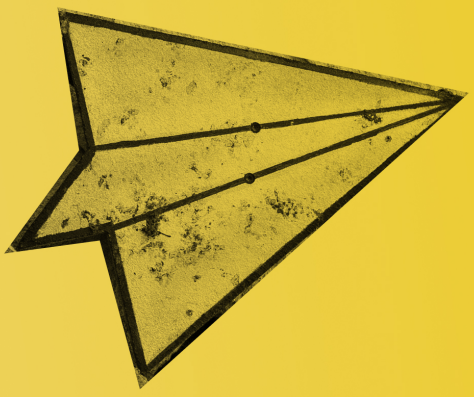
The Better Email

On Design



The Better Email: On Design

A practical introduction to HTML email design and development.

To learn more about email marketing, **visit thebetter.email online**.

Table of Contents

Not sure why this broke *Basic Email Development Tools* into two lines here. Tried everything to fix it but ¯_(ツ)_/¯

A Note on Sanity

If you've spent any time around web designers, you know they're largely a crazy bunch. Dealing with browser inconsistencies and keeping abreast of new developments in technology will do that to people. Email designers are no different. If anything, they're likely crazier. Blame it on Outlook and Lotus Notes.

This is just a note to say it's OK to feel frustrated and crazy when designing email. It's a hard racket—accompanied by little praise from others in the web industry (let alone most stakeholders and upper management).

Don't let it get to you. Email design can be amazingly powerful and you should feel proud to be part of a vibrant and growing community.

On a related note, you shouldn't get down when things break. And you shouldn't strive for pixel perfection in every email client. It's a ghost. It simply doesn't exist.

The beauty of email lies in the fact that most people forget about a campaign immediately after hitting delete. That sucks for posterity, but is fantastic since you can always send another email and fix whatever went wrong in the last campaign. It's a beautifully iterative medium.

A Note on Resources

I've tried to make the code examples in this book as easy to understand as possible. Unfortunately, email code can sometimes still get unwieldy. For more complex examples or longer bits of code, I've included links to examples on CodePen, which will allow you to explore, update, and experiment with the code in real-time in your browser.

If you notice anything wrong in the examples, please don't hesitate to email me at **jason.rodriguez@thebetter.email**.

I also maintain a resources page on my website at **thebetter.email/resources**. It has a ton of information on email marketing, design, and development including links to tutorials, tools for building emails, and a lot of code examples. It's helped thousands of people dig in and learn more about email and I hope it will do the same for you.

Why email?

You've seen the headlines, plastered on every blog in the industry:

> *Email is dead.*

It's in a seemingly constant cycle of being replaced by social networks, native apps, VR, and even your refrigerator. So, why write a book about email?

Because it fucking works, despite what anyone else says. Email is not only alive, it's thriving. And it won't be going anywhere anytime soon.

Consider this:

In 2014, there were 4.1 billion active email accounts. That number is expected to increase to 5.6 billion by 2019. On its own, Gmail accounts for over 1 billion users. **Source**.

Does that sound like the death of a platform to you? It doesn't to me.

Email is a vital part of everyone's digital existence. It's the gateway to the internet. You often hear that mobile is devouring the world and native is the way to go, but nearly every new mobile platform and app requires an

email address to participate. Without email, our digital lives would suffer the real death.

So why is it that so many people talk about the demise of email? And how can so many companies half-ass, or simply ignore, their email strategy?

Because email is hard.

Web and app developers are always surprised when they tackle their first email campaign. Surprised at the outdated approach to coding, the lack of support for standard HTML and CSS, and the sheer number of email clients that all render code in different and frustrating ways.

And this is just the design and development phase. Understanding your audience, envisioning great content, writing killer copy, and worrying about deliverability—there seems to be no end to the unique challenges of email.

But, email doesn't have to be so hard (at least the coding part). The past few years have seen huge advances in coding methods that have helped alleviate some of the challenges inherent in the medium. More importantly, a growing number of email designers are openly sharing their knowledge and building a community around email design. These two factors are helping to solve the problem of email design.

While email clients will always be in a state of flux—and our code right along with them—there is a solid foundation of coding techniques that can be used to make email design easier.

This book is a guide to those techniques.

It distills a lot of the information out there into a practical reference for nearly any design and coding challenge you'll face as an email designer. It's meant for the people in the trenches, digging through code every day. Maybe we'll talk about strategy and content some other time.

Chapter 1

Basic Email Development Tools

Chapter 1

Basic Email Development Tools

Before we dig into any code, though, we need to set up an environment for developing emails.

Fortunately for us, little is needed to get started. Unless you really want to, you won't need to worry about installing complicated applications, command line tools, or a boat load of dependencies.

At the most basic level, all you need to code an email is a text editor and a web browser. The text editor is for writing the code and the browser is for previewing your work.

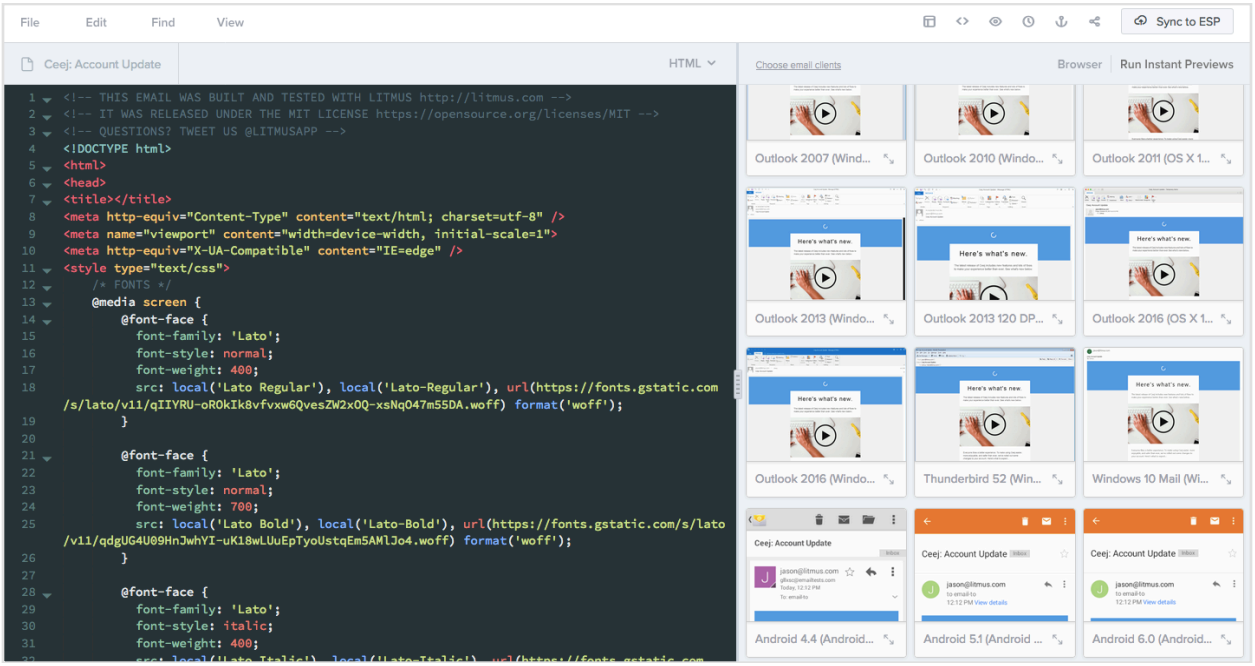
Text Editors

Honestly, it doesn't really matter that much which text editor you choose. I've used several over the years, including **BBedit**, **Sublime Text**, **Coda**, **Dreamweaver**, **Atom**, **Brackets**, and even **Visual Studio**. They all have their charms (and plenty of problems), but they all get the job done.

So long as you can create, edit, and save HTML files, you're good to go. Some people like having extra tools to navigate emails, see the structure of tables, or even help generate code. Try a few on for size and see what fits.

As of writing, I rely on two text editors on a regular basis. The first is **Sublime Text** which is rock-solid, has a great community around it, and a ton of tools to improve workflows.

The second text editor I use—and would have a hard time living without—is **Litmus Builder**. Builder is an online text editor built solely for email development and, as such, has custom tools for making email development as effortless as possible. The killer feature is that it's built on the Litmus platform, so you can instantly preview your email campaign in over 70 (at the time of writing) different email clients.



Honestly, though, it doesn't matter which text editor you choose. Just choose one that works with your budget, feels comfortable, and doesn't stand in your way.

Web Browsers

The same can be said for browsers. I love both Firefox and Chrome's developer tools, but I usually work in Apple's Safari. Really, any modern web browser will work for most things. If you're going to be doing anything fancy with CSS3 or animations (we'll get to those later), you'll likely need to use a Webkit-based browser. Which means Chrome, Chromium, Opera, or Safari.

Again, as long as you can open up an HTML file and resize a window, you're ready to start designing and building emails.

Other Tools

Unless you're exclusively sending plain text email, your design will likely involve graphic elements. These can take the form of photos, illustrations, icons, or (cringe) image-based buttons.

There are a variety of tools out there for all price points. Here are a few of my favorites:

- **Affinity Designer + Photo**: Super powerful tools for editing vector and bitmap work, respectively. I really, really love Designer for vector illustrations, as their tools make more sense than Sketch's. And it's a lot lighter-weight than Adobe Illustrator.
- **Sketch**: Elegant, fast, and continuously improved by the team at Bohemian Coding. Perfect for vector graphics, but useless for editing photographs or creating animated GIFs. Mac only.
- **Photoshop**: The industry stalwart. A bit long in the tooth, but can't be beat for sheer power and number of tools. Also necessary if you want to create, edit, and optimize animated GIFs. Cross platform.
- **Pixelmator**: Surprisingly powerful for both bitmap and vector work, but lacks any animation capabilities. Much more affordable than Photoshop and is incredibly useful for quick edits. Mac only.
- **Paint.NET**: When I was on Windows, I used to love me some Paint.NET. Very capable and free, absolutely recommend it to anyone on a PC with a small budget.

No matter which program you use to create images, one of the most important things to do is to compress those images before uploading them to a server and linking to them in an email campaign. As subscribers become more mobile, bandwidth constraints play a larger role in email design. Image compression is a vital part of keeping email designs quick to load and subscribers happy.

My favorite tools for compressing images are:

- **ImageOptim**: The best lossless image compressor for Mac. Plus, they now have a **Sketch plugin** that automatically compresses your images on export. Pretty nifty.
- **JPEGmini**: The cross-platform option, this one is also great (but costs money).
- **TinyPNG**: A great option if you're only using PNG files. It also has a handy Photoshop plugin.

This book won't go into how to use these specific tools. Just be aware that, if you're working as a professional email designer, you need to be comfortable with whatever tools you are using. Study them, practice using them, and get fast with them. Your clients (or boss) will thank you.

Chapter 2

The Building Blocks of Email

Chapter 2

The Building Blocks of Email

Email is a weird world, one that often relies on outdated practices to accomplish seemingly simple tasks. Don't let that put you off, though—things are slowly getting better and there's a surprising amount that you can accomplish in HTML email campaigns.

Just be warned: if you're coming into email design with prior experience as a web designer, you're going to have to push some of your instincts aside and relearn how to do even the most basic stuff. Which brings us to one of the most important points in this book…

> *Email is not the web.*

Most web designers complain about the half dozen or so browsers that they need to support. Even with the quirks inherent in some of the browsers (looking at you, Internet Explorer), coding for the web and getting consistent results across browsers isn't terribly difficult anymore.

Contrast that with email clients. There are dozens of popular email clients that need to be taken into account. Starting up a new Litmus test, at the time of this writing, allows you to test in over 70 different email clients.

And, unlike the web, there are no agreed upon standards that email client vendors need to support. Sure, you will be writing HTML and CSS, but every single email client supports a limited subset of both. Rarely do any email clients support the same HTML elements or CSS properties. That makes things… complicated, to say the least.

Web ≠ Email

As a web designer, your first thought might be to structure and code an email in the same manner as a web page. Unless your skills plateaued sometime around 2002, that means marking your content up with semantic elements and using external stylesheets to properly style everything.

Unfortunately, few email clients properly support semantic elements (at least for structuring content). While we can (and will) use semantic elements for text, HTML5 sectioning elements like **`header`**, **`article`**, **`aside`**, and **`footer`** have very limited support, and will be avoided for the time being.

To successfully build HTML email campaigns, you will be trading in what you know of web design for what everyone else knew of web design before the era of web standards. Although we'll look at a few approaches to creating emails using more modern markup and methods, the main

techniques used in this book will rely on HTML tables for building out the structure of email campaigns.

At a very high level, here's what you can expect:

On the web	In email
External CSS	Embedded and inline CSS
Floats	The align attribute and table cells
Shorthand CSS properties	Longhand CSS properties
Box model support	Table layouts
JavaScript	CSS animations
Interactive forms	Checkbox hacks
Web standards	Chaos and fun

You may be used to marking up tabular data on the web, but email design is its own art form. It requires dropping some table elements, structuring your document and tables in a very specific way, and making extensive use of inline-styles and deprecated HTML attributes to make things work.

Don't get too down about the state of email, though. We've seen some major improvements from email clients recently (looking at you, Gmail) that will allow us to clean up our markup and use some slightly more modern methods in our campaigns. And, while we won't rely on semantic

sectioning elements, we will absolutely use semantic text elements to make our emails more accessible, as well as the occasional embedded CSS and div element for certain techniques.

A Basic Document Structure

Before we add any content to our emails or start styling that content, we need to have a solid base on which to build. Let's look at how a basic HTML document for email development should be structured.

At it's heart, every email campaign is an HTML file. HTML—which stands for Hypertext Markup Language—is the de facto language of the web. It allows us to write and structure content in a way that both humans and computers can easily read. Each HTML file ends with the **`.html`** file extension. So, if you're working on a monthly newsletter campaign, you might have a file called **`november-2017-newsletter.html`**.

HTML files require a few pieces of code to be included in order to work properly. The following section goes over those pieces and provides a good base document on which to build.

HTML doctype

There's been a surprising amount written about which doctype to use for email. Unsurprisingly, there's no consensus on which is the best. For a long time, most people either a) used no doctype at all or b) rallied

around the XHTML 1.0 Transitional doctype. Today, I heartily recommend transitioning to using the HTML5 doctype.

```
<!DOCTYPE html>
```

Not only is it more succinct, it ensures that content is rendered just as well as when using the older XHTML 1.0 Transitional doctype, if not better. And, moving forward, if email clients add better support for HTML5 and CSS3, you'll be covered.

