- Roadblock spec
- SmartPick
- Handover notes
- Archived
  - [Manager/Mentor Document] New
  - Angular 2 Framework Upgrade
    - Angular 2 Migration Findings
    - New Zoosk Responsive Site: CS
    - **Results from testing CSS sco**
    - Suggested .scss file structure in
    - UX Deliverables for Site Re-des
  - Best Practices for Team Collaborat
  - Browser Gotchas and Quirks
  - Browser Roadmap 2019
  - Browser Sprint Retrospectives
  - Browser Team Workflow
  - Browser Tips and Tricks
  - Code Review Guidelines
  - CSS Bootcamp Curriculum
  - Cupid
  - Cupid Setup and Build Guide
  - Deprecated - Browser
  - Engineering Project Specs / Propo
  - Framework Upgrade Path(s)
  - Git

Space tools

Space tools

Results from testing CSS scoping in Angular 2

Created by Sue Anna Joe, last modified on Dec 27, 2018

Intro

Since I'm new to Angular 2, I looked into the options for adding CSS to an app built on this framework. I was particularly interested in the idea of scoping CSS within Angular 2 components and how that affects the ecosystem of style rendering. I forked a simple Angular 2 demo app on Stackblitz, made it look like our Store v3 Upgrades UI, and tested scoped CSS. This demo helped me to:

- learn about the options for adding styles to an Angular 2 app
- understand the effects of scoped CSS
- determine the best route for devs to build UI on Angular 2

tl;dr

- We should use either emulated mode (which is the default) or "none" for View Encapsulation for the following few reasons:
  - Native Shadow DOM does not have wide browser support.
  - CSS scoping in Shadow DOM keeps us from customizing a component element's styles based on context.
- We should use global styles as we do now. Scoping styles causes the following issues:
  - It affects CSS selector specificity.
  - It loads our styles in the browser in an order that can work against us.
  - If we have a scoped parent component, we can't customize the styles of its children components.
  - Scoped styles via the **styleUrls** component property does not recognize global mixins and variables.
- We should use a separate component HTML file instead of adding our DOM to a component decorator's **template** property. This will make the code more readable and manageable.

View Encapsulation in Angular 2

Overview

According to this article, Angular 2 has a concept called View Encapsulation. This is how it handles Shadow DOM, and it does it in three ways:

> - **None**: All elements are spit out - no Shadow DOM at all.
> - **Emulated**: This actually tries to emulate Shadow DOM to give us the feel that we are scoping our styles. This is not a real Shadow DOM but a strategy to make all browsers smile at our code.
> - **Native**: This is the real deal as shadow DOM is completely enabled. **Older browsers can go to hell.**

Emulated is the default mode.

Below are some general points from other sources:

- Scoped CSS associated with both native Shadow DOM and emulated mode does not affect other elements outside itself or its children components, if any. In effect, this prevents classname/selector collisions.
- Native Shadow DOM also ignores styles outside itself (global CSS).
- Native Shadow DOM does not have wide browser support.
- In emulated mode a component can be affected by global styles but not always. See results from scope testing.

Takeaways

- Due to the lack of browser support for native Shadow DOM, we should not use this for our new responsive site.
- Setting a component as native Shadow DOM also prevents us from customizing its CSS in different contexts.
- Because of how we code our CSS, we don't have an issue with classname/selector collisions.

Options for adding styles

Six ways to add styles to a component

All but #6 would cause the CSS to be scoped to the component.

```
/*************************************************/
1 - Inline styles on elements
/*************************************************/
@Component({
    template: `
        <h1 style="color: blue;">Zoosk</h1>
    `
})
/*************************************************/
2 - Using style tags within the template
/*************************************************/
@Component({
    template: `
        <style>
            button {color: blue;}
        </style>
    `
})
/*************************************************/
3 - Using a link tag
/*************************************************/
@Component({
    template: `
        <!-- Per the Angular guide we must use a relative URL so that the AOT compiler can find the stylesheet -->
        <link rel="stylesheet" href="../assets/hero.component.css">
        <h1>Zoosk</h1>
    `
})
/*************************************************/
4 - Using the styles array to add declaration blocks
/*************************************************/
@Component({
```

```
    styles: [`
        button {color: blue;}
        footer {background: gray;}
    `]
})
/*************************************************/
5 - Adding an array of external css or scss files
/*************************************************/
@Component({
    styleUrls: ['button.component.scss']
})
/*************************************************/
6 - Using global styles
/*************************************************/
You can choose to use none of the above methods and instead rely on globally scoped css/scss partials that are imported into a primary scss file. This i
```

Takeaways

- Considering general best practices and the fact we use Sass, we should avoid options 1 - 4.
- Creating a separate component HTML file would give us better organization, especially for longer DOM structures, than using the component decorator's **template** property.

