Modularise all the things
(now including CSS)

Hello, I’m Lewis Cowper both here and on the internet, I’m a Junior Front End Developer at Hitfox, here in Berlin, and we’re @HitFoxTechDev on Twitter if you want to give us a follow or tweet how much you’re enjoying this talk, or whatever.

Modularise all the things (now including CSS)

So! I’m here tonight to talk to you about modularising all the things, although specifically I’ll be talking about the principles behind css-modules, and taking you through a bit of code to help you understand what the project is about.

Modules, Components, Atoms…

Classes, Objects, Functions…

Containers, VMs, APIs…

No matter what level of our applications we look at, we see repeated attempts at creating more general pieces out of our system, to make it easier to work with.

Separation of Concerns

This part of system design and architecture really fascinates me, and as I've been trying to learn more web development, a big part of that becomes learning how and when to refactor into a more general function, or class, or object, or anything really.

So in order to try and explain that, I thought I might turn to philosophy.

> [Reduction is...] one of the most used and abused terms in the philosophical lexicon
>
> — The Oxford Companion to Philosophy

Reductionism refers to several related but different philosophical positions regarding the connections between phenomena, or theories, "reducing" one to another, usually considered "simpler" or more "basic"

Phenomena that can be explained completely in terms of relations between other more fundamental phenomena, are called epiphenomena.

While I was procrastinating writing this talk, reading about Reductionism I kept seeing things being described as phenomena, and epiphenomena. It really made me think about how we think about our application, and behaviour both in the overall system sense, and also in the behaviour of developers and designers working within our applications.

> **Often there is an implication that the epiphenomenon exerts no causal agency on the fundamental phenomena that explain it**
>
> — Wikipedia

So, although phenomena which can be explained using only the relationship between other phenomena are called epiphenomena, it also often holds that epiphenomena don't affect the fundamental phenomena that they rely on.

Phenomena & Epiphenomena

I feel like we have a few phenomena in our daily lives as people working within the web. Your CSS, whether preprocessed or not, your markup, and by extension your class/id structure, and the browser that the user is viewing your page in, and all the environmental issues that extend that, like extensions, or the device.

That's not to mention things like ad networks, or external web content being loaded in to your web page that you aren't in full control of.

I believe that the whole construction of the DOM, the CSSOM, and the visuals of the webpage itself are the epiphenomena of all those phenomena. We *should* be able to go from one to the other if we know all the fundamental phenomena, thereby meaning that they are all epiphenomena.

When we don't have the full knowledge of all the CSS on the page, because we're using some CSS that depends on a specific hierarchy in the markup, and we're just plain not quite representing our ideas in the right combination of CSS and HTML, it means that we're highly likely to be bitten by global scope in CSS. This is usually the point where, depending on how much time we have, we slap an !important declaration in, we refactor our idea into a class that makes a bit more sense, and apply it more specifically, or we figure out exactly why we're not seeing the right result, and then modify as little code as possible to leave us with the intended result.

As I'm comparably new to to learning about CSS and how to represent all the different moving parts that go into a modern website in the simplest way, I've been looking for good ways to think about all the things that CSS can, and regularly does, do, that messes with my expectations.

BEM, SMACSS, ITCSS, SUIT

I've heard about BEM, SMACSS, ITCSS, and other naming or architecting methods to reduce the cognitive overhead surrounding a large CSS project. However these are all just naming conventions or things that describe what order to declare things in, and that becomes difficult to enforce. All it takes is one person not reading the README, or misunderstanding what layer of the inverted triangle they should put their style in, or a lax code review, and then all of a sudden, regardless of how many acronyms you use to organise your CSS, it's all gone a bit wrong.

In my limited experience, CSS has problems when multiple people of varying amounts of knowledge try and collaborate on a project. Either the wizard or witch of CSS in the group becomes a bottleneck very quickly, and winds up being unable to magic up a lot of really beautiful CSS because all their day is spent reviewing and fixing other people's CSS. Or people get left behind and can't understand what that witch or wizard did because it all moved too quickly for them.

Keep your HTML's CSS dependencies in code.