All that being said, don't live or die by the doctype. It's largely included to ensure a few clients render a very small number of elements properly and that you have something to validate your markup against.

Don't worry too much about validating an email campaign. While running your code through the **W3C Markup Validation Service** can be useful to identify potential problems with an email, HTML email design is such a hack-riddled art that it's rare when an email template fully validates. Use a validator to uncover things like missing closing elements or a stray quote mark, but don't slam your head against your desk when a well-tested email campaign won't validate. The ultimate judge of an email design is testing it across email clients, either in **Litmus** or on actual devices.

You'll also want to include the **`lang`** attribute in your root HTML element. This is useful for assistive technologies and helping to ensure certain characters are properly rendered (some glyphs, quotation marks,

hyphenation, spaces, etc.). Since I send to a mostly English-speaking audience, I always use the following:

```
<html lang="en">
```

The head Section

The **head** of your document is where you can set up document-specific information for an email. It is also where you include some CSS, like reset styles and styles that progressively enhance any essential, inlined styles.

I typically include a **`title`** element and populate it with either the sender name or some descriptive text about the campaign's content. While email clients don't display title information, it can be useful when campaigns are viewed as a web page (which happens surprisingly often). Depending on your email service provider, you can use their templating language to automatically populate the title with a subject line, sender name, or something pulled from your database.

```
<title></title>
```

Next, we'll include three meta tags. The first sets the document's charset to ensure that symbols like HTML character entities are properly displayed. The second sets the document's viewport sizing, which is essential when building responsive emails (which you should be doing, and which is what is taught in this book). The third is specific to Microsoft devices and clients to keep them playing nicely with our code.

```
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-
scale=1">
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

This second element is probably the most important. It tells the device that your email should adapt to the width of the device itself, allowing it to flow when the device's size changes.

The last part of the head section is a style block. This is where any non-essential CSS goes. Most of your CSS will be inlined on specific HTML elements themselves, but the style block in the head will contain CSS resets you need to include, as well as CSS that progressively enhances content or is meant for mobile email clients.

```
<style type="text/css">
/* CLIENT-SPECIFIC STYLES */
body, table, td, a { -webkit-text-size-adjust: 100%; -ms-text-
size-adjust: 100%; }
table, td { mso-table-lspace: 0pt; mso-table-rspace: 0pt; }
img { -ms-interpolation-mode: bicubic; }

/* RESET STYLES */
img { border: 0; height: auto; line-height: 100%; outline: none;
text-decoration: none; }
table { border-collapse: collapse !important; }
body { height: 100% !important; margin: 0 !important; padding: 0
!important; width: 100% !important; }
</style>
```

The head is naturally followed by the body and closing html elements. When put together, a solid base document for any email campaign looks like the following:

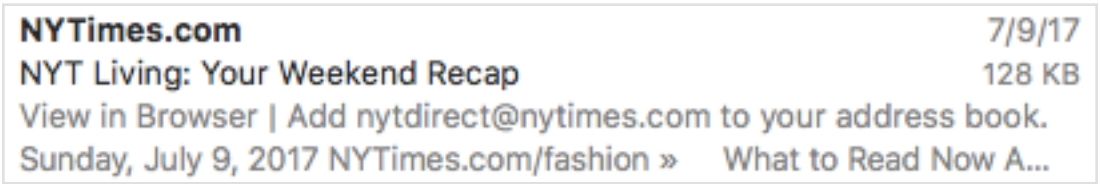
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
        <meta name="viewport" content="width=device-width,
initial-scale=1">
        <meta http-equiv="X-UA-Compatible" content="IE=edge" />
        <style type="text/css">
            /* CLIENT-SPECIFIC STYLES */
            body, table, td, a { -webkit-text-size-adjust: 100%;
-ms-text-size-adjust: 100%; }
            table, td { mso-table-lspace: 0pt; mso-table-rspace:
0pt; }
            img { -ms-interpolation-mode: bicubic; }

            /* RESET STYLES */
            img { border: 0; height: auto; line-height: 100%;
outline: none; text-decoration: none; }
            table { border-collapse: collapse !important; }
            body { height: 100% !important; margin: 0 !
important; padding: 0 !important; width: 100% !important; }
        </style>
    </head>
    <body style="margin: 0 !important; padding: 0 !important;">
    </body>
</html>
```

View on CodePen

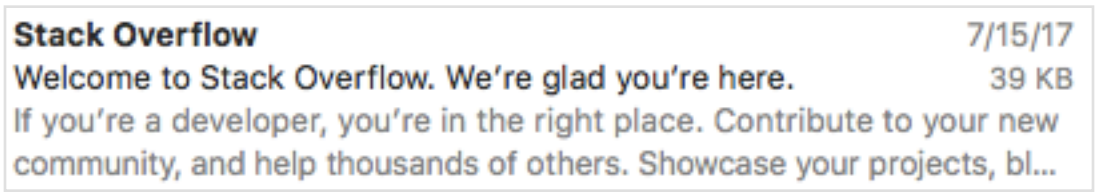
You'll notice that I've included both **`margin`** and **`padding`** styles inlined on the **`body`** tag. These are just to ensure that no email clients add extra white space around our campaigns.

Preview Text

One thing you'll nearly always want to add to your email is preview text. Preview text is the bit of text you see below the sender name and subject line in the inbox of an email client. Without coding anything, the first bit of copy in an email is automatically pulled into the inbox in most email clients.



However, it's almost always better to control that copy and make it work well with the subject line.

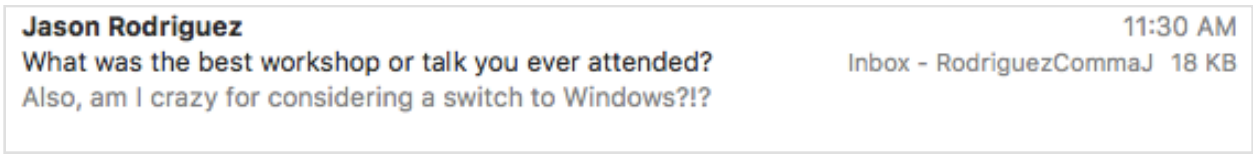


To add preview text, which is then hidden in the actual email view, you can use the following code:

```
<div style="display: none; max-height: 0; overflow: hidden;">
  Preview text message goes here
</div>
```

By using **`display: none;`**, **`max-height: 0;`**, and **`overflow: hidden;`**, whatever text you use will be displayed in most email clients. Different email clients display different character counts for the preview text, so it's worth testing out copy to see what works. One really cool trick, though, is to use really short preview text to draw attention to your campaign.

Since most emails aren't currently using this trick, they pull in a lot of text. Adding some space behind a short message is a good way to visually call attention to your campaign.



The way to do this is by taking advantage of the fact that combining two HTML character entities, you can get a character-wide space. By combining dozens of those HTML character entities, you can get dozens of character-wide spaces, effectively blocking any body copy in your email from displaying behind your preview text.

The two HTML character entities used are the non-breaking space (` `) and the zero-width non-joiner (`‌`). You wrap that in the same code as the preview text to ensure that it's hidden from view in the email.

```
<div style="display: none; max-height: 0px; overflow: hidden;">
 &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;&nbs
p;&zwnj; &zwnj; &zwnj; &zwnj; &zwnj; &z
wnj; &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;
 &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;&nbs
p;&zwnj; &zwnj; &zwnj; &zwnj; &zwnj; &z
wnj; &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;
 &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;&nbs
p;&zwnj; &zwnj; &zwnj; &zwnj; &zwnj; &z
wnj; &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;
 &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;&nbs
p;&zwnj; &zwnj; &zwnj; &zwnj; &zwnj; &z
wnj; &zwnj; &zwnj; &zwnj; &zwnj; &zwnj;
 &zwnj; &zwnj; &zwnj;
</div>
```

View on CodePen

It's a big chunk of code, but definitely worth it.

Revisiting Tables

When was the last time you marked up content using HTML tables? Unless you're building dashboards or work in the scientific community

presenting tons of data, it has likely been quite a while since you used HTML tables for anything.

That's about to change. As an email designer, you'll be drowning in tables. You won't be able to wash the stench of tables off for years to come. Scared yet? Don't be—tables are easy.
You may have seen the following table structure before:

```
<table>
      <thead>
            <tr>
                <th>Header A</th>
                <th>Header B</th>
            </tr>
      </thead>
      <tfoot>
            <tr>
                <td>Footer A</td>
                <td>Footer B</td>
            </tr>
      </tfoot>
      <tbody>
            <tr>
                <td>Body A</td>
                <td>Body B</td>
            </tr>
      </tbody>
</table>
```

In this example, you see most HTML table elements represented, including the **`table`**, **`thead`**, **`tr`**, **`th`**, **`tfoot`**, **`tbody`**, and **`td`** elements. When working on the web, it's necessary to use these elements to give

context to any tabular data that is being presented. However, in the email world, email clients don't really care about context or semantics. Which actually works out nicely—we effectively only need to worry about three elements for marking up any and all content: the **`table`**, **`tr`**, and **`td`** elements. There are some techniques that rely in the **`th`** element, but those are beyond the scope of this book. **Check out this article** for more information.

Table Elements

It's essentially pointless to use any other table-related elements in an email design. The only thing you'll be adding is unnecessary markup, making templates harder to maintain. So don't do it. Only use table, tr, and td in an email campaign. So, at a very basic level, an email campaign would be structured in a table that looks like this:

```
<table>
    <tr>
        <td>Content</td>
    </tr>
</table>
```

That's it, we're done. That's all you need to know to build email campaigns.

Just kidding, there's a bit more to it than that. Like…

Table Attributes

According to the **Mozilla Developer Network**, there are 9 attributes which can be applied to tables:

- **`align`**
- **`bgcolor`**
- **`border`**
- **`cellpadding`**
- **`cellspacing`**
- **`frame`**
- **`rules`**
- **`summary`**
- **`Width`**

Of these, we will be using everything except **`frame`**, **`rules`**, and **`summary`**. You can safely ignore all three. We'll also be adding one more to make our emails more accessible. We'll get to that in a minute.

The **`align`**, **`border`**, **`cellpadding`**, **`cellspacing`**, and **`width`** attributes are essential structural components for email. And the **`bgcolor`** attribute is great when it comes to styling email layouts.

These attributes are typically applied to two elements: both the **`table`** and the **`td`**. Generally speaking, you will apply **`border`**, **`cellpadding`**, **`cellspacing`**, and **`width`** to every table in your design. Sometimes you will need to align a table, too. For table cells, you will often need to include **`align`** attributes as well as the occasional **`width`** and **`bgcolor`** if

you are attempting to constrain a cell's width or apply a background color in your design.

If that sounds confusing, this example should make it easier to understand:

```
<table border="0" cellpadding="0" cellspacing="0" width="100%">
    <tr>
        <td>
            <table border="0" cellpadding="0" cellspacing="0"
width="600">
                <tr>
                    <td>
                        <table border="0" cellpadding="0"
cellspacing="0" width="100%">
                            <tr>
                                <td>
                                    <!-- CONTENT GOES HERE -->
                                </td>
                            </tr>
                        </table>
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
```

You can see in the example above that we have a container table with nested tables running two levels deep. All three tables have the border, cellpadding, cellspacing, and width attributes defined. While the **`align`**

and **`bgcolor`** attributes are not set, we will see how those come into play later on.

It's important to note that I have used both fluid and fixed widths for the tables. The outer and innermost tables are both fluid, as denoted by the **`width="100%"`** attribute. The middle table, however, is fixed to 600 pixels wide with **`width="600"`**. This is because not all email clients understand percentage-based widths. Therefore, that table acts as a container table for the fluid content within. Later on, when talking about responsive emails, we will see how this container table comes into play with the help of CSS selectors and the media query. We'll also look at different approaches to laying out emails that use fluid-by-default techniques and even (nearly) table-free techniques.

Accessibility in Tables

An important aspect of email design (and any type of design) is accessibility. Accessibility is the idea that things we put online (including email) should be accessible to the widest range of users possible. We never want to exclude users from understanding and using our content. When it boils down to it, it's discrimination, whether or not it's intentional.

There are a few things we'll cover throughout this book to make sure our campaigns are accessible as possible. The first has to do with using tables to structure our campaigns.

As mentioned before, tables are traditionally used to display tabular data. Think of a spreadsheet. That spreadsheet has a bunch of information in it, and column and row headings to provide some context for that information. By default, someone with a disability that requires the use of a screen reader will have a table read out to them at length. The screen reader software will go through the table and every column, row, and table cell, reading the contents of each element out loud.

For a great illustration of this, check out **this fantastic blog post** from Mark Robbins at **Rebel**. You can hear exactly how a screen reader presents unoptimized tables to the user.

This is less-than-ideal for email, because the information in an email is usually not tabular data. We want to be able to prevent screen reader software from reading every table element out loud and allow them to skip to the content instead. To do this, we can apply the **`role`** attribute with a value of **`presentation`** to every table in our email.

```
<table border="0" cellpadding="0" cellspacing="0" width="100%"
role="presentation">
```

It's important to note that it does need to be applied to every table. Unfortunately, that behavior isn't inherited by child elements of the main container table. With that applied, screen reader software will skip to the content of an email. It's the single easiest way to make your campaigns more accessible for your subscribers, and should be baked into every campaign you create moving forward.

Inline Styles

Before moving on with using tables to structure our layouts, we need to talk about where and how to apply styles in HTML email.

On the web, we have three methods for including CSS in a document.

1. External stylesheets: Allow you to keep your styles in a separate file, making maintenance and updates quick and easy.
2. Embedded style blocks: Styles applied within a **`style`** section, typically in the **head** of the HTML document. They are specific to the HTML document, but still separate styles and content in a respectable manner.
3. Inline styles: Styles applied directly on an HTML element, making styles much harder to maintain.

Most modern websites opt for using external stylesheets, since they make maintenance far easier than the other two options. Separating content from presentation is one of the foundational principles of the web standards movement and should be adhered to whenever possible.

Unfortunately, email client vendors have no concept of web standards. Some email clients will block an email from linking to and downloading an external stylesheet. Some email clients will strip out the **head** section of an HTML document, meaning embedded style blocks living up there

will also be killed. Therefore, if we want to reliably style our emails, we are left with the least desirable option: inline styles.

As mentioned above, inline styles are applied directly to an HTML element. For example, let's say we wanted to change the color of some text. We could do that by adding an inline style to a **span** tag in a larger block of copy.

```
This is some copy. <span style="color: red;">This is red.</span>
And this is the default color.
```

That is an inline style. While not ideal (it generally makes troubleshooting and maintenance more time-consuming), it is the most reliable way to add styles to a campaign so that those styles display across the widest range of email clients.

