29 NOVEMBER, 2015

Building and shipping functional CSS

Some months ago, I took on the process of refactoring the distributed CSS library we use to build our products' UI at TrialReach. The result of this refactoring was pretty impressive — for example, the library's file size dropped from an already-lean 68kb down to a very respectable 34kb (file sizes quoted without compression). A big reason for this reduction in size lies in a style of CSS architecture, sometimes referred to as 'atomic' or 'functional' CSS. You might assume that with a 50% reduction in code, our CSS library would become necessarily less flexible, and thus less useful. In fact, the reality has been quite the opposite.

I've been meaning to write for some time now about how and why we've pursued this approach to writing CSS, and how I've been working to ship the same compact, atomic/functional-style CSS library with all of our products to save our users time and cost of bandwidth.

So, here it is.

Atomic? Functional? WTF?

In the past couple years, a particular approach to CSS architectures has evolved in the wake of Nicole Sullivan's Object-Oriented CSS. Alternately termed atomic or functional CSS (by Thierry Koblentz and Jon Gold, respectively), this evolutionary school of thought focuses on the creation and use of primarily single-purpose classes which do one thing extremely well.

The core concepts behind what I'll herein refer to as functional CSS (purely for the sake of convenience) are simple and elegant. I've listed the ones that I feel are key below. None of these are 'official', but rather cohesive statements I've taken from relevant articles.

- **Performance is critical.** Bloated stylesheets cost users time, money, and processing cycles spent computing values. They also quickly push a website's payload towards the initial congestion window (which is a mere 14.6kb, compressed).
- **Clarity trumps cleverness.** Via Brent Jackson: "Simple, obvious styles are quicker to internalize, easier to use, and more widely adopted."
- **Classes should be as reusable as possible.** Via Adam Morse: "For every property you add to a class, the less reusable it becomes." (Also see: Unix philosophy)
- **Classes should be composable and free of side-effects.** Via Jon Gold: "Not only should a class do the same thing every time, but it should never, ever change anything other than what you're targeting.

CSS libraries like Brent Jackson's BassCSS and Adam Morse's Tachyons are the bastions of this approach to CSS, and it's easy to see why. Tachyons weighs in at 62kb, and BassCSS at 14kb — and both of those figures can be reduced by utilising only the specific modules of the library you need and enabling gzip compression. Even at such tiny file sizes, a huge variety of styles are available to be used in both libraries, which means you get huge flexibility for designing in the browser, while delivering a fraction of the file size compared to other popular libraries. Consider Bootstrap, which comes in at 111kb, Foundation at 183kb, or Materialise at 208kb (all quoted sizes are uncompressed).

Because functional CSS places the utmost importance on tiny, reusable, highly recombinable classes, the resultant stylesheets are necessarily free of bloat, which makes for substantially smaller file sizes. This in turns gives us much faster transfer speeds, less time and processing spent on stylesheets, and most importantly, time and money saved on the part of the end user.

Bringing functional CSS to TrialReach

When I took on the task of creating a new distributable CSS library at TrialReach, where the majority of our users are based in the US and are highly skewed to mobile usage (55% of our audience accesses our site from a mobile device or a tablet), there was little question that I would pursue a functional CSS architecture. There were two big reasons for this.

First, given the prevalence of mobile users in the US, where prices for mobile data remain much higher than in Europe, it was critical that our site cost our users as little time and money as possible. (This imperative becomes greater when considering our target audience: people with chronic and terminal illnesses. They're already dealing with enough to not have to worry about us contributing to their monthly mobile bill.) A functional CSS approach seemed to fit this goal nicely.

Second, because this new library was intended to be used across at least three distinct properties (our static website, our clinical trial search interface, and our set of sponsor-facing tools), it was important that our classes be free of content-specific bias and work in a wide variety of contexts — that is, not tied to the design of particular components, modules, or views. The relative neutrality of functional CSS classes again comes out on top here.

This all served to set the stage for how I would proceed with building our new CSS library.

The approach

I basically plagiarised mrmrs and Jackson's libraries, and that was the end of that.

No, but really, I took a huge amount of inspiration from BassCSS and Tachyons, choosing my favourite parts of each as jumping-off points for what was to follow. (One of the best things about designing and developing for the web is that we're all able to riff off each other, hopefully doing a hell of a lot of learning in the process — and, if we're lucky, adding our own little iterations and progressions along the way.)

In terms of technical decisions, I knew would need to create styles that would work well across a range of devices, screen sizes, and browsers (including Internet Explorer, but thankfully only down to IE9). I was also aware that this CSS library would need to be sensible and learnable for our front-end developers (and, eventually, any other designers that joined our team), who would be making use of it as well.

I began by constructing a hypothetical architectural design for our CSS library, which took heavy inspiration from Harry Roberts' ITCSS. With Harry's expertise in creating CSS architectures that scale well within large organisations, I knew this would be a great place to start. This lead me to defining the following categories of styles and matching namespaces for our library (which, again, is nearly verbatim to ITCSS):

1. **Settings:** Code that does not output CSS styles directly. This includes CSS custom properties, custom media queries, etc. These are processed, in our case, by PostCSS and cssnext, which allow us to make use of future CSS syntax. Things like brand colours, modular spacing increments, and breakpoints are defined as variables here.
2. **Elements:** Styles that target bare HTML elements. This mostly consists of normalize.css, and core typographic styles for things like headings, body text, and links which will rarely, if ever, change. No classes are defined here.
3. **Objects:** Content-agnostic classes that serve to define a high-level structure. At present the only objects we use are flexbox-powered media objects and grid objects (containers, rows, and columns; and we only go up to four columns). Object classes all have an `o-` namespace, e.g. `o-container`.
4. **Components:** Content-specific classes that generate distinct UI appearances. This is largely limited to buttons and (lightly styled) custom form inputs. Component classes use a `c-` namespace, e.g. `c-button`.
5. **Utilities:** The bread and butter of functional CSS. These are single-purpose, highly reusable classes that do one thing extremely well. We have utilities for things like margins, padding, text alignment, display properties, positioning properties, and more. In our interpretation of utilities, only properties concerning structure and layout are defined. Utility classes use a `u-` namespace, e.g. `u-inlineBlock`.
6. **Visuals:** Like utilities, but for defining specific visual details like colour, font weight, borders, box shadows, etc. If utilities are for structure, visuals are for surface. Visual classes use a `v-` namespace, e.g. `v-bold`. (In the open source CSS library I'm currently