Not in your head

Either way, a lot of time is invested making the CSS as clear as possible, and at the end of the day, hopefully the team can feel like they levelled up their knowledge, without the witch/wizard feeling under-utilised. But unfortunately, up until very recently, there was no way to solve these problems. To create components that are logically clear and simple to understand, without worrying about running into one of the problems that can occur when developing CSS, following the principles of reduction, to reduce our cognitive load and start keeping your HTML's CSS dependencies in code instead of in your head.

Styles should be scoped to a single component

All CSS Selectors are global

We need a convention for ensuring globally-unique selectors

So there are some goals of the internal architecture of “good” CSS.

Read the slide

This is a problem that BEM tries to solve by namespacing your class names with it’s block element modifier syntax, but it’s little more than a convention, and Naming Things is Hard.

Loading CSS into your JS

One of the fundamental features of the Webpack loader (which is also core to JSPM and easily possible with Browserify) is the ability to explicitly describe each file's dependencies regardless of the type of source file.

This means that you can specify your CSS as a dependency, and it is guaranteed to be present, much like anything other dependency. But with great power comes great responsibility.

This is where this new spec comes in.

CSS Variables => JS

By treating the CSS as a dependency of our JavaScript, we gain access to a fundamental new power. Passing variables through from our CSS into our JavaScript. Let's talk about why that's a nice idea.

```
require('./my-component.css');
```

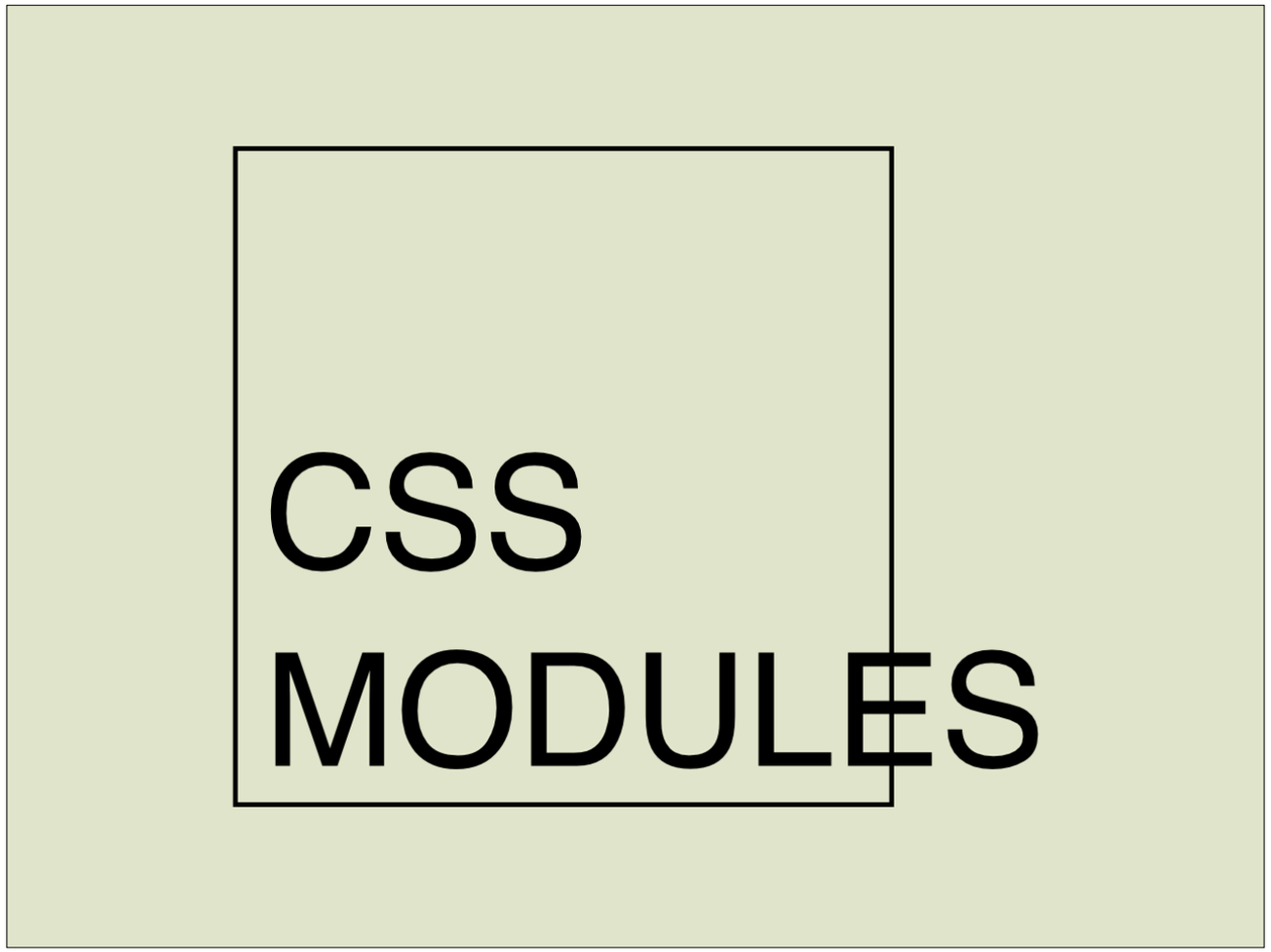
```
var styles = require('./my-component-css');

elem.addClass(styles.elemClass);
```