We will still use embedded styles for certain tasks, but get used to using inline styles, too. Nearly all of our default styling will be applied inline directly to HTML elements.

Google's Gmail is traditionally the main culprit when it comes to removing embedded styles. However, an update to Gmail that launched in the fall of 2016 remedied this behavior. While you can now safely use embedded CSS to target elements in an email instead of relying on inline styles, there are still a handful of email clients that don't play well with embedded styles. For this reason, we'll still use inline styles for critical styling, with embedded styles providing mostly reset and progressive

enhancement functionality. It's the approach you're most likely to encounter out in the wild and is still the most bulletproof method of styling email content.

Structuring Table Layouts

So, getting back to tables.

Different types of email call for different layouts. You may opt for a nice, minimal, single-column layout or a 2x3 product grid. You could have article sections mixed in with image-heavy features. Regardless of the content of your campaign, you're going to have to use tables to structure everything.

This is accomplished with a few things: inline styles, the **`align`** attribute, nested tables, and stacked tables. Let's take a look at each in turn to see how they allow us to structure email layouts.

Inline Styles for Table Structures

If you've been following along, you know we'll be applying the majority of our styles inline on HTML elements. But, what CSS properties will we actually be using? When it comes to using tables for structuring our content, the single most useful CSS property is **`padding`**.

Combined with specified widths on tables or table cells and the **`align`** attribute, the **`padding`** property does 90% of the work in any email design. The **`padding`** property allows us to—you guessed it—add padding between sections of our email. Whitespace is a vital component in any design and **`padding`** is how we can best add that whitespace.

But, why not use the **`cellpadding`** or **`cellspacing`** attributes? That's a valid question. They are inherent to tables, but have two major drawbacks:

1. You can't specify different values for different sides of an element.
2. If you're using nested tables or specifying a lot of widths, keeping track of the math can get complicated.

That first point is the crux for me. When designing an email, I want full control over the spacing between elements on all four sides: the top, right, bottom, and left. Neither **`cellpadding`** or **`cellspacing`** allow you that level of granularity. The **`padding`** property, however, works like a charm. When I'm specifying **`cellpadding`** and **`cellspacing`** in this book, you'll always see that the value is set to zero. This is simply to override any padding or spacing an email client automatically adds to a table or cells. It's a reset that allows us to use padding to add space instead.

There are some designers who will argue against relying on padding since it doesn't have support everywhere. While that is true, the email clients that lack support also lack large user bases. The main suspects are

Lotus Notes versions 6 and 7. Both hate padding with a passion, which is OK, because everyone hates Lotus Notes more. Outlook 2007+ has some issues with padding, but they are minimal and there are hacks to work around them.

Padding can be applied in one of two ways: by explicitly declaring the padding for each individual side of an element, or by using the shorthand property.

Explicit:

```
<td style="padding-top: 10px; padding-right: 20px; padding-
bottom: 10px; padding-left: 20px;"></td>
```

Shorthand:

```
<td style="padding: 10px 20px 10px 20px;"></td>
```

Generally speaking, some email clients don't handle shorthand declarations well. However, where padding is supported, I've never seen issues using the shorthand declaration. It's a great way to keep your code concise. Just be sure to declare each side within the shorthand, as some clients can choke on the truncated version of a shorthand property, which explicitly declares two values and relies on the rendering engine to infer the others. Using shorthand, each of the four values correspond to a particular side, in this order:

```
padding: top right bottom left;
```

On the web, you can safely write the first two values and let the browser fill in the blanks. So this:

```
padding: 10px 20px;
```

Becomes this:

```
padding: 10px 20px 10px 20px;
```

Just don't do that in email, and you'll be fine. Or at least test your emails before sending if you do use that method.

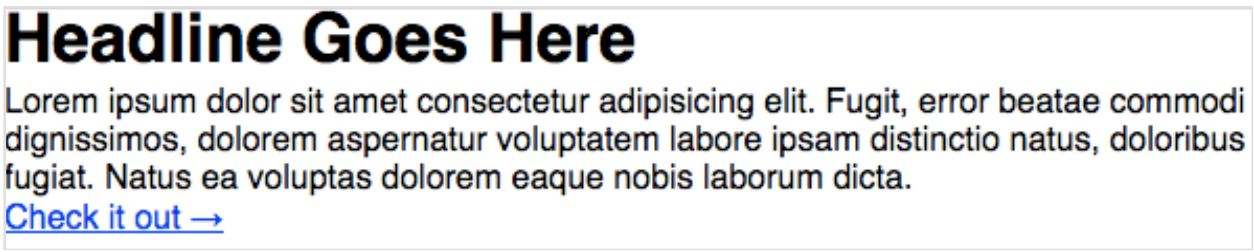
Getting back to using padding and tables. Let's say we have an email layout that has a featured article section, which consists of a headline, some text, and a link. Without any real structure applied, that markup might look like the following:

```
<table border="0" cellpadding="0" cellspacing="0" width="100%"
role="presentation">
    <tr>
        <td>
            <table border="0" cellpadding="0" cellspacing="0"
width="600" role="presentation">
              <!-- FEATURED ARTICLE -->
                <tr>
                    <td>
                        <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                            <tr>
```

```
                        <td>
                            <!-- FEATURED HEADLINE -->
                        </td>
                    </tr>
                    <tr>
                        <td>
                            <!-- SOME COPY TEXT -->
                        </td>
                    </tr>
                    <tr>
                        <td>
                            <!-- A LINK -->
                        </td>
                    </tr>
                </table>
            </td>
        </tr>
        <!-- /FEATURED ARTICLE -->
    </table>
</td>
</tr>
</Table>
```

View on CodePen

If we were to preview that in a browser, all of the content would be jammed together.



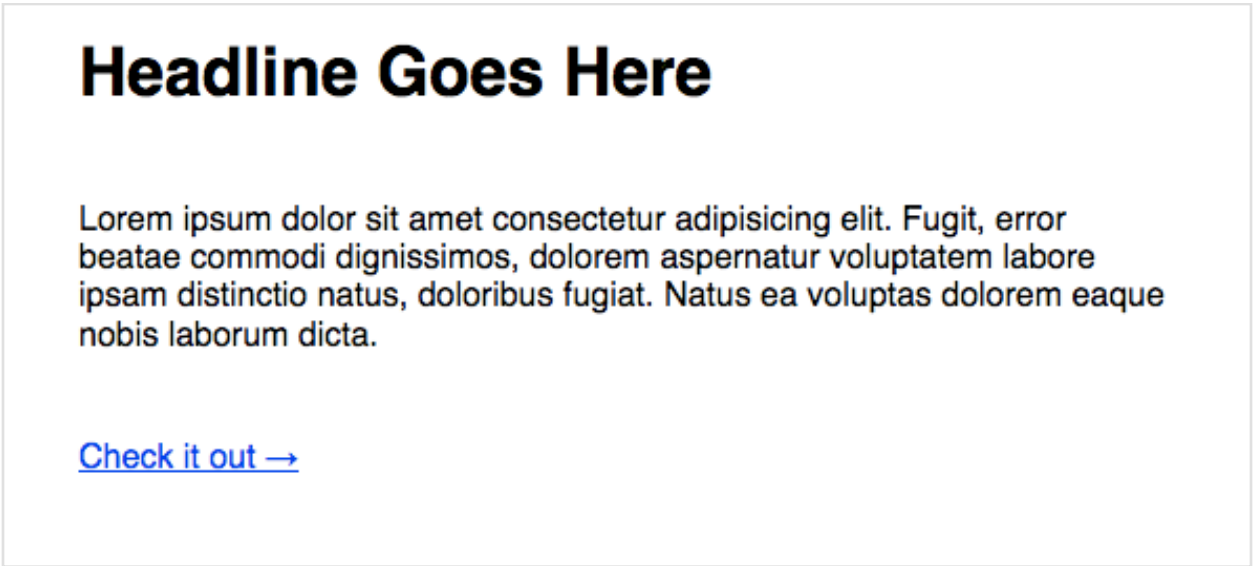
If we want to space out that content, we can use padding to add whitespace between elements. There are two places we typically add this padding.

You'll notice that we have a few tables here: an overall container table, an inner container table to constrain width to 600 pixels, and then a third, fluid table for the actual content.

The best place to apply padding is to the table cell in which the actual content resides. For example, on the featured headline section, we can add whitespace around that headline using padding on that table cell:

```
<td style="padding: 10px 20px 10px 20px;">
    <!-- FEATURED HEADLINE -->
</td>
```

Using the padding shorthand, we can add ample white space to a design. Now, when previewed in a browser, the content sections are discrete and easy to read.



Aligning Table Content

Separating our content with padding gets us half of the way there when it comes to structuring an email. The second half consists of aligning the content.

It's rare that every single thing in an email is left-aligned. More often than not, you will need some copy centered and some pulled to the right, too. For this, we rely on the align attribute applied to (where else?) a table cell.

```
<td align="center" style="padding: 10px 20px 10px 20px;">
    <!-- FEATURED HEADLINE -->
</td>
```

View on CodePen

In the example above, the featured headline will be centered on the screen.

The align attribute accepts three possible values: **`left`**, **`center`**, and **`right`**. I bet you can guess what each value does.

One thing to keep in mind is that some versions of Outlook will apply a container table's alignment to all children of that table. This sucks since, in most emails, the container table will be centered to keep the entire email in the middle of a preview pane. Without declaring alignment on any nested content, every element would be centered, too.

If you want to align content within a centered table either left or right, then you need to apply the appropriate alignment to that child table cell. Don't ever assume that Outlook (or any email client) will understand your intentions without explicitly declaring them in your code.

Multiple Columns in Email

Although I love single-column layouts, it's sometimes necessary to have multiple columns in an email. Multiple columns are a great way to present a lot of information in a very compact manner. And, if you're working in retail, they're almost required for showing off your products (whether or not that's always a good tactic).

When it comes to coding multi-column emails, there's not really anything special going on. Following the patterns laid out in the preceding sections, we will wrap our content in a 100% wide container table, followed by a fixed width table to constrain all of our content.

Then, we can apply percentage-based widths to the table cells for our columns.

You can also use pixel-based widths for your columns. However, keeping most of your tables percentage-based (except for those containers) makes responsive design much easier. Generally speaking, responsive design relies on overriding certain values to make them fluid. So,

overriding one value instead of multiple values makes things less complicated.

```
<table border="0" cellpadding="0" cellspacing="0" width="100%"
role="presentation">
      <tr>
        <td align="center">
          <table border="0" cellpadding="0" cellspacing="0"
width="600" role="presentation">
            <!-- TWO COLUMN SECTION -->
            <tr>
              <td align="center" valign="top" style="padding:
20px 10px 60px 10px;">
                <!-- TWO COLUMNS -->
                <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                  <tr>
                    <td>
                      <!-- LEFT COLUMN -->
                      <table border="0" cellpadding="0"
cellspacing="0" width="47%" align="left" role="presentation">
                        <tr>
                          <td>
                            <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                              <tr>
                                <!-- CONTENT -->
                              </tr>
                            </table>
                          </td>
                        </tr>
                      </table>
                      <!-- RIGHT COLUMN -->
                      <table border="0" cellpadding="0"
cellspacing="0" width="47%" align="right" role="presentation">
```

```
                        <tr>
                          <td>
                            <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                              <tr>
                                <!-- CONTENT -->
                              </tr>
                            </table>
                          </td>
                        </tr>
                      </table>
                    </td>
                  </tr>
                </table>
              </td>
            </tr>
            <!-- /COLUMN SECTION -->
          </table>
        </td>
      </tr>
    </table>
```

View on CodePen

Along with our typical **`align`** and **`width`** attributes, it's not a bad idea to include the **`valign`** attribute on the table cell. Multiple columns are usually the breaking point in most email designs, and vertically aligning your columns can help squash design oddities before they crop up.

While some designers will add their content directly to the outer column table cells, I prefer including the 100% wide content table above for added control over that content. If we want to add some whitespace

around the column content, we now have two options: on that column's table cell, or on any of the table cells within the content table. Plus, we can keep our components within that content table modular, too.

Dumping all of your content directly in the column table cell would require you to use hard-coded line breaks for adding spacing between sections in that column, which isn't very maintainable.

Sure, the content table adds some markup, but the flexibility and power it affords us far outweighs the slightly bloated code.

You can use the exact same method for introducing a third column.

```
<table border="0" cellpadding="0" cellspacing="0" width="100%"
role="presentation">
      <tr>
        <td align="center">
          <table border="0" cellpadding="0" cellspacing="0"
width="600" role="presentation">
            <!-- THREE COLUMN SECTION -->
            <tr>
              <td align="center" valign="top" style="padding:
20px 10px 60px 10px;">
                <!-- THREE COLUMNS -->
                <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                  <tr>
                    <td>
                      <!-- LEFT COLUMN -->
                      <table border="0" cellpadding="0"
cellspacing="0" width="33%" align="left" role="presentation">
```

```
                    <tr>
                      <td>
                        <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                          <tr>
                            <!-- CONTENT -->
                          </tr>
                        </table>
                      </td>
                    </tr>
                  </table>
                  <!-- CENTER COLUMN -->
                  <table border="0" cellpadding="0"
cellspacing="0" width="33%" align="left" role="presentation">
                    <tr>
                      <td>
                        <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                          <tr>
                            <!-- CONTENT -->
                          </tr>
                        </table>
                      </td>
                    </tr>
                  </table>
                  <!-- RIGHT COLUMN -->
                  <table border="0" cellpadding="0"
cellspacing="0" width="33%" align="right" role="presentation">
                    <tr>
                      <td>
                        <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                          <tr>
                            <!-- CONTENT -->
                          </tr>
                        </table>
```

```
                    </td>
                  </tr>
                </table>
              </td>
            </tr>
          </table>
        </td>
      </tr>
      <!-- /COLUMN SECTION -->
    </table>
  </td>
</tr>
</table>
```

View on CodePen

When you start adding multiple columns to your email, the math can sometimes get hairy. Some email clients (yep, Outlook) have problems with multiple columns. This typically happens because Outlook can add padding, margins, or borders—or screw up the math—between table cells. When this happens, columns will shift or drop below their intended positions.

A good way to account for this is to declare the widths of your columns to be slightly less than the total width of your email. In the examples above, I use **48%** and **33%** for two- and three-columns layouts, respectively. This is just a bit of insurance baked into the email, so even if extra spacing is added, things display as intended.