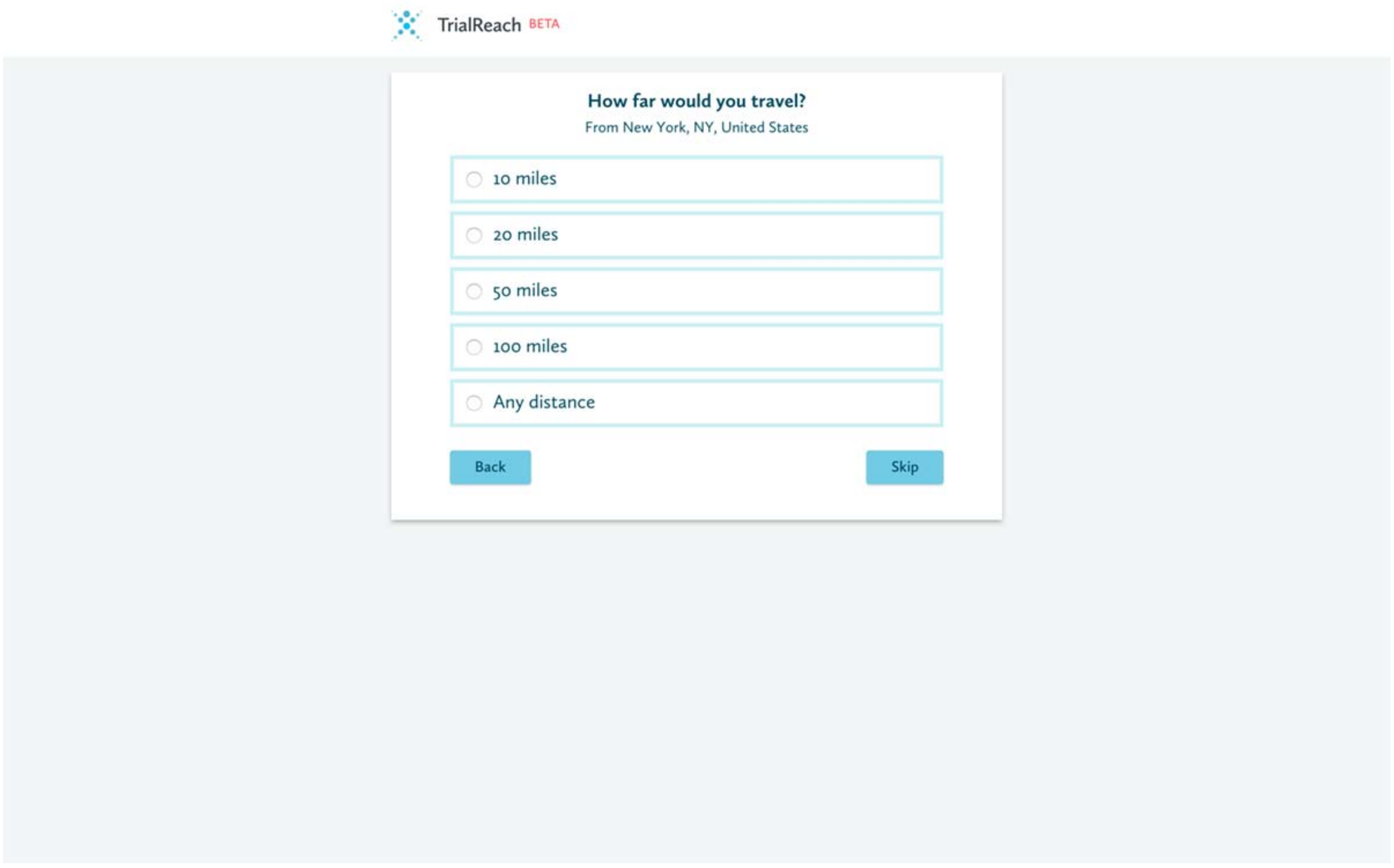
working on — Gemma — I've renamed these to 'Surface' classes, which I feel is a more accurate nomenclature.)

(What are namespaces and why use them in CSS? Read this article by Harry Roberts for the answer.)

With this architectural design in mind, I got down to writing and testing it from the ground up, with the goal of creating a library with which I could refactor the UI of our main product, a question-based search engine for diabetes clinical trials. To walk you through the whole process could be an article unto itself, so to save you some time and me some keystrokes, let's skip ahead and look at some examples of how things turned out.

A few examples in practice

Let's start with the core UI of our question-based search engine. The premise is simple: a card with some text asking a question, and several form inputs for answering, plus buttons to skip the question or return to the previous one.



