React: CSS in JS

Christopher “vjeux” Chedeau

I’m Christopher Chedeau, working at Facebook in the front-end infrastructure team and among other things helping build React

Plan

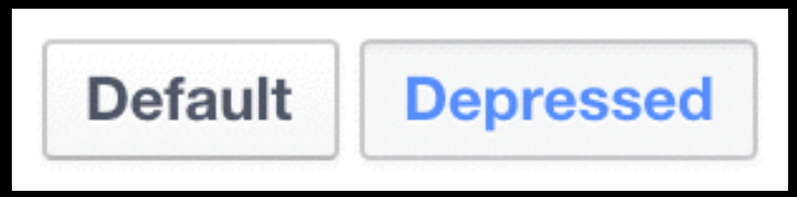
- Problems with CSS at scale

1. Global Namespace
2. Dependencies
3. Dead Code Elimination
4. Minification
5. Sharing Constants
6. Non-deterministic Resolution
7. Isolation

Before we get to the crazy JS part, I'm going to go over all the issues we've been facing when trying to use CSS at scale and how we worked around them.

When I'm saying at scale, it means in a codebase with hundreds of developers that are committing code everyday and where most of them are not front-end developers

Let’s build a button




```
/* button.css */

.button {
  background: #f6f7f8 url(/images/button/background.png) repeat-x;
  border: 1px solid #cdced0;
  border-radius: 2px;
  box-shadow: 0 1px 1px rgba(0, 0, 0, 0.05);
}

.button-depressed {
  background-color: #4e69a2;
  border-color: #c6c7ca;
  color: #5890ff;
}
```

During the entire talk we’re going to build a button to illustrate the issues. We got a mock from a designer for a button that has a normal state and a depressed one. As a web developer, we start writing some CSS

1 - Global Namespace

```
/* button.css */

.button {
```

Globals!

```
.button-depressed {
```

But … it turns out that we just introduced two global variables!

w3schools.com

JavaScript Best Practices

Avoid Global Variables

Avoid using global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

Use local variables instead, and learn how to use closures.

It is really crazy to me that the best practices in CSS is still to use global variables. We've learned in JS for a long time that globals are bad.

If you look at w3schools, my favorite website to learn JS, the first point of the best practice guide clearly says "Avoid Global Variables". To make sure you don't use global variables, they write it twice!

We've learned to use local variables, self invoking functions, modules to deal with globals

Bootstrap

.active .affix .alert .alert-danger .alert-dismissable .alert-dismissible .alert-info .alert-link .alert-success .alert-warning .arrow .badge .bg-danger .bg-info .bg-primary .bg-success .bg-warning .blockquote-reverse .bottom .bottom-left .bottom-right .breadcrumb .btn .btn-block .btn-danger .btn-default .btn-group .btn-group-justified .btn-group-lg .btn-group-sm .btn-group-vertical .btn-group-xs .btn-info .btn-lg .btn-link .btn-primary .btn-sm .btn-success .btn-toolbar .btn-warning .btn-xs .caption .caret .carousel .carousel-caption .carousel-control .carousel-indicators .carousel-inner .center-block .checkbox .checkbox-inline .clearfix .close .collapse .collapsing .container .container-fluid .control-label .danger .disabled .divider .dl-horizontal .dropdown .dropdown-backdrop .dropdown-header .dropdown-menu .dropdown-menu-left .dropdown-menu-right .dropdown-toggle .dropup .embed-responsive .embed-responsive-16by9 .embed-responsive-4by3 .embed-responsive-item .fade .focus .form-cont ol-static .form-group .form-group-lg .form-group-sm .form .h4 .h5 .h6 .has-error .has-feedback .has-success .has den-lg .hidden-md .hidden-print .hidden-sm .hidden- on-prev .img-circle .img-responsive .img-rounded .i .input-group .input-group-addon .input-group-btn p-sm .input-lg .input-sm .invisible .item .jumbotron .label .label-danger .label-default .label-info .label-primary .label-success .label-warning .lead .left .list-group .list-group-item .list-group-item-danger .list-group-item-heading .list-group-item-info .list-group-item-success .list-group-item-text .list-group-item-warning .list-inline .list-unstyled .mark .media .media-body .media-bottom .media-heading .media-left .media-list .media-middle .media-right .modal .modal-backdrop .modal-body .modal-content .modal-dialog .modal-footer .modal-header .modal-lg .modal-open .nav-pills .navbar- coll bar- head .navbar-inverse .navbar-left .navbar-link .navbar-nav .navbar-right .navbar-static-top .navbar-text .navbar-toggle .next .open .page-header .pager .pagination .pagination-lg .pagination-sm .panel .panel-body .panel-collapse .panel-danger .panel-default .panel-footer .panel-group .panel-heading .panel-info .panel-primary .panel-success .panel-title .panel-