Modularity in Email

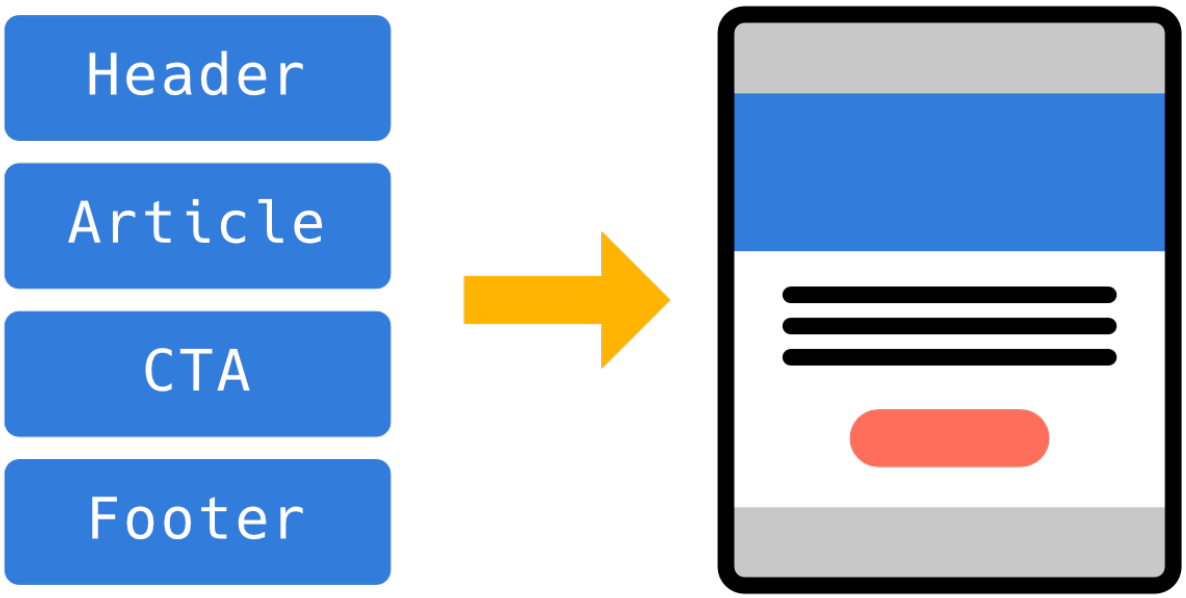
All of the examples above are basically single content sections, not entire emails. When it comes to building up entire email templates, my best piece of advice is to make them modular. The techniques for laying out tables described above allow you to build emails as collections of modular components.

Essentially, each horizontal section in your email is its own modular component. Those modular components consist of their own table structure which contains all of that section's content, just like we looked at in this chapter. Then, those modular sections are stacked to build an entire template.

Let's take a basic email newsletter as an example. The modular components, housed in their own 100% wide tables, would be broken down as follows:

- Header section
- Hero/featured item section
- Article section
- Article section
- Closing CTA section
- Footer section

To make things more visual, here's an illustration of that layout, with the modules identified.



While we could very easily wrap every content section as table rows and cells of a single parent table, using stacked, modular tables has a number of advantages.

First, they are reusable and allow us to mix and match different components to build out emails for a variety of purposes. We can easily build up a library of content components that can be swapped in and out of templates to cover pretty much any use case. This is massively helpful when you get into more complex email systems, too, for example when working on emails generated by automatic build systems.

Second, they make troubleshooting problems in an email much easier. Since each component is separate, we can quickly identify where a visual problem is. Instead of spending time hunting down what's causing a problem in one big table, we can use that time to fix the actual problem itself, no matter which module is affected.

Third, using stacked, modular tables makes it easier to deal with a weird Outlook bug that can break some designs. Basically, Outlook has an internal page height (based on the Word rendering engine) that can break really long emails. If your entire email is contained within one table, things end up looking fairly ugly when broken. However, using the modular method gives Outlook more natural breaking points, keeping your design largely intact.

You don't have to house each section in its own modular component, but it definitely makes things a lot easier. Just take into account your workflow, that of your team's, and what you need to accomplish before deciding on your approach to email design.

Every email campaign is basically a remix or combination of the above. Using single or multiple columns allow you to build up a near infinite variety of campaigns, especially when you keep your sections and components modular. Once you have your layout structured, it's time to start adding and styling content.

In the next chapter, we'll take a look at how to style the single most important element of any email campaign: text.

Chapter 3

Typography in Email

Chapter 3

Typography in Email

There's an **old article** from the design studio Information Architects that makes a great point:

> *Web design is 95% typography.*

As an offshoot of web design, the same is true about email. No matter what your message, you need to be able to communicate it clearly using type on screen. Clear type is only half the battle, though. We want to make email a delightful experience for our subscribers, so making that type accessible, beautiful, and enjoyable to read is our ultimate goal.

This chapter takes a look at how we can create accessible, readable, and aesthetically pleasing email campaigns that work well across email clients, even when using web fonts.

Accessible Typography in Email

As mentioned in the last chapter, one of our main goals when developing email campaigns is to make those emails accessible to the widest range of users possible. This is where semantic elements come into play in email.

Up until recently, email designers generally avoided using semantic elements like heading and paragraph tags because of the way most email clients handled them. Email clients and browsers naturally apply default styles to those elements which, when unaccounted for by the designer, can create undesirable results in a design. Copy was generally just dumped into a table cell, without any thought as to providing contextual information about that copy to anyone using a screenreader.

We want our copy to include that contextual information—or semantic value—so we will mark up copy with the appropriate HTML elements. Fortunately, for most email campaigns, that really only entails using the following tags:

- **Heading tags:** These are **`h1`** through **`h6`** depending on the level heading you need. Headings help create hierarchy and are indispensable to people using screen reader software.
- **Paragraph tags:** The humble **`p`** tag, which denotes a block of copy, typically longer than one sentence.
- **Span tags:** The **`span`** tag is useful for smaller bits of copy or as a hook to style anything within a larger heading or paragraph tag.

We can also make use of tags for meant for denoting emphasis, most notably the emphasis tag itself (**`em`**) and the strong tag (**`strong`**).

While there are other semantic elements we can use, like **`date`**, **`time`**, **`address`**, and **`blockquote`**, we'll be focusing on headings, paragraphs, and span tags in this book. Depending on your content, it may be worth

adding in major sectioning elements like **header**, **footer**, **main**, **article**, and **section** to provide extra semantic value for screen readers. Not all screen readers and email clients support those elements, but they will simply be ignored in those cases. However, for applications that do support them, it's a fantastic way to add extra structure and navigational waypoints to your campaigns. For the most part, you can use the techniques I'm about to describe for styling those elements, too.

Since we're using semantic elements, screen readers will create the proper hierarchy in our emails and convey that to the user, allowing them to more easily navigate and understand the content in our email campaigns.

An example of a semantically correct section of an email looks like this (taken from my own email newsletter):

```
<h1 style="font-size: 36px; font-weight: bold; line-height:
36px; margin: 40px 0px 100px 0px;">
    The Intermittent Newsletter<br>
    <span style="color: #ff2e66; font-size: 18px; font-weight:
normal;">by Jason Rodriguez</span>
</h1>
<h2 style="font-size: 24px; line-height: 28px; margin: 60px 0px
40px 0px;">
    The Nerves of a New Product Manager
</h2>
<p style="margin-bottom: 20px;">
    Some of you may know that I've changed roles at work. Where
once I was the community manager (god, I hate that term) and
marketing team member—among other things—I am now a product
```

```
manager focusing on improving and building out the <a
href="https://litmus.com/community/discussions">Litmus
Community</a>. Also, among other things. We all wear a lot of
hats.
</p>
<p style="margin-bottom: 20px;">
    When I made that switch late last year, I started keeping a
kind of diary online, in the guise of a Medium publication,
recording my thoughts on making the transition to product
management. I've since stopped posting there, but want to
continue talking about the subject on my own site.
</p>
```

View on CodePen

Overriding Default Styles

You can see in the example above that there are a few style rules applied to those semantic elements.

I mentioned before that email clients will apply default styles to your HTML. Headings are typically displayed larger and in bold text, differentiating them from paragraph text. And, since both headings and paragraphs are block-level elements, margins are usually added to them as well, creating white space in an email.

While this is a nice default feature of a rendering engine, we typically want to override that behavior and give ourselves a blank canvas on which to paint our own styles. For now, we will simply override the

margins on headings and paragraphs by including the margin property as an inline style.

```
<h1 style="margin: 0;">This is a heading.</h1>
<p style="margin: 0;">This is a paragraph.</p>
```

If you want to include white space around those elements, simply use the margin shorthand property to add in white space.

```
<p style="margin: 0 0 20px 0;">This is a paragraph with 20
pixels of white space applied to the bottom of the element.</p>
```

That covers white space issues across clients, but how about actually styling the look of the text?

Applying Styles to Text

There are generally two ways in which you will apply styles to text in an email.

The first is by applying text styles to the **td** element that contains that text, then using semantic elements within that table cell.

```
<td style="color: #000000; font-family: sans-serif; font-size:
18px; font-weight: normal; line-height: 24px;">
    <p style="margin: 0;">This is a paragraph.</p>
</td>
```

The second is by setting all text styles on the semantic elements themselves.

```
<td>
    <p style="color: #000000; font-family: sans-serif; font-
size: 18px; font-weight: normal; line-height: 24px; margin:
0;">This is a paragraph.</p>
</td>
```

While it's a personal preference, I encourage you to apply the default text styles you want on the table cell, then override where necessary on specific elements within that table cell. This approach has two major benefits:

1. Lightweight markup: Setting styles in one place means less code duplication on multiple elements, keeping your code lean.
2. Maintainability: Since styles are set in one place, the look of the email is easy to update as you're changing values in one spot instead of many.

In the example taken from my own newsletter, I don't rely on tables. Instead, the default text styles are applied to a div element that wraps all of the text. The styles shown above are overrides for those specific elements, most notably to make the headings more prominent.

I still use the **`margin`** property on semantic elements within that table cell, but the bulk of the work happens within the table cell.

This approach will be applied to all of our text with the exceptions of:

- Links, which will be styled using the **a** tag. We'll get to those later.
- Text within a section that needs special styling, which will be handled with the **span** tag.
- Things like dates, times, phone numbers, etc. that some email clients insist on making links.

Basic Font Styles

To cover our bases, if a table cell contains text, it should always have the following styles defined:

- **color**
- **font-family**
- **font-size**
- **font-weight**
- **line-height**

Let's look at an unstyled block of text:

```
<td><p>The next time you email your subscribers, stop and think
before telling your boss that you sent out the latest blast.
Instead, show her that you value your subscribers as people by
saying you sent the latest message to your audience. You sent a
campaign. You sent the next part in an ongoing conversation.
</p></td>
```

When viewed in a browser or most email clients, it will be displayed like so:

The next time you email your subscribers, stop and think before telling your boss that you sent out the latest blast. Instead, show her that you value your subscribers as people by saying you sent the latest message to your audience. You sent a campaign. You sent the next part in an ongoing conversation.

Let's say we want to make the text lighter, use a sans-serif font, increase the size a bit, and add some **`line-height`** to make the reading experience more comfortable. We can accomplish this with the properties listed above:

```
<td style="color: #666666; font-family: sans-serif; font-size: 18px; font-weight: normal; line-height: 28px;”><p style=“margin: 0;>The next time you email your subscribers, stop and think before telling your boss that you sent out the latest blast. Instead, show her that you value your subscribers as people by saying you sent the latest message to your audience. You sent a campaign. You sent the next part in an ongoing conversation.</p></td>
```

View on CodePen

The next time you email your subscribers, stop and think before telling your boss that you sent out the latest blast. Instead, show her that you value your subscribers as people by saying you sent the latest message to your audience. You sent a campaign. You sent the next part in an ongoing conversation.

That's better, huh? You could just copy and paste that formula into all of your emails, but then you wouldn't learn anything, would you? Let's take a look at each of those properties and see how they work.

Color

The color property controls the color of the text inside a given HTML element. It accepts a variety of values, including:

- The CSS **`color`** keyword (e.g. **`red`**, **`yellow`**, **`blue`**).
- A three-character hexadecimal value (e.g. **`#f00`** produces red).
- A six-character hexadecimal value (e.g. **`#ff0000`** also produces red).
- An RGB or RGBa value (e.g. **`rgb(0, 0, 0)`** produces black, **`rgba(0, 0, 0, 0.5)`** produces black at half opacity).

Like most other things, these all work on the web but not in every email client. The safest way to declare colors in HTML email is by using the six-character hexadecimal value. Therefore, anytime a color is declared in this book, it will take the form:

```
color: #ffffff;
```

Some email clients, typically ones which use the WebKit rendering engine, can handle the other methods, too. If you know your audience is largely opening emails in WebKit-based email clients (Apple Mail, iOS

Mail), then you can use something like RGBa, which is useful for adding transparency to text.

Font Stacks

The font-family property controls what typefaces are used to display text. If left undefined, operating systems and email clients will default to system fonts (e.g. Times New Roman, Helvetica, San Francisco, etc.) that come installed on most operating systems. You should absolutely define a font stack—an explicitly stated preferred ranking of fonts—for your text.

A font stack allows you to declare back up fonts for when a preferred font isn't installed on a computer or, in the case of web fonts, is unable to be downloaded. When declaring your font stack in the font-family property, you put your preferred font first, followed by increasingly generic backup options:

```
font-family: Futura, 'Trebuchet MS', Arial, sans-serif;
```

In this instance, if Futura is installed, text will be displayed using that font. In Futura's absence, it will then fall back to Trebuchet MS first, Arial second, and whatever sans-serif font is installed on the operating system as a last resort.

You will notice that Trebuchet MS is surrounded by quotes. This is because it consists of two separated words. Many fonts have complex

names and, if they include spaces, will need to be wrapped in single quotes. This handles any email clients that can't properly parse spaces in the font-family property.

Font Sizes

On the web, there are a number of ways to declare font sizes. You can use keywords like **`large`** or **`xx-small`**, fixed units like **`px`**, or relative units like **`em`** and **`rem`**. Since we're working towards always building responsive emails, you would be right in thinking that relative units like **`em`** and **`rem`** would be ideal. However, a lot of email clients don't understand font sizing unless declared using pixels.

Therefore, all of our font sizes will use the px value:

```
font-size: 18px;
```

You can make that font size as big or little as you want, but you should keep a few things in mind.