How far would you travel?
From New York, NY, United States

- 10 miles
- 20 miles
- 50 miles
- 100 miles
- Any distance

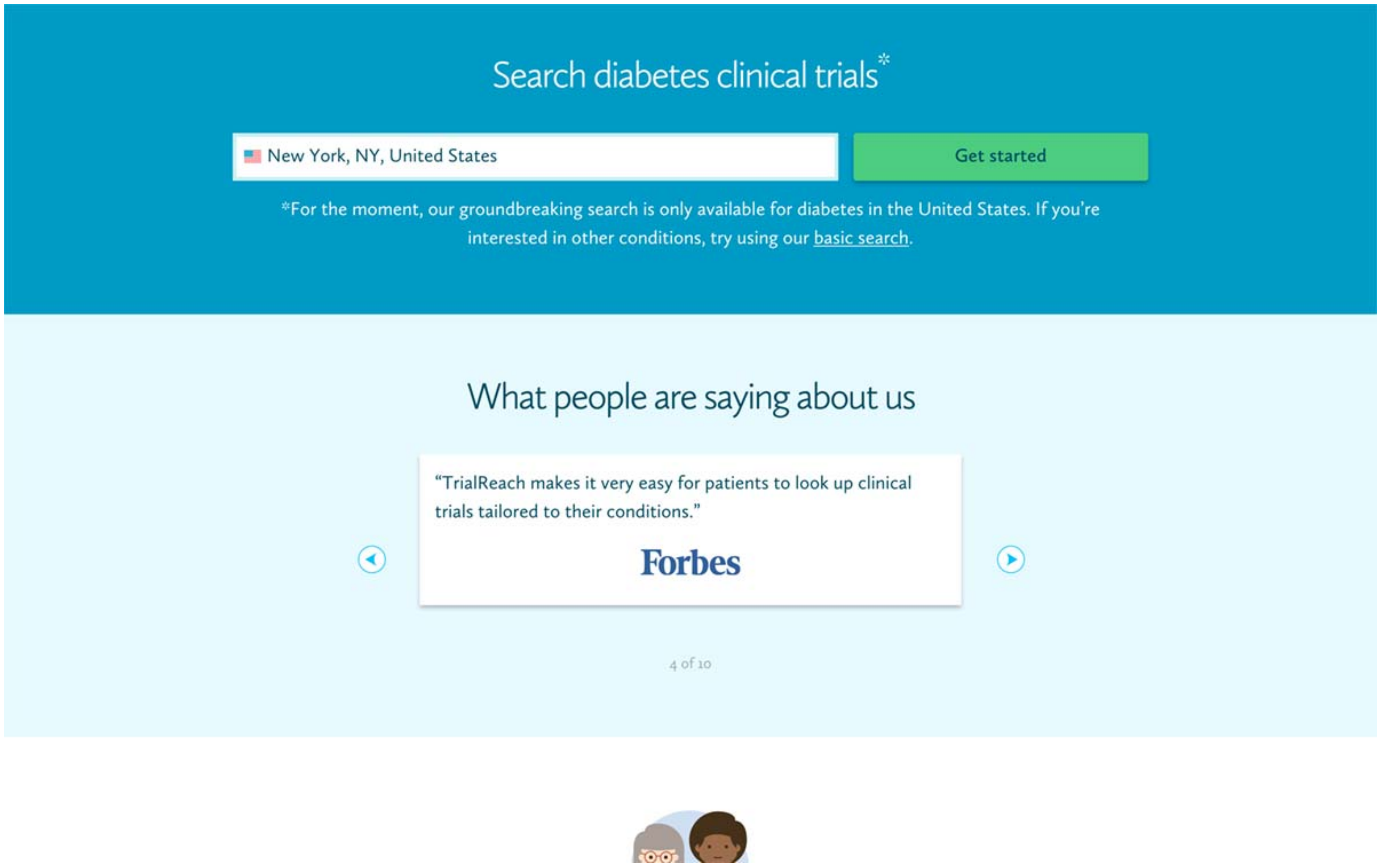
Initially, the markup for that card looked something like this:

```
<div class='questionCard'>
  <!-- Contents of the card -->
</div>
```

...with the matching CSS like this:

```
.questionCard {
  position: relative; /* So buttons can be absolutely positioned inside the card 
  margin-top: $scale1;
  padding: $scale2;
  background-color: #fff;
  box-shadow: $boxShadow-2;
}
```

This was all well and good at the time, but what if we wanted to start using similar (but slightly modified) cards across the rest of the site — say, with less padding? As we do, for example, here:


Search diabetes clinical trials*

New York, NY, United States

Get started

*For the moment, our groundbreaking search is only available for diabetes in the United States. If you're interested in other conditions, try using our basic search.

What people are saying about us

"TrialReach makes it very easy for patients to look up clinical trials tailored to their conditions."

Forbes

4 of 10

Using a traditional approach, we might've constructed a pattern like this:

```
.siteCard {
  padding: $scale1;
  background-color: #fff;
  box-shadow: $boxShadow-2;
}
```

Or, perhaps we might've constructed a basic card component class that would work with modifiers:

