CSS NOVICE TO NINJA

by **Nick Salloum**

THE IMPORTANCE OF CSS

Understanding basic CSS is important, and here's why:

- We work with JavaScript, a predominantly front-end language
- CSS has been around since 1997, nearly as long as JavaScript
- HTML, CSS, and JavaScript make a powerful combo when used in tandem
- CSS brings visual life to our apps
- A world without design would be very boring

THE MYSTERIES OF CSS

Let's uncover them! By the end of this talk, you'll be able to confidently approach CSS. You'll also:

1. have a strong command of the fundamentals
2. learn some best practices for structuring CSS
3. be comfortable exploring modern CSS

I'll break this talk into 2 main parts.

SOME KEYWORDS

In this talk, you'll hear me say a few keywords often. Let's define them with a few lines of CSS:

```
p {
  color: #333;
}

.el {
  transform: scale(1.1);
}
```

- to the left of the opening curly brace is the **selector**
- between the curly braces are the **property** and **value** pairs

PART 1

THE FUNDAMENTALS

There are 4 fundamentals that must be mastered:

1. the box model
2. the display property
3. position
4. floats

After that, CSS becomes a lot more fun.

THE BOX MODEL

LET'S TALK ABOUT THE BOX MODEL.

- This is a standardized way for browsers to understand how to render certain element properties.
- In a document, every element is represented as a rectangular block.
- Browsers try to render these elements based on the box model.

THE EDGES OF A BOX

Each of these boxes has 4 sides, and each of these sides has 4 edges to consider:

1. the content edge
2. the padding edge
3. the border edge
4. the margin edge

These edges are in order, from the edge of the content go out.

A VISUAL REPRESENTATION

BOX MODEL BEHAVIOUR

The box model behaviour is governed by this property:

```
box-sizing
```

It defines how the browser calculates the widths and heights of elements.

A LIKELY SCENARIO

Let's assume we're targeting an element with a class name of `box`. Our CSS might look like this:

```
.box {
}
```

Inside, we might:

- declare explicit width and height properties
- add some borders
- throw in some padding

What do you expect to happen to the final dimensions?

THE BOX SIZING DEFAULT

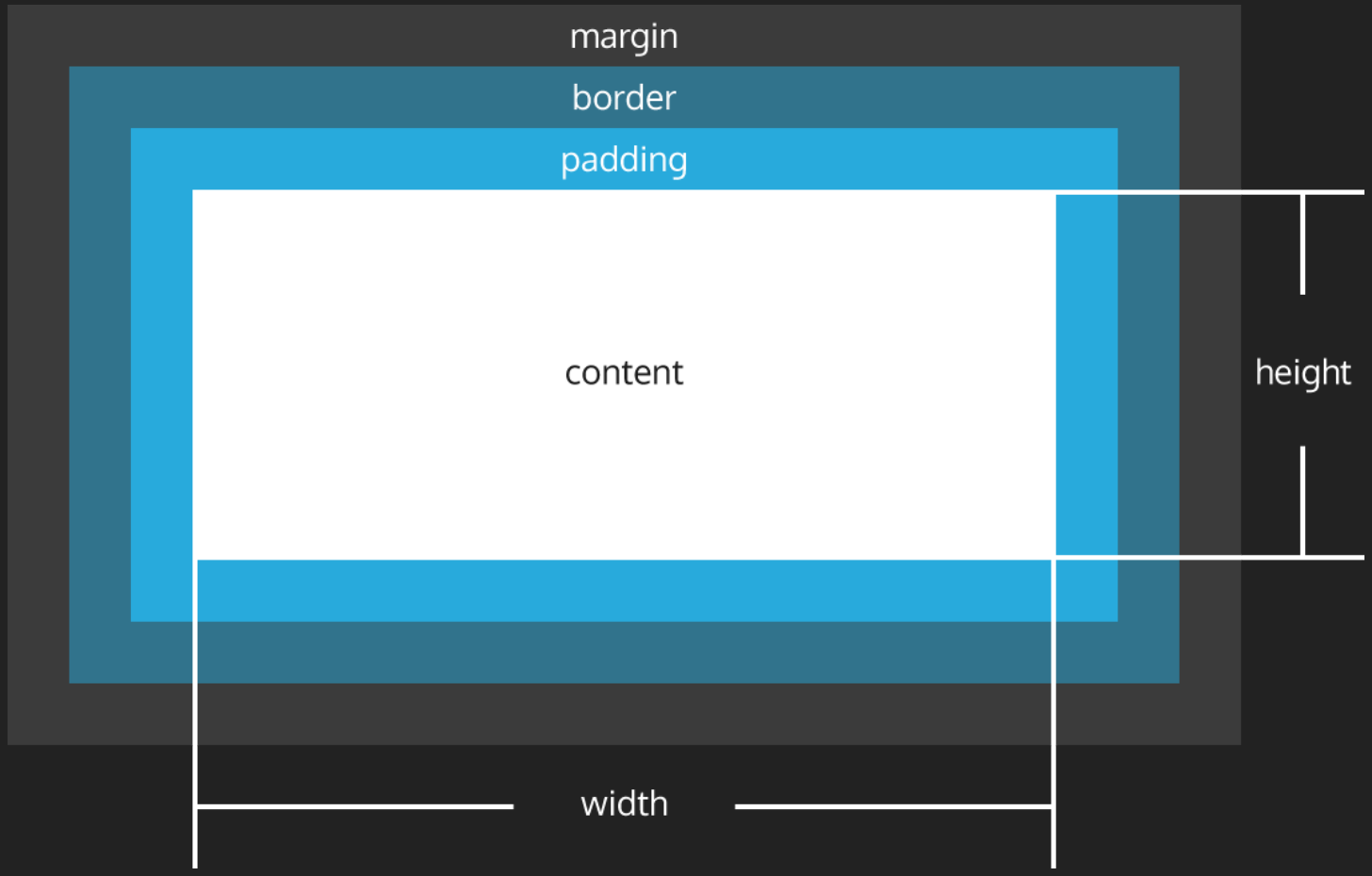
By default, the `box-sizing` property is set to:

```
box-sizing: content-box
```

