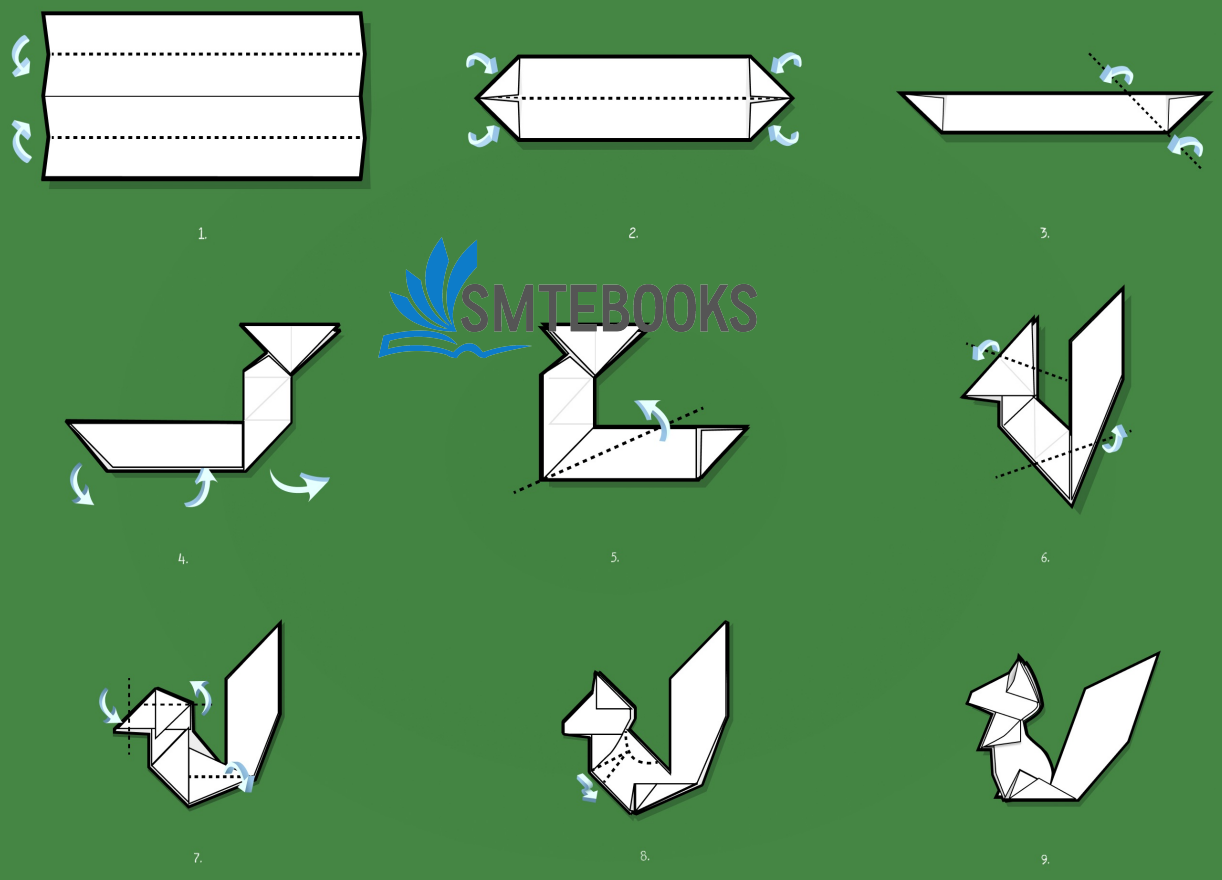
sitepoint

THE ADVANCED CSS COLLECTION



A THREE VOLUME SET

The Advanced CSS Collection

Ebook ISBN: 978-1-925836-20-2

Project editor: Craig Buckler

Cover Design: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Preface

CSS has grown from a language for formatting documents into a robust language for designing web applications. Its syntax is easy to learn, making CSS a great entry point for those new to programming. Indeed, it's often the second language that developers learn, right behind HTML.

As CSS's feature set and abilities have grown, so has its depth. In this collection of books, we'll be exploring some of the amazing things that developers can do with CSS today; tasks that in the past might only have been achievable with some pretty complex JavaScript, if at all. This collection contains:

- *Modern CSS*, which explores topics like variable fonts and transforms, and shows how they might be used in the real world
- *CSS Grid Layout: 5 Practical Projects*, which shows five complete projects that utilize the Grid Layout Layout module
- *CSS Tools & Skills* which looks at essential CSS tools and skills for modern front-end developers

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
```

```
    ⋮
new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
➥design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

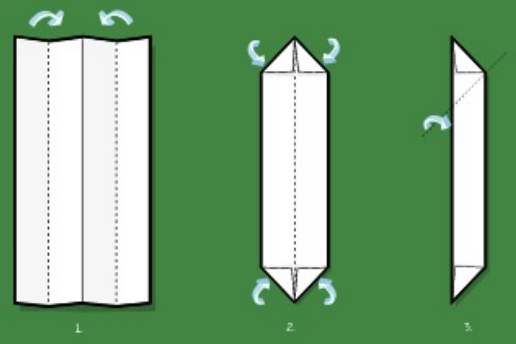
Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Book 1: Modern CSS

sitepoint

MODERN CSS



CUTTING EDGE TECHNIQUES

Chapter 1: Using CSS Transforms in the Real World

by Ilya Bodrov-Krukowski

In this article, we'll learn how CSS transforms can be used in the real world to solve various tasks and achieve interesting results. Specifically, you'll learn how to adjust elements vertically, create nice-looking arrows, build loading animations and create flip animations.

Transformations of HTML elements became a CSS3 standard in 2012 and were available in some browsers before then. Transformations allow you to transform elements on a web page — as explained in our 101 article on CSS transforms. You can rotate, scale, or skew your elements easily with just one line of code, which was difficult before. Both 2D and 3D transformations are available.

As for browser support, 2D transforms are supported by all major browsers — including Internet Explorer, which has had support since version 9. As for 3D transformations, IE has only partial support since version 10.

Ahem, Excuse Me ...

This article won't focus on the basics of transformations. If you don't feel very confident with transformations, I recommend reading about 2D and 3D transforms first.

Vertically Aligning Children

Any web designer knows how tedious it can be to vertically align elements. This task may sound very simple to a person who's not familiar with CSS, but in reality there's a jumble of techniques that are carefully preserved between generations of developers. Some suggest using `display: inline` with `vertical-align: middle`, some vote for `display: table` and accompanying styles, whereas true old school coders are still designing their sites with tables

(just joking, don’t do that!). Also, it’s possible to solve this task with Flexbox or Grids, but for smaller components, transforms may be a simpler option.

Vertical alignment can be more complex when element heights are variable. However, CSS transformations provide one way to solve the problem. Let’s see a very simple example with two nested `divs`:

```
<div class="parent">
  <div class="child">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
    ➥incididunt ut labore et dolore magna aliqua. Ut enim ad minim v
    ➥nostrud exercitation ullamco laboris nisi ut aliquip ex ea comm
    ➥Duis aute irure dolor in reprehenderit in voluptate velit esse
  </div>
</div>

<div class="parent">
  <div class="child">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
    ➥incididunt ut labore et dolore magna aliqua. Ut enim ad minim v
  </div>
</div>
```

Nothing complex: just two nested blocks with a Lorem Ipsum text of different length.

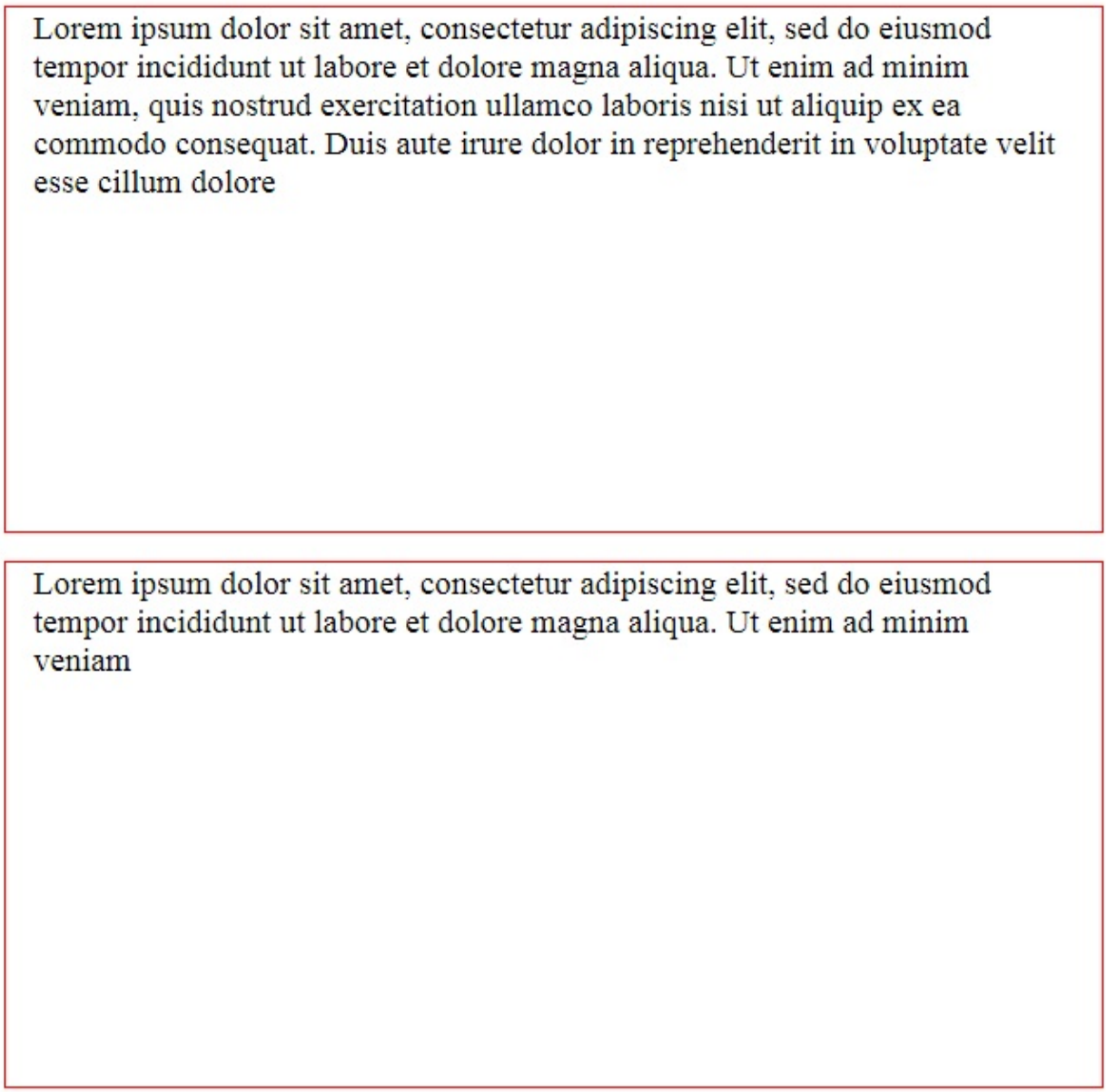
Let’s set width, height, and border for the parent, as well as some spacing to make it look nicer:

```
.parent {
  height: 300px;
  width: 600px;
  padding: 0 1em;
  margin: 1em;
  border: 1px solid red;
}
```

Also, enlarge the children font size a bit:

```
.child {
  font-size: 1.2rem;
}
```

As a result, you’ll see something like this:



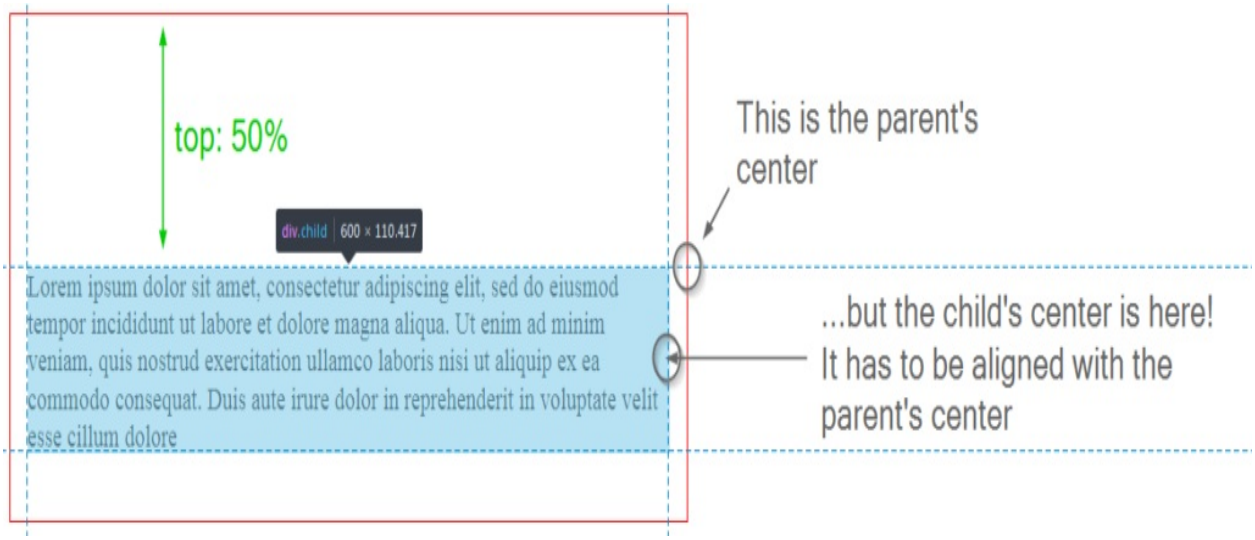
And now your customer says: “Please align the text vertically so that it appears in the middle of these red boxes”. Nooo! But fear not: we’re armed with transformations! The first step is to position our children relatively and move them 50% to the bottom:

```
.child {
  font-size: 1.2rem;
  position: relative;
  top: 50%;
}
```

After that, apply a secret ingredient — the `translateY` function — which will reposition the elements vertically:

```
.child {
  font-size: 1.2rem;
  position: relative;
  top: 50%;
  transform: translateY(-50%);
}
```

So, what's happening here? `top: 50%` moves the top of the child element to the center of the parent:



But this is not enough, because we want the child's center to be aligned with the parent's center. Therefore, by applying the `translateY` we move the child up 50% of its height:

Some developers have reported that this approach can cause the children to become blurry due to the element being placed on a “half pixel”. A solution for this is to set the perspective of the element:

```
.child {
  // ...
  transform: perspective(1px) translateY(-50%);
}
```

This is it! Your children are now aligned properly even with variable text, which is really nice. Of course, this solution is a little hacky, but it works in older browsers in contrast to CSS Grid. The final result can be seen on CodePen:

Live Code

The final result can be seen on CodePen: see the Pen CSS Transforms: Vertical-Align.

Creating Arrows

Another very interesting use case for transformations is creating speech bubble arrows that scale properly. I mean, you can definitely create arrows with a graphical editor, but that’s a bit tedious, and also they may not scale well if you use a bitmap image.

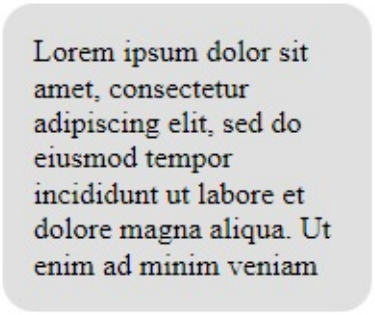
Instead, let’s try to stick with a pure CSS solution. Suppose we have a box of text:

```
<div class="box">
  <div class="box-content">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
    ➥incididunt ut labore et dolore magna aliqua. Ut enim ad minim v
  </div>
</div>
```

It has some generic styling to make it look like a speech bubble:

```
html {
  font-size: 16px;
}

.box {
```

```
  width: 10rem;
  background-color: #e0e0e0;
  padding: 1rem;
  margin: 1rem;
  border-radius: 1rem;
}
```



I'm using `rem` units here so that if the root's font size is changed, the box is scaled accordingly.

Next, I'd like to make this bubble become more "bubble-ish" by displaying an arrow to the right. We'll achieve that by using a ::before pseudo-selector:

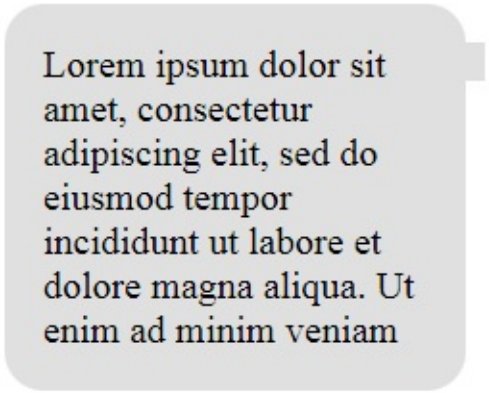
```
.box::before {
  content: '';
  width: 1rem;
  height: 1rem;
  background-color: #e0e0e0;
  overflow: hidden;
}
```

The larger the width and height you assign, the larger the arrow will be.

Now move the arrow to the right:

```
.box {
  // ...
  position: relative;
}

.box::before {
  // ...
  position: absolute;
  right: -0.5rem;
}
```

Currently, however, this element doesn’t look like an arrow at all:



Let’s align this small box vertically. As long as its dimensions are static, this is a simple task:

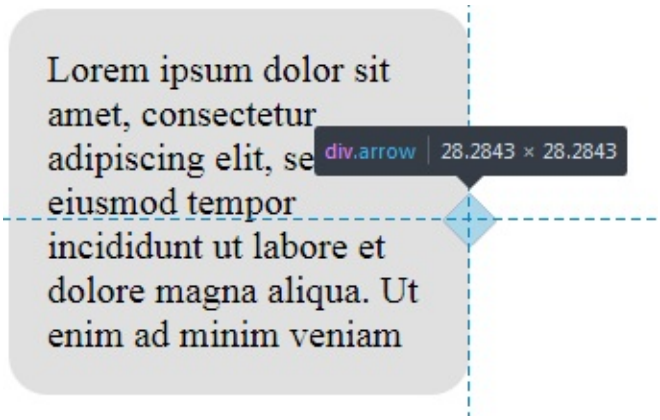
```
.box::before {
  // ...
  top: 50%;
}
```

It is not precisely aligned, but we’ll fix that later.

Now it’s time for transformation to step in. All we need to do is rotate this small box to turn it to a triangle, which will look just like an arrow:

```
.box::before {
  // ...
  transform: rotate(45deg);
}
```

Here’s a screenshot from the developer console (the box is highlighted with blue so you can see how it’s actually positioned):



Then, just move the arrow a bit to the top using negative margin so that it's positioned precisely at the middle of the box:

```
.box::before {
  // ...
  margin-top: -0.5rem;
}
```

That's it! Now, even if you need to adjust font size on the page, your box and the arrow are going to be scaled properly! HereÆs the final result:

Live Code

Here's the final result: See the Pen CSS Transformations: Arrows

Creating a "Jumping Ball" Loader

Unfortunately, things on the web don't happen instantly. Users often have to wait for an action to complete, be it sending an email, posting their comment, or uploading photos. Therefore, it's a good idea to display a "loading" indicator so that users understand they'll have to wait for a few seconds.

Previously, when there were no CSS animations and transformations, we would probably use a graphical editor to create an animated GIF loader image. This is no longer necessary because CSS3 offers powerful options. So, let's see how transformations can be utilized in conjunction with animations to create a nice-looking jumping ball.

To start, create an empty `div`:

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

Balls are usually round (Captain Obvious to the rescue!), so let's give it width, height, and border radius:

```
.loader {
  border-radius: 50%;
  width: 50px;
  height: 50px;
}
```

Let's also use a CSS background gradient editor to generate some gradient:

```
.loader {
  border-radius: 50%;
  width: 50px;
  height: 50px;
  background: linear-gradient(to bottom, #cb60b3 0%,#c146a1 50%,#a80
  ➥ 100%);
}
```

Here's our shiny purple ball that resembles pokeball:

How do we make it jump? Well, with a help of CSS animations! Specifically, I'm going to define an infinite animation called `jump` that takes 1.5 seconds to complete and performs the listed actions back and forth:

```
.loader {
  // ...
  animation: jump 1.5s ease-out infinite alternate;
}
```

Next, let's ask ourselves: what does it mean when we say "to jump"? In the simplest case, it means moving up and down on the Y axis (vertically). So, let's use the `translateY` function again and define keyframes:

```
@keyframes jump {
  from {
    transform: translateY(0px)
  }
  to {
```

```
    transform: translateY(-50px)
  }
}
```

So, initially the ball is at coordinates `(0,0)`, but then we move it up to `(0,-50)`. However, currently we might not have enough space for the ball to actually jump, so let's give it some margin:

```
.loader {
  margin-top: 50px;
}
```

Of course, we can do more. For instance, let's rotate this ball while it jumps:

```
@keyframes jump {
  from {
    transform: translateY(0px) rotate(0deg)
  }
  to {
    transform: translateY(-50px) rotate(360deg)
  }
}
```

Also, why don't we make it smaller? For that, let's utilize a `scale` function that changes the element's width and height using the given multipliers:

```
@keyframes jump {
  from {
    transform: translateY(0px) rotate(0deg) scale(1,1);
  }
  to {
    transform: translateY(-50px) rotate(360deg) scale(0.8,0.8);
  }
}
```

Note, by the way, that all functions should be listed for the `transform` property in both `from` and `to` sections, because otherwise the animation won't work properly!

Lastly, let's add some opacity for the ball:

```
@keyframes jump {
  from {
    transform: translateY(0px) rotate(0deg) scale(1,1);
    opacity: 1;
  }
```

```
  to {
    transform: translateY(-50px) rotate(360deg) scale(0.8,0.8);
    opacity: 0.8;
  }
}
```

Live Code

That's it! Our loader element is ready, and here's the final result: See the Pen CSS Transformations: Loader Ball.

Creating a "Spinner" Loader with SVG

We've already seen how to create a simple "jumping ball" loader with just a few lines of code. For a more complex effect, however, you can utilize SVGs which are defined with a set of special tags.

More on SVG

This article isn't aimed at explaining how SVG images work and how to create them, but SitePoint has a couple of articles on the topic.

As an example, let's create a spinner loader. Here's the corresponding SVG:

```
<svg class="spinner" viewBox="0 0 66 66" xmlns="http://www.w3.org/20
➥<!-- 1 -->

  <circle class="path spinner-border" cx="33" cy="33" r="31" stroke=
  ➥(#gradient)"></circle> <!-- 2 -->

  <linearGradient id="gradient"> <!-- 3 -->
    <stop offset="50%" stop-color="#000" stop-opacity="1"></stop>
    <stop offset="65%" stop-color="#000" stop-opacity=".5"></stop>
    <stop offset="100%" stop-color="#000" stop-opacity="0"></stop>
  </linearGradient>

  <circle class="path spinner-dot" cx="37" cy="3" r="2"></circle> <!

</svg>
```

Main things to note here:

1. Our canvas has a viewport of 66x66.
2. This defines the actual circle that's going to spin. Its center is located at `(33,33)` and the radius is 31px. We'll also have a stroke of 2px, which means `31 * 2 + 2 * 2 = 66`. `stroke="url(#gradient)` means that the color of the stroke is defined by an element with an ID of `#gradient`.
3. That's our gradient for the spinner's stroke. It has three breakpoints that set different opacity values, which is going to result in a pretty cool effect.
4. That's a dot that's going to be displayed on the spinner's stroke. It will look like a small "head".

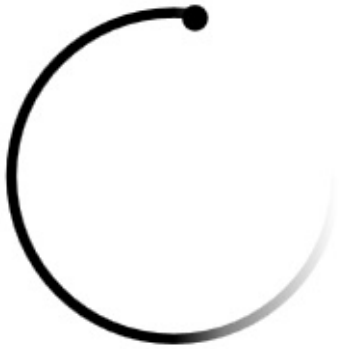
Now let's define some styling for the canvas and scale it up to 180x180:

```
.spinner {
  margin: 10px;
  width: 180px;
  height: 180px;
}
```

Now the `.spinner-border`, `.spinner-dot`, and some common styles for them:

```
.spinner-dot {
  stroke: #000;
  stroke-width: 1;
  fill: #000;
}

.spinner-border {
  fill: transparent;
  stroke-width: 2;
  width: 100%;
  height: 100%;
}

.path {
  stroke-dasharray: 170;
  stroke-dashoffset: 20;
}
```

Here's how our spinner looks at this stage. For now it doesn't spin, of course:



Now let’s make it spin, which effectively means rotating it by 360 degrees:

```
.spinner {
  // ...
  animation: rotate 2s linear infinite;
}

@keyframes rotate {
  to {
    transform: rotate(360deg);
  }
}
```

This is an infinite animation that takes 2 seconds to complete.

Also, we can achieve an interesting effect of a “snake trying to bite its tail” with a skew function. Remember that I’ve called that small dot a “head”? Why don’t we pretend that it is a snake’s head then? In order to make it look more realistic, we’ll skew it on the X axis:

```
.spinner-dot {
  // ...
  animation: skew 2s linear infinite alternate;
}

@keyframes skew {
  from {
    transform: skewX(10deg)
  }
  to {
    transform: skewX(40deg)
  }
}
```

Now it seems like the snake is really trying to drag to its tail:

Live Code

Here's the final result of our spinner: See the Pen CSS Transformations: Loaders with SVGs by SitePoint.

Creating a Flip Animation

The last example is a photo with a flip animation. When you hover over a photo, it flips and its description is shown. It can be useful for Instagram-like websites.

So, first of all, let's create a layout:

```
<section class="container">

  <figure class="photo">
    <img src="https://images.freeimages.c../Images/large-previews/53
    ➥wonders-1400924.jpg" class="front side">

    <figcaption class="back side">
      Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed d
      ➥ incididunt ut labore et dolore magna aliqua.
    </figcaption>
  </figure>

  <figure class="photo">
    <img src="https://images.freeimages.c../Images/large-previews/6d
    ➥cottage-1372381.jpg" class="front side">

    <figcaption class="back side">
      Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed d
      ➥ incididunt ut labore et dolore magna aliqua.
    </figcaption>
  </figure>
```

```
  <figure class="photo">
    <img src="https://images.freeimages.c../Images/large-previews/a0
    ➥ree-1382832.jpg" class="front side">

    <figcaption class="back side">
      Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed d
      ➥ incididunt ut labore et dolore magna aliqua.
    </figcaption>
  </figure>

</section>
```

Here we have a container with three photos. These photos are going to actually have two sides: front and back, just like a coin has heads and tails. The front contains the actual image, whereas the back (which is not visible initially) contains the description.

Style the container by giving it some margin:

```
.container {
  margin: 10px auto;
}
```

Each photo adjusts according to the viewport's width and floats to the left:

```
.photo {
  width: 22vw;
  height: 20vw;
  min-width: 130px;
  min-height: 100px;
  float: left;
  margin: 0 20px;
}
```

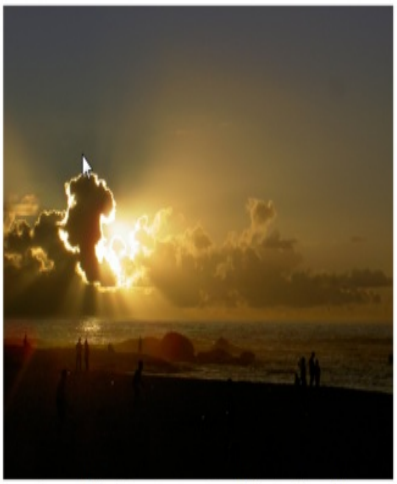
The images themselves should maintain the aspect ratio and try to fill the parent:

```
.photo img {
  object-fit: cover;
}
```

Each side should occupy 100% of the parent's width and height:

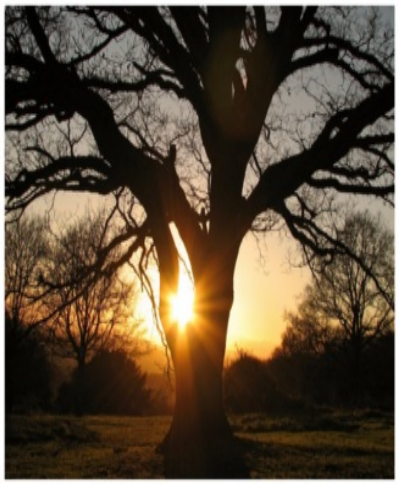
```
.side {
  width: 100%;
  height: 100%;
}
```

Now we have images with descriptions below which looks pretty generic:



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Next, what I'd like to do is place the description right above the image (on the Z axis). For that, let's adjust some `position` rules:

```
.photo {
  // ...
  position: relative;
}

.side {
  // ...
  position: absolute;
}
```

The text is now displayed right over the image:

Now it's time for some magic. I'd like children of the `.photo` block to be positioned in 3D with the help of `transform-style` property. This will allow us to achieve a feeling of perspective:

```
.photo {
  // ...
  transform-style: preserve-3d;
}
```

The backface should not be visible initially:

```
.side {
  // ...
  backface-visibility: hidden;
}
```

As for the `.back` itself, rotate it 180 degrees on the Y axis:

```
.back {
  transform: rotateY(180deg);
  text-align: center;
  background-color: #e0e0e0;
}
```

This will result in the description being hidden.

Next, define the hover animation:

```
.photo:hover {
  transform: rotateY(180deg);
}
```

Now when you hover over the container, the `.photo` will rotate and you'll be able to see its back with the description. You'll probably want this to happen more smoothly, so add a simple CSS transition:

```
.photo {
  // ...
  transition: transform 1s ease-in-out;
}
```

Live Code

Here's the final result: See the Pen CSS Transformations: 3D and flip.

A Word of Caution

Without any doubts, CSS transformations and animations are very powerful tools that can be used to create interesting and beautiful effects. However, it's important to be reasonable about their usage and not abuse them. Remember that you're creating websites for users and not for yourself (in most cases, anyway). Therefore, CSS should be utilized to introduce better user experience, rather than to show all the cool tricks you've learned so far.

For one thing, too many effects on the page distracts the users. Some visitors may have motion sickness or vestibular disorders, so they'll find it very hard to use a website with fast animations. On top of that, you should make sure that the page works with older browsers, because it may happen that some important element is hidden or inaccessible when the animations don't work.

Conclusion

In this article, we have seen how CSS transformations in conjunction with other techniques can be used to solve various design tasks. We've seen how to vertically align elements on the page, create scalable arrows, bouncing and

spinning loaders, and how to implement flip animation. Of course, you've learned only some techniques, but the only limit is your imagination.

Chapter 2: Variable Fonts: What They Are, and How to Use Them

by Claudio Ribeiro

In this article, we'll take a look at the exciting new possibilities surrounding variable fonts — now bundled with the OpenType scalable font format — which allows a single font to behave like multiple fonts.

How We Got Here

When HTML was created, fonts and styles were controlled exclusively by the settings of each web browser. In the mid ’90s, the first typefaces for screen-based media were created: Georgia and Verdana. These, together with the system fonts — Arial, Times New Roman, and Helvetica — were the only fonts available for web browsers (not exactly the only ones, but the ones we could find in every operating system).

With the evolution of web browsers, innovations like the `<font>` tag on Netscape Navigator and the first CSS specification allowed web pages to control what font was displayed. However, these fonts needed to be installed on the user's computer.

In 1998, the CSS working group proposed the support of the `@font-face` rule to allow any typeface to be rendered on web pages. IE4 implemented the technology but the distribution of fonts to every user's browser raised licensing and piracy issues.

The early 2000s saw the rise of image replacement techniques which substituted HTML content with styled-text images. Each piece of text had to be sliced in programs like Photoshop. This technique had the major advantage of allowing designers to use any typeface available without having to deal with font licensing.

In 2008, `@font-face` finally made a comeback when Apple Safari and Mozilla

Firefox implemented it. This came out of the necessity of providing a simple way for designers and developers to use custom fonts rather than inaccessible images.

It wasn't until the arrival of the CSS3 Fonts Module in 2012 that font downloading became viable. Once implemented, a font downloaded by a web page could only be used on that page and not copied to the operating system. Font downloading allowed remote fonts to be downloaded and used by the browser, meaning that web designers could now use fonts that were not installed on the user's computer. When web designers found the font they wished to use, they just needed to include the font file on the web server, and it would be automatically downloaded to the user when needed. These fonts were referenced using the `@font-face` rule.

To use the `@font-face` rule we have to define a font name and point to the font file:

```
@font-face {
  font-family: Avenir Next Variable;
  src: url(AvenirNext_Variable.woff);
}
```

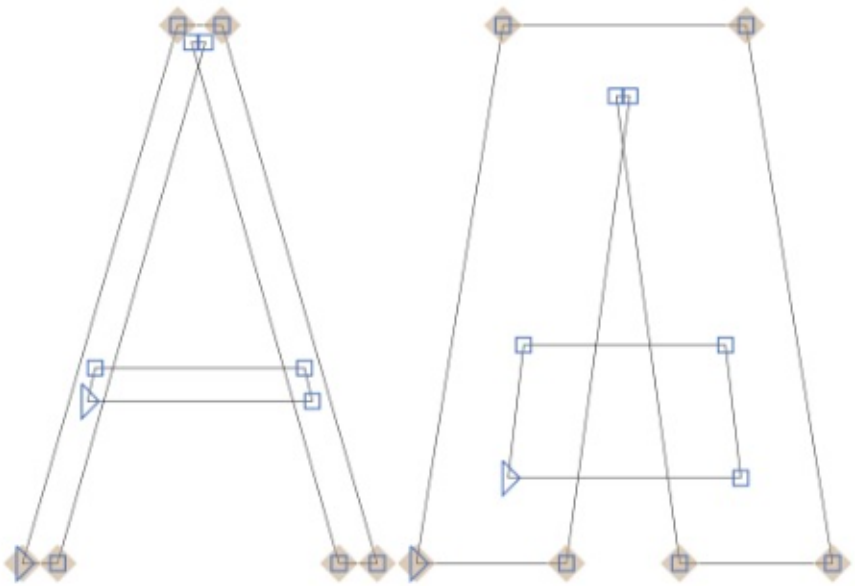
The font file can be one of five different formats: TTF, WOFF, WOFF2, SVG or EOT. Each has its own advantages and disadvantages. Putting it simply, EOT was created by Microsoft and only is supported by Internet Explorer. TTF is the basic type font created by Microsoft and Apple, and it works almost perfectly everywhere. SVG is based on image replacement techniques and is only suitable for the Web. Finally, WOFF and WOFF2 were also created exclusively for the Web and are basically TTF files with extra compression.

Variable Fonts

Version 1.8 of OpenType (the computer scalable font format) was released in 2016. A brand new technology shipped with it: OpenType font variations, also known as **variable fonts**.

This technology allows a single font to behave like multiple fonts. It's done by defining variations within the font. These variations come from the fact that each character only has one outline. The points that construct this outline have instructions on how they should behave. It’s not necessary to define multiple

font weights because they can be interpolated between very narrow and very wide definitions. This also makes it possible to generate styles in between — for example, semi-bold and bold. These variations may act along one or more axes of the font. On the image below, we have an example of this outline interpolation on the letter A.



Why are Variable Fonts Relevant?

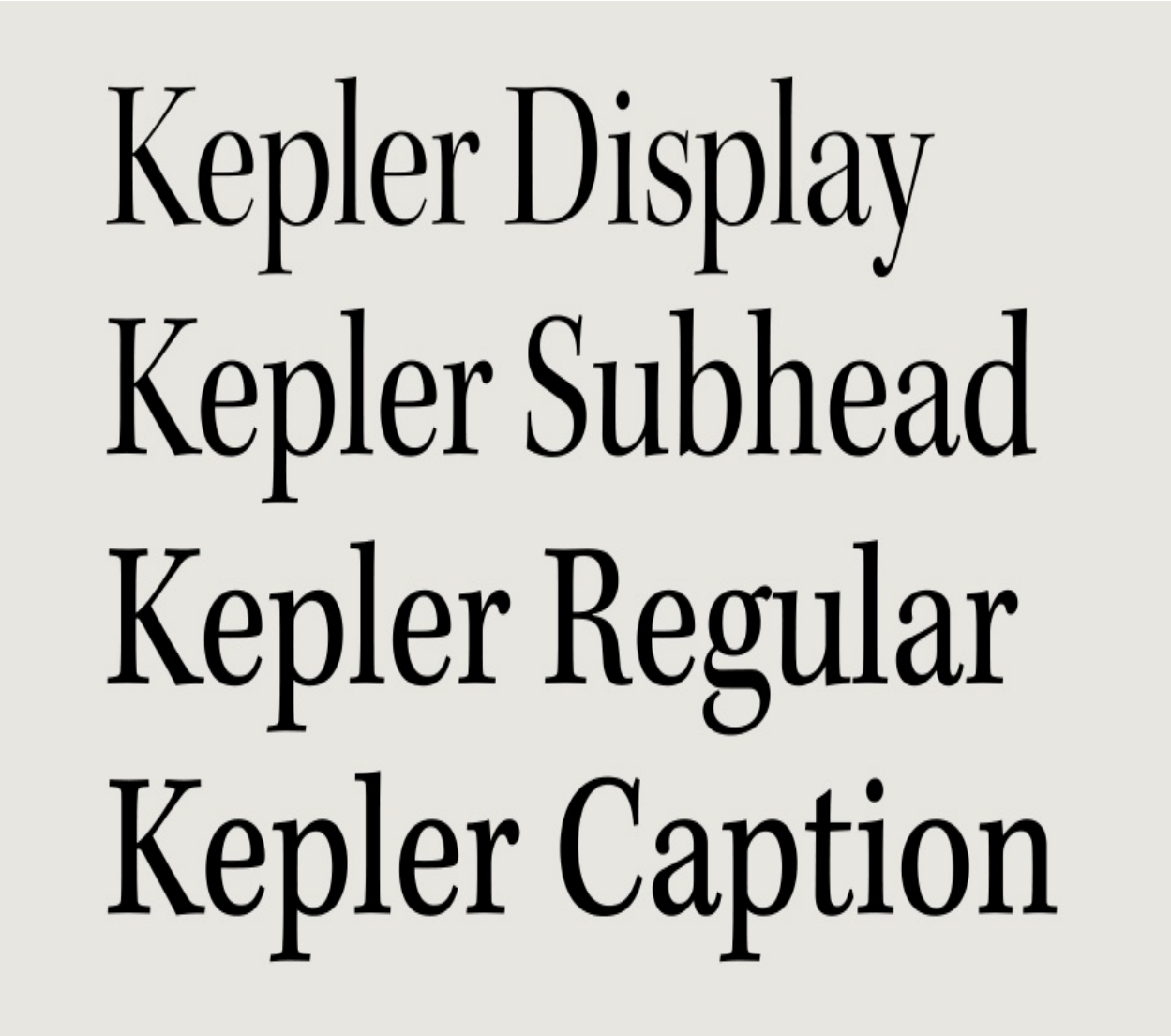
Variable fonts can bring both simplicity to our font structure and performance improvements. Take for example a situation where our website needs five font styles. It would be significantly smaller and faster to provide a single variable font capable of rendering those five styles than to have to load five different font files.

Using Variable Fonts

There are currently two different ways to use variable fonts. First, we'll look at the modern way of implementing those. The CSS specification strongly prefers using `font-optical-sizing`, `font-style`, `font-weight` and `font-stretch` for controlling any of the standard axes.

`font-optical-sizing`

This property allows developers to control whether or not browsers render text with slightly different visual representations to optimize viewing at different sizes. It can take the value `none`, for when the browser cannot modify the shape of glyphs, or `auto` for when it can. On a browser that supports `font-optical-sizing`, a font where the value is set to `auto` can vary like the font in the image below:



With the value set to `none` there would be no variation to the font.

`font-style`

This property specifies whether a font should be styled with a normal, italic, or oblique face from its font family. It can take the `normal`, `italic` or `oblique`

values.

`font-weight`

This property specifies the weight (or boldness) of the font. One thing to note is that, with normal fonts, named instances can be defined. For example, `bold` is the same as `font-weight: 700` or `extra-light` is the same as `font-weight: 200`. The `font-weight` property can also be any number between 1 and 1000, but when using variable fonts, due to the interpolarity, we can have a much finer granularity. For example, a value like `font-weight: 200.01` is now possible.

`font-stretch`

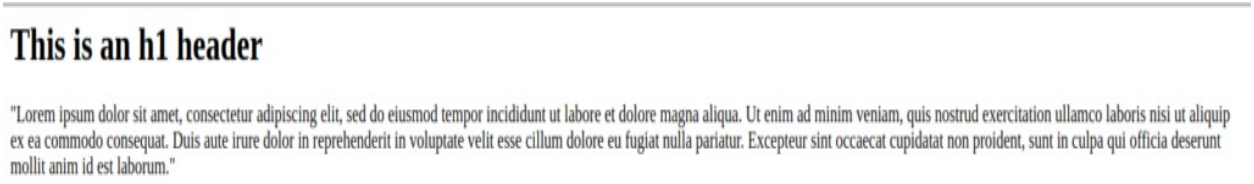
This property selects a normal, condensed, or expanded face from a font. Just like the `font-weight` property, it can be a named instance like `extra-condensed` or `normal` or a percentage between 0% and 100%. Also, named instances will map to known percentages. For example, `extra-condensed` will map to 62.5%.

For this example, I created a very simple page with a single `<h1>` heading and `<p>` paragraph.

Live Code

See the Pen Variable Fonts HTML.

Currently, our unstyled webpage looks like this:



To use a variable font, we must find a suitable file. The v-fonts project provides a font repository where we can search and experiment with all type of variable fonts. I decided to use the `AvenirNext` font and link it from its official GitHub

page.

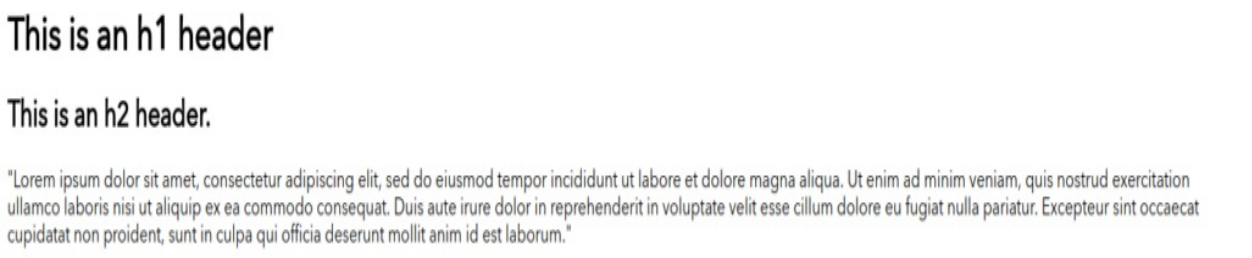
Then we need to create a CSS file to load this new font:

```
@font-face {
  font-family: 'Avenir Next Variable';
  src: url('AvenirNext_Variable.ttf') format('truetype');
}

body {
  font-family: 'Avenir Next Variable', sans-serif;
}
```

Live Code

See the Pen Loaded variable font.

Due to browser support issues, this example will only apply font variations in Safari and Chrome.



With our font loaded, we can now use the `font-weight` property to manipulate the weight axis of our variable font.

```
h1 {
  font-family: 'Avenir Next Variable';
  font-weight: 430;
}
```

Live Code

See the Pen SourceSans variable font.

Most of the time we'll also need two different font files: one for italic and one

for regular fonts (roman). This happens because the construction of these fonts can be radically different.

Using font-variation-settings

The second way of using variable fonts is by using the `font-variation-settings` CSS property. This property was introduced to provide control over OpenType or TrueType font variations, by specifying the four-letter axis names of the features you want to vary along with their variation values. Currently, we have access to the following aspects of the font:

- **wght** — weight, which is identical to the `font-weight` CSS property. The value can be anything from 1 to 999.
- **wdth** — width, which is identical to by the `font-stretch` CSS property. It can take a keyword or a percentage value. This axis is normally defined by the font designer to expand or condense elegantly.
- **opsz** — optical sizing, which can be turned on and off using the `font-optical-sizing` property.
- **ital** — italicization, which is controlled by the `font-style` CSS property when is set to `italic`.
- **slnt** — slant, which is identical to the `font-style` CSS property when it's set to `oblique`. It will default to a 20 degree slant, but it can also accept a specified degree between -90deg and 90deg.

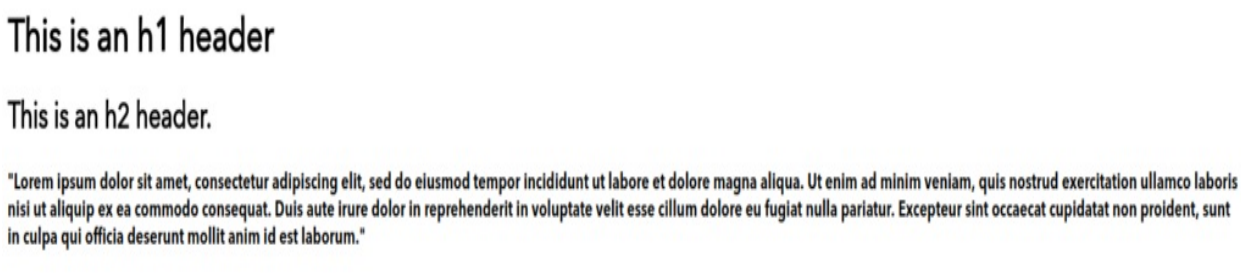
According to the specification, this property is a low-level feature designed to handle special cases where no other way to enable or access an OpenType font feature exists. Because of that, we should try to use `@font-face` instead.

Using the same webpage and font as above, let's try to set the weight and width on our font using the low-level controller:

```
p {
  font-variation-settings: 'wght' 630, 'wdth' 88;
}
```

Live Code

See the Pen Variable fonts 1.



The declaration is equivalent to the following CSS declaration:

```
p {
  font-weight: 630;
  font-stretch: 88;
}
```

Performance

Performance is a key advantage of variable fonts. The `AvenirNext_Variable.ttf` font file is only 89kB but creates a range of weights. A comparable standard font may have a smaller file but would only support one weight and style. Further options require additional files and raise the page weight accordingly.

But we can go even further. When we talked about font formats, we said that WOFF2 files are essentially TTF files with extra compression. WOFF2 files are smaller because they use custom preprocessing and compression algorithms to deliver ~30% file-size reduction over other formats.

Google offers a command line tool that will compress our file converting it to a `woff2` format.

On the tool's official GitHub page, we can find all the information about it. To install it on a Unix environment, we can use the following commands:

```
git clone --recursive https://github.com/google/woff2.git
cd woff2
make clean all
```

After installing it, we can use it to compress our variable font file by using:

```
woff2_compress AvenirNext_Variable.ttf
```

And we end up with a 42kb file, which halved the file size. To use this file, we just need to modify the sourced file and its format to accommodate this new file:

```
src: url('AvenirNext_Variable.woff2') format('woff2');
```

We now have a single 42Kb file which could be used for all the font weights on our page. The only downside to the `woff2` format is that it's not supported by Internet Explorer.

Serving Different Files for Different Browsers

While some modern browsers already support variable fonts (Firefox developer editions have some level of support, Chrome 62, Chrome Android, Safari iOS, and Safari), there might be the case where we find one that doesn't.

To get around this, we'll need to either serve non-variable for those browsers or use an operating system font fallback.

Browsers that support variable fonts will download the file marked as `format('woff2-variations')`, while browsers that don't will download the single style font marked as `format('ttf')`. This is possible because we can repeat references to variables in each rule. If the first fails, the second will be loaded. Just like the following:

```
@font-face {
  font-family: 'Avenir Next Variable';
  src: url('AvenirNext_Variable.woff2') format('woff2-variations');
  src: url('AvenirNextLTPro-Bold.otf') format('truetype');
}

html {
  font-family: 'Avenir Next Variable', sans-serif;
}

h1 {
  font-family: 'Avenir Next Variable';
  font-weight: 430;
}

h2 {
  font-family: 'Avenir Next Variable';
  font-weight: 630;
```

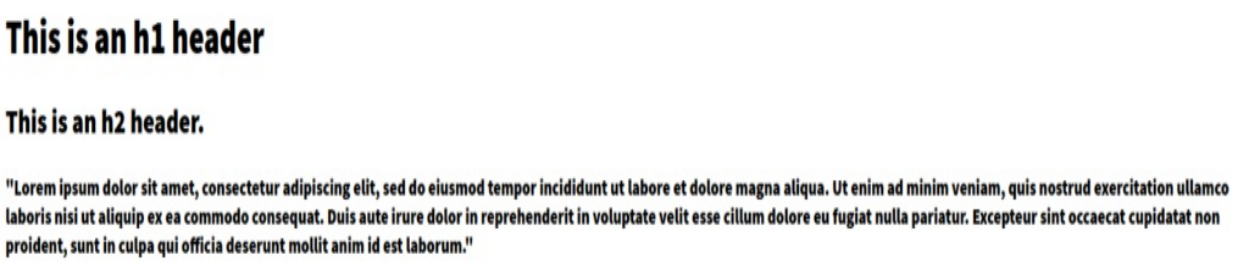
```
}
```

The next example uses a different file format from the one we're working with, but uses the same principle:

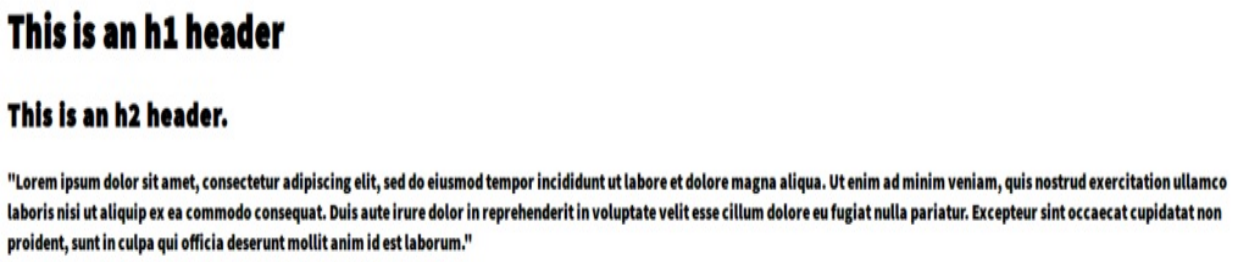
Live Code

See the Pen Multiple fonts.

If we check the result on a browser that supports variable fonts, like Chrome, we can see the following:



On a browser that doesn't support variable fonts, like Firefox, the second font will be loaded and the result will look like this:



But if we still have to serve non-variable fonts to browsers that don't support variable ones, don't we lose all the performance we just gained with the variable font? If a browser can only load the standard font, we lose the performance and rendering benefits of variable fonts. In those situations, it may be preferable to fallback to a comparable operating system font rather than substitute it with many fixed fonts.

Conclusion

Despite being available for several years now, variable fonts are still in their infancy. Browser support is scarce and there are few fonts to choose from. However, the potential is enormous, and variable fonts will permit better performance while simplifying development. For example, SitePoint's own front page currently loads eight font files with a total of 273kB. Variable fonts could reduce this dependency and cut page weight further.

