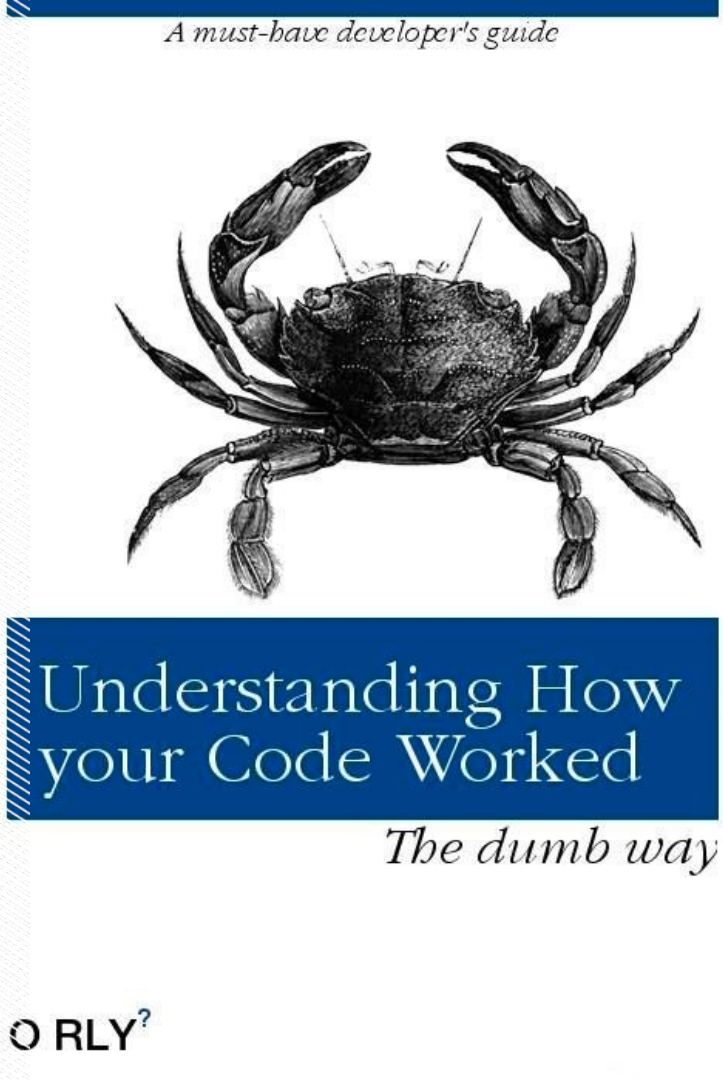
CSS Methodologies

or how to add hundreds of utility classes to each ~~project~~ module

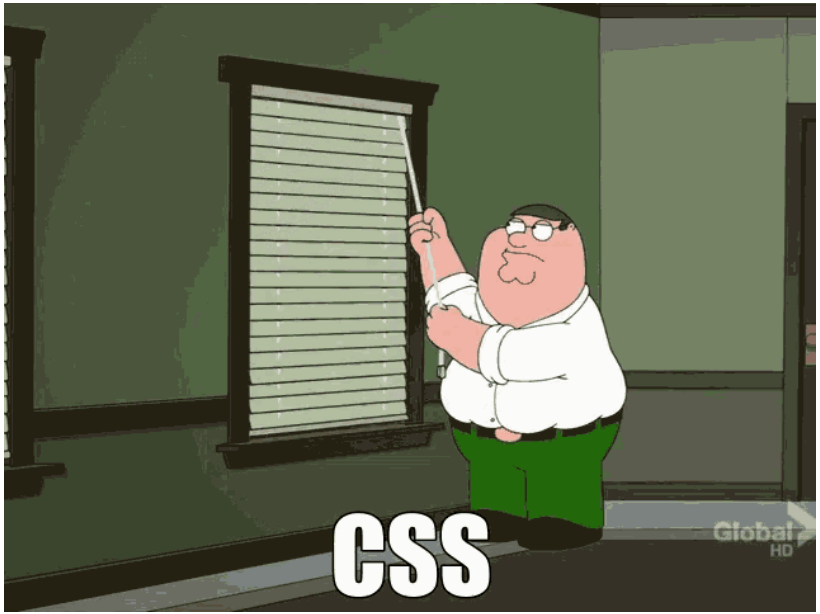


Frontend Junior Program - 2022

Agenda

1. Fighting with the Complexity
2. GOALS
3. BEM
4. OOCSS, SMACSS
5. Atomic CSS, tailwindcss

FIGHTING WITH THE COMPLEXITY

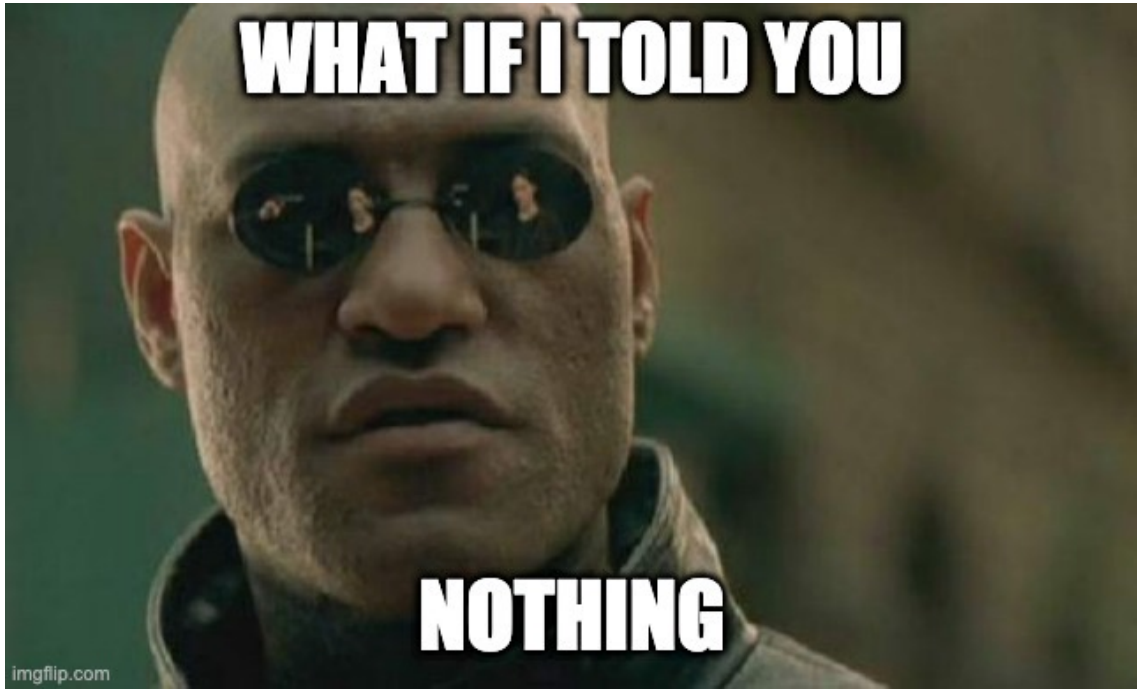


the obligatory CSS gif – been there, done that

“Writing CSS is easy.