This means that:

- the width and height properties are measured using only the content
- additional size-rendering properties like padding and borders are added onto that after
- **360px** box + **20px** border = **400px** box
- :(

DEFAULT BEHAVIOUR VISUALIZED



PRESERVING SANITY WITH A RESET

This leads to a lot of confusion, and it's highly unintuitive. Let's fix that! Here's a universal reset to save you #333 hairs:

```
*,
*::before,
*::after {
  box-sizing: inherit;
}

html {
  box-sizing: border-box;
}
```

DISSECTING THE RESET

Let's go through it step by step:

- the `*` selector is the universal selector
- the `::before` and `::after` parts are "pseudo elements"
- we're telling everything (including the pseudo elements) in our document to inherit a `box sizing` property
- we're setting the topmost selector (the `html` selector) to have a `box-sizing` property of `border-box`.

WHAT EXACTLY IS "BORDER-BOX"?

Unlike `content-box`, the `border-box` property tells the browser to:

- calculate the width and height of elements by **including** the padding and border properties
- **360px** box + **20px** border = **360px** box with **20px** inner border
- no more `#333` hairs
- :)

THE BORDER-BOX RESET VISUALIZED

A QUICK RECAP OF THE BOX MODEL

Here are some key takeaways from the box model:

1. The box model governs the rendering of certain element properties in the browser.
2. Every element occupies its own rectangular box.
3. The default behaviour is unintuitive.
4. We can reset it to a much more intuitive implementation with a few lines of CSS.
5. You should do this at the top of every CSS file you author.

DISPLAY

THE DISPLAY PROPERTY

The display CSS property specifies the type of rendering box used for an element.

There are many display types out there in the wild, but we will only focus on 3:

- inline
- block
- inline-block

Master these, and your CSS skills will skyrocket.

INLINE DISPLAY

Inline display means that elements are displayed inline, inside the current block, on the same line.

In general, you'll want to use this property on:

- elements that contain text and sit inline, like `p` tags
- elements that contain text-altering properties without creating a new block context, like `span`, `em`, and `bold`
- elements that you don't want to create a new block context

INLINE DISPLAY GOTCHAS

When using inline display, take into account the following:

- only left and right margins will be respected, top and bottom values will be ignored
- padding will be respected in all directions, but will only push surrounding elements to the left and right
- tags that you'd expect to be displayed inline by default most likely do

Let's look at a couple examples.

