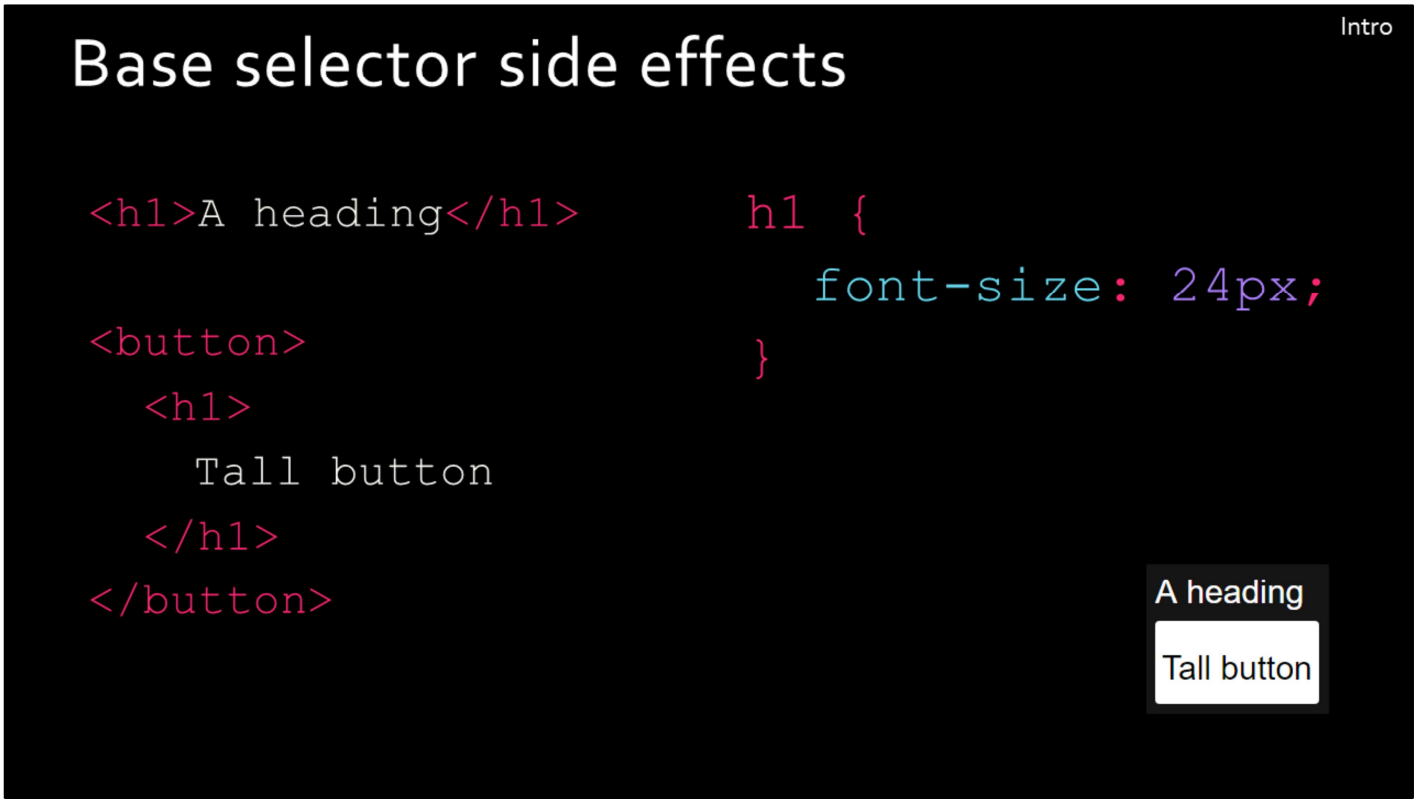
- Intro – I'll explain what this global scope thing is all about and the issues we may have to tackle with that
- Different approaches on how to circumvent global scope issues

- Scope = Is an area in which certain program logic applies – in our case CSS rules – global by nature
- The scope of „plain" CSS always spans over the whole HTML document
- This same concept has existed since the inception of CSS
- In general, **cascading** rules to the whole HTML structure is just what had been intended, hence the name
- Following I'll show you some short examples about what problems could arise from that
- Frankly, some of these examples may seem a bit far-fetched, but I'd intended to keep the intro as short as possible



Base selector side effects – Intro

- Here we got two elements - a standalone heading, but also a tall button - h1 font size tall!
- The h1 selector with the font size rule applies to both the discrete heading itself and the h1 which is shown as the button content
- When modifiying such a generic base selector this may have a broad impact everywhere we use h1 markup elements

Base selector side effects – The Problem

- You may have guessed it, of course we'd like to change the heading font size now
- Because we have such a generic selector, this also – unintentionally? – affects the button font and actual size, „breaking" the UI design
- In a big project, not all affected elements may have been appropriately checked and adapted now
- Most of the time nobody would dare to modify such a generic selector in a big project (Oh god what could be the side effects?!)