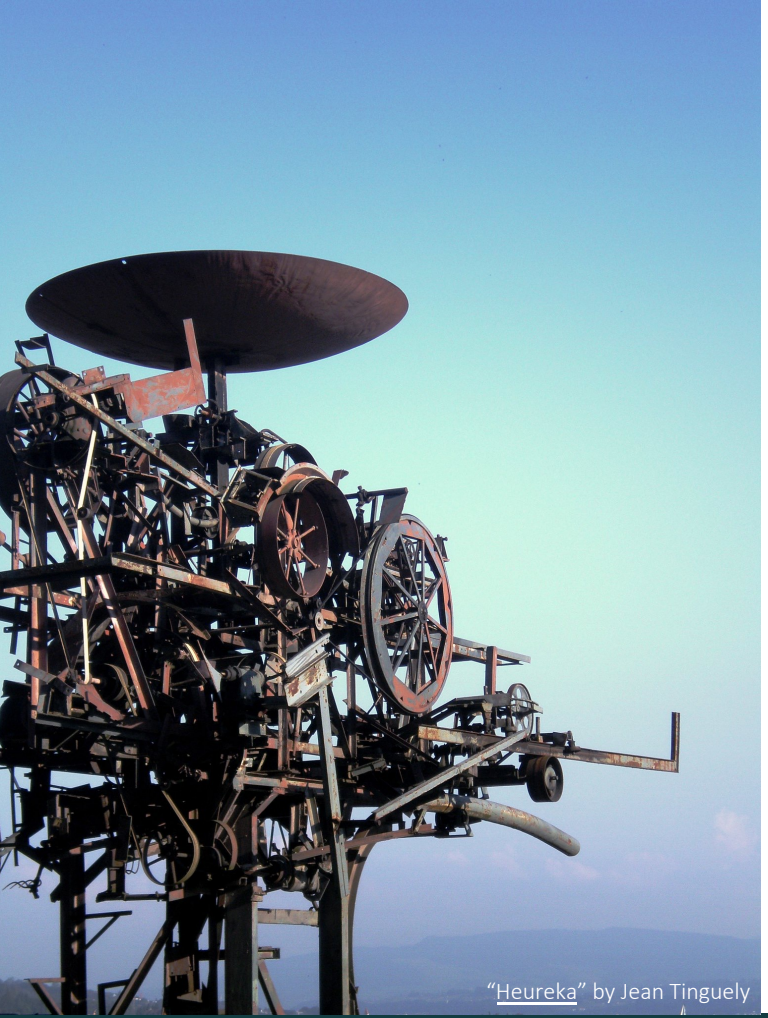
Scaling CSS and keeping it maintainable is not.”

What is wrong with CSS?



To put it simply: CSS is a complex solution for a complex problem.

CSS doesn't suck, you're just doing it wrong, but still: What's Wrong with CSS?



"Heureka" by Jean Tinguely

Software systems are complex by nature –
and we simply cannot run away from that.

"Alas, this complexity we speak of seems to be an essential property of all large software systems. By essential we mean that we may master this complexity, but we can never make it go away."

– Grady Booch

CSS - External complexity

Size	Lines of Code (**LoC** / project; LoC / component)
Interdependence and interrelations	between components, between global/theme CSS and components
Goals and objectives	readability, consistency, modularity, reusability, scalability, maintainability; part of **SDLC** (Software Development Life Cycle); must fulfil **NRFs** (Non-Functional Requirements)
Stakeholders, concurrent engineering	Product Owner, designer, developer (local / distributed, library / component), test engineer, 3rd party agencies (usability, accessibility, performance, security)
Management practices	roles and responsibilities (RACI), processes, communication, delegation
Technology	• **tech stack**: IDE, UX tools (Zeplin), CSS, Preprocessor (Sass), Methodology (BEM), Modularity (CSS modules, Styled components), Build tools, minifiers (Webpack), Quality gates (Stylelint, SonarQube) • **source**: legacy code, tech debts, 3rd party components, project architecture • **target**: different devices, browsers, RWD, native mobile
Diversity	a higher number of elements and a higher variety across elements increase complexity
Ambiguity	uncertainty in project requirements
Flux	requirements are constantly changing (Agile!)

CSS - Internal complexity

Standard	continuously evolving standard
Target	intended as a document presentation layer, not designed for web applications; the target media is flexible viewport (size, zoom), even the media itself can be changed (e.g., screen, print)
Interdependence and interrelations	tightly coupled with HTML; depends on browsers' internal style; inheritance add unpredictable complexity; can be interfered by JavaScript; some properties amends or disables other properties (e.g., display, position); relative units;
Implementation	browsers follow the standard at a different pace, many times differently; vendor prefixes; obsolete browsers needs to be supported; input elements design varies highly by OS; global by nature;
Flexibility	in some cases, it is very rigid: input HTML elements' style cannot be / very tricky to override (select, checkbox, radio)
Requires tooling	inability to use as is, requires wide range of external tools and methodologies to use
Polymorphic language	shorthand properties
Ambiguity	seemingly similar results can be achieved with completely different approaches (sometimes even with JavaScript); actual result is unpredictable (e.g., missing web-fonts)
Learning curve	very easy to start, extremely hard to master; non-trivial parts (e.g., box-model)

just for reference, no need to learn! → Complexity and Project Management: A General Overview

GOALS

The Grammar

You don’t write code for the browser;
you don’t write it for the end users.

You write it for the person who takes over the job from you (maybe a future yourself). Much like you should use good grammar.

CSS in large projects can quickly become a mess.

Minor changes fix one problem but create three more and can require ugly hacks, and small CSS changes can break JavaScript functionality.