INLINE DISPLAY EXAMPLE #1

Here are some key takeaways:

- notice that the top and bottom margins have no effect whatsoever
- notice also that the top and bottom paddings, while they do affect the targeted element, don't affect the surrounding ones
- the left and right margins and paddings have a more expected effect

INLINE DISPLAY EXAMPLE #2

Here are some key takeaways:

- notice that the top and bottom margins have no effect whatsoever
- notice also that the top and bottom paddings, while they do affect the targeted element, don't affect the surrounding ones
- the left and right margins and paddings have a more expected effect

BLOCK DISPLAY

Block elements do not sit inline. They create a new "block" context, and are usually "container" type elements like `div`s, and `section`s, or text blocks like `p` and `h1`. By default:

- UA stylesheets set quite a few elements to exhibit block display
- they take up as much horizontal space as possible...
- ...as long as a width isn't specified

BLOCK DISPLAY GOTCHAS

When using block display, take into account the following:

- block elements behave like you'd expect them to
- margins, paddings, widths, and heights are all respected as defined
- block elements can't be placed side by side, unless we use floats (which we will cover later)

Let's look at some examples.

BLOCK DISPLAY EXAMPLE #1

The span created a new block context, and forced the following content to a new line.

Let's look at demo number 2.

BLOCK DISPLAY EXAMPLE #2

Even though the sum of the two box widths is less then or equal to its container's width, they still create new block contexts due to the display property.

INLINE-BLOCK DISPLAY

Inline-block does (almost) what you'd expect, in the sense that:

- it inherits some behaviour from inline display
- it also inherits some behaviour from block display

Elements with an inline-block display sit inline with the natural flow of text, like inline elements. They do, however, respect all width, height, margin, and padding properties in the same way that a block element would.

INLINE-BLOCK DISPLAY GOTCHAS

- elements take on an additional `vertical-align` property, which can be tricky to manage sometimes until you get the hang of it
- depending on how you markup your HTML, two inline-block of width 100px elements may either occupy 200px of space, or ~204px of space
- inline-block should not be used for major layout purposes

It's demo time!

INLINE-BLOCK DISPLAY EXAMPLE #1

Margins, paddings, widths, and heights have all been respected!

INLINE-BLOCK DISPLAY EXAMPLE #2

The elements don't sit side by side, even though our width calculations are precise. Why?

INLINE-BLOCK DISPLAY EXAMPLE #3

What's changed?

- the two elements now sit side by side in the markup

Woah, what's the deal with that!?

- think of inline-block elements as words
- if two words are on separate lines in your markup, the browser will render them with a space in-between
- inline-block elements behave the same way

POSITION

THE POSITION PROPERTY

What is position?

Position is everything.

Mastering position will be fundamental in your quest for CSS mastery, particularly when it comes to animation effects

BUT SERIOUSLY

What is position?

- the position property defines alternative rules for positioning elements
- when we talk about position, we talk about the flow of the page
- certain position values can cause an element to exit (or jump out) the natural flow of the page, causing a page reflow

THE BIG 4

We're going to look at the following 4 position properties:

- static
- relative
- absolute
- fixed

The properties `top`, `left`, `right`, `bottom`, and `z-index` work in tandem with position, so we will tackle them too.

STATIC POSITIONING

By default, all elements inherit this value for position. What does it mean?

- elements will stick to the normal page flow
- `top`, `left`, `right`, `bottom`, and `z-index` will have no affect whatsoever on static-positioned elements

RELATIVE POSITIONING

Setting relative positioning kicks off the beauty of positioning, and here's why:

- elements will still conform to the normal page flow
- this time, though, `top`, `left`, `right`, `bottom`, and `z-index` properties will have an effect
- elements can now be bumped from their original position with these properties
- these positional movements don't cause the page to reflow

Let's look at a live demo.

ABSOLUTE POSITIONING

Absolute positioning is beautiful and powerful, once you know what it does. Let's take a look at some of its side-effects:

- elements that are absolutely positioned break the normal page flow
- that means, these elements are removed from the flow, and other elements disregard its position
- the `top`, `left`, `right`, `bottom`, and `z-index` properties all have an affect on absolutely positioned elements

...but wait, there's more to absolute positioning.

ABSOLUTE POSITIONING

Let's take a look at some of the more intricate effects of absolute positioning:

- the positional properties move the element in relation to the nearest relatively positioned element
- if no outer element is relatively positioned, it's positional properties are calculated in relation to the document body
- absolute positioning on a block element (like a `div`) causes its width to collapse to the width of its content, unless otherwise specified

ABSOLUTE POSITIONING

(Intricate effects cont'd)

- the `top`, `left`, `right`, and `bottom` properties do not respect the outer container's padding values
- absolutely positioned elements create a relative context for absolutely positioned child elements

Alright, it's demo time!