First, some mobile devices (looking at you, iPhone and iPad) automatically resize text that is smaller than 14px in an effort to make it easier to read on smaller screens. Anything below 14px will be resized up to 14px. So, if you're using small text for something like a disclaimer, you will need to account for this in your code or risk your design being broken.

This can be accomplished by using the **`text-size-adjust`** CSS property, prefixed for the appropriate rendering engine. WebKit on iOS is the main perpetrator, but Windows Mobile does something similar. As an example, we can target both. You can do this in one of two places.

As a style reset in the head of our document:

```
<head>
    <style type="text/css">
        body, table, td, a {
            -webkit-text-size-adjust: 100%;
            -ms-text-size-adjust: 100%;
        }
    </style>
</head>
```

Or inline on an element in case the head is stripped out of the document:

```
<td style="-webkit-text-size-adjust: 100%; -ms-text-size-adjust:
100%;">CONTENT</td>
```

Since these prefixed properties are only understood by rendering engines that don't strip embedded CSS, you can safely declare them in the head of your HTML document, keeping your inline styles cleaner and easier to read.

Second, you should use a comfortable font size for all readers. A good baseline is to have text around 16-18px. This is big enough for most

readers to easily read, while not being so big it only allows you to fit 20 characters on a line.

Finally, don't make your text too big. While you want your headlines to stand out from your body copy, don't make them absurdly massive (unless dictated by your design team or branding guidelines). Use other properties like font-weight and color to create hierarchy, in tandem with **`font-size`**.

When we look at making emails responsive, we'll come back to **`font-size`** and see how we can change the size of type on different devices.

Font Weights

The **`font-weight`** property allows you to specify how heavy a font should appear. The weight refers to the thickness of the lines that make up a letter. A great typeface will have a lot of different weights available. Other typefaces will only have one or two.

A font's weight can be declared using either a keyword (e.g. **`normal`**, **`bold`**, **`bolder`**) or a numerical value (e.g. 100, 200… 900). In most cases, you will likely only use the **`normal`** or **`bold`** keyword. However, if you are using a typeface with a lot of variations, say Proxima Nova, you may want to rely on numerical values for font weights, since it allows greater flexibility of design and access to all of the available weights.

Let's say we have a large headline set in a thin variation followed by some smaller body copy that needs to be set in a medium weight. You can mark that up like so:

```
<tr>
    <td style="color: #222222; font-family: 'Proxima Nova', Arial, sans-serif; font-size: 32px; font-weight: 100; line-height: 32px;">
    <h1 style="margin: 0;">Blastphemy</h1>
    </td>
</tr>
<tr>
  <td style="color: #666666; font-family: 'Proxima Nova', Arial, sans-serif; font-size: 18px; font-weight: 400; line-height: 22px;"><p style="margin: 0;">The next time you email your subscribers, stop and think before telling your boss that you sent out the latest blast. Instead, show her that you value your subscribers as people by saying you sent the latest message to your audience. You sent a campaign. You sent the next part in an ongoing conversation.</p></td>
</tr>
```

The typical design will likely call for a bold headline and normally weighted text. This is where you can simply use keywords:

```
<tr>
    <td style="color: #222222; font-family: Arial, sans-serif; font-size: 32px; font-weight: bold; line-height: 32px;">
        Blastphemy
    </td>
</tr>
<tr>
```

```
    <td style="color: #666666; font-family: Arial, sans-serif;
font-size: 18px; font-weight: normal; line-height: 22px;">
        The next time you email your subscribers, stop and think
before telling your boss that you sent out the latest blast.
Instead, show her that you value your subscribers as people by
saying you sent the latest message to your audience. You sent a
campaign. You sent the next part in an ongoing conversation.
    </td>
</tr>
```

Both approaches are perfectly valid, it all comes down to how specific you need to be with weights and whether you want to keep track of a bunch of number values or a few keywords.

Line Heights

The **`line-height`** property allows you to specify the amount of space between lines of text. Again, you can specify values a few ways, including a unit-less number, fixed values like **`px`**, and relative units like **`em`** and **`%`**.

On the web, it is common practice to declare line-heights using a unit-less number to avoid inheritance issues and unpredictable results. Unfortunately, not all email clients understand unit-less numbers. So, your safest bet is to use a pixel value when setting the line-height property, like so:

```
<td style="line-height: 20px;"></td>
```

The space between lines has a very real correlation to how easily we can read text. If text has too little space between lines, it feels cramped and uncomfortable to read. If there is too much space, the eye is forced to make too large of movements between lines and text becomes disjointed.

The line height of a block of text works in tandem with the font size of that text, as well as the width of the block of text itself. Typically speaking, smaller text should have larger line heights, while larger text can afford to have smaller line heights. My general rule of thumb is to have the line height 4-8 pixels larger than the font-size for most normal text. For headings, that will drop to either equal that value or even be two or four pixels smaller than the font-size value. However, you should always test out different line height in your designs to see what feels comfortable to read.

Font Styles

While not mentioned above as an absolutely necessary style to apply, another useful CSS tool is the font-style property. This property allows you to specify one of three variations of a font: **`normal`**, **`italic`**, or **`oblique`**.

Without declaring a font style, text will be displayed using the **`normal`** value. Both **`italic`** and **`oblique`** are a means to a similar end in that they will slant a typeface. Italic fonts typically use a properly designed,

italic variation that is included in a typeface. Oblique fonts rely on skewing the regular version of a typeface to create slanted fonts.

You will likely only ever use the **`italic`** value to make your text, you know, italic. This can be useful to provide emphasis on a word or phrase, or to set a particular section apart from the surrounding text, for example, in a photo caption or a disclaimer.

If you want to accomplish the same thing using a semantic tag, you can use the **`em`**, or emphasis, element. This is my preferred method, as it adds added contextual information for people with disabilities relying on screen reader software.

Targeting Specific Copy

Every once in a while, you will need to style some text separately from what surrounds it. While we've looked at using the **`em`** and **`strong`** tags to semantically markup that text, we often need a more generic solution to targeting and styling small snippets of text. That's where the **`span`** tag comes in.

Let's say we have some text:

```
<td style="color: #666666; font-family: Arial, sans-serif; font-size: 18px; font-weight: normal; line-height: 22px;">The next time you email your subscribers, stop and think before telling your boss that you sent out the latest blast. Instead, show her that you value your subscribers as people by saying you sent the
```

```
latest message to your audience. You sent a campaign. You sent
the next part in an ongoing conversation.</td>
```

You want to emphasize the line, 'You sent the next part in an ongoing conversation." Similar to how we apply styles to a table cell, we can simply wrap that line in a span and set the font-style to italic.

```
<td style="color: #666666; font-family: Arial, sans-serif; font-
size: 18px; font-weight: normal; line-height: 22px;">The next
time you email your subscribers, stop and think before telling
your boss that you sent out the latest blast. Instead, show her
that you value your subscribers as people by saying you sent the
latest message to your audience. You sent a campaign. <span
style="font-style: italic;">You sent the next part in an
ongoing conversation.</span></td>
```

Now, that line will inherit the values set for every other property in the table cell but will be emphasized, too. If we wanted to also change the color of that text, or make it bold, we can apply those rules on the same span. We can override any rules set in the table cell or add CSS on top of the inherited styles.

Admittedly, that's not the best example. If I were coding that in real life, I'd use the **em** tag, which automatically makes the text italic for nearly all rendering engines. Although, if I wanted to change the color of that text, too, I'd still use an inline style on the em tag.

A better example comes from my own newsletter, which has a title with a byline, which is all marked up as a level 1 heading.

```
<h1 style="font-size: 36px; font-weight: bold; line-height:
36px; margin: 40px 0px 100px 0px;">
    The Intermittent Newsletter<br>
    <span style="color: #ff2e66; font-size: 18px; font-weight:
normal;">by Jason Rodriguez</span>
</h1>
```

I use a line break (**br**) to dump the byline onto its own line, then wrap that in a **span**, where I apply different **color**, **font-size**, and **font-weight** values to override the ones set on the parent **h1**. In either example, though, you can see how useful a well-placed **span** is.

This is the best way to fine-tune specific parts of our copy. If you wanted to get really crazy and make your email look like a ransom letter, you could wrap individual letters, words, and phrases in their own span and apply different styles to each.

The Custom Font Problem

Earlier, I talked a bit about declaring font stacks. When declaring a font stack, you specify your preferred font followed by a series of backup fonts. Without doing anything on our end, the display of those fonts depends entirely on what fonts are installed on a user's machine.

A lot of brands pay large sums of money to license a typeface. Each typeface has its own personality, and brands use typefaces to express their personality. However, in many cases, most users do not have these

typefaces installed on their computers. So, even if a designer was to specify one of these typefaces in their font stack, there is a good chance that one of the backups will be displayed instead.

How can we, as email designers, work around this? There are currently two solutions which you'll encounter.

Using Images

Until a few years ago, most email designers would turn to including that text as part of an image and then just dumping the image inside of an email. Hell, a lot of major brands still take this route.

While using images in place of HTML text allows you to throw any typeface you want into a design, it has some major drawbacks:

- A lot of email clients block images by default, which means that your subscribers won't see any of that pretty text unless they take some extra action to display images.
- Forcing your mobile users to download a ton of images is just plain mean, especially as data plans become more expensive.
- Images aren't as easy to adjust for mobile devices as they typically just get scaled down, making that text harder to read.
- Images pose accessibility issues, even with alt text specified.

Unless it's absolutely necessary, you should avoid using images to serve any text. HTML text trumps image-based text every time. Just use good fallbacks and get comfortable with the fact that some subscribers will see those fallbacks.

When it comes to fallback fonts, you should strive to pick fonts that are similar to your preferred font. Use characteristics like x-height, letter spacing, color, contrast, and overall style to determine solid fallbacks.

Using Web Fonts

The best, and only real, alternative is to use what are known as web fonts. Web fonts are fonts that are specifically licensed to be served on websites (and in email). The fonts themselves are not installed on a user's computer, but are instead temporarily downloaded and displayed from a server just like any other asset, like an HTML file or an image. This allows us to get around the problem of users not having a wide variety of fonts installed.

In order to use a web font, we need to have a way to call that font from within an email and signal it to download and display. There are essentially three ways to do this:

- Using the **`@font-face`** rule.
- Using a **`<link>`** to an external stylesheet.
- Using the **`@import`** feature.

Let's see how each of these work in practice.

@font-face Method

The **`@font-face`** rule allows you to build up your own custom font by linking to actual font files within CSS and specifying things like **`font-family`**, **`font-style`**, and **`font-weight`** for each variation you need. It is the most involved option as you need to link to a few different font formats for compatibility when building your web font rule.

Let's say we wanted to use Futura in an email. Using the @font-face rule, we would declare the following within a style block in the head of an email:

```
@font-face {
    font-family: 'Futura';
    src:
    url('https://yourserver.com/fonts/futura-regular.woff')
format('woff'),
    url('https://yourserver.com/fonts/futura-regular.ttf')
format('truetype');
    font-weight: 400;
    font-style: normal;
}
```

Then, in our table cell, we can call that font at the beginning of our font stack:

```
<td style="color: #000000; font-family: Futura, 'Trebuchet MS',
Arial, sans-serif; font-size: 18px; font-weight: 400; line-
height: 22px;">CONTENT</td>
```

If we wanted to include multiple weights or variations of that font, then we would need to write additional **`@font-face`** rules for each one, specifying the variation via the **`font-weight`** or **`font-style`** properties. This can lead to some bloated code and is, quite frankly, not always the best route.

<link> Method

The second method uses the **`link`** tag to pull in an external stylesheet that essentially holds all of those **`@font-face`** rules for us. This method works best with services that host web fonts for you, like **Google Web Fonts**. If you're coming from the web, chances are you've seen this method before. It looks like this:

```
<link href='http://fonts.googleapis.com/css?family=Open+Sans:
700,400' rel='stylesheet' type='text/css'>
```

After that link is added to the head of your document, you can call the font directly in your inline styles:

```
<td style="color: #000000; font-family: 'Open Sans', Helvetica,
Arial, sans-serif; font-size: 18px; font-weight: 400; line-
height: 22px;">CONTENT</td>
```

This method, while still not great, does have better support than the vanilla **`@font-face`** option.

@import Method

Similar to the previous method, using **`@import`** allows us to pull in an external stylesheet. The difference is that it is pulled into a style block within the head instead of directly into the head itself.

```
<head>
    <style type="text/css">
        @import url(http://fonts.googleapis.com/css?
family=Open+Sans:700,400);
    </style>
</head>
```

Now, you can use it the same as the methods before, by simply declaring that font at the beginning of your font stack.

While this method has the best support across email clients, it still doesn't work everywhere. That's why choosing good fallback fonts is so damned important.

The Outlook Problem

Aside from a lack of universal support, the biggest problem with using web fonts is that some versions of Microsoft Outlook will not only ignore

the web fonts, but will fall back to Times New Roman instead of any specified fallbacks.

Although there are a few ways to prevent this, my preferred method is to wrap any web font imports in their own **`@media`** block, like so:

```
<style>
    @media screen {
        @import url('https://fonts.googleapis.com/css?
family=Anonymous+Pro:400,700');
    }
</style>
```

For whatever reason, Outlook will essentially ignore that web font and simply fall back to your other font choices.

There is a lot more to typography outside the scope of this book. If you really want to dig in, I highly recommend Paul Airy's book, ***A Type of Email***. For now, this should get you started in building robust, text-centric email campaigns. Next, we'll talk about using that text to take people places by means of the most important component of the internet: the hyperlink.

Chapter 4

Taking People Places

Chapter 4

Taking People Places

Aside from having access to all the world's information, the beauty of the internet is in the ability to connect people and take them places all on a simple screen. This ability is made possible by the humble, but unbelievably powerful, anchor tag. And, just like on the internet at large, the **a** tag is what makes email one of the most valuable communications channels today.

Providing information in an email is always good, but anyone sending email wants the recipient to take some action. These actions, this engagement, is facilitated by the link. So, it's important to understand how best to use and style links when developing an email campaign.

Basic Text Links

At the most basic level, links are applied to text within the body of an email. These work in exactly the same way as on the web.

```
<a href="http://example.com">Read the article now</a>
```

By default, a link will inherit the basic font styles from its parent element which, in most cases, will be the table cell containing our text. Since it is

an anchor tag, the operating system, browser, email client, or rendering engine will apply additional styles to the element. That usually looks something like this:

Ah, the old default link styles of days past!

While the blue underlined links of yore work beautifully in plain text emails, most people need to apply additional styling to those links so that they feel a part of the overall design of a campaign.