Instead of the one up the top, where we load CSS as a side effect, we can now (for example) import an object that has some dynamically generated class names. But we get something in our JavaScript that we can pass around like any other object.

- **Read the Spec**
- **Try it out**

In order to help out, you can read the spec, and ask your loader of choice to support the format. Webpack and JSPM already support it natively, and Browserify has plugins that support it.

In short, Interoperable CSS is the specification that describes the mechanism by which the exported symbols are passed around before they get to your JavaScript component. I'm not going to go any further into the Interoperable CSS stuff just now, we should get back to css-modules.



CSS-Modules is a new project that builds upon the work that people wanting to control styling in JavaScript are doing, but with the express purpose of not sacrificing all the great things about CSS.

No more global scope pollution

Everything inside a CSS module is local by default. Each file gets compiled separately so your selectors can be simple and generic. The compilation will take care of the naming of things, and we can take advantage of Interoperable CSS's really nice import and export pseudo selectors.

With SUIT/BEM naming

```
.Button { /* all styles for Normal */ }

.Button--disabled { /* overrides for
Disabled */ }

.Button--error { /* overrides for Error */ }

.Button--in-progress { /* overrides for In
Progress */ }


<button class="Button Button--in-
progress">Processing...</button>
```

With this SUIT/BEM style naming we avoid nesting our selectors, but we still get variants of the original Button class. We start Button with a capital letter to hopefully avoid clashes with any external dependencies, or our own previous styles. Plus we've also got the double dash modifier syntax, so that we know that the base class needs to be applied.

However the only thing that provides any of these safety nets is a naming convention.

With CSS Modules

```
/* components/submit-button.css */

.normal { /* all styles for Normal */ }

.disabled { /* all styles for Disabled */ }

.error { /* all styles for Error */ }

.inProgress { /* all styles for In Progress */ }


/* components/submit-button.js */

import styles from './submit-button.css';

buttonElem.outerHTML = `<button class=$
{styles.normal}> Submit</button>`
```

Because each file is compiled separately, you can use the filename to namespace your CSS, in the same way that you would use a web component, a React component, or any javascript based modular UI builder thing. There's probably been a few more ways of doing modular UI components released this week that I haven't heard of.

Of note here is the fact that all styles for the selector go inside the selector. This is different from how we do it with BEM, because we don't have overrides inside our selectors. Let's talk about what happens with css-modules.

Composition

We use the wonderful: Composition!