FIXED POSITIONING

Fixed positioning is largely similar to absolute positioning:

- everything I mentioned about absolute positioning applies to fixed positioning, except for two key points
- fixed positioned elements are always relative to the document body
- they are unaffected by scrolling

Let's demo it up to see it in action.

DEMYSTIFYING Z-INDEX

All too often, I come across some CSS like this:

```
.on-top {
  z-index: 999;
}

.no__im-on-top {
  z-index: 99999;
}

.i-should-be-on-top-but-im-not {
  z-index: 9999999999;
}
```

This is a clear sign of a misunderstanding of the `z-index` property at a basic level.

WHAT Z-INDEX ACTUALLY DOES

The `z-index` property has one role, and that's to add a stacking context to elements that have either relative, absolute, or fixed positioning.

In other words, if two elements are overlapping due to positional shift, the element with the higher z-index value will be shown on top.*

WHAT Z-INDEX ACTUALLY DOES

(cont'd)

If no z-index is specified, the element that appears further down in the markup will get the preference.**

Let's demo that.

WHAT Z-INDEX DOESN'T DO

The `z-index` property is **NOT** a catch-all solution for getting elements to show up at the very top.

Applying a `z-index` of a high number does **NOT** guarantee that your element will show up on top of another.

That's because the `z-index` property is relative to its container. That sounds abstract at first, but let's demo it to make it clear.

HOW TO MANAGE Z-INDEXES

Now that you know what z-indexes do, here's how to manage them:

- use them as sparingly as possible. You will very rarely need to have multiple stacked contexts with different z-index values
- use numbers like 1 to 10, 100, and -1
- reserve a high number like 9999 for topmost instances of stacking, and only have it active on a single element at a time

FLOATS

I FLOATED SOMETING...

...and everything broke! Now here's what I'm thinking:

- floats suck
- why do floats even exist?
- CSS is dark magic
- we need to outsource this to some CSS wizard in the cloud

GUESS WHAT...

...floats don't actually suck. In fact:

- they are very easy, and there are many ways to handle them
- their conceptual existence predates CSS itself
- CSS wasn't born broken, we made it break by exhibiting a misunderstanding of the basics
- the default float behaviour actually makes a lot of sense
- you really don't need to outsource to a wizard

HISTORY CRASH COURSE

In a nutshell:

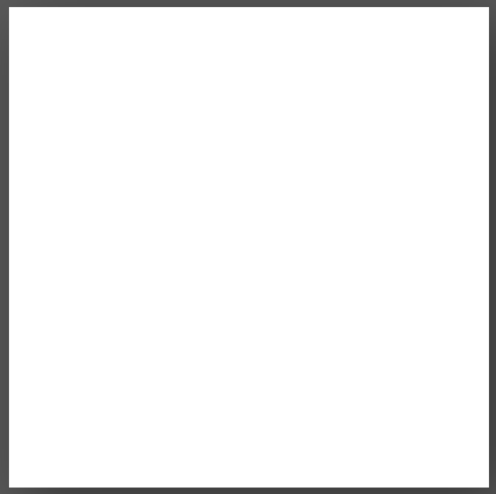
- HTML was invented as a way to interpret and compose text, and serve it on the web
- it was also meant to serve the purpose of delivering print media across the web
- CSS came along, paving the way for visual enhancements of web documents
- layout and design made its way into the web
- floats became the preferred way to describe a web pages layout (as opposed to tables)

THE CONCEPT OF A FLOAT

In print media, conceptual floats exist all the time. Let's illustrate an example:

- we may have an article, with several paragraphs
- somewhere, we may want an image to be displayed
- we align that image to the top left, and reflow the text around it
- new paragraphs also stay reflowed around the image

A VISUAL REPRESENTATION



Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

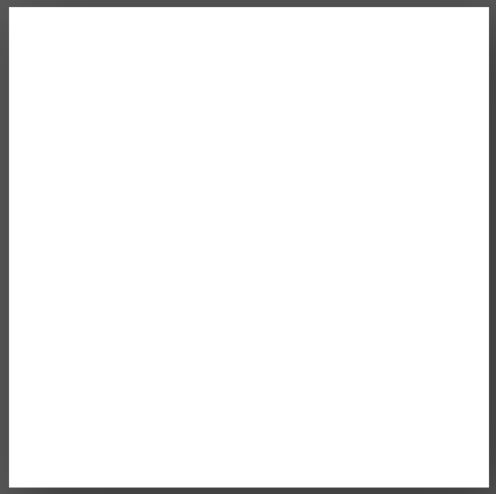
Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus modo typi, qui nunc nobis videntur parum clari, fiant sollemnes in futurum. id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus modo typi, qui nunc nobis videntur parum clari, fiant sollemnes in futurum id quod mazim placerat facer possim assum.

