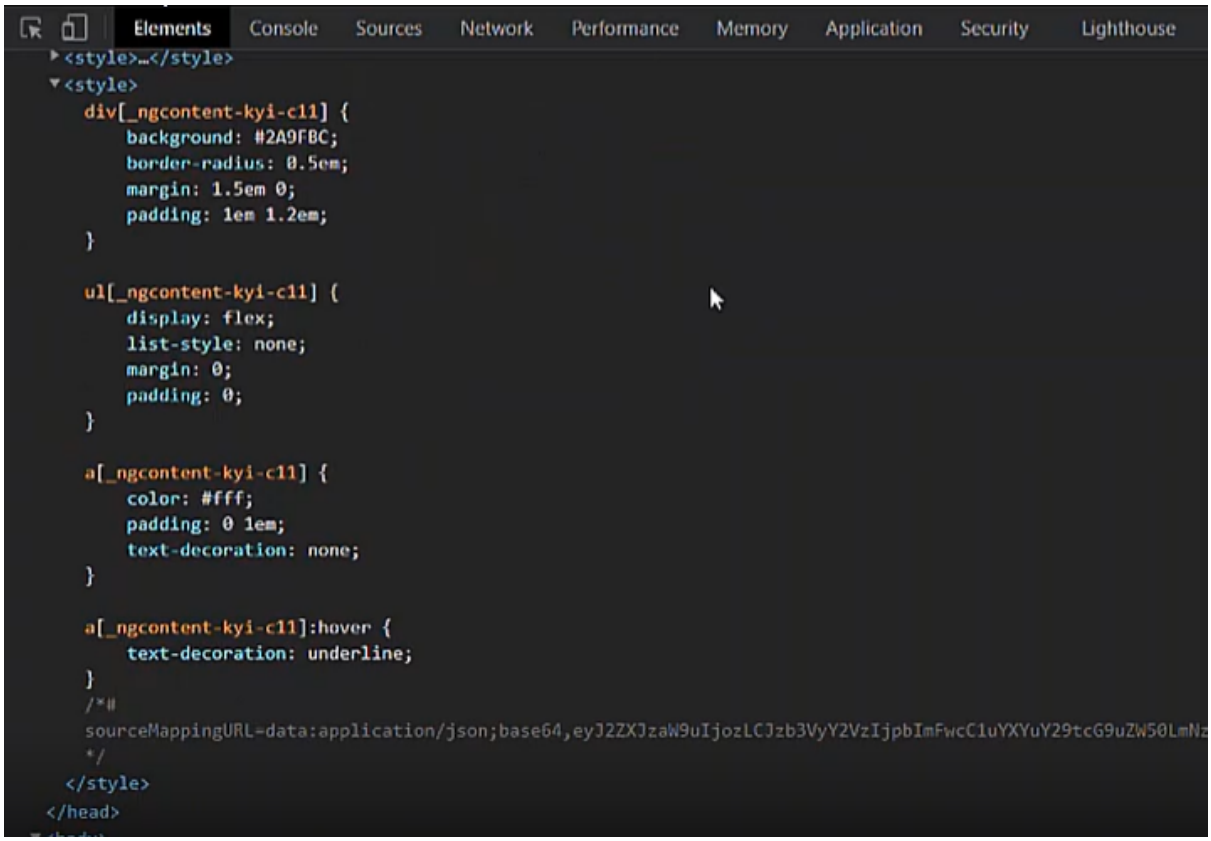


In this case, the style sheet is in the same directory as our component, so we can use the ./. Okay, so how does Angular handle this? Well, let's take a look.

When we inspect our component, we can see once again that the scoping attributes are added, but where and how are the styles added this time? Well, the other methods put styles in the head, so let's take a look there, and there they are.



So Angular simply inserts these styles into a style block in the head, just like the other methods.

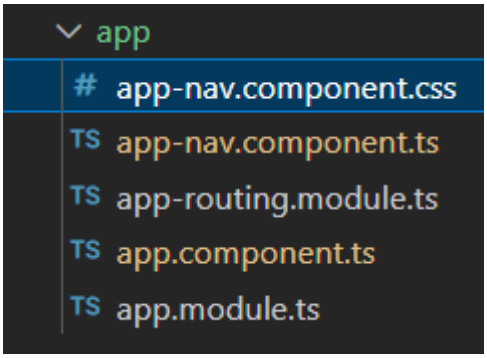


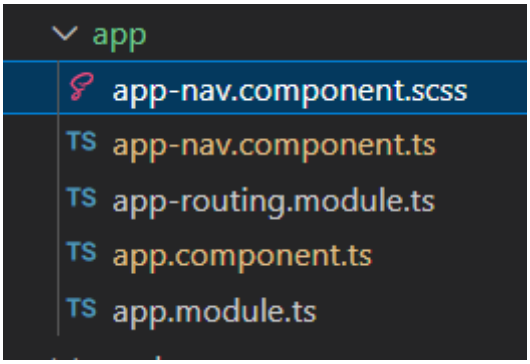
Pretty cool stuff going on here. This approach is very similar to the traditional link element to point to our style sheet, but the main difference is that it's simpler to set up. It just requires the styleUrls property and the name of the file, as opposed to the link element with a rel and href.

At this point, it's important to note that we can take the same approach with our template, and in most cases, this will be a better way to work as well. So we simply add an HTML file next to our component with the same name, and then we reference it from our component metadata using the templateUrl property.

So up to this point, we haven't used a preprocessor like Sass, Less, Stylus, etc., but I would recommend that we do. It just makes working with CSS in large applications so much easier to work with, so for the rest of this section we'll be working with Sass.

In order to do this, we'll simply convert our file to the scss extension.





There, now we can use Sass.

NEXT: Emulated CSS Selectors