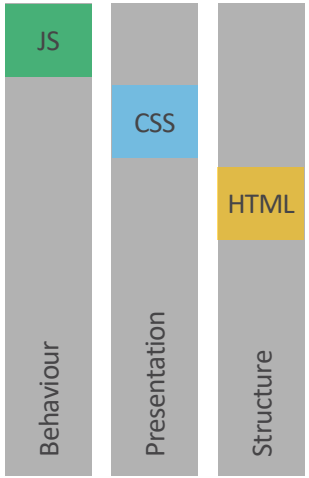
CSS methodologies creates a grammar

+ Clear rules for selector definitions
+ Fewer issues with specificity
+ Consistency
+ Easier collaboration between team members
+ Long-term maintainability

− Can add initial complexity, a new grammar needs to be learned

Target: The Separation of Concerns (SoC)

At first sight, it seems to be trivial to implement the SoC on a technology level:



This idea is reflected in csszengarden.com, where the site can be redesigned completely just by swapping the stylesheets:

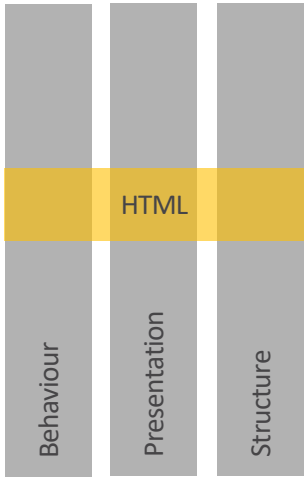
Target: The Separation of Concerns (SoC)

However, as web technologies target much more than a structural system for accessing semantically marked documents, keeping the different layers separated is not that trivial.

Also, with the emerging of new technologies, the separation becomes even less clear. What good is a CANVAS without any scripting? Should we animate in CSS or in JavaScript? Is animation behavior or presentation? Should elements be needed solely for visual effects placed in the HTML or generated with JavaScript or with CSS :after and :before?

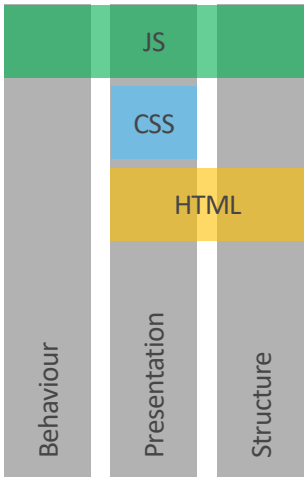
Wikipedia: Separation of Concerns

The Separation of Concerns, the history



HTML

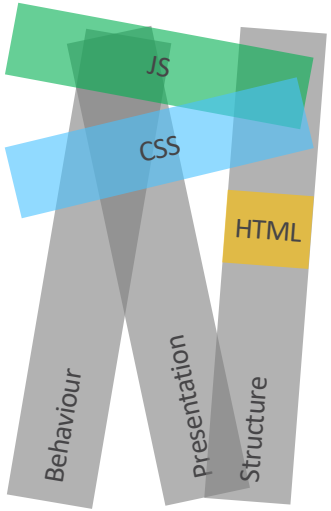
HTML: presentation attributes in markup (align, font, center, width, etc.)



DHTML

JS: animation, DOM manipulation, changing inline styles

HTML: inline styles, table design, separator gif, FIR (Fahrner image replacement)



Reality

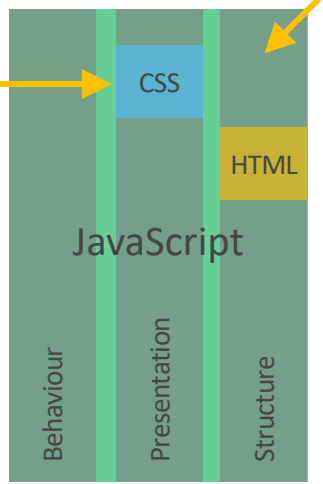
JS: everything, too (jQuery)

CSS: accordion, menus on hover, RWD

The Separation of Concerns, frameworks

CSS in JS, using the full power of JavaScript, but the CSS code still separated from the business logic and from the markup

CSS code is heavily modularized



JavaScript builds the DOM, but only in a declarative way (React, Angular), JQuery like direct DOM manipulation is anti-pattern.

HTML only exists as a template in framework code (e.g., jsx)

Separation of Concerns

According to SoC (by technology layers), the HTML should only contain information about your content, and all of the styling decisions should be made in your CSS.

This example seems harmless at first, as it does not contain styling. Does it really not?

an infamous utility class, which brings styling directly into the markup.

```
<p class="text-center">Hello there!</p>

<style>
    .text-center {
        text-align: center;
    }
</style>
```

While even some CSS methodologies are built around the utility classes, from SoC viewpoint, they are more than questionable.

a semantic class

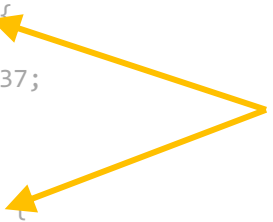
```
<p class="greeting">Hello there!</p>

<style>
    .greeting {
        text-align: center;
    }
</style>
```

Concerns are Not So Separated

Here, the CSS is like a mirror for the markup, perfectly reflecting the HTML structure with nested CSS selectors.
The markup isn't concerned with styling decisions, but CSS is very concerned with the markup structure.

```
<div class="press-release">
  <h1>PRESS RELEASE</h1>
  <div>Apple earns historic Academy Award nominations…</div>
</div>
```

```
.press-release > h1 {
  font-size: 12px;
  line-height: 1.33337;
  font-weight: 700;
}

.press-release > div {
  font-size: 19px;
  line-height: 1.21053;
  font-weight: 700;
  letter-spacing: .012em;
}
```



markup references in CSS

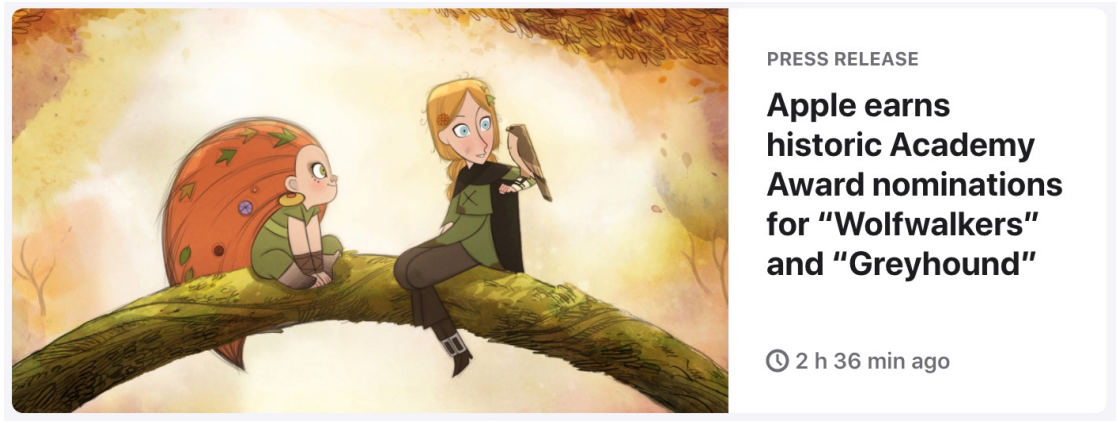
In future, HTML cannot be changed without changing the CSS code as well.