WHAT IF...

Now that you're a master of the display property, let's create a "what if" situation:

- what if the concept of floats didn't exist?
- what if the image created a new block context?
- what if floating that image to the left only affected its sibling paragraph?

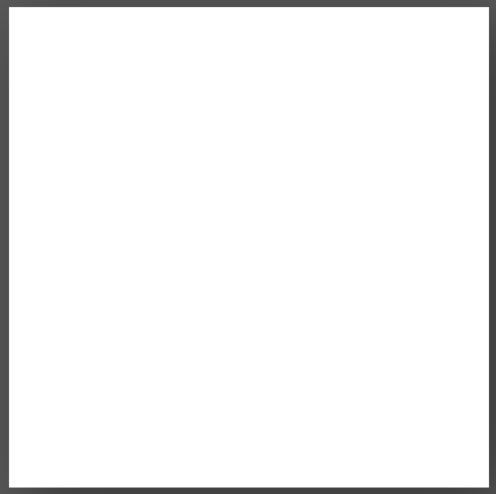
NEW BLOCK CONTEXT VISUAL



Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam

SIBLING PARAGRAPH VISUAL



Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus modo typi, qui nunc nobis videntur parum clari, fiant sollemnes in futurum. id quod mazim plac-

DEFAULT BEHAVIOUR

In order to mimic the print industry (and rightfully so), floats exhibit the following bevaiour:

- floated elements remain a part of the natural flow of the web page
- floated elements cause parent elements to "collapse"
- if a containing element contains one float and some non-floated content, the containing element will only expand to the the height of the non-floated content
- if a containing element contains only floats, it will collapse to nothing

BUT...WHY!?

At this point, let's jump out of the conceptual train of thought and get to some demos.

Let's look at our first demo, which explores the basic principle of floating and collapsing containers.

FLOATS EXAMPLE #1

Here are some key takeaways from this demo:

- floated elements do indeed exhibit the behaviour we'd expect them to, i.e. they force content to flow around it
- floated elements cause containing elements to collapse, allowing the content to re-flow as expected