Base selector side effects – "solution"

```
<h1>A heading</h1>

<button>
  <h1>
    Tall button
  </h1>
</button>
```

```
h1 {
  font-size: 36px;
}

button h1  {
  font-size: 24px;
}
```

Base selector side effets – possible solution – increase specifity

- Two classes have a higher specifiy as one class
- CSS specifity is a moderately complex topic, it may not be really clear which rules apply when
- If there is no agreement in the team about how to incorporate CSS specifity, this could lead to a uncontrolled growth in CSS selector numbers
- This also may lead to ever more specific CSS rules and tons of obsolete CSS selectors

Matches on child elements - Intro

- Initially we have got a task list with action buttons to style
- There is a CSS rule for those action buttons in the task items which defines a green text color

Red task list items – match on children

```
<div class="tasks">
  <ul class="task-items">
    <li>
      Task 1
      <ul class="actions">
        <li>Complete</li>
        <li>Delete</li>
      </ul>
    </li>
    <li>
      Task 2
      <ul class="actions">
        <li>Complete</li>
        <li>Delete</li>
      </ul>
    </li>
  </ul>
</div>
```

```
.actions li {
  color: green;
}

.tasks .task-items li {
  color: red;
}
```

- Task 1 Complete Delete
- Task 2 Complete Delete

Match on child elements - Problem

- Now we'd like to change the design of the task list items so that they have a red text color
- Unintentional side effect: The buttons now also have the same text color
- Reason: Match on child elements through the new selector, which has a higher specifity because it include an additional class
- Could happen easily in a real world scenario if you include third party components as child elements somewhere in your project

Match on child elements – bad solution

```
<div class="tasks">
  <ul class="task-items">
    <li>
      Task 1
      <ul class="actions">
        <li>Complete</li>
        <li>Delete</li>
      </ul>
    </li>
    <li>
      Task 2
      <ul class="actions">
        <li>Complete</li>
        <li>Delete</li>
      </ul>
    </li>
  </ul>
</div>
```

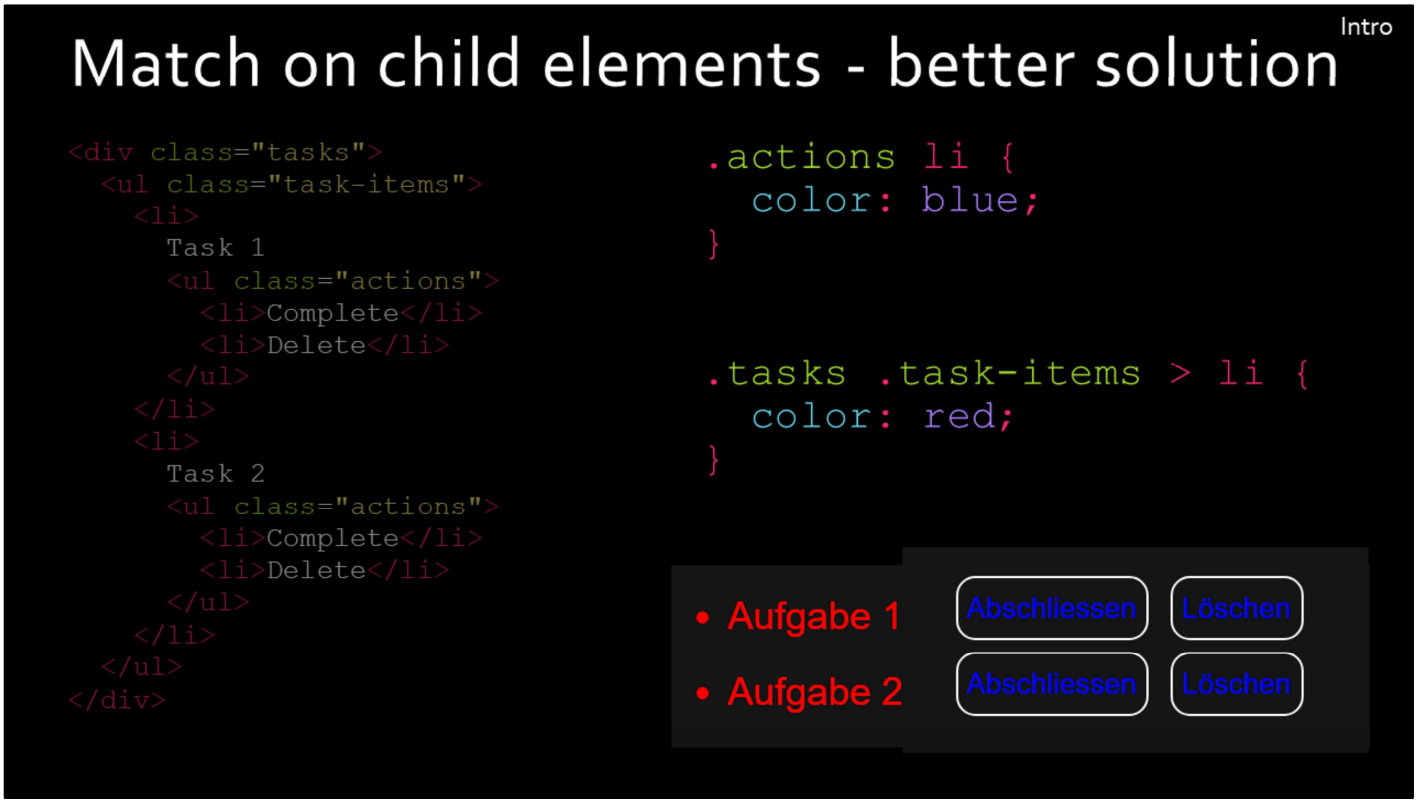
```
.actions li {
  color: green !important;
}

.tasks .task-items li {
  color: red;
}
```