>600 globals

Yet, we still use global variables everywhere in CSS-land. For example Bootstrap introduces a whooping 600 global variables :(

CSS Extension

```
/* button.css */
.button/container {
```

At Facebook, we've ran into so many issues with name conflicts that we had to do something about it. We extended the CSS language to allow a different way to define a class name. If you put a / in the name, it's now going to be a local variable

Local by Default

```
/* button.css */

.button/container {
```

```
/* dropdown.css */

.dropdown/container .button/container {
```

Does not build!

The way it's working is that `button/container` can only be used in the file called `button.css`. If you try to use it outside, it is not going to build.

Explicit Export

```
/* button.css */

.button/container/public {
```

```
/* dropdown.css */

.dropdown/container .button/container/public {
```

Does build!

But, this is sometime a valid use case. To make it work, you can append `/public` at the end of the name and now the variable is exported. It is now explicit what variables are global.

Note that this is probably not the best way to solve the issue but this is the one that we came up with.

Callsite

```
/* button.css */
.button/container {
```

```
/* button.js */
<div className={cx('button/container')}>
```

Since `button/container` is not a valid class name, we need to change the call site and make it go through a function that we call cx (class extension).

2 - Dependencies

```
/* button.css */

.button/container {
```

```
/* button.js */
requireCSS('button');
<div className={cx('button/container')}>
```

We've solved the global variable issue but we still have a lot of work to do. We're past the way where we can bundle all our CSS into a single file and have to split it into many files and therefore deal with dependencies.

For a long time we asked the developer to call requireCSS with the file you need. Unfortunately there is a very pernicious side effect with CSS which is that if some other file already required the CSS and you forget to, it's still going to work.

2 - Dependencies

```
/* button.css */
.button/container {
```

```
/* button.js */
requireCSS('button');
<div className={cx('button/container')}>
```

But, now that we have `cx`, if we can make sure that it is statically analyzable, then we can automatically inject the `requireCSS` call. And, at the same time, solving this dreadful issue once and for all.

3 - Dead Code Elimination

```
/* button.css */
.button/container {
```

```
/* button.js */
<div className={cx('button/container')}>
```

grep for `button/container`

Since cx is the only way to generate the name, we can also solve the hardest problem of CSS: “how do you remove dead code?”

If you have a rule `.button/container`, then just search for `button/container` in your codebase and you’ll find all the call sites. If there are none left, then you can kill it!

4 - Minification

```
/* button.css */
._f8z {
```

```
/* button.js */
<div className={'_f8z'}>
```

One side benefit is that we can minify all the class names and send both the JS and CSS a bit faster to users.

This also ensures that all the developers are using cx since they cannot guess that name :)

4 - Sharing Constants

```css
/* button.css */
.button/container {
  /* Keep in sync with button.js
  padding: 5px;
```

```js
/* button.js */
// Keep in sync with button.css
var buttonPadding = 5;
```

Sharing constants between CSS and JS is not ideal but there are unfortunately many real world use cases where you need to do it. For the longest time we've been using comments to solve this issue.

Unfortunately, it doesn't really scale. You can change the file name and not the comment, or one comment is applied to two blocks of code and you only update one, or the developer just ignores it …

“CSS Variables”

```
/* button.css */
.button/container {
  padding: var(button-padding);
```

```
/* button.js */
var buttonPadding = cssVar('button-padding');
```

```
/* CSSVar.php */
$CSSVar = array(
  'button-padding' => 5,
);
```

To fix it, we’ve borrowed the var syntax specification of CSS, and exposed a cssVar function to JS.

To generate those, because Facebook is still mainly PHP driven, we’re doing it from PHP :)

Plan

- Problems with CSS at scale

1. Global Namespace
2. Dependencies
3. Dead Code Elimination
4. Minification
5. Sharing Constants

"Solved"

6. Non-deterministic Resolution
7. Isolation

We managed to solve many of the issues we faced with CSS. But, we're really not using stock CSS anymore. We had to extend CSS and write **a lot** tooling for it.

Also, there are still problems we have no idea how to fix

6 - Non-deterministic Resolution

```css
/* button.css */

.button/container/public {
  background-color: white;
}

/* overlay-button.css */

.overlay-button/container {
  background-color: black;
}
```

```jsx
/* overlay-button.js */

<div className={cx(
  'button/container/public',
  'overlay-button/container'
)} />
```

Our designers came with a new request, having a button that needs to look fine on an overlay. What it means for us is that it has a black background instead white.