Composition is how you represent shared styles within css modules. Let’s look at some examples.

```
.common {
  /* all the common styles you want */
}
.normal {
  composes: common;
  /* anything that only applies to Normal
*/
}
.disabled {
  composes: common;
  /* anything that only applies to
Disabled */
}
```

The composes keyword says that .normal includes all the styles from .common, much like the @extends keyword in Sass. But while Sass rewrites your CSS selectors to make that happen, CSS Modules changes which classes are exported to JavaScript.

```
/* BEM Style */
innerHTML = `<button class=
"Button Button--in-progress">`

/* CSS Modules */
innerHTML = `<button class=
"${styles.inProgress}">`
```

So with BEM, we make two class declarations, but with css modules, we can use only one style, and the namespace is taken care of by the componentised nature of our application.

Inter-file co-operation and sharing

Here we see another really cool thing about the composes keyword.

```
/* colors.css */
.primary {
  color: #720;
}
.secondary {
  color: #777;
}
/* other helper classes... */
```

If we have the following css file in the shared/ directory, or you could assume that we have a brand variables file or something, what happens if we want to use a style from that in another file?

```
/* submit-button.css */
.common {
/* font-sizes, padding, border-radius */
}
.normal {
  composes: common;
  composes: primary from "../
shared/colors.css";
}
```

Using composition, we are able to reach into a totally general file like colors.css and reference the one class we want using its local name.

That’s tremendously powerful, and reads well to me at least.

Single Responsibility Principle

So composition is like a superpower because it lets you describe what an element is, not what styles make it up. It's a different way of mapping elements to rules. Let's have a look at some CSS.

```
.some_element {
  font-size: 1.5rem;
  color: rgba(0,0,0,0);
  padding: 0.5rem;
  box-shadow: 0 0 4px -2px;
}
```

This is just, this element, has these styles. Not too difficult, right? But everything here is specified in full, despite the fact that we'll probably want to reuse these styles elsewhere. So let's use SASS variables to break this apart.

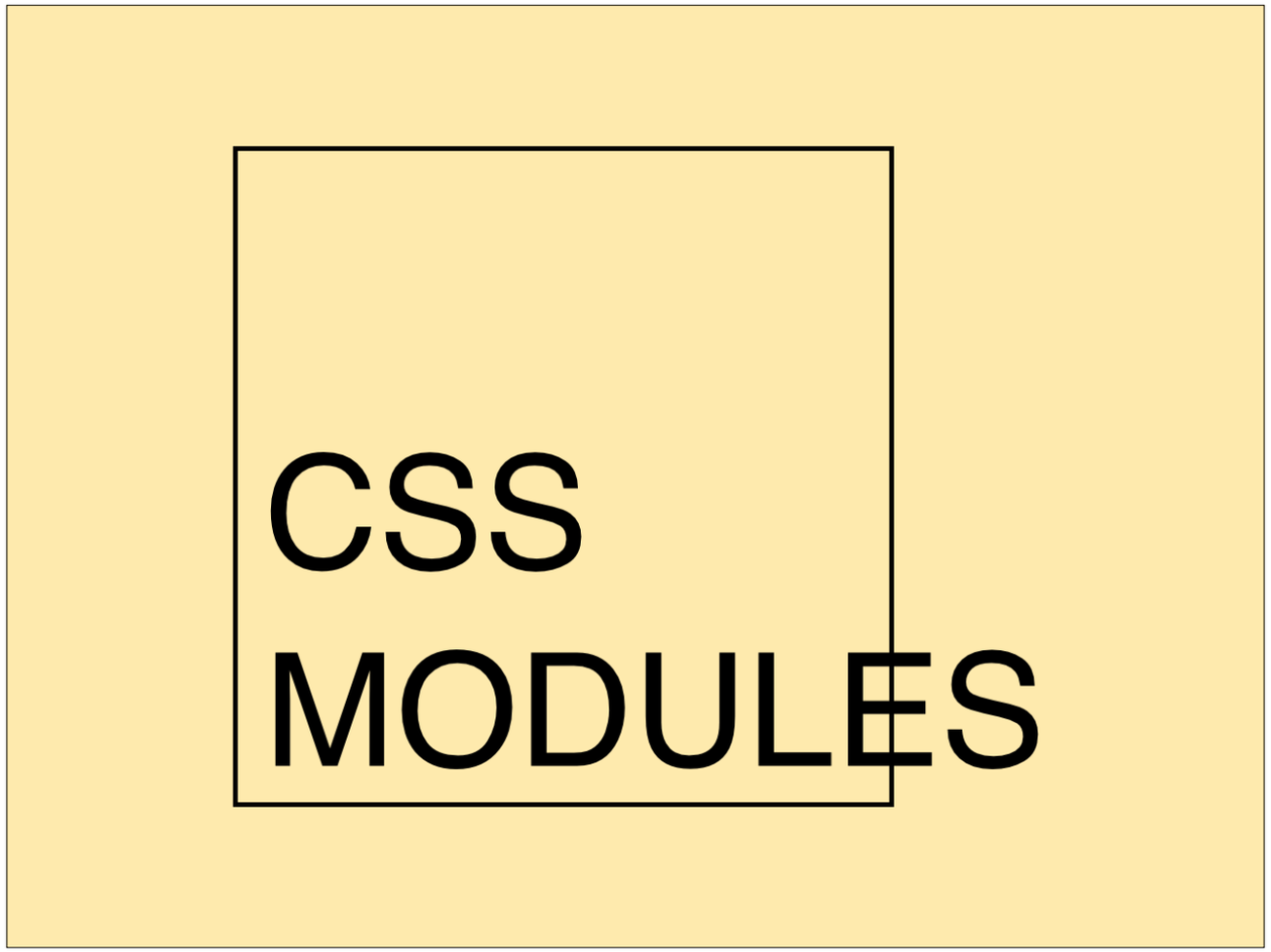
```
$large-font-size: 1.5rem;
$dark-text: rgba(0,0,0,0);
$padding-normal: 0.5rem;
@mixin subtle-shadow {
  box-shadow: 0 0 4px -2px;
}

.some_element {
  @include subtle-shadow;
  font-size: $large-font-size;
  color: $dark-text;
  padding: $padding-normal;
}
```