- Task 1 Complete Delete
- Task 2 Complete Delete

Matches on child elements – „Solution"

- Using the „nuclear" !important CSS rule modificator
- Every other rule is overwritten, well at least if those other rules aren't „!important" as well (then the usual css specifity algorithm kicks back in – more confusion)
- Harry Roberts (csswizardry.com) says that „!important" should only be used proactively and never reactively (e.g. to define a CSS rule which should always take precedence over any other matching rule), I'd prefer to never use it at all, it could possibly always lead to later issues



Matches on child elements – a better solution

- Restrict selector match for red color to task list item LI element itself

Global CSS Scope

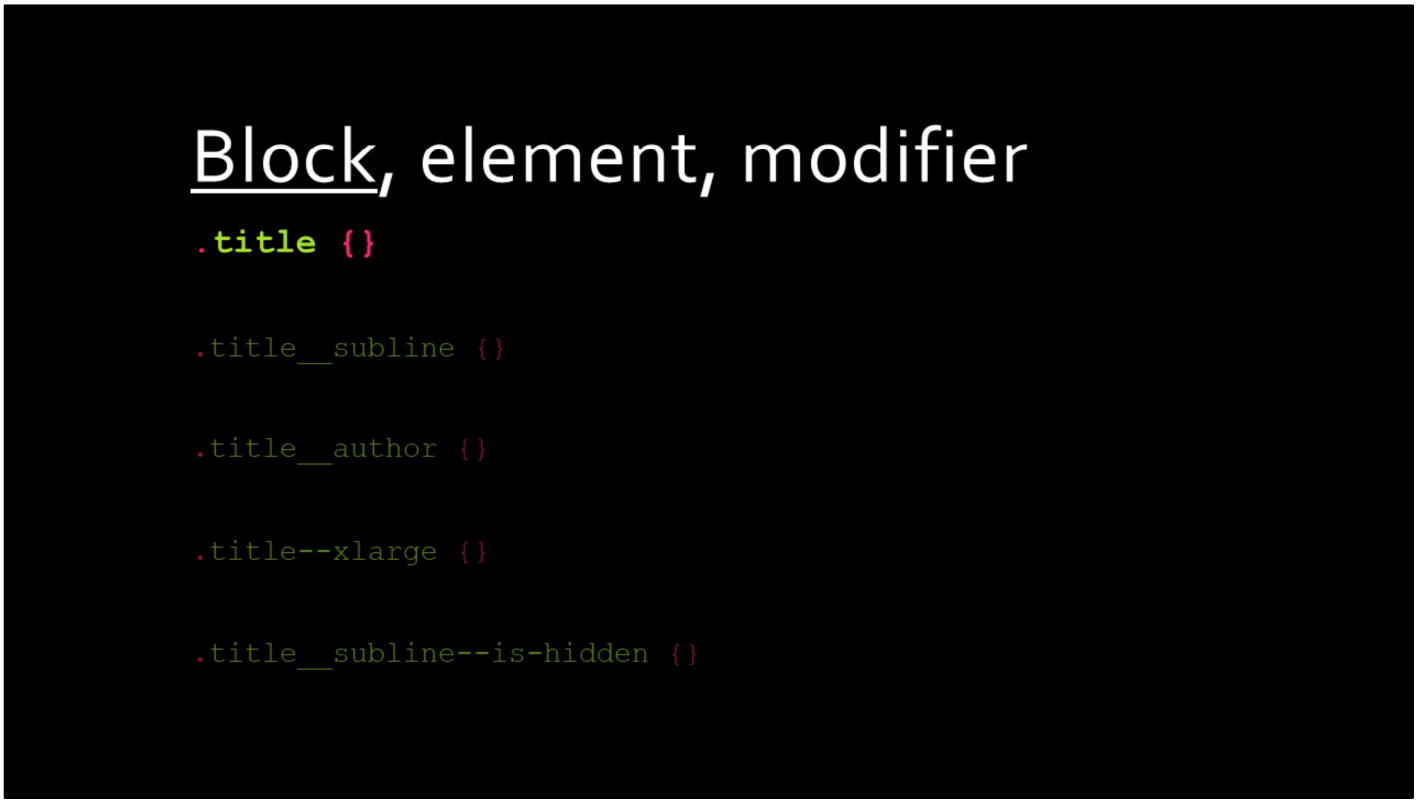
ISSUES ROUNDUP

Global CSS Scope – which problems could arise

- Issues with the layout of 3rd party widgets because of own selectors
- Increasing effort to check if styles don't break upon modification in growing projects
- Ever more specific selectors or usage of the „!important" modifier to make sure that rule overrides work (hard to understand and maintain)
- Almost every other programming language aims for a modular approach – or finds workarounds to mimic modularity, how to do this with CSS?