Decoupling Styles From Structure

Keeping selector specificity low and making your CSS less dependent on your particular DOM structure.

```
<div class="tile__head">
  <div class="tile__category">PRESS RELEASE</div>
  <div class="tile__headline">Apple earns historic Academy Award nominations…</div>
</div>
```

```
.tile__category {
  font-size: 12px;
  line-height: 1.33337;
  font-weight: 700;
}

.tile__headline {
  font-size: 19px;
  line-height: 1.21053;
  font-weight: 700;
  letter-spacing: .012em;
}
```



Apple is concerned about SoC, this is the original code

Target: Structure

CSS is global by nature*, and methodologies can bring a natural structure to the CSS code.

Organizing the CSS is far from trivial, even when we think about components, but methodologies can provide clear guidelines about how to break down, how to name and how to structure.

* there is a hope.

MDN: Organizing your CSS

Targets: readability, consistency, modularity, reusability, scalability

Don’t use the ID selector!

The ID provides a unique name for an HTML element. If the name is unique, you can’t reuse it, which prevents you from reusing the code.

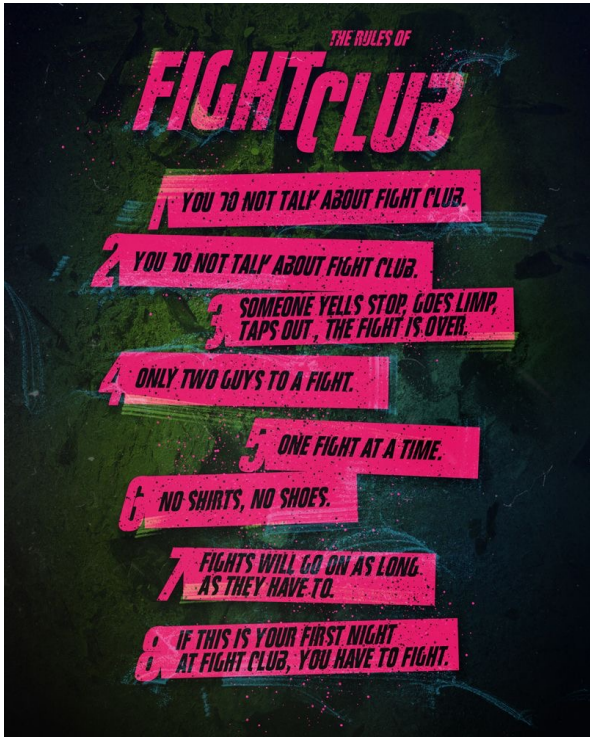
Don’t use the Type selector!

HTML page markup is unstable: A new design can change the nesting of the sections, heading levels (from <h1> to <h3>) or turn the <p> into the tag.

Any of these changes will break styles that are written for tags.

Don’t use the Universal selector (*)

The universal selector indicates that the project features a style that affects all nodes in the layout. This limits reuse of the layout in other projects. In addition, a universal selector can make the project code unpredictable: it can affect other parts of the markup.



rules, rules and rules…

… with even more rules

Don’t use the CSS Reset

CSS reset is a set of global CSS rules created for the whole page. These styles affect all layout nodes, violate the independence of components, and make it harder to reuse them.

Don’t use Nested selectors

Nested selectors increase code coupling and make it difficult to reuse the code.

The BEM methodology doesn’t prohibit nested selectors, but it recommends not to use them too much. For example, nesting is appropriate if you need to change styles of the elements depending on the block’s state or its assigned theme.

BEM

While we'll try to provide a wider view on CSS methodologies,
we will focus exclusively on BEM. Why?

It ticks every checkboxes we need in a professional development flow:

- very simple, yet extremely powerful
- has a clear a grammar
- restrictive enough to prevent issues
- decouples CSS from HTML (SoC)
- provides readable and maintainable markup
- forces consistent, modular, reusable and scalable component structure
- makes it easier to migrate the codebase to Angular or React

BLOCK, ELEMENT, MODIFIER

BEM

BEM (Block, Element, Modifier) is simply a naming convention for classes.

It can be considered as a component-based approach to web development. The idea behind it is to divide the user interface into independent blocks. This makes interface development less troublesome even with a complex UI, and it allows reuse of existing code without copying and pasting.

BEM makes your code scalable and reusable, thus increasing productivity and facilitating teamwork.

getbem.com is an excellent source of information about BEM. It is highly suggested starting here!

CSS Tricks: BEM

The Basics Of BEM

The BEM methodology is a set of universal rules that can be applied regardless of the technologies used, such as CSS, Sass, HTML, JavaScript or React.

BEM helps to solve the following tasks:

- Reuse the layout
- Move layout fragments around within a project safely
- Move the finished layout between projects
- Create stable, predictable and clean code
- Reduce the project debugging time

BEM 101

Block

A functionally independent component that can be reused.
In HTML, blocks are represented by the class attribute.

The block name describes its purpose ("What is it?" — menu or button), not its state ("What does it look like?" — red or big).

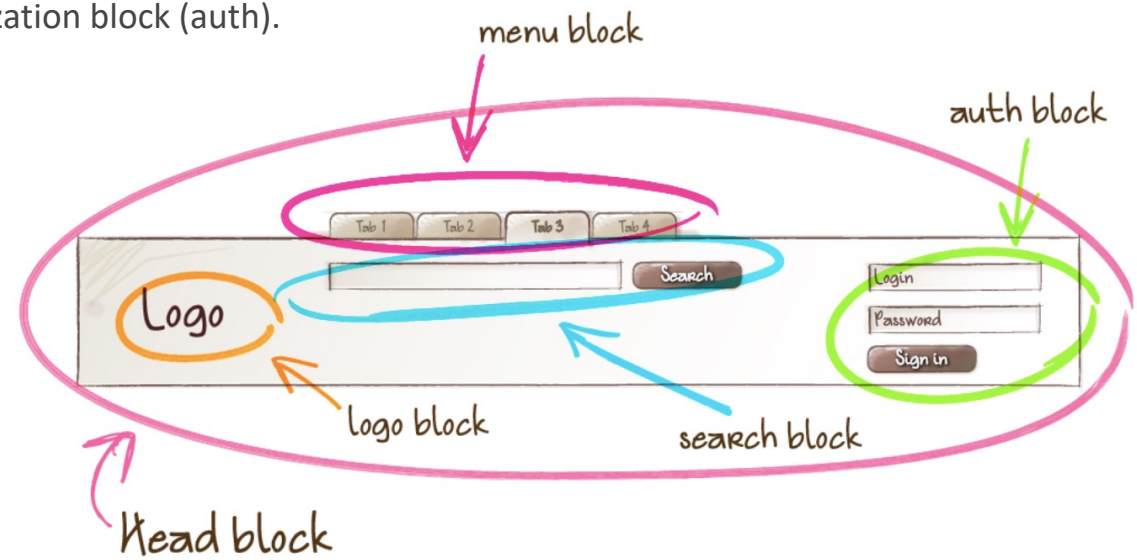
Usually, block names are unique in the application and act as a namespace.

```
<!-- Correct. The 'error' block is semantically meaningful -->
<div class="error-message"></div>

<!-- Incorrect. It describes the appearance -->
<div class="red-text"></div>
```