Chapter 3: Scroll Snap in CSS: Controlling Scroll Action

by Tiffany B. Brown

The following is a short extract from Tiffany's new book, CSS Master, 2nd Edition.

As the web platform grows, it has also gained features that mimic native applications. One such feature is the CSS Scroll Snap Module. Scroll snap lets developers define the distance an interface should travel during a scroll action. You might use it to build slide shows or paged interfaces—features that currently require JavaScript and expensive DOM operations.

Scroll snap as a feature has undergone a good deal of change. An earlier, 2013 version of the specification — called Scroll Snap *Points* at the time — defined a coordinates-and-pixels-based approach to specifying scroll distance. This version of the specification was implemented in Microsoft Edge, Internet Explorer 11, and Firefox.

Chrome 69+ and Safari 11+ implement the latest version of the specification, which uses a box alignment model. That's what we'll focus on in this section.

Watch Out!

Many of the scroll snap tutorials currently floating around the web are based on the earlier CSS Scroll Snap **Points** specification. The presence of the word "points" in the title is one sign that the tutorial may rely on the old specification. A more reliable indicator, however, is the presence of the `scroll-snap-points-x` or `scroll-snap-points-y` properties.

Since scroll snap is really well-suited to slide show layouts, that's what we'll build. Here's our markup.

```
<div class="slideshow">
  <img src="avocado-and-bacon-salad.jpg" alt="avocado and bacon sala
```

```
  <img src="salad-eggs-and-scallops.jpg" alt="salad topped with hard
  ➥ and seared scallops">
  <img src="seafood-and-noodles.jpg" alt="seafood stew over noodles"
  <img src="grilled-salmon-and-side-salad.jpg" alt="grilled salmon s
  ➥ avocado and side salad">
  <img src="avocado-toast-with-egg.jpg" alt="avocado toast with egg"
</div>
```

That's all we need. We don't need to have an outer wrapping element with and an inner sliding container. We also don't need any JavaScript.

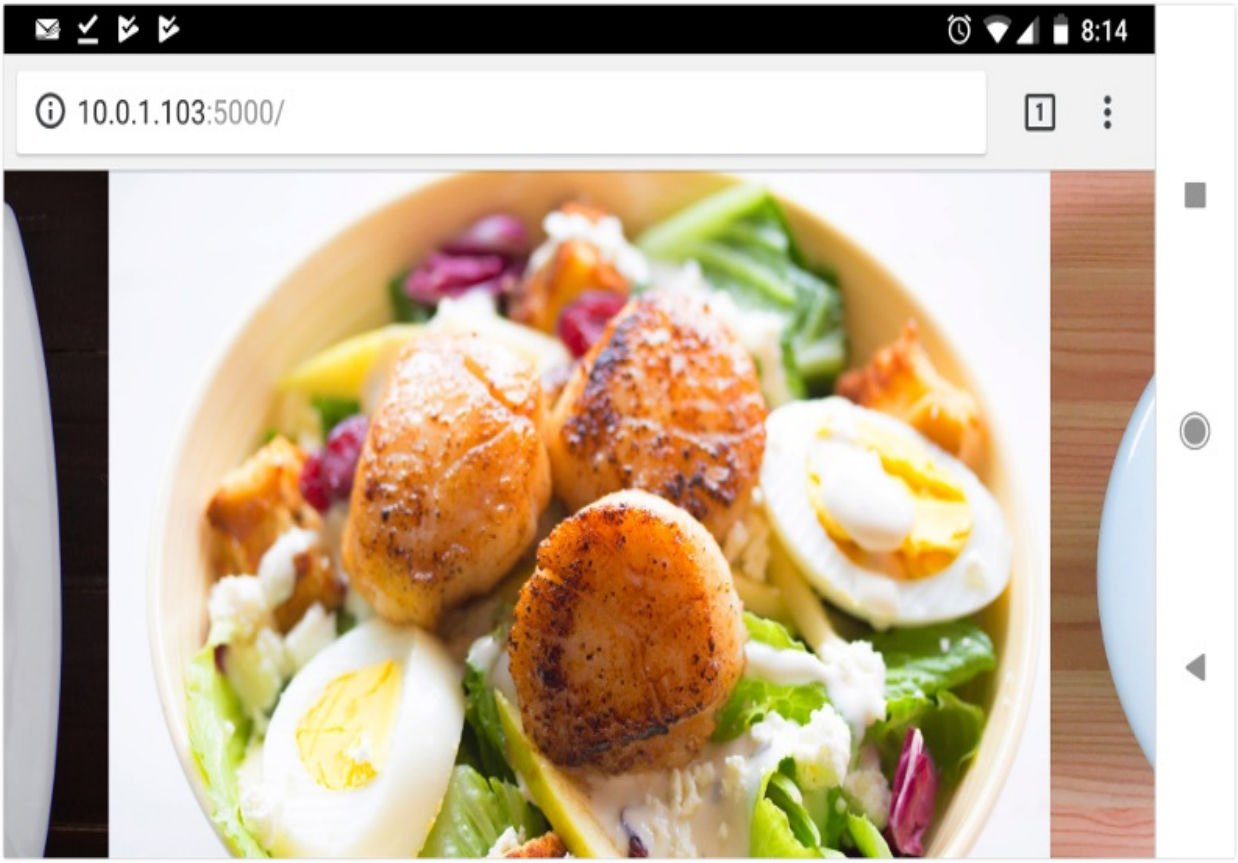
Now for our CSS:

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

html, body {
    padding: 0;
    margin: 0;
}

.slideshow {
  scroll-snap-type: x mandatory; /* Indicates scroll axis and behavi
  overflow-x: auto;              /* Should be either `scroll` or `au
  display: flex;
  height: 100vh;
}

.slideshow img {
  width: 100vw;
  height: 100vh;
  scroll-snap-align: center;
}
```

Adding `scroll-snap-type` to `.slideshow` creates a scroll container. The value for this property, `x mandatory` describes the direction in which we'd like to scroll, and the *scroll snap strictness*. In this case, the `mandatory` value tells the browser that it *must* snap to a snap position when there is no active scroll operation. Using `display: flex` just ensures that all of our images stack horizontally.

Now the other property we need is `scroll-snap-align`. This property indicates how to align each image's scroll snap area within the scroll container's snap port. It accepts three values: `start`, `end`, and `center`. In this case, we've used the

`center` which means that each image will be centered within the viewport as shown below.



For a more comprehensive look at Scroll Snap, read *Well-Controlled Scrolling with CSS Scroll Snap* from Google Web Fundamentals guide.

Chapter 4: Real World Use of CSS with SVG

by Craig Buckler

SVG is a lightweight vector image format that's used to display a variety of graphics on the Web and other environments with support for interactivity and animation. In this article, we'll explore the various ways to use CSS with SVG, and ways to include SVGs in a web page and manipulate them.

The Scalable Vector Graphic (SVG) format has been an open standard since 1999, but browser usage became practical in 2011 following the release of Internet Explorer 9. Today, SVG is well supported across all browsers, although more advanced features can vary.

SVG Benefits

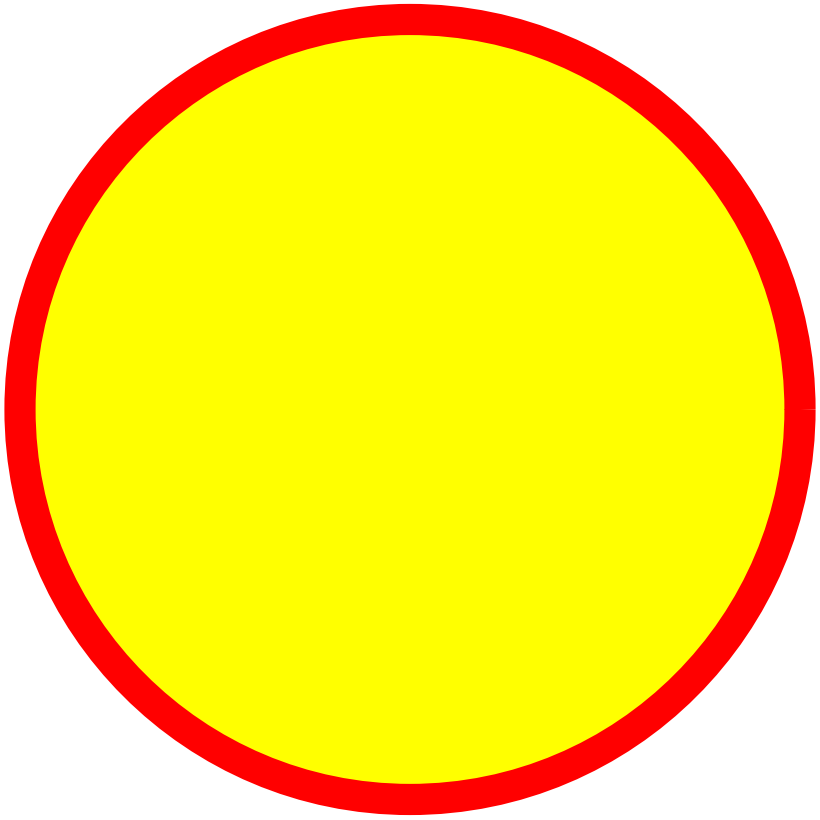
Bitmap images such as PNGs, JPGs and GIFs define the color of individual pixels. An 100x100 PNG image requires 10,000 pixels. Each pixel requires four bytes for red, green, blue and transparency so the resulting file is 40,000 bytes (plus a little more for meta data). Compression is applied to reduce the file size; PNG and GIF use ZIP-like lossless compression while JPG is lossy and removes less noticeable details.

Bitmaps are ideal for photographs and more complex images, but definition is lost as the images are enlarged.

SVGs are vector images defined in XML. Points, lines, curves, paths, ellipses, rectangles, text, etc. are *drawn* on an SVG canvas. For example:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 800 600">
  <circle cx="400" cy="300" r="250" stroke-width="20" stroke="#f00"
</svg>
```

The `viewBox` defines a co-ordinate space. In this example, an 800x600 area starting at position 0,0 has a yellow circle with a red border drawn in the center:



The benefits of vectors over bitmaps:

- the SVG above uses less than 150 bytes, which is considerably smaller than an equivalent PNG or JPG
- SVG backgrounds are transparent by default
- the image can scale to any size without losing quality
- SVG code/elements can be easily generated and manipulated on the server or browser
- in terms of accessibility and SEO, text and drawing elements are machine and human-readable.

SVG is ideal for logos, charts, icons, and simpler diagrams. Only photographs are generally impractical, although SVGs have been used for lazy-loading placeholders.

SVG Tools

It's useful to understand the basics of SVG drawing, but you'll soon want to create more complex shapes with an editor that can generate the code. Options include:

- Adobe Illustrator (commercial)
- Affinity Designer (commercial)
- Sketch (commercial)
- Inkscape (open source)
- Gravit Designer (free, online)
- Vectr (free, online)
- SVG-Edit (open source, online)
- Boxy SVG (free, app, online but Blink browsers only)
- Vecteezy - (free, online but Blink browsers only)
- SVG charting libraries — which generally create SVG charts using data passed to JavaScript functions.

Each has different strengths, and you'll get differing results for seemingly identical images. In general, more complex images require more complex software.

The resulting code can usually be simplified and minimized further using SVGO (plugins are available for most build tools) or Jake Archibold's SVGOMG interactive tool.

SVGs as Static Images

When used within an HTML `<img>` tag or CSS `background-url`, SVGs act identically to bitmaps:

```
<!-- HTML image -->
<img src="myimage.svg" alt="a vector graphic" />
```

```
/* CSS background */
.myelement {
  background-image: url('mybackground.svg');
}
```

The browser will disable any scripts, links, and other interactive features embedded into the SVG file. You can manipulate an SVG using CSS in an identical way to other images using `transform`, `filters`, etc. The results of

using CSS with SVG are often superior to bitmaps, because SVGs can be infinitely scaled.

CSS Inlined SVG Backgrounds

An SVG can be inlined directly in CSS code as a background image. This can be ideal for smaller, reusable icons and avoids additional HTTP requests. For example:

```
.mysvgbackground {
  background: url('data:image/svg+xml;utf8,<svg xmlns="http://www.w3
  viewBox="0 0 800 600"><circle cx="400" cy="300" r="50" stroke-widt
  fill="#ff0" /></svg>') center center no-repeat;
}
```

Standard UTF-8 encoding (rather than base64) can be used, so it's easy to edit the SVG image if necessary.

CSS with SVG: Responsive SVG Images

When creating a responsive website, it's normally practical to omit `<img>` `width` and `height` attributes and allow an image to size to its maximum width via CSS:

```
img {
  display: block;
  max-width: 100%;
}
```

However, an SVG used as an image has no implicit dimensions. You might discover the `max-width` is calculated as zero and the image disappears entirely. To fix the problem, ensure a default `width` and `height` is defined in the `<svg>` tag:

```
<svg xmlns="http://www.w3.org/2000/svg" width="400" height="300">
```

Don't Add Diemensions

The dimensions should not be added to inlined SVGs, as we'll discover in the next section …

HTML-Inlined SVG Images

SVGs can be placed directly into the HTML. When this is done, they become part of the DOM and can be manipulated with CSS and JavaScript:

```
<p>SVG is inlined directly into the HTML:</p>

<svg id="invader" xmlns="http://www.w3.org/2000/svg"
viewBox="35.4 35.4 195.8 141.8">
  <path d="M70.9 35.4v17.8h17.7V35.4H70.9zm17.7 17.8v17.7H70.9v17.7H
  ➥4V124h17.8v17.7h17.7v17.7h17.7v-17.7h88.6v17.7h17.7v-17.7h17.7V12
  ➥7v35.4h-17.7V70.9h-17.7V53.2h-17.8v17.7H106.3V53.2H88.6zm88.6 0h1
  ➥7v17.8zm17.7 106.2v17.8h17.7v-17.8h-17.7zm-124 0H53.2v17.8h17.7v-
  ➥8h17.7v17.7H88.6V88.6zm70.8 0h17.8v17.7h-17.8V88.6z"/>
  <path d="M319 35.4v17.8h17.6V35.4H319zm17.6 17.8v17.7H319v17.7h-17
  ➥7V159.4h17.7V124h17.7v35.4h17.7v-17.7H425.2v17.7h17.7V124h17.7v35
  ➥3h-17.7V88.6H443V70.9h-17.7V53.2h-17.7v17.7h-53.2V53.2h-17.7zm88.
  ➥4h-17.7v17.8zm0 106.2h-35.5v17.8h35.5v-17.8zm-88.6 0v17.8h35.5v-1
  ➥8h17.7v17.7h-17.7V88.6zm70.9 0h17.7v17.7h-17.7V88.6z"/>
</svg>

<p>The SVG is now part of the DOM.</p>
```

No `width` or `height` attributes are defined for the SVG, so it can size to the containing element or be directly sized using CSS:

```
#invader {
  display: block;
  width: 200px;
}

#invader path {
  stroke-width: 0;
  fill: #080;
}
```

Live Code

See the Pen HTML-Inlined SVG.

SVG elements such as paths, circles, rectangles etc. can be targeted by CSS selectors and have the styling modified using standard SVG attributes as CSS properties. For example:

```
/* CSS styling for all SVG circles */
circle {
  stroke-width: 20;
  stroke: #f00;
  fill: #ff0;
}
```

This overrides any attributes defined within the SVG because the CSS has a higher specificity. SVG CSS styling offers several benefits:

- attribute-based styling can be removed from the SVG entirely to reduce the page weight
- CSS styling can be reused across any number of SVGs on any number of pages
- the whole SVG or individual elements of the image can have CSS effects applied using `:hover`, `transition`, `animation` etc.

SVG Sprites

A single SVG file can contain any number of separate images. For example, this `folders.svg` file contains folder icons generated by IcoMoon. Each is contained within a separate `<symbol>` container with an ID which can be targeted:

```
<svg xmlns="http://www.w3.org/2000/svg">
```

```
  <defs>
    <symbol id="icon-folder" viewBox="0 0 32 32">
      <title>folder</title>
      <path d="M14 4l4 4h14v22h-32v-26z"></path>
    </symbol>
    <symbol id="icon-folder-open" viewBox="0 0 32 32">
      <title>open</title>
      <path d="M26 30l6-16h-26l-6 16zM4 12l-4 18v-26h9l4 4h13v4z">
      </path>
    </symbol>
    <symbol id="icon-folder-plus" viewBox="0 0 32 32">
      <title>plus</title>
      <path d="M18 8l-4-4h-14v26h32v-22h-14zM22 22h-4v4h-4v-4h-4v-4h
      </path>
    </symbol>
    <symbol id="icon-folder-minus" viewBox="0 0 32 32">
      <title>minus</title>
      <path d="M18 8l-4-4h-14v26h32v-22h-14zM22 22h-12v-4h12v4z"></p
    </symbol>
    <symbol id="icon-folder-download" viewBox="0 0 32 32">
      <title>download</title>
      <path d="M18 8l-4-4h-14v26h32v-22h-14zM16 27l-7-7h5v-8h4v8h5l-
    </symbol>
    <symbol id="icon-folder-upload" viewBox="0 0 32 32">
      <title>upload</title>
      <path d="M18 8l-4-4h-14v26h32v-22h-14zM16 15l7 7h-5v8h-4v-8h-5
    </symbol>
  </defs>
</svg>
```

The SVG file can be referenced as an external, cached resource in an HTML page. For example, to show the folder icon at `#icon-folder`:

```
<svg class="folder" viewBox="0 0 100 100">
  <use xlink:href="folders.svg#icon-folder"></use>
</svg>
```

and style it with CSS:

```
svg.folder { fill: #f7d674; }
```

The method has a couple of drawbacks:

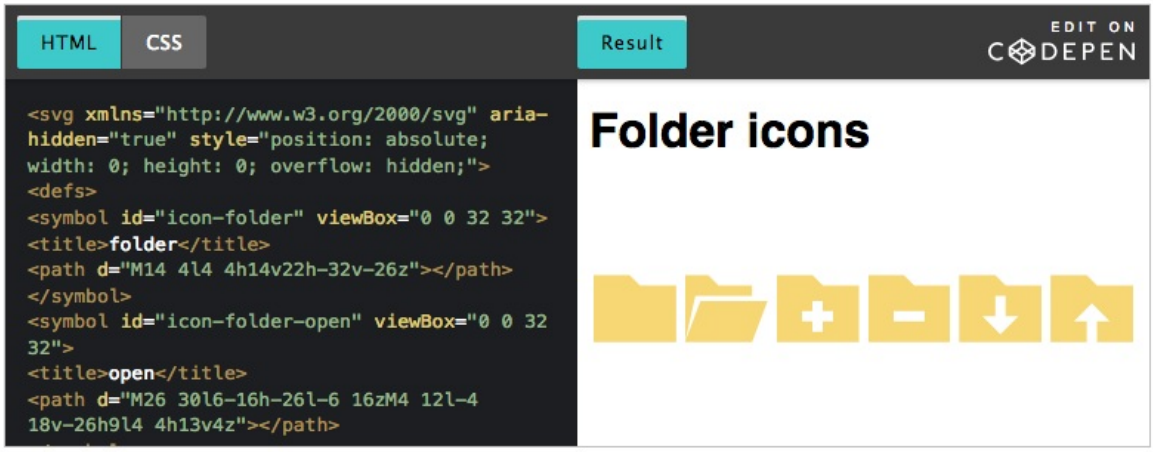
1. It fails in IE9+.
2. CSS styling only applies to the `<svg>` element containing the `<use>`. The `fill` here makes every element of the icon the same color.

To solve these issues, the SVG sprite can be embedded within page HTML, then hidden using `display: none` or similar techniques. An individual icon can be placed by referencing the ID:

```
<svg><use xlink:href="#icon-folder"></use></svg>
```

Live Code

See the Pen SVG sprites.



This works in all modern browsers from IE9+ and it becomes possible to style individual elements within each icon using CSS.

Unfortunately, the SVG set is no longer cached and must be reproduced on every page where an icon is required. The solution *(to this solution!)* is to load the SVG using Ajax — which is then cached — and inject it into the page. The IcoMoon download provides a JavaScript library, or you could use SVG for Everybody.

SVG Effects on HTML Content

SVG has long supported:

- **masks**: altering the visibility of parts of an element
- **clipping**: removing segments of an element so a standard regular box becomes any other shape
- **filters**: graphical effects such as blurring, brightness, shadows, etc.

These effects have been ported to the CSS `mask`, `clip-path`, and `filter` properties. However, it's still possible to target an SVG selector:

```
/* CSS */
.myelement {
  clip-path: url(#clip);
}
```

This references an effect within an HTML-embedded SVG:

```
<svg xmlns="http://www.w3.org/2000/svg" aria-hidden="true"
    style="position: absolute; width: 0; height: 0; overflow: hidden
  <defs>
    <clipPath id="clip">
      <text x="0" y="200" font-family="Arial" font-size="10em" font-
      Text Clip</text>
    </clipPath>
  </defs>
</svg>
```

to produce effects such as clipped text with an image or gradient background:

Live Code

See the Pen SVG clipping.

Portable SVGs

Finally, standalone SVGs can contain CSS, JavaScript and base64-encoded fonts or bitmap images! Anything outside the realms of XML should be contained

within `<![CDATA[ ... ]]>` sections.

Consider the following `invader.svg` file. It defines CSS styling with hover effects and a JavaScript animation which modifies the `viewBox` state:

```
<svg id="invader" xmlns="http://www.w3.org/2000/svg"
    viewBox="35.4 35.4 195.8 141.8">
  <!-- invader images: https://github.com/rohieb/space-invaders !-->
  <style>/* <![CDATA[ */
    path {
      stroke-width: 0;
      fill: #080;
    }

    path:hover {
      fill: #c00;
    }
  /* ]]> */</style>
  <path d="M70.9 35.4v17.8h17.7V35.4H70.9zm17.7 17.8v17.7H70.9v17.7H
  ➥4V124h17.8v17.7h17.7v17.7h17.7v-17.7h88.6v17.7h17.7v-17.7h17.7V12
  ➥7v35.4h-17.7V70.9h-17.7V53.2h-17.8v17.7H106.3V53.2H88.6zm88.6 0h1
  ➥7v17.8zm17.7 106.2v17.8h17.7v-17.8h-17.7zm-124 0H53.2v17.8h17.7v-
  ➥8h17.7v17.7H88.6V88.6zm70.8 0h17.8v17.7h-17.8V88.6z"/>
  <path d="M319 35.4v17.8h17.6V35.4H319zm17.6 17.8v17.7H319v17.7h-17
  ➥7V159.4h17.7V124h17.7v35.4h17.7v-17.7H425.2v17.7h17.7V124h17.7v35
  ➥3h-17.7V88.6H443V70.9h-17.7V53.2h-17.7v17.7h-53.2V53.2h-17.7zm88.
  ➥4h-17.7v17.8zm0 106.2h-35.5v17.8h35.5v-17.8zm-88.6 0v17.8h35.5v-1
  ➥8h17.7v17.7h-17.7V88.6zm70.9 0h17.7v17.7h-17.7V88.6z"/>
  <script>/* <![CDATA[ */
    const
      viewX = [35.4, 283.6],
      animationDelay = 500,
      invader = document.getElementById('invader');

    let frame = 0;

    setInterval(() => {
      frame = ++frame % 2;
      invader.viewBox.baseVal.x = viewX[frame];
    }, animationDelay);

  /* ]]> */</script>
</svg>
```

When referenced in an HTML `<img>` or CSS background, the SVG becomes a static image of the initial state (in essence, the first animation *frame*).

However, open the image in its own browser tab and all the effects will return.

This could be useful for distributing images, demonstrations or small documents which require a level of embedded interactivity.

Sophisticated SVGs

SVGs offer a wide range of technical possibilities both within and outside web pages. When combining CSS with SVG, it becomes possible to style and animate the whole image or individual elements in interesting ways.

This article describes ways to manipulate SVG images, but they are regularly used for smaller visual enhancements such as:

- form focus highlights and validation
- turning a hamburger menu into a an `X` close icon
- creating lava-lamp-like morphing.

Despite the age of SVG technology, web developers are still discovering ways to transform boring block-based pages with subtle effects through using CSS with SVG.

Chapter 5: CSS and PWAs: Some Tips for Building Progressive Web Apps

by David Attard

In recent years we've seen a number of major shifts in the online experience, mostly coming from the proliferation of mobile devices. The evolution of the Web has taken us from single versions of a website, to desktop versus mobile versions, to responsive sites that adapt according to screen size, then to native mobile apps, which either recreate the desktop experience as a native app, or act as a gateway to the responsive version.

The latest iteration of all of this is the progressive web app (PWA). A PWA is a software platform that aims to combine the best of both the Web and the native experience for website/app users.

In this article on CSS and PWAs, we're going to discuss a number of techniques that can be used when creating the CSS required for the development of PWAs.

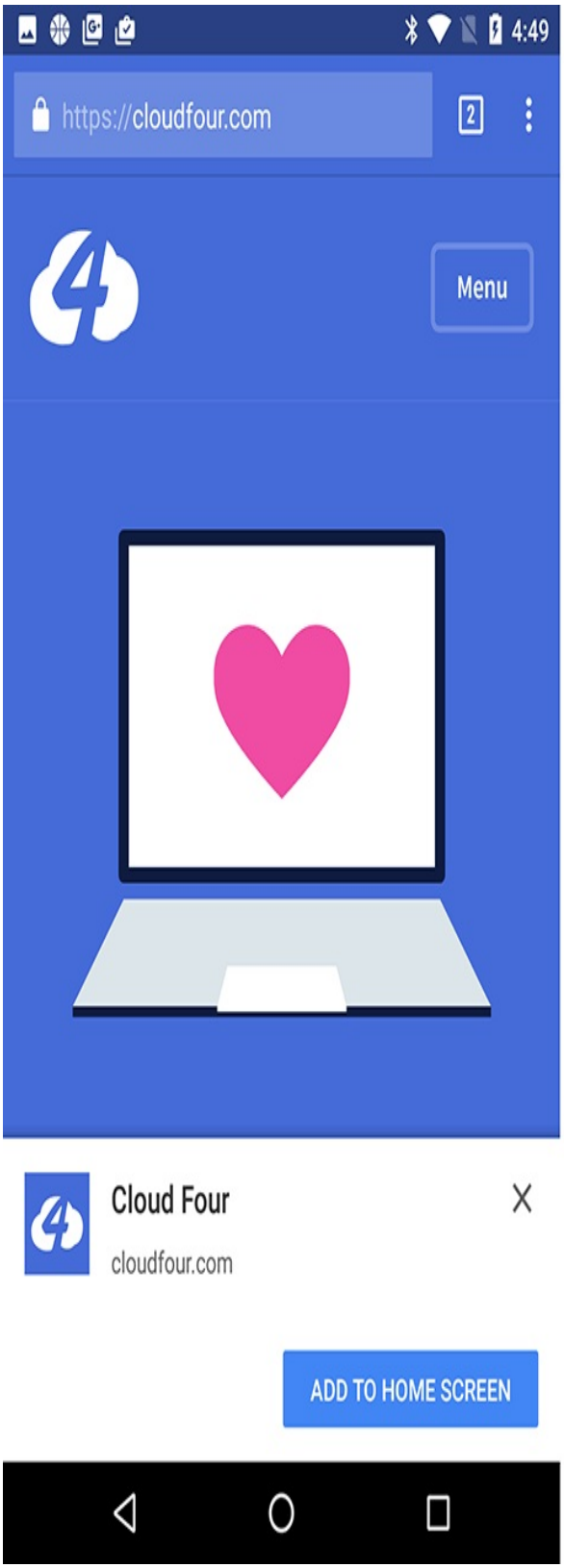
What are PWAs?

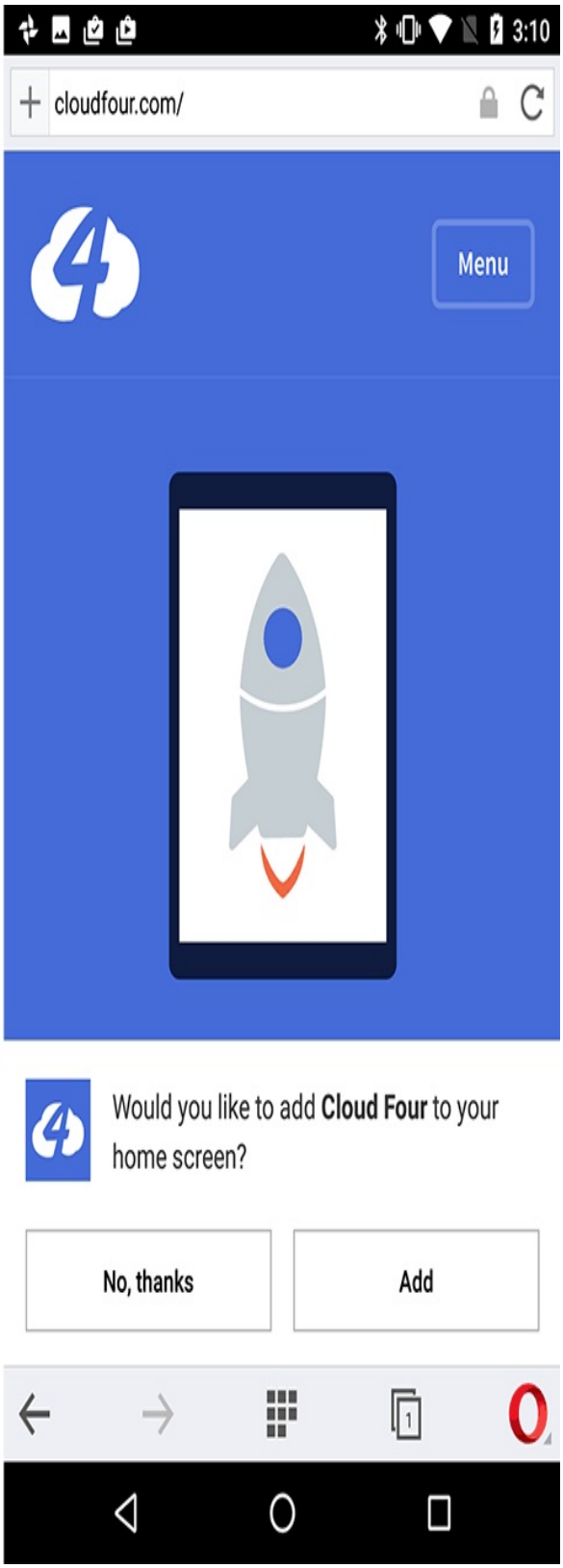
There are three main features of a PWA. As you'll see, what makes a web app progressive is the "fixing" of problems typically associated with web apps, by adopting some of the techniques used by native apps to resolve these issues.

1. **Reliable**. A PWA should reliably load like a native app (no matter the state of the network). This is contrary to a web page, which typically does not load if the device is disconnected from the network.

2. **Fast**. The performance of a PWA should be independent of such things as geography, network speed, load or other factors that are beyond the control of the end user.

3. **Engaging**. PWAs should mimic the native app's immersive, full-screen experience without requiring the need of an app store, even supporting such features as push notifications.





There are other features PWA features, but for now, we'll keep to the most

important ones described above.

Google has been at the forefront of pushing these kind of apps, but the adoption of PWAs has been picking up with vendors and plenty of other companies on the Web helping the adoption and embracing the concept of PWAs.

The following are comments from Itai Sadan, CEO of Duda, who was present at Cloudfest 2018:

> Progressive web apps represent the next great leap in the evolution of web design and online presence management … they take the best aspects of native apps, such as a feature-rich experience and the ability to be accessed offline, and incorporate them into responsive websites. This creates an incredible web experience for users without the need to download anything onto their device.
>
> Anyone providing web design services to clients is going to want to offer PWAs because over time, just like with mobile and responsive web design, it will become the industry standard.

What is Required for Developing PWAs?

Developing a PWA is not different from developing a standard web application, and it may be possible to upgrade your existing codebase. Note that for deployment, HTTPS is a requirement, although you can do testing on the localhost. The requirements for an app to become a PWA are discussed below.

1. Create a Manifest File

PWAs must be available to install directly via a website which has been visited, meaning there's no need for a third-party app store to get the app installed.

To enable this, the app needs to provide a manifest.json file — a JSON file that allows the developer to control how the PWA appears, what needs to be launched and other parameters.

A typical manifest file appears below. As we can see, the properties are setting a number of look-and-feel settings that will be used on the home screen of the device where the app will be installed.

The styling of the PWA starts from the manifest file, but there's no real CSS involved in this part. It's just straight up properties, which define the

application’s name, icons, primary colors, etc.

2. Using a Service Worker

A service worker is essentially a specific type of web worker, implemented as a JavaScript file that runs independently of the browser — such that it’s able to intercept network requests, caching or retrieving resources from the cache, and delivering push messages as necessary.

The service worker is what makes the PWA truly offline capable.

3. Install the Site Assets, Including CSS

The first time the Service worker is registered, an install event is triggered. This is where all of the site assets are installed, including any CSS, JS and other media and resource files required by the application:

```
self.addEventListener('install', function(e) { e.waitUntil(
caches.open('airhorner').then(function(cache) { return
cache.addAll([ '/', '/index.html', '/index.html?homescreen=1', '/?
homescreen=1', '/styles/main.css', '/scripts/main.min.js',
'/sounds/airhorn.mp3' ]); }) ); });
```

Developing PWAs is not very different from developing web apps, as long as the fundamental requirements have been met.

This is where the CSS starts to get involved, with the files defined that will be used to style the progressive web app.

CSS and PWAs: Some Considerations

When considering CSS and PWAs, there are things we need to keep in mind. All of these are decisions that need to be taken before the development of a progressive web app starts.

Should the App Follow Platform-specific UIs?

If we opt for one platform in favor of another (let’s say Android in favor of iOS) we risk alienating or putting at a disadvantage that part of the audience we didn’t consider.

We're also tying our fortunes to that platform — whether good fortunes or bad ones. It's also quite likely that platform designs change as they evolve between different versions.

My opinion is that vendor tie-in should be avoided as much as possible.

Platform-agnostic Design

Based on our previous consideration, the ideal is to opt for a mostly platform-neutral design.

If this path is chosen, we should ensure that the result doesn't stray too much in form and function from the UI that the native platform exposes.

One needs to use standard behaviors and perform-extensive user testing to ensure no UX problems have been introduced on specific platforms. As an example, it's highly recommended to avoid custom-written components and opt for standard HTML5 controls, which the browser can optimize for the UI and best experience.

Device Capabilities

The way forward for PWAs — even if at this point they're mostly focused on devices — is to become a holistic solution for all platforms, including desktops. As of May 2018, Chrome supports PWAs on desktops, and other vendors will soon be supporting this too.

Your CSS and styling considerations need to factor all of this and design for this from the get-go.

The beauty of working with a PWA, though, is that you can use a combination of CSS and the Service Worker implementation to enhance or limit the functionality based on the resources available.

Graceful Degradation and Progressive Enhancement

CSS in general is able to fall back gracefully; any unsupported properties are simply ignored.

Having said that, one also needs to make sure that critical elements have the right fallbacks, or are not missing any essential styling.

An alternative approach to graceful degradation is progressive enhancement. This is a concept that we should always keep in mind when working on our PWA. For example, we should test first for the support of a Service Worker API before we attempt to use it, using code similar to this: `if (!('serviceWorker' in navigator)) { console.log('Service Worker not supported'); return; }`

Variations of this logic can be used to handle different use cases, such as the styling for specific platforms, and others that we'll mention later on.

General Suggestions

Although PWAs have a number of advantages when it comes to the user experience, they shift a lot of responsibility back to the developer when it comes to dealing with the nuances of different technology.

Speaking as a developer/Product Manager, who understands the delicate balance between user needs and the limited availability of development resources, I would always recommend finding a middle ground that covers as many use cases as possible, without putting too much overhead on the development and testing teams.

The emergence of design standards such as Material Design, and frameworks such as Bootstrap, helps to establish platform-agnostic designs.

The actual framework used is typically able to address devices of different capabilities, while the design school provides a homogeneous look and feel across platforms, allowing the developer to focus on the App's features and functions.

If, on the other hand, you'd rather go down the whole separate look and feel, you'll be able to use your service worker to be able to do this.

JavaScript provides a number of functions that can help to take decisions based on the platform and capabilities available. You can, therefore, use your code to test for the platform and then apply a stylesheet accordingly.

For example, the navigator.platform method returns the platform on which the app is running, while the navigator.userAgent returns the browser being used.

The browser agent is unreliable when it comes to detecting the browser, so the code below is more of a demonstration of a concept rather than code that should be used in a live environment.

The navigator.platform is a more reliable method, but the sheer number of platforms available makes it cumbersome to use in our example.

```
/**
* Determine the mobile operating system.
* This function returns one of 'iOS', 'Android', 'Windows Phone', or
*
* @returns {String}
*/

function getMobileOperatingSystem()
{
  var userAgent = navigator.userAgent || navigator.vendor || window.
  // Windows Phone must come first because its UA also contains "And
  if (/windows phone/i.test(userAgent))
  {
    return  "Windows Phone";
  }

  if (/android/i.test(userAgent))
  {
    return  "Android";
  }

  if (/iPad|iPhone|iPod/.test(userAgent) && !window.MSStream)
  {
    return  "iOS";
  }

  return  "unknown";
  // return “Android” - one should either handle the unknown or fall
}
```

Using the return value of `getMobileOperatingSystem()` above, you can then register a specific CSS file accordingly.

From the code above, we can see that such logic can get very convoluted and difficult to maintain, so I would only recommend using it in situations where a platform-agnostic design is not suitable.

Another option is to use a simpler color scheme, only CSS applied to the primary styles that matche the underlying OS, though this could still “break” in the case where users have skinned their device.

PWA Frameworks

When learning how to develop a PWA, it’s great to create everything manually: it’s an excellent way of learning all the fundamentals concepts of building progressive web apps.

Once you’ve become familiar with all the important aspects, you might then start using a few tools to help you out, increasing your development efficiency.

As with most development scenarios, frameworks are available to make development of PWAs faster and more efficient.

Each of these frameworks uses specific CSS techniques to ensure that the development process is maintainable, scalable and achieves the needs of both the developer and the end user.

By using such frameworks, you can ensure that your PWA works nicely on most devices and platforms, because the frameworks usually have cross-platform capabilities, although they may offer limited backward compatibility. This is another of those decisions you’ll need to take when deciding what you’ll be using to develop your progressive web app. By using frameworks, you cede some of the control you’d have if writing everything from scratch.

Below we’ll suggest a number of frameworks/tools that can be used to aid development of PWAs.

A word of advice, though: frameworks add a lot of overhead when it comes to performance.

We recommend that you only use these resources when starting out, eventually opting out of using them and going for minimalistic, lean stylesheets, using a platform-agnostic design.

1. Create React App

React has all of the components in place to allow the development of a PWA, by using such libraries as the Create React App.

This is a great example of creating a React PWA with this library.

2. Angular

Given that Angular is a product of Google and how we've seen the company pushing for PWAs, it's no surprise that Angular has full support for PWAs.

If you're used to working with Angular, you could consider using this as your framework of choice.

Angular 2+ supports the implementation of PWA features (such as service workers and manifest files) natively through the framework using the following commands:

```
ng add @angular/pwa --project project_name
```

This is a great article which guides you through creating a PWA with Angular.

Ionic

Ionic is another framework for PWAs. The framework

- leverages Angular to enable the creation of native apps using web technologies
- uses Cordova to run the app on devices such as phones
- has a built-in service worker and manifest support.

This is a premium framework that includes a number of developer-oriented and team-oriented features such as rapid prototyping, to make development faster.

PWAs and Performance

One of the fundamentals of progressive web apps remains that of a fast and engaging user experience.

For this reason, when considering the CSS, one needs to ensure to keep things as

lean and minimalistic as possible.

This is one of the aspects where frameworks start to suffer. They add extra CSS resources that you're not using, which can reduce performance in PWAs.

A number of considerations you might want to keep in mind:

- use HTTP/2 on your server
- use such hints as `rel=preload` to allow early fetching of critical CSS and other resources
- use the NetworkInformationAPI and a caching implementation to access cached resources rather than downloading them
- inline critical CSS directly into the HTML document to optimize performance — which typically should be done for anything above the fold
- keep resources as lean and as small as possible
- minify all of your CSS resources and implement other optimizations such as compressing resources, optimizing images and use optimized image and video formats.

The Google guidelines on performance have other details you should keep in mind.

Google Lighthouse

Speaking of performance, the Google Lighthouse is a performance monitoring tool centered specifically around increasing performance, both of websites and progressive web apps.

Lighthouse, which used to be a plugin for Chrome, is today built-in with the Chrome Developer tools. It can be used to run tests against the progressive web app. The test generates a report which has plenty of detail to help you keep your development within the performance parameters of a PWA.

Wrapping Up

Using CSS and PWAs together has a few differences from using CSS to develop your web application or website (particularly in terms of performance and responsiveness). However, most techniques that can be used with web development can be suitably adopted for development of progressive web apps.

Whether you use frameworks or build everything from scratch, weigh the benefits against the disadvantages, take an informed decision and then stick with it.

Chapter 6: 20 Tips for Optimizing CSS Performance

by Craig Buckler

In this article, we look at 20 ways to optimize your CSS so that it's faster-loading, easier to work with and more efficient.

According to the latest HTTP Archive reports, the web remains a bloated mess with the mythical median website requiring 1,700Kb of data split over 80 HTTP requests and taking 17 seconds to fully load on a mobile device.

The Complete Guide to Reducing Page Weight provides a range of suggestions. In this article, we'll concentrate on CSS. Admittedly, CSS is rarely the worst culprit and a typical site uses 40KB spread over five stylesheets. That said, there are still optimizations you can make, and ways to change how we use CSS that will boost site performance.

1. Learn to Use Analysis Tools

You can't address performance problems unless you know where the faults lie. Browser DevTools are the best place to start: launch from the menu or hit `F12`, `Ctrl` + `Shift` + `I` or `Cmd` + `Alt` + `I` for Safari on macOS.

All browsers offer similar facilities, and the tools will open slowly on badly-performing pages! However, the most useful tabs include the following …

The **Network** tab displays a waterfall graph of assets as they download. For best results, disable the cache and consider throttling to a lower network speed. Look for files that are slow to download or block others. The browser normally blocks browser rendering while CSS and JavaScript files download and parse.

The **Performance** tab analyses browser processes. Start recording, run an activity such as a page reload, then stop recording to view the results. Look for:

1. Excessive *layout/reflow* actions where the browser has been forced to recalculate the position and size of page elements.
2. Expensive *paint* actions where pixels are changed.
3. *Compositing* actions where the painted parts of the page are put together for displaying on-screen. This is normally the least processor-intensive action.

Chrome-based browsers provide an **Audits** tab which runs Google's Lighthouse tool. It's often used by Progressive Web App developers, but also makes CSS performance suggestions.

Online Options

Alternatively, use online analysis tools that are not influenced by the speed and capabilities of your device and network. Most can test from alternative locations around the world:

- Pingdom Website Speed Test
- GTmetrix
- Google PageSpeed Insights
- WebPageTest

2. Make Big Wins First

CSS is unlikely to be the direct cause of performance issues. However, it may load heavy-hitting assets which can be optimized within minutes. Examples:

- Activate HTTP/2 and GZIP compression on your server
- Use a content delivery network (CDN) to increase the number of simultaneous HTTP connections and replicate files to other locations around the world
- Remove unused files.

Images are normally the biggest cause of page bulk, yet many sites fail to optimize effectively:

1. Resize bitmap images. An entry-level smartphone will take multi-megapixel images that can't be displayed in full on the largest HD screen. Few sites will require images of more than 1,600 pixels in width.
2. Ensure you use an appropriate file format. Typically, JPG is best for

photographs, SVG for vector images, and PNG for everything else. You can experiment to find the optimum type.
3. Use image tools to reduce file sizes by striping metadata and increasing compression factors.

That said, be aware that *x*Kb of image data is **not** equivalent to *x*Kb of CSS code. Binary images download in parallel and require little processing to place on a page. CSS blocks rendering and must be parsed into an object model before the browser can continue.

3. Replace Images with CSS Effects

It's rarely necessary to use background images for borders, shadows, rounded edges, gradients and some geometric shapes. Defining an "image" using CSS code uses considerably less bandwidth and is easier to modify or animate later.

4. Remove Unnecessary Fonts

Services such as Google Fonts make it easy to add custom fonts to any page. Unfortunately, a line or two of code can retrieve hundreds of kilobytes of font data. Recommendations:

1. Only use the fonts you need.
2. Only load the weights and styles you require — for example, roman, 400 weight, no italics.
3. Where possible, limit the character sets. Google fonts allows you to pick certain characters by adding a `&text=` value to the font URL — such as `fonts.googleapis.com/css?family=Open+Sans&text=SitePon` for displaying "SitePoint" in Open Sans.
4. Consider variable fonts, which define multiple weights and styles by interpolation so files are smaller. Support is currently limited to Chrome, Edge, and some editions of Safari but should grow rapidly. See How to Use Variable Fonts.
5. Consider OS fonts. Your 500Kb web font may be on-brand, but would anyone notice if you switched to the commonly available Helvetica or Arial? Many sites use custom web fonts, so standard OS fonts are considerably less common than they were!

5. Avoid `@import`

The `@import` at-rule allows any CSS file to be included within another. For example:

```
/* main.css */
@import url("base.css");
@import url("layout.css");
@import url("carousel.css");
```

This appears a reasonable way to load smaller components and fonts. *It's not.* `@import` rules can be nested so the browser must load and parse each file in series.

Multiple `<link>` tags within the HTML will load CSS files in parallel, which is considerably more efficient — especially when using HTTP/2:

```
<link rel="stylesheet" href="base.css">
<link rel="stylesheet" href="layout.css">
<link rel="stylesheet" href="carousel.css">
```

That said, there may be more preferable options …

6. Concatenate and Minify

Most build tools allow you to combine all partials into one large CSS file that has unnecessary whitespace, comments and characters removed.

Concatenation is less necessary with HTTP/2, which pipelines and multiplexes requests. In some cases, separate files may be beneficial if you have smaller, regularly-changing CSS assets. However, most sites are likely to benefit from sending a single file that is immediately cached by the browser.

Minification may not bring considerable benefits when you have GZIP enabled. That said, there's no real downside.

Finally, you could consider a build process that orders properties consistently within declarations. GZIP can maximize compression when commonly used strings are used throughout a file.

7. Use Modern Layout Techniques

For many years it was necessary to use CSS `float` to lay out pages. The technique is a hack. It requires lots of code and margin/padding tweaking to ensure layouts work. Even then, floats will break at smaller screen sizes unless media queries are added.

The modern alternatives:

- CSS Flexbox for one-dimensional layouts which (can) wrap to the next row according to the widths of each block. Flexbox is ideal for menus, image galleries, cards, etc.
- CSS Grid for two-dimensional layouts with explicit rows and columns. Grid is ideal for page layouts.

Both options are simpler to develop, use less code, can adapt to any screen size, and render faster than floats because the browser can natively determine the optimum layout.

8. Reduce CSS Code

The most reliable and fastest code is the code you need never write! The smaller your stylesheet, the quicker it will download and parse.

All developers start with good intentions, but CSS can bloat over time as the feature count increases. It's easier to retain old, unnecessary code rather than remove it and risk breaking something. A few recommendations to consider:

- Be wary of large CSS frameworks. You're unlikely to use a large percentage of the styles, so only add modules as you need them.
- Organize CSS into smaller files (partials) with clear responsibilities. It's easier to remove a carousel widget if the CSS is clearly defined in `widgets/_carousel.css`.
- Consider naming methodologies such as BEM to aid the development of discrete components.
- Avoid deeply nested Sass/preprocessor declarations. The expanded code can become unexpectedly large.
- Avoid using `!important` to override the cascade.

- Avoid inline styles in HTML.

Tools such as UnCSS can help remove redundant code by analyzing your HTML, but be wary about CSS states caused by JavaScript interaction.

9. Cling to the Cascade!

The rise of CSS-in-JS has allowed developers to avoid the CSS global namespace. Typically, randomly generated class names are created at build time so it becomes impossible for components to conflict.

If your life has been improved by CSS-in-JS, then carry on using it. However, it's worth understanding the benefits of the CSS cascade rather than working against it on every project. For example, you can set default fonts, colors, sizes, tables and form fields that are universally applied to every element in a single place. There is rarely a need to declare every style in every component.

10. Simplify Selectors

Even the most complex CSS selectors take milliseconds to parse, but reducing complexity will reduce file sizes and aid browser parsing. Do you really need this sort of selector?!

```
body > main.main > section.first h2:nth-of-type(odd) + p::first-line
➥".pdf"]
```

Again, be wary of deep nesting in preprocessors such as Sass, where complex selectors can be inadvertently created.

11. Be Wary of Expensive Properties

Some properties are slower to render than others. For added jankiness, try placing box shadows on all your elements!

```
*, ::before, ::after {
box-shadow: 5px 5px 5px rgba(0,0,0,0.5);
}
```

Browser performance will vary but, in general, anything which causes a

recalculation before painting will be more costly in terms of performance:

- `border-radius`
- `box-shadow`
- `opacity`
- `transform`
- `filter`
- `position: fixed`

12. Adopt CSS Animations

Native CSS transitions and animations will always be faster than JavaScript-powered effects that modify the same properties. CSS animations will not work in older browsers such as IE9 and below, but those users will never know what they're missing.

That said, avoid animation for the sake of it. Subtle effects can enhance the user experience without adversely affecting performance. Excessive animations could slow the browser and cause motion sickness for some users.

13. Avoid Animating Expensive Properties

Animating the dimensions or position of an element can cause the whole page to re-layout on every frame. Performance can be improved if the animation only affects the *compositing* stage. The most efficient animations use:

- `opacity` and/or
- `transform` to translate (move), scale or rotate an element (the original space the element used is not altered).

Browsers often use the hardware-accelerated GPU to render these effects. If neither are ideal, consider taking the element out of the page flow with `position: absolute` so it can be animated in its own layer.

14. Indicate Which Elements Will Animate

The will-change property allows CSS authors to indicate an element will be animated so the browser can make performance optimizations in advance. For

example, to declare that an element will have a `transform` applied:

```
.myelement {
will-change: transform;
}
```

Any number of comma-separated properties can be defined. However:

- use `will-change` as a last resort to fix performance issues
- don't apply it to too many elements
- give it sufficient time to work: that is, don't begin animations immediately.