Block – Element – Modifier

- BEM is not a tool, its a methodology for a unified semantic model, which applies to HTML, Styles, Code and UX
- In the next minutes we'll concentrate on the naming convention side of things only
- Aims for easier readable CSS code, promotes reuse and improves maintainability
- How? By dividing parts of the user interface into independent blocks (the B in BEM)



Introducing: Block

- Functionally independent, reusable component – in our case, the title block
- Name of the Block describes its purpose, not its style or state
- A block in BEM could include elements, acting as their parent

Block, element, modifier

```
.title {}

.title__subline {}

.title__author {}

.title--xlarge {}

.title__subline--is-hidden {}
```

Introducing: Element

- An element is depending on a block as its parent, it couldn't exist on its own (subline, author)
- Name includes block name, separated through two underscores
- An element in the BEM hierarchy couldn't be a parent of an other element (this doesn't apply to the HTML structure, as we'll see later)

Block, element, modifier

```
.title {}

.title__subline {}

.title__author {}

.title--xlarge {}

.title__subline--is-hidden {}
```

Introducing: Modifier

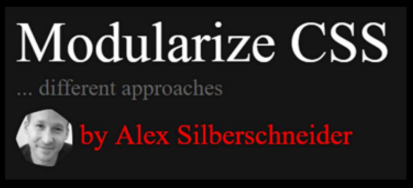
- Belongs to a block or an element
- Describes a certain state and/or a specific style, which constitute a deviation to the default state or style
- The modifier selector never exists on its own
- Just for information, these examples show the so called „double-dash" style syntax, there are other possibilities as well
- Until now we just saw the CSS, but there is already an idea forming in our head about how the HTML could look like

HTML for the selector samples

- Because of the structure of the CSS it‘s easy to conclude on how the HTML will look like and vice versa
- Modification of the structure of the HTML doesn‘t break what was laid forth as the Block – Element structure, selector matches will stay intact
- Example: move the author element into the subline element
- This also showcases that elements could be nested in the HTML as needed

HTML

```
<div class="title title--xlarge">
  Modularize CSS
  <div class="title__subline">
    … different approaches
  </div>
  <div class="title__author">
    <div class="avatar">
        <div class="avatar__thumb"></div>
    </div>
    by Alex Silberschneider
  </div>
</div>
```



HTML for the selector samples

- Other blocks could be inserted into this block as desired
- E.g. an avatar block, but this would bring its own selectors

No nesting

```
<div class="title__subline">
   Subline text
   <div class="title__subline__date">
     01.12.2017
   </div>
 </div>
```

Prevent nesting of Block – Element – Element in the BEM selector

- Most likely you would mix the HTML structure with the semantical meaning
- Makes it harder to maintain when you start to move elements in the HTML structure
- Most likely you'd want to introduce a new block instead

Example where Nesting is „OK“

```
.grid--is-editing .grid__cell {
  border: 1px solid black;
}
```

Certain kinds of nesting are OK but should be used sparingly

- In our example we nest a block selector with a certain modifier with the block element
- For such a use case this is the most effective way to have the rule apply