Block Features - Nested Structure

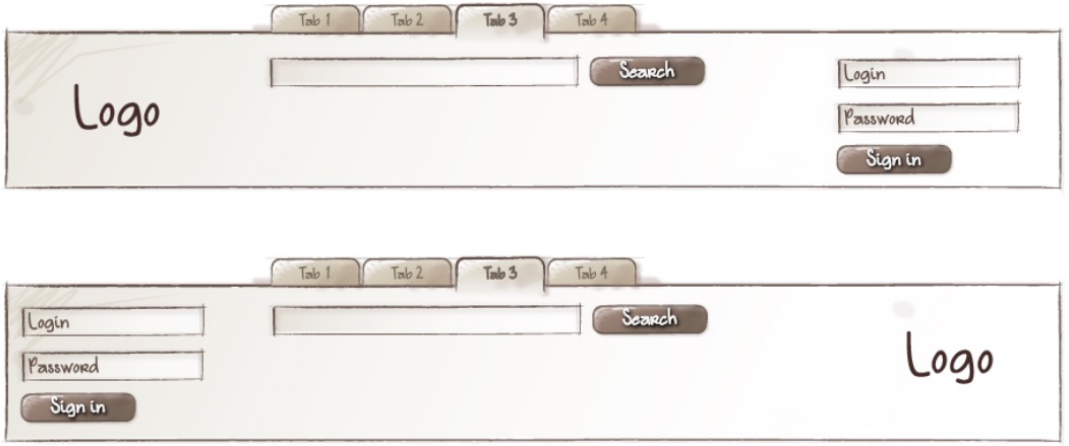
Blocks can be nested inside any other blocks.

For example, a head block can include a logo (logo),
a search form (search), and an authorization block (auth).



Block Features - Arbitrary Placement

Blocks can be moved around on a page, moved between pages or projects. The implementation of blocks as independent entities makes it possible to change their position on the page and ensures their proper functioning and appearance.



Block Features - Re-use

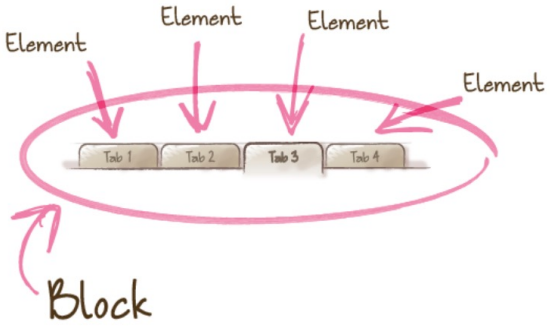
An interface can contain multiple instances of the same block.

Element

An element is a composite part of a block that can't be used separately from it.

The structure of an element's full name is **block-name__element-name**.

The element name is separated from the block name with a separator, such as double underscore (__).




```
<!-- 'search-form' block -->
<form class-"search-form">
    <!-- 'input' element in the 'search-form' block -->
    <input class="search-form__input"/>

    <!-- 'button' element in the 'search-form' block -->
    <button class="search-form__button">Search</button>
</form>
```

Guidelines For Using Elements - Nesting

- Elements can be nested inside each other.
- You can have any number of nesting levels.
- An element is always part of a block, not another element.
  This means that element names can't define a hierarchy such as block__elem1__elem2.

```
<form class="search-form">
    <div class="search-form__content">
        <input class="search-form__input">

        <button class="search-form__button">Search</button>
    </div>
</form>
```

```
<form class="search-form">
    <div class="search-form__content">
        <!-- Recommended: 'search-form__input' or 'search-form__content-input' -->
        <input class="search-form__content-input">

        <!-- Recommended: 'search-form__button' or 'search-form__content-button' -->
        <button class="search-form__content-button">Search</button>
    </div>
</form>
```

Guidelines For Using Elements - Membership

An element is always part of a block, and you shouldn't use it separately from the block.

```
<!-- 'search-form' block -->
<form class="search-form">
    <!-- 'input' element in the 'search-form' block -->
    <input class="search-form__input">
    <!-- 'button' element in the 'search-form' block -->
    <button class="search-form__button">Search</button>
</form>
```

```
<!-- 'search-form' block -->
<form class="search-form">
</form>

<!-- 'input' element in the 'search-form' block -->
<input class="search-form__input">

<!-- 'button' element in the 'search-form' block -->
<button class="search-form__button">Search</button>
```

Guidelines For Using Elements. Optionality

An element is an optional block component.

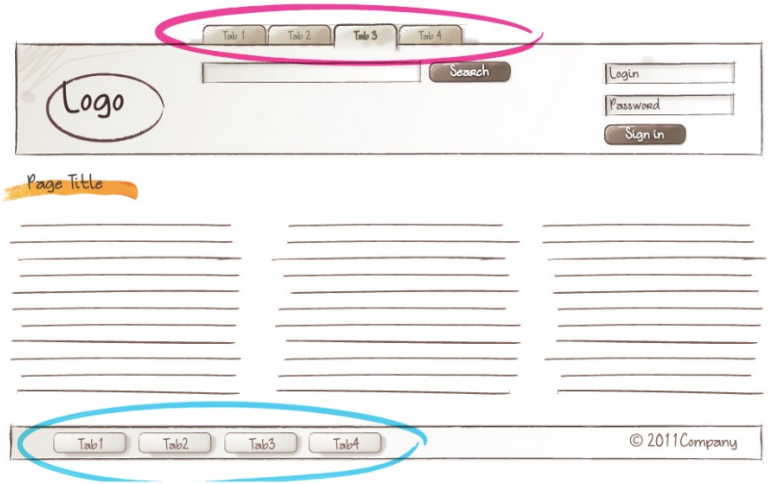
Not all blocks have elements.

```
<!-- 'search-form' block -->
<div class="search-form">

    <!-- 'input' block -->
    <input class="input">

    <!-- 'button' block -->
    <button class="button">Search</button>
</div>
```

Modifier

Modifiers are similar to HTML attributes.

The same block looks different due to the use of a modifier. For instance, the appearance of the menu block (menu) may change depending on a modifier that is used on it.