Style priorities in the browser

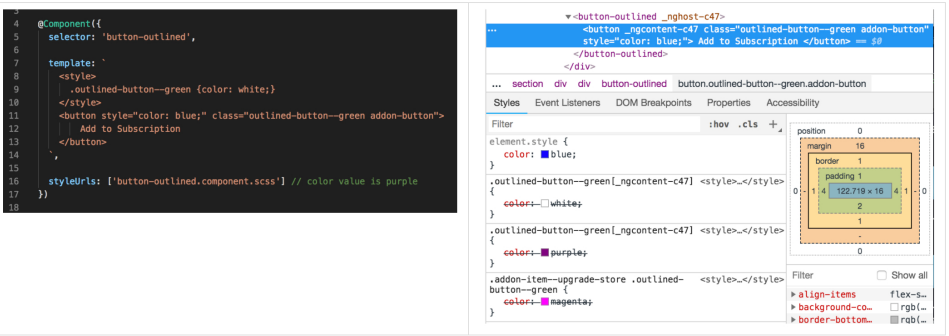
For informational purposes, I've included a priority list, from highest to lowest, and screenshots showing the order in which Angular 2 styles are loaded in the browser.

Outline form

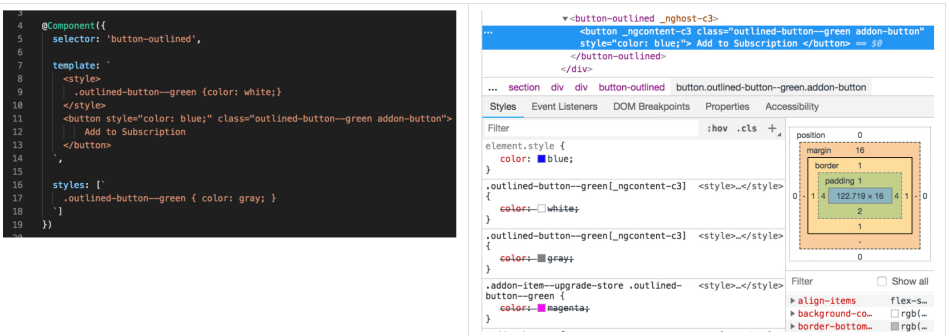
```
/*************************************************/
PRIORITY 1 *
/*************************************************/
@Component({
    template: `
        <h1 style="color: blue;">Zoosk</h1>
    `
})
-- OR --
<h1 style="color: red;">Zoosk</h1> (This would be in a component HTML file)
/*************************************************/
PRIORITY 2:
/*************************************************/
@Component({
    template: `
        <style>...</style>
    `
})
/*************************************************/
PRIORITY 3 *
/*************************************************/
@Component({
    styleUrls: []
})
-- OR --
@Component({
    styles: []
})
/*************************************************/
PRIORITY 4:
/*************************************************/
Global styles

* You can only have one or the other in your .ts file. If you have both, the one that is farther down the document will parse.
```

Screenshots

CSS ordering with styleUrls: [ ]



CSS ordering with styles: [ ]



Scoped styles

By default an Angular 2 component is implemented in emulated Shadow DOM mode. This means the DOM is not rendered as true Shadow DOM; however, CSS can still be scoped to a component in emulated mode.

For our Zoosk product we often run into these scenarios when creating UI:

- A UI element can appear both by itself and grouped with other things within a larger UI element.
- A UI element needs custom styles depending on where it appears, like margin.

Because scoped CSS affects how styles are applied to elements on a page, I tested scoped styles on an element as well as its parent to find any limitations for us.

Problems with scoping styles

1. Scoping styles causes an **_ngcontent-*** attribute to be rendered on the parent HTML element in a component template. This attribute in turn is used as a selector in that element's CSS declaration block and increases specificity. It is undesirable for the app the affect selector specificity in our styles. e.g.:

```
footer[_ngcontent-c3] { bottom: 0; left: 0; position: fixed; right: 0; color: pink; } /* base CSS (scoped) for all footers */


.account-settings footer {color: blue;} /* non-scoped customization based on context */


/* Both declaration blocks have the same specificity, but the footer's scoped CSS will render in the browser after the custom override. Therefore, the s
```

2. Global mixins and variables are not recognized in separate component-based scss files when specifying them in the **styleUrls** array. We would have to import global files into the scoped stylesheet. I would prefer not to do this as it's an extra thing we need to do with each component-based scss file.

Results from testing scoped vs. non-scoped styles

Elements	Scoped	Not scoped (global)	Receives component attribute _ngcontent-* (which increases selector specificity in the CSS)	Results
Parent component styles	x		x	Any parent CSS trying to override a child's CSS will not be parsed.
Child component styles	x		x	
Parent component styles		x		Any parent CSS trying to override a child's CSS will be parsed. Child CSS can possibly take higher priority because: • It is loaded after the parent's styles. • Its declaration block includes the child's component attribute selector which increases the child CSS specificity.
Child component styles	x		x	
Parent component styles	x		x	Any parent CSS trying to override a child's CSS will not be parsed.
Child component styles		x		
Parent component styles		x		Any parent CSS trying to override a child's CSS will be parsed and takes higher priority.
Child component styles		x		

Takeaways

- A component's scoped CSS, which would likely be the base styles for that component, are loaded last in a browser. This gives those styles higher priority, and potentially our customizations would be overridden.
- Scoped parent styles do not affect any children component styles which is limitation for us.
- Global styles would give us the flexibility we need.