We can apply any number of inline styles on an anchor tag. Typically, you will want to override the **`color`** property. Often, you will want to remove the underline, as well. This is easily accomplished via the **`text-decoration`** property.

```
<a href="http://example.com" style="color: #6aa7a4; text-decoration: none;">Read the article now</a>
```

Feel free to play with other CSS properties like **`font-weight`**, **`font-style`**, and even CSS3 like **`text-shadow`**. Just don't go crazy with those shadows–your subscribers' eyes will thank you.

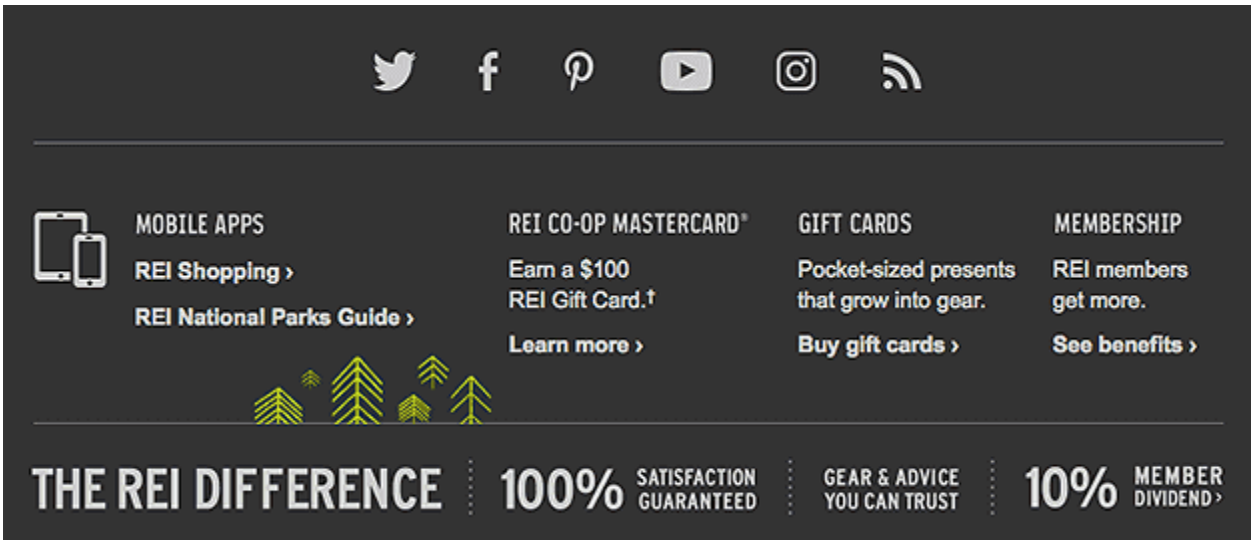
Link Clusters

One thing to keep in mind with basic text links, especially as more people are checking their email on mobile devices, is that tightly packed groups of links are frustrating to actually use. Usually referred to as link clusters, these groupings typically manifest themselves in two ways.

The first is when using larger blocks of text with multiple links throughout:

> Lorem ipsum dolor, sit amet consectetur adipisicing elit. Aliquid vel delectus accusantium voluptatibus enim dolorem eaque aperiam, voluptas laborum necessitatibus fugit at sunt quo deserunt facilis eius nam ab maiores. Read the article now

The second is usually found in sub-navigation. You've likely seen an email similar to the following:



In both cases, you can see that links are tightly packed. Especially for large-thumbed folks, these links can be mind-numbingly frustrating to interact with. Hoping to be taken to one page, subscribers accidentally tap one of the surrounding links and are taken to another. This kind of frustration is what leads to deleted emails and unsubscribes.

The best way to avoid this kind of frustration is by adding space around your text links.

When dealing with large blocks of text, this can largely be handled by increasing the **`line-height`** of the text. Experiment with increasing the value of **`line-height`** on your table cell until links feel comfortable to use in that copy block. If your design allows it, you can also increase the **`font-size`** to make the actual target bigger and easier to tap.

Honestly, for sub-nav elements, my first recommendation would be to just get rid of them. A lot of times, these links are forced into a campaign by a marketing team or stakeholder, regardless of the fact that most subscribers don't ever use them. While it's not true for every audience, I'd wager that most emails would be better off without this kind of cruft.

But, if your boss is breathing down your neck and insists on having those damned links in the email, here's what you should do: add some space around the links!

In this case, there are typically two ways to handle adding space to your navigation links–depending on how you initially coded them.

A lot of email designers will code those links inline, all in one table cell, with non-breaking spaces separating the links. In this case, the obvious remedy is adding more non-breaking spaces, sometimes to an absurd extent, until there is ample whitespace between links on both desktop and mobile.

The better, and more flexible solution (especially when you start adding in responsive styles), is to code each link in its own table cell. Then, similar to how we do most things in email, we can add additional styles to those cells. In this case, we can increase the padding on the table cells, spacing them to our heart's delight.

Still, you may want to consider just dropping them anyways...

iOS Blue Links

A curious thing happens with text on some mobile devices. Apple's iOS operating system, in an attempt to help out the user, will automatically turn certain types of text into links. Without you ever wrapping the text in an anchor tag, you will likely see the following in ugly, blue, underlined text:

- Dates
- Phone numbers
- Addresses

- Times
- Email addresses

The reasoning is sound on Apple's part: that's all useful information that can be added to your contacts, calendars, or mapped out, so why not make that process easier? Unfortunately, this tends to anger a lot of designers and can, in some cases, turn out to be a disaster for users.

Let's say you have a footer section with a dark background. Naturally, you're complying with CAN-SPAM rules and include your mailing address. Now, when viewed on an iOS device, the blue links are nearly unreadable:

Wouldn't it be great if we could override these automatic link styles? Well, we can.

There are two approaches to handling blue links.

The first is by including a reset style for those blue links in the head of your HTML document:

```
/* iOS BLUE LINKS */
a[x-apple-data-detectors] {
    color: inherit !important;
    text-decoration: none !important;
    font-size: inherit !important;
    font-family: inherit !important;
    font-weight: inherit !important;
    line-height: inherit !important;
}
```

View on CodePen

This is a quick and easy way to kill blue links on iOS. However, since this targets Apple's operating system specifically, it won't have an effect on some other clients that add blue links.

If you encounter blue links outside of iOS, you can add a style in that targets content that is likely to be auto-linked and override the default styles of any added links.

We don't want to run the risk of overriding any other link styles in text blocks, so instead of targeting the table cell, it's a good idea to just wrap any of the above mentioned suspects (date, time, address, etc.) in a **span** and target that with an appropriately named class:

```
<span class="blue-links">
    1234 Main St. <br>
    Anywhere, MA 56789
</span>
Then, in the head of the document, we can target any links
within that class and set our styles:
<style type="text/css">
    .blue-links a {
        color: #888888;
        text-decoration: none;
    }
</style>
```

Now, any automatically linked text will appear as intended. Our designs will still be readable and, more importantly, useful as users will still be able to press that element and add it to their contacts just like normal.

Gmail Blue Links

In October of 2017, Google started rolling out an update to Gmail that adds similar auto-linking behavior. Fortunately for us, a nearly identical fix as the one for iOS was quickly discovered. It applies the same CSS rules but has different targeting:

```
u + #body a {
```

```
    color: inherit;
    text-decoration: none;
    font-size: inherit;
    font-family: inherit;
    font-weight: inherit;
    line-height: inherit;
}
```

View on CodePen

You will need to add **id="body"** to the **body** tag of your email for this to work. The fix works because Gmail changes the doctype of an email to an underline tag (**u**). Then, by targeting the body using that **id** attribute and any links contained within, we can override any applied styles with those CSS properties.

One thing to note is that **you cannot chain the Gmail and iOS fixes together into one CSS declaration.** This is because the iOS blue links fix uses the attribute selector method of targeting (those square brackets). Gmail, unfortunately, does not currently support selecting HTML elements using attribute selectors and will rip that style out of the document, leaving your links as blue as can be. Separating them, though, allows you to keep your links looking good in both clients.

Buttons

Aside from linked images (which we'll discuss in the next chapter), the other major link type in an email is the button. Just like on the web,

buttons are a great way to style links and draw attention to some action you want a user to perform.

However, due to the limitations of email clients (both through CSS support and image-blocking behaviors), buttons in an email campaign require very specific considerations.

Image-Based Buttons

The old way of including a button was to use an image for the button and wrap that image in an anchor tag. While this has the benefit of allowing you to style your button however you want (I'm looking at you, Photoshop layer styles), a lot of email clients block images by default. More often than not, your beautifully beveled button wouldn't even be seen by a subscriber. Think they would actually press it?

Unless you absolutely have to heavily style a button, you should avoid image-based buttons at all costs. There's a better way…

Bulletproof Buttons

Bulletproof buttons refer to a method of building and styling buttons using code instead of images. This has a few benefits, depending on the approach:

- Your buttons are visible even when images are disabled

- Emails are quicker to build (no time-consuming trips in and out of Photoshop)
- Emails are easier to maintain and customize
- Your buttons can be adjusted for mobile devices, making them easier to use

There are typically three approaches when it comes to crafting bulletproof buttons, each with its own advantages and disadvantages. While there are a few hybrid solutions, too, these three approaches are the most accessible and can easily be implemented in any email campaign.

VML-Based Buttons

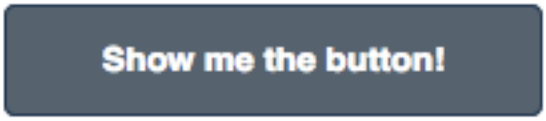
The first approach is the classic bulletproof button. Made popular by Campaign Monitor developer **Stig Morten-Myre**, these bulletproof buttons rely on Microsoft's proprietary language VML (Vector Markup Language). Using Outlook-specific conditional comments, the VML wraps a simple, styled link tag to get the buttons working in most email clients:

```
<div><!--[if mso]>
      <v:roundrect xmlns:v="urn:schemas-microsoft-com:vml"
xmlns:w="urn:schemas-microsoft-com:office:word" href="https://"
style="height:40px;v-text-anchor:middle;width:200px;"
arcsize="10%" strokecolor="#1e3650" fill="t">
    <v:fill type="tile" src="https://i.imgur.com/0xPEf.gif"
color="#556270" />
    <w:anchorlock/>
```

```
    <center style="color:#ffffff;font-family:sans-serif;font-
size:13px;font-weight:bold;">Show me the button!</center>
      </v:roundrect>
    <![endif]--><a href="https://"
    style="background-color:#556270;background-
image:url(https://i.imgur.com/0xPEf.gif);border:1px solid
#1e3650;border-radius:4px;color:#ffffff;display:inline-
block;font-family:sans-serif;font-size:13px;font-
weight:bold;line-height:40px;text-align:center;text-
decoration:none;width:200px;-webkit-text-size-adjust:none;mso-
hide:all;">Show me the button!</a></div>
```

View on CodePen

Here's what that button looks like:



Does that markup look confusing? It does to me, too. I'd wager there are about ten people in the world that could eloquently explain VML. I'm definitely not one of them. This leads us to the biggest problem with VML-based buttons:

They are incredibly difficult to update and maintain.

The second problem is that they aren't the most flexible when it comes to building responsive emails. However, they do work most everywhere,

even in Outlook. And you can pull off some fancy effects using VML. I'd recommend using them when you need either a) full Outlook compatibility or b) near-Photoshop styling capabilities.

Fortunately, Campaign Monitor has a handy tool for generating the markup for you without having to learn the ins-and-outs of VML. Head on over to **buttons.cm** to get going.

Padding-Based Buttons

The second approach is to use a single-row, single-cell table with a link. Styling is then applied to the table cell to structure the button, with additional styles added to the link itself (**`color`**, **`font-size`**, etc.). It looks something like this:

```
<!-- PADDING-BASED BUTTON -->
<table border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td bgcolor="#357edd" style="padding: 20px 40px 18px
40px; -webkit-border-radius: 8px; border-radius: 8px;"
align="center">
            <a href="https://thebetter.email" target="_blank"
style="font-size: 18px; font-family: sans-serif; color: #ffffff;
text-decoration: none; display: block;">Read more here &rarr;</
a>
        </td>
    </tr>
</table>
```

View on CodePen

You can see that this markup is incredibly easy to update and maintain long-term. It uses the exact same styles as everything else in our campaigns and we also have the added benefit of a nice, flexible button when we start making things responsive. What's more, we can use background images on the button and can pull off some interesting looks by applying styles to both the table cell and the anchor tag.

However, there are two main drawbacks to this approach:

1. Some email clients don't support padding, so the buttons will collapse
2. Only the text of the link is clickable, not the entire button

While that second point puts some designers off, I have yet to hear a subscriber actually complain about it. If you want to only use HTML and CSS and still need your entire button clickable, it's time to look at our final method.

Border-Based Buttons

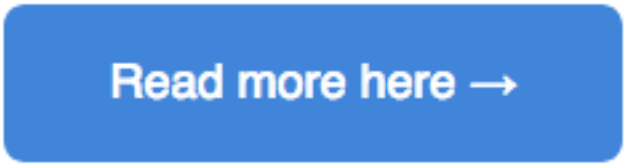
These buttons take an almost identical approach.

```
<!-- BORDER-BASED BUTTON -->
<table border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td>
```

```
            <a href="https://thebetter.email" target="_blank"
style="font-size: 18px; font-family: sans-serif; color: #ffffff;
text-decoration: none; border-radius: 8px; -webkit-border-
radius: 8px; background-color: #357edd; border-top: 20px solid
#357edd; border-bottom: 18px solid #357edd; border-right: 40px
solid #357edd; border-left: 40px solid #357edd; display: inline-
block;">Read more here &rarr;</a>
        </td>
    </tr>
</table>
```

View on CodePen

The major difference is that, instead of the structure being built with padding on the table cell, it is built with extremely thick borders on the anchor tag itself. With all of the styling on the anchor tag, the entire button ends up being clickable. And, our button still looks great:



The drawbacks here are that Outlook doesn't always like those large borders (shrinking them down), background images aren't supported, and you can't apply more advanced styles (splitting them between the table cell and anchor tag).

Keep in mind, there are other solutions out there for building buttons in email. Some email designers take a hybrid approach and combine elements of the padding- and border-based buttons, along with some conditional styles for Outlook. It all depends on your needs and where your audience is opening emails.

In the next chapter, we'll look at how to build on the solid, text-based campaigns you're hopefully coding by adding in everyone's favorite element: images.