Modifier

An entity that defines the appearance, state, or behavior of a block or element.

Features:

- The modifier name describes its appearance ("What size?" or "Which theme?" and so on — size_s or theme_islands), its state ("How is it different from the others?" — disabled, focused, etc.) and its behavior ("How does it behave?" or "How does it respond to the user?" — such as directions_left-top).
- The modifier name is separated from the block or element name by a single underscore (_).

Types of Modifiers. Boolean

Used when only the presence or absence of the modifier is important, and its value is irrelevant. For example, disabled. If a Boolean modifier is present, its value is assumed to be true.

The structure of the modifier's full name follows the pattern:

- *block-name_modifier-name*
- *block-name__element-name_modifier-name*

```
<!-- The 'search-form' block has the 'focused' Boolean modifier -->
<form class="search-form search-form_focused">
    <input class="search-form__input">

    <!-- The 'button' element has the 'disabled' Boolean modifier -->
    <button class="search-form__button search-form_button_disabled">Search</button>
</form>
```

Types of Modifiers. Key-value

Used when the modifier value is important. For example, "a menu with the islands design theme": menu_theme_islands.

The structure of the modifier's full name follows the pattern:

- *block-name_modifier-name_modifier-value*
- *block-name__element-name_modifier-name_modifier-value*

```
<!-- The 'search-form' block has the 'theme' modifier with the value 'islands' -->
<form class="search-form search-form_theme_islands">
    <input class="search-form__input">

    <!-- The 'button' element has the 'size' modifier with the value 'm' -->
    <button class="search-form__button search-form__button_size_m">Search<button>
</form>
```

A Modifier Can't Be Used Alone

From the BEM perspective, a modifier can't be used in isolation from the modified block or element. A modifier should change the appearance, behavior, or state of the entity, not replace it.

```
<form class="search-form search-form_theme_islands">
    <input class="search-form__input">

    <button class="search-form__button">Search</button>
</form>
```

```
<form class="search-form_theme_islands">
    <input class="search-form__input">

    <button class="search-form__button">Search</button>
</form>
```

Mixing BEM classes

A technique for using different BEM entities on a single DOM node.

Mixes allow you to:

- combine the behavior and styles of multiple entities without duplicating code
- create semantically new UI components based on existing ones

```
<!-- 'header' block -->
<div class="header">
    <!--
        The 'search-form' block is mixed with the 'search-form' element
        from the 'header' block
    -->
    <div class="search-form header__search-form"></div>
</div>
```

Modifier naming convention

BEM is just a guideline; teams independently decide how they wish to implement it and which naming convention they will use.

using single underscore

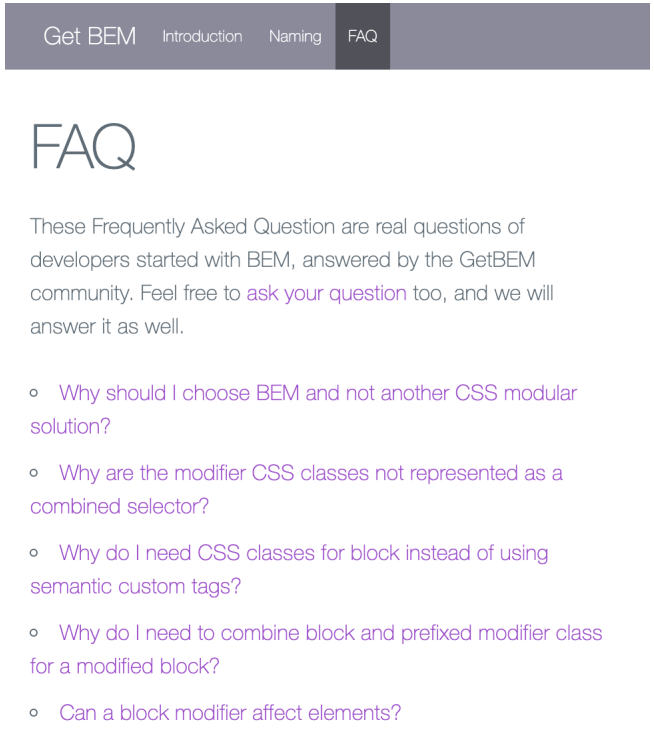
```
block__element_modifier
```

using double hyphen

```
block__element--modifier
```

Pick one and stick to it!

Next steps



As you will discover BEM, you will have questions, inevitably. The BEM is a restrictive tool (as the others), so at first you will constantly run into obstacles.

This is fine. This is the sign that you turned into an uncharted territory, so you have to turn back.

The BEM experts know that, and they already collected a FAQ for you, targeting exactly this kind of questions.

We cannot recommend enough to check that.

Will you use BEM with React or Angular?

Probably not*, but its similar philosophy will provide you the same mindset: it will be easier to understand why and how to break down the components. You don't need to unlearn it later - you can utilize these principles even with the next generation of SPAs.

** That being said, we used the BEM + Angular combination successfully in a mid-scale project (100+ developers), where we built a complete component library with BEM + Sass, and utilized it in the angular components exclusively, almost eliminating the need for CSS code on a component level.*

OBJECT-ORIENTED CSS

OOCSS

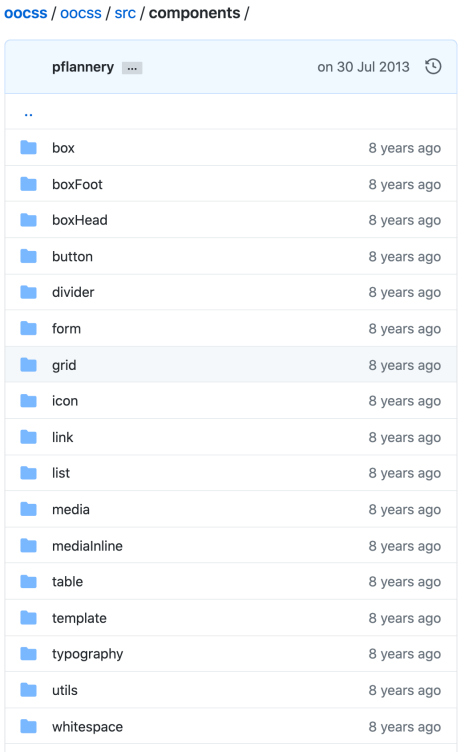
Object-Oriented CSS basically thinks about a site as a complex user interface (UI), built by smaller, functional building blocks – objects –, like box, grid, icon.

As a consequence, the OOCSS starts with a component library, and composes the site from these components.

While the OOCSS is an interesting concept, and it is one of the first methodologies, it never got a momentum – we refer it only as a part of the history.