15. Adopt SVG Images

Scalable vector graphics (SVGs) are typically used for logos, charts, icons, and simpler diagrams. Rather than define the color of each pixel like JPG and PNG bitmaps, an SVG defines shapes such as lines, rectangles and circles in XML. For example:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 800 600">
<circle cx="400" cy="300" r="50" stroke-width="5" stroke="#f00" fill
<svg>
```

Simpler SVGs are smaller than equivalent bitmaps and can infinitely scale without losing definition.

An SVG can be inlined directly in CSS code as a background image. This can be ideal for smaller, reusable icons and avoids additional HTTP requests. For example:

```
.mysvgbackground {
background: url('data:image/svg+xml;utf8,<svg xmlns="http://www.w3.o
➥viewBox="0 0 800 600"><circle cx="400" cy="300" r="50" stroke-width
➥stroke="#f00" fill="#ff0" /></svg>') center center no-repeat;
}
```

16. Style SVGs with CSS

More typically, SVGs are embedded directly within an HTML document:

```
<body>
<svg class="mysvg" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 8
```

```
    <circle cx="400" cy="300" r="50" />
<svg>
</body>
```

This adds the SVG nodes directly into the DOM. Therefore, all SVG styling attributes can be applied using CSS:

```
circle {
stroke-width: 1em;
}

.mysvg {
stroke-width: 5px;
stroke: #f00;
fill: #ff0;
}
```

The volume of embedded SVG code is reduced and the CSS styles can be reused or animated as necessary.

Note that using an SVG within an `<img>` tag or as a CSS background image means they're separated from the DOM, and CSS styling will have no effect.

17. Avoid Base64 Bitmap Images

Standard bitmap JPGs, PNGs and GIFs can be encoded to a base64 string within a data URI. For example:

```
.myimg {
background-image: url('');
}
```

Unfortunately:

- base64 encoding is typically 30% larger than its binary equivalent
- the browser must parse the string before it can be used
- altering an image invalidates the whole (cached) CSS file.

While fewer HTTP requests are made, it rarely provides a noticeable benefit — especially over HTTP/2 connections. In general, avoid inlining bitmaps unless the image is unlikely to change often and the resulting base64 string is unlikely to exceed a few hundred characters.

18. Consider Critical CSS

Those using Google page analysis tools will often see suggestions to *"inline critical CSS"* or *"reduce render-blocking stylesheets"*. Loading a CSS file blocks rendering, so performance can be improved with the following steps:

1. Extract the styles used to render elements above the fold. Tools such as criticalCSS can help.
2. Add those to a `<style>` element in the HTML `<head>`.
3. Load the main CSS file asynchronously using JavaScript (perhaps after the page has loaded).

The technique undoubtedly improves performance and could benefit Progressive Web or single-page apps that have consistent interfaces. Gains may be less clear for other sites/apps:

- It's impossible to identify the "fold", and it changes on every device.
- Most sites have differing page layouts. Each one could require different critical CSS, so a build tool becomes essential.
- Dynamic, JavaScript-driven events could make above-the-fold changes that are not identified by critical CSS tools.
- The technique mostly benefits the user's first page load. CSS is cached for subsequent pages so additional inlined styles will *increase* page weight.

That said, Google will love your site and push it to #1 for every search term. *(SEO "experts" can quote me on that. Everyone else will know it's nonsense.)*

19. Consider Progressive Rendering

Rather than using a single site-wide CSS file, **progressive rendering** is a technique that defines individual stylesheets for separate components. Each is loaded immediately before a component is referenced in the HTML:

```
<head>

<!-- core styles used across components -->
<link rel='stylesheet' href='base.css' />

</head>
<body>
```

```
<!-- header component -->
<link rel='stylesheet' href='header.css' />
<header>...</header>

<!-- primary content -->
<link rel='stylesheet' href='content.css' />
<main>

    <!-- form styling -->
    <link rel='stylesheet' href='form.css' />
    <form>...</form>

</main>

<!-- header component -->
<link rel='stylesheet' href='footer.css' />
<footer>...</footer>

</body>
```

Each `<link>` still blocks rendering, but for a shorter time, because the file is smaller. The page is usable sooner, because each component renders in sequence; the top of the page can be viewed while remaining content loads.

The technique works in Firefox, Edge and IE. Chrome and Safari *"optimize"* the experience by loading all CSS files and showing a white screen while that occurs — but that's no worse than loading each in the `<head>`.

Progressive rendering could benefit large sites where individual pages are constructed from a selection of different components.

20. Learn to Love CSS

The most important tip: *understand your stylesheets!*

Adding vast quantities of CSS from StackOverflow or BootStrap may produce quick results, but it will also bloat your codebase with unused junk. Further customization becomes frustratingly difficult, and the application will never be efficient.

CSS is easy to learn but difficult to master. You can't avoid the technology if you want to create effective client-side code. A little knowledge of CSS basics can

revolutionize your workflow, enhance your apps, and noticeably improve performance.

Chapter 7: Advanced CSS Theming with Custom Properties and JavaScript

by Ahmed Bouchefra

Throughout this tutorial on CSS theming, we'll be using CSS custom properties (also known as CSS variables) to implement dynamic themes for a simple HTML page. We'll create dark and light example themes, then write JavaScript to switch between the two when the user clicks a button.

Just like in typical programming languages, variables are used to hold or store values. In CSS, they're typically used to store colors, font names, font sizes, length units, etc. They can then be referenced and reused in multiple places in the stylesheet. Most developers refer to "CSS variables", but the official name is **custom properties**.

CSS custom properties make it possible to modify variables that can be referenced throughout the stylesheet. Previously, this was only possible with CSS preprocessors such as Sass.

Understanding :root and `var()`

Before creating our dynamic theming example, let's understand the essential basics of custom properties.

A custom property is a property whose name starts with two hyphens (**--**) like `--foo`. They define variables that can be referenced using `var()`. Let's consider this example:

```
:root {
  --bg-color: #000;
  --text-color: #fff;
}
```

Defining custom properties within the `:root` selector means they are available in the global document space to all elements. **`:root`** is a CSS pseudo class which matches the root element of the document — the `<html>` element. It's similar to the `html` selector, but with higher specificity.

You can access the value of a `:root` custom property anywhere in the document:

```
div {
  color: var(--text-color);
  background-color: var(--bg-color);
}
```

You can also include a fallback value with your CSS variable. For example:

```
div {
  color: var(--text-color, #000);
  background-color: var(--bg-color, #fff);
}
```

If a custom property isn't defined, their fallback value is used instead.

Defining custom properties inside a CSS selector other than the `:root` or `html` selector makes the variable available to matching elements and their children.

CSS Custom Properties vs Preprocessor Variables

CSS pre-processors such as Sass are often used to aid front-end web development. Among the other useful features of preprocessors are variables. But what's the difference between Sass variables and CSS custom properties?

- CSS custom properties are natively parsed in modern browsers. Preprocessor variables require compilation into a standard CSS file and all variables are converted to values.
- Custom properties can be accessed and modified by JavaScript. Preprocessor variables are compiled once and only their final value is available on the client.

Writing a Simple HTML Page

Let’s start by creating a folder for our project:

```
$ mkdir css-variables-theming
```

Next, add an `index.html` inside the project’s folder:

```
$ cd css-variables-theming
$ touch index.html
```

And add the following content:

```
<nav class="navbar">Title</nav>
<div class="container">
  <div>
    <input type="button" value="Light/Dark" id="toggle-theme" />
  </div>
    <h2 class="title">What is Lorem Ipsum?</h2>
    <p class="content">Lorem Ipsum is simply dummy text of the print
    ➥typesetting industry...</p>
</div>
<footer>
  Copyright 2018
</footer>
```

We are adding a navigation bar using a `<nav>` tag, a footer, and a container `<div>` that contains a button (that will be used to switch between light and dark themes) and some dummy *Lorem Ipsum* text.

Writing Basic CSS for Our HTML Page

Now let’s style our page. In the same file using an inline `<style>` tag in the `<head>` add the following CSS styles:

```
<style>
* {
  margin: 0;
}
html{
  height: 100%;
}
body{
  height: 100%;
  font-family: -apple-system, BlinkMacSystemFont“Segoe UI”, “Roboto”
  ➥“Ubuntu”, “Cantarell”,“Fira Sans”, “Droid Sans”, “Helvetica Neue”
  display: flex;
```

```
  flex-direction: column;
}
nav{
  background: hsl(350, 50%, 50%);
  padding: 1.3rem;
  color: hsl(350, 50%, 10%);
}
.container{
  flex: 1;
  background:hsl(350, 50%, 95%);
  padding: 1rem;
}
p.content{
  padding: 0.7rem;
  font-size: 0.7rem;
  color: hsl(350, 50%, 50%);
}
.container h2.title{
  padding: 1rem;
  color: hsl(350, 50%, 20%);
}
footer{
  background: hsl(350, 93%, 88%);
  padding: 1rem;
}
input[type=button] {
  color:hsl(350, 50%, 20%);
  padding: 0.3rem;
  font-size: 1rem;
}
</style>
```

CSS3 HSL (Hue, Saturation, Lightness) notation is used to define colors. The hue is the angle on a color circle and the example uses 350 for red. All page colors use differing variations by changing the saturation (percentage of color) and lightness (percentage).

Using HSL allows us to easily try different main colors for the theme by only changing the hue value. We could also use a CSS variable for the hue value and switch the color theme by changing a single value in the stylesheet or dynamically altering it with JavaScript.

This is a screen shot of the page:

Live Code

Check out the related pen: See the Pen CSS Theming 1.

Let's use a CSS variable for holding the value of the hue of all colors in the page. Add a global CSS variable in the `:root` selector at the top of the `<style>` tag:

```
:root{
  --main-hue : 350;
}
```

Next, we replace all hard-coded *350* values in *hsl()* colors with the `--main-hue` variable. For example, this is the *nav* selector:

```
nav{
  background: hsl(var(--main-hue) , 50%, 50%);
```

```
  padding: 1.3rem;
  color: hsl(var(--main-hue), 50%, 10%);
}
```

Now if you want to specify any color other than red, you can just assign the corresponding value to `--main-hue`. These are some examples:

```
:root{
  --red-hue: 360;
  --blue-hue: 240;
  --green-hue: 120;
  --main-hue : var(--red-hue);
}
```

We are defining three custom properties for red, blue and green, then assigning the `--red-hue` variable to `--main-hue`.

This a screen shot of pages with different values for `--main-hue`:

CSS custom properties offer a couple of benefits:

- A value can be defined in a single place.
- That value can be named appropriately to aid maintenance.
- The value can be dynamically altered using JavaScript. For example, the `--main-hue` can be set to any value between 0 and 360.

Using JavaScript to dynamically set the value of `--main-hue` from a set of predefined values or user submitted value for hue (it should be between *0* and *360*) we can provide the user with many possibilities for colored themes.

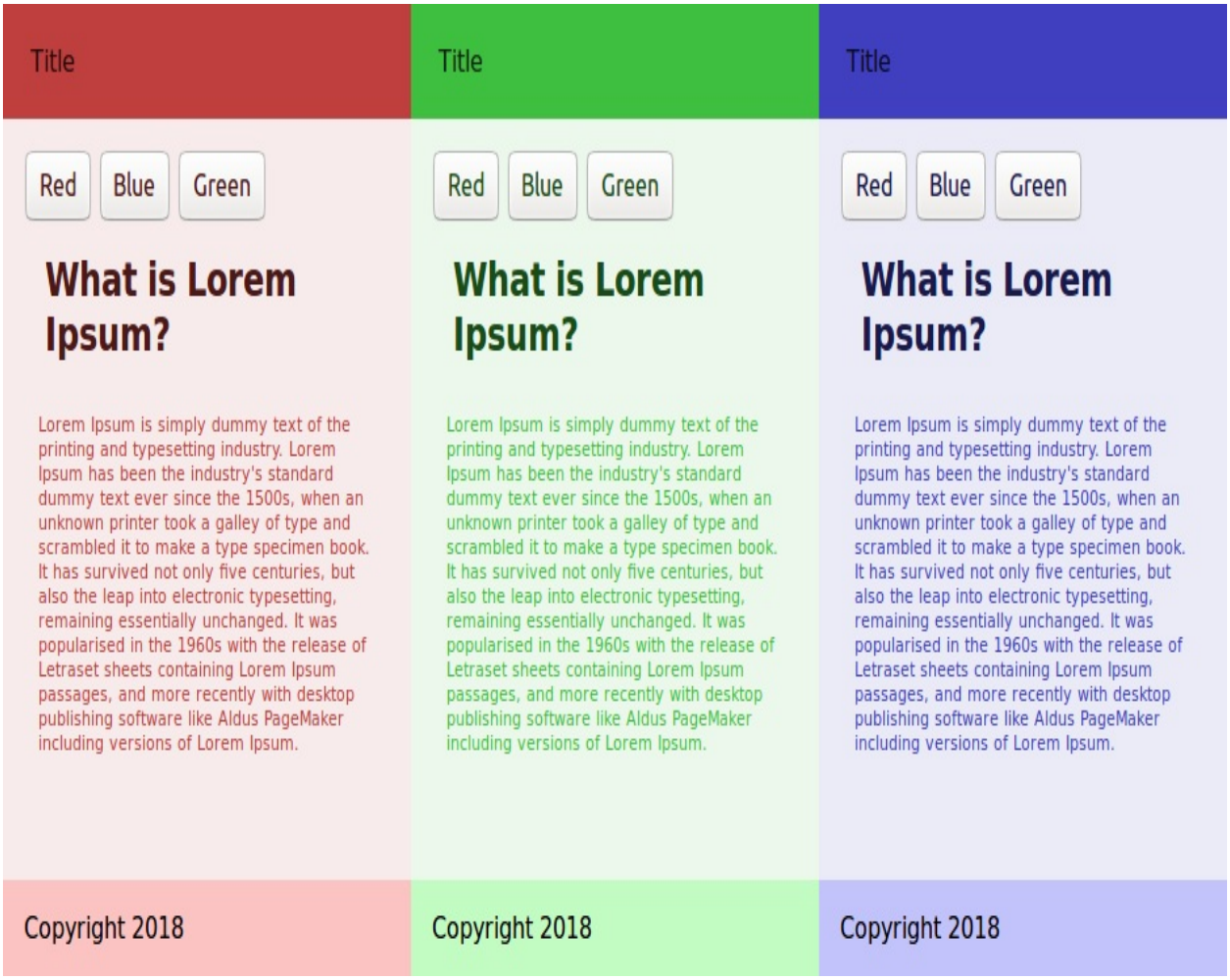
The following line of code will set the value of `--main-hue` to 240 (blue):

```
document.documentElement.style.setProperty('--main-hue', 240);
```

Live Code

Check out the following pen, which shows a full example that allows you to dynamically switch between red, blue and green colored themes: See the Pen CSS Theming 2.

This is a screen shot of the page from the pen:



Adding a CSS Dark Theme

Now let’s provide a dark theme for this page. For more control over the colors of different entities, we need to add more variables.

Going through the page’s styles, we can replace all HSL colors in different selectors with variables after defining custom properties for the corresponding colors in `:root`:

```
:root{
  /*...*/
  --nav-bg-color: hsl(var(--main-hue) , 50%, 50%);
  --nav-text-color: hsl(var(--main-hue), 50%, 10%);
  --container-bg-color: hsl(var(--main-hue) , 50%, 95%);
  --content-text-color: hsl(var(--main-hue) , 50%, 50%);
  --title-color: hsl(var(--main-hue) , 50%, 20%);
  --footer-bg-color: hsl(var(--main-hue) , 93%, 88%);
```

```
  --button-text-color: hsl(var(--main-hue), 50%, 20%);
}
```

Appropriate names for the custom properties have been used. For example, `--nav-bg-color` refers to *the color of the nav background*, while `--nav-text-color` refers to *the color of nav foreground/text*.

Now duplicate the `:root` selector with its content, but add a theme attribute with a *dark* value:

```
:root[theme='dark']{
  /*...*/
}
```

This theme will be activated if a *theme* attribute with a *dark* value is added to the `<html>` element.

We can now play with the values of these variables manually, by reducing the lightness value of the HSL colors to provide a dark theme, or we can use other techniques such as CSS filters like `invert()` and `brightness()`, which are commonly used to adjust the rendering of images but can also be used with any other element.

Add the following code to `:root[theme='dark']`:

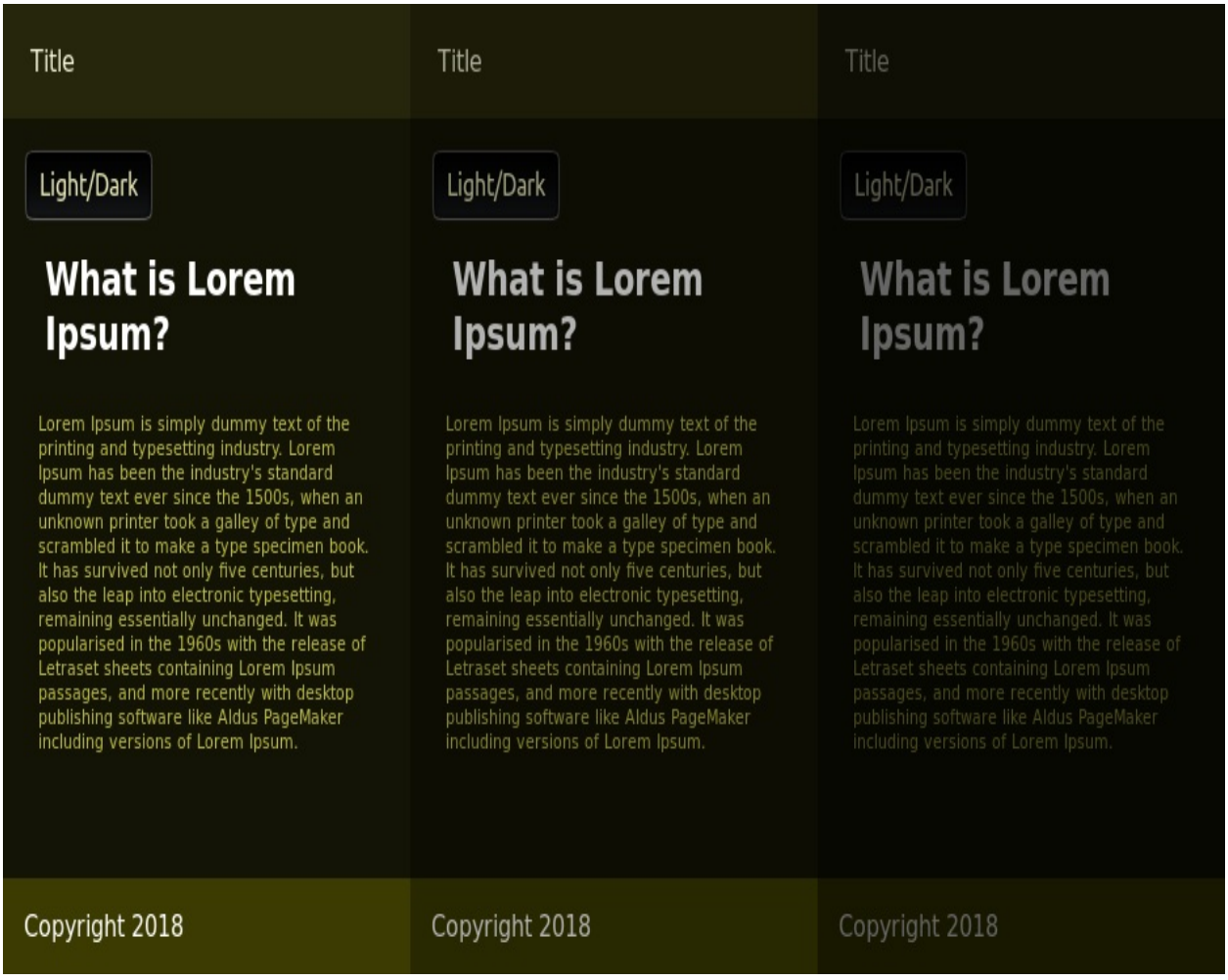
```
:root[theme='dark'] {
  --red-hue: 360;
  --blue-hue: 240;
  --green-hue: 120;
  --main-hue: var(--blue-hue);
  --nav-bg-color: hsl(var(--main-hue), 50%, 90%);
  --nav-text-color: hsl(var(--main-hue), 50%, 10%);
  --container-bg-color: hsl(var(--main-hue), 50%, 95%);
  --content-text-color: hsl(var(--main-hue), 50%, 50%);
  --title-color: hsl(--main-hue, 50%, 20%);
  --footer-bg-color: hsl(var(--main-hue), 93%, 88%);
  --button-text-color: hsl(var(--main-hue), 50%, 20%);
  filter: invert(1) brightness(0.6);
}
```

The `invert()` filter inverts all the colors in the selected elements (every element in this case). The value of inversion can be specified in percentage or number. A value of `100%` or `1` will completely invert the hue, saturation, and lightness values of the element.

The `brightness()` filter makes an element brighter or darker. A value of `0` results in a completely dark element.

The `invert()` filter makes some elements very bright. These are toned down by setting `brightness(0.6)`.

A dark theme with different degrees of darkness:



Switching Themes with JavaScript

Let’s now use JavaScript to switch between the dark and light themes when a user clicks the *Dark/Light* button. In `index.html` add an inline `<script>` before the closing `</body>` with the following code:

```
const toggleBtn = document.querySelector("#toggle-theme");
toggleBtn.addEventListener('click', e => {
```

```
  console.log("Switching theme");
  if(document.documentElement.hasAttribute('theme')){
    document.documentElement.removeAttribute('theme');
  }
  else{
    document.documentElement.setAttribute('theme', 'dark');
  }
});
```

Document.documentElement refers to the the root DOM Element of the document — that is, `<html>`. This code checks for the existence of a *theme* attribute using the `.hasAttribute()` method and adds the attribute with a *dark* value if it doesn't exist, causing the switch to the dark theme. Otherwise, it removes the attribute, which results in switching to the light theme.

Changing CSS Custom Properties with JavaScript

Using JavaScript, we can access custom properties and change their values dynamically. In our example, we hard-coded the brightness value, but it could be dynamically changed. First, add a slider input in the HTML page next to the *dark/light* button:

```
 <input type="range" id="darknessSlider" name="darkness" value="1" m
max="1" step="0.1" />
```

The slider starts at `1` and allows the user to reduce it to `0.3` in steps of `0.1`.

Next, add a custom property for the darkness amount with an initial value of `1` in `:root[theme='dark']`:

```
:root[theme='dark']{
  /*...*/
  --theme-darkness: 1;
}
```

Change the `brightness` filter to this custom property instead of the hard-coded value:

```
filter: invert(1) brightness(var(--theme-darkness));
```

Finally, add the following code to synchronize the value of `--theme-darkness`

with the slider value:

```
const darknessSlider = document.querySelector("#darknessSlider");
darknessSlider.addEventListener('change', (e)=>{
  const val = darknessSlider.value
  document.documentElement.style.setProperty('--theme-darkness', val
});
```

We're listening for the change event of the slider and setting the value of `--theme-darkness` accordingly using the `setProperty()` method.

We can also apply the `brightness` filter to the light theme. Add the `--theme-darkness` custom property in the top of the `:root` selector:

```
:root{
  /*...*/
  --theme-darkness: 1;
}
```

Then add a `brightness` filter in the bottom of the same selector:

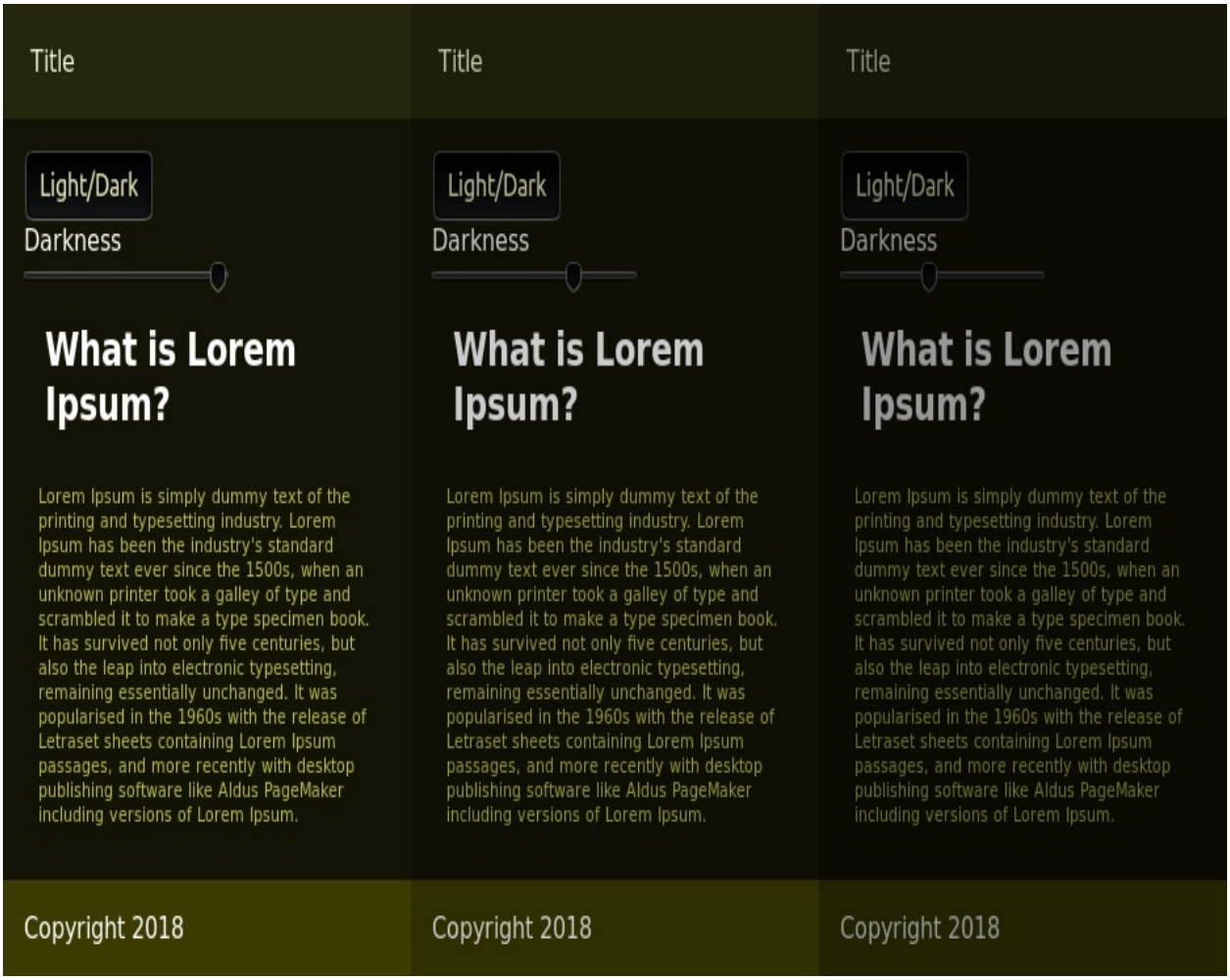
```
:root{
  /*...*/
  filter: brightness(var(--theme-darkness));
}
```

Live Code

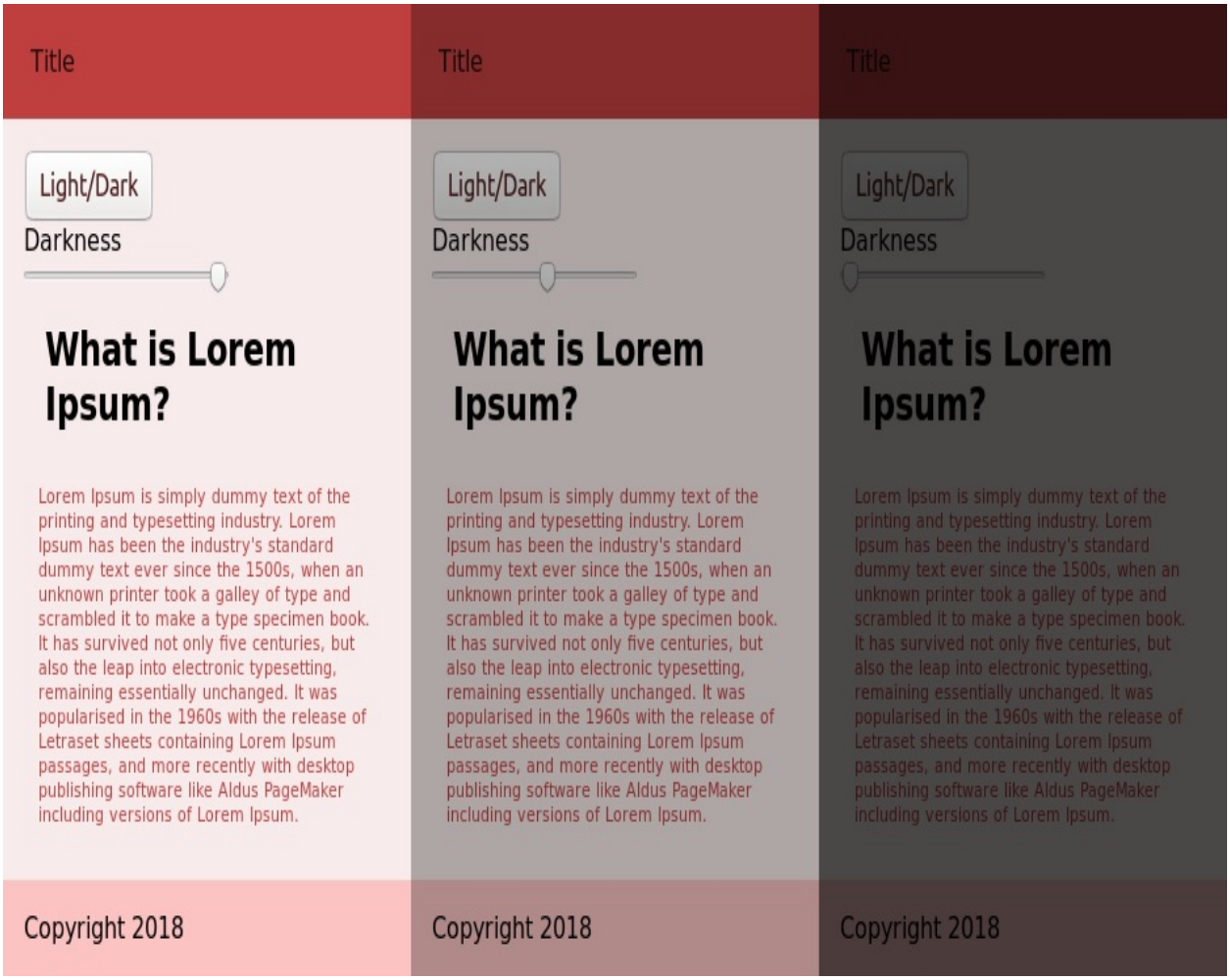
You can find the code of this example in the following pen: See the Pen CSS Theming.

You can find the code of this example in the following pen:

Here's a screenshot of the dark theme of the final example:



Here’s a screenshot of the light theme:



Conclusion

In this tutorial, we've seen how to use CSS custom properties to create themes and switch dynamically between them. We've used the HSL color scheme, which allows us to specify colors with hue, saturation and lightness values and CSS filters (`invert` and `brightness`) to create a dark version of a light theme.

Here are some links for further reading if you want to learn more about CSS theming:

- Using CSS custom properties (variables)
- HSL and HSV
- CSS Custom Properties and Theming
- Dark theme in a day

Book 2: CSS Grid Layout: 5 Practical Projects

sitepoint

CSS GRID LAYOUT:
5 PRACTICAL PROJECTS

MODERN LAYOUT

Chapter 1: Redesigning a Site to Use CSS Grid Layout

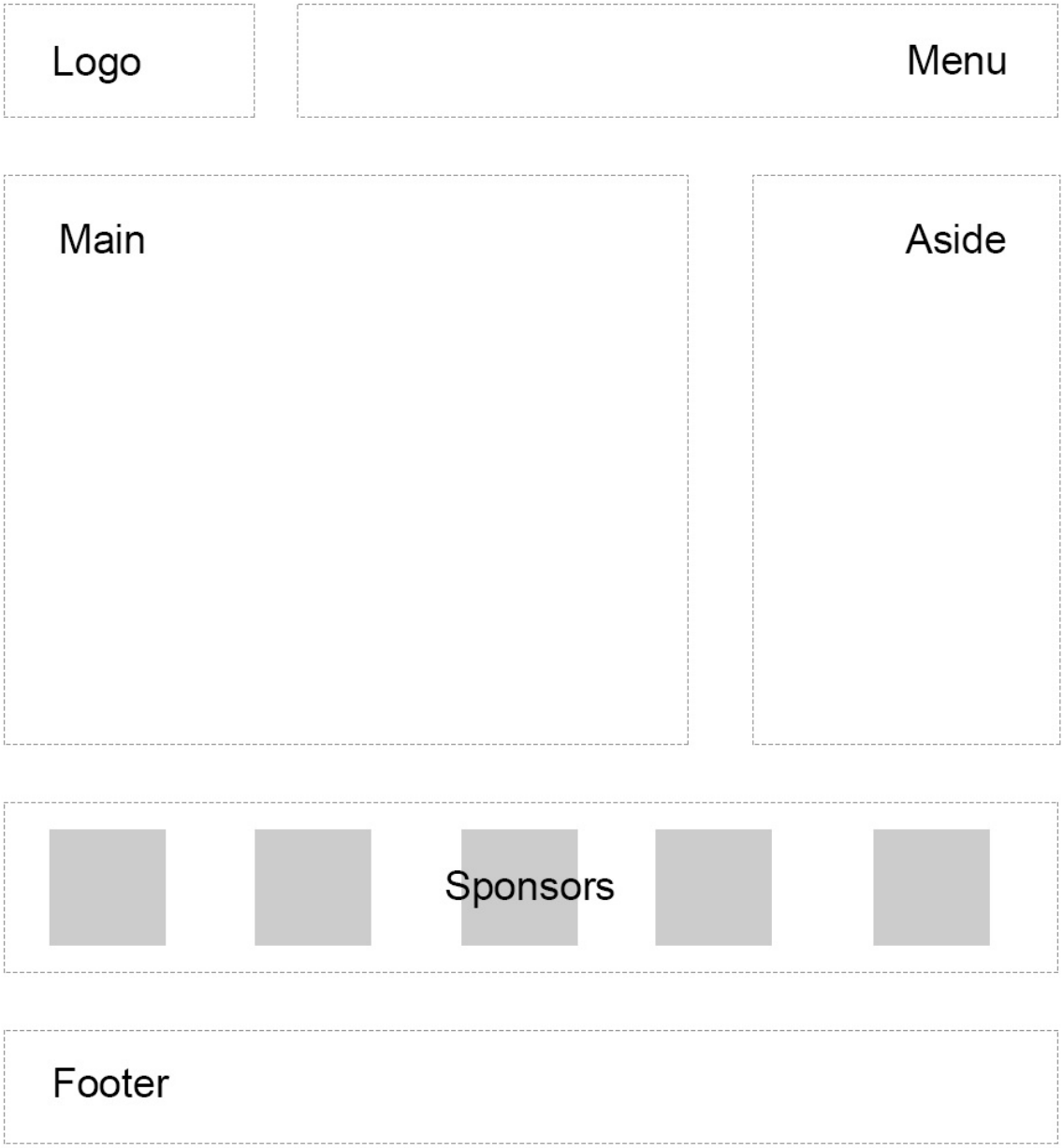
by Ilya Bodrov

CSS Grid is a new hot trend in web development these days. Forget about table layouts and floats: a new way to design websites is already here! This technology introduces two-dimensional grids which define multiple areas of layout with a handful of CSS rules. Grid can make third-party frameworks such as 960gs or Bootstrap grid redundant, as you may easily do everything yourself! This feature is supported by all major browsers, though Internet Explorer implements an older version of the specification.

In this article we are going to see CSS Grid in action by creating a responsive multi-column website layout.

What We Are Going to Build

So, we were asked to create a typical website layout with a header, main content area, sidebar to the right, a list of sponsors, and a footer:



Another developer has already tried to solve this task and came up with a solution that involves floats, `display: table`, and some clearfix hacks. We are going to refer to this existing layout as "initial".

Live Code

See the Pen Multi-Column Layout With Floats.

Until recently, floats were considered to be the best option to create such layouts. Prior to that, we had to utilize HTML tables but they had a number of downsides. Specifically, such table layout is very rigid, it requires lots of tags (`table`, `tr`, `td`, `th` etc), and semantically these tags are used to present table data, not to design layouts.

But CSS continues to evolve, and now we have CSS Grid. Conceptually it is similar to an old table layout but can use semantic HTML elements with a more flexible layout.

Planning The Grid

First things first: we need to define a basic HTML structure for our document. Before that, let's briefly talk about how the initial example works. It has the following main blocks:

- `.container` is the global wrapper that has small margins to the left and to the right.
- `.main-header` is the header that contains the `.logo` (occupies 20% of space, floats to the left) and the `.main-menu` (occupies 79% of space, floats to the right). The header is also assigned with a hacky fix to clear the floats.
- `.content-area-wrapper` wraps the main `.content-area` (occupies 66.6% of space minus `1rem` reserved for margin, floats to the left) and the `.sidebar` (occupies 33.3% of the space, floats to the right). The wrapper itself is also assigned with a clearfix.
- `.sponsors-wrapper` contains the logos of the sponsors. Inside, there is a `.sponsors` section with the `display` property set to `table`. Each sponsor, in turn, is displayed as a table cell.
- `.footer` is our footer and spans to 100% of space.

Our new layout will be very similar to the initial one, but with one exception: we won't add the `.main-header` and `.content-area-wrapper` wrappers because the clearfixes won't be required anymore. Here is the new version of the HTML:

```
<div class="container">
   <header class="logo">
     <h1><a href="#">DemoSite</a></h1>
   </header>

   <nav class="main-menu">
```

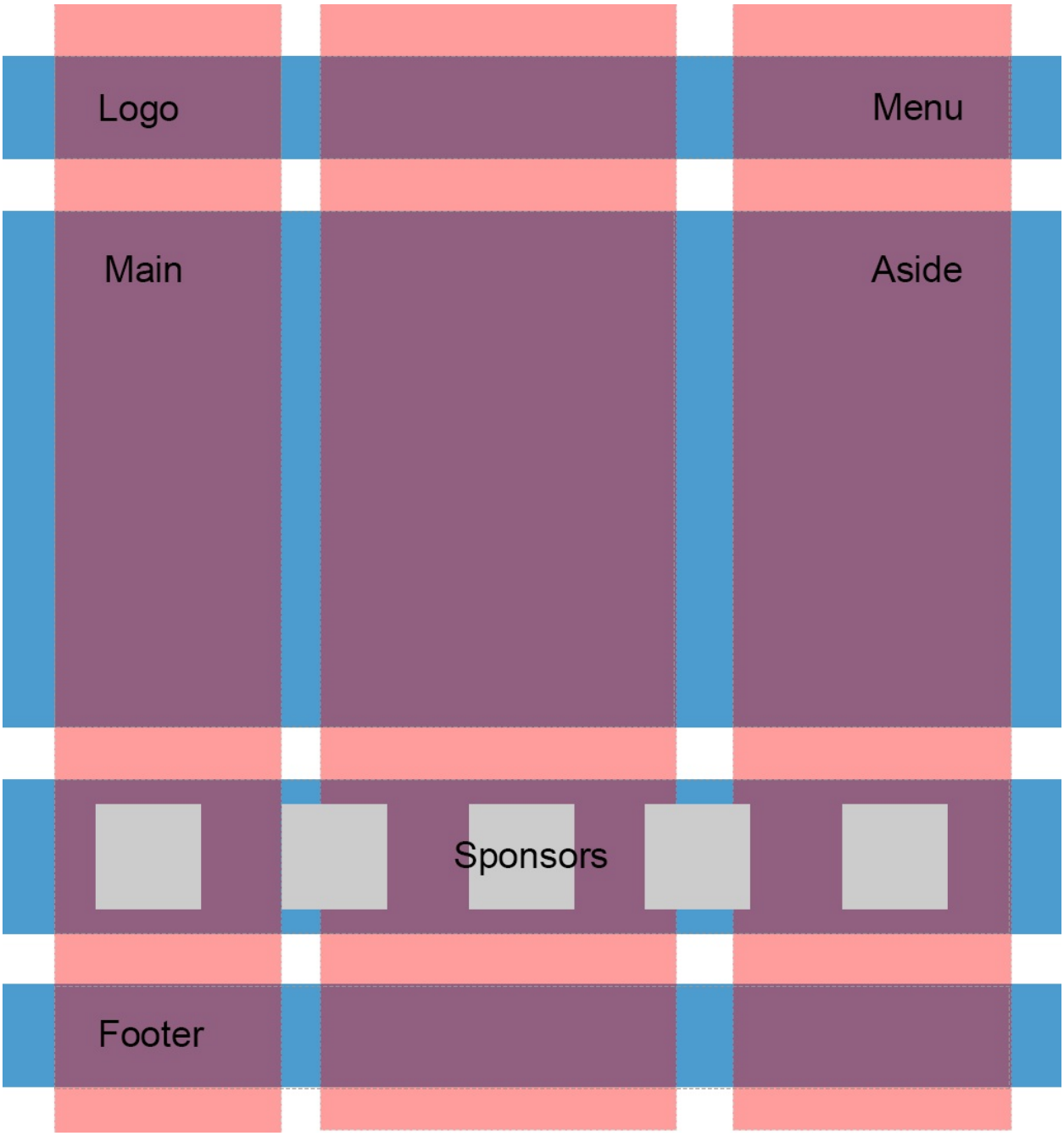
```
    <ul>
      <li class="main-menu__item"><a href="#">Our clients</a></li>
      <li class="main-menu__item"><a href="#">Products</a></li>
      <li class="main-menu__item"><a href="#">Contact</a></li>
    </ul>
  </nav>

  <main class="content-area">
    <h2>Welcome!</h2>

    <p>
      Content
    </p>
  </main>

  <aside class="sidebar">
    <h3>Additional stuff</h3>

    <ul>
      <li>Items</li>
      <li>Are</li>
      <li>Listed</li>
      <li>Here</li>
      <li>Wow!</li>
    </ul>
  </aside>

  <section class="sponsors-wrapper">
    <h2>Our sponsors</h2>

    <section class="sponsors">
      <figure class="sponsor">
        <img src="https://via.placeholder.com/150x150">
      </figure>

      <figure class="sponsor">
        <img src="https://via.placeholder.com/200x150">
      </figure>

      <figure class="sponsor">
        <img src="https://via.placeholder.com/100x200">
      </figure>

      <figure class="sponsor">
        <img src="https://via.placeholder.com/100x100">
      </figure>

      <figure class="sponsor">
        <img src="https://via.placeholder.com/200x200">
```

```
      </figure>
    </section>

  </section>

  <footer class="footer">
    <p>
      &copy; 2018 DemoSite. White&amp;amp;Sons LLC. All rights (perhaps
    </p>
  </footer>
</div>
```

Note that you may utilize the `body` as the global `.container` — that's just a matter of preference in this case. All in all, we have six main areas:

1. Logo
2. Menu
3. Main content
4. Sidebar
5. Sponsors
6. Footer

Usually it is recommended to to implement mobile-first approach: that is, start from the mobile layout and then designing for larger screens. This is not necessary in this case because we are adapting an initial layout which already falls back to a linearized view on small-screen devices. Therefore, let's start by focusing on the grid's implementation, and after that talk about responsiveness and fallback rules. So, return to our scheme and see how the grid columns can be arranged:



So, I propose having three columns (highlighted with red color) and four rows (highlighted with blue). Some areas, like logo, are going to occupy only one column, whereas others, like main content, are going to span multiple columns. Later we can easily modify the layout, move the areas around, or add new ones.

Following the scheme, give each area a unique name. These will be used in the layout defined below:

```
.logo {
   grid-area: logo;
 }

 .main-menu {
   grid-area: menu;
 }

 .content-area {
   grid-area: content;
 }

 .sidebar {
   grid-area: sidebar;
 }

 .sponsors-wrapper {
   grid-area: sponsors;
 }

 .footer {
   grid-area: footer;
 }
```

Now set the `display` property to `grid`, define three columns and add small margins to the left and right of the main container:

```
.container {
   display: grid;
   margin: 0 2rem;
   grid-template-columns: 2fr 6fr 4fr;
 }
```

`display: grid` defines a grid container and sets a special formatting context for its children. `fr` is a special unit that means "fraction of the free space of the grid container". `2 + 6 + 4` gives us `12`, and `6 / 12 = 0.5`. It means that the middle column is going to occupy 50% of the free space.

I would also like to add some spacing between the rows and columns:

```
.container {
   // ...
   grid-gap: 2rem 1rem;
 }
```

Having done this we can work with individual areas. But before wrapping up this section let's quickly add some common styles:

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

html {
  font-size: 16px;
  font-family: Georgia, serif;
}

body {
  background-color: #fbfbfb;
}

h1, h2, h3 {
  margin-top: 0;
}

header h1 {
  margin: 0;
}

main p {
  margin-bottom: 0;
}
```

Good! Now we can proceed to the first target which is going to be the header.

Designing the Header

Our header occupies the first row that should have a specific height set to `3rem`. In the initial layout this is solved by assigning the `height` property for the header wrapper:

```
.main-header {
  height: 3rem;
}
```

Also note that the logo and the menu are vertically aligned to the middle which is achieved using the `line-height` trick:

```
.logo {
   // ...
   height: 100%;
   line-height: 3rem;
 }
```

With CSS Grid, however, things are going to be simpler: we won't require any CSS hacks.

Start by defining the first row:

```
.container {
   // ...
   grid-template-rows: 3rem;
 }
```

Our logo should occupy only one column, whereas the menu should span two columns. We can express our intent with the help of `grid-template-areas` property which references the `grid-area` names assigned above:

```
.container {
   // ...
   grid-template-areas:
           "logo menu menu";
 }
```

What is going on here? Well, by saying `logo` only once we are making sure that it occupies only one, the left-most column. `menu menu` means that the menu occupies two columns: the middle and the right-most one. See how straightforward this rule is!

Now align the logo on the Y axis:

```
.logo {
   grid-area: logo;
   align-self: center;
 }
```

The menu should be centered vertically and pulled to the right:

```
.main-menu {
```

```
  grid-area: menu;
  align-self: center;
  justify-self: end;
}
```

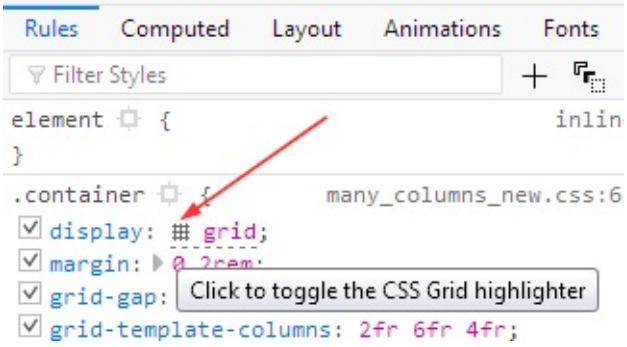
Our menu is built with the `ul` and `li` tags, so let's also style them a bit by removing markers, nullifying margins/paddings, and setting turning the menu to a flex container:

```
.main-menu ul {
  margin: 0;
  padding: 0;
  display: flex;
}

.main-menu__item {
  list-style-type: none;
  padding: 0;
  font-size: 1.1rem;
  margin-right: 0.5rem;
}

.main-menu .main-menu__item:last-of-type {
  margin-right: 0;
}
```

That's pretty much it. To observe the result, I am going to use Firefox browser with a handy CSS Grid highlighter tool enabled (there are similar tools for other browsers available: for instance, Gridman for Chrome). To gain access to this tool, press `F12` and select the `.container` element which should have a `grid` label:

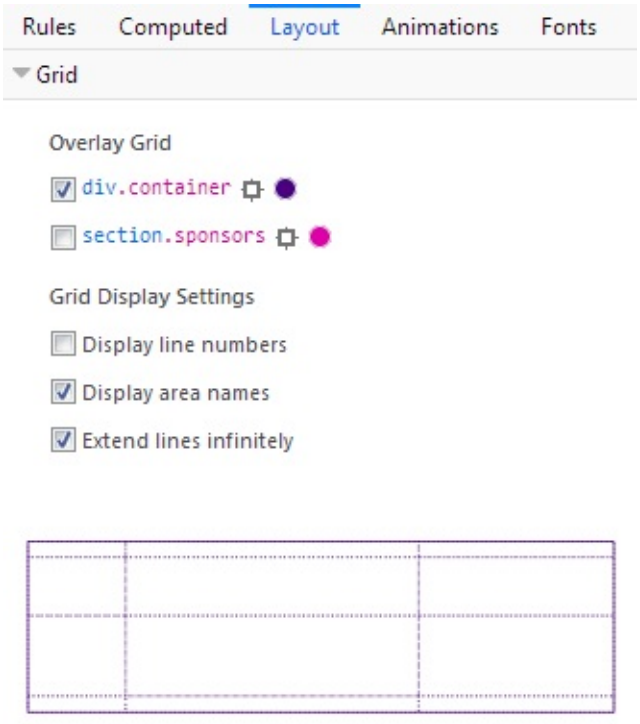
After that, proceed to the CSS rules tab, and find the `display: grid` property. By pressing on the small icon next to the `grid` value you may enable or disable

the highlighter:



Here is the result:

Highlighter displays all your rows and columns, as well as the margins between them and the areas' names. You may customize the output inside the Layout section which also lists all the grids on the page:



So, we've dealt with the header, therefore let's proceed to the main content area

and the sidebar.

Main Content and Sidebar

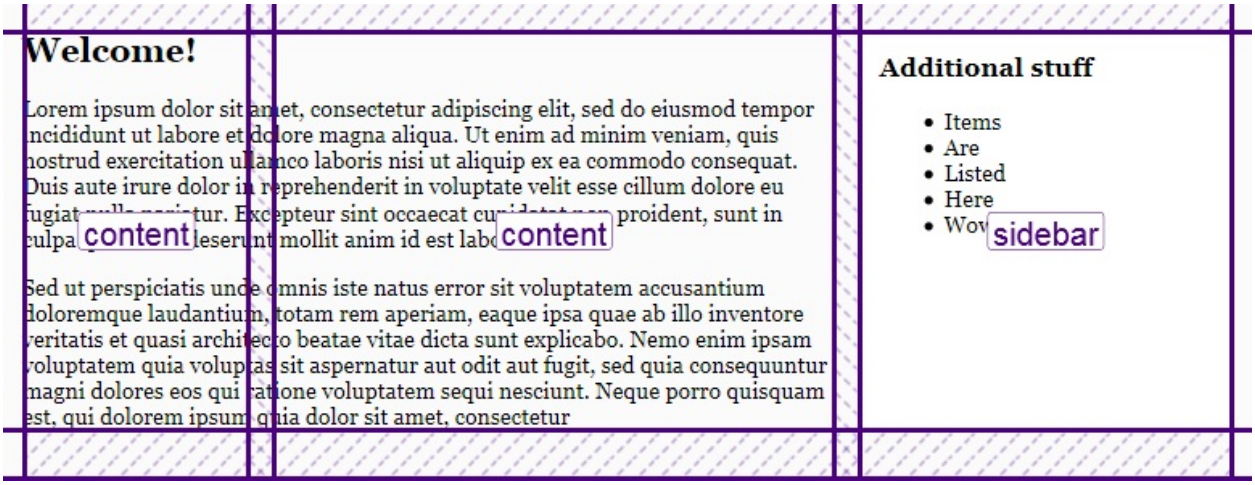
Our main content area should span two columns, whereas the sidebar should occupy only one. As for the row, I would like its height to be set automatically. We can update the `.container` grid accordingly:

```
.container {
   //  ...
   grid-template-rows: 3rem auto;
   grid-template-areas:
           "logo menu menu"
           "content content sidebar";
 }
```

I'd like to add some padding for the sidebar to give it some more visual space:

```
.sidebar {
   grid-area: sidebar;
   padding: 1rem;
 }
```

Here is the result as viewed in Firefox's Grid tool:



Sponsors

The sponsors section should contain five items with equal widths and heights. Each item, in turn, will have one image.