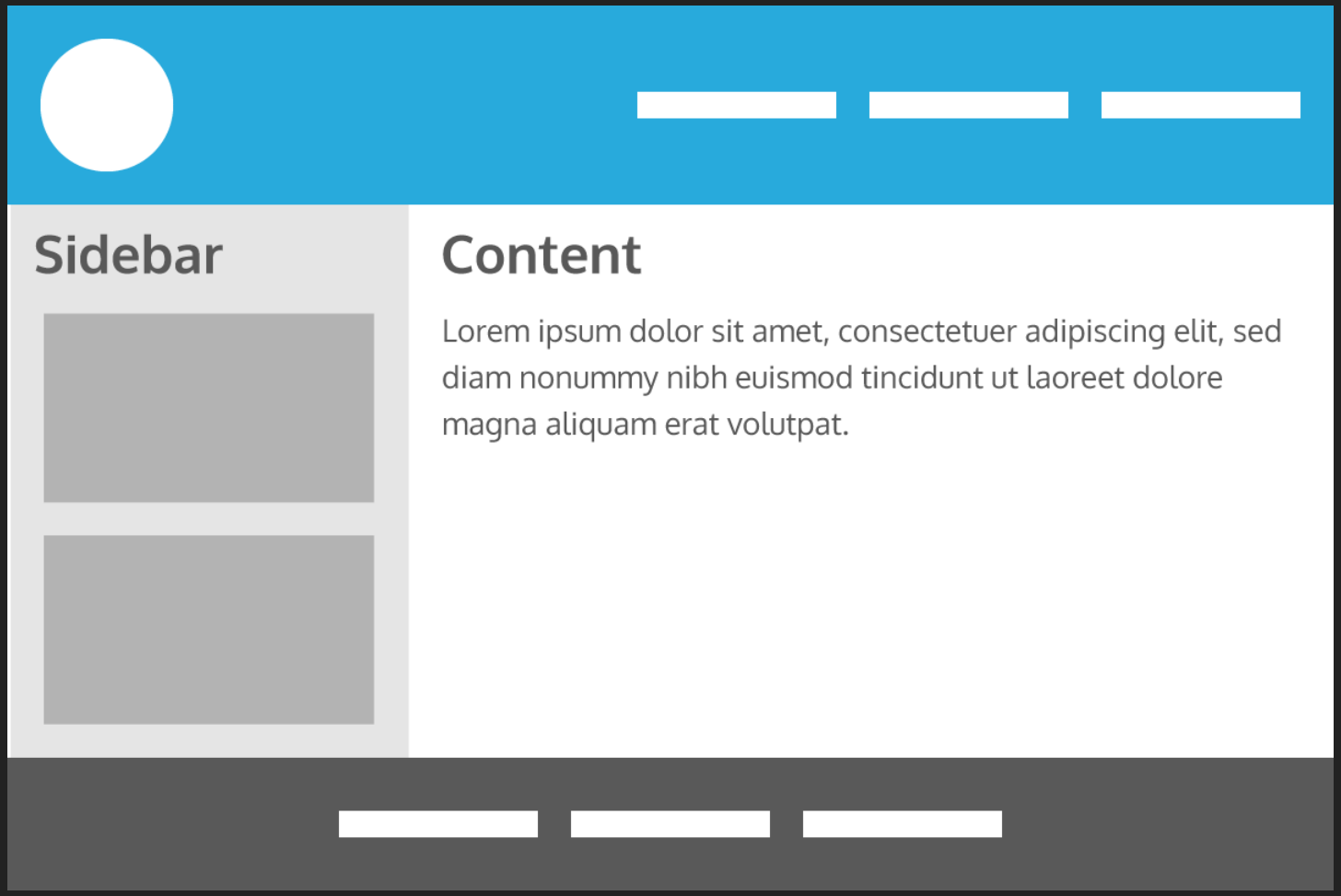
ON FLOATS & LAYOUT

By now, you're probably wondering how floats can be used to manage layouts if they cause containing elements to collapse.

Let's think about a possible result that we'd like to achieve.

A typical layout might consist of a header, main area with a content container and sidebar, and a footer.

A VISUAL REPRESENTATION



ANALYZING THE LAYOUT

Let's break down the layouts into major components:

- the page header runs across the top (forget about the logo and navigation for now)
- underneath that, we have the main area
- the main area acts as a container to the sidebar and content areas
- finally, we have a footer

Let's jump into live demo mode to see it all in action.

OH SNAP!

Floats have just wreaked havoc upon our layout.

But we should've expected that by now, based on our previous discussion on floats.

At this point, floated elements are behaving totally normal, and we need to change that.

CLEARING FLOATS

The idea behind "clearing floats" is that you issue some sort of declaration that forces the containing element to expand to contain its child elements.

There are many methods, but we're going to stick with the bulletproof and modern "clearfix" technique.

The clearfix technique defines a `clearfix` class that we can add to any float-containing element in our markup.

THE CLEARFIX CLASS

Here's what the `clearfix` class looks like:

```
.clearfix::after {
  content: "";
  display: table;
  clear: both;
}
```

This sets the `content` property of the psuedo-element `::after` to be empty, and clears it. Woah...things just got a little weird.

THE CLEARFIX CLASS IN ACTION

Now, we can do something like this:

```
<div class="header"></div>

<div class="container clearfix">
  <div class="float"></div>
  <div class="float"></div>
</div>

<div class="footer"></div>
```

And our desired layout will be achieved. Let's put this to the test!

FLOATS...?

Yeah, you're now a master of them.

- But should they really be used for layout?
- Some people say yes, some say no. I say go for it if you feel comfortable
- For the more curious minds, I highly recommend exploring Flexbox, the new super-power of layout (I use it all the time)

PART 2

WHAT WE'LL COVER

In this part, we're going to get interactive and build some stuff. CSS is mastered by practicing small bits at a time, so let's do just that. Here's what we'll do:

- create a basic, responsive, two column page layout
- introduce some best practices by extracting CSS into reusable components
- experience the magic of CSS animations at a basic level, by using transitions and transforms