Original blogpost,
presentation (video), presentation (slides),
Github, Wiki

the original OOCSS component library



the project is pretty much abandoned now...

SCALABLE AND MODULAR ARCHITECTURE FOR CSS

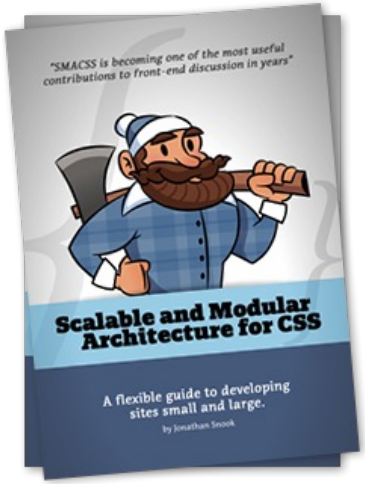
SMACSS

At the very core of SMACSS is categorization.
By categorizing CSS rules, we begin to see patterns and can define better practices around each of these patterns.

There are five types of categories:

- *Base*
- *Layout*
- *Module*
- *State*
- *Theme*

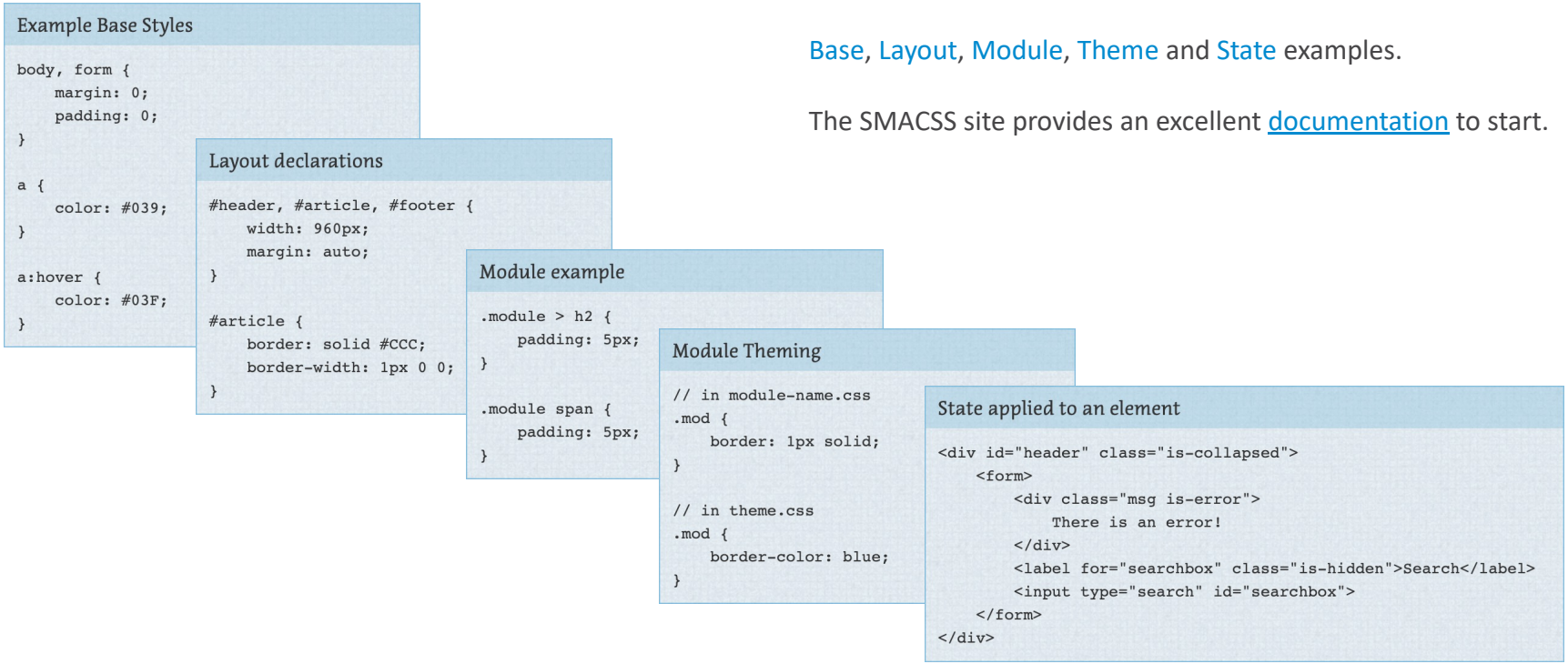
SMACSS (pronounced “smacks”) is more style guide than rigid framework.



“I have long lost count of how many web sites I’ve built. … What I have discovered are techniques that can keep CSS more organized and more structured, leading to code that is easier to build and easier to maintain.”

Scalable and Modular Architecture for CSS

SMACSS

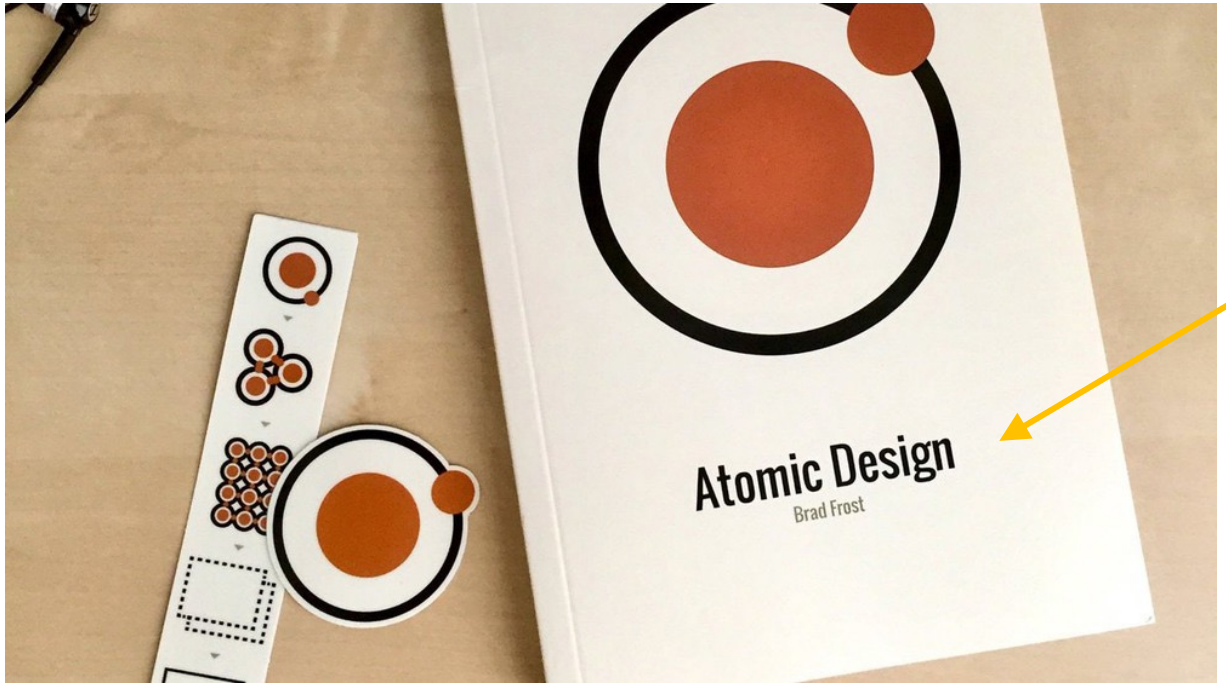


Base, Layout, Module, Theme and State examples.