This is an improvement, but we've only extracted half of most of the lines. The fact that $large-font-size is for typography and $padding-normal is for layout is merely expressed by the name, not enforced anywhere. When the value of a declaration like box-shadow doesn't lend itself to being a variable, we have to use a @mixin or @extends.

```
.element {
  composes: large from "./typography.css";
  composes: dark-text from "./colors.css";
  composes: padding-all-medium from
            "./layout.css";
  composes: subtle-shadow from "./effect.css";
}
```

By using composition, we can declare our component in terms of reusable parts.

The format naturally lends itself to having lots of single-purpose files, using the file system to delineate styles of different purposes rather than namespacing



In short, CSS Modules may not be a tool that you immediately go away and convert all your applications to, but it’s a step in the right direction toward making CSS less of a stressful thing where few people really understand it, and mostly people are just hacking their way toward the visual layout they want. It allows newcomers to look at your code and understand what is coming from where, and hopefully why. It removes convention and tribal knowledge from understanding your CSS architecture, and is generally awesome.

> **Fear leads to anger;
> anger leads to hate;
> hate leads to suffering.**
>
> **— Developers discussing CSS**

Remember this the next time you see someone declaring CSS is dead.

Then think about how that fear affects people who want to write CSS. They get angry, they hate it, and they suffer for it, and will refuse to write CSS or learn more about it. They inflict their anger filled viewpoints on the world, and I think a good way of fighting that is to use tools that help you write your CSS in a better way.

In the end, for those of you who've seen Star Wars, CSS, HTML and JavaScript, along with all the web technologies we use on a day to day basis are basically the Force.

With the right instructions, good mentorship, and a lot of discipline, you can do some truly remarkable things, scale your CSS architecture without problems, and focus your energy on the next big thing.

However, if all you have is convention, it can be easy to let an !important slip in, and then adding some CSS onto the end of a file becomes a reasonable hack, and then you start running into more problems, and more hacks.

Hopefully, the more our tooling can help, the more likely we are to be able to take steps from the outset that make it so much easier to be a Jedi Knight, and use our powers for good.

In short, modularise things, and create phenomena instead of relying on epiphenomena.

Thanks very much. Tweet me questions or comments, or find me later. :)