Chapter 5

Images in Email

Chapter 5

Images in Email

I love a good, all-text email. If you can pull off an elegant, interesting design using only type then I'll buy you a beer.

That's just me, though. Most people like seeing a beautiful photograph or a quirky illustration in an email, too. In this chapter, we'll take a look at how best to incorporate images in our campaigns. We'll start by looking at which image formats work best, how to mark them up, using retina-ready images, and what to do when those damned email clients block our pretty pictures.

Which formats work best?

How many file formats are there for images? A dozen? Two dozen? More?

Fortunately for us, we only really need to concern ourselves with three image formats for everyday use. When it comes to email design, you'll likely only ever encounter the following:

- JPEG
- PNG
- GIF

You may run into the occasional BMP, and more recently SVG files, but email designers have largely standardized on the three formats above.

Let's look at the benefits and drawbacks of each.

JPEG (.jpg)

The JPEG file format, which stands for Joint Photographic Experts Group, is a great option when you need to include complex graphics like photographs or illustrations. Really, anything with either a lot of colors or subtle gradations between colors or shades.

JPEGs are lossy, which means that, when compressed to save space, they discard information that can lead to a loss in quality or the introduction of visual artifacts. However, unless you're using extreme compression settings, most people won't notice the loss in quality.

They are perfect for pictures of people, product shots, and intricate illustrations. I'd wager that most images you see in an email are likely JPEGs. They're a great option. More importantly, JPEGs are supported across virtually every email client.

The two main drawbacks of JPEGs are that they do not support transparency and they cannot be animated. If we want transparency, we'll want to use…

PNG (.png)

The Portable Network Graphics format, or PNG for short, is similar to JPEG in that it is a great option for anything with lots of colors. It works equally well for simpler graphics that need to be nice and crisp. In contrast to JPEG, it is a lossless file format—meaning that it does not discard any information when compressed. Because of this, file sizes for complex images can be larger than when using a JPEG.

There are a few varieties of the PNG format: PNG-8, PNG-24, and PNG-32. PNG-8 only supports up to 256 colors, compared to the millions for the other two options. PNG-8 also lacks support for transparency. Therefore, I would highly recommend using PNG-24 when saving images.

While widely supported, PNG does not work in the worst email client in the world: Lotus Notes, specifically versions 6 and 7. For most of us, though, we can safely use PNGs since we aren't catering to dinosaurs. However, if you have an audience that likes using frustratingly slow, archaic enterprise email clients, you will need to rely on either JPEGs or…

GIF (.gif)

Ah, the GIF. Let's all say it together: GIF, not JIF. Seriously, if you call it a JIF, you can show yourself the door. I'm looking at you, **Kevin**.

GIFs are an interesting format. They are definitely not the best for complex images, since they only support up to 256 colors. But, they excel at displaying text. And they support transparency. And they work everywhere.

Most importantly, GIFs are the only option that allow for animated images. Have you ever received an email campaign showing a devastatingly good-looking model cycling through several different outfits? That's the GIF in action. When you want to add some motion to an email, the GIF is still the best way to do it (we'll talk about motion in email a bit later).

Apart from the color support, the main drawback to using GIFs is that they have the tendency to produce larger file sizes.

Breaking It Down

So, if you are wondering which format to use, here are some handy little guidelines to help you out:

- JPEG Use for any complex images that don't require transparency (e.g. photos, gradients, product shots).
- PNG Use for complex or simple images that require transparency.
- GIF Use for simple graphics like logos or icons, or when you need animation in an email.

Now, let's look at how to incorporate images into an actual email template.

Progressively Enhancing Images

Beyond JPEG, PNG, and GIF, there are a lot of other file formats. While most don't have support across email clients, one format could be beneficial to use: SVG. SVG stands for scalable vector graphics and essentially uses code to build a graphic. SVGs are most commonly used for things like icons, illustrations, and logos. The beauty of SVGs is that they are scalable, meaning they won't lose fidelity when their dimensions are increased, and that they can be manipulated with CSS. You can even animated them using code.

A wonderful trick for using SVGs where supported is to rely on the source set (**`srcset`**) attribute in your **`img`** tag. The srcset attribute allows you to define multiple images to use at either a) multiple screen widths or b) multiple screen pixel densities. An example looks like this:

```
<img src="image.png" srcset="image.png 1x, image@2x.png 2x,
image@3x.png 3x"  />
```

In this scenario, the following images will be displayed:

- **With a device pixel ratio of 1:** image.png will be displayed
- **With a device pixel ratio of 2:** image@2x.png will be displayed
- **With a device pixel ratio of 3:** image@3x.png will be displayed

When **`srcset`** isn't supported, the image will fall back to the file declared in the **`src`** attribute (which we'll look at in the next section). So, if we wanted to progressively enhance our images and use SVG, we could code up the following:

```
<img src="image.png" srcset="image.svg 1x"  />
```

Where supported (generally SVG and **`srcset`** support go hand-in-hand), the email client will display the SVG file. Everywhere else, it's no big deal: the email client will just read the **`src`** file and display it like any other image.

Coding Images

Adding an image to an email is actually surprisingly simple. All it really requires is the img tag. But, there are a few things that you should always keep in mind when adding images.

First, you always need to use absolute paths for your images. You have to host an image on a publicly accessibly server, whether it's on your own or your ESP's. Never use relative image paths. So, instead of doing this, which is common on the web:

```
<img src="/path/to/image.jpg" />
```

You need to do this:

```
<img src="http://site.com/path/to/image.jpg" />
```

That will successfully get your image into a campaign, but it might not look its best. Depending on where you view your email, you'll likely run into a few problems. Let's dive in and see how we can combat these issues.

Problem 1: Images Aren't Properly Sized

Unless you're slicing up a design in Photoshop or know precisely what size you need to save an image, chances are good that just dumping images into an email will result in improperly sized images and broken designs.

While we can easily constrain images within a parent element on the web–without specifying an image's size—we need to explicitly state the width of an image to ensure that it displays at its intended size. Especially

in Outlook, which can't interpret an image's dimensions from the file. However, this isn't typically done in an inline style. Instead, we use the width attribute on the **`img`** element.

```
<img src="http://site.com/path/to/image.jpg" width="600" />
```

This works for most email clients. Occasionally, you may run into a problem with an image's height being improperly interpreted. In this case, you can either add the height attribute or rely on CSS to set the height property to auto.

Attribute Method:

```
<img src="http://site.com/path/to/image.jpg" width="600"
height="200" />
```

Inline CSS Method:

```
<img src="http://site.com/path/to/image.jpg" width="600"
style="height: auto;" />
```

Internal CSS Method:

```
<style type="text/css">
    img { height:  auto !important; }
</style>
```

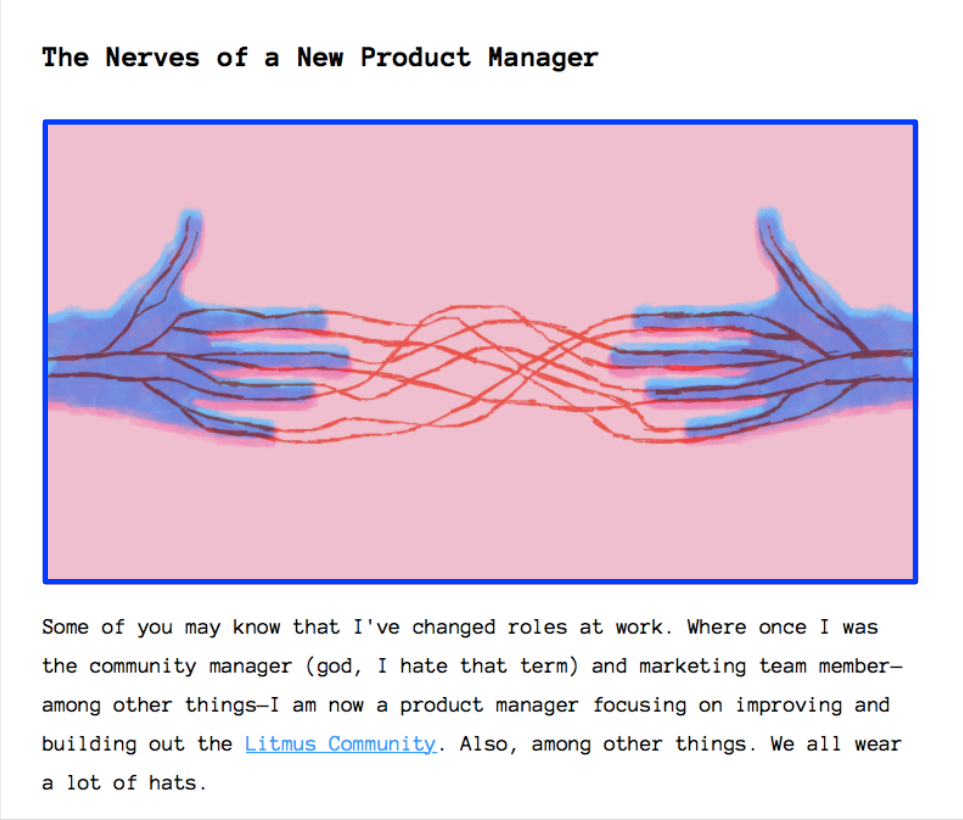
Outlook tends to be particularly bad at sizing Retina images (which we'll discuss in a page or two). If you're running into trouble, just explicitly

declare the width and height attributes on your **img** tag and you should be fine.

At the very minimum, you should always declare a width for your images. So, moving forward, we will incorporate that into our boilerplate **img** tag.

Problem 2: Borders Around Images

In most cases, you'll likely want to link an image out to a landing page. People love looking at images and their used to them being linked since most are on the web. So, it almost always makes sense to wrap your **img** tag in an **a** tag.



When you wrap your images with a link, though, some email clients will automatically add a blue border around the image to denote that it is a link. While this is great from a usability standpoint, it sucks from a designer's perspective.

To prevent blue borders, we add the border attribute to our boilerplate image tag and make the value zero.

```
<img src="http://site.com/path/to/image.jpg" width="600"
border="0" />
```

Now, even when images are linked, we can sleep easy knowing that they won't have any nasty blue borders around them ruining our design.

Problem 3: Gaps Around Images

Some designs call for images to be placed next to or on top of each other, without any space between the two images.

While this is easy enough to pull off, you may notice some email clients displaying a gap between or below the images, making the email look broken. This is due to how some rendering engines (justifiably) handle images in regards to the baseline of text elements in the document.

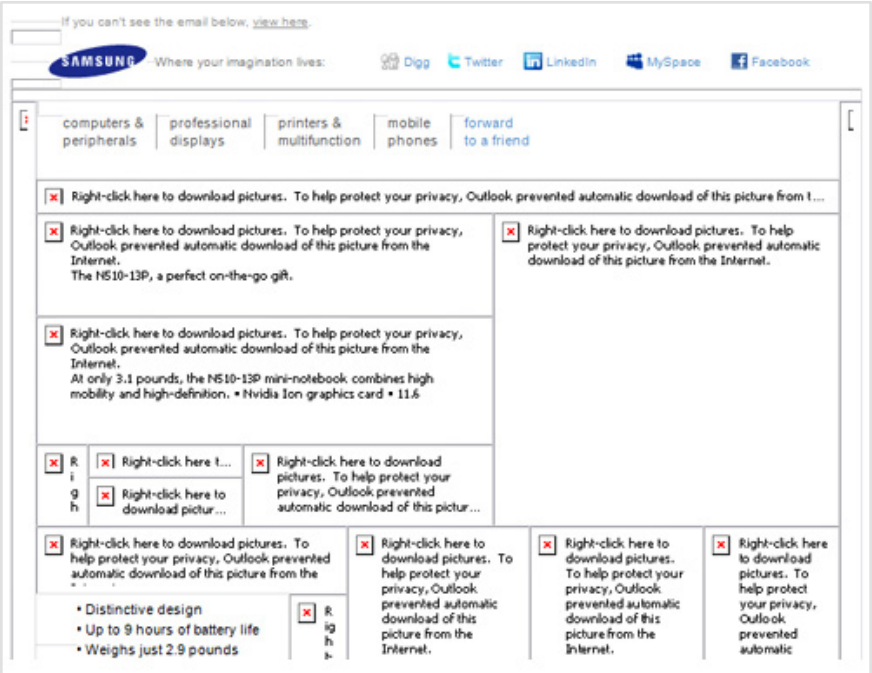
The easiest fix for ensuring your images appear without gaps is to add the following to your **`img`** tag:

```
<img src="http://site.com/path/to/image.jpg" width="600"
border="0" style="display: block;" />
```

That **`display: block;`** rule takes the baseline out of the equation and keeps everything tidy. Just keep an eye on where you're applying it, as not every image needs to be made into a block-level element. If you have something like an icon next to a line of text, making that icon a block-level element will force it down on its own line.

Problem 4: Image Blocking

This is the big one when it comes to using images in email. As mentioned earlier in the book, a lot of email clients (and some users, too) will disable the loading of images in an email until you specifically tell them to download and display images. Ever see this? (Sorry, Samsung!)



That's image blocking in action. Gmail used to be one of the main culprits, but enabled images by default a few years back. Litmus actually did a great study when this happened and found out that **nearly 43% of all Gmail users had images disabled**. While that doesn't necessarily apply to every other email client, we can certainly make some assumptions…

Image blocking is a huge problem for email campaigns. If your message is even partially tied up in an image, subscribers won't see it. That's why I advocate using as much HTML text as is humanly possible. But, nearly everyone will need to have images in an email at some point. So, how do we mitigate the effects of image blocking?

Simple: we use the alt attribute to provide some contextual information about our images.

```
<img src="http://site.com/path/to/image.jpg" alt="Some
Descriptive Text" width="600" border="0" style="display: block;"
/>
```

Now, when our images are blocked, most email clients will display the alternative (ALT) text, providing subscribers with at least some information about what the hell's going on.

The cool thing about ALT text is that we can actually get fairly creative with it. Using inline styles on the img tag, we can style our ALT text so that it fits into our design aesthetic. We can make use of font styling and

background colors to make a design that holds up beautifully even in the absence of images.

```
<img src="http://site.com/path/to/image.jpg" alt="Some
Descriptive Text" width="600" border="0" style="display: block;
color: #888888; font-family: sans-serif; font-size: 24px;" />
```

ALT text should be baked into your image tags, at least for most images in an email. In some cases, it doesn't make sense to use ALT text, like with logos or icons. These images tend to be smaller in size, so trying to cram a bunch of ALT text in that space doesn't make a whole lot of sense. Likewise, those elements usually don't have a textual equivalent, so you don't really need to describe their content to users with images disabled or anyone using a screen reader.