The SMACSS site provides an excellent documentation to start.

ATOMIC CSS / TAILWINDCSS

Not to be confused with Atomic Design



Not this

Atomic Web Design

Atomic CSS

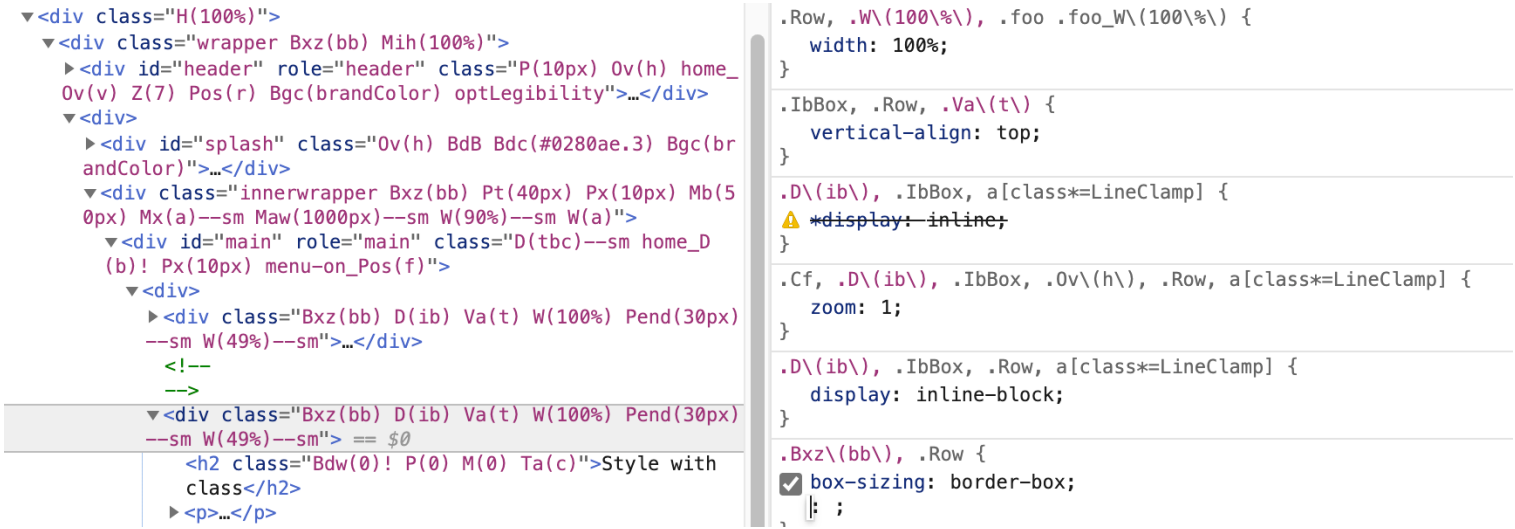
Atomic CSS, like inline styles, offers single-purpose units of style, but applied via classes. This allows for the use of handy things such as media queries, contextual styling and pseudo-classes. The lower specificity of classes also allows for easier overrides. And the short, predictable classnames are highly reusable and compress well.

With Atomic CSS, a separate class is created for each reusable property. For example, margin-top: 1px; assumes creation of a class *mt-1* or *width: 200px;* the class *w-200*.

This style allows you to minimize the amount of CSS-code by reusing declarations, and it's also relatively easy to enter changes into modules.

Atomic CSS

Atomic CSS as an extensive grammar



an example of the Atomic CSS code – it speaks for itself: basically you need to learn a new CSS language, just to be able to use that as an inline CSS.

Atomic CSS / tailwindcss - pros

Having a restrictive tool and a precise grammar, always come with benefits:

- you don't write CSS code anymore (or significantly less), you write *Atomic classes*
- by tightly coupling the presentation with the structure, it is easier to reason about the styling
- after you get used to the new language, creating a new component will be a breeze: you just copy+paste the markup, modify it, and you are done
- you can combine typical combinations into a new atomic class
- my mixing atomic css with semantic classes testing and debugging will be possible
- when using an additional templating language as well (loops, conditionals, variables), you can modify the styling on the fly by manipulating the classes directly in the markup
- the markup with atomic classes can be reusable
- atomic classes are having very low specificity, easy to override

The benefits are promising, right? The main problem is, that while it solves issues which you should not have in the first place in any professional team, it will introduce other problems and a complete lock down into a very unique and specific toolset. When you will face with the issues, there is no way back.

Atomic CSS / tailwindcss - cons

While Atomic CSS and similar frameworks (tailwindcss) are fairly popular now, these are anti-patterns in every aspect we were considering regarding the complexity:

- it creates a tight coupling between different layers (presentation vs structure)
- it requires to learn a complex and non-trivial language
- it makes the markup cryptographic and hard to maintain
- from now on, every commit regarding styles will be done in the markup, which
  - will invalidates the snapshot tests
  - makes hard to decide whether there was a change in the structure / behavior, or it was just a style update
- as color names / metrics are hardwired into the markup, theming will be almost impossible, or will result in inconsistent / nonsensical markup and css
- makes teamwork very hard, as style / behavior changes will be done in the same file, brace yourself for git conflicts
- it is a vendor locking into your codebase, you will depend on another library…
- …or libraries, because to overcome its fundamental problems you will need to utilize many plugins

and we've just started to scratch the surface.

OOCSS/Atomic CSS are Responsive Web Design 'anti-patterns'

Remember: your ultimate goal is the *simplicity*

Your job will be very complex, because the problems we need to solve;
the solutions we need to implement are very complex.

To deal with the complexity in the product domain is your real task.

Not to fight with the technology. HTML, CSS, JavaScript and any framework are just tools.
Like a hammer. Any unnecessary complexity in these tools will detract you
from this quest and forces you to focus on the tool itself.

And then you've just doomed Fantastica.


epam
Q&A
edu_hu@epam.com