WHAT WE WON'T COVER

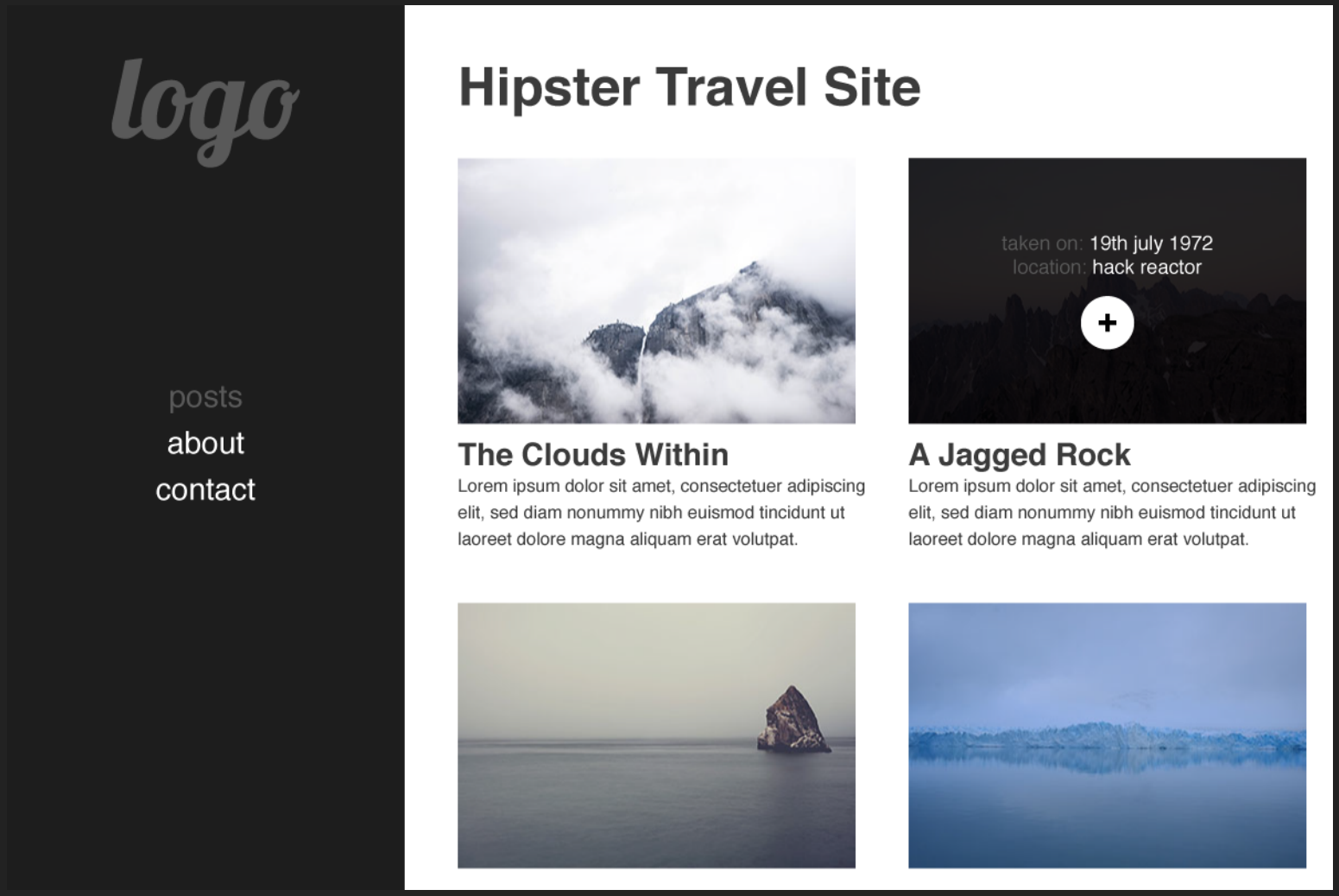
Because CSS is such a wide topic, everything can't be covered. We won't be covering:

- web fonts with @font-face
- pre-processors like Sass
- post-processors like auto-prefixer

You'll be equipped to research these hugely important and helpful topics though, so go on full steam ahead!

WHAT WE'LL CREATE

Here's a sketch of what we'll create:



WHAT WE'LL CREATE

By building this together, we'll get to experience first hand some of the challenges we face when debugging CSS.

We'll also incorporate some best practices that will help you architect scalable, reusable, modular, DRY CSS in the future.

Lots of buzzwords up there, but they are all very important.

Let's dive in!

THANK YOU