We use the `button/container` class we made before. in this case we agreed that it was a good idea to make it public.

6 - Non-deterministic Resolution

```
/* overlay-button.css */

.overlay-button/container {
  background-color: black;
}

/* button.css */

.button/container/public {
  background-color: white;
}
```

```
/* overlay-button.js */

<div className={cx(
  'button/container/public',
  'overlay-button/container'
)} />
```

CSS was designed with a single file in mind. The way it works is that if two rules have the same “specificity”, then the last one in the file wins.

But this is a nightmare when you are bundling files and loading them asynchronously.

6 - Non-deterministic Resolution

```
/* overlay-button.css */

.overlay-button/container {
  background-color: black;
}

/* button.css */

.button/container/public {
  background-color: white;
}
```

```
/* overlay-button.js */

<div className={cx(
  'button/container/public',
  'overlay-button/container'
)} />
```

This causes bugs where you have to first go to one page, then go to another page that dynamically loads CSS before you can see the bug. But if you land on the last page directly it's working fine. Good luck trying to get a repro!

6 - Non-deterministic Resolution

```
/* overlay-button.css */
div.overlay-button/container {
  background-color: black;
}

/* button.css */

.button/container/public {
  background-color: white;
}
```

```
/* overlay-button.js */
<div className={cx(
  'button/container/public',
  'overlay-button/container'
)} />
```

The most popular way to workaround this issue is to increase the specificity of the rule that conflicts but this is super brittle.

We’re in a situation where we have a sword of Damocles above our head. We have no idea when we’re going to get the next issue, but we do know it’s going to happen and we can’t do anything about it :(

7 - Breaking Isolation

```css
/* product.css */

.product/button > div {
  /* override everything! */
}
```

```jsx
/* product.js */

<div className={cx('product/button')}>
  <Button />
</div>
```

We've got a team dedicated to build core components such as buttons, dropdowns, menus, images… They spend a huge amount of time designing a very good API that supports all the use cases.

Ideally, when a designer/engineer wants to use a variant that's not yet supported (eg: make the text red), they should talk to the maintainer of the component to figure out what's the best way to go forward.

7 - Breaking Isolation

```
/* product.css */

.product/button > div {
  /* override everything! */
}
```

```
/* product.js */

<div className={cx('product/button')}>
  <Button />
</div>
```

However, they have the ability to modify the style of the internals via selectors. The override looks like regular CSS, so it's often not being caught by code review. It's also nearly impossible to write lint rules against it.

When this code gets checked in, it puts the maintainer of the component in a very bad spot because when he changes the internals of the component, she is going to break all those call sites. It makes you feel fearful of changing code, which is very bad.

Plan

- Problems with CSS at scale
    1. Global Namespace
    2. Dependencies
    3. Dead Code Elimination
    4. Minification
    5. Sharing Constants
    6. Non-deterministic Resolution
    7. Isolation

Unsolved

So at this point, those two problems are unsolved and are still triggering recurrent bugs that we don’t really know how to prevent :(

CSS in JS

The moment you've all been waiting for

We're already at 3/4 of the talk and I haven't yet talked about JS…

If I just started by introducing CSS in JS, you would probably have just dismissed it as me being crazy. It's super important for you to have an idea of all the hacks we had to do on-top of CSS to just make it work.

Let’s build a button

```
/* button.js */

var styles = {
  container: {
    background: '#f6f7f8 url(/images/button/background.png) repeat-x',
    border: '1px solid #cdced0',
    borderRadius: 2,
    boxShadow: '0 1px 1px rgba(0, 0, 0, 0.05)',
  },
  depressed: {
    backgroundColor: '#4e69a2',
    borderColor: '#c6c7ca',
    color: '#5890ff',
  },
};
```

The first step we need to do is to translate the CSS rules into JS.

Turns out that it’s pretty easy

Differences

```
/* button.js */

var styles = {
  container: {
    background: '#f6f7f8 url(/images/button/background.png) repeat-x',
    border: '1px solid #cdced0',
    borderRadius: 2,
    boxShadow: '0 1px 1px rgba(0, 0, 0, 0.05)',
  },
  depressed: {
    backgroundColor: '#4e69a2',
    borderColor: '#c6c7ca',
    color: '#5890ff',
  },
};
```

You have to quote values (React automatically adds ‘px’ so you can just use numbers), replace semi-columns by commas and use camelCase.