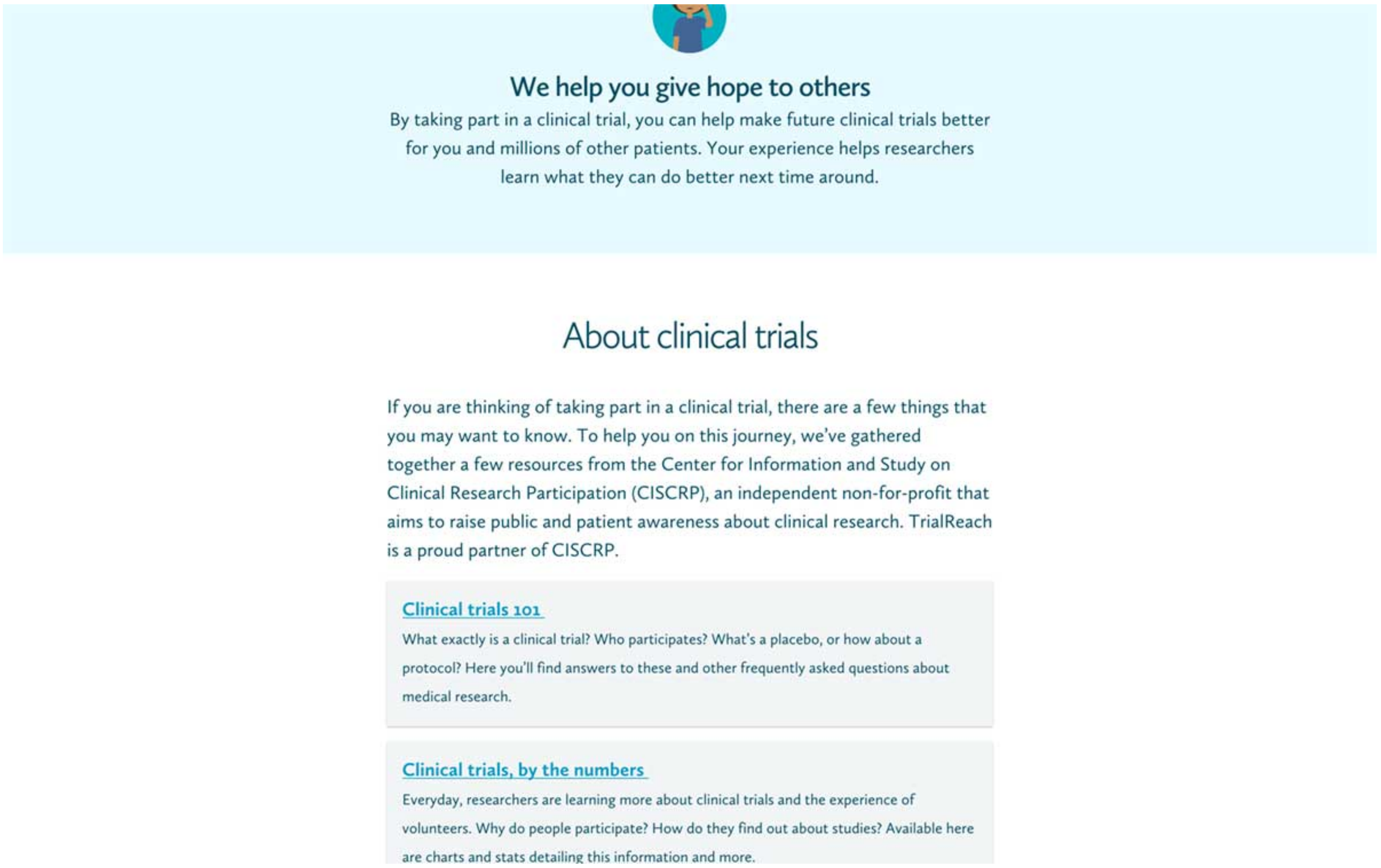
```
.card {
  background-color: #fff;
  box-shadow: $boxShadow-2;
}

.card--question {
  position: relative;
```

```
    margin-top: $scale1;
    padding: $scale2;
  }

  .card--site {
    padding: $scale1;
  }
```

What then would we do if we wanted to use a card on a white background, and with a smaller box shadow to make it stand out just a little less? Like we do here, for example:



We help you give hope to others

By taking part in a clinical trial, you can help make future clinical trials better for you and millions of other patients. Your experience helps researchers learn what they can do better next time around.

About clinical trials

If you are thinking of taking part in a clinical trial, there are a few things that you may want to know. To help you on this journey, we've gathered together a few resources from the Center for Information and Study on Clinical Research Participation (CISCRP), an independent non-for-profit that aims to raise public and patient awareness about clinical research. TrialReach is a proud partner of CISCRP.

Clinical trials 101

What exactly is a clinical trial? Who participates? What's a placebo, or how about a protocol? Here you'll find answers to these and other frequently asked questions about medical research.

Clinical trials, by the numbers

Everyday, researchers are learning more about clinical trials and the experience of volunteers. Why do people participate? How do they find out about studies? Available here are charts and stats detailing this information and more.

Would we construct *another* class, say `.card--site--onWhite--smallShadow`? This is getting out of control, and all because we want to use these cards in a variety of situations. We're placing the burden of context on the CSS, writing more and more styles every time we want to make an alteration to a basic pattern. Not only does this begin to bloat our style sheets, it also means that designers and developers have to be able to see what card classes are available to them, and write new classes if they can't find one already in use that works for them.

Here, then, is the solution I landed on using our new functional CSS library.

That original card used on our question-based search is now marked up like this:

```
<div class='u-relative u-mt1 u-p2 v-bg-white v-bs2'>
  <!-- The contents of the card -->
</div>
```

As you might've guessed, we're using single-purpose classes to resolve this card's layout and appearance. The classes, in order of declaration, provide us with relative positioning, a top margin of 1 scaling unit, padding of 2 scaling units, a white background, and the #2 variant of our box shadow style.

This means that the second example (the white card with smaller padding) is marked up like this:

```
<div class='u-p1 v-bg-white v-bs2'>
  <!-- Card contents -->
</div>
```