In the initial layout this block is styled with `display: table` property, but we won't rely on it. Actually, the sponsors section may be great candidate for applying CSS grid as well!

First of all, tweak the `grid-template-areas` to include the `sponsors` area:

```
.container {
   // ...
   grid-template-areas:
           "logo menu menu"
           "content content sidebar"
           "sponsors sponsors sponsors"
 }
```

Now turn the `.sponsors` section to a grid as well:

```
.sponsors {
   display: grid;
 }
```

As long as we need five items with equal widths, a `repeat` function can be utilized to define the columns:

```
.sponsors {
   display: grid;
   grid-template-columns: repeat(5, 1fr);
 }
```

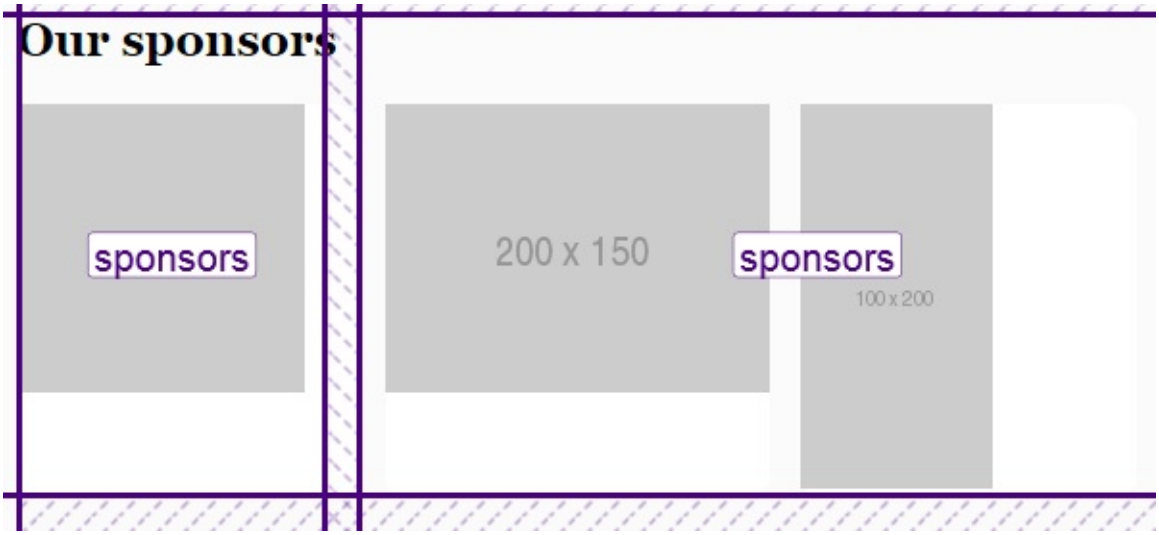
As for the row, its height should be set automatically. The gap between the columns should be equal to `1rem`:

```
.sponsors {
   display: grid;
   grid-template-columns: repeat(5, 1fr);
   grid-template-rows: auto;
   grid-column-gap: 1rem;
 }
```

Style each item:

```
.sponsor {
   margin-left: 0;
   margin-right: 0;
   background-color: #fff;
   border-radius: 0.625rem;
 }
```

Here is the intermediate result:



This example illustrates that you may safely nested grids, and this will work without any problems. Another solution might be using Flexbox, but in this case the sponsors may wrap if there is not enough width for them.

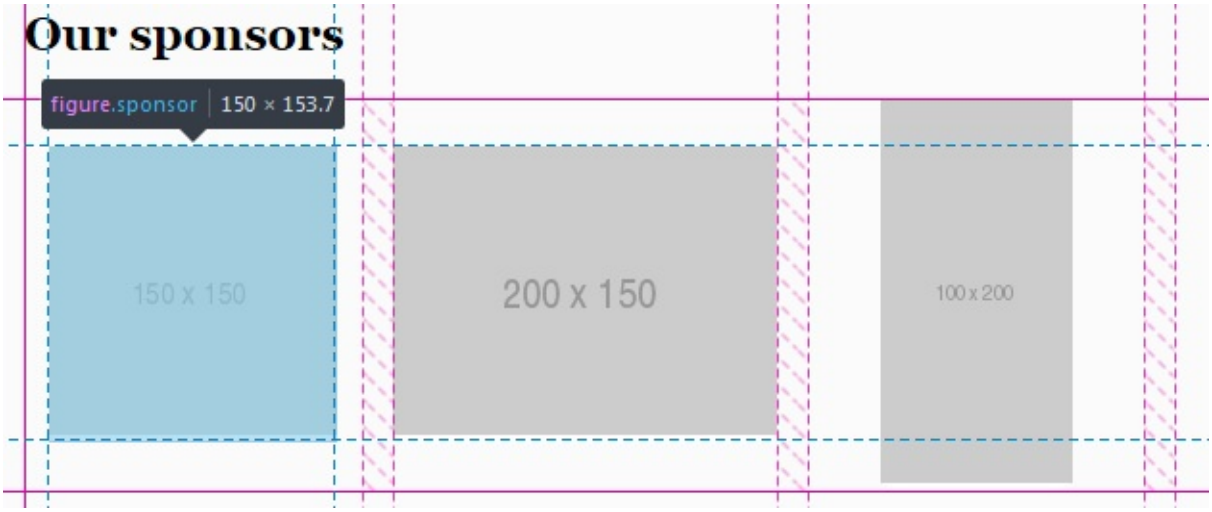
Now I would like to center the images both vertically and horizontally. We might try doing the following:

```
.sponsor {
   place-self: center;
 }
```

`place-self` aligns the element on X and Y axises. It is a shorthand property to `align-self` and `justify-self`.

The images will indeed be aligned but unfortunately the white background is gone. This is because each `.sponsor` now has width and height equal to the

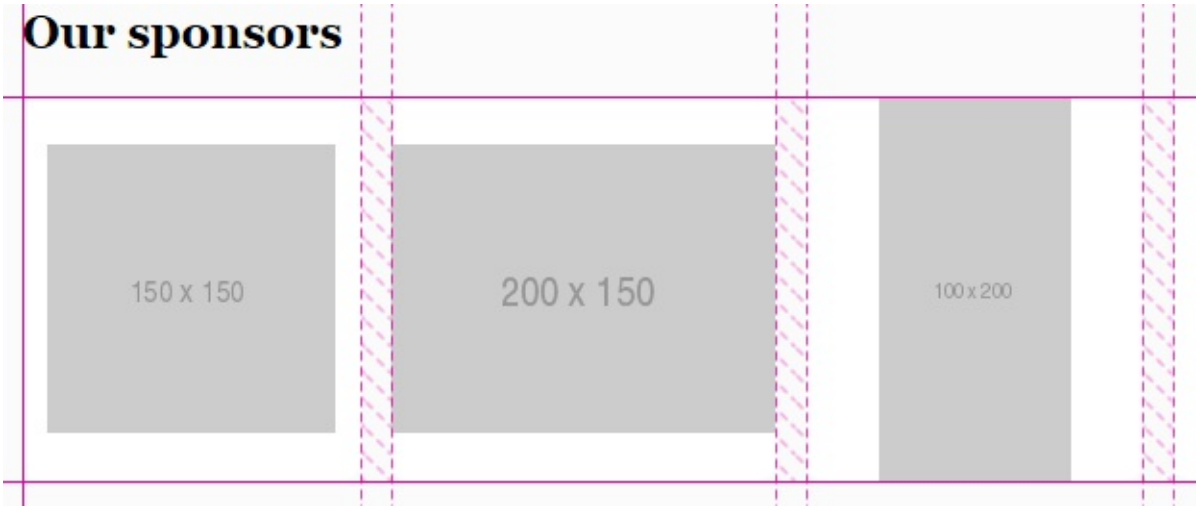
image's dimensions:



It means that we need a different approach here, and one of the possible solutions is to employ Flexbox:

```
.sponsor {
   // ...
   display: inline-flex;
   align-items: center;
   justify-content: center;
 }
```

Now everything is displayed properly, and now we know that Grid plays nicely with Flexbox:



Footer

Our last section is footer, and it is actually the simplest section. All we have to do is span it to all three columns:

```
.container {
   // ...
   grid-template-areas:
           "logo menu menu"
           "content content sidebar"
           "sponsors sponsors sponsors"
           "footer footer footer";
 }
```

Basically, the layout is finished! However, we are not done yet: the site also has to be responsive. So, let's take care of this task in the next section.

Making the Layout Responsive

Having CSS Grid in place, it is actually very easy to introduce responsiveness, because we can quickly reposition the areas.

Large Screens

Let's start with large screens (in this article I'll be sticking to the same breakpoints as defined in Bootstrap 4). I would like to decrease horizontal margin of the main container and the gap between individual sponsors:

```
@media all and (max-width: 992px) {
   .container {
     margin: 0 1rem;
   }

   .sponsors {
     grid-column-gap: 0.5rem;
   }
 }
```

Medium Screens

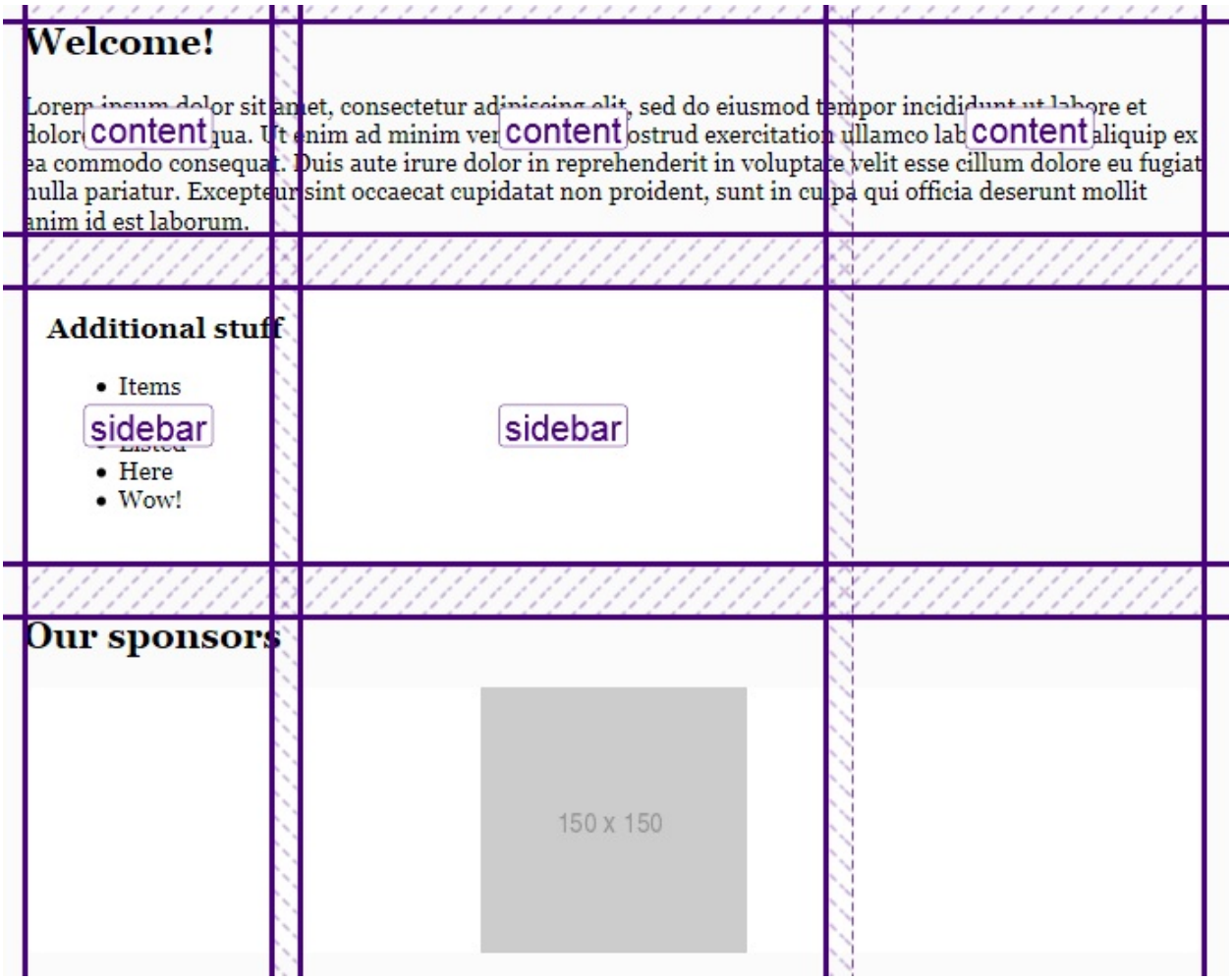
On the medium screens, I would like the main content area and the sidebar to occupy all three columns:

```
@media all and (max-width: 768px) {
   .container {
     grid-template-areas:
             "logo menu menu"
             "content content content"
             "sidebar sidebar sidebar"
             "sponsors sponsors sponsors"
             "footer footer footer";
   }
 }
```

Let's also decrease font size and stack the sponsors so they are displayed one beneath another. The gap between the columns should be zero (because actually there will be only one column). Instead, I'll set a gap between the rows:

```
@media all and (max-width: 768px) {
   // ...
   html {
     font-size: 14px;
   }

   .sponsors {
     grid-template-columns: 1fr;
     grid-column-gap: 0;
     grid-row-gap: 1rem;
   }
 }
```

This is how the site looks on medium screens now:



Small Screens

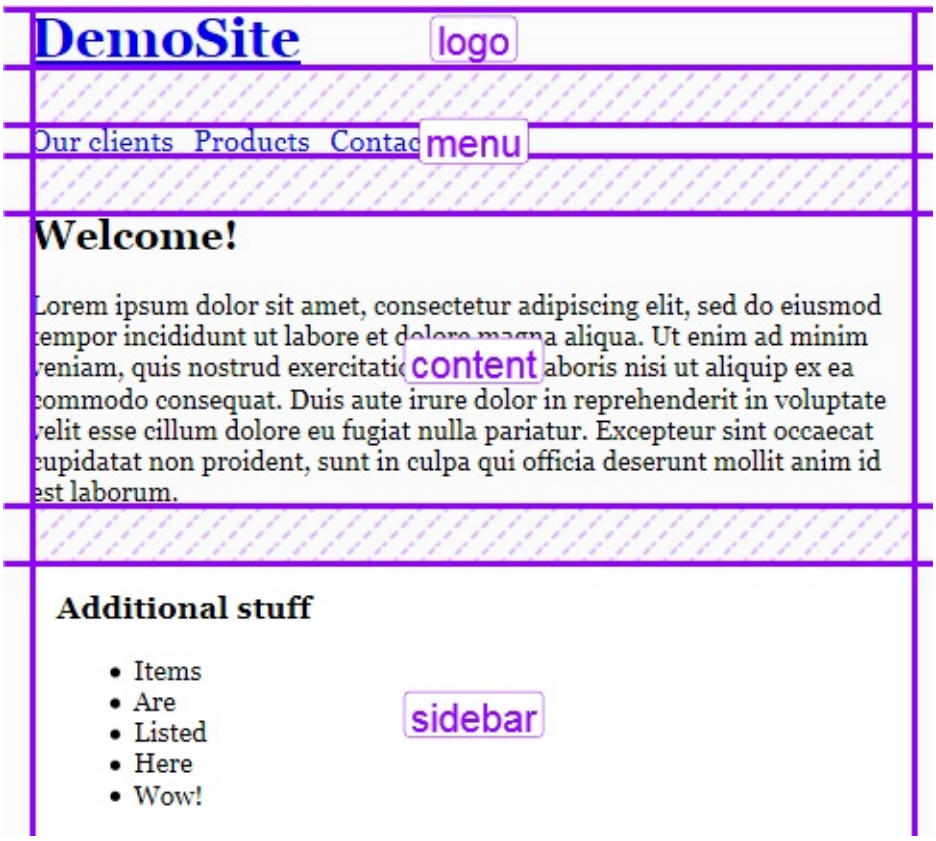
On small screens we are going to display each area on a separate row, which means that there will be only one column now:

```
@media all and (max-width: 540px) {
   .container {
     grid-template-columns: 1fr;
     grid-template-rows: auto;
     grid-template-areas:
             "logo"
             "menu"
             "content"
             "sidebar"
             "sponsors"
             "footer";
   }
 }
```

The menu should not be pulled to the right in this case, and we also don't need the gap between the columns:

```
@media all and (max-width: 540px) {
   .container {
     // ...
     grid-gap: 2rem 0;
   }

   .main-menu {
     justify-self: start;
   }
 }
```

The job is done:



Note that you may even rearrange the grid items easily for various screens. Suppose we would like to put the menu to the bottom on small screens (so that the visitors do not have to scroll up after they've finished reading material on the

page). To do that, simply tweak the `grid-template-areas`:

```
@media all and (max-width: 540px) {
   .container {
     // ...
     grid-template-areas:
             "logo"
             "content"
             "sidebar"
             "sponsors"
             "footer"
             "menu";
   }
 }
```

Without Media Queries?..

It is worth mentioning that we can make the sponsors block responsive without any media queries at all. This is possible with the help of `auto-fit` property and `minmax` function. To see them in action, tweak the styles for the `.sponsors` like this:

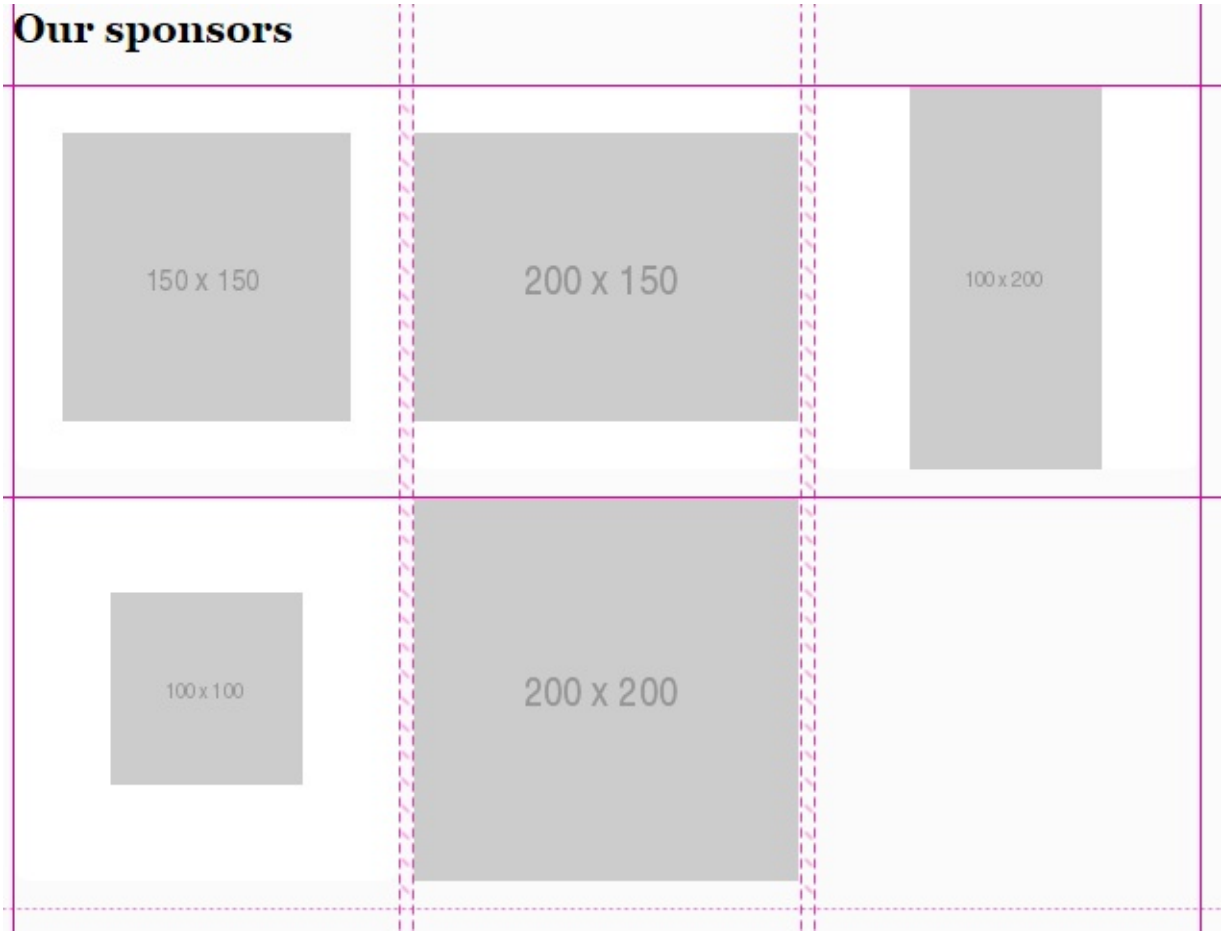
```
.sponsors {
   // ...
   grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
 }
```

`repeat` function, as you already know, repeats the columns as many times as necessary.

`auto-fill` means "fill the row with as many columns as possible". If there is not enough space for the column, it will be placed to the next line.

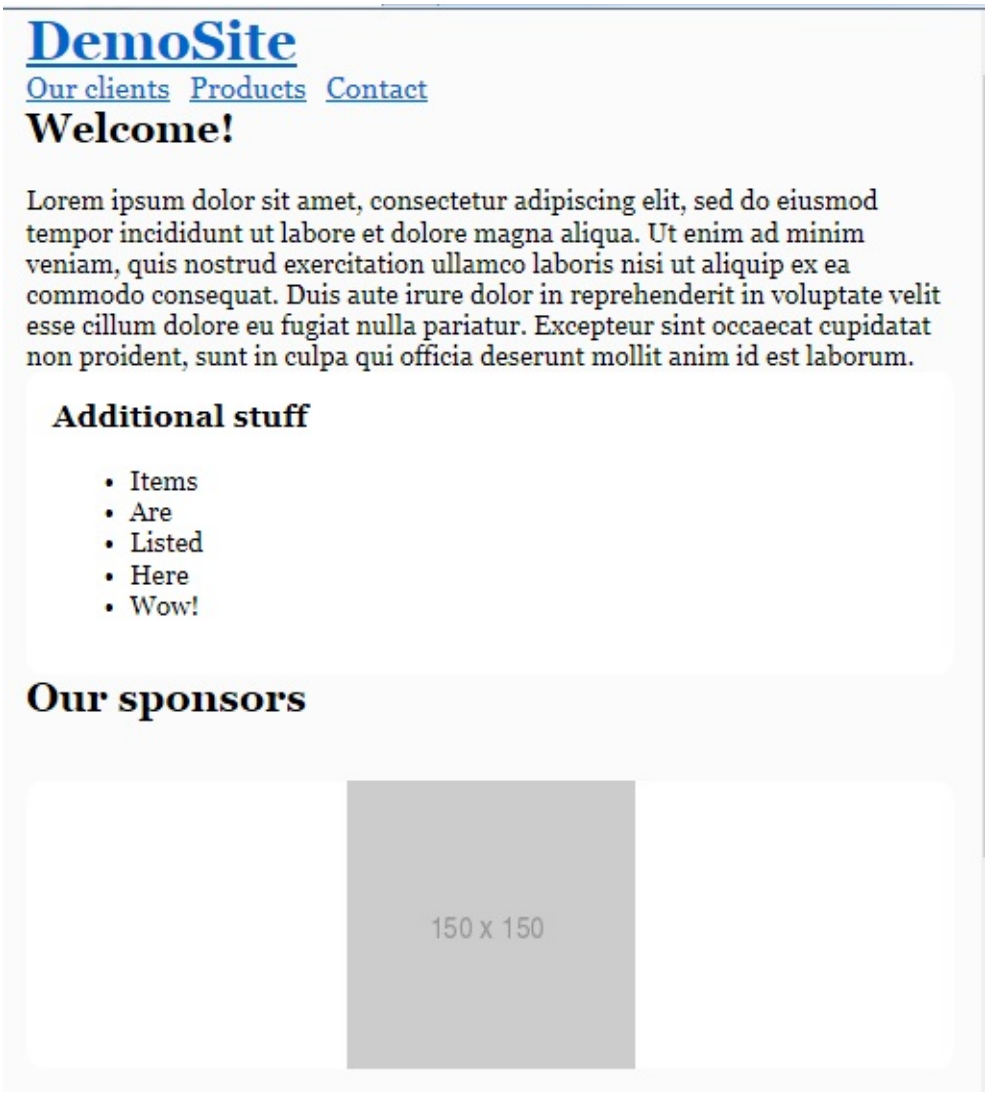
`minmax` allows us to specify the minimum and maximum value for the columns' widths. In this case each column should span 1 fraction of free space, but no less than 200 pixels.

Al this means that on smaller screens the columns may be shrinked down to at most `200px` each. If there is still not enough space, one or multiple columns will be placed to a separate line. Here is the result of applying the above CSS rules:



Fallbacks

Unfortunately, CSS Grid is not yet fully supported by all browsers, and you may guess which one is still implementing an older version of the specification. Yeah, it's Internet Explorer 10 and 11. If you open the demo in this browser, you'll see that the grid does not work at all, and the areas are simply stacked:



Of course, this is not the end of the world, as the site is still usable, but let's add at least some fallback rules. The good news is that if the element is floated and also has grid assigned, then the grid takes precedence. Also, the `display`, `vertical-align`, and some other properties also have no effect on grid items, so let's take advantage of that fact.

The stacked menu looks nice as is, but the sidebar should be probably placed next to the main content, not below it. We can achieve this by using `display: inline-block`:

```
.content-area {
   display: inline-block;
   vertical-align: top;
 }
```

```
 .sidebar {
   display: inline-block;
   vertical-align: top;
 }
```

In all browsers that support grid, these properties will have no effect, but in IE they'll be applied as expected. One more property we need to tweak is the `width`:

```
.content-area {
   width: 69%;
   display: inline-block;
   vertical-align: top;
 }

 .sidebar {
   width: 30%;
   display: inline-block;
   vertical-align: top;
 }
```

But having added these styles, our grid layout will now look much worse, because the `width` property is not ignored by grid items. This can be fixed with the help of `@supports` CSS query. IE does not understand these queries, but this is not needed anyways: we'll use it to fix the grid!

```
@supports (display: grid) {
   .content-area, .sidebar  {
     width: auto;
   }
 }
```

Now let's take care of the sponsor items and add some top margin for each block:

```
.sponsor {
   vertical-align: middle;
 }

 .main-menu, .content-area, .sidebar, .sponsors-wrapper, .footer  {
   margin-top: 2rem;
 }
```

We don't need any top margin when the grid is supported, so nullify it inside the `@supports` query:

```
@supports (display: grid) {
   // ...
   .main-menu, .content-area, .sidebar, .sponsors-wrapper, .footer,
     margin-top: 0;
   }
 }
```

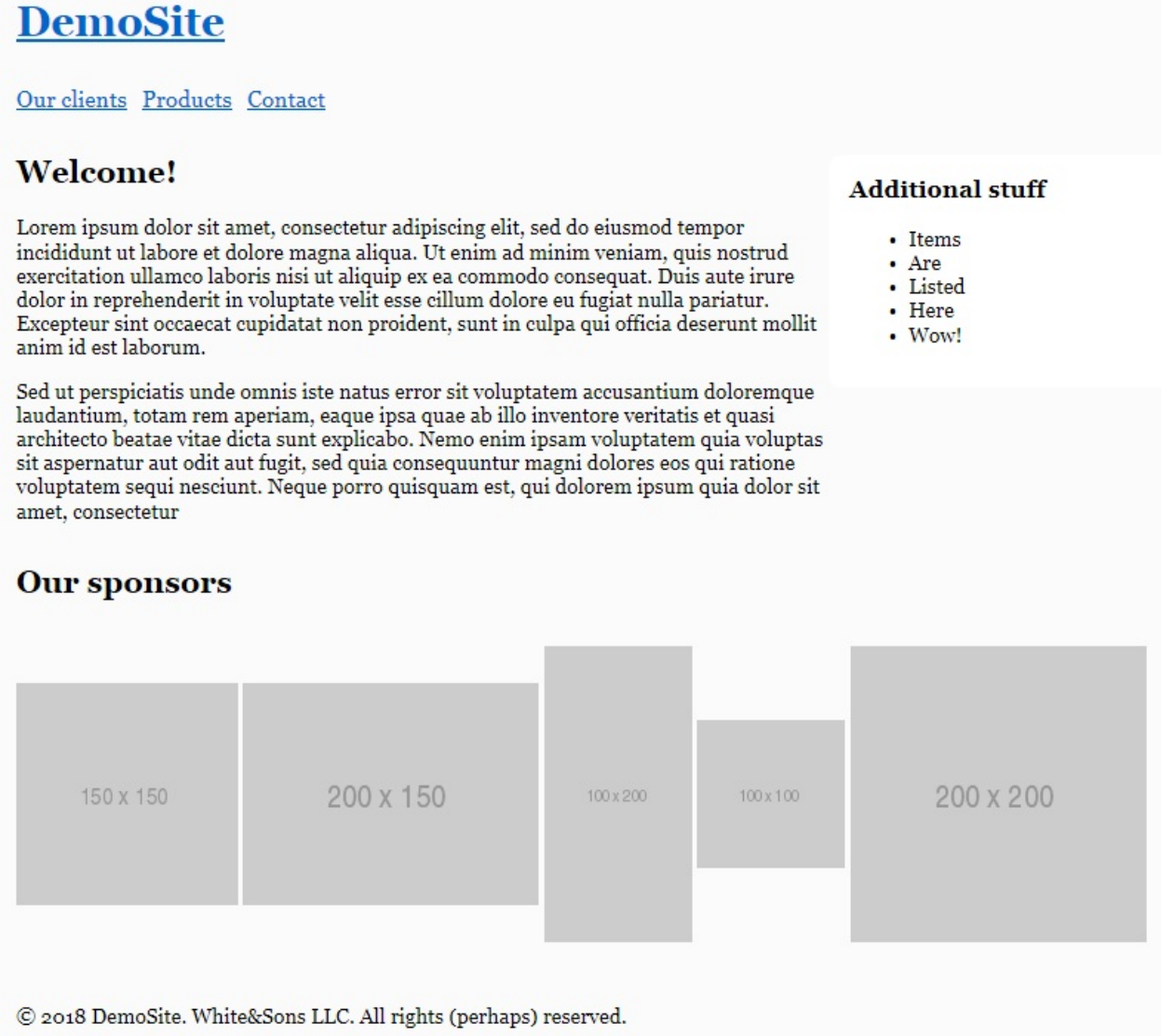
Lastly, let's add some responsiveness for IE. We'll simply stretch main content, sidebar, and each sponsor to full width on smaller screens:

```
@media all and (max-width: 760px) {
   .content-area, .sidebar {
     display: block;
     width: 100%;
   }

   .sponsor {
     width: 100%;
     margin-top: 1rem;
   }
 }
```

Don't forget to fix the sponsor's width for the browsers that support grid:

```
@supports (display: grid) {
   // ..

   .sponsor {
     width: auto;
   }
 }
```

Here is how the site looks in Internet Explorer now:



Live Code

See the Pen Multi-Column Layout With Floats.

Conclusion

In this article we have seen CSS Grid in action and utilized it to redesign an existing float-based layout. Comparing these two solutions, we can see that the HTML and CSS code of the "grid" solution is smaller (not counting the fallbacks, of course), simpler and more expressive. With the help of the `grid-template-areas` property it is easy to understand how individual areas are laid out, and we may quickly reposition them or adjust their sizes. On top of that, we

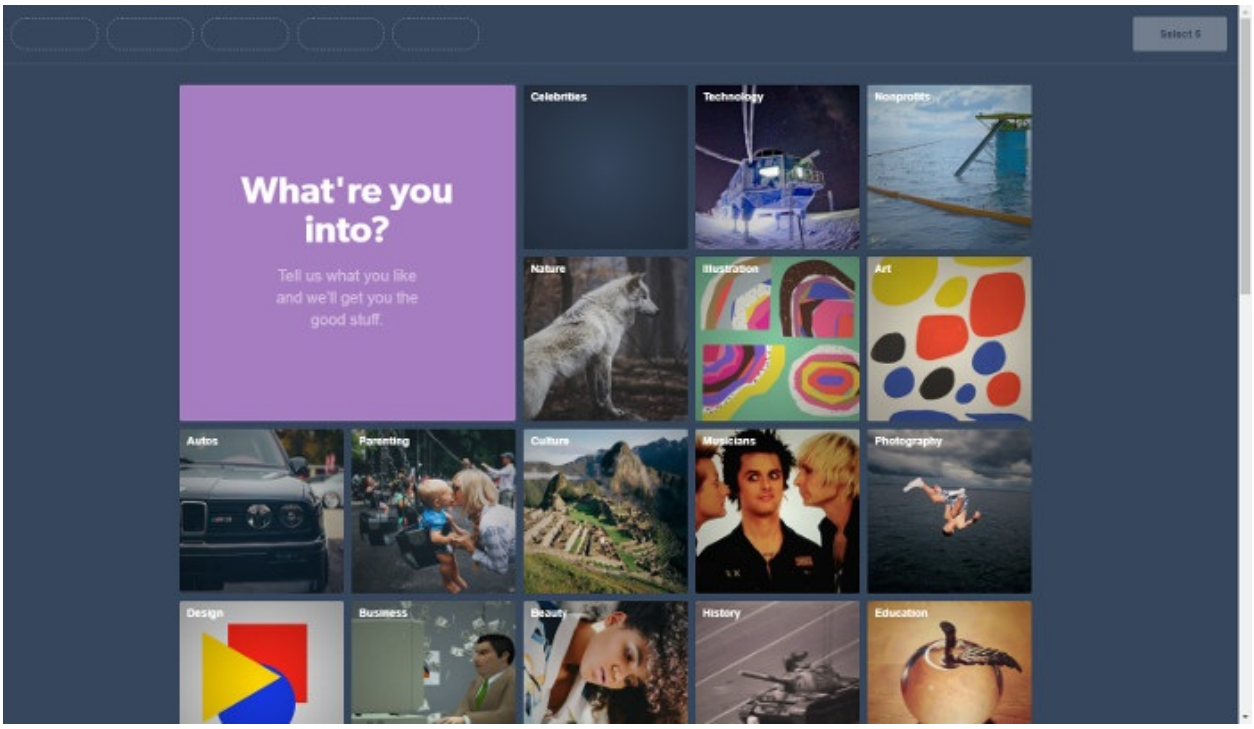
don't need to rely on various hacky tricks (like clearfix, for instance).

So, as you see, CSS Grid is a great alternative to floats, and it is production-ready. You may need to provide some fallback rules for Internet Explorer (that implements an older version of the specification), but as you see they are not very complex and in general the site is still usable even without any backwards compatibility at all.

Chapter 2: Redesigning a Card-based Tumblr Layout with CSS Grid

by Giulio Mainardi

In this tutorial we're going to re-implement a grid-based design concept inspired by the What're you into? Tumblr page, where the user can select a set of topics to tailor her recommended content.



Only the visual design of the grid is executed, not the selection functionality, as shown in the Pen we will be building:

Live Code

See the Pen MBdNav.

The main goal is to implement the design with CSS Grid, but a fallback layout with floats is outlined in the Support section below.

Markup

Essentially, the page content consists of list of cards:

```
<ul class="grid">
  <li class="card header">
    <h1>Which foods do you like?</h1>
    <p>Tell us what you crave for and we'll get you the tasty bits</
  </li>
  <li class="card">
    <a href="...">
      <h2>Pizza</h2>
      <img src="..." alt="A salami pizza">
    </a>
  </li>
  <!-- ... -->
</ul>
```

A card that represents a topic proposed to the user(food in our example) has a title and an illustrative image, both wrapped in a link element. Other s could be adopted, see for instance the excellent article on the card component on Inclusive Components, where the pros and cons of such alternatives are analyzed.

Structural Layout

In this section, the foundations of the grid design will be implemented. The next section will style the cards. This Pen shows the bare-bones layout using placeholders for grid items. Run it on a browser that supports CSS Grid.

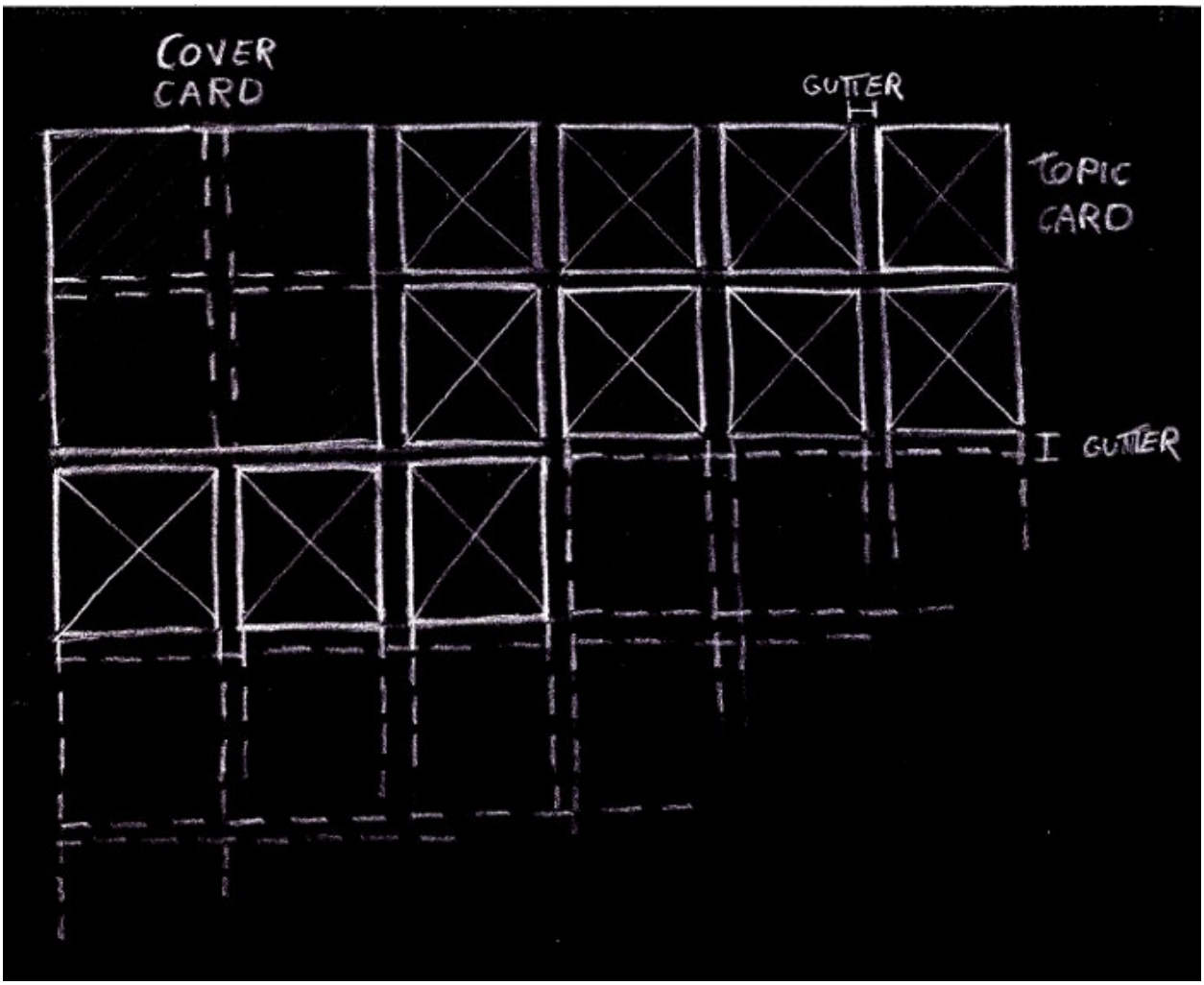
Live Code

See the Pen JBqgGm.

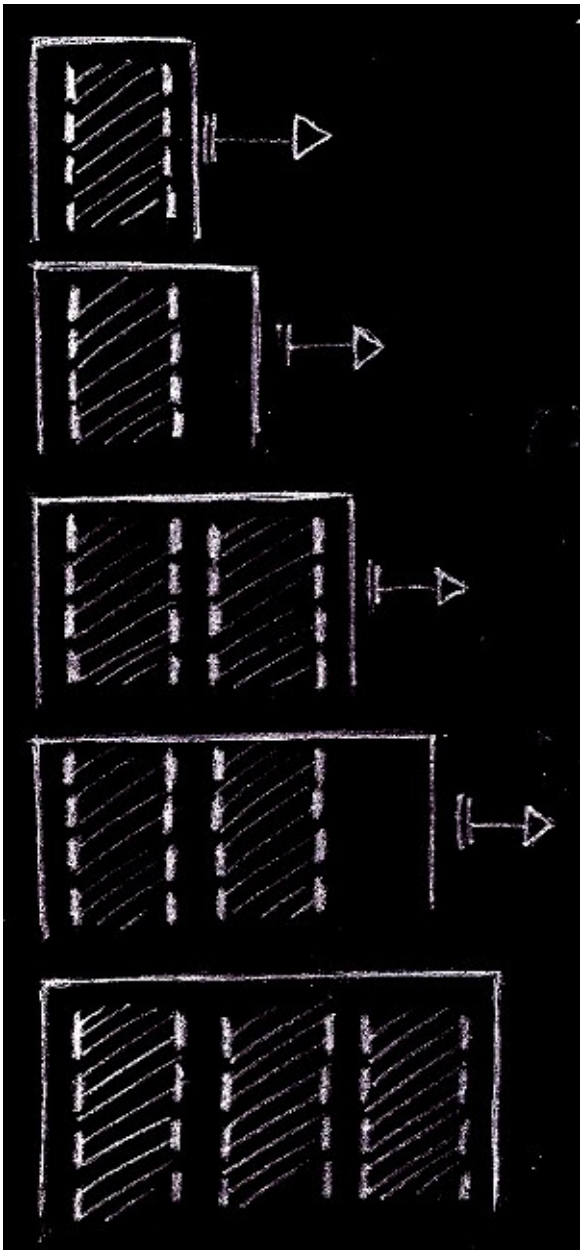
Before going ahead with the code, it is important to specify the features and the responsive behavior of the grid. Let's try to write down some properties it must satisfy.

Design Specs

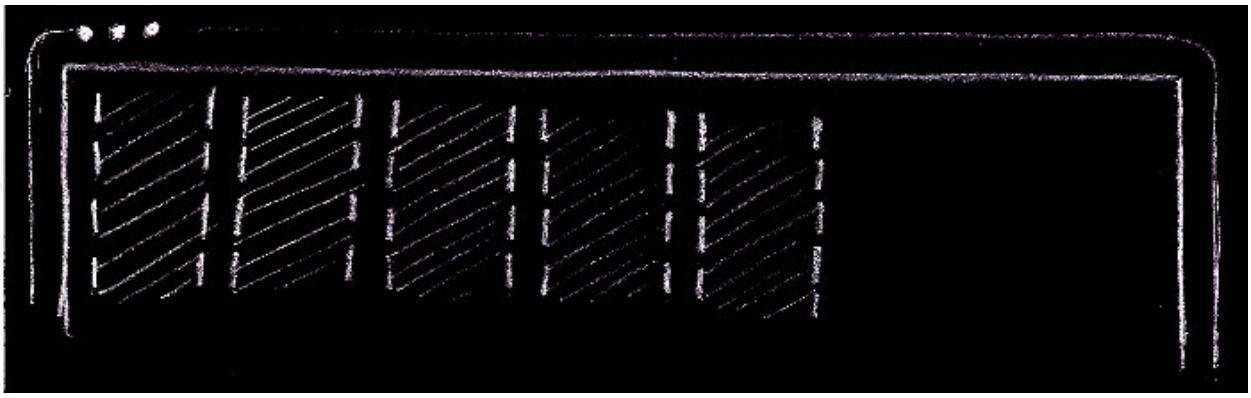
Two kinds of cards are featured in the design: a series of topic cards, and an introductory cover card. We arrange them on an underlying grid composed of square cells of fixed size. Each topic card occupies just one of these cells, while the cover spans a larger area of adjacent cells, whose extent depends on the viewport width. Furthermore, rows and columns are separated by the same fixed size gutter.



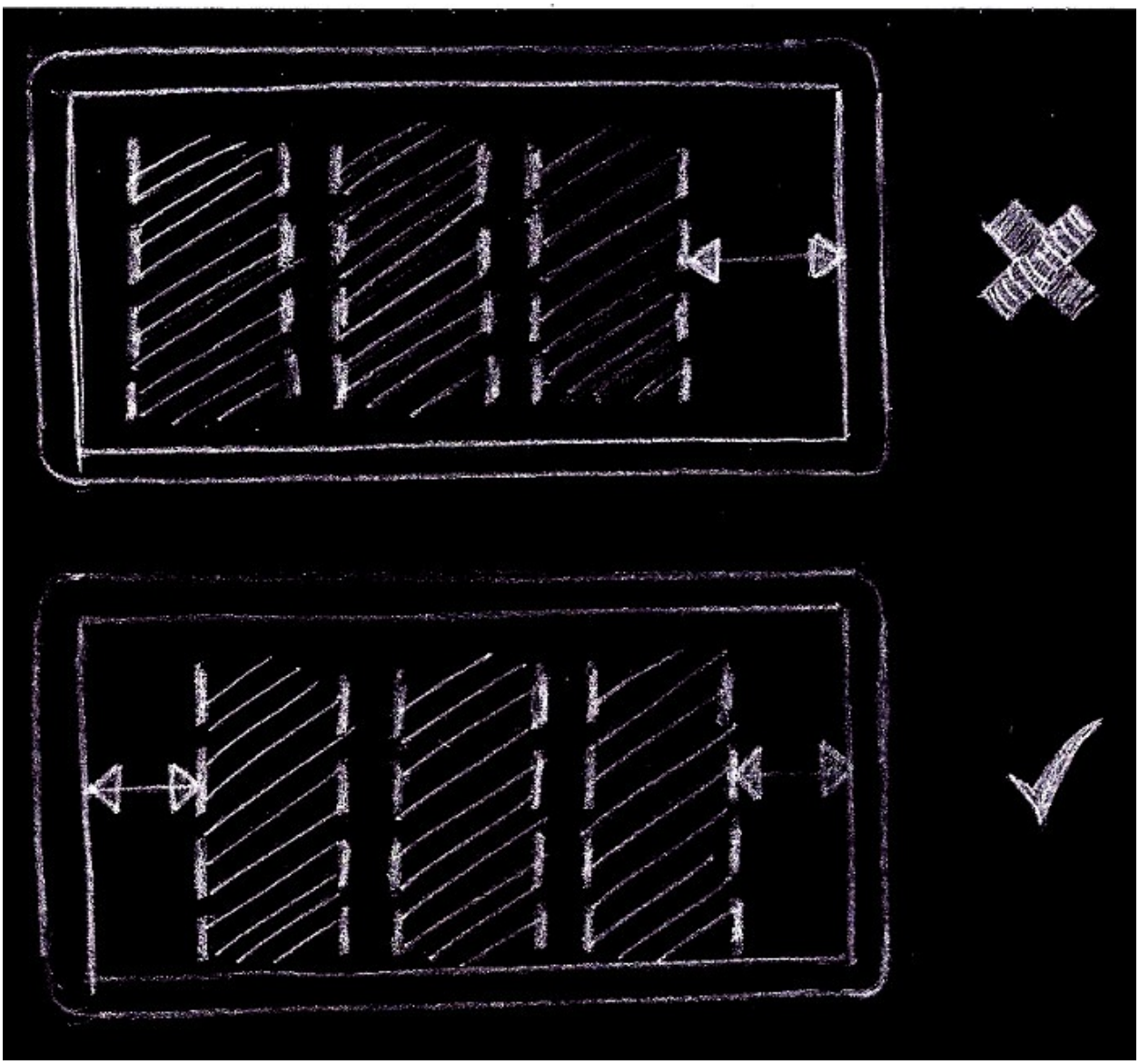
The grid has as many (fixed sized)columns as they fit into the viewport:



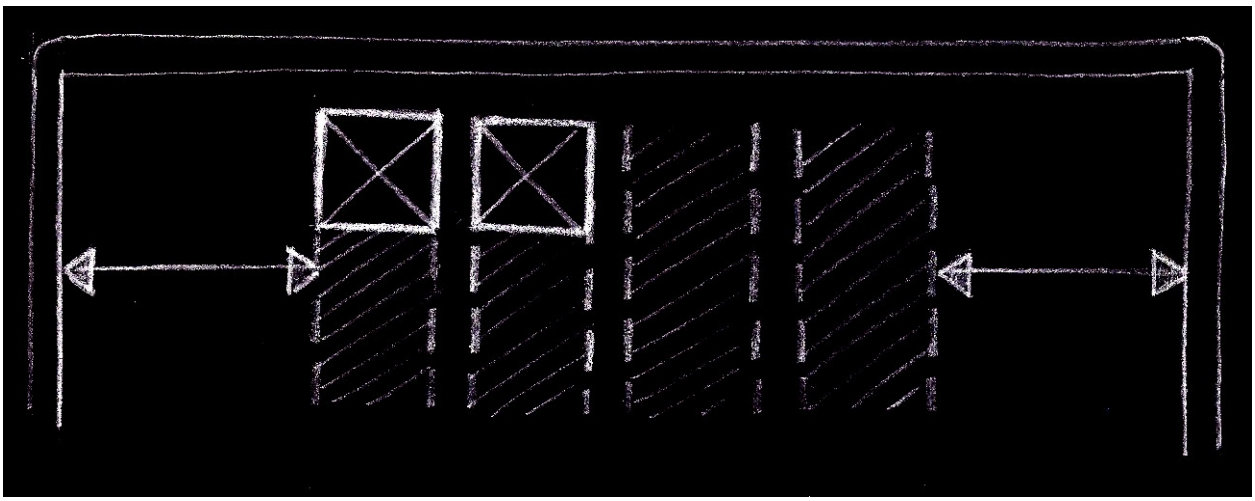
But we don't want a zillion columns on large screens, so let's limit the maximum number of columns:



The columns are always horizontally centered in the viewport.