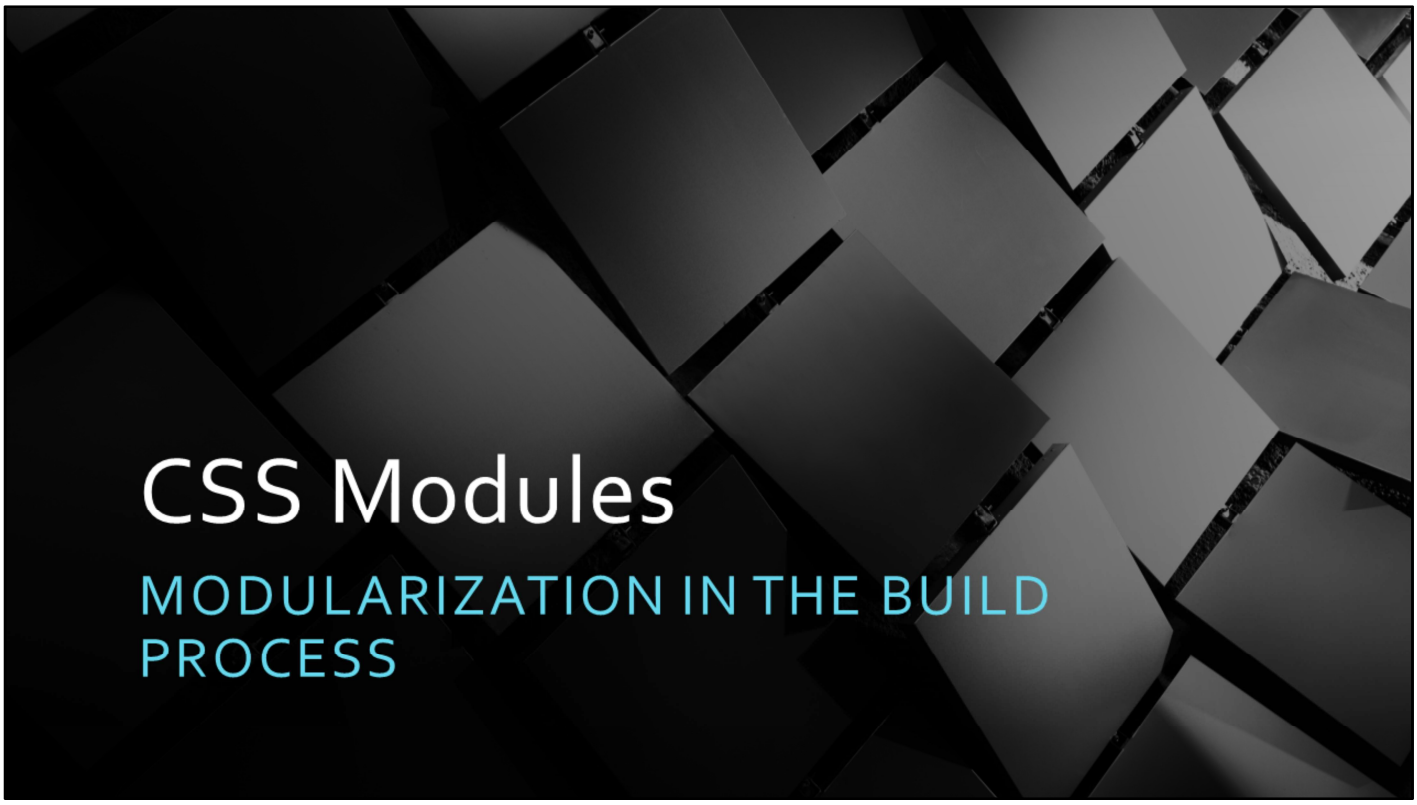
BEM benefits

BEM Benefits

- Precise semantics, relationship easy to recognize, simplifies communication in a team
- CSS is not directly coupled with HTML structurally
- More confidence when introducing changes, minimize side effects
- Easily locate selectors which need to be modified, no CSS specifity to worry about
- Could use this on little projects from the start as well

BEM Issues

- Many classes, long class names, everyone should/must adhere to the naming convention, could be hard sometimes ;)



CSS Modules Intro

- CSS Modules = roughly speaking, CSS files in which all class names are scoped locally instead of globally
- This isn't an official spec or a browser implementation, its a step in the build / bundling process through a tool lik Webpack or Browserify
- Build tool: Package application modules, tranformation and transpilation of source files (e.g. SASS to CSS, TypeScript to JavaScript etc.)
- To emulate a local scope with CSS modules the source HTML and CSS need to be modified in the build process

Initial code

```
<h1 class="bigtitle">
  A heading
</h1>

.bigtitle {
  color: red;
  font-size: 36px;
}
```

Initial example, both plain HTML, CSS

- We got to rewrite this to be able to use CSS modules

Instead of HTML, Javascript + Template string

```
import styles from "./styles.css";

element.innerHTML =
  `<h1 class="${styles.bigtitle}">
     A heading
   </h1>`;
```

Restructurizing the initial source files

- The compoments HTML template is now defined as an ES6 template string
- Styles will be „imported" from the CSS File
- For those of you familiar with ES6, this seems a little bit strange – how could we import module content from a CSS file??
- This is a little trick where CSS modules come into play, with Webpack the CSS modules loader parses the underlying CSS file and maps the individual classes to properties of the imported object
- Class names which did exist in the original CSS file will be replaced with unique string keys, these are available as values behind the properties of the imported object, e.g. "styles.bigtitle"

Resulting output

```
<h1 class="_styles__bigtitle_471156789">
  A heading
</h1>

._styles__bigtitle_471156789 {
  color: red;
  font-size: 3rem;
}
```

CSS Modules resulting output

- After the build, the result consists of HTML and CSS again
- The „bigtitle" Selector has been replaced through a new unique string
- CSS rules have been mapped to this new selector name
- This means that selector always have „local scope" with their templates
- In our case you can recognize the original selector name, this is complete configurable though

React

```
import React from 'react'
import styles from './styles.css'

export default class CoolTitle extends React.Component {
  render() {
    return (
       <h1 className={styles.bigtitle}>{this.props.text}</h1>
    )
  }
}
```

CSS Modules with React and JSX

- For those of you who already wondered why on earth we'd like to rewrite our markup to template strings
- Much more elegant with JSX templates, almost looks like the markup itself

Angular

```
import {Component} from '@angular/core';
import styles from './styles.css';

@Component({
    'selector': 'my-component',
    'template': `<h1 class="${styles.bigtitle}">
            Big title
          </h1>`
})
```

Since we talked about React, what about Angular?