Before we go to the next slide, I want you to take a moment and forget everything you know about web development.

Keep an open mind

Inline Styles!!1!

```
/* button.js */

var styles = {
  container: {
    background: '#f6f7f8 url(/images/button/background.png) repeat-x',
    border: '1px solid #cdced0',
    borderRadius: 2,
    boxShadow: '0 1px 1px rgba(0, 0, 0, 0.05)',
  },
  depressed: {
    backgroundColor: '#4e69a2',
    borderColor: '#c6c7ca',
    color: '#5890ff',
  },
};

<div style={styles.container}>
```

We’re going to use inline styles to render the styles.

Inline Styles

```
<div style={styles.container}>
```

It turns out that in this context, inline styles are not so bad.

First, we're not writing the styles "inline", we give a reference to a rule that's somewhere else in the file.

Second, style is actually a much better name than class. You want to "style" the element, not "class" it.

Finally, this is not applying the style directly, this is using React virtual DOM and is being diff-ed the same way elements are.

Plan

- Problems with CSS at scale

1. Global Namespace

All your styles are local JS variables and you can export them if you want to

2. Dependencies

You can use a module system like CommonJS/AMD

3. Dead Code Elimination

Most styles are local variables that linters/minifier can remove

4. Minification

Use Closure Compiler or Uglifyjs or …

5. Sharing Constants

Everything is JS

6. Non-deterministic Resolution
7. Isolation

Solved without hacks

It turns out that all the first 5 problems are super boring when we're in JS world.

Over the years, we've developed tools to solve all of them elegantly.

Yet, they are still super-hard in CSS :(

Conditionals

```
/* button.js */

propTypes: {
  isDepressed: React.PropTypes.bool,
},
```

```
<div style={m(
  styles.container,
  this.props.isDepressed && styles.depressed
)} />
```

To fix the two last to points, we have to do a bit more work.

We want to add the depressed style only if the button actually is. To do that, we define a new attribute `isDepressed` on the object via React `propTypes`.

Then, we're going to use a simple JavaScript function that just merges all the objects from last to first and ignores falsy values. No more errors because of packaging.

Conditionals

```
function m() {
  var res = {};
  for (var i = 0; i < arguments.length; ++i) {
    if (arguments[i]) {
      Object.assign(res, arguments[i]);
    }
  }
  return res;
}
```

The `m` function in this example is really simple and powerful, but you don't have to use this one. You can use any JS functions that eventually returns a JS object with style attributes.

Customization

```
/* button.js */

propTypes: {
  isDepressed: React.PropTypes.bool,
  style: React.PropTypes.object,
},
```

```
<div style={m(
  styles.container,
  this.props.isDepressed && styles.depressed,
  this.props.style
)} />
```

In order to get feature parity with CSS, we need to let the call site be able to change the style of the element, —if we want to—.

Turns out that this is really simple, you take an object as props and you merge it with the styles.

Customization

```
/* button.js */

propTypes: {
  isDepressed: React.PropTypes.bool,
  style: React.PropTypes.object,
},
```

```
<div style={m(
  styles.container,
  this.props.style,
  this.props.isDepressed && styles.depressed
)} />
```

And we have total control in how the user defined style is going to be applied, we can make sure that it overrides the base style but not the depressed style.

Customization

```
/* button.js */

propTypes: {
  isDepressed: React.PropTypes.bool,
  color: React.PropTypes.string,
},
```

```
<div style={m(
  styles.container,
  this.props.color && {color: this.props.color},
  this.props.isDepressed && styles.depressed
)} />
```

So far we've let the call site override any style, but we can restrict what styles can be overridden. For example, we can only let the color be changed.

Plan

- Problems with CSS at scale

1. Global Namespace
2. Dependencies
3. Dead Code Elimination
4. Minification
5. Sharing Constants
6. Non-deterministic Resolution
7. Isolation

Solved without hacks

It turns out that if you write your styles in JS, a large class of really hard problems with CSS just disappear instantly.

Conclusion

Christopher “vjeux” Chedeau

My goal with this talk is not to convince you that you should drop CSS and use JS instead.

I want to cast light on fundamental problems with CSS that no one is talking about or trying to solve. Sure there are many libraries on CSS like Less, Sass… but none of them try to address the 7 points I highlighted.

CSS is also not the only part of web that has deep flaws. With React, we tried to solve some that the DOM has but there’s plenty more.

I want you to quit the room and think about all the hard problems we’re facing when building on the web, talk about them and maybe even fix them :)