Only the columns are centered, not the grid items. This means that the cards on an incomplete row are aligned to the left of the grid, not at the center of the viewport:



Check out these requirements in the above Pen. Also, it's useful to inspect the layout with the CSS Grid tools provided by some browsers, such as the Firefox's Grid Inspector.

Keeping in mind this checklist, we can fire up our favorite development environment and start coding.

Implementation

Let's introduce a couple of Sass global variables to represent the layout parameters defined in the specs, namely:

- `$item-size` for the size of the side of the grid cells
- `$col-gutter` for the gutter between the tracks of the grid
- `$vp-gutter` for a safety space to leave between the grid items and the viewport edges
- `$max-cols` for the maximum number of columns the grid can have

We could use CSS custom properties for these variables, avoiding the need of a preprocessor and allowing us to edit them with the in-browser development tools and watch the changes happen instantly. But we are going to use these values even for a fallback layout suitable for older browsers, where CSS variables are not supported. Moreover, we use expressions with these values even in media

query selectors, where custom properties and the `calc()` function are not fully available even on recent browsers.

```
$item-size: 210px;
$col-gutter: 10px;
$vp-gutter: $col-gutter;
$max-cols: 5;
```

We must establish a grid formatting context on the grid element:

```
.grid {
  display: grid;
}
```

The `grid-gap` property separates the grid tracks by the specified amount of space. But these gutters are only inserted between the tracks and not before the first track and after the last one. An horizontal padding on the grid container prevents the columns from touching the viewport edges:

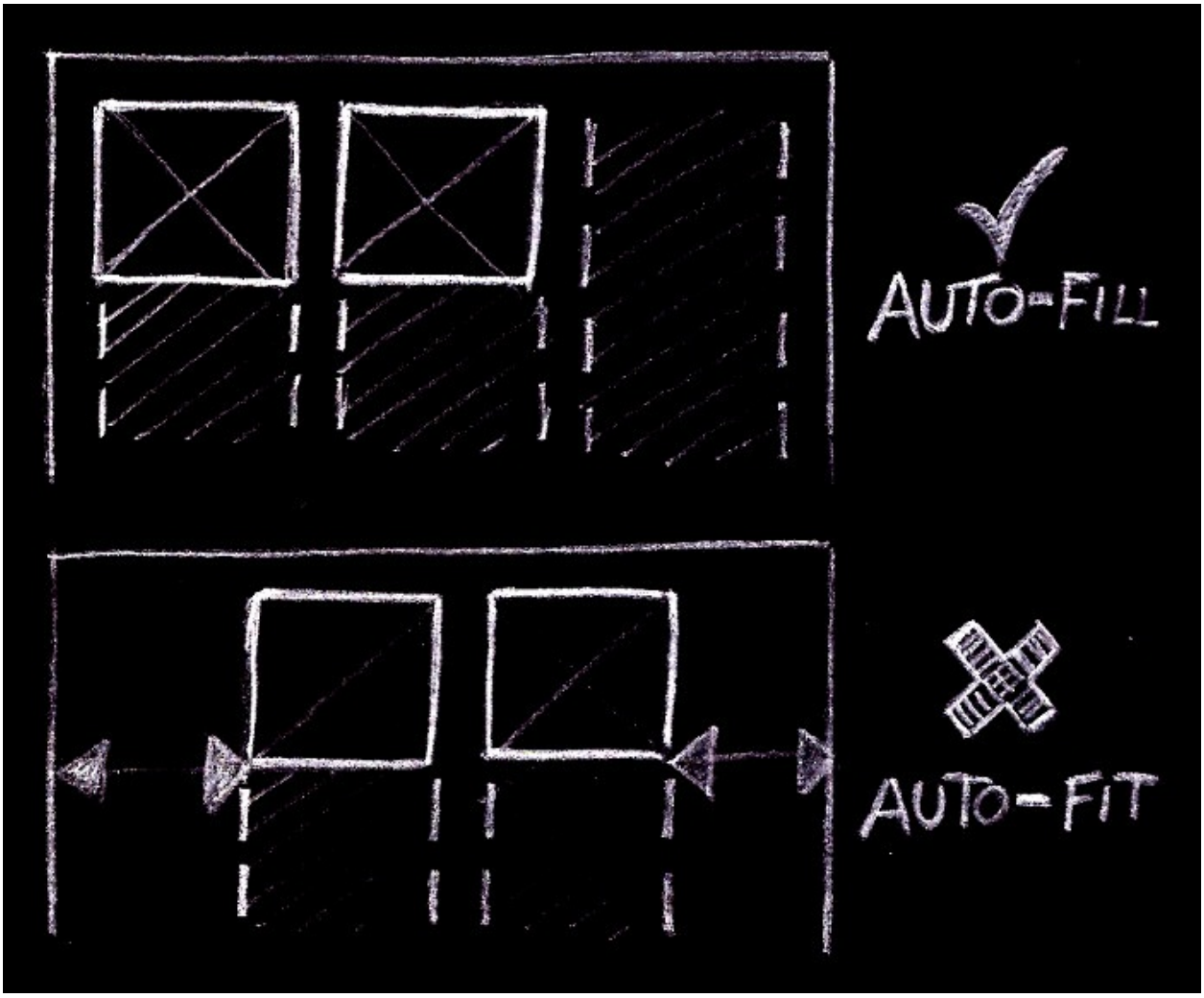
```
.grid {
  grid-gap: $col-gutter;
  padding: 0 $vp-gutter;
}
```

The columns of the grid can be defined with the `grid-template-columns` property and the `repeat` function with the `auto-fill` value as the repetition number and the `$item-size` variable for the track list argument:

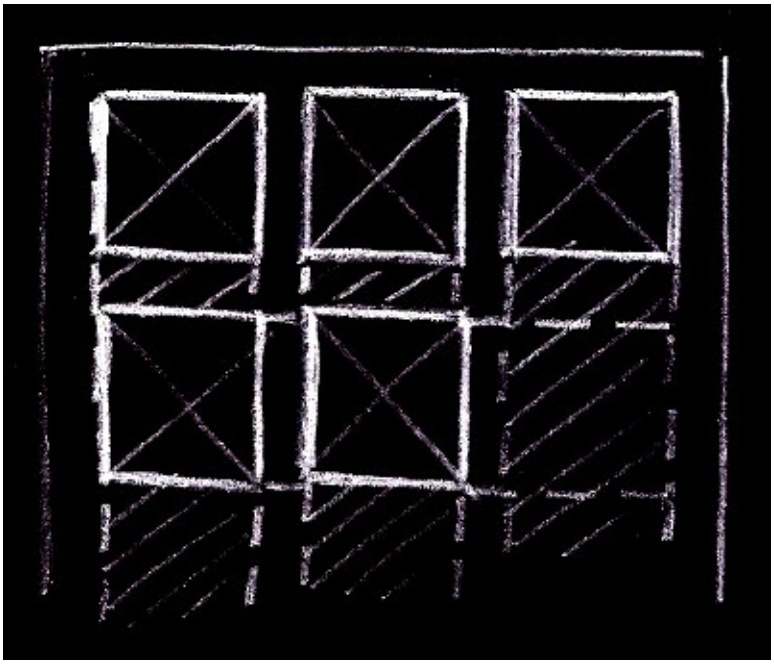
```
.grid {
  grid-template-columns: repeat(auto-fill, $item-size);
}
```

This tells the browser to fill the grid container(the `.grid` element) width with as many fixed size columns as possible, keeping account of the vertical gutters.

It is worth pointing out that we could have used the `auto-fit` mode, and in many combinations of viewport sizes and number of grid items we could not tell the difference with `auto-fill`. But when there are only a few items in the grid, with just an incomplete row, with `auto-fit` the items would be centered, instead of starting from the left of the grid, as detailed in the design specs. This happens because while with `auto-fill` the grid has always as many columns as possible, with `auto-fit` empty columns are removed, and the centering of the remaining columns places the items at the center of the viewport:



If the first row of the grid is complete, no columns are removed and there is no difference between the two modes:



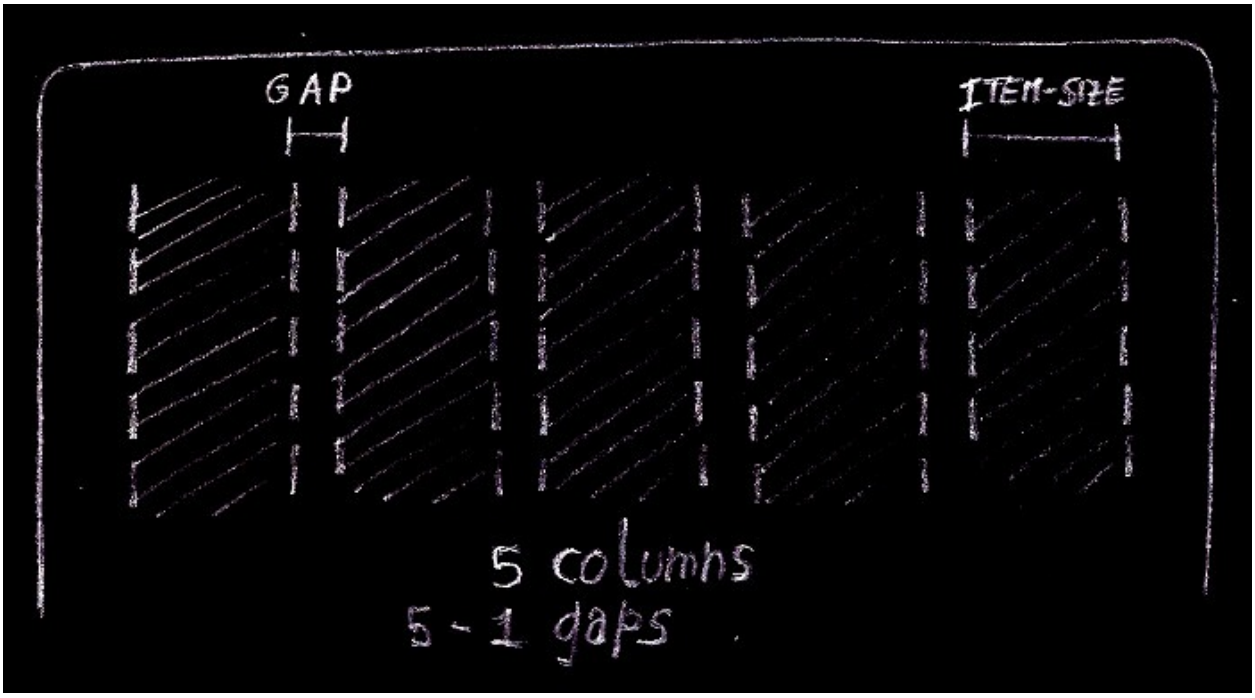
Back to the grid columns. Up to this point, the number of columns had no limit, it can arbitrarily grow as the viewport width increases. But according to the spec, the grid must have a maximum number of columns. It's possible to fix this with the `max-width` property:

```
.grid {
  max-width: grid-width($max-cols);
}
```

`grid-width` is a custom Sass function that returns the width of a grid with n columns:

```
@function grid-width($num-cols) {
  @return $num-cols * $item-size + ($num-cols - 1) * $col-gutter;
}
```

The first multiplication accounts for the size required by the columns, while the second one represents the space required by the gutters.



According to the specs, the grid must be always horizontally centered. We can combine the old auto margins trick with `justify-content` to accomplish this task:

```
.grid {
  justify-content: center;
  margin: 40px auto;
}
```

`justify-content` centers the columns when there is available space left inside the grid container. This happens when the container bleeds from one viewport edge to the other. The lateral auto margins centers the `.grid` container itself when it has reached its maximum width.

Now for the rows. They are not explicitly specified, as done with the columns. Rather, they are implicitly added by the browser as needed, and we just tell it their size with the `grid-auto-rows` property. Reusing the `$item-size` variable, each grid cell is shaped like a square, as per the specs:

```
.grid {
  grid-auto-rows: $item-size;
}
```

Let's move on by sizing the cards. On small viewports, when the grid configures

itself on a single column, the cover card spans only a grid cell, while when there are two or more columns, it must span a 4x4 grid area:

```
@include when-n-cols(2) {
  .grid .header {
    grid-row: span 2;
    grid-column: span 2;
  }
}
```

`when-n-cols()` is a Sass mixin to express a media query suitable for a grid with the given number of columns:

```
@mixin when-n-cols($n) {
  @media screen and (min-width: #{grid-width($n) + 2 * $vp-gutter +
    @content;
  }
}
```

The CSS rules represented by `@content` are active whenever the viewport width is equal or greater than the width of a grid with `$n` columns plus the two safety spaces to separate the grid items from the viewport edges. `$scrollbar-size` is just an upper bound on the size of a vertical scrollbar, to account for the fact that the width reported in the media query is the entire viewport width, including an eventual vertical scrollbar.

Regarding the topic cards, there is nothing to do, because the default Grid behavior makes them the same size as their assigned grid cells.

Ok, we got it! Refer to the structural Pen at the beginning of this section to see all these code snippets put together in the complete code.

The Cards

Here we build the cards, their inner content to be more precise.

Let's address the topic cards first. To make them clickable, the link element is expanded to fill the entire card area:

```
.grid .card a {
  display: block;
  width: 100%;
```

```
  height: 100%;

  position: relative;
}
```

Then we make sure the card image covers all the card surface:

```
.grid .card img {
  display: block;
  width: 100%;
  height: 100%;
  object-fit: cover;
}
```

In the example, the thumbnails have a square aspect ratio, therefore they scale nicely inside their grid items. To handle arbitrary image sizes, `object-fit: cover` scales(preserving the aspect ratio) and eventually clips the image to fit inside the container.

It's the turn of the card title:

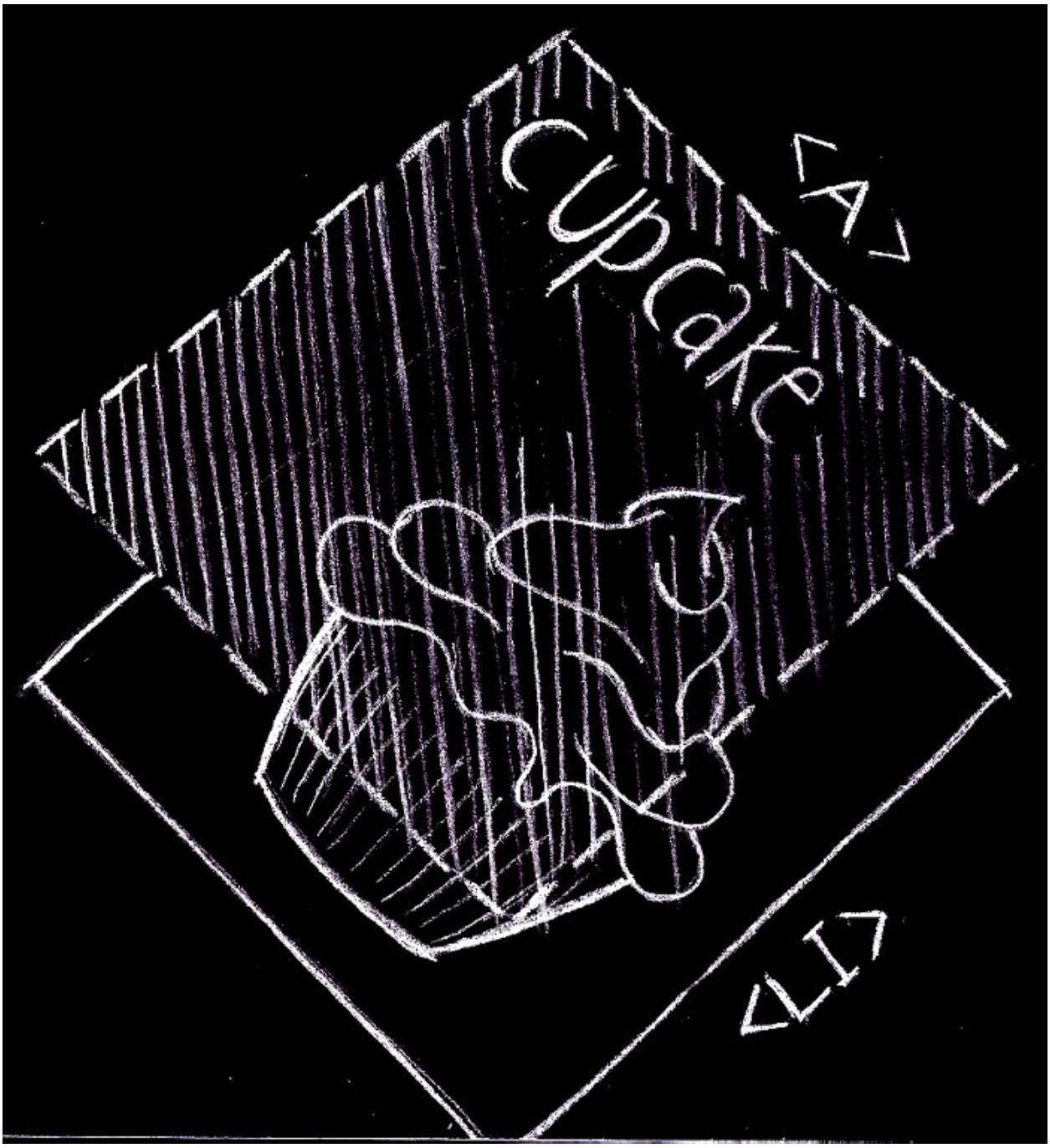
```
.grid .card h2 {
  margin: 0;

  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;

  padding: 10px;

  text-decoration: none;
  font-family: Raleway, sans-serif;
  font-size: 1em;
  letter-spacing: 1px;

  color: #fff;
}
```

With the absolute positioning the heading element is removed from the flow and positioned above the image, at the top left corner of the card.

In order to improve the contrast between the label and an arbitrary underlying card image, a partially transparent layer is sandwiched between these two graphic elements. Here I'm using the same exact technique employed on the

original Tumblr page, where this overlay consists in a radial gradient which starts completely transparent at the center of the card and ends with a partially opaque black towards the borders, in a circular fashion, giving the image a sort of subtle spotlight effect. Let's render this layer as a background image of the card link, which it has just been extended to cover all the card surface:

```
.grid .card h2 {
  background-image: radial-gradient(ellipse at center, transparent 0
}
```



As for the cover card, plenty of techniques could be used here, but let's keep it simple. The card features two centered blocks of text. With a side padding their horizontal extent is limited and with `text-align` their content is centered. After that, the blocks are vertically centered just by pushing them down with a bit of top padding applied to the card container.

```
.grid .header {
  box-sizing: border-box;
```

```
  text-align: center;
  padding-top: 23px;

  font-size: 1.6875em;
  line-height: 1.3;

  background: radial-gradient(ellipse at center, transparent 0, rgba
}

.grid .header h1 {
  margin-top: 0;
  margin-bottom: 0.67em;

  font-family: 'Cherry Swash', cursive;
  text-transform: capitalize;
  font-size: 1em;
  font-weight: normal;

  padding-left: 28px;
  padding-right: 28px;
}

.grid .header p {
  margin: 0;

  font-family: 'Raleway', sans-serif;
  font-size: 0.52em;

  padding-left: 34px;
  padding-right: 34px;
}
```

With a media query the font size is increased and the padding adjusted when the grid displays two ore more columns:

```
@include when-n-cols(2) {
  .grid .header {
    font-size: 2.5em;

    padding-top: 100px;
  }

  .grid .header h1 {
    padding-left: 80px;
    padding-right: 80px;
  }

  .grid .header p {
```

```
    padding-left: 120px;
    padding-right: 120px;
  }
}
```

It's the time to add some interactivity to the topic cards. They must reduce the size on hover:

```
.grid .card:hover {
  transform: scale(0.95);
}

.grid .header:hover {
  transform: none;
}
```

With a CSS transition this change of visual state is smoothed out:

```
.grid .card {
  transition-property: transform;
  transition-duration: 0.3s;
}
```

The cards can be navigated with a keyboard, so why don't customize their focus style to match the same look&feel of the mouse over? We must scale a .card container when the `A` link inside it receive the focus. Mmmhh...we need to style an element only when one of its children get the focus...wait a moment...there's the :focus-within pseudo-class for that:

```
.grid .card a:focus {
  outline: none;
}
.grid .card:focus-within {
  transform: scale(0.95);
}
```

First the default link focus style is reset, and then the same transform as before is used when the card(its link) gets the focus.

Support

CSS Grid

Nowadays CSS Grid has a wide support, but what can we do with the other browsers? In the demo pen there is a fallback layout, implemented with floats, that behaves exacly as the Grid layout. Let's have a quick look at how it works and interacts with the Grid implementation.

The cards are sized, floated to the left and have a bottom margin for the horizontal grid gutter:

```
.card {
  float: left;
  width: $item-size;
  height: $item-size;
  margin: 0;
  margin-bottom: $col-gutter;
}
```

At this point, we could just set a `max-width` on the `.grid` container to limit the number of columns on large screen, use auto margins to center the grid, and the layout would be almost the same as the Grid one, but with the important difference than when the container doesn't reach its maximum width, the cards are aligned to the left of the viewport. This is a fallback layout for browsers not supporting CSS Grid, so we could be happy with this. Else, we could go on and add some more code to fix this difference. Let's have a try.

We set the width of the `.grid` container when there is only one column, center it into the viewport, and separate it from the screen edges:

```
.grid {
  width: grid-width(1);
  margin: 40px auto;
  padding-left: $vp-gutter;
  padding-right: $vp-gutter;
}
```

Note how we reused the `grid-width` Sass function introduced above.

Now, reusing the media query mixin, we define the width of the `.grid` container when there are 2 columns:

```
@include when-n-cols(2) {
  width: grid-width(2);
  .card:nth-child(2n) {
    margin-right: $col-gutter;
  }
```

```
  .header {
    $header-size: grid-width(2);
    width: $header-size;
    height: $header-size;
  }
}
```

We also doubled the size of the header card and assigned the right margin for the grid horizonal gutter.

This pattern is repeated for a grid of 2, 3, ..., `$max-cols` columns, taking care of resetting and assigning the `margin-right`'s of the proper cards. For instance, when the grid has three columns:

```
@include when-n-cols(3) {
  width: grid-width(3);
  .card {
    margin-right: $col-gutter;
  }
  .card:nth-child(2),
  .card:nth-child(3),
  .card:nth-child(3n + 6) {
    margin-right: 0;
  }
}
```

Please refer to the pen for the rest of code.

Now that we have two blocks of CSS which implement the same layout on the same page, we must be sure that they don't conflict with each other. In particular, on a browsers that supports Grid, the float layout must not disturb the Grid layout. Thanks to the inherent CSS Grid overriding capabilities, it is sufficient to reset the grid container width, to set its `max-width`, and to reset the margins of the cards(`grid-gap` takes care of the grid gutters now):

```
@supports (display: grid) {
  .grid {
    width: auto;
    max-width: grid-width($max-cols);
    .card {
      margin: 0 !important;
    }
  }
}
```

These overrides have to occur only when CSS Grid is supported, else they would break the float layout. This conditional overriding is performed with the `@support` at-rule.

The code in the pen is organized so that these two blocks of code for the layout are independent, that is, one of them can be removed without affecting the other. To achieve this, the CSS rule common to both layout implementations were grouped together:

```
.grid {
  margin: 0;
  padding: 0;
  list-style: none;

  margin: 40px auto;
  padding-left: $vp-gutter;
  padding-right: $vp-gutter;
}
```

So, if some day we want to get rid of the float layout, it is easy to do so.

To end this discussion on the fallback layout, note how its code looks more complicated and not intuitive as the CSS Grid. This highlights the power of a CSS technique specifically developed for layout, as opposed to an older CSS feature(floats) whose original intent was (ab)used for many years for the lack of more specific methods.

Other Features

Regarding `:focus-within`, it is not supported on Microsoft browsers. With the CSS defined above, on these user agents the user misses the visual feedback when a grid item link receives the focus. Here is a way to fix this:

```
.grid .card a:focus {
  outline: 2px solid #d11ee9;
}
.grid .card:focus-within {
  transform: scale(0.95);
}
.grid .card:focus-within a:focus {
  outline: none;
}
```

A not-supporting browser uses only the first rule, which customize the look of the default focus outline. Even a supporting browser uses this rule, but the outline is reset again in the third line.

Another alternative is to use the `:hover` and `:focus` pseudo classes on the link element, not on the card root element. In fact, for more generality, I preferred to scale the entire card, but in this particular case, where the link stretches to cover the full extent of the card, we could just do this:

```
.grid .card a:hover {
  transform: scale(0.95);
}
.grid .card a:focus {
  outline: none;
  transform: scale(0.95);
}
```

This way, there is no need of `focus-within`, and the card on focus behaves the same way as when it is hovered, even in older browsers.

object-fit has some troubles on Microsoft browsers too, hence, if the images have not the same aspect ratio as the cards, they may appear stretched.

Finally, on IE 9 and below the CSS gradients are not implemented. As a consequence, the transparent layer which improves the contrast between the card title and the underlying image is not visible. On IE 9 this can be fixed by adding a background color specified with the `rgba()` color function:

```
.grid .card h2 {
  background: rgba(0,0,0, 0.2);
  background: radial-gradient(ellipse at center, transparent 0, rgba
}
```

Final Words

We have already pointed out the power of CSS Grid in the previous section. Furthermore, we may also note how the responsive behavior of the grid outlined in the design specs was achieved without media queries. In fact, the only two queries utilized in the previous snippets were both for the cover card, one to assign the card to its grid area, and the other for its content.

So, after some coding, and lots of pizzas and cakes, we have come to the end. But an end is just a beginning of something else, and we could go ahead with further work, such as adding some entrance animations to the cards, performing an accessibility review, and trying to carry out a selection functionality similar to the original Tumblr page.

Thanks to Freepik for the tasty pictures. Follow the card links to view the original images.

Chapter 3: Easy and Responsive Modern CSS Grid Layout

by Ahmed Bouchefra

In a previous article we explored four different techniques for easily building responsive grid layouts. This article was written back in 2014 before CSS Grid was available so in this tutorial, we'll be using a similar HTML structure but with modern CSS Grid layout.

Throughout this tutorial, we'll create a demo with a basic layout using floats and then enhance it with CSS Grid. We'll demonstrate many useful utilities such as centering elements, spanning items, and easily changing the layout on small devices by redefining grid areas and using media queries.

Live Code

You can find the code in this CodePen.

Before, we dive into creating our responsive grid demo, let's first introduce CSS Grid.

CSS Grid is a powerful 2-dimensional system, which was added to most modern browsers in 2017, that's dramatically changing the way we are creating HTML layouts. Grid Layout allows us to create grid structures in CSS and not HTML.

CSS Grid is supported in most modern browsers except for IE11 which supports an older version of the standard that could give a few issues. You can use caniuse.com to check for support.

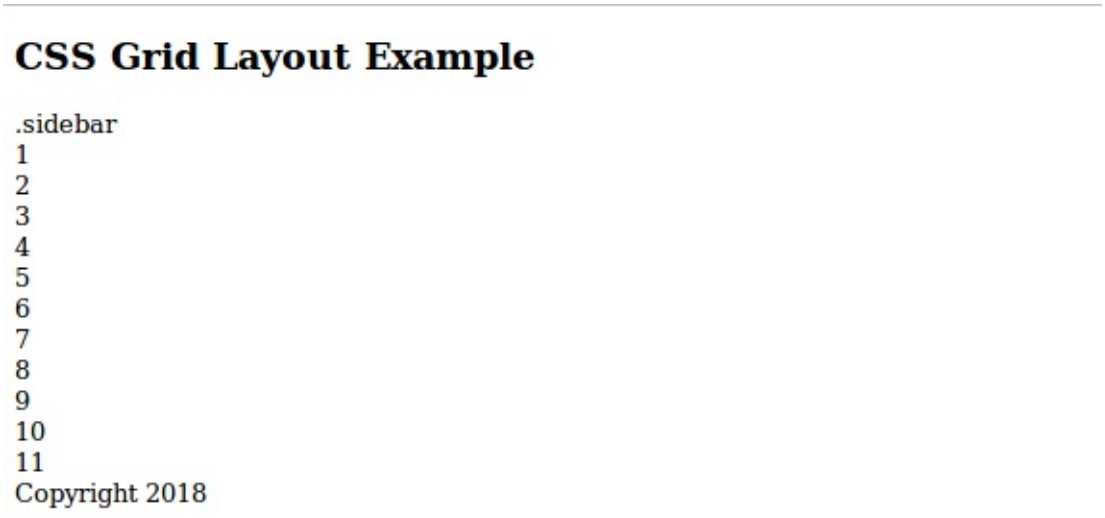
A Grid Layout has a parent container with the `display` property set to `grid` or `inline-grid`. The child elements of the container are grid items which are implicitly positioned thanks to a powerful Grid algorithm. You can also apply different classes to control the placement, dimensions, position and other aspects of the items.

Let's start with a basic HTML page. Create an HTML file and add the following content:

```
<header>
    <h2>CSS Grid Layout Example</h2>
</header>
<aside>
  .sidebar
</aside>

<main>
  <article>
    <span>1</span>
  </article>
  <article>
    <span>2</span>
  </article>
  <!--... -->
  <article>
    <span>11</span>
  </article>
</main>

<footer>
  Copyright 2018
</footer>
```

We use HTML semantics to define the header, sidebar, main and footer sections of our page. In the main section, we add a set of items using the `<article>` tag. `<article>` is an HTML5 semantic tag that could be used for wrapping independent and self-contained content. A single page could have any number of `<article>` tags.

This is a screen shot of the page at this stage:



Next, let's add basic CSS styling. Add a `<style>` tag in the head of the document and add the following styles:

```
body {
  background: #12458c;
  margin: 0rem;
  padding: 0px;
  font-family: -apple-system, BlinkMacSystemFont,
            "Segoe UI", "Roboto", "Oxygen", "Ubuntu", "Cantarell",
            "Fira Sans", "Droid Sans", "Helvetica Neue",
            sans-serif;

}
header {
  text-transform: uppercase;
  padding-top: 1px;
  padding-bottom: 1px;
  color: #fff;
  border-style: solid;
  border-width: 2px;
}
aside {
  color: #fff;
  border-width:2px;
  border-style: solid;
  float: left;
  width: 6.3rem;
}
footer {
  color: #fff;
  border-width:2px;
```

```
  border-style: solid;
  clear: both;
}
main {
  float: right;
  width: calc(100% - 7.2rem);
  padding: 5px;
  background: hsl(240, 100%, 50%);
}
main > article {
  background: hsl(240, 100%, 50%);
  background-image: url('https://source.unsplash.com/daily');
  color: hsl(240, 0%, 100%);
  border-width: 5px;
}
```

This is a small demonstration page so we'll style tags directly to aid readability rather than applying class naming systems.

We use floats to position the sidebar to the left and the main section to the right and we set the width of the sidebar to a fixed *6.3rem* width then we calculate and set the remaining width for the main section using CSS `calc()` function. The main section contains a gallery of items organized as vertical blocks.

The layout is not perfect. For example, the sidebar does not have the same height as the main content section. There are various CSS techniques to solve the problems but most are hacks or workarounds. Since this layout is a fallback for Grid, it will be seen by a rapidly diminishing number of users. The fallback is usable and good enough.

The latest versions of Chrome, Firefox, Opera and Safari have support for CSS Grid so that means if your visitors are using these browsers you don't need to worry about providing a fallback. Also you need to account for evergreen browsers. The latest versions of Chrome, Firefox, Edge, and Safari are **evergreen browsers**, i.e. they automatically update themselves silently without

prompting the user. To ensure your layout works in every browser, you can start with a default float-based fallback then use Progressive Enhancement techniques to apply a modern Grid layout. Those with older browsers will not receive an identical experience but it will be good enough.

Progressive Enhancement: You Don't Have to Override Everything

When adding the CSS Grid layout on top of your fallback layout, you don't actually need to override all tags or use completely separate CSS styles:

- In a browser that doesn't support CSS Grid, the grid properties you add will be simply ignored.
- If you are using floats for laying out elements, keep in mind that a grid item takes precedence over float i.e if you add a `float: left|right` style to an element which is also a grid element (a child of a parent element that has a `display: grid` style) the float will be ignored in favor of grid.
- Specific feature support can be checked in CSS using @supports rules. This allows us to override fallback styles where necessary while older browsers ignore the @supports block.

Now, let's add CSS Grid to our page. First let's make the `<body>` a grid container and set the grid columns, rows and areas:

```
body {
  /*...*/
  display: grid;
  grid-gap: 0.1vw;
  grid-template-columns: 6.5rem 1fr;
  grid-template-rows: 6rem 1fr 3rem;
  grid-template-areas: "header   header"
                       "sidebar content"
                       "footer footer";
}
```

We use `display:grid` property to mark `<body>` as a grid container. We set a grid gap of *0.1vw*. Gaps lets you create gutters between Grid cells instead of using margins.

We also use `grid-template-columns` to add two columns, the first column takes a fixed width which is *6.5rem* and the second column takes the remaining width.

`fr` is a fractional unit and *1fr* equals 1 part of the available space.

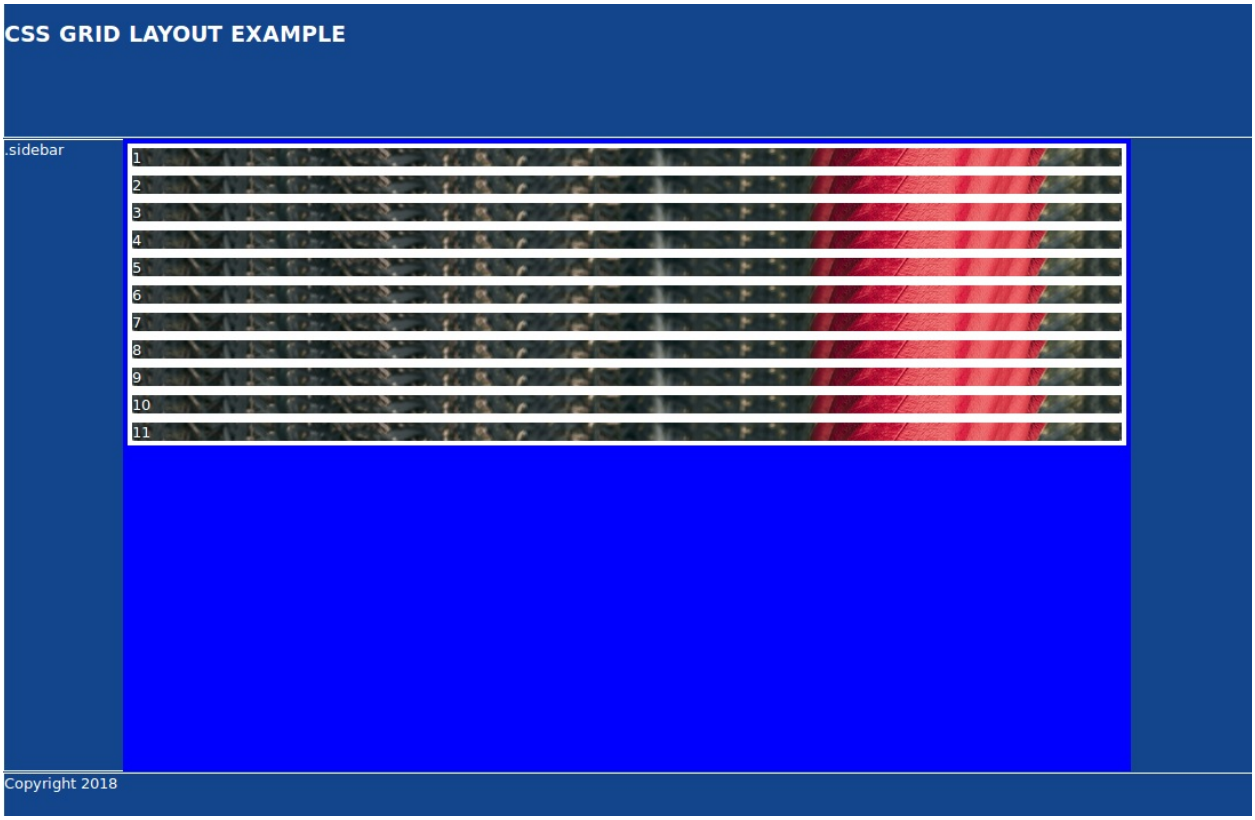
Next, we use `grid-template-rows` to add three rows: The first rows takes a fixed height which equals *6rem*, the third row takes a fixed height of *3rem* and the remaining available space (*1fr*) is assigned to the second row.

We then use `grid-template-areas` to assign the virtual cells, resulted from the intersection of columns and rows, to areas. Now we need to actually define those areas specified in the areas template using `grid-area`:

```
header {
  grid-area: header;
  /*...*/
}
aside {
  grid-area: sidebar;
  /*...*/
}
footer {
  grid-area: footer;
  /*...*/
}
main {
  grid-area: content;
  /*...*/
}
```

Most of our fallback code doesn't have any side effects on the CSS Grid except for the width of the main section `width: calc(100% - 7.2rem);` which calculates the remaining width for the main section after subtracting the width of the sidebar plus any margin/padding spaces.

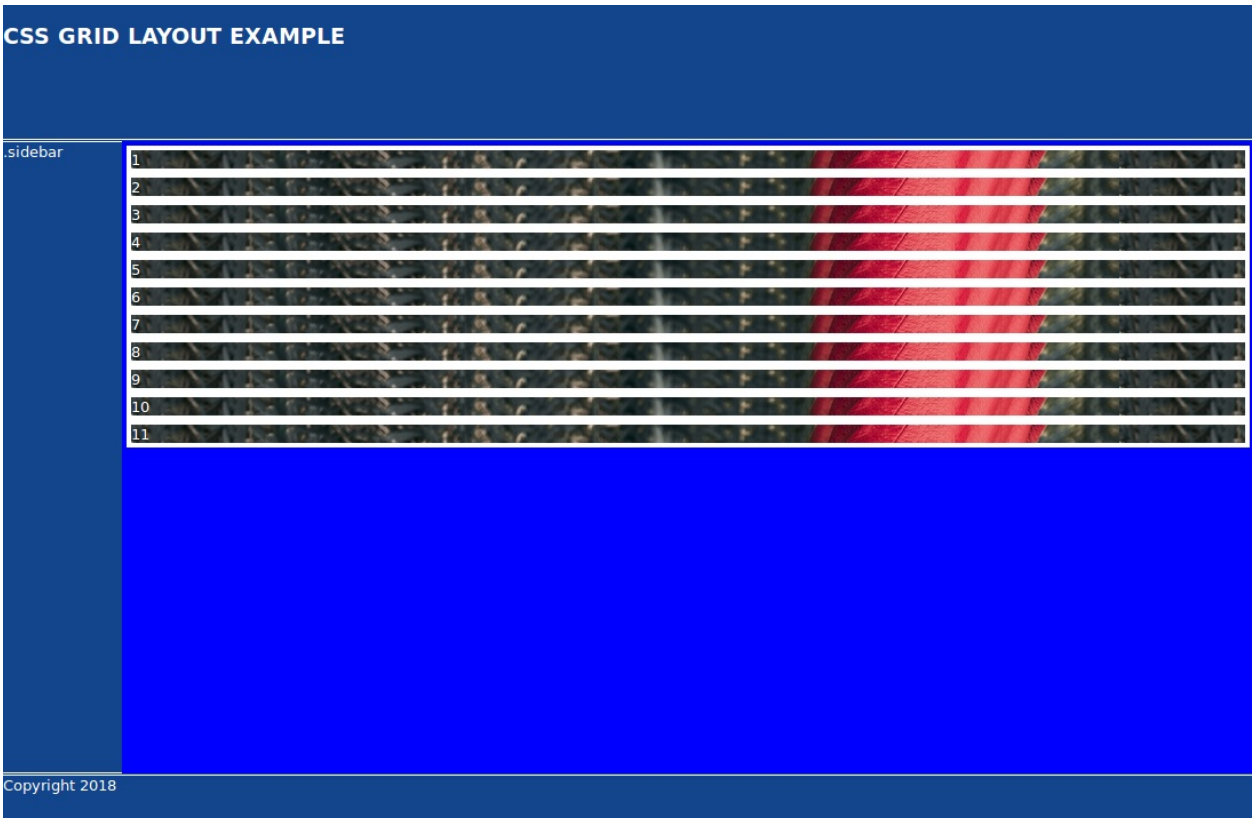
This is a screen shot of the result. Notice how the main area is not taking the full remaining width:



To solve this issue, we can add `width: auto;` when Grid is supported:

```
@supports (display: grid) {
  main {
    width:  auto;
  }
}
```

This is the screen shot of the result now:



Adding a Nested Grid

A Grid child can be a Grid container itself. Let's make the main section a Grid container:

```
main {
  /*...*/
  display: grid;
  grid-gap: 0.1vw;
  grid-template-columns: repeat(auto-fill, minmax(12rem, 1fr));
  grid-template-rows: repeat(auto-fill, minmax(12rem, 1fr));
}
```

We use a grid gap of *0.1vw* and we define columns and rows using the `repeat(auto-fill, minmax(12rem, 1fr));` function. The `auto-fill` option tries to fill the available space with as many columns or rows as possible creating implicit columns or rows if needed. If you want to fit the available columns or rows into the available space you need to use `auto-fit`. Read a good explanation of the differences between `auto-fill` and `auto-fit`.

This is the screen shot of the result:

Using the Grid `grid-column`, `grid-row` and `span` Keywords

CSS Grid provides `grid-column` and `grid-row` which allows you to position grid items inside their parent grid using grid lines. They are short-hand for the properties `grid-row-start` (specifies the start position of the grid item within the grid row), `grid-row-end` (specifies the end position of the grid item within the grid row), `grid-column-start` (specifies the start position of the grid item within the grid column) and `grid-column-end` (specifies the end position of the grid item within the grid column).

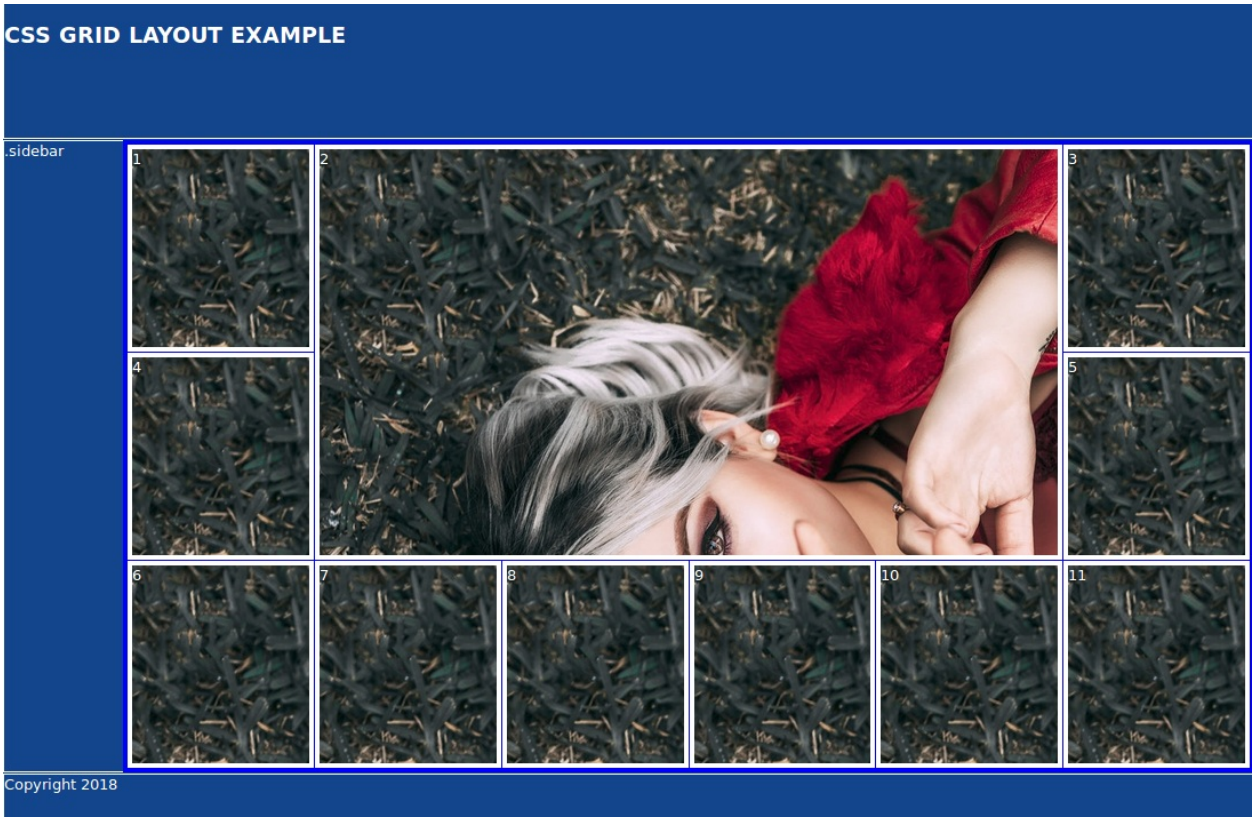
You can also use the keyword `span` to specify how much columns or rows to span.

Let's make the second child of the main area span 4 columns and 2 rows and position it from column line 2 and row line 1 (which is also its default location).

```
main article:nth-child(2) {
  grid-column: 2/span 4;
```

```
  grid-row: 1/span 2;
}
```

This is a screen shot of the result:



Using Grid Alignment Utilities

We want to center the text inside header, sidebar and footer and the numbers inside the `<article>` elements.

CSS Grid provides six properties `justify-items`, `align-items`, `justify-content`, `align-content`, `justify-self` and `align-self` which can be used to align and justify grid items. They are actually part of CSS box alignment module.

Inside header, aside, article and footer selectors add:

```
display: grid;
align-items: center;
justify-items: center;
```

- `justify-items` is used to justify the grid items along the row axis or horizontally.
- `align-items` aligns the grid items along the column axis or vertically. They can both take the *start*, *end*, *center* and *stretch* values.

This is a screen shot after centering elements:

Restructuring the Grid Layout in Small Devices

Our demo layout is convenient for medium and large screens but might not be the best way to structure the page in small screen devices. Using CSS Grid, we can easily change this layout structure, to make it linear in small devices, by redefining Grid areas and using Media Queries.

This is a screen shot before adding code to re-structure the layout on small devices:

Now, add the following CSS code:

```
@media all and (max-width: 575px) {
  body {
    grid-template-rows: 6rem  1fr 5.5rem  5.5rem;
    grid-template-columns: 1fr;
    grid-template-areas:
      "header"
      "content"
      "sidebar"
      "footer";
    }
}
```

On devices with `<= 575px` we use 4 rows with *6rem, 1fr, 5.5rem, and 5.5rem* widths respectively and one column that takes all the available space. We also redefine Grid areas so the sidebar can take the third row after the main content

area on small devices:

Notice the width of the sidebar, it's not taking the full available width. This is caused by the fallback code so all we need to do is overriding the `width: 6.3rem;` pair with `width: auto;` on browsers supporting Grid:

```
@supports (display: grid) {
  main, aside {
    width: auto;
  }
}
```

This is a screen shot of the final result:

Live Code

You can find the final code in this CodePen.

Conclusion

Throughout this tutorial, we've created a responsive demo layout with CSS Grid.

We've demonstrated using a fallback code for old browsers, adding CSS Grid progressively, restructuring the layout in small devices and centering elements using the alignment properties.

Chapter 4: Progressively Enhanced CSS Layouts from Floats to Flexbox to Grid

by Diogo Souza

It can be difficult to achieve complex, yet flexible and responsive grid layouts. Various techniques have evolved over the years but most, such as faux columns, were hacks rather than robust design options.

Most of these hacks were built on top of CSS *float* property. When flexbox layout module was introduced to the list of *display* property options, a new world of options became possible. Now you can not only define the direction the container is going to stack the items but also wrap, align (items and lines), order, shrink, etc. them in a container.

With all that power in hands, developers started to create their own combinations of rules for all sorts of layouts. Flexibility reigned like a charm. However, flexbox was designed to approach the one-dimensional layouts: either a row or a column. CSS Grid Layout, in turn, permitted two-dimensional row and column layouts.

Progressive Enhancement vs Graceful Degradation

It's difficult to create a website which supports every user's browser. Two options are commonly used:

Graceful degradation ensures a website continues to function even when something breaks. For example, `float: right` may fail if an element is too big for the screen but it wraps to the next empty space so the block remains usable.

Progressive enhancement takes the opposite approach. The page starts with

minimum functionality and features are added when they are supported. The example above could use a CSS media query to verify the screen is a minimum width before allowing an element to float.

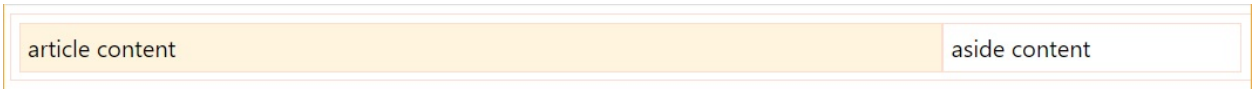
When it comes to grid layouts, each browser determines the appearance of their components. In this article, you're going to understand with some real samples how to evolve some web contents from an old strategy to a new one. More specifically, how to progressively enhance the model from a float-based layout to flexbox, and then CSS Grid, respectively.

Your Old Float Layout for A Page

Take a look at the following HTML page:

```
<main>
  <article>
    article content
  </article>

  <aside>
    aside content
  </aside>
</main>
```

It's a small, perhaps the most common, example of grid disposition you can have in a webpage: two divs sharing the same container (*body*).



The following CSS can be used in all the examples we'll create to set the body style:

```
body {
  font-family: Segoe UI;
  font-style: normal;
  font-weight: 400;
  font-size: 1rem;
}
```

Plus, the CSS snippet for each of our divs, enabling the floating effect:

```
main {
```

```
  width: 100%;
}

main, article, aside {
  border: 1px solid #fcddd1;
  padding: 5px;
  float: left;
}

article {
  background-color: #fff4dd;
  width: 74%;
}

aside {
  width: 24%;
}
```

Live Code

You can see the example in action here: Float Layout Example

While it is common, you can float as many elements as you want, one after another, in a way all of them suit the whole available width. However, this strategy has some downsides:

- It’s hard to manage the heights of the container elements;
- Vertical centering, if needed, could be painfully hard to manage;
- Depending on the content you have inside the elements (along with each inner container’s CSS properties), the browser could wrap elements to the next free line and break the layout.

One solution is the `display: table` layout:

```
main {
  display: table;
}

main, article, aside {
  display: table-cell;
}
```

However, this does not work in older browsers and more complex layouts become increasingly difficult.

Flexbox Approach

The float box module, known by the name of **flexbox**, is one of the available layout models that work distributing space and powerfully aligning the items of a container (the box) in a one-dimensional way. Its one dimension characteristic, though, does not impede you to design multidimensional layouts (rows and columns), but flexbox may not result in reliable row stacking.