- Well you COULD use webpack with the CSS Module loader with Angular (link to demo in appendix)
- Since Angular 2.x+ there is a built in mechanism for component based styling called „ViewEncapsulation"
- More on that later

CSS Modules benefits

CSS Modules benefits

- CSS selector names freely defineable, no side effect
- Styles are scoped locally per default, saves you many headaches
- Plain CSS could be written, plain CSS goes to the browser
- Makes sense in bigger projects – slightly overkill with little projects
- Only really makes sense if you already could use some kind of templating for your UI framework

CSS in JS

- Many different libraries, in general they use two different approaches
- #1 appraoch emission of modular CSS via Javascript, Injection into good old CSS style blocks
- #2 approach creation of inline style attributes on DOM elements with addtional JS event listeners to support pseudo states like :hover
- Most well known #1 approach: Styled-components (React, ReactNative)
- Most well known #2 aproach: Radium (React)
- Writing styles in the different frameworks is quite diverse
- Authoring approach #1: Write more or less plain CSS as ES6 template strings
- Authoring approach #2: Define CSS selectors and rules as JS objects, property names are selector names or rule names

Aphrodite

Intro Aphrodite

- Why Aphrodite: Chosen as an example b/c it is generally framework independent
- Also generates style tag, injects styles as needed
- Supports pseudo states, media queries, keyframe animations by generating „real CSS"

Aphrodite JS

```
import {StyleSheet, css} from
'aphrodite'


const styles = StyleSheet.create({
  bigtitle: {
    color: 'red',
    fontSize: '3rem',
    backgroundColor: 'transparent',
    height: '2em',
    ':hover': {
      color: 'white',
      backgroundColor: 'palevioletred'
    }
  }
});
```

```
const app =
document.getElementById('app');

app.innerHTML = `

  <h1
class="${css(styles.bigtitle)}">

    A heading again!

  </h1>`;


console.log(css(styles.bigtitle));

// "bigtitle_195i0wx"
```

A heading again!

Aphrodite JS Sample

- The first call to the „css" mehtod injects the styles into the header block
- Unuses declarations won't be injected

Aphrodite JS

```
import {StyleSheet, css} from
'aphrodite'


const styles = StyleSheet.create({
  bigtitle: {
    color: 'red',
    fontSize: '3rem',
    backgroundColor: 'transparent',
    height: '2em',
    ':hover': {
      color: 'white',
      backgroundColor: 'palevioletred'
    }
  }
});
```

```
console.log(css(styles.bigtitle));
// "bigtitle_195i0wx"


const app =
document.getElementById('app');

app.innerHTML = `

  <h1
class="${css(styles.bigtitle)}">

    A heading again!

  </h1>`;
```

A heading again!

Aphrodite JS Sample

- Hover style is a „real“ CSS declaration

Aphrodite CSS

```
<style type="text/css" data-aphrodite="">
.bigtitle_wnu5ld {
    background-color:red !important;
    color:blue !important;
    height:2em !important;
}
.bigtitle_wnu5ld:hover {
    color:white !important;
    background-color:black !important;
}
</style>
```

Aphrodite JS Sample – Output

- Used declarations were generated as CSS
- „!important" modificator is generated per default, the documentation describes the reasonas „This is intended to make integrating with a pre-existing codebase easier." – this could be turned off though

CSS in JS roundup

CSS in JS roundup

- Mostly realized via CSS injection, the final outcome is very similar to what CSS modules does
- Enables easy bundling of a compoments JS and CSS in the same file
- Many CSS in JS frameworks also support server side rendering out of the box
- „Preprocessing" has to be done / could be done in JavaScript instead of one of the usual preprocessors like LESS/SASS – finally with a „real programming language" now ;) – this also means that style writers aren't restricted to what is possible with LESS/SASS
- Syntax for writing styles feels unfamiliar at first, because it is always just a wrapper for real CSS
- In the appendix you can find a link to a github repository where someone has made it their task to list different CSS in JS implementations in a well arranged comparison

Shadow DOM – what's that?