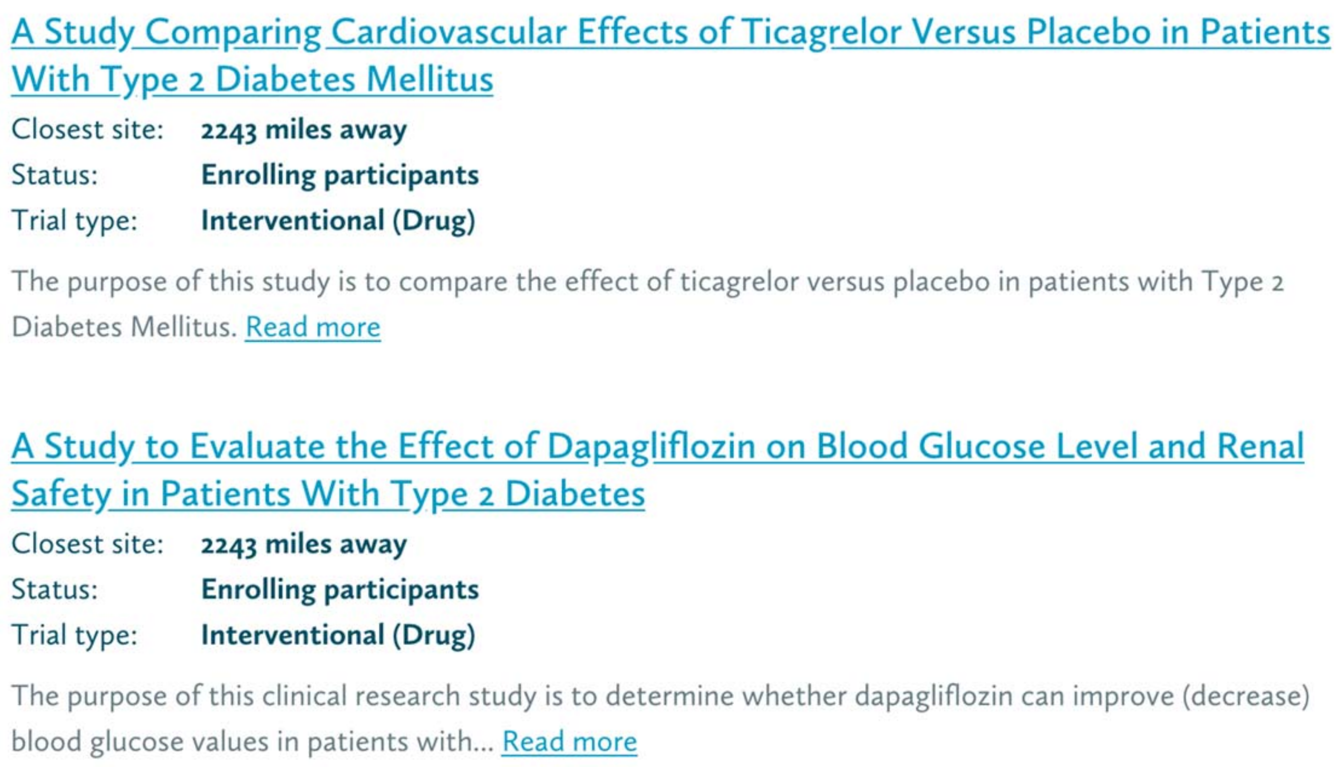
…and the darker card with the smaller shadow is marked up like this:

```
<div class='u-p1 v-bg-gray-300 v-bs1'>
  <!-- Card contents -->
</div>
```

Constructing our UI in this fashion, we've freed up each of the properties required for these cards to be used in any other context, which encourages reuse and removes repetitive declarations and disparate conglomerations of properties in our stylesheets. This also means that **we can consider contextual variances in styling right where the context is created** — within our markup.

This approach extends to almost every piece of our UI that we build — our website, our search product, and our sponsor-facing tool are now all built with functional CSS.

Occasionally, though, we do find cause to create some classes for an application-specific piece of UI. For example, in our search results, we use a text shadow to break up our custom link underlines as they pass through descending characters on result titles.


A Study Comparing Cardiovascular Effects of Ticagrelor Versus Placebo in Patients With Type 2 Diabetes Mellitus

Closest site:	**2243 miles away**
Status:	**Enrolling participants**
Trial type:	**Interventional (Drug)**

The purpose of this study is to compare the effect of ticagrelor versus placebo in patients with Type 2 Diabetes Mellitus. Read more

A Study to Evaluate the Effect of Dapagliflozin on Blood Glucose Level and Renal Safety in Patients With Type 2 Diabetes

Closest site:	**2243 miles away**
Status:	**Enrolling participants**
Trial type:	**Interventional (Drug)**

The purpose of this clinical research study is to determine whether dapagliflozin can improve (decrease) blood glucose values in patients with... Read more

To do this, we create an `a-` or `application`-level class, which applies the relevant text shadow:

```
.a-textShadow-white {
  text-shadow: 0.03em 0 0 #fff, 0.06em 0 0 #fff, 0.09em 0 0 #fff, 0.12em 0 0 #fff
}
```

This class is attached to the otherwise standard markup:

```
<h2 class='v-h5 v-bold u-mt1 u-mb1 u-md-mb0 a-textShadow-white'>
  <a href='{{url}}'>
    {{brief_title}}
  </a>
</h2>
```

Again, in order, those classes are giving us: the text size used by default on `h5` elements, bold font-weight, top and bottom margins, and the text shadow (I'll get to that interesting `u-md-mb0` in a little while). Application-level classes are stored within the app's own codebase, and are not distributed with our CSS library, in order to keep those concerns strictly where they belong.

The outcome

The card demonstration is just a tiny example of how I shifted the bulk of our UI from a component-based architecture to one that is more modular and functional. When my refactoring work on our search application's UI was complete, I opened a pull request on GitHub, and was pleased to see that, in total, my changes introduced 1,141 additions and 1,792 deletions of lines of code — a net **37% reduction in overall code**, which manifested as a **50% cut in CSS file size**.

The improvements weren't just in file size, either. Running both the pre- and post-refactor stylesheets through CSS Stats, I found that the changes had:

- reduced the number of rules in our CSS from 1,299 to 494
- taken our selector count from 1,533 to 731
- cut our declaration count from 2,072 to 909
- kept our number of utilised properties the same, at 108

Other interesting before-and-after stats on our declarations:

- `font-size`: 92 before, 39 after
- `float`: 10 before, 5 after
- `width`: 244 before, 15 after
- `height`: 47 before, 16 after
- `color`: 68 before, 38 after
- `background-color`: 71 before, 42 after

What these stats all point to is a more focused application of CSS, which in turns goes a long way in promoting reusability, scalability, and maintainability — all very important aspects within the front-end code base of a growing startup.

The other big win here is that we now see these benefits across our entire product suite. We are continually cutting out application- and component-specific styles, and embracing creating UI out of our tiny ‘Lego blocks’ of CSS.