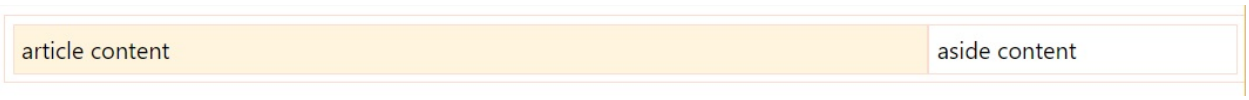
Besides the float approach being very popular and broadly adopted by popular grid frameworks, flexbox presents a series of benefits over float:

- Vertical alignment and height equality for the container's items on each wrapped row;
- The container (box) can increase/decrease based on the available space, and you determine whether it is a column or a row;
- Source independence. That is, the order of the items does not matter, they just need to be inside of the box.

To initiate a flexbox formatting strategy all you need to do is set the CSS *display* property with *flex* value:

```
main {
  width: 100%;
  display: flex;
}

main, article, aside {
  border: 1px solid #fcddd1;
  padding: 5px;
  float: left;
}

article {
  background-color: #fff4dd;
  width: 74%;
}

aside {
  width: 24%;
}
```

The following image shows the result:



Live Code

Here's a live example: See the Pen Flexbox Layout Example

Enhancing to CSS Grid Layout

The CSS **Grid** layout follows up closely the flexbox one, the big difference is that it works directly with the two-dimensional world; that is if you need to design a layout that deals with both rows and columns, the grid layout would probably suit better. It has the same aligning and space distribution factors of flexbox, but now acting directly to the two dimensions of your container (box). In comparison to the float property, it has even more advantages: easy elements disposition, alignment, row/column/cell control, etc.

Working with CSS Grid is as simple as changing the *display* property of your container element to `grid`. Inside the container, you can also create columns and rows with divs, for example. Let’s consider an example of an HTML page with four inner container divs.

Regarding the CSS definitions, let's start with the grid `container` div:

```
div.container {
  display: grid;
  grid-template-columns: 24% 75%;
  grid-template-rows: 200px 300px;
  grid-column-gap: 15px;
  grid-row-gap: 15px;
}
```

The property `grid-template-columns` defines the same configuration you had before: two grid columns occupying 24% and 75% of the whole container width, respectively. The `grid-template-rows`, in turn, do the same applying 200px and 300px as height for the first and second rows, respectively.

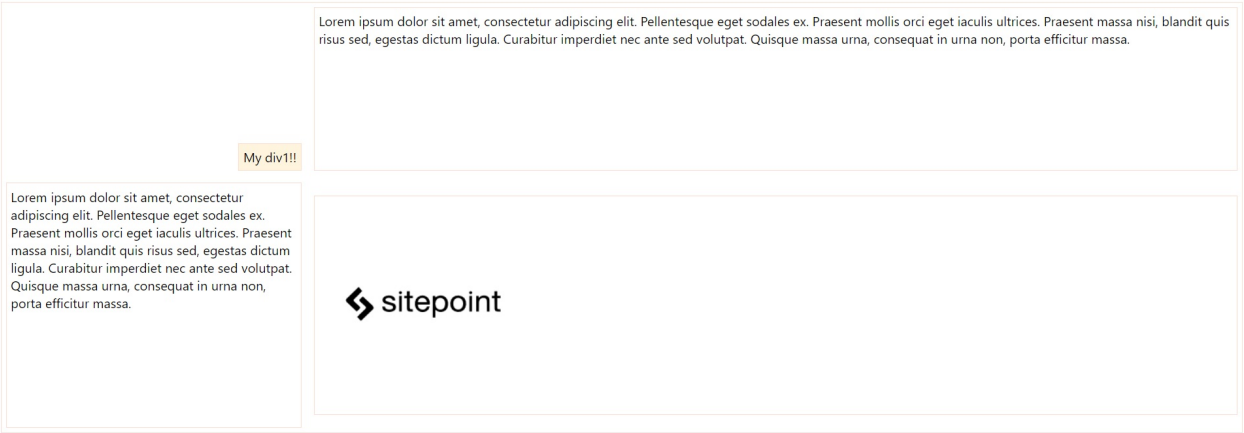
Use the properties `grid-column-gap` and `grid-row-gap` to allocate space around the grid elements.

In regards to applying grid alignment properties to specific cells, let's take a look:

```
.div1 {
  background-color: #fff4dd;
  align-self: end;
  justify-self: end;
}

.div4 {
  align-self: center;
}
```

For the div of class `div1`, you're aligning and justifying it at the *end* of the grid cell. The properties `align-self` and `justify-self` are valid for all flex items you have in the layout, including, in this case, the grid cells. `div4` was set only the centered alignment, for you to check the difference between both;

Here's how the elements are positioned now:



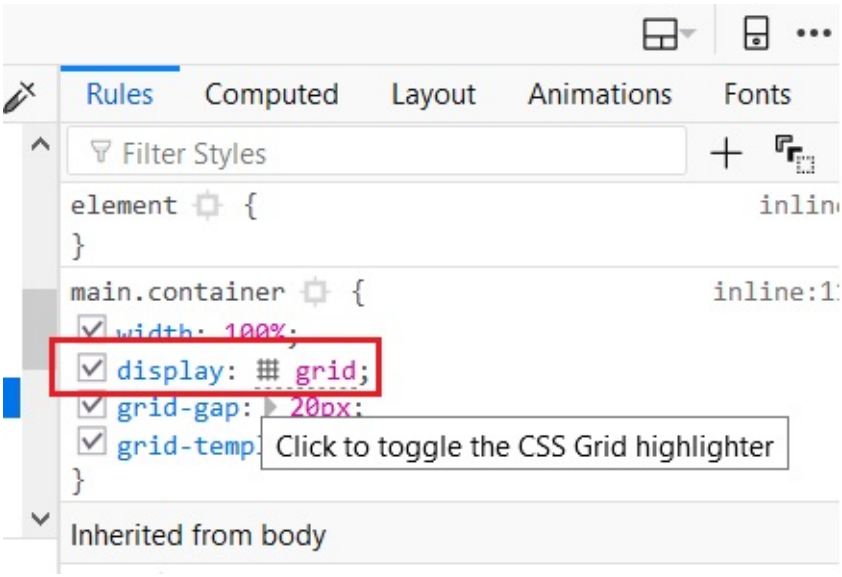
You can check out the final grid layout example here:

Live Code

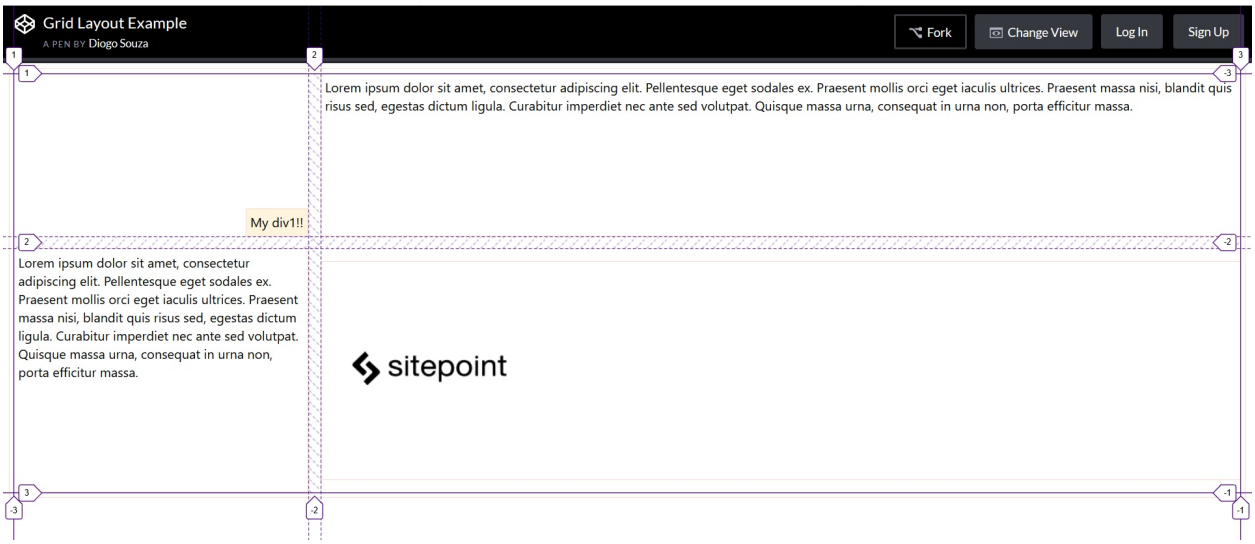
You can check out the final grid layout example here: See the Pen Grid Layout Example.

Most browsers also offer native support for grid layout inspections, which is great to see how it handles the grid mechanism internally. Let's try our grid example on Firefox with its Grid Inspector available via Firefox DevTools. In

order to open it, right-click the container element and, at the CSS pane's Rules view, find and click the grid icon right after the `display: grid`:



This will toggle the Grid highlighter. You can also control more display settings at the CSS pane's Layout view like the line numbers or the area names exhibition:

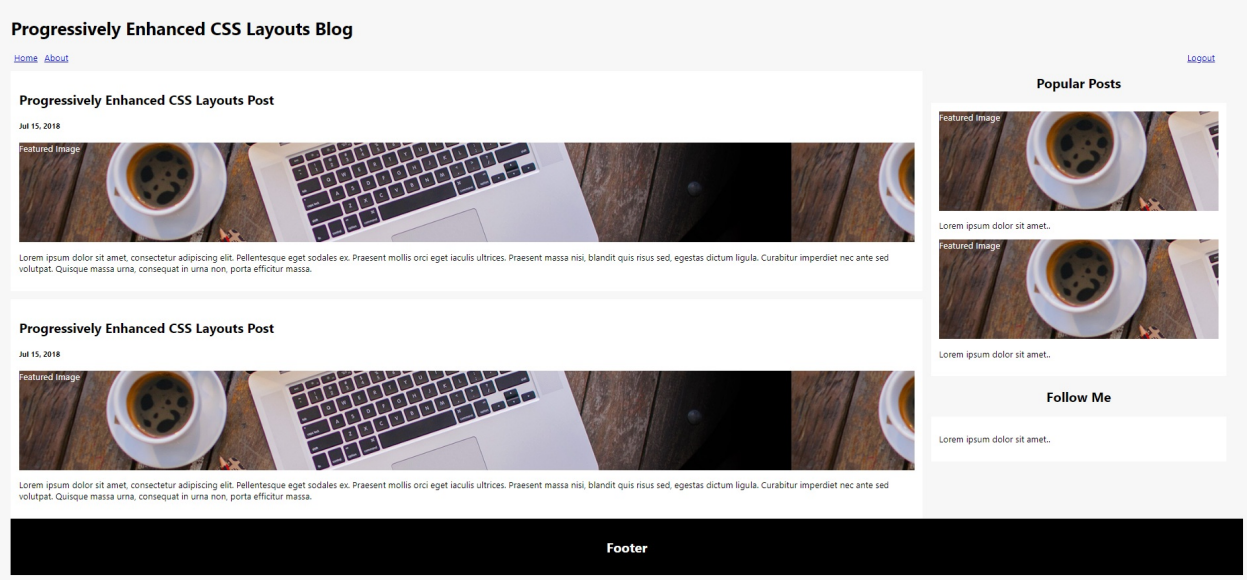


Progressively Enhancing a Blog Layout

In this next example, we’re going to use a blog page as a reference to enhance from a totally float-based page to a CSS Grid layout, exploring the way the old layout can be completely transformed to a layout that embraces both flexbox and

grid worlds.

In the following updates, we'll keep an eye to the old browsers that don't support Flexbox or CSS Grid, as well as how the blog behaves on mobile versions. This is the look and feel of our blog page totally based on divs and CSS float property:



Live Code

You can check out this example: See the Pen Blog Layout with Float.

Note that the HTML structure is common for who's already familiar with semantic tags: a div (the container) containing all the inner elements that'll compose the final layout based on who's floating who.

It's basically made of a simple `header` with a menu that defines how its items (links) will be displayed and aligned:

```
nav.menu {
  width: 98%;
}

ul {
  list-style: none;
  padding: 0px;
  margin: 0px;
}
```

```
.left {
  float: left;
}

.right {
  float: right;
}

a {
  padding: 4px 6px;
}
```

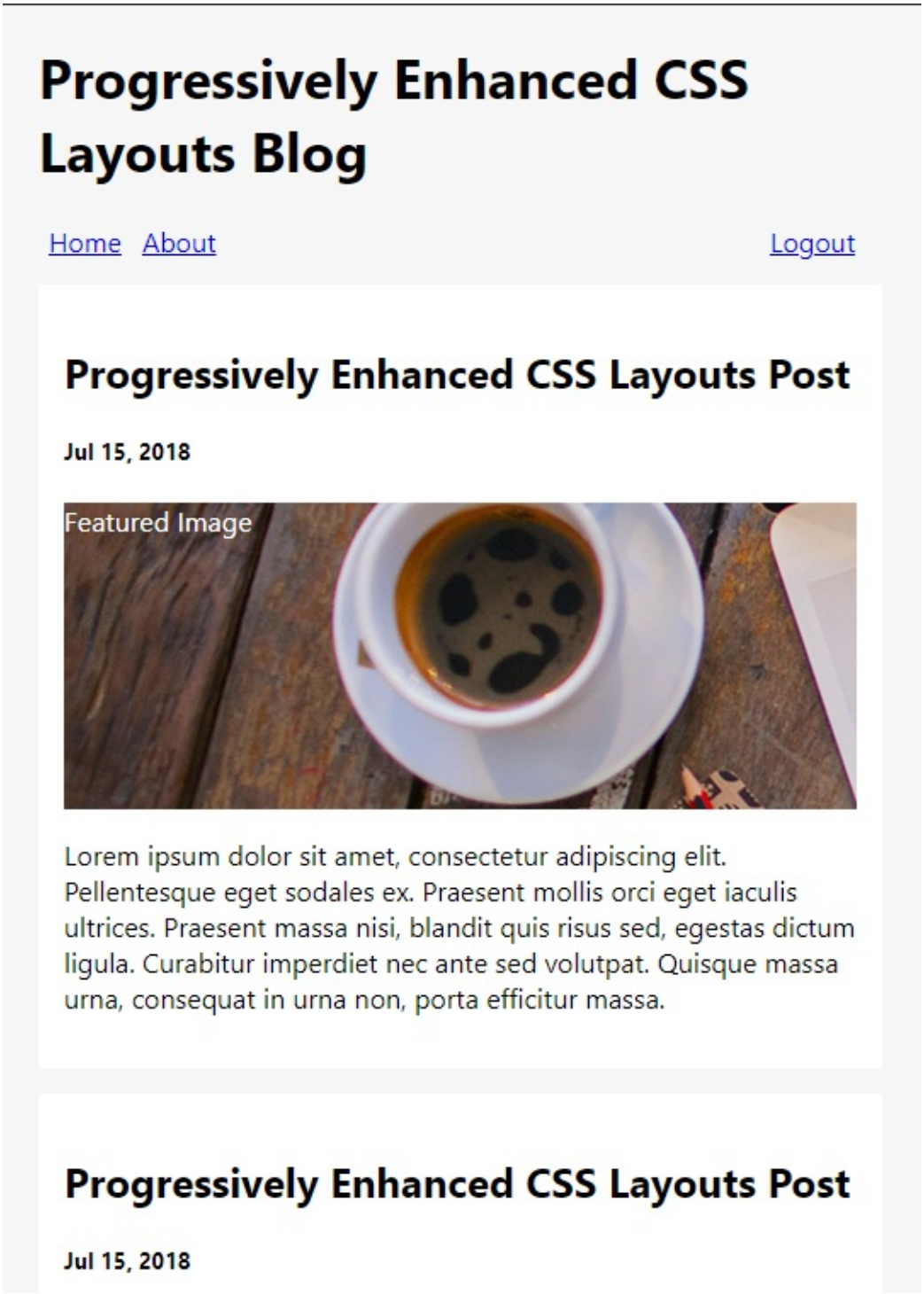
The content of the page is divided into two ones, the `section` and `aside` elements:

```
main {
  float: left;
  width: 74%;
}

aside {
  float: left;
  width: 24%;
  margin-left: 15px;
  margin-bottom: 15px;
}

aside h2 {
  text-align: center;
}
```

Nothing exceptional, just left floating the divs and determining the maximum width of each one on top of the full width. Note also the need for clearing (`clear`) things whenever we need to disable the floating effect after a div:

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

The end of the CSS brings a `@media` rule that'll break the sidebar to the next column when the page is opened in small screen devices. The end of the HTML brings a simple footer with just a text.



Note: the beginning of the HTML brings also the import of the [HTML5 Shiv script] in order to enable the use of HTML5 sectioning elements in legacy Internet Explorer.

Menu Enhancements with Flexbox

As seen before, flexbox works better with one-dimensional layouts like a horizontal menu, for example. The same menu structure could be enhanced to this one:

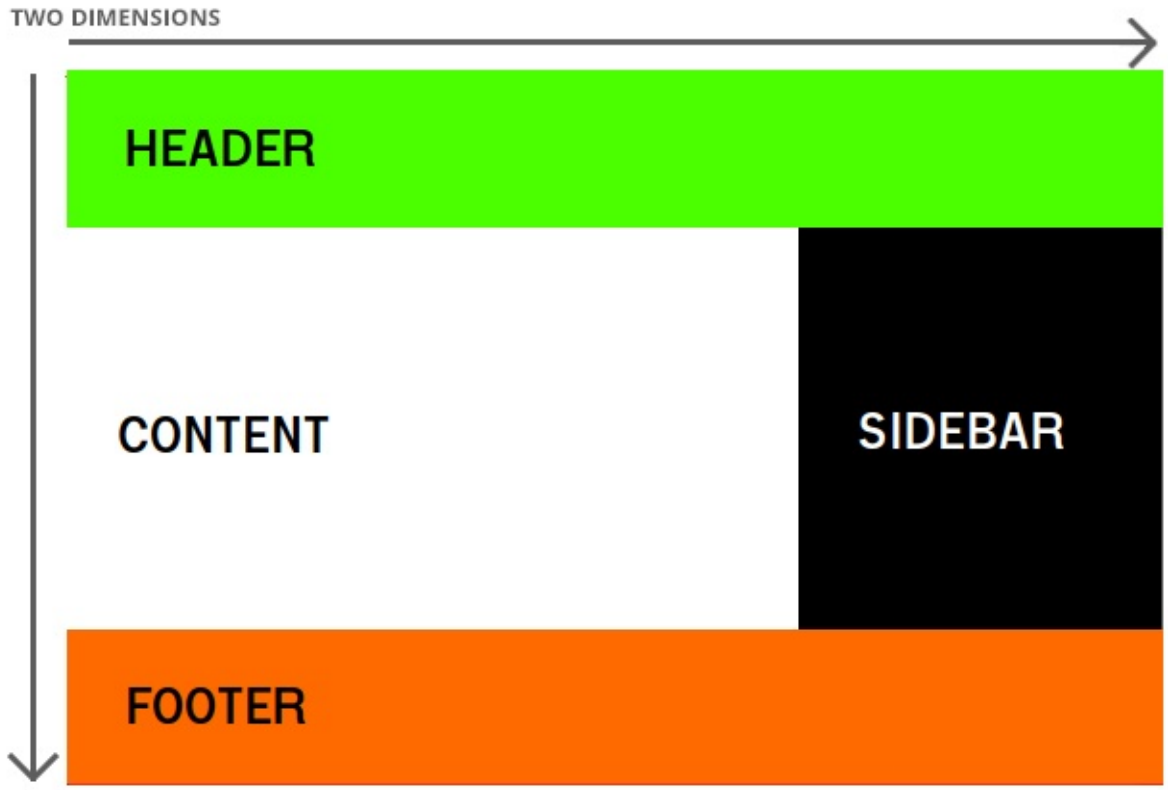
```
nav.menu {
  width: 98%;
}

nav.menu ul {
  display: flex;
}

nav.menu ul > li:last-child {
  margin-left: auto;
}
```

The use of flexbox property automatically disables the float for newer browsers. However, since we preserved the old CSS configurations for the `float` property, the float will still apply when a browser does not support flexbox.

At the same time, the `display: flex` was added to layout the menu using flexbox along with the `margin-left: auto` to the last menu item. This will ensure that this item will be pushed to the right of the layout, separating them into distinct groups.

Enhancing to Grid Areas

CSS Grid gives us more flexibility when it comes to the disposition of inner elements. Once we defined that flexbox was a perfect choice for a horizontal container (the menu), a grid would be perfect for such a two-dimensional blog layout, that basically divides into three parts:



The Grid Area feature consists of the combination of two properties: `grid-template-areas` and `grid-area`. The first one defines how the whole layout height and width would be divided into groups by explicitly putting the names of each group as they'd be in the final layout:

```
div.container {
  width: 100%;
  display: grid;
  grid-gap: 20px;
  grid-template-areas:
    "header header header"
    "content content sidebar"
    "footer footer footer"
}
```

Here, we're dividing the whole container space into 3 rows and 3 columns. The names of the areas repeated alongside the definition say how much space each of them will occupy vertically and horizontally.

Then, we can use the second property by setting for each of the grid areas the respective `grid-area` value:

```
header {
  grid-area: header;
}

main {
  grid-area: content;
  /* other properties */
}

aside {
  grid-area: sidebar;
  /* other properties */
}

footer {
  grid-area: footer;
  /* other properties */
}
```

Just like with flexbox, grid items automatically disable float declarations in browsers which support CSS Grid. The rest of the properties remain the same (the float will still apply when a browser does not support grid.).

Live Code

This example has a Live Codepen.io Demo you can play with: See the Pen Blog Layout with CSS Grid Areas.

Enhancing Grid Templates

The property `grid-template` is also very useful when we need to define a template that'll follow a pattern for our grid's definitions.

Let's take a look at the following `.container` CSS:

```
div.container {
  width: 100%;
  display: grid;
  grid-gap: 20px;
  grid-template-columns: auto auto auto;
}
```

Here, we're basically saying that the size of the columns is determined by the size of the container and on the size of the content of the items in the column.

Then, each of our grid areas will have to adapt:

```
header {
  grid-column: 1/4;
}

section {
  grid-column: 1/3;
  /* other properties */
}

aside {
  grid-column: 3/4;
  /* other properties */
}

footer {
  grid-column: 1/4;
  /* other properties */
}
```

Here, `grid-column: 1/4` instructs the browser to start a column at track 1 and end at track 4. Tracks are the separators between each grid cell so this element would occupy cells one to three:

Live Code

This example has a live Codepen.io Demo you can play with: See the Pen Blog Layout with CSS Grid Templates.

You can go now and test it in different browser versions as well as your mobile phone to attest to how the pages were progressively enhanced.

More About CSS Grid Layout

For more information, refer to the SitePoint's CSS Grid Layout Introduction and CSS Flexbox Introduction guides.

Mozilla has also provided a great article about CSS Grid Layout and Progressive Enhancement that will certainly add a lot. Good studies!

Chapter 5: Make Forms Great with CSS Grid

by Craig Buckler

Form design is a fundamental yet frustrating part of web design and development. Ask anyone who's ever tried to style a `<select>` box or align a label consistently in all browsers.

In 2016, I wrote Make Forms Fun with Flexbox which identified how several form difficulties could be solved with Flexbox. A key benefit was HTML source order consistency with the `<label>` always following its associated field tag in a container:

```
<div>
  <input id="name" name="name" type="text" />
  <label for="name">name</label>
</div>

<div>
  <select id="experience" name="experience"><!-- options --></select
  <label for="experience">experience</label>
</div>

<div>
  <input id="html" name="html" type="checkbox" />
  <label for="html">HTML</label>
</div>
```

Flexbox could then be used to:

- reposition the label if necessary, i.e. move it to the left of the field on text inputs, select boxes, and textareas.
- vertically align the label and field.

It also became possible to style labels based on the state of their field using adjacent sibling selectors, e.g. apply bold to a label when its associated checkbox is checked:

```
input:checked + label {
  font-weight: bold;
}
```

Flawed Flexboxed Forms

Unfortunately, there are a number of problems using Flexbox to layout a form. Flexbox creates a one-dimensional layout where each item follows another and wraps to a new line when necessary. Field/label pairs must be placed in a container elements with `display: flex;` applied to guarantee each appears on a new row.

It was also necessary to define a fixed label width, such as `10em`. If a long label required more room, its text would either overflow or resize the element and push the field out of alignment with others.

Finally, forms are normally laid out in a grid. Shouldn't we be using CSS Grid now it's fully supported in all mainstream browsers? *Absolutely!*

Development Approach

Most CSS Grid articles demonstrate the concepts and may provide graceful degradation fallbacks for older browsers. That approach is ideal when the layout is mostly decorative, e.g. positioning page content, headers, footers and menus. It rarely matters when *oldBrowserX* shows linear blocks in an unusual order because the page content remains usable.

Form layout is more critical: a misaligned label could lead the user to enter information in the wrong box. For this reason, this tutorial takes a progressive enhancement approach:

1. An initial floated layout will work in all browsers including IE8+ (which did not support Flexbox either). It will not be perfect, but floats never were!
2. Enhance the layout using CSS Grid in all modern browsers.

The examples below contain very few CSS classes and styling is applied directly to HTML elements. That is not the BEM way, but it is intentional to keep the code clean and understandable without distractions.

You could consider using similar code as the base for all forms on your site.

The HTML

A typical HTML form can be kept clean since there is no need for containing (`<div>`) elements around field/label pairs:

```
<form action="get">
  <fieldset>
    <legend>Your web development skillset</legend>

    <div class="formgrid">

      <input id="name" name="name" type="text" />
      <label for="name">name</label>

      <select id="experience" name="experience">
        <option value="1">1 year or less</option>
        <option value="2">2 years</option>
        <option value="3">3 - 4 years</option>
        <option value="5">5 years or more</option>
      </select>
      <label for="experience">experience</label>

      <input id="html" name="html" type="checkbox" />
      <label for="html">HTML</label>

      <input id="css" name="css" type="checkbox" />
      <label for="css">CSS</label>

      <input id="javascript" name="javascript" type="checkbox" />
      <label for="javascript">JavaScript</label>

      <textarea id="skills" name="skills" rows="5" cols="20"></texta
      <label for="skills">other skills</label>

      <button type="submit">SUBMIT</button>

    </div>

  </fieldset>
</form>
```

The only additional element is `<div class="formgrid">`. Browsers cannot apply `display: grid` or `display: flex` to `fieldset` elements. That may eventually be fixed but an outer container is currently required.

Form Float Fallback

After some initial font and color styling, the float layout will allocate:

- 70% of the space to fields which are floated right, and
- 30% of the space to labels which are floated left

```
/* fallback 30%/70% float layout */
input, output, textarea, select, button {
  clear: both;
  float: right;
  width: 70%;
}

label {
  float: left;
  width: 30%;
  text-align: right;
  padding: 0.25em 1em 0 0;
}
```

Checkbox and radio buttons are positioned before the label and floated left. Their intrinsic width can be used (`width:auto`) but a left margin of 30% is required to align correctly:

```
button, input[type="checkbox"], input[type="radio"] {
  width: auto;
  float: left;
  margin: 0.5em 0.5em 0 30%;
}

input[type="checkbox"] + label, input[type="radio"] + label {
  width: auto;
  text-align: left;
}
```

Live Code

The layout works in all browsers including IE8+: See the Pen form grid 1: float layout.

A less conscientious developer would go home for the day but this layout has several problems:

- the padding and margin tweaks are fragile and can look inconsistent across browsers
- if longer labels or different-sized fonts are ever required, the CSS spacing will require adjustment, and
- the design breaks at smaller screen sizes and labels can overflow fields.

Get Going with Grid

The Grid module adds 18 new CSS properties in order to create a layout with rows and columns. Elements within the grid can placed in any row/column, span multiple rows and/or columns, overlap other elements, and be aligned horizontally and/or vertically. There are similarities to Flexbox, but:

- Flexbox is one-dimensional. Elements come one after the other and may or may not wrap to a new "row". Menus and photo galleries are a typical use-case.
- Grid is two-dimensional and respects both rows and columns. If an element is too big for its cell, the row and/or column will grow accordingly. Grid is ideal for page and form layouts.

It is possibly better to compare CSS Grid with table-based layouts but they're considerably more flexible and require less markup. It has a steeper learning curve than other CSS concepts but you're unlikely to require all the properties and the minimum is demonstrated here. The most basic grid is defined on a containing element:

```
.container {
  display: grid;
}
```

More practically, layouts also require the number of columns, their sizes, and the gap between rows and columns, e.g.

```
.container {
  display: grid;
  grid-template-columns: 10% 1fr 2fr 12em;
  grid-gap: 0.3em 0.6em;
}
```

This defines four columns. Any measurement unit can be used as well as the `fr` fractional unit. This calculates the remaining space in a grid and distributes

accordingly. The example above defines total of `3fr` on columns two and three. If 600 pixels of horizontal space was available:

- `1fr` equates to `(1fr / 3fr) * 600px` = `200px`
- `2fr` equates to `(2fr / 3fr) * 600px` = `400px`

A gap of `0.3em` is defined between rows and `0.6em` between columns.

All child elements of the `.container` are now grid items. By default, the first child element will appear at row 1, column 1. The second in row 1, column 2, and the sixth in row 2, column 2. It's possible to size rows using a property such as `grid-template-rows` but heights will be inferred by the content.

Grid support is excellent. It's not available in Opera Mini but even IE11 offers an older implementation of the specification. In most cases, fallbacks are simple:

- Older browsers can use flexbox, floats, inline-blocks, or `display:table` layouts. All Grid properties are ignored.
- When a browser supports grid, all flexbox, floats, inline-blocks and table layout properties assigned to a grid item are disabled.

Grid tools and resources:

- MDN Grid Layout
- A Complete Guide to Grid
- Grid by Example
- Grid “fallbacks” and overrides
- CSS Grid Playground
- CSS Grid Garden
- Layoutit!

Firefox and Chrome-based browsers have excellent DevTool Grid layout and visualisation tools.

Form Grid

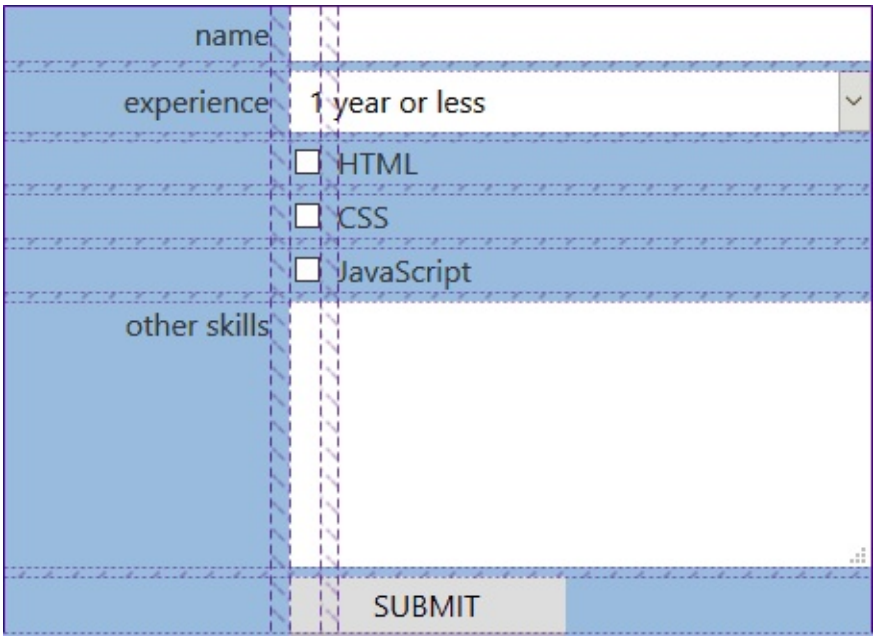
To progressively enhance the existing form, Grid code will be placed inside an `@supports` declaration:

```
/* grid layout */
```

```
@supports (display: grid) {
  ...
}
```

This is rarely necessary in most grid layouts. However, this example resets all float paddings and margins; that must only occur when a CSS Grid is being applied.

The layout itself will use a three column design:



where:

- standard labels appear in column one
- checkbox and radio buttons span columns one and two (but are aligned right)
- checkbox and radio labels appear in column three
- all other fields span columns two and three.

The outer container and child field properties:

```
.formgrid {
  display: grid;
  grid-template-columns: 1fr 1em 2fr;
  grid-gap: 0.3em 0.6em;
  grid-auto-flow: dense;
  align-items: center;
```

```
}

input, output, textarea, select, button {
  grid-column: 2 / 4;
  width: auto;
  margin: 0;
}
```

`grid-column` defines the starting and ending grid tracks. Tracks are the *edges* between cells so the three-column layout has four tracks:

1. the first track on the left-hand side of the grid before column one
2. the track between columns one and two
3. the track between columns two and three
4. the final track on the right-hand edge of the grid after column three

`grid-column: 2 / 4;` positions all fields between tracks 2 and 4 - *or inside columns two and three*.

The first HTML element is the name `<input>`. It spans columns two and three which means column one (track 1 / 2) is empty on that row. By default, the `name` label would therefore drop to row 2, column 1. However, by setting `grid-auto-flow: dense;` in the container, the browser will attempt to fill empty cells earlier in the grid before progressing to a new row.

Checkboxes and radio buttons can now be set to span tracks 1 to 3 (columns one and two) but align themselves to the right-hand edge using `justify-self: end`:

```
input[type="checkbox"], input[type="radio"] {
  grid-column: 1 / 3;
  justify-self: end;
  margin: 0;
}
```

Labels on the grid will handle themselves and fit into whichever row cell is empty. However, the default widths and spacing from the float layout are now unnecessary:

```
label, input[type="checkbox"] + label, input[type="radio"] + label {
  width: auto;
  padding: 0;
  margin: 0;
}
```

Finally, `<textarea>` labels can be vertically positioned at the top of the cell rather than centered:

```
textarea + label {
  align-self: start;
}
```

Live Code

The final grid layout: See the Pen form grid 2: grid applied

Unlike floats, the design will not break at small dimensions or require tweaking when different fonts, sizes or labels are added.

Grid Enlightenment

It's taken several years to become viable, but CSS Grid is well supported and offers layout possibilities which would have been difficult with floats or flexbox. Forms are an ideal use-case and the resulting CSS is short yet robust.

If you're looking to learn another CSS technique, Grid should be at the top of your list.

Book 3: CSS: Tools & Skills

sitepoint

CSS:
TOOLS & SKILLS

Chapter 1: How to Use Gulp.js to Automate Your CSS Tasks

by Craig Buckler

In this article, we look at how you can use Gulp.js to automate a range of repetitive CSS development tasks to speed up your workflow.

Web development requires little more than a text editor. However, you' ll quickly become frustrated with the repetitive tasks that are essential for a modern website and fast performance, such as:

- converting or transpiling
- minimizing file sizes
- concatenating files
- minifying production code
- deploying updates to development, staging and live production servers.

Some tasks must be repeated every time you make a change. The most infallible developer will forget to optimize an image or two and pre-production tasks become increasingly arduous.

Fortunately, computers never complain about mind-numbing work. This article demonstrates how use Gulp.js to automate CSS tasks, including:

- optimizing images
- compiling Sass .scss files
- handling and inlining assets
- automatically appending vendor prefixes
- removing unused CSS selectors
- minifying CSS
- reporting file sizes
- outputting sourcemaps for use in browser DevTools
- live-reloading CSS in a browser when source files change.

All the code is available from GitHub, and it works on Windows, macOS or

Linux.

Why Use Gulp?

A variety of task runners are available for web projects including Gulp, Grunt, webpack and even npm scripts. Ultimately, the choice is yours and it doesn't matter what you use, as *your site/app visitors will never know or care*.

Gulp is a few years old but stable, fast, supports many plugins, and is configured using JavaScript code. Writing tasks in code has several advantages, and you can modify output according to conditions — such as only minifying CSS when building the final files for live deployment.

Example Project Overview

The task code assumes Gulp 3.x will be used. This is the most recent stable version and, while Gulp 4 is available, it's not the default on npm. If you're using Gulp 4, refer to How do I update to Gulp 4? and tweak the `gulpfile.js` code accordingly.

Image file sizes will be minimized with gulp-imagemin which optimizes JPG, GIF and PNG bitmaps as well as SVG vector graphics.

Sass `.scss` files will be pre-processed into a browser-compatible `main.css` file. Sass isn't considered as essential as it once was, because:

- standard CSS now offers features such as variables (Custom Properties)
- HTTP/2 reduces the need for file concatenation.

That said, Sass remains a practical option for file splitting, organization, (static) variables, mixins and nesting (presuming you don't go too deep).

The resulting Sass-compiled CSS file will then be processed using PostCSS to provide further enhancements such as asset management and vendor-prefixing.

Getting Started with Gulp

If you' ve never used Gulp before, please read An Introduction to Gulp.js. These

are the basic steps for getting started from your terminal:

1. Ensure Node.js is installed.
2. Install the Gulp command-line interface globally with `npm i gulp-cli -g`.
3. Create a new project folder — for example, `mkdir gulpcss` — and enter it (`cd gulpcss`).
4. Run `npm init` and answer each question (the defaults are fine). This will create a `package.json` project configuration file.
5. Create a `src` sub-folder for source files: `mkdir src`.

The example project uses the following sub-folders:

- `src/images` — image files
- `src/scss` — source Sass files
- `build` — the folder where compiled files are generated

Test HTML Page

This tutorial concentrates on CSS-related tasks, but an `index.html` file in the root folder is useful for testing. Add your own page code with a `<link>` to the final stylesheet. For example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,initial-scale=1.
  <title>Using Gulp.js for CSS tasks</title>
  <link rel="stylesheet" media="all" href="build/css/main.css">
</head>
<body>

  <h1>My example page</h1>

</body>
</html>
```

Module Installation

Most Node.js modules will be installed as project dependencies so the CSS can be built on development or production servers. To install Gulp and all plugins

locally, do the following:

```
npm i gulp gulp-imagemin gulp-newer gulp-noop gulp-postcss gulp-sass
➥gulp-sourcemaps postcss-assets autoprefixer cssnano usedcss
```

Assuming you're using a recent version of `npm`, all modules will be listed in the `"dependencies"` section of `package.json`.

browser-sync is installed as a development dependency since, it's not required on live production servers:

```
npm i browser-sync --save-dev
```

The module will be listed in the `"devDependencies"` section of `package.json`.

Create a Gulp Task File

Gulp tasks are defined in a JavaScript file named `gulpfile.js`. Create it, then open the file in your editor (VS Code is the current favorite). Add the following code:

```
(() => {

  'use strict';

  /**************** gulpfile.js configuration ****************/

  const

    // development or production
    devBuild  = ((process.env.NODE_ENV || 'development').trim().toLo
    ➥'development'),

    // directory locations
    dir = {
      src         : 'src/',
      build       : 'build/'
    },

    // modules
    gulp          = require('gulp'),
    noop          = require('gulp-noop'),
    newer         = require('gulp-newer'),
    size          = require('gulp-size'),
    imagemin      = require('gulp-imagemin'),
```

```
    sass          = require('gulp-sass'),
    postcss       = require('gulp-postcss'),
    sourcemaps    = devBuild ? require('gulp-sourcemaps') : null,
    browsersync   = devBuild ? require('browser-sync').create() : nu

  console.log('Gulp', devBuild ? 'development' : 'production', 'buil

})();
```

This defines a self-executing function and constants for:

- `devBuild` — `true` if `NODE_ENV` is blank or set to `development`
- `dir.src` — the source file folder
- `dir.build` — the build folder
- Gulp and all plugin modules

Note that `sourcemaps` and `browsersync` are only enabled for development builds.

Gulp Image Task

Create the `src/images` folder then copy some image files into it or any subfolder.

Insert the following code below the `console.log` in `gulpfile.js` to create an `images` processing task:

```
  /***************** images task ****************/

  const imgConfig = {
    src           : dir.src + 'images/**/*',
    build         : dir.build + 'images/',

    minOpts: {
      optimizationLevel: 5
    }
  };

  gulp.task('images', () =>

    gulp.src(imgConfig.src)
      .pipe(newer(imgConfig.build))
      .pipe(imagemin(imgConfig.minOpts))
      .pipe(size({ showFiles:true }))
```

```
    .pipe(gulp.dest(imgConfig.build))

);
```

Configuration parameters are defined in `imgConfig`, which sets:

- a `src` folder to any image inside `src/images` or a subfolder
- a `build` folder to `build/images`, and
- gulp-imagemin optimization options.

A `gulp.task` named `images` passes data through a series of pipes:

1. The source folder is examined.
2. The gulp-newer plugin removes any newer image already present in the build folder.
3. The gulp-imagemin plugin optimizes the remaining files.
4. The gulp-size plugin shows the resulting size of all processed files.
5. The files are saved to the `gulp.dest` build folder.

Save `gulpfile.js`, then run the `images` task from the command line:

```
gulp images
```

The terminal will show a log something like this:

```
Gulp development build
[16:55:12] Using gulpfile gulpfile.js
[16:55:12] Starting 'images'...
[16:55:12] icons/alert.svg 306 B
[16:55:12] cave-0600.jpg 47.8 kB
[16:55:12] icons/fast.svg 240 B
[16:55:12] cave-1200.jpg 112 kB
[16:55:12] cave-1800.jpg 157 kB
[16:55:12] icons/reload.svg 303 B
[16:55:12] gulp-imagemin: Minified 3 images (saved 205 B - 19.4%)
[16:55:12] all files 318 kB
[16:55:12] Finished 'images' after 640 ms
```

Examine the created `build/images` folder to find optimized versions of your images. If you run `gulp images` again, nothing will occur because only newer files will be processed.

Gulp CSS Task

Create a `src/scss` folder with a file named `main.scss`. This is the root Sass file which imports other partials. You can organize these files as you like but, to get started, add:

```
// main.scss
@import 'base/_base';
```

Create a `src/scss/base` folder and add a `_base.scss` file with the following code:

```
// base/_base.scss partial
$font-main: sans-serif;
$font-size: 100%;

body {
  font-family: $font-main;
  font-size: $font-size;
  color: #444;
  background-color: #fff;
}
```

Insert the following code below the `images` task in `gulpfile.js` to create a `css` processing task:

```
  /***************** CSS task ****************/

  const cssConfig = {

    src         : dir.src + 'scss/main.scss',
    watch       : dir.src + 'scss/**/*',
    build       : dir.build + 'css/',
    sassOpts: {
      sourceMap       : devBuild,
      outputStyle     : 'nested',
      imagePath       :../Images/',
      precision       : 3,
      errLogToConsole : true
    },

    postCSS: [
      require('postcss-assets')({
        loadPaths: ['images/'],
        basePath: dir.build
      }),
      require('autoprefixer')({
        browsers: ['> 1%']
      })
```

```
    ]

  };

  // remove unused selectors and minify production CSS
  if (!devBuild) {

    cssConfig.postCSS.push(
      require('usedcss')({ html: ['index.html'] }),
      require('cssnano')
    );

  }

  gulp.task('css', ['images'], () =>

    gulp.src(cssConfig.src)
      .pipe(sourcemaps ? sourcemaps.init() : noop())
      .pipe(sass(cssConfig.sassOpts).on('error', sass.logError))
      .pipe(postcss(cssConfig.postCSS))
      .pipe(sourcemaps ? sourcemaps.write() : noop())
      .pipe(size({ showFiles:true }))
      .pipe(gulp.dest(cssConfig.build))
      .pipe(browsersync ? browsersync.reload({ stream: true }) : noo

  );
```

Configuration parameters are defined in `cssConfig`, which sets:

- the `src` file `src/scss/main.scss`
- a `watch` folder to any file within `src/scss` or its subfolders
- the `build` folder to `build/css`
- gulp-sass options in `sassOpts`: these are passed to node-sass, which ultimately calls LibSass.

`cssConfig.postCSS` defines an array of PostCSS plugins and configuration options. The first is postcss-assets, which can resolve image URL paths and pass information to CSS files. For example, if `myimage.png` is a 400 x 300 bitmap:

```
.myimage {
  width: width('myimage.png'); /* 400px */
  height: height('myimage.png'); /* 300px */
  background-size: size('myimage.png'); /* 400px 300px */
}
```

It’s also possible to inline bitmap or SVG images. For example:

```
.mysvg {
  background-image: inline('mysvg.svg');
  /* url('data:image/svg+xml;charset=utf-8,... */
}
```

autoprefixer is the famous PostCSS plugin which adds vendor prefixes according to information from caniuse.com. In the configuration above, any browser with a global market share of 1% or more will have vendor prefixes added. For example:

```
filter: opacity(0.5);
```

becomes:

```
-webkit-filter: opacity(0.5);
filter: opacity(0.5);
```

Two further PostCSS plugins are added when `NODE_ENV` is set to `production`:

1. usedcss, which removes unused selectors by examining the example `index.html` file.
2. cssnano, which minifies the resulting CSS file by removing all comments, whitespace, etc.

A `gulp.task` named `css` is then defined. It runs the `images` task first, since the CSS may depend on images. It then passes data through a series of pipes:

1. The source folder is examined.
2. If `devBuild` is true, the gulp-sourcemaps plugin is initialized. Otherwise, the gulp-noop does nothing.
3. The gulp-sass plugin transpiles `main.scss` to CSS using the `cssConfig.sassOpts` configuration options. Note the `on('error')` event handler prevents Gulp terminating when a Sass syntax error is encountered.
4. The resulting CSS is piped into gulp-postcss, which applies the plugins described above.
5. If sourcemaps are enabled, they're now appended as a data URI to the end of the CSS.
6. The gulp-size plugin displays the final size of the CSS file.
7. The files are saved to the `gulp.dest` build folder.
8. Finally, if `browsersync` is set, a instruction is sent to browser-sync to refresh the CSS in all connected browsers (see below).

Save `gulpfile.js`, then run the task from the command line:

```
gulp css
```

The terminal will show a log similar to this:

```
Gulp development build
[09:22:18] Using gulpfile gulpfile.js
[09:22:18] Starting 'images'...
[09:22:18] gulp-imagemin: Minified 0 images
[09:22:18] Finished 'images' after 80 ms
[09:22:18] Starting 'css'...
[09:22:18] main.css 1.07 kB
[09:22:18] Finished 'css' after 91 ms
```

Examine the created `build/css` folder to find a development version of the resulting `main.css` file containing a sourcemap data:

```
body {
  font-family: sans-serif;
  font-size: 100%;
  color: #444;
  background-color: #fff; }

/*# sourceMappingURL=data:application/json;charset=utf8;base64,...
```

Automating Your Workflow

Running one task at a time and manually refreshing your browser is no fun. Fortunately, Browsersync provides a seemingly magical solution:

- It implements a development web server or proxy to an existing server.
- Code changes are dynamically applied and CSS can refresh without a full page reload.
- Connected browsers can mirror scrolling and form input. For example, you complete a form on your desktop PC and see it happening on a mobile device.
- It's fully compatible with Gulp and other build tools.

Insert the following code below the `css` task in `gulpfile.js` to define:

1. a `browsersync` processing task

2. a `default` Gulp task for monitoring file changes:

```
  /***************** browser-sync task ****************/

  const syncConfig = {
    server: {
      baseDir   : './',
      index     : 'index.html'
    },
    port        : 8000,
    files       : dir.build + '**/*',
    open        : false
  };

  // browser-sync
  gulp.task('browsersync', () =>
    browsersync ? browsersync.init(syncConfig) : null
  );

  /***************** watch task ****************/

  gulp.task('default', ['css', 'browsersync'], () => {// image chang
gulp.watch(imgConfig.src, ['images']);

// CSS changes
gulp.watch(cssConfig.watch, ['css']);
  });
```

The browser-sync configuration parameters are defined in `syncConfig`, which set options such as the port and default file. The `browsersync` task then initiates accordingly.

Browsersync is able to watch for file changes itself but, in this case, we want to control it via Gulp to ensure that refreshes only occur when CSS changes occur.

The `default` task is one that runs when `gulp` is called without a task name. It runs the `css` and `browsersync` tasks to build all files initially (the `images` task is a dependency of `css`). Then, `gulp.watch` is passed a folder to monitor and the associated tasks to run.

Save `gulpfile.js` and run the default task from the command line:

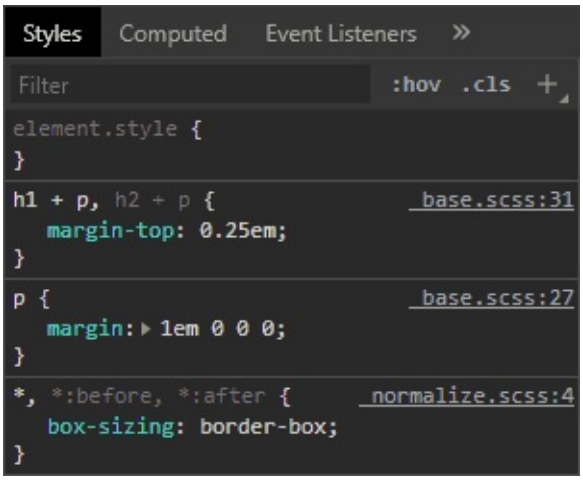
```
gulp
```

The terminal will show a log but, unlike before, it won't terminate and remain running:

```
Gulp development build
[09:43:17] Using gulpfile gulpfile.js
[09:43:17] Starting 'images'...
[09:43:17] Starting 'browsersync'...
[09:43:17] Finished 'browsersync' after 23 ms
[Browsersync] Access URLs:
 ------------------------------------
       Local: http://localhost:8000
    External: http://1.2.3.4:8000
 ------------------------------------
          UI: http://localhost:3001
 UI External: http://1.2.3.4:3001
 ------------------------------------
[Browsersync] Serving files from: ./
[09:43:17] gulp-imagemin: Minified 0 images
[09:43:17] Finished 'images' after 237 ms
[09:43:17] Starting 'css'...
[09:43:17] main.css 11.3 kB
[Browsersync] 1 file changed (main.css)
[09:43:17] Finished 'css' after 173 ms
[09:43:17] Starting 'default'...
[09:43:17] Finished 'default' after 21 ms
```

Your PC is now running a web server from `http://localhost:8000`. Other devices on the network can connect to the **External** URL. Open the URL in a browser or two, then make changes to any `.scss` file. The results are immediately refreshed.

Examine any element in the DevTools and the **Styles** panel will show the location of the pre-compiled Sass code. You can click the filename to view the full source:



Finally, press `Ctrl` + `C` to stop the Gulp task running in your terminal.

Live Production Code

The `NODE_ENV` environment variable must be set to `production` so Gulp tasks know when to produce final code — that is, remove unused CSS, minify files, and disable sourcemap generation. On Linux and macOS terminals:

```
NODE_ENV=production
```

Windows Powershell:

```
$env:NODE_ENV="production"
```

Windows legacy command line:

```
set NODE_ENV=production
```

You can do either of the following:

- Create production code locally then upload to live servers.
- Run Gulp tasks directly on the live server. Ideally, `NODE_ENV` should be permanently set on production machines by modifying the start-up script — for example, add `export NODE_ENV=production` to the end of a Linux `~/.bashrc` file.

Run `gulp css` to generate the final code.

To return to development mode, change `NODE_ENV` to `development` or an empty string.

Next Steps

This article demonstrates a possible Gulp CSS workflow, but it can be adapted for any project:

- There are more than 3,500 Gulp plugins. Many help with CSS and pre-processing, but you can find others for HTML, templating, image handling, JavaScript, server-side languages, linting and more.
- There are hundreds of PostCSS plugins and it' s simple to write your own.

Whichever tools you choose, I recommend you:

- Automate the most frustrating, time-consuming or performance-improving tasks first. For example, optimizing images could shave hundreds of kilobytes from your total page weight.
- Don't over-complicate your build process. A few hours should be adequate to get started.
- Try other task runners, but don't switch on a whim!

Github

The code above is available from github.com/craigbuckler/gulp-css. Please use it as you wish!

Chapter 2: CSS Optimization Tools for Boosting PWA Performance

by Ahmed Bouchefra

When styling websites or PWAs with CSS, you should analyze how CSS resources will affect performance. In this tutorial, we'll use various tools and related techniques to help build a better PWA by focusing on CSS optimization. Specifically, we'll remove the unused CSS, inline the critical path CSS, and minify the resulting code.

The techniques can also be used to improve the performance of general websites and apps. We'll be focusing on CSS optimization for PWAs since they should be fast and feel native on user devices.

Progressive web apps (PWAs) are web experiences that bring the best of both worlds: native mobile apps (installable from a store) and web apps (reachable from public URLs). Users can start using the application right away from their web browser without waiting for a download, installing, or needing extra space in the device.

Service workers and caching allow the app to work offline and when network connectivity is poor. Over time, the app could become faster as more assets are cached locally. PWAs can also be installed as an icon on the home screen and launched full-screen with an initial splash screen.

The Demo PWA to Audit

Github

This example has a code repository available at Github.com.

Before learning how to audit a PWA for any CSS issues, you can get the code of

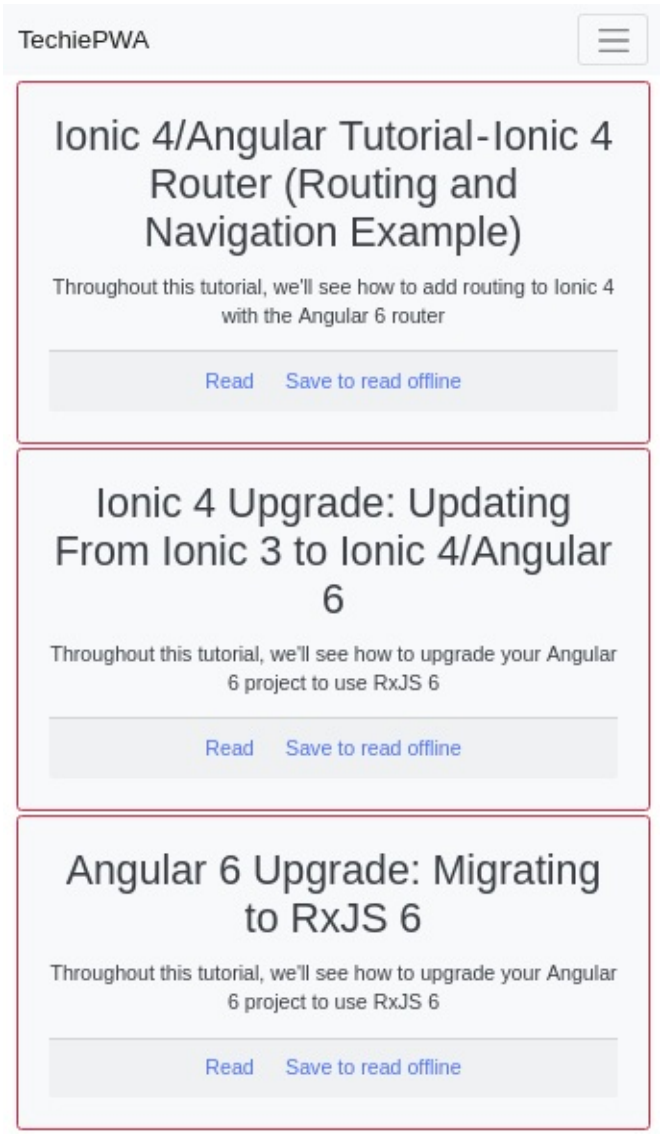
a simple website with PWA features from this GitHub repository. The PWA uses an unminified version of Bootstrap v4 for CSS styling and displays a set of posts fetched from a statically generated JSON API. You can also use the hosted version of this demo, since learning how to build a PWA is beyond the scope of this tutorial.

PWAs are simply web apps with additional features, including these elements:

- **A manifest file**. A JSON file provides the browser with information about the web application such as name, description, icons, the start URL, display factors etc.
- **A service worker**. A JavaScript file is used to cache the application shell (the minimum required HTML, CSS, and JavaScript for displaying the user interface) and proxying all network requests.
- **HTTPS**. PWAs must be served from a secure origin.

Here's a screen shot of the application shell:

A screen shot of the application with data:



Auditing with Google's Lighthouse

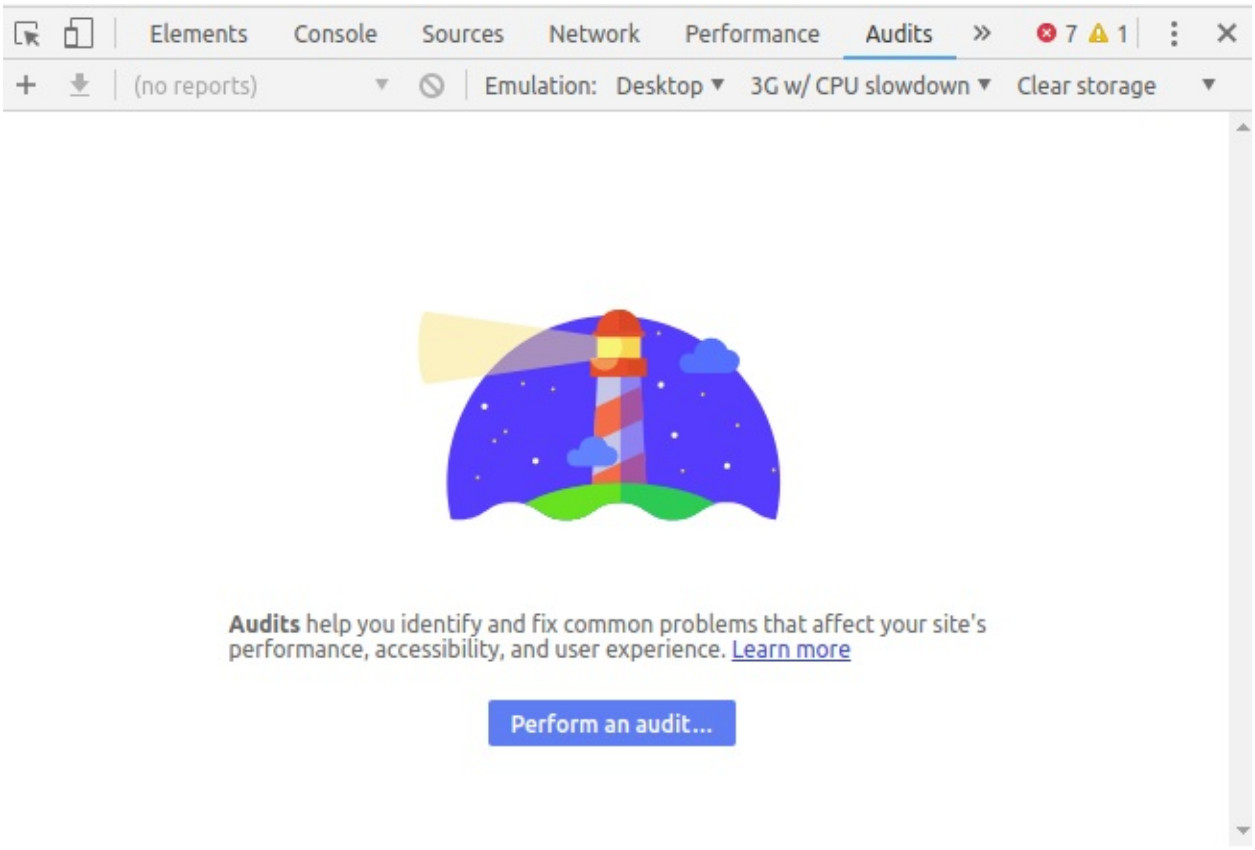
Lighthouse is an open-source auditing tool developed by Google. It can be used to improve the performance, accessibility and SEO of websites and progressive web apps.

Lighthouse can be accessed from the *Audit* tab in Chrome DevTools, programatically as a Node.js module and also as a CLI tool. It takes a URL and runs a series of audits to generate a report with optimization suggestions.

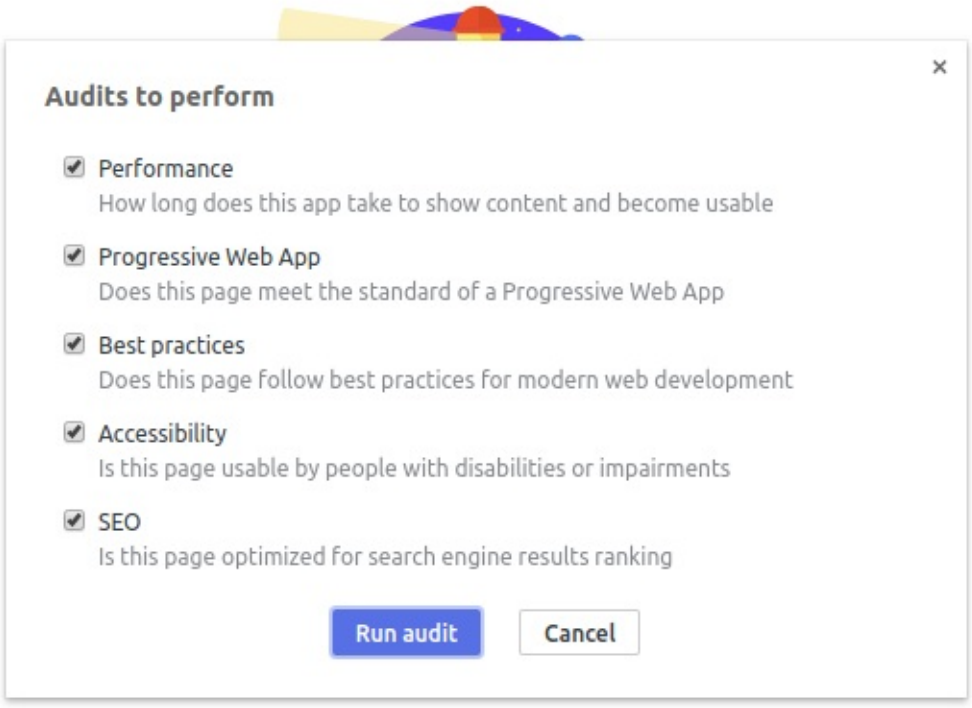
You can apply different techniques either manually or using tools. This article describes how such tools can be used to remove redundant styles, extract the

above-the-fold critical CSS, load the remaining CSS with JavaScript, and minify the resulting code.

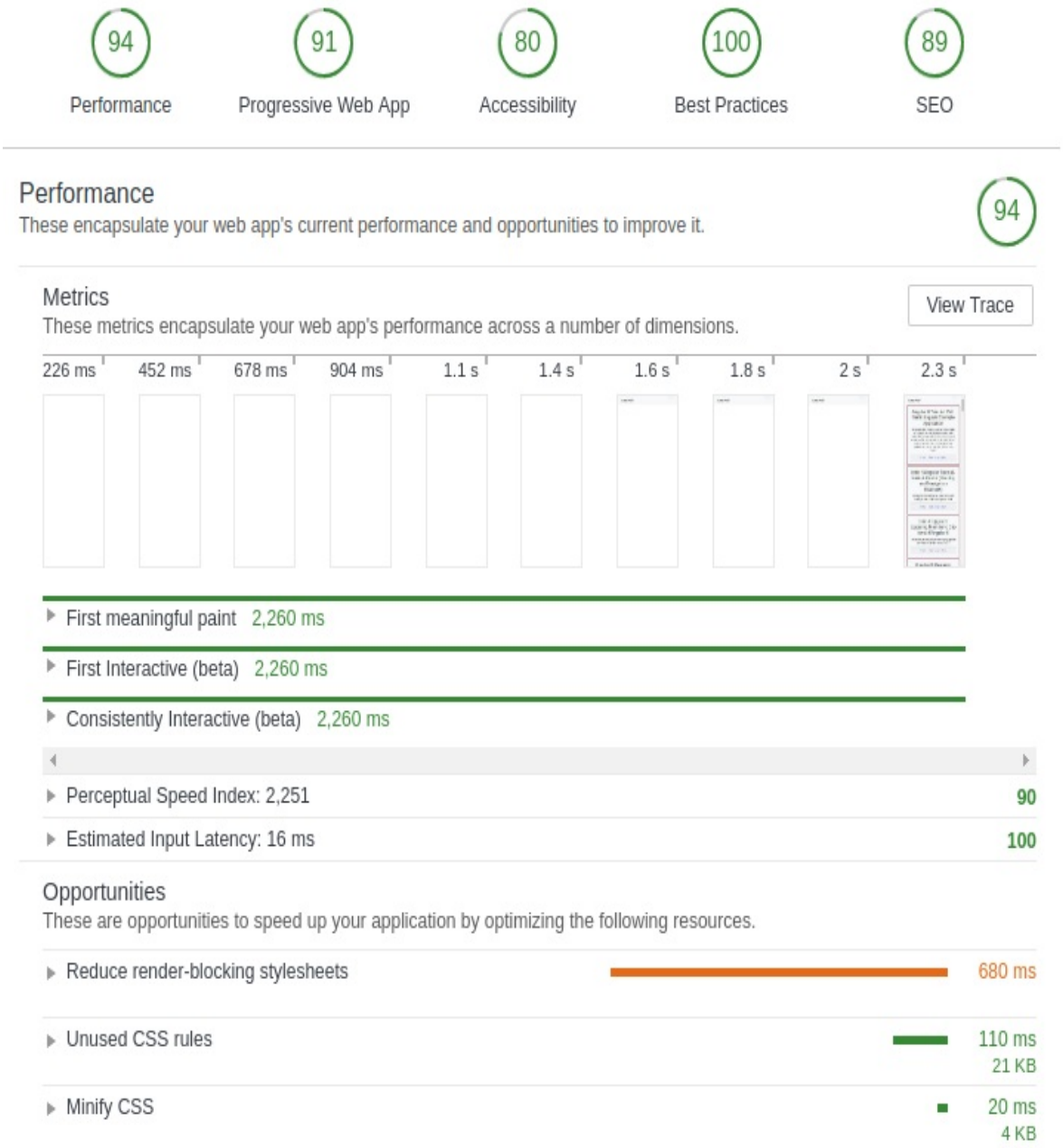
Launch Chrome, visit the PWA address `https://www.techiediaries.com/unoptimizedpwa/` and open *Developer Tools* (`CTRL-Shift-I`). From the Developer Tools, click the *Audits* panel:



Next, click on *Perform an audit…*. A dialog will prompt you for the types of audit you want to perform. Keep all types selected and click the *Run audit* button.



Wait for Lighthouse to complete the auditing process and generate a report:



The scores are calculated in a simulated environment. You're unlikely to get the same results on your machine because they depend on hardware and network capabilities.

From the report, you can see a timeline which visually shows how the page is loaded. *First meaningful paint*, *First Interactive* and *Consistently Interactive* are key time points that describe how fast the page loaded. Our goal is to optimize these metrics according to the *Critical Rendering Path*.

Critical Rendering Path

The critical rendering path represents the steps the browser must take before it can render an initial page view (the visible area of the page or the “above-the-fold” area). When you visit a URL, the browser:

- starts to download the HTML which is parsed as it streams
- identifies and downloads external assets such as CSS, JavaScript, fonts and images
- parses and renders as necessary.

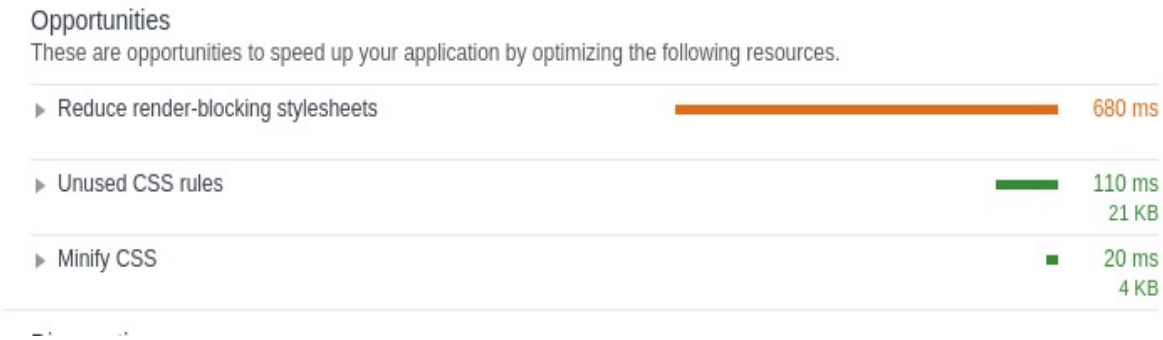
Asset downloading and parsing can be done in parallel for assets such as images. However, CSS and JavaScript are render-blocking: the browser halts further processing until the file is downloaded and parsed. This is necessary because browsers are single-threaded and anything could occur. For example, the page redirects to another URL or changes layout styles.

Page performance should improve if the number of render-blocking assets is reduced.

CSS Optimization Using Lighthouse Opportunities

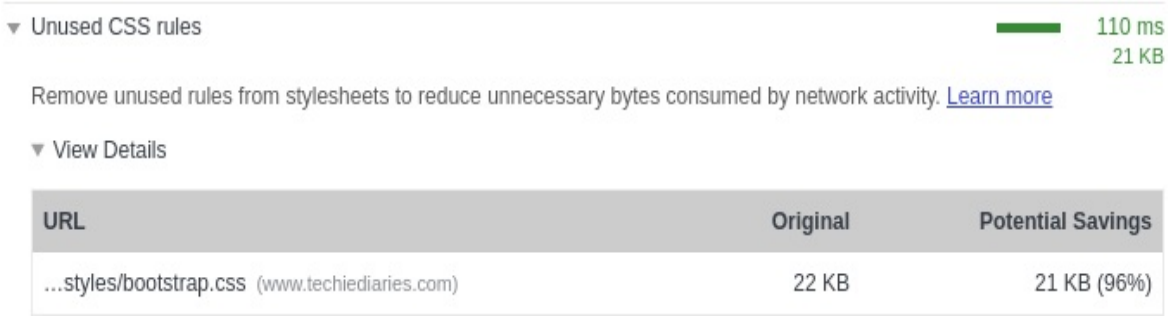
Lighthouse provides guidelines to help optimize the application in the *Opportunities* section of the report. There are three suggestions to improve performance:

- reduce render-blocking CSS
- eliminate unused CSS rules
- minify CSS.



CSS Optimization: Removing Unused CSS Rules

Let's start with the second Lighthouse opportunity, which relates to unused CSS rules. Expand the *Unused CSS rules* opportunity:



Lighthouse estimates that 96% of the CSS in the `bootstrap.css` file is unused by the application. If we eliminate the unused CSS, network activity will reduce accordingly.

All the tools described here can be used as part of build system such as webpack, Gulp or Grunt, but this tutorial uses them on the command line.

To eliminate the unused CSS styles, we'll use PurifyCSS. Make sure you have Node.js installed on your machine, then install the tool globally using `npm`:

```
$ npm install -g purify-css
```

Next, make sure you're inside your `unoptimizedpwa` folder and run the command:

```
$ purifycss styles/bootstrap.css index.html -o styles/purified.css -
```

- The first argument is the CSS file to *purify*.

- The second argument is the HTML file to check for used styles.
- The `-o` option specifies the path and name of the result file to create.
- The `-i` option instructs the tool to display information about how much CSS was removed.

(An `-m` option instructs the tool to minify the purified CSS, but we'll use another tool for minification below.)

In this case, we see `PurifyCSS has reduced the file size by ~ 84.5%`. This is not quite the 96% identified by Lighthouse, but the tools use different techniques.

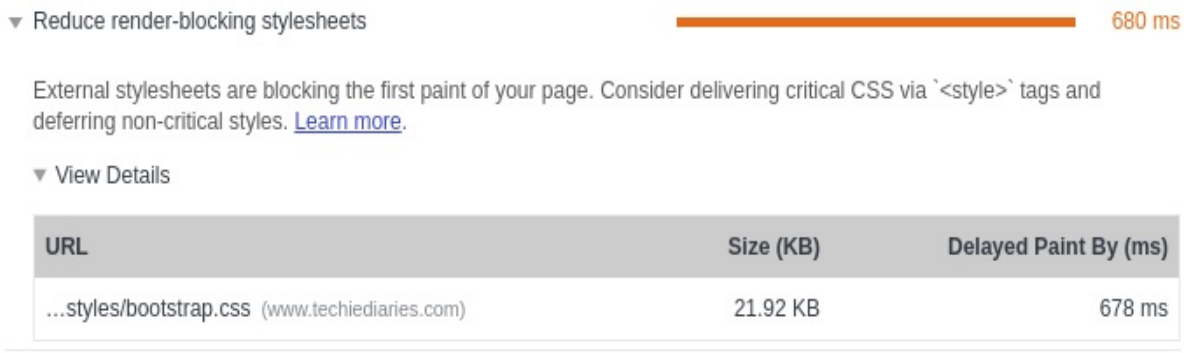
```
bb/home/ahmed/Desktop/Desktop/newtechie/unoptimizedpwa$ purifycss styles/bootstrap.css
index.html -o styles/purified.css -i


    |
    |   PurifyCSS has reduced the file size by ~ 84.5%
    |
```

You can also use other tools to remove unused CSS such as uncss and you can read this article by Addy Osmani for more information.

You can now remove the original `styles/bootstrap.css` file and rename `styles/purified.css` to `styles/bootstrap.css`.

Reduce Render-Blocking CSS

If we expand the *Reduce render-blocking stylesheets* opportunity, we can see more details:

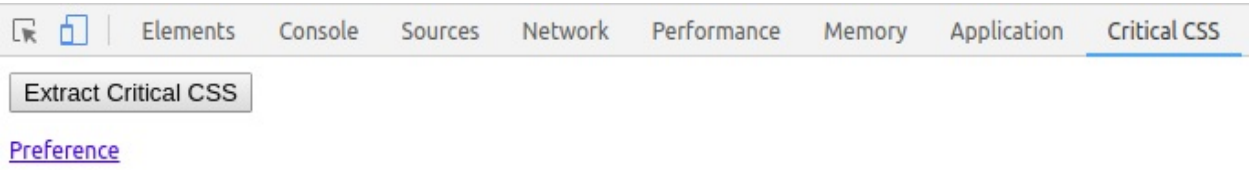


The `bootstrap.css` file delays the first paint of our application by 678ms. Not all styles are necessary for above-the-fold rendering, so we can do the following:

1. Extract the critical CSS from `bootstrap.css` and inline it in the `index.html` file using a `<style>` tag. This reduces the number of HTTP requests and is less code for the browser to parse.
2. Deliver the remaining non-critical styles after the top of the page has rendered.

We'll use Critical CSS Extractor, a Chrome extension to extract Critical CSS rules for the current page. Once you've installed this extension, you'll see a new panel — *Critical CSS* — in DevTools.

Chrome does the hard work for us. Open DevTools again with the site still active and select the *Critical CSS* tab. Click the *Extract Critical CSS* button to download the critical CSS file:

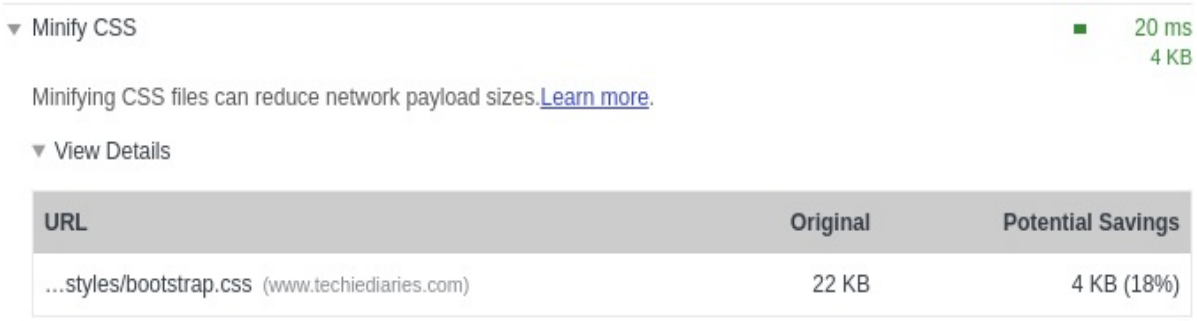


Open `index.html` and add the content of this critical CSS file within a `<style>` tag in the HTML `<head>`.

Unminified CSS

The CSS file is still required with all styles, but we can minify it to reduce its size.

Expand the *Minify CSS* opportunity in the Lighthouse audit:



To minify the file, we can use tools such as:

- cssnano— a modular minifier, built on top of PostCSS
- csso — a CSS minifier with structural optimizations.

To use *cssnano*, from the `unoptimizedpwa` folder, run the following command to install *cssnano* locally:

```
npm install cssnano
```

Next, install the PostCSS CLI globally using:

```
npm install postcss-cli --global
```

Add a new file named `postcss.config.js` with the following (default) cssnano configuration parameters:

```
module.exports = {
    plugins: [
        require('cssnano')({
            preset: 'default',
        }),
    ],
};
```

The cssnano guide provides more information about configuration presets.

Minify the `bootstrap.css` file by running:

```
postcss styles/bootstrap.css > styles/bootstrap.min.css
```

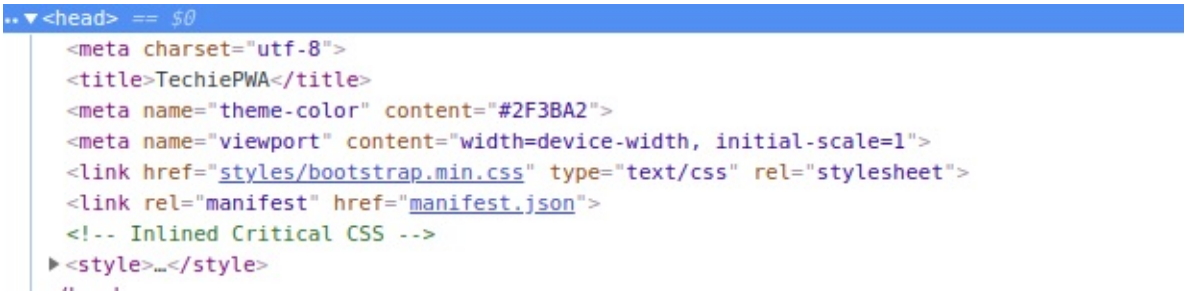
Deferring Bootstrap Loading with JavaScript

To prevent the CSS file from render-blocking, it can be loaded at the end of the

page using JavaScript after the DOM is ready. Add the following JavaScript code snippet to `index.html` just before the closing `</body>` tag:

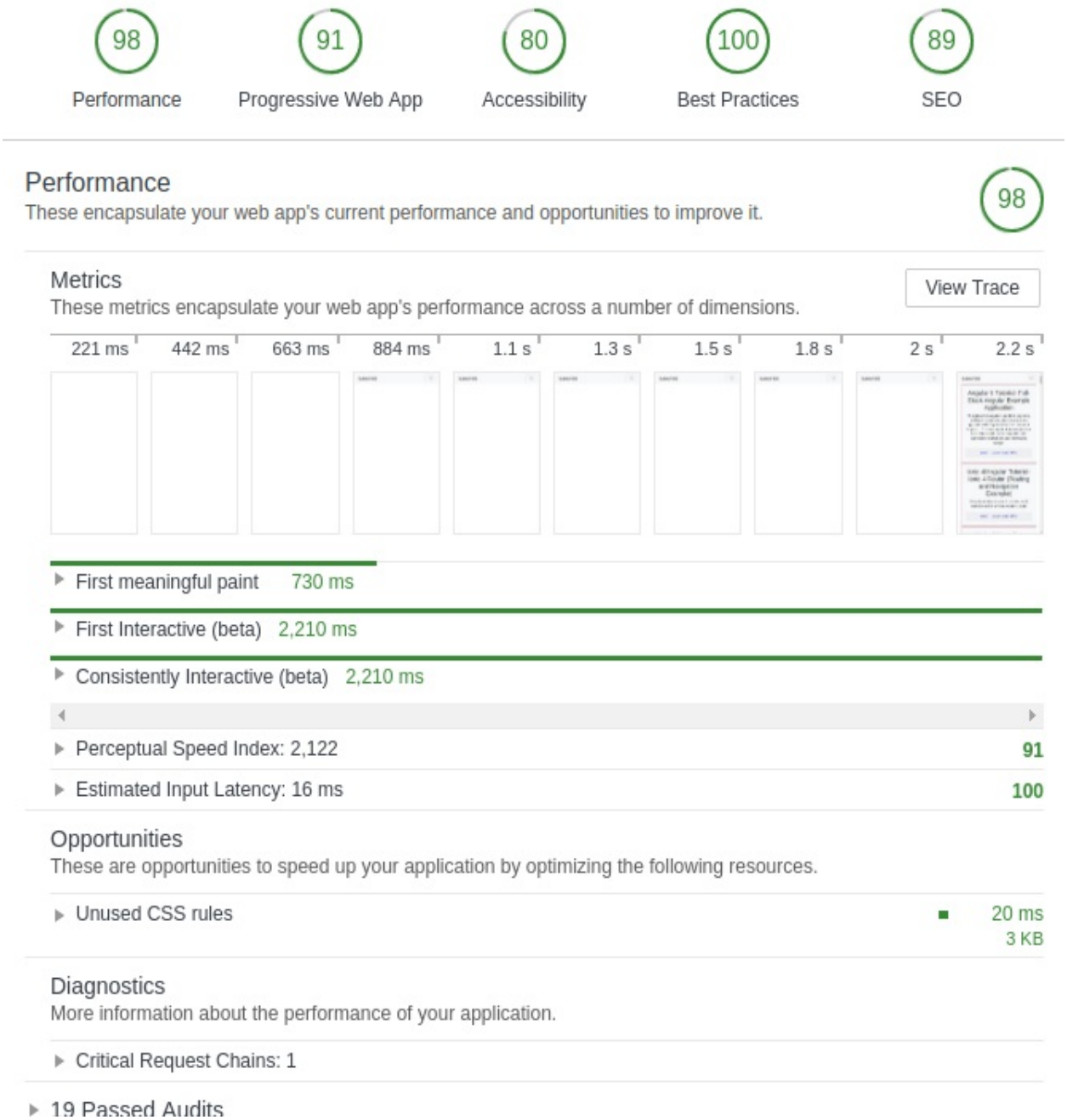
```
<script>
const link = document.createElement('link');
link.href = 'styles/bootstrap.min.css';
link.type = 'text/css';
link.rel = 'stylesheet';
const  link0 = document.getElementsByTagName('link')[0];
link0.parentNode.insertBefore(link, link0);
</script>
```

If you inspect your DOM after the page has loaded, you'll find the inserted `<link>` tag inside the `<head>` tag:



Checking the Optimizations

We can run Lighthouse again against the optimized PWA. The results:



The performance score has improved.

Lighthouse still reports further unused CSS rules could be removed. This could occur because [PurifyCSS doesn’tt remove similar named selectors](#).

The final optimized PWA is available from this [GitHub repository](#).

Similar Tools

There are many alternative tools to the ones we used in this tutorial. Here's a list of well-known tools for purifying, minifying and extracting critical CSS:

- csso — a CSS minifier with structural optimizations
- critical, a tool by Addy Osmani to extract and inline critical-path CSS in HTML pages
- uncss, a tool for removing unused CSS from your stylesheets that works across multiple files and supports Javascript-injected CSS
- purgecss, a tool for removing unused CSS.

Conclusion

In this tutorial, we moved towards CSS optimization by removing redundant code, inlining critical assets and minifying the resulting CSS. As a result, the PWA will download quicker and render faster. Similar tools and techniques can be used on your sites and apps for optimizing CSS to increase performance.

Chapter 3: CSS Debugging and Optimization: Code Quality Tools

by Tiffany B. Brown

The following introduction to CSS code-quality tools is an extract from Tiffany's new book, CSS Master, 2nd Edition.

On your road to becoming a CSS master, you'll need to know how to troubleshoot and optimize your CSS. How do you diagnose and fix rendering problems? How do you ensure that your CSS creates no performance lags for end users? And how do you ensure code quality?

Knowing which tools to use will help you ensure that your front end works well.

In this article, we'll discuss tools that help you analyze the quality of your CSS. We'll focus on two:

- stylelint
- UnCSS

stylelint is a linting tool. A **linter** is an application that checks code for potential trouble spots, enforcing coding conventions such as spaces instead of tabs for indentation. stylelint can find problems such as duplicate selectors, invalid rules, or unnecessary specificity. These have the greatest impact on CSS maintainability.

UnCSS, on the other hand, checks your CSS for unused selectors and style rules. It parses a stylesheet and a list of HTML pages, returning a CSS file that's stripped of unused rules.

Both of these tools use Node.js and can be installed using npm.

If you're working on a small site, such as a personal blog or a few pages that are updated infrequently, many of the problems that these tools flag can safely be ignored. You'll spend time refactoring for little gain in maintainability and

speed. For larger projects, however, they're invaluable. They'll help you head off maintainability problems before they start.

stylelint

stylelint helps you avoid errors and enforce conventions in your styles. It has more than 160 error-catching rules and allows you to create your own as well via plugins.

stylelint Installation and Configuration

Install stylelint as you would any other npm package:

```
npm install -g stylelint
```

Once it's installed, we'll need to configure stylelint before using it. stylelint doesn't ship with a default configuration file. Instead, we need to create one. Create a `.stylelistrc` file in your project directory. This file will contain our configuration rules, which can use JSON (JavaScript Object Notation) or YAML (YAML Ain't Markup Language) syntax. Examples in this section use JSON.

Our `.stylelistrc` file must contain an object that has a `rules` property. The value of `rules` will itself be an object containing a set of `stylelist` rules and their values:

```
{
    "rules": {}
}
```

If, for example, we wanted to banish `!important` from declarations, we can set the `declaration-no-important` to `true`:

```
{
    "rules": {
        "declaration-no-important": true
    }
}
```

stylelint supports over 150 rules that check for syntax errors, indentation and line-break consistency, invalid rules, and selector specificity. You'll find a complete list of rules and their available values in the stylelint User Guide.

Starting with a Base stylelint Configuration

You'll probably find it easier to start with a base configuration and then customize it to your project needs. The `stylelint-config-recommended` base configuration is a good starting configuration. It enables all of the "possible errors" rules. Install it using npm:

```
npm install -g stylelint-config-recommended
```

Then, in your project directory, create a `.stylelistrc` file that contains the following lines:

```
{
    "extends": "/absolute/path/to/stylelint-config-recommended"
}
```

Replace `/absolute/path/to/` with the directory to which `stylelint-config-recommended` was installed. Global npm packages can usually be found in the `%AppData%\npm\node_modules` directory on Windows 10 systems, and in `/usr/local/lib/node_modules` on Unix/Linux and macOS systems. Type `npm list -g` to locate your global `node_modules` directory.

We can then augment our configuration by adding a `rules` property. For example, to disallow vendor prefixes, our `.stylelistrc` file would look similar to the what's below:

```
{
    "extends": "/absolute/path/to/stylelint-config-recommended",
    "rules": {
       "value-no-vendor-prefix": true
    }
}
```

What if we wanted to limit the maximum specificity of our selectors to `0,2,0`? That would permit selectors such as `.sidebar .title` but not `#footer_nav`. We can do this by adding a `selector-max-specificity` rule to our configuration:

```
{
    "extends": "/absolute/path/to/stylelint-config-recommended",
    "rules": {
       "value-no-vendor-prefix": true,
       "selector-max-specificity": "0,2,0"
    }
```

```
}
```

Using stylelint

To lint your CSS files using `stylelint`, run the `stylelint` command, passing the path to your CSS file as an argument:

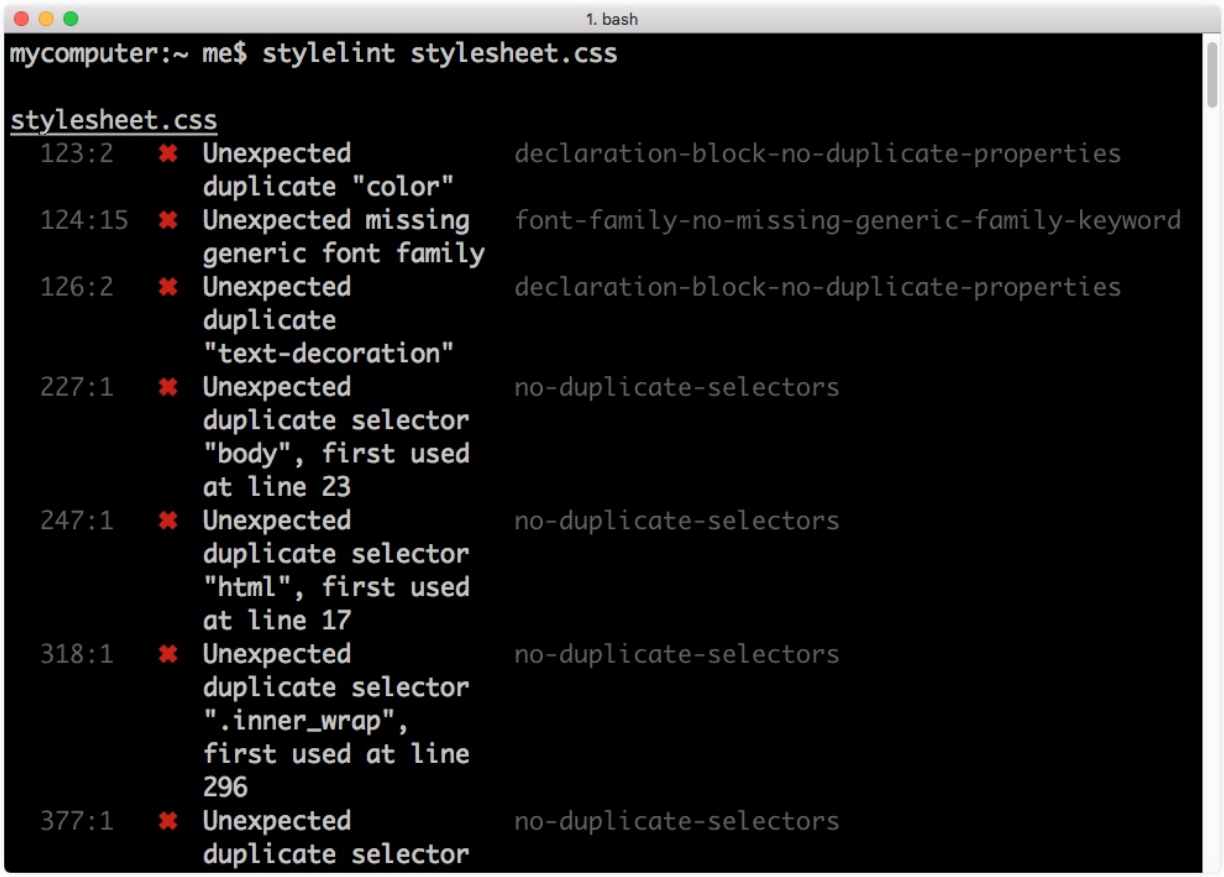
```
stylelint stylesheet.css
```

Alternatively, you can lint all of the CSS files in a particular directory, even recursively:

```
stylelint "./css/**/*.css"
```

stylelint can also lint CSS that's embedded in HTML files using the `style` element. Just pass the path to an HTML file as the argument.

When complete, stylelint will display a list of files that contain errors, along with their type and location, as shown in the image below.



UnCSS

UnCSS parses your HTML and CSS files, removing unused CSS. If your projects include a CSS framework such as Bootstrap or use a reset stylesheet, consider adding UnCSS to your workflow. It will shave unnecessary CSS—and bytes—from your code.

UnCSS Installation

As with other npm packages, you can install UnCSS using the following command:

```
npm install -g uncss
```

Using UnCSS from the Command Line

UnCSS requires the file path or URL of an HTML page that contains a linked CSS file. For example:

```
uncss http://sitepoint.com/
```

UnCSS will parse the HTML and its linked stylesheets, and print the optimized CSS to standard output. To redirect to a file, use the redirect operator (>):

```
uncss http://sitepoint.com/ > optimized.css
```

You can also pass multiple file paths or URLs to the command line. UnCSS will analyze each file and dump optimized CSS that contains rules affecting one or more pages:

```
uncss index.html article-1.html article-2.html > optimized.css
```

For a full list of commands—and an example of how to use UnCSS with a Node.js script—consult the UnCSS docs.

Consider a Task Runner or Build Tool

Running these tools probably seems like a lot of extra work. To that end, consider adding a task runner or build system to your workflow. Popular ones

include Grunt, Gulp, and webpack. All three have robust documentation and sizable developer communities.

What's great about these task runners and build systems is that they automate concatenation and optimization tasks. They're not limited to CSS either. Most build tools also include plugins for optimizing JavaScript and images.

Because the configuration and build script files are typically JSON and JavaScript, you can easily reuse them across projects or share them with a team. Each of the tools mentioned in this article can be integrated with Grunt, Gulp, or webpack with the help of a plugin.

Above all, however, take a pragmatic approach to building your toolkit. Add tools that you think will enhance your workflow and improve the quality of your output.

Chapter 4: CSS Debugging and Optimization: Developer Tools

by Tiffany B. Brown

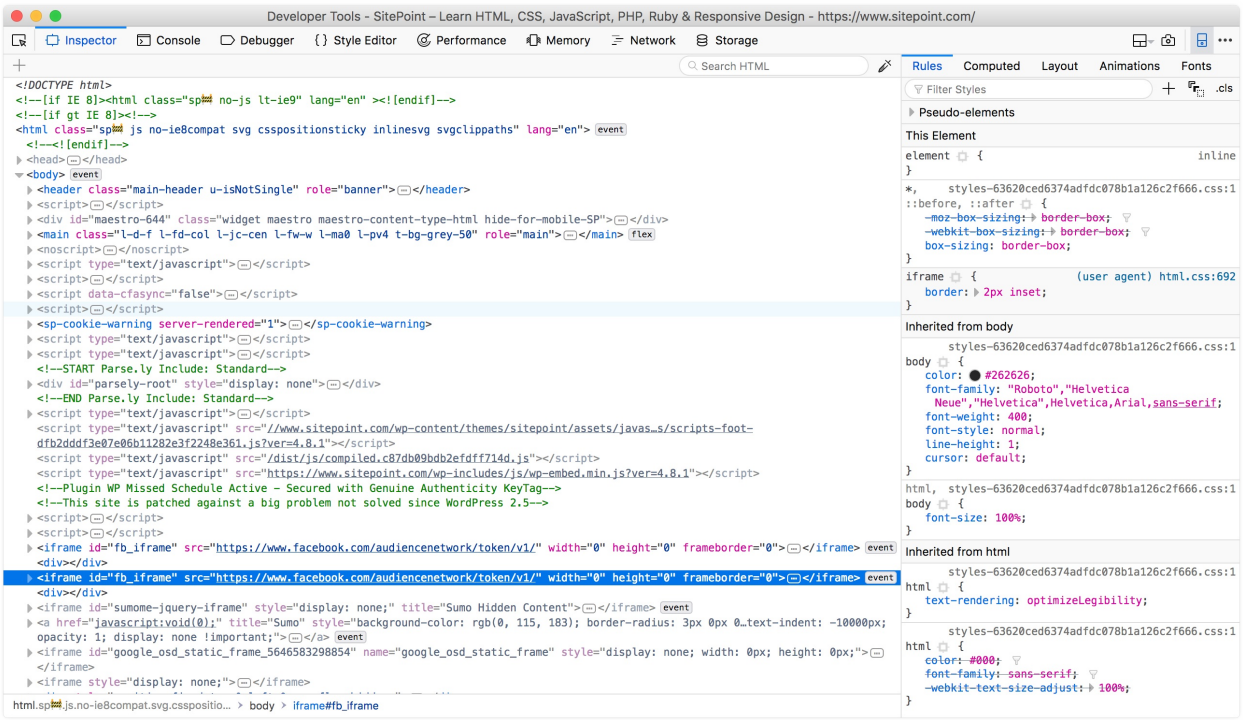
The following introduction to CSS developer tools is an extract from Tiffany's new book, CSS Master, 2nd Edition.

On your road to becoming a CSS master, you'll need to know how to troubleshoot and optimize your CSS. How do you diagnose and fix rendering problems? How do you ensure that your CSS creates no performance lags for end users? And how do you ensure code quality?

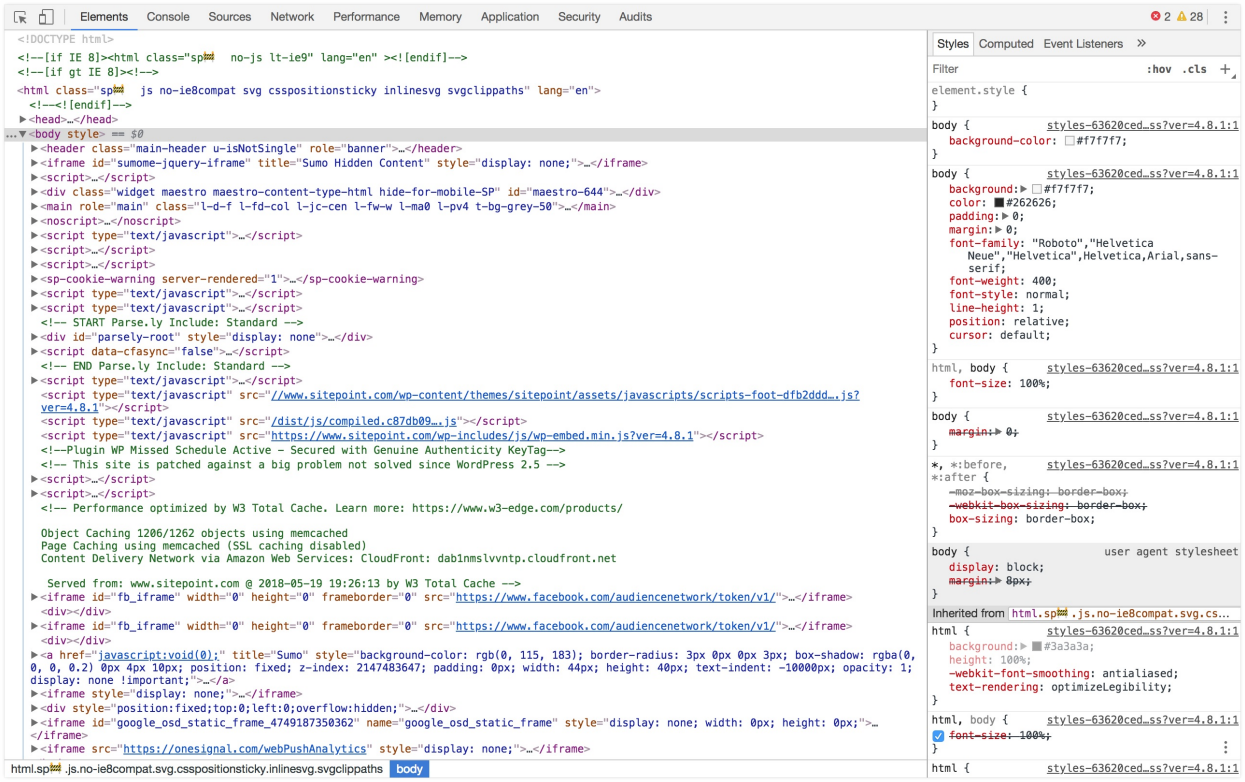
Knowing which tools to use will help you ensure that your front end works well.

In this article, we'll delve into the browser-based developer tools for Chrome, Safari, Firefox, and Microsoft Edge.

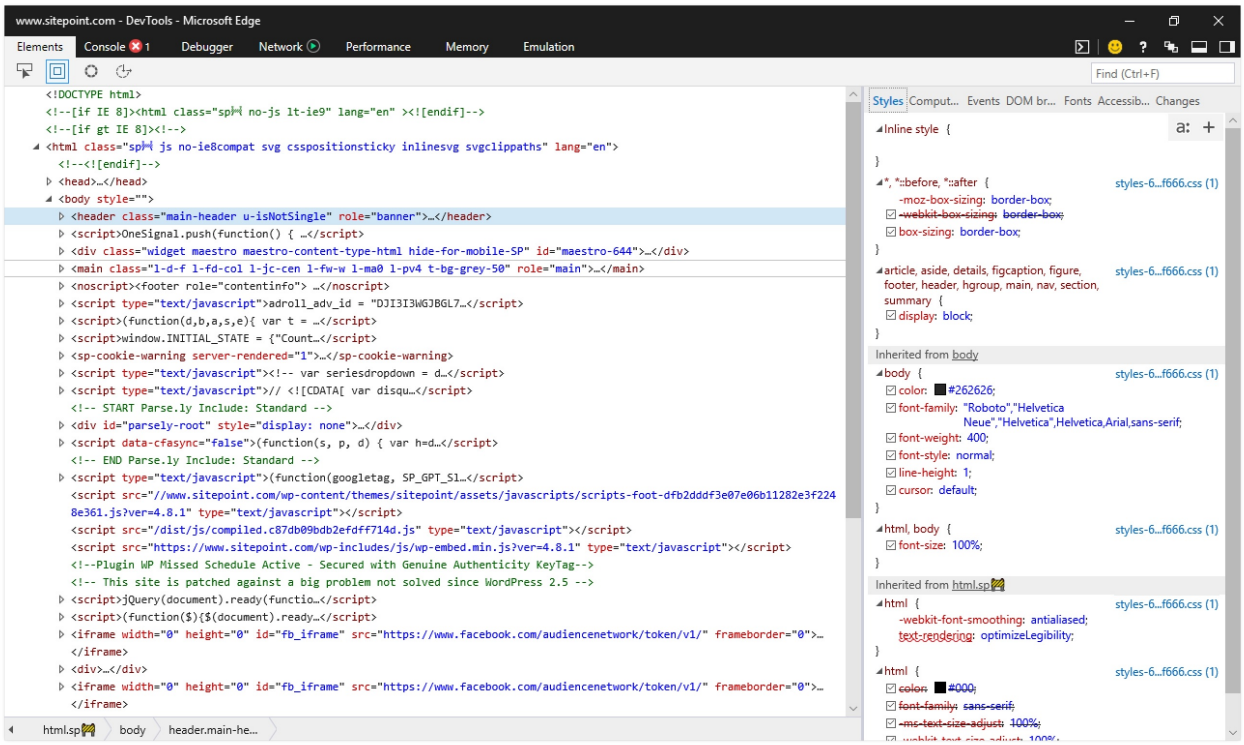
Most desktop browsers include an element inspector feature that you can use to troubleshoot your CSS. Start using this feature by right-clicking and selecting **Inspect Element** from the menu. Mac users can also inspect an element by clicking the element while pressing the **Ctrl** key. The image below indicates what you can expect to see in Firefox Developer Edition.



In Firefox, Chrome and Safari you can also press **Ctrl** + **Shift** + **I** (Windows/Linux) or **Cmd** + **Option** + **I** (macOS) to open the developer tools panel. The screenshot below shows the Chrome developer tools.



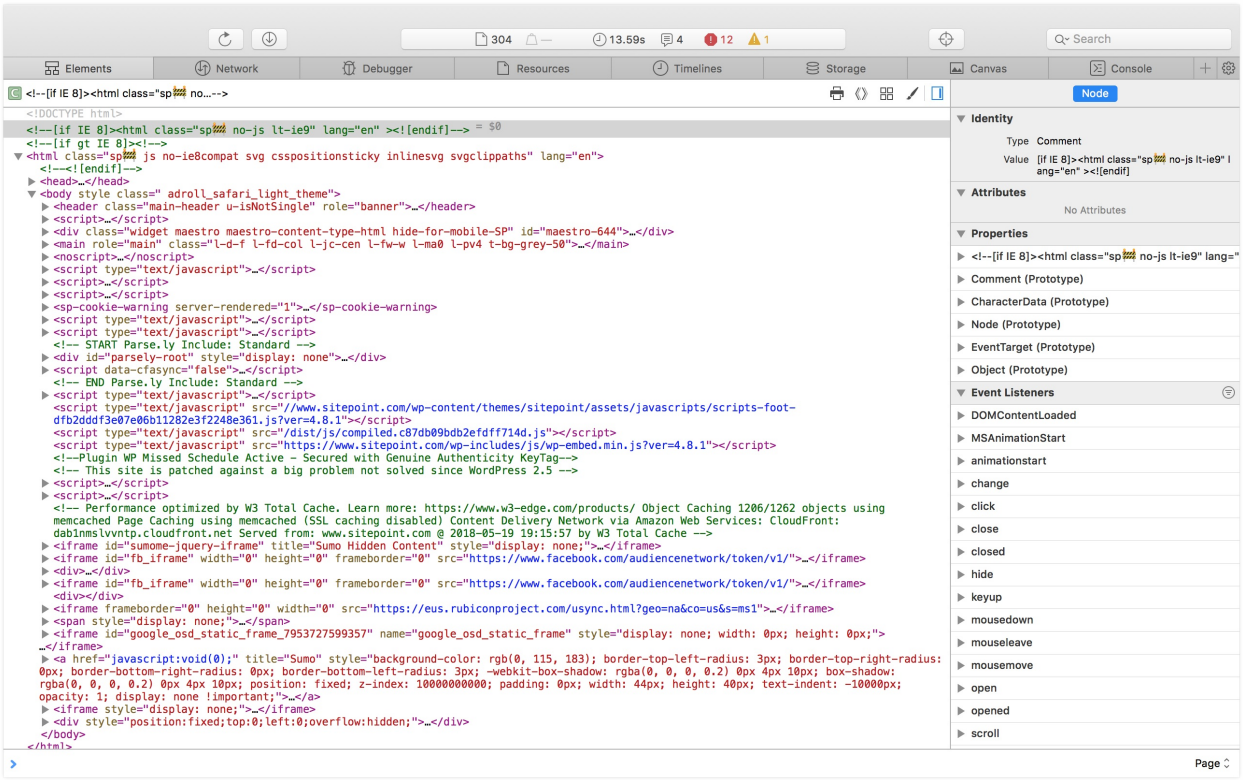
While in Microsoft Edge, open developer tools by pressing the **F12** key, as shown in figure 2-3.



You can also open each browser's developer tools using the application's menu:

- Microsoft Edge: **Tools > Developer Tools**
- Firefox: **Tools > Web Developer**
- Chrome: **View > Developer**
- Safari: **Develop > Show Web Inspector**

In Safari, you may have to enable the **Develop** menu first by going to **Safari > Preferences… > Advanced** and checking the box next to **Show Develop menu in menu bar**. The view for Safari developer tools is illustrated below.



After opening the developer tools interface, you may then need to select the correct panel:

- Microsoft Edge: **DOM Explorer**
- Firefox: **Inspector**
- Chrome: **Elements**
- Safari: **Elements**

You'll know you're in the right place when you see HTML on one side of the panel, and CSS rules on the other.

Generated Markup

The markup you'll see in the HTML panel is a representation of the DOM. It's generated when the browser finishes parsing the document and may differ from your original markup. Using **View Source** reveals the original markup, but keep in mind that for JavaScript applications there may not be any markup to view.

Using the Styles Panel

Sometimes an element isn’t styled as expected. Maybe a typographical change failed to take, or there’s less padding around a paragraph than you wanted. You can determine which rules are affecting an element by using the **Styles** panel of the Web Inspector.

Browsers are fairly consistent in how they organize the **Styles** panel. Inline styles, if any, are typically listed first. These are styles set using the `style` attribute of HTML, whether by the CSS author or programmatically via scripting.

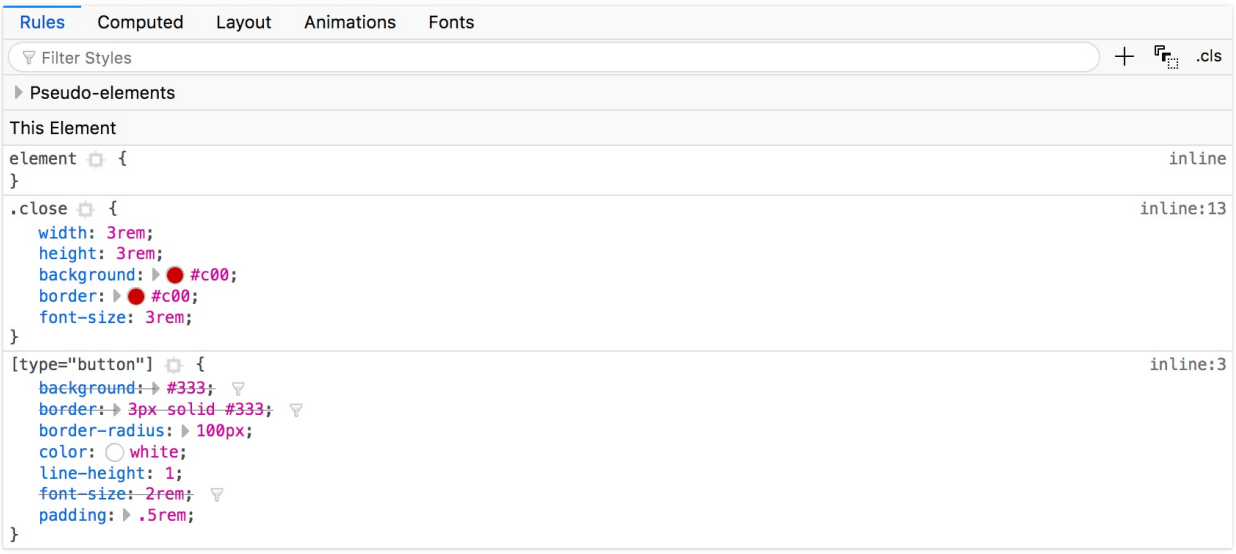
Inline styles are followed by a list of style rules applied via author stylesheets—those written by you or your colleagues. Styles in this list are grouped by media query and/or filename.

Authored style rules precede user agent styles. User agent styles are the browser’s default styles. They too have an impact on your site’s look and feel.In Firefox, you may have to select the **Show Browser Styles** option in order to view user agent styles. You can find this setting in the **Toolbox Options** panel.

Properties and values are grouped by selector. A checkbox sits next to each property, letting you toggle specific rules on and off. Clicking on a property or value allows you to change it, so you can avoid having to edit, save and reload.

Identifying Cascade and Inheritance Problems

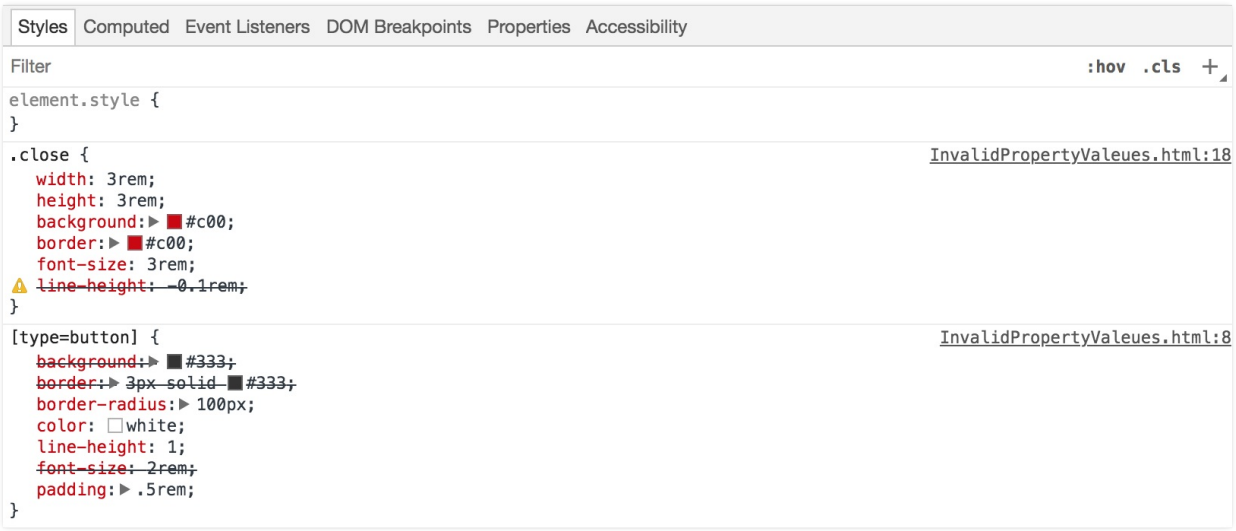
As you inspect styles, you may notice that some properties appear crossed out. These properties have been overridden either by a cascading rule, a conflicting rule, or a more specific selector, as depicted below.



In the image above, the `background`, `border`, and `font-size` declarations of the `[type=button]` block are displayed with a line through them. These declarations were overridden by those in the `.close` block, which succeeds the `[type=button]` in our CSS.

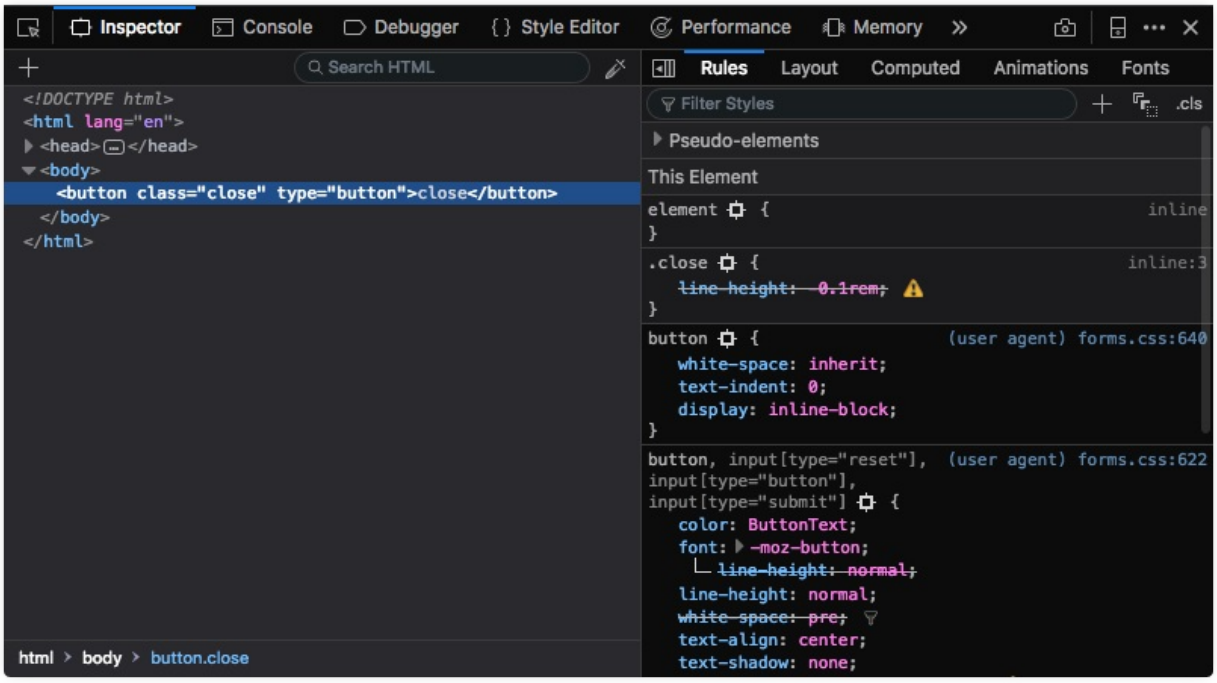
Spotting Invalid Properties and Values

You can also use the element inspector to spot invalid or unsupported properties and property values. In Chromium-based browsers, invalid CSS rules both have a line through them and an adjacent warning icon, which can be seen below.

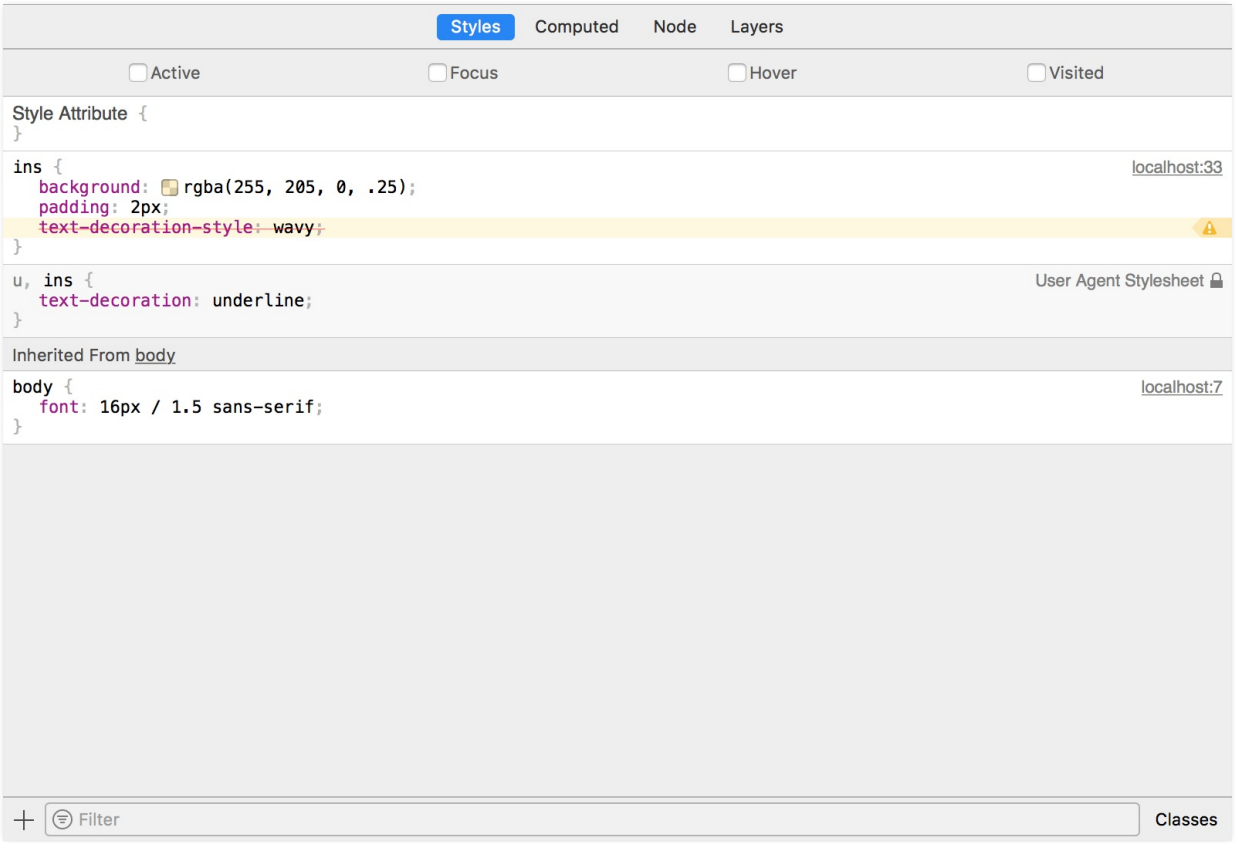


Firefox also strikes through invalid or unsupported properties and values.

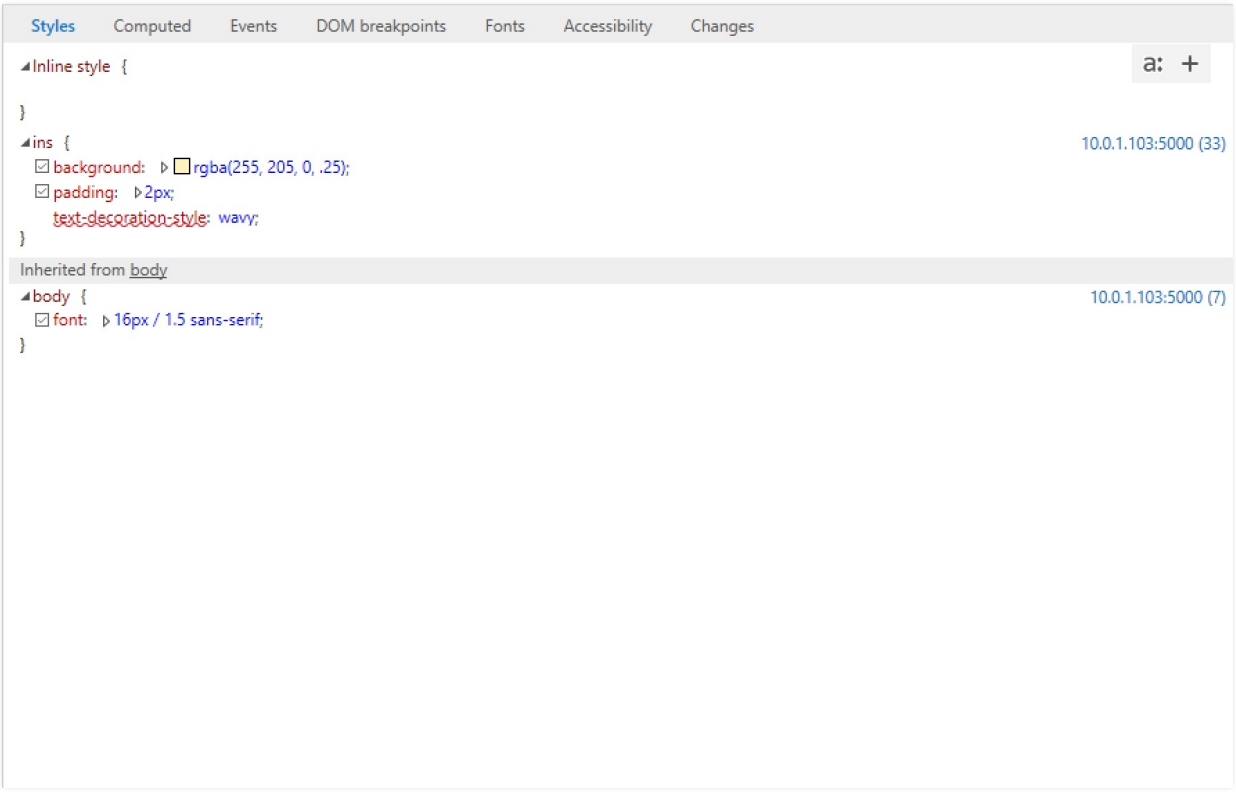
Firefox Developer Edition also uses a warning icon, as shown below. Standard Firefox displays errors similarly, but doesn't include the warning icon.



In the screenshot below, Safari strikes through unsupported rules with a red line, and highlights them with a yellow background and warning icon.



Microsoft Edge instead uses a wavy underline to indicate unsupported properties or values.



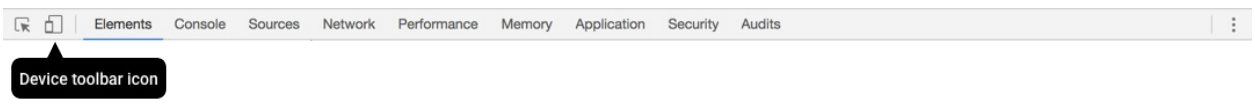
When it comes to basic debugging and inheritance conflicts, whichever browser you choose doesn't matter. Familiarize yourself with all of them, however, for those rare occasions when you need to diagnose a browser-specific issue.

Debugging Responsive Layouts

On-device testing is always best. During development, however, it's helpful to simulate mobile devices with your desktop browser. All major desktop browsers include a mode for responsive debugging.

Chrome

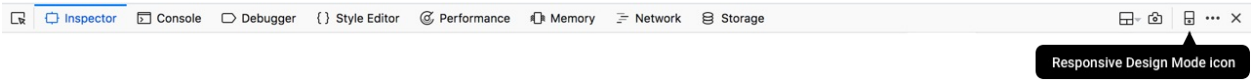
Chrome offers a device toolbar feature as part of its developer toolkit. To use it, click the device icon (pictured below) in the upper-left corner, next to the **Select an element** icon.



Device mode lets you mimic several kinds of Android and iOS devices, including older devices such as the iPhone 5 and Galaxy S5. Device mode also includes a network throttling feature for approximating different network speeds, and the ability to simulate being offline.

Firefox

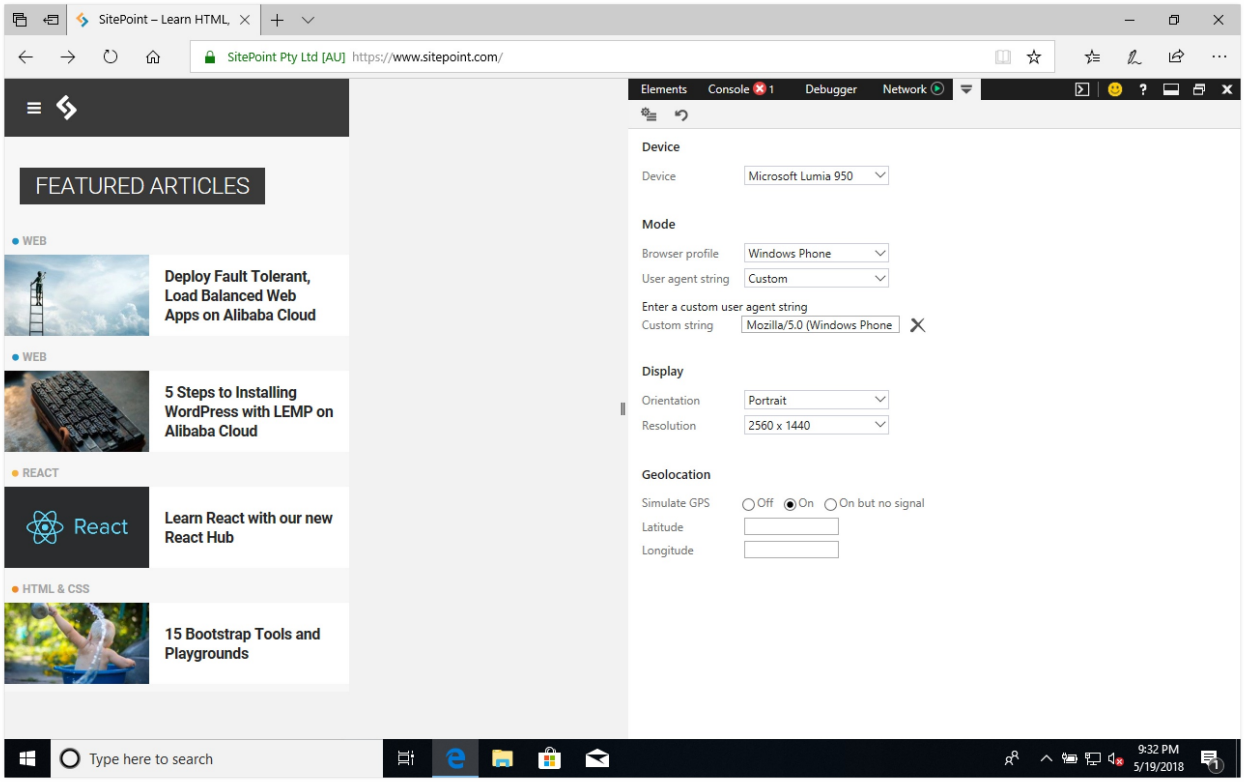
In Firefox, the equivalent mode is known as **Responsive Design Mode**. Its icon resembles early iPods. You'll find it on the right side of the screen, in the developer tools panel, as shown below.



In responsive mode, you can toggle between portrait and landscape orientations, simulate touch events, and capture screenshots. Like Chrome, Firefox also allows developers to simulate slow connections via throttling.

Microsoft Edge

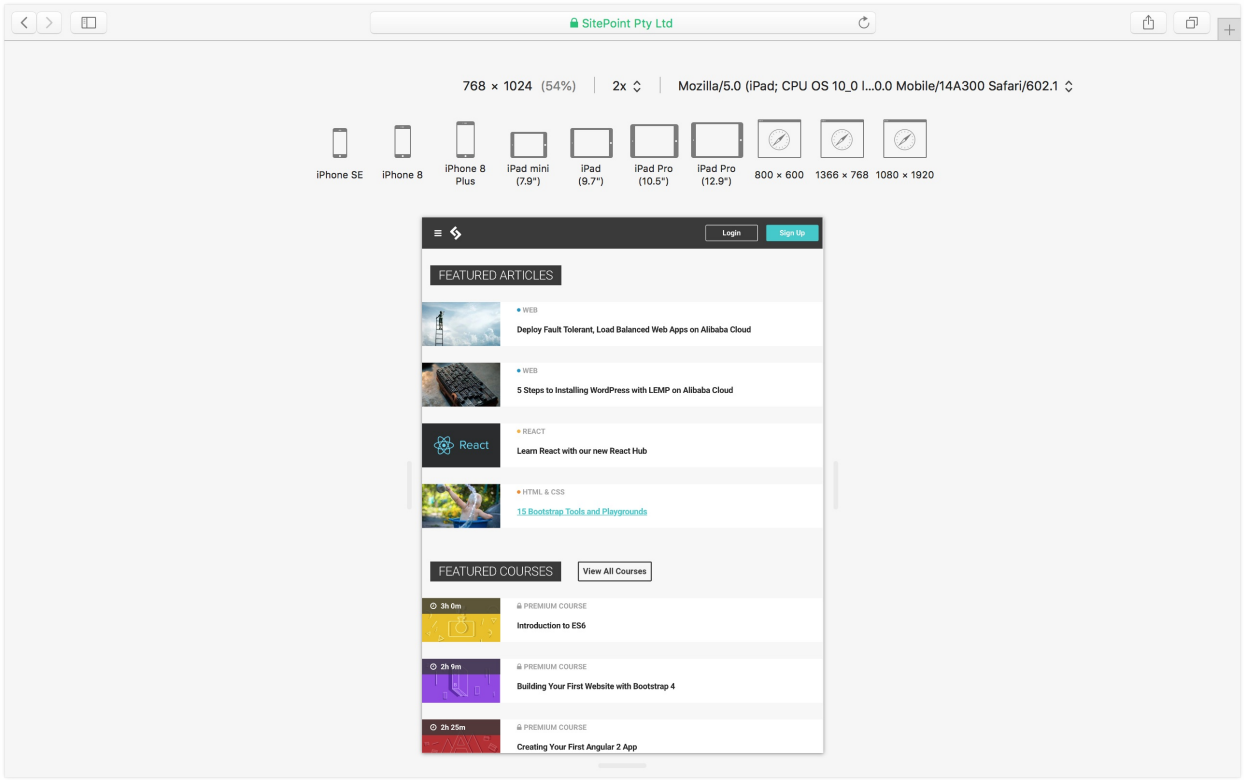
Microsoft Edge makes it possible to mimic Windows mobile devices—such as the Surface—with its **Emulation** tab. Select **Windows Phone** from the **Browser profile** menu, as shown below.



In addition to mimicking orientation and resolution, emulation mode enables you to test geolocation features. However, you can't use its emulation mode to simulate network conditions.

Safari

Safari's Responsive Design Mode is in its developer toolkit. It's similar to Emulation Mode in Firefox, but adds the ability to mimic iOS devices, as illustrated below.



To enter Safari's responsive design mode, select **Develop > Enter Responsive Design Mode**, or **Cmd** + **Ctrl** + **R**.

Chapter 5: CSS Debugging and Optimization: Minification with CSSO

by Tiffany B. Brown

The following introduction to CSS minification tools is an extract from Tiffany's new book, CSS Master, 2nd Edition.

On your road to becoming a CSS master, you'll need to know how to troubleshoot and optimize your CSS. How do you diagnose and fix rendering problems? How do you ensure that your CSS creates no performance lags for end users? And how do you ensure code quality?

Knowing which tools to use will help you ensure that your front end works well. In this article, we'll discuss CSS minification.

Developer tools help you find and fix rendering issues, but what about efficiency? Are our file sizes as small as they can be? For that, we need minification tools.

Minification in the context of CSS simply means "removing excess characters." Consider, for example, this block of code:

```
h1 {
    font: 16px / 1.5 'Helvetica Neue', arial, sans-serif;
    width: 80%;
    margin: 10px auto 0px;
}
```

That's 98 bytes long, including line breaks and spaces. Let's look at a minified example:

```
h1{font:16px/1.5 'Helvetica Neue',arial,sans-serif;width:80%;
    ➥margin:10px auto 0}
```

Now our CSS is only 80 bytes long—an 18% reduction. Fewer bytes, of course, means faster download times and data transfer savings for you and your users.

In this section, we'll look at CSS Optimizer, or CSSO, a minification tool that runs on Node.js. To install CSSO, you'll first have to install Node.js and npm. npm is installed as part of the Node.js installation process, so you'll only need to install one package.

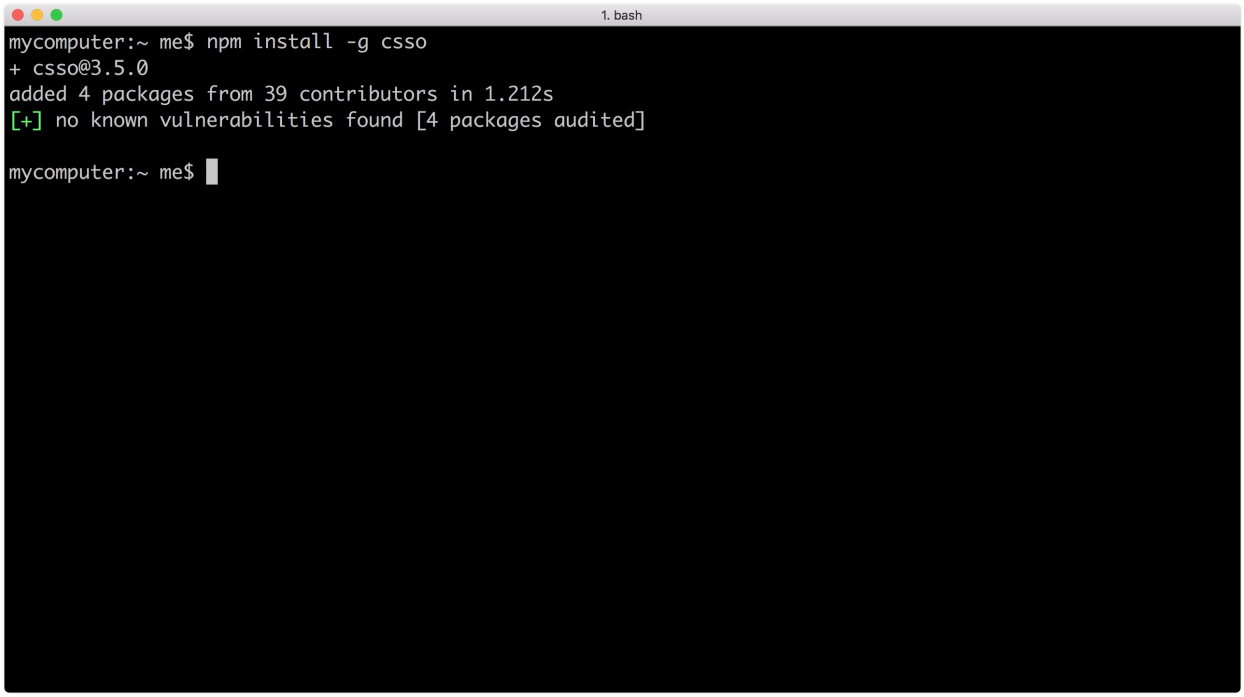
Using CSSO does require you to be comfortable using the command-line interface. Linux and macOS users can use the Terminal application (**Applications > Terminal.app** for macOS). If you're using Windows, utilize the command prompt. Go to the **Start** or **Windows** menu and type `cmd` in the search box.

Installing CSSO

Once you've set up Node.js and npm, you can install CSSO. In the command line, type:

```
npm install -g csso
```

The `-g` flag installs CSSO globally so that we can use it from the command line. npm will print a message to your terminal window when installation is complete:



Now we're ready to minify our CSS.

Minification with CSSO

To minify CSS files, run the `csso` command, passing the name of a file as an argument:

```
csso style.css
```

This will perform basic compression. CSSO strips unneeded whitespace, removes superfluous semicolons, and deletes comments from your CSS input file.

Once complete, CSSO will print the optimized CSS to standard output, meaning the current terminal or command prompt window. In most cases, however, we'll want to save that output to a file. To do that, pass a second argument to `csso`—the name of the minified file. For example, if we wanted to save the minified version of `style.css` as `style.min.css`, we'd use the following:

```
csso style.css style.min.css
```

By default, CSSO will rearrange parts of your CSS. It will, for example, merge

declaration blocks with duplicated selectors and remove some overridden properties. Consider the following CSS:

```
body {
    margin: 20px 30px;
    padding: 100px;
    margin-left: 0px;
}
h1 {
    font: 200 36px / 1.5 sans-serif;
}
h1 {
    color: #ff6600;
}
```

In this snippet, `margin-left` overrides the earlier `margin` declaration. We've also repeated `h1` as a selector for consecutive declaration blocks. After optimization and minification, we end up with this:

```
body{padding:100px;margin:20px 30px 20px 0}h1{font:200 36px/1.5 sans
➥color:#f60}
```

CSSO removed extraneous spaces, line breaks, and semicolons, and shortened `#ff6600` to `#f60`. CSSO also merged the `margin` and `margin-left` properties into one declaration (`margin: 20px 30px 20px 0`) and combined our separate `h1` selector blocks into one.

If you're skeptical of how CSSO will rewrite your CSS, you can disable its restructuring features. Just use the `--restructure-off` or `-off` flags. For example, running `csso style.css style.min.css -off` gives us the following:

```
body{margin:20px 30px;padding:100px;margin-left:0}h1{font:200 36px/1
```

Now our CSS is minified, but not optimized. Disabling restructuring will keep your CSS files from being as small as they could be. Avoid disabling restructuring unless you encounter a problem.

Preprocessors and post-processors (such as Sass, Less and PostCSS) offer minification as part of their toolset. However, using CSSO can shave additional bytes from your file sizes.

Chapter 6: How to Create Printer-friendly Pages with CSS

by Craig Buckler

In this article, we review the art of creating printer-friendly web pages with CSS.

“Who prints web pages?” I hear you cry! Relatively few pages will ever be reproduced on paper. But consider:

- printing travel or concert tickets
- reproducing route directions or timetables
- saving a PDF for offline reading
- accessing information in an area with poor connectivity
- using data in dangerous or dirty conditions — for example, a kitchen or factory
- outputting draft content for pen annotations
- printing web receipts for bookkeeping purposes
- providing documents to those with disabilities who find it difficult to use a screen
- printing a page for your colleague who refuses to use this newfangled t’internet nonsense.

The Web and apps can’t cover all situations, but printing pages can be a frustrating experience:

- text can be too small, too large, or too faint
- columns can be too narrow or overflow page margins
- sections are cropped or disappear entirely
- ink is wasted on unnecessary colored backgrounds and images
- link URLs can’t be seen
- advertisements are shown which could never be clicked!

Developers may advocate web accessibility, yet few remember to make the printed web accessible.

Converting responsive, continuous media to paged paper of any size and orientation can be challenging. However, CSS print control has been possible for many years, and a basic stylesheet can be completed within hours. The following sections describe well-supported and practical options for making your pages printer-friendly.

Print Stylesheets

Print CSS can either be:

1. **Applied in addition to screen styling.** Taking your screen styles as a base, the printer styles override those defaults as necessary.
2. **Applied as separate styles.** The screen and print styles are entirely separate; both start from the browser's default styles.

The choice will depend on your site/app. Personally, I use screen styles as a print base for most websites. However, I have used separate styles for applications with radically different outputs — for example, a conference session booking system which displayed a timetable grid on-screen but a chronological schedule on paper.

A print stylesheet can be added to the HTML `<head>` after the standard stylesheet:

```
<link href="main.css" />
<link media="print" href="print.css" />
```

The `print.css` styles will be applied in *addition* to screen styles when the page is printed.

To separate screen and print media, `main.css` should target the screen only:

```
<link media="screen" href="main.css" />
<link media="print" href="print.css" />
```

Alternatively, print styles can be included within an existing CSS file using `@media` rules. For example:

```
/* main.css */
body {
  margin: 2em;
```

```
  color: #fff;
  background-color: #000;
}

/* override styles when printing */
@media print {
  body {
    margin: 0;
    color: #000;
    background-color: #fff;
  }
}
```

Any number of `@media print` rules can be added, so this may be practical for keeping associated styles together. Screen and print rules can also be separated, but it's a little more cumbersome:

```
/* main.css */

/* on-screen styles */
@media screen {
  body {
    margin: 2em;
    color: #fff;
    background-color: #000;
  }
}

/* print styles */
@media print {
  body {
    margin: 0;
    color: #000;
    background-color: #fff;
  }
}
```

Testing Printer Output

It's not necessary to kill trees and use horrendously expensive ink every time you want to test your print layout! The following options replicate print styles on-screen.

Print Preview

The most reliable option is the print preview option in your browser. This shows how page breaks will be handled using your default paper size.

Alternatively, you may be able to save or preview the page by exporting to a PDF.

Developer Tools

The DevTools can emulate print styles, although page breaks won't be shown.

In Chrome, open the Developer Tools and select **More Tools**, then **Rendering** from the three-dot icon menu. Change the **Emulate CSS Media** to **print** at the bottom of that panel.

In Firefox, open the Developer Toolbar (Shift + F2) and enter `media emulate print`. Print emulation doesn't remain active between page refreshes, but press the up cursor key followed by `enter` to re-execute the command.

Hack Your Media

Presuming you're using a `<link>` tag to load printer CSS, you could temporarily change the `media` attribute to `screen`:

```
<link href="main.css" />
<link media="screen" href="print.css" />
```

Again, this will not reveal page breaks. Don't forget to restore the attribute to `media="print"` once you finish testing.

Remove Unnecessary Sections

Before doing anything else, remove and collapse redundant content with `display: none;`. Typical unnecessary sections on paper could include navigation menus, hero images, headers, footers, forms, sidebars, social media widgets, and advertising blocks (usually anything in an `iframe`):

```
/* print.css */
header, footer, aside, nav, form, iframe, .menu, .hero, .adslot {
  display: none;
}
```

It may be necessary to use `display: none !important;` if CSS or JavaScript functionality is showing/hiding elements on demand. Using `!important` isn't normally recommended but we can justify it here!

Linearize the Layout

It pains me to say this, but *neither Flexbox nor Grid play nicely with printer layouts* in any browser. That may eventually be addressed but, for the moment, set all layout boxes to `display: block;`. It may also be necessary to apply `width: 100%;` on some elements to ensure they span the full page.

Printer Styling

Printer-friendly styling can now be applied. Recommendations:

- Ensure you use dark text on a white background.
- Consider using a serif font, which may be easier to read.
- Adjust the text size to `12pt` or higher.
- Modify paddings and margins where necessary. Standard `cm`, `mm`, or `in` units may be more practical.

Further suggestions include …

Adopt CSS Columns

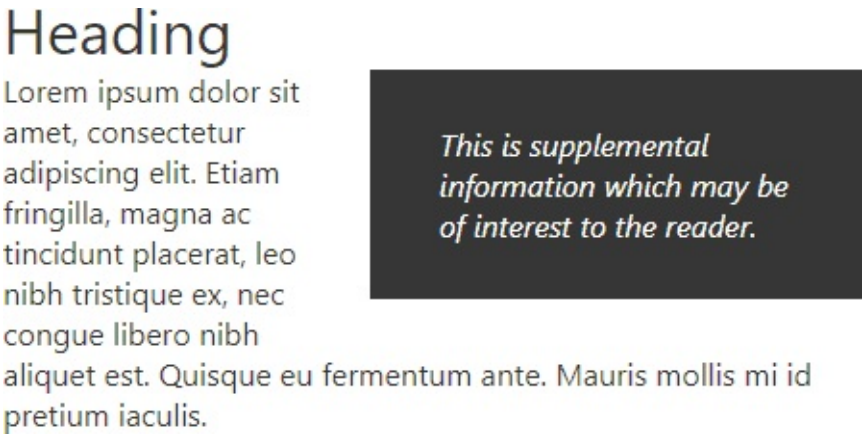
Standard A4 or US letter paper can result in longer and less readable lines of text. Consider using CSS columns in print layouts. For example:

```
/* print.css */
article {
  column-width: 17em;
  column-gap: 3em;
}
```

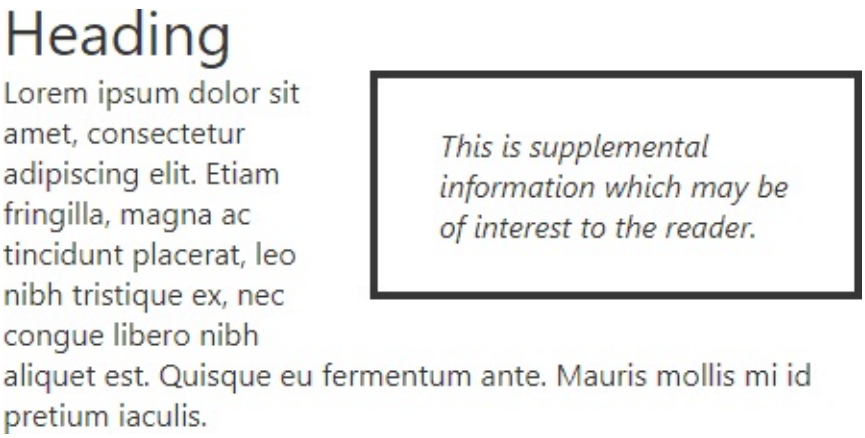
In this example, columns will be created when there's at least `37em` of horizontal space. There's no need to set media queries; additional columns will be added on wider paper.

Use Borders Instead of Background Colors

Your template may have sections or call-out boxes that are denoted by darker or inverse color schemes:



Save ink by representing those elements with a border:

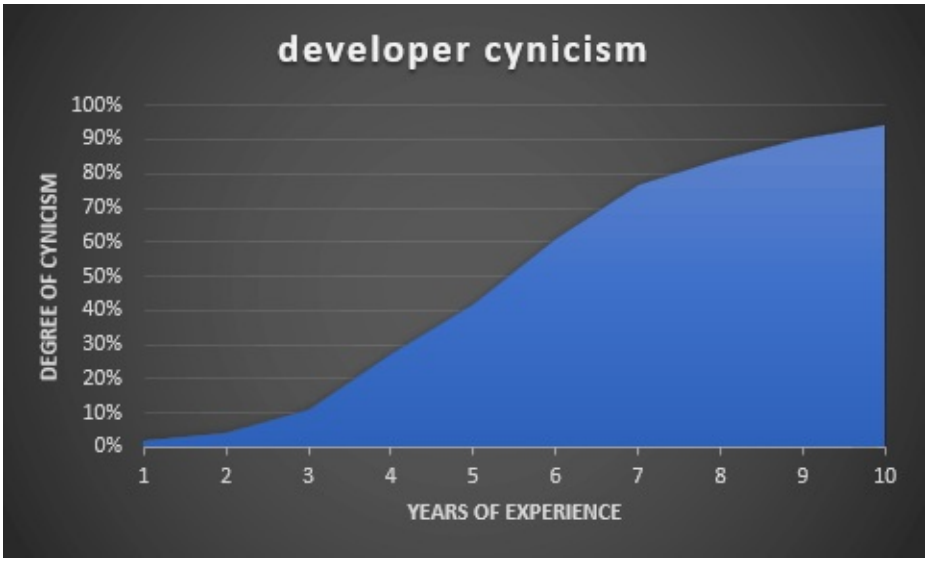


Remove or Invert Images

Users won't want to print decorative or other non-essential images. You could consider a default where all images are hidden unless they have a `print` class:

```
/* print.css */
img, svg {
  display: none;
}

img.print, svg.print {
  display: block;
  max-width: 100%;
}
```

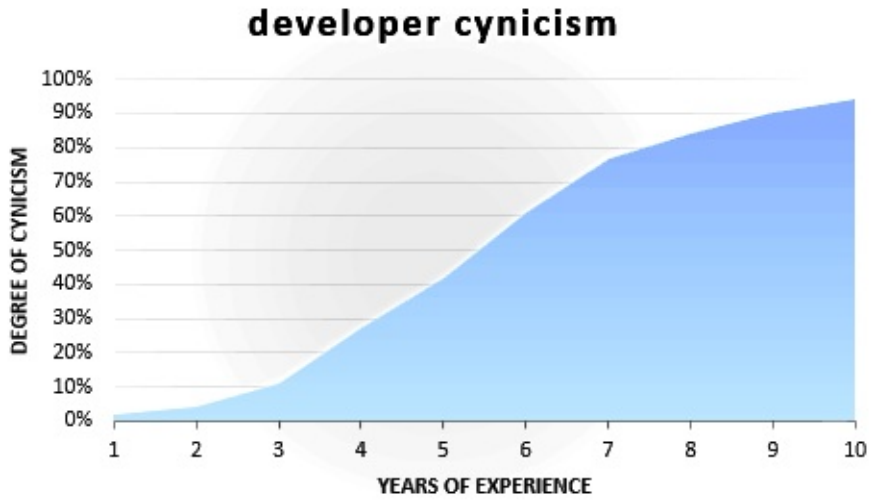
Ideally, printed images should use dark colors on a light background. It may be possible to change HTML-embedded SVG colors in CSS but there will be situations where you have darker bitmaps:



A CSS filter can be used to invert and adjust colors in the print stylesheet. For example:

```
/* print.css */
img.dark {
  filter: invert(100%) hue-rotate(180deg) brightness(120%) contrast(
}
```

The result:



Add Supplementary Content

Printed media often requires additional information which would not be necessary on-screen. The CSS content property can be employed to append text to `::before` and `::after` pseudo-elements. For example, display the URL in brackets immediately after a standard link:

```
/* print.css */
a[href^="http"]::after {
  content: " (" attr(href) ")";
}
```

Append a message:

```
/* print.css */
main::after {
  content: "Copyright site.com";
  display: block;
  text-align: center;
}
```

For more complex situations, consider using a class such as `print-only` on elements which should only be visible when printed. For example:

```
<p class="print-only">Article updated on 11 July 2018. Please see
https://site.com/page for the latest version.</p>
```

The CSS:

```
/* hidden on-screen */
.print-only {
  display: none;
}

@media print {

  /* visible when printed */
  .print-only {
    display: block;
  }
}
```

Headers, Footers

Most browsers display the URL and current date/time on the printed page's header and/or footer.

Page Breaks

The CSS3 properties break-before and break-after allow you to control how page, column, or region breaks behave before and after an element. Support is good, although Firefox only permits the deprecated — but very similar — page-break-before and page-break-after properties.

The following before and after values have good cross-browser support:

- `auto`: the default; a break is permitted but not forced
- `avoid`: avoid a break on the page, column or region
- `avoid-page` (`break-*` only): avoid a page break
- `page` (`break-*` only): force a page break
- `always`: an alias of `page` supported in `page-break-*`
- `left`: force page breaks so the next is a left page
- `right`: force page breaks so the next is a right page.

For example, here's how to force a page break immediately prior to any `<h1>` heading:

```
/* print.css */
h1 {
  page-break-before: always;
  break-before: always;
}
```

The break-inside and older page-break-inside properties specify whether a page break is permitted inside an element. The commonly-supported values:

- `auto`: the default; a break is permitted but not forced
- `avoid`: avoid an inner break where possible
- `avoid-page` (`break-inside` only): avoid an inner page break where possible

To prevent page breaks occurring within a table of data:

```
/* print.css */
table {
```

```
  page-break-inside: avoid;
  break-inside: avoid;
}
```

The widows property specifies the minimum number of lines in a block which must be shown at the *top* of a page. Imagine a block with five lines of text. The browser wants to make a page break after line four so the last line appears at the top of the next page. Setting `widows: 3;` breaks on or before line two so at least three lines carry over to the next page.

The orphans property is similar to `widows`, except it controls the minimum number of lines to show at the *bottom* of a page.

The box-decoration-break property controls element borders across pages. When an element with a border has an inner page break:

- `slice`: the default, splits the layout. The top border is shown on the first page and the bottom border is shown on the second page.
- `clone`: replicates the margin, padding, and border. All four borders are shown on both pages.

Finally, CSS Paged Media (@page) has limited browser support but provides a way to target specific pages so alternative margins or breaks can be defined:

```
/* print.css */

/* target all pages */
@page {
  margin: 2cm;
}

/* target the first page only */
@page :first {
  margin-top: 6cm;
}

/* target left (even-numbered) pages only */
@page :left {
  margin-right: 4cm;
}

/* target right (odd-numbered) pages only */
@page :right {
  margin-left: 4cm;
}
```

The CSS page break properties can be placed within your `screen` or `print` styles because they only affect printing.

Unfortunately, page break control is little more than a suggestion to the browser. There is no guarantee a break will be forced or avoided, because layout is limited to the confines of the paper.

Printing Pains

Control over printing web media will always be limited and results can vary across browsers. That said:

- printer-friendly stylesheets can be retro-fitted to any site
- it’s unlikely to affect or break existing functionality
- the development costs are minimal.

Adding a few page breaks and removing unnecessary sections will delight users and raise your site above competitors. *Add it to your to-do list!*