- In functionality it's comparable to an IFRAME – it enables completely decoupled content within the rest of the page
- If you already had a look at web components, you surely have heard of it
- Shadow DOM enables the encapsulation of CSS and HTML without any further tools natively through the browser
- This is already being used in modern browser e.g. To encapsulate the HTML5 video player controls when using video tags

Simple example

HTML

```
<h1>Default Title</h1>
<div class="my-host"></div>
```

CSS

```
h1 {
  color: blue !important;
}
```

```
const myElement =
document.querySelector('.my-host');

const shadow =
myElement.attachShadow({
  mode: 'closed'
});
shadow.innerHTML = `
  <style>
    h1 {
      color: red;
    }
  </style>
  <h1>Title in Shadow DOM</h1>
`;
```

Shadow DOM example

- The current spec draft doesn't provide for any inheritance of styles into the shadow DOM part by its surrounding context (as opposing to the initial vo draft where this was still possible if explicitly enabled)
- As you can see in the example the shadow DOM part is not being affected by selectors of the surrounding HTML

How to apply styles

HTML

```
<h1>Default Title</h1>
<div class="my-host"></div>
```

CSS

```
h1 {
  color: blue !important;
}

html {
  --my-custom-h1-color: green;
}
```

```
let myElement =
document.querySelector('.my-host');

let shadow =
myElement.attachShadow({
  mode: 'closed'
});
shadow.innerHTML = `
  <style>
    h1 {
 color: var(--my-custom-h1-color,
red);
    }
  </style>
  <h1>Title in Shadow DOM</h1>
`;
```

Default Title

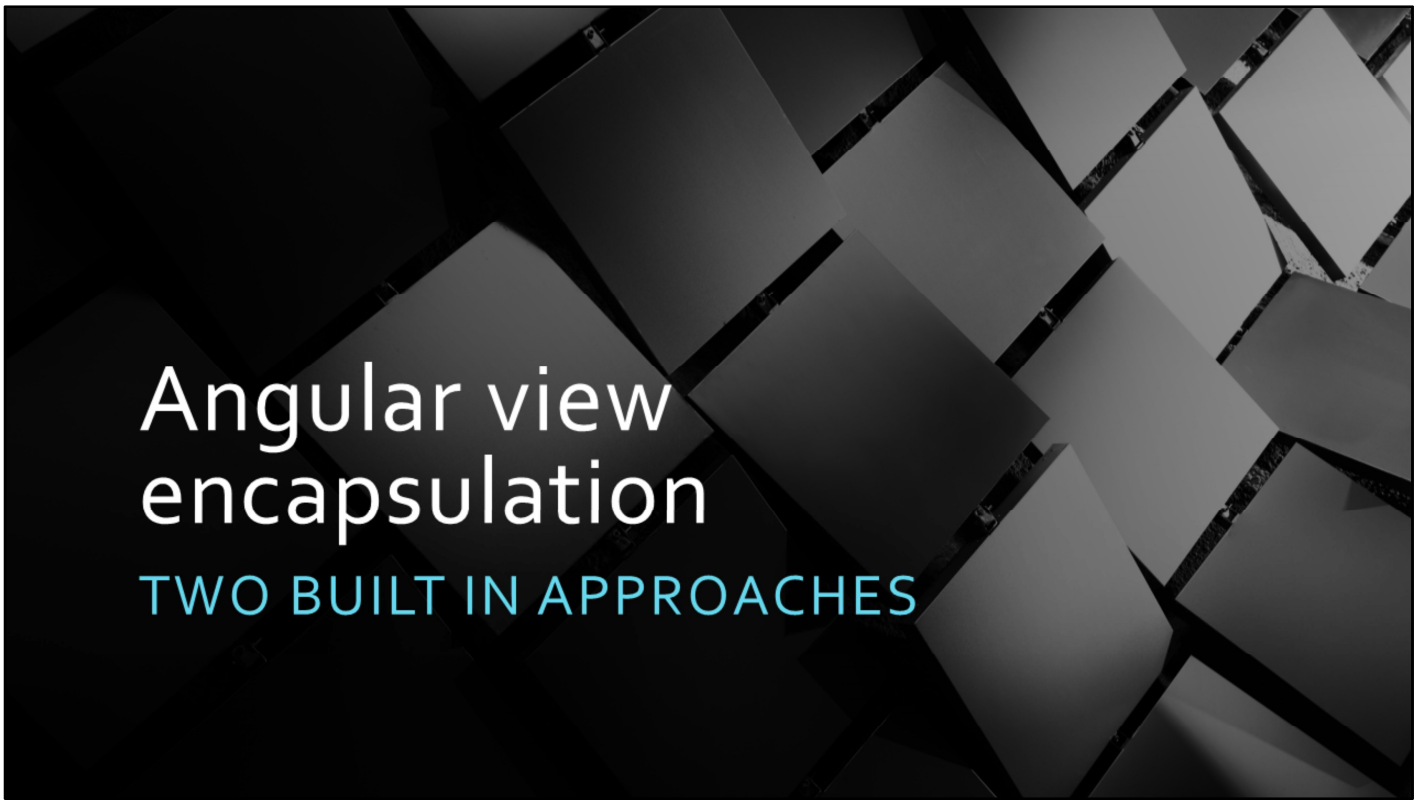
Title in Shadow DOM

How to apply theming to e.g. 3rd Party widgets using Shadow DOM

- CSS custom properties / CSS variables are accessible in Shadow DOM css
- 3rd Party widget must define which properties are available for theming