This has also been a boon for rapid prototyping, and designing and developing new interfaces — with so many small pieces to play with right within our markup, we’re able to quickly draw up new UIs in the browser, and iterate on them without ever touching a stylesheet.

Gripes and gains — getting comfortable with functional CSS

I’ve listed off a bunch of benefits and given some convincing statistics for how functional CSS has transformed our design and development process for the better, but what about the challenges? What are they, and how can you get around them?

Component-driven design vs. component-driven CSS

Component-driven design is a wonderful methodology for creating consistent, coherent patterns in visual design. In my opinion, though, component-driven CSS is a different beast entirely.

Some of the main values espoused by those in favour of component-driven CSS seem, in my eyes, to have a lot to do with the designers and developers writing the code, and little to do with those who consume the code. I don’t mean this at all maliciously, and I do understand the value of shared understanding being made easy.

For example, David Clark (a seasoned developer whose views I respect) writes in his article on utility classes that:

> By using component classes, you get the communicative value of naming element groups and relationships; you encourage the creation of coherent, relatively self-contained components intended for reuse; and you allow for the flexibility to vary styling, to some degree, without changing markup.

When it comes to code, however, as with design, my priorities are placed first and foremost on the end user. If I can shave even 10 kilobytes of code from a user's mobile data consumption, or several hundred milliseconds off the time it takes them to access the products I work on, I'm going to do it. My experience has shown me that a functional approach to writing CSS makes this (and other benefits) easily achievable. If that means I need to work harder to create a shared understanding for those who design and develop alongside me, that's fine.

So, how then could we reconcile the differences between a component-based design system, and a more atomic approach to UI development?

Make documentation and code reviews a priority

Documentation is always important, but within a more functional-style code base, it moves closer to critical. The approach I've taken is to maintain a component-based style guide within TrialReach, but one which demonstrates how the components are built up from smaller, functional CSS classes. Documentation for a 'secondary button' might, for example, look like this:

```
# Secondary buttons
Secondary buttons have a blue background with white text:

<button class='c-button v-bg-blue-500 v-white'>
  A secondary button
</button>
```

(Aside: The `c-button` class defines things like border-radius, display properties, and box-shadows — which are the same for all buttons — in addition to default padding and colours, the latter two of which

can be reset by our utility and visual classes as above.)

Separately from our style guide, I've written extensive documentation about the architectural design of our CSS so that both current and future team members can quickly get up to speed with how and why we do things the way we do.

Much of the development process for our CSS library involved lots of back-and-forth discussion with our front-end developers. Although they don't often work on our CSS, they do often work *with* it, and as such had some great insights into how we might make the new library work best for everyone. Their approach to writing and architecting code in general has been hugely valuable, and we continue to iterate and improve upon our front-end code base together.

We also embrace a frequent, highly detailed approach to reviewing code, which naturally includes pieces of code that affect design (both within and without the CSS library). This means that before any code is merged, everyone has the opportunity to review changes, and comment on how the author might solve the problem at hand more effectively. For me, this means that I get the benefit of other (and, I'd humbly say, better) developers' eyes on my code, and also that I am able to keep an eye on how our UI codebase evolves over time. Any problems that might arise are spotted quickly ('You could probably build this component out of some padding and colour classes'), and we're able to develop new solutions to problems collaboratively and on a frequent basis ('These properties look like they could be used elsewhere, let's break them into some new utility classes').

Between writing and maintaining solid documentation, and creating an environment where code is reviewed by everyone involved, we're able to create a great sense of shared understanding for designers and developers, while also leaving room for constant collaboration and improvement on the work we've done. This also means that, while our CSS library allows designers and developers to create designs that may not be 'in line' with our style guide (because we do not enforce strict component design through our CSS), we do allow for the easy exploration of new patterns and designs, while our frequent reviews help to ensure we don't go too far off the tracks.

Wrap frequently-used markup in partials or custom components

One of the biggest complaints about functional, multi-class CSS is that it ‘pollutes the markup’. I personally don’t mind having my HTML littered with class names, as long as it’s formatted in a way that maintains clarity and readability — which isn’t too hard. There are, however, some great ways to work around this if it proves to be a real point of contention.

Many of our UI components are defined within partials or custom React components, so that a designer or developer can implement a component without having the write out the requisite classes by hand. A template for a secondary button, for example, might look in part like this:

```
<button class='c-button v-bg-blue-500 v-white' on-click='{{on-click}}'>
  {{label}}
</button>
```

...which means we could implement it like this:

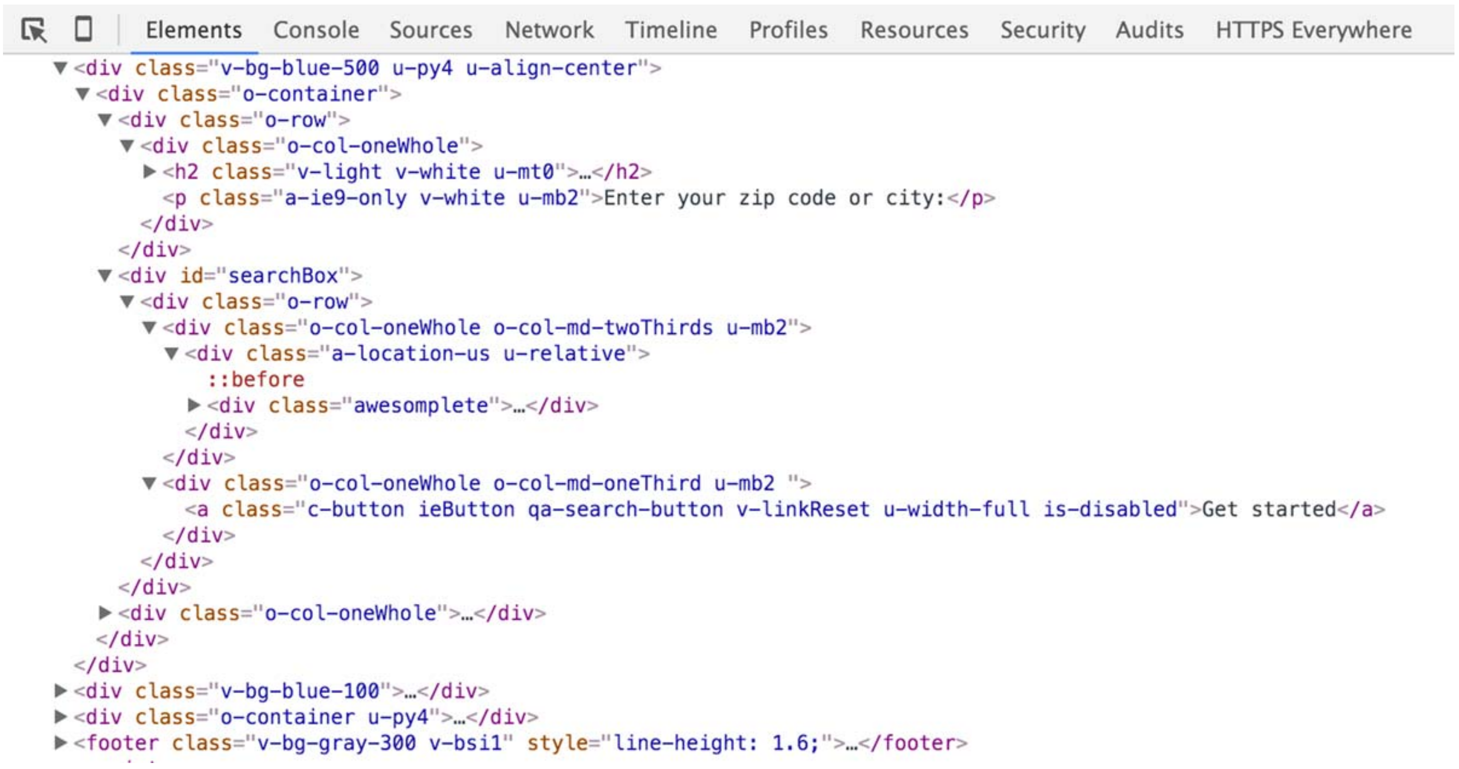
```
<SecondaryButton on-click='myFunctionName' label='My secondary button' />
```

Even if you’re not utilising a JavaScript framework (which could allow for the creation of custom components), a tool like Handlebars can give you some great flexibility for creating reusable chunks of markup, which can help to avoid cluttering your source code with a ton of repetitive class names.

Speaking of repetition, however...

Repetitions: some are good, others less so

It has been noted that removing repetitive declarations from a stylesheet often means shifting that repetition to your HTML. In the case of functional CSS, this is very much the case. An inspection of the TrialReach home page in your dev tools, for example, will quickly reveal a scene that seems to be spawning class names like rabbits. But this isn’t necessarily a bad thing.



```
▼<div class="v-bg-blue-500 u-py4 u-align-center">
  ▼<div class="o-container">
    ▼<div class="o-row">
      ▼<div class="o-col-oneWhole">
        ▶<h2 class="v-light v-white u-mt0">…</h2>
        <p class="a-ie9-only v-white u-mb2">Enter your zip code or city:</p>
      </div>
    </div>
    ▼<div id="searchBox">
      ▼<div class="o-row">
        ▼<div class="o-col-oneWhole o-col-md-twoThirds u-mb2">
          ▼<div class="a-location-us u-relative">
              ::before
            ▶<div class="awesomplete">…</div>
          </div>
        </div>
        ▼<div class="o-col-oneWhole o-col-md-oneThird u-mb2 ">
          <a class="c-button ieButton qa-search-button v-linkReset u-width-full is-disabled">Get started</a>
        </div>
      </div>
    </div>
    ▶<div class="o-col-oneWhole">…</div>
    </div>
  </div>
▶<div class="v-bg-blue-100">…</div>
▶<div class="o-container u-py4">…</div>
▶<footer class="v-bg-gray-300 v-bsi1" style="line-height: 1.6;">…</footer>
```

Because external stylesheets are a render-blocking asset, adding extra weight to the CSS also means adding time spent waiting for a page to be displayed. (A good entry point for understanding how browsers parse and render web pages and assets is this article on the Google Developers site.) Simply put: a webpage cannot begin to be displayed in the browser until both the HTML and the CSS have been parsed (JavaScript has its own implications but is outside the scope of this article).

The quantity of time spent on these tasks will be dependent on several factors — the speed of the network being used to download these assets, the speed of the user's hardware and software in its ability to parse and 'paint' the document, and so forth. Thus, we may be speaking in terms of seconds, or milliseconds — but, either way, a smaller, shorter CSS file will always allow a web page to be displayed faster than than a larger, longer CSS file.

Compression tools such as gzip can greatly reduce the in-transit file size of assets like HTML and CSS (for example, the CSS for the TrialReach website is downloaded at a compressed 8.3kb, but weighs in at 31kb uncompressed). What this means is that compression can save time spent downloading, but the browser will still have to parse all of the 31kb of uncompressed CSS before rendering the TrialReach homepage to the user.

So, what does this mean for our choices in architectural design when choosing where to place our repetition? Nicolas Gallagher, in his excellent article about HTML semantics and front-end architecture, writes:

> In [an] experiment, I removed every class attribute from a 60KB HTML file pulled from a live site (already made up of many reusable components). Doing this reduced the file size to 25KB. When the original and stripped files were gzipped, their sizes were 7.6KB and 6KB respectively – a difference of 1.6KB. The actual file size consequences of liberal class use are rarely going to be worth stressing over.

This is good news, though earlier in the same article, he also claims that '[t]he benefits of more maintainable "CSS" code via pre-processors should trump concerns about the aesthetics or size of the raw and minified output CSS', because HTTP compression can make differences in transmitted file size negligible. To a certain extent I do agree with this point, but taking for example a 50% reduction in CSS file size as we saw in our move to a more functional approach at TrialReach, I feel that file size can also act as a harbinger for complexity in CSS. A 400kb stylesheet is necessarily going to be more complex than a 200kb stylesheet, regardless of whether their compressed file sizes leave little room for quibbling.

With all of this in mind, my bet is firmly placed on reducing CSS file size and complexity by way of writing small, utilitarian classes, and building up complexity within HTML. This means pages can be parsed and rendered faster, by cutting down the time browsers spend on downloading and interpreting stylesheets. I will, however, admit the differences in time here may or may not be of consequence to you — whether individual seconds or milliseconds are important depends on individual contexts. What I can say, though, is that when dealing with an audience short on time and available attention, as ours is at TrialReach, every millisecond has to count.

Functional classes in responsive design

One other oft-cited critique of 'utility classes' is that they pose challenges for creating responsive designs. For example, what if I use `u-align-center` to centre my text, but I later decide I want that same text left-aligned on wider screens? The argument goes that with a component-based system, I could simply declare one alignment property, and change it up with a media query where necessary. For example:

```
<p class='myCoolText'>
  I am centre-aligned by default but left-aligned on wider screens
</p>

.myCoolText {
  text-align: center;
}

@media screen and (min-width: 32em) {
  .myCoolText {
    text-align: left
  }
}
```

I find two issues problematic with this approach. One is that, as someone viewing this bit of markup for the first time, I will have no clue that this component will change its appearance at a certain breakpoint. The second is that all I'm changing is the `text-align` property, and declaring it on this component seems nonsensical — what if I want to apply the same behaviour to something else? I could write a class like `.centerAligned--leftOnBiggerScreens`, but what if I also wanted the ability to left-align something on smaller screens and centre-align it on larger screens? `.leftAligned--centerOnBiggerScreens`? This is getting chaotic. (Also these class names suck. `.myCoolText`? Who wrote this?)

The approach I've landed on is one that I first came across on this article by Harry Roberts, which mentions the concept of 'responsive suffixes'. Harry suggests an approach which would lead to classes like '`o-media@md`, which means *be the media object at this breakpoint*.'

What this lead to in our CSS library was a similar approach, which utilised prefixes instead of suffixes. This brings me back to the `u-md-mb0` class I mentioned ~~two billion words ago~~ earlier in this article. That component was marked up like this:

```
<h2 class='v-h5 v-bold u-mt1 u-mb1 u-md-mb0 a-textShadow-white'>
  <a href='{{url}}'>
    {{brief_title}}
  </a>
</h2>
```

What this means in terms of the component's margins (represented by the `u-m*` classes) is that, by default, this `h2` will have a top margin of 1 scaling unit, and a bottom margin of 1 scaling unit. From our medium (or `md-`) breakpoint and wider, however, it will have no bottom margin. We use breakpoint-prefixed classes like these to allow for resetting certain properties at certain breakpoints — of which we have three: standard (no `min-width`), medium (a little bit of `min-width`, and wider), and large (a bigger amount of `min-width`, and wider). Within our margin utility classes, then, this is defined as:

```
@media (--screen-md) {
  …
  .u-md-mb0 { margin-bottom: 0; }
  .u-md-mb1 { margin-bottom: var(--scale-1); }
  …
}
```

We utilise breakpoint-scoped-and-prefixed classes for properties that we find commonly change depending on the media query at play — `margin`, `padding`, grid column count, and so forth. This allows us to quickly change the layout and appearance of our UI without having to resort to adding component-specific classes to our CSS. So far, this is working out really well for us.

(For what it's worth, I've decided on a slightly different approach to breakpoint-scoped class naming in my own CSS library. Instead of a name like `u-md-mb0`, I instead prepend an underscore-prefixed breakpoint abbreviation — e.g. `_md-u-mb0` — which I feel more clearly draws attention to the fact that the class is scoped to a certain condition.)

I never promised you a rose-coloured panacea

If you’ve made it this far into this article, give yourself a pat on the back. As a treat for spending your time with these 5,500-some words, here’s an open, honest closing remark:

Despite the advantages I’ve tried hard to detail in this article, functional CSS may not work out for you.

As I’ve observed mrmrs quip on several occasions: I only know how to solve the problems that I have solved already. A functional approach to CSS has thus far played out really well for us at TrialReach — we’ve significantly reduced complexity and bloat in our CSS, and find ourselves reusing existing styles far more often than we create new ones — but outside of our contexts and circumstances, and the specific front-end and design-related problems we have to solve, this may not be a repeatable case. Clearly, this approach works well for some other people and organisations (again, observe the substantial praise for, and use of, libraries like BassCSS and Tachyons), but this is not to suggest that these approaches are a one-size-fits-all solution to solve all your CSS-related ills.

Technology in general (and front-end web design and development in particular) move far too fast to suggest that no better solutions to UI performance and scalability will pop up in the distant or even near future (or that a better solution doesn’t already exist), but for the problems I face regularly as a designer and developer on the web, writing and building with tiny, recombinable, highly focused classes has served me incredibly well. I’ve since gone on to use our functional CSS library across the entire suite of products in TrialReach, and it continues to be a positively useful and effective tool. I’ve even started writing my own open-source CSS library with what I’ve learned so far, and recently redesigned and rebuilt my personal website and the new First Things First 2014 website with an in-production version of it. In all instances, I’m seeing huge amounts of flexibility, and teeny tiny resultant file sizes (though I’ll grant that neither of these two websites are particularly overwhelming in size).

I hope this article has served as a useful point of reference for those of you working to design and code on the web, and that if nothing else, the last ~20 minutes you’ve spent reading this have been enjoyable.

Now, then: what’s next?

RSS | colepeters.com

You can go home again.