Shadow DOM roundup

- Enables real encapsulation of CSS
- Still in active development – spec could also still change!
- Browser support currently still at only 56% see http://caniuse.com/#feat=shadowdomv1, Chrome, Safari, Opera, NO Firefox (in dev), NO IE/Edge (under consideration)
- Polyfills slow, can't be recommended thus
- Could be used as a progressive enhancement, fallback to iframe injection - Example: Twitter timeline: tweets



Since Angular 2+ Angular includes a mechanism called view encapsulation, two approaches available

„Emulated" - default

```
import { Component,
ViewEncapsulation } from
'@angular/core';

@Component({

selector: 'app-first-component',

templateUrl: './first-
component.component.html',

styleUrls: ['./first-
component.component.css'],

encapsulation:
ViewEncapsulation.Emulated

})

export class
FirstComponentComponent {}
```

```
h1 {
    color: red;
}
```

First component H1

Another components H1

- Component declaration + css
- Could also leave out declaration of Emulated because it is the default value
- Won't affect other components

„Emulated“ – how does this work

```
<h1 _ngcontent-
c1="">
First component H1
</h1>

h1[_ngcontent-c1] {
color: red;
}
```

- Every element of component gets attribute
- Selector restrained to elements with matching attribute

„Native" – Shadow DOM

```
import { Component,
ViewEncapsulation } from
'@angular/core';

@Component({
selector: 'app-first-component',
templateUrl: './first-
component.component.html',
styleUrls: ['./first-
component.component.css'],
encapsulation:
ViewEncapsulation.Native
})
export class
FirstComponentComponent {}
```

- Use Shadow DOM by simply switching encapsulation mode

„Native" – how does this work

```
import { Component,
ViewEncapsulation

 } from '@angular/core';

@Component({

selector: 'app-root',

templateUrl:
'./app.component.html',

styleUrls:
['./app.component.css'],

encapsulation:
ViewEncapsulation.Native

})

export class AppComponent {}
```

```
▼<app-first-component> == $0
  ▼#shadow-root (open)
      <style>h1 {
          color: red;
      }</style>
      <style></style>
      <h1>
          First component H1
      </h1>
  </app-first-component>
```

First component H1

Another components H1

- Use Shadow DOM by simply switching encapsulation mode
- Generates component within Shadow DOM host
- As said earlier: Needs polyfills with certain browsers

- Also depends highly on the support with your primary UI framework
- In the end - mix and match approaches to suit your needs


Closing
Thank you for listening!

Appendix

Link CSS in general

- https://philipwalton.com/articles/side-effects-in-css/
- http://thesassway.com/intermediate/avoid-nested-selectors-for-more-modular-css
- http://www.standardista.com/css3/css-specificity/
- http://thesassway.com/beginner/the-inception-rule
- https://csswizardry.com/2012/11/code-smells-in-css/

Links BEM

- https://en.bem.info/
- https://css-tricks.com/bem-101/
- https://csswizardry.com/2013/01/mindbemding-getting-your-head-round-bem-syntax/
- https://www.smashingmagazine.com/2016/06/battling-bem-extended-edition-common-problems-and-how-to-avoid-them/
- https://www.sitepoint.com/bem-smacss-advice-from-developers/
- https://medium.com/@stowball/bemantic-dry-like-you-mean-it-133ea3843d98
- https://www.smashingmagazine.com/2014/07/bem-methodology-for-small-projects/
- https://mattstauffer.co/blog/organizing-css-oocss-smacss-and-bem/
- https://csswizardry.com/2015/03/more-transparent-ui-code-with-namespaces/

Links CSS Modules

- https://css-tricks.com/css-modules-part-1-need/ +2+3 (Attention: Breaking webpack changes in the tutorial! Ask me..)
- https://glenmaddern.com/articles/css-modules
- https://www.sitepoint.com/understanding-css-modules-methodology/
- https://github.com/nkbt/ng-modular
- https://vuejs.org/v2/guide/single-file-components.html
- https://github.com/joaogarin/css-modules-angular2

Links CSS in JS

- https://github.com/MicheleBertoli/css-in-js
- https://medium.freecodecamp.org/css-in-javascript-the-future-of-component-based-styling-70b161a79a32
- http://engineering.khanacademy.org/posts/aphrodite-inline-css.htm
- https://github.com/airbnb/javascript/tree/master/css-in-javascript
- https://www.styled-components.com/docs
- https://github.com/threepointone/glamor
- https://medium.com/@gajus/stop-using-css-in-javascript-for-web-development-fa32fb873dcc ;)

Links Shadow DOM

- http://caniuse.com/#feat=shadowdomv1
- https://css-tricks.com/playing-shadow-dom/
- https://frontend.namics.com/2014/05/27/web-components-shadow-dom/
- https://developers.google.com/web/fundamentals/architecture/building-components/shadowdom
- https://www.smashingmagazine.com/2016/12/styling-web-components-using-a-shared-style-sheet/
- https://www.polymer-project.org/2.0/start/first-element/step-5

Pens

- https://codepen.io/silberxander/