That being said, nearly every image also provides the opportunity to get cheeky and have some fun with ALT text. Some senders use ALT text as a place to include inside jokes, weird musings, and even emoticons. While not everyone sees them, having little easter eggs like that in an email is a killer way to build a relationship with subscribers and just make someone's day.

Making Images Responsive by Default

One thing I've gotten into the habit of doing for all of my images is making them responsive by default. This is something I was turned onto by Julie Ng in her awesome–but now defunct–**newsletter**.

This technique allows for images to resize based on the device screen size. When viewed on larger screens, the images display as normal. When viewed on smaller screens, they fluidly scaled down to fit within their container—without having to do anything special in our code (like applying classes, overriding sizes in CSS, etc.).

By taking advantage of how CSS sizing works, we can build up a cascade of sizes that allow the images to flow. The code looks like this:

```
<img src="http://site.com/path/to/image.jpg" alt="Some
Descriptive Text" width="600" border="0" style="display: block;
color: #888888; font-family: sans-serif; font-size: 24px; width:
100%; max-width: 100%;" />
```

View on CodePen

We still have the 600px wide attribute on the img, but we've also added the width and max-width CSS properties inline, too. With those in place, and set to 100%, the image will reflow based on a few rules:

- When on a large screen, the image will be the full 600 pixels wide.
- When on a screen narrower than 600 pixels, the image will be 100% of the container wide.
- Although 100% wide on narrower screens, the image will never be more than 100% wide, which will prevent scrolling and make it resize based on the screen width.

Nearly all of my images have these properties applied by default, saving me the trouble of having to worry about them on different screen sizes. Not all images need to be responsive by default, though. For things like logos and icons, it makes sense to have them maintain their dimensions across screen sizes. They are usually small and can be hard to visually parse when too small, so keeping them the same size makes sense.

Using Background Images

Another useful technique in email is the use of background images. Background images are applied to HTML elements and allows you to stack other bits of HTML on top of an image. This can be beneficial because it lets you use live HTML text on top of an image, which aids in accessibility and for when email clients disable images by default. You can also do some cool tricks with swapping background images out based on device size.

There are three main ways to include background images in an email: the **`background`** HTML attribute, the **`background`** CSS property, and via VML.

The **`background`** HTML attribute can be applied to a **`table`** element and takes a **URL** as its value.

```
<table background="http://site.com/img/background.jpg">
```

This method works in some email clients but isn't very flexible or reliable, so I wouldn't recommend it. You may see it in legacy email templates, though, so it's worth knowing about.

My preferred method is to use the CSS **background** property on an element, usually a **div** or a table cell.

```
<td style="background: #000000 url('http://site.com/img/
background.jpg');">
```

The **background** property is shorthand and allows for a few values. In the example above, I set a fallback color of black for when the background image doesn't load. This is absolutely vital to ensure that any text within that element is still visible even if the image isn't loaded. In the example above, there could be white text on top of that background image. If no fallback color was declared, then that white text would be unreadable on the default white background.

Another useful property to include on your element is **background-size**. The **background-size** property dictates how the image is scaled to fit the element in which it is contained. It takes a few values like **cover**, **contain**, or individual values. I'd wager that the most common value used is **cover**, which makes the background image cover the dimensions of the containing element.

The last method for including background images is by using VML coupled with HTML. This method is commonly referred to as the

bulletproof background approach and was popularized by Stig Morten Myre, the same guy that created the bulletproof buttons approach mentioned in the last chapter. The code looks something like this:

```
<div style="background-color:#7bceeb;">
  <!--[if gte mso 9]>
  <v:background xmlns:v="urn:schemas-microsoft-com:vml"
fill="t">
    <v:fill type="tile" src="https://i.imgur.com/YJOX1PC.png"
color="#7bceeb"/>
  </v:background>
  <![endif]-->
  <table height="100%" width="100%" cellpadding="0"
cellspacing="0" border="0">
    <tr>
      <td valign="top" align="left" background="https://
i.imgur.com/YJOX1PC.png">
      </td>
    </tr>
  </table>
</div>
```

It's not as complex as the bulletproof buttons code, but it's still adding to your code and maintenance costs. The benefits of this approach is that it's more widely supported. Writing that code can be annoying though. Thankfully, Stig and Campaign Monitor have a handy tool to generate it for us at **Buttons.cm**.

Regardless of which approach you decide to use, I would just remind you to always set a fallback color for your container and background image so that your text and message is always readable.

Retina Images

With the introduction of the iPhone 4, Apple popularized the Retina display. Retina displays pack a huge number of pixels into a screen, far more than traditional computer displays. This increased pixel density makes the screen exceedingly sharp and clear, so much so that the human eye can no longer distinguish individual pixels.

One of the side effects of a Retina display is that images which are not optimized for them appear to be fuzzy or blurred. This can get frustrating, especially for notoriously picky designers. The way to get around this is to use higher resolution images and scale them down to the appropriate size using code.

As an example, let's say we have a photograph that needs to be 600x400 pixels. Normally, we would export that image from our graphics editor at that size. To make it appear crisp on Retina displays, we would instead save it out at least twice the intended size. In this case, we would export it at 1200x800 pixels.

Then, in our code, we would just use the 600x400 sizing:

```
<img src="http://site.com/path/to/retina-image.jpg" alt="So
Crisp" width="600" height="400" border="0" style="display:
block;" />
```

Now, that picture looks absolutely beautiful on Retina displays.

The major drawback of using Retina images in an email is that the increased image dimensions equate to an increased file size, too. For mobile users with limited data plans, this can be a very real concern. There are typically two methods that can help out if you're concerned about image file sizes (which you should be).

Optimizing Images

One way to keep file size down, not only for Retina images, is to optimize and compress your images using software. While most designers play with the quality setting in Photoshop, applications built specifically for compressing images are typically more effective than whatever Photoshop can do.

It's good practice to run your images through an optimization tool before uploading them to your server. There are a variety of tools available, my favorites being **ImageOptim** and **JPEGmini**. If you're using Retina images, running them through one of these is a great way to keep your mobile subscribers happy.

Using Compressive Images

Another solution for reducing the file size of a Retina image is to use what are referred to as compressive images. Compressive images aren't a new

file format, but a way to export a JPEG to keep the file size small while still retaining good quality when scaled down.

Essentially, you would have an image that is at least four times its intended size. This enormous image is then saved with an extremely low quality setting in Photoshop. Even though the large image looks terrible when viewed at full size, when it is scaled down in the code, it retains its detail—all without the added file weight.

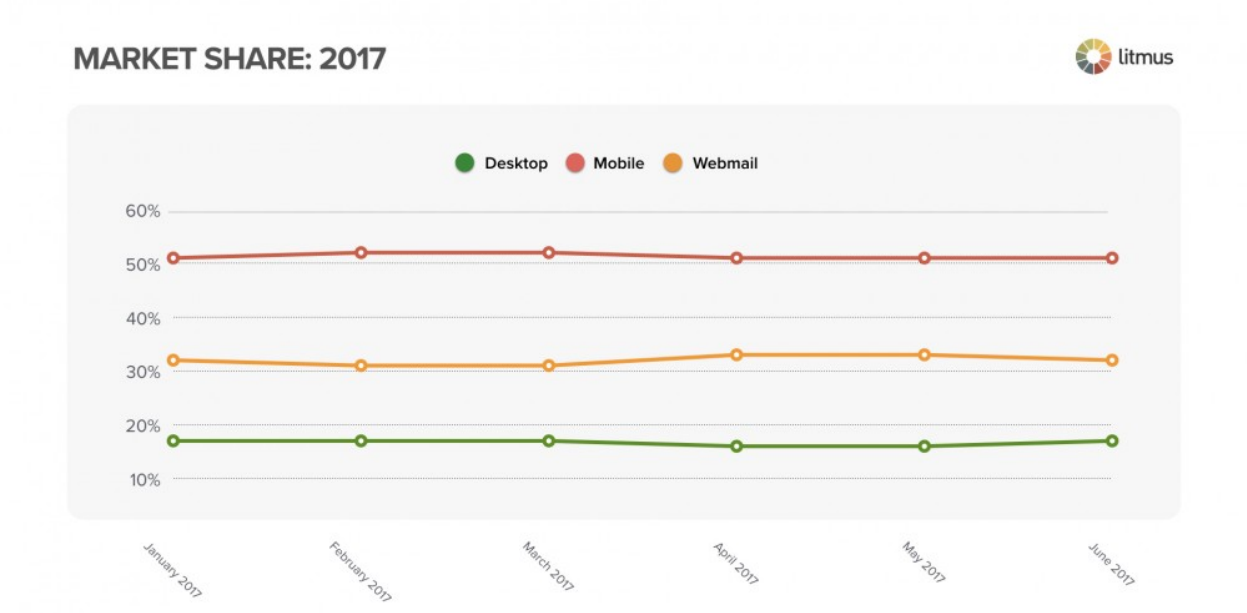
Chapter 6

Understanding Mobile

Chapter 6

Understanding Mobile

Take a look at your jeans. Notice anything? That faded rectangle permanently branded into your favorite pair of Levi's? Take note: that's the most important development in email marketing any of us are likely to see. Hell, it's likely the most significant technological innovation most of us will ever see, apart from the invention of the World Wide Web.



Mobile opens account for half of all opens. ***Source***

Somewhere in the mid-2000s, mobile got big. Like, really big.

Overnight, it seemed as if nearly everyone had a smart phone. And, with the introduction of the iPhone in 2007, an increasing number of those smart phones used touch as their main interaction paradigm.
For email designers, the shift to mobile has three main implications:

1. More people are checking their email on mobile devices, whatever "mobile device" means. The context in which they are viewing emails is always in flux. It's no longer the case that they are sitting at a desk when checking email.
2. Every year, those devices evolve. Screen dimensions and device sizes are fragmenting at an alarming rate. And new email apps are constantly entering the market, each with their own rendering quirks.
3. The idea of a "click" no longer applies. Most of these mobile devices rely on touch. The way we talk about interactions in email needs to change, along with the way we approach designing our emails to account for touch interactions.

Regardless of your industry, it is no longer acceptable to ignore mobile email users. Even if an industry has a low percentage of mobile users, they still exist. For example, at Litmus, our mobile open rates are traditionally very low (around 10-15%). Most of our opens are on desktop. I imagine that's the case for most B2B companies. Still, our subscribers expect our emails to work on their mobile devices, and so will yours.

So, if mobile is so important, how the hell do we actually handle these challenges?

Scalable, Fluid, Whatever...

Email designers typically take one of four approaches when dealing with mobile:

1. Do nothing.
2. Make their emails mobile-aware.
3. Make their emails fluid.
4. Make their emails responsive.

I bet you can guess how well that first option works, huh? While a few industries will likely stagger on for some time without rethinking their “do nothing” approach, it's bound to catch up to them sometime. And when it does, they'll be bleeding subscribers–not to mention profits.

So, let's take a look at the other three options.

Mobile-Aware

When an email designer builds a mobile-aware (or scalable, as it's sometimes called) email, they're basically saying, "I'm not going to do anything special in my code, but I'll still make this email usable in a mobile context."

Mobile-aware emails use a single, fixed layout for every environment. However, unlike the “do nothing” approach, mobile-aware has mobile users in mind from the outset. Mobile-aware emails typically use large text, large buttons, simplified layouts, and shorter copy to ensure that the design is both readable and usable on desktop and smaller devices.

Even though it looks like the designer is being lazy, it's actually a great approach. Without having to change their coding methods, they can still improve the experience for mobile users by doing nothing more than making sure copy is easy to read and buttons are easy to interact with when an email client scales the campaign to fit a screen.

However, this approach doesn't offer much in the way of flexibility. It'd be nice to further tailor our campaigns, so let's kick things up a notch.

Fluid

Fluid emails are very similar to mobile-aware in the sense that they use the same layout for every environment. However, they make that layout more flexible by using tables that allow the content to flow to fill nearly any screen size.

A fluid email uses **`width="100%"`** on tables to make them flow across screen sizes. This works a treat on mobile devices, but on desktop clients, it can sometimes get unwieldy. Without any constraints, emails can become grossly wide, making blocks of text difficult to read. And, when

images are included, they can often look awkward and even lose their context when they slide away from related text.

There are a few things we can do to remedy this situation, but that's diving into…

Responsive

Responsive emails build on the previous two approaches by adding an extra element of control that makes designing for mobile a hell of a lot better.

Like mobile-aware, responsive emails account for mobile users by making text, buttons, and images easy to read and interact with on any device. And, like fluid emails, responsive emails use fluid tables and images to keep things flowing no matter the screen size.

However, responsive emails aren't constrained to a single layout. Using CSS media queries, responsive emails can fine-tune nearly any element in a design to make it better suited to mobile use. Element sizes can be adjusted, layouts can be changed, and, in some cases, components can even be selectively shown or hidden for mobile subscribers.

It's the most powerful approach available to email designers, and the one we'll spend the next chapter looking at in detail.

There are a few additional approaches to handling mobile emails, which will be discussed later on in the book. While they are extremely interesting and can be very useful, they tend to be a bit more complicated. The responsive approach discussed in the next chapter is a good foundation on which to understand all other approaches, so that's what we'll be spending the most time on.

Chapter 7

Responsive Email Design

Chapter 7

Responsive Email Design

Hopefully you've heard of responsive web design. Fluid, flexible, liquid–whatever you want to call them–layouts have been around since the early 2000s. But, it wasn't until Ethan Marcotte published his **groundbreaking article** on A List Apart in 2010 that responsive web design as we know it today was born. Over the course of the article, and 2011's **book of the same name**, Mr. Marcotte codified the three tenets of responsive web design:

1. Fluid Layouts
2. Fluid Images
3. Media Queries

Despite all of the differences between the web and email, most responsive email design works using the exact same principles. While there are different techniques (which will be discussed in the next chapter), edge cases, and the occasional hack, we're still relying on fluid layouts and images, manipulated via media queries, to optimize emails where we can.

This chapter is all about implementing those three techniques in email. By understanding these three techniques, we'll be building some

foundational skills that make understanding the more complicated techniques found in the next chapter easier.

Media Queries

Since enabling fluid layouts and images hinge on using media queries, that's what we will look at first.

Media queries are part of the **CSS3 recommendation** and are, essentially, a logical operator for toggling styles based on some criteria which you specify. Sound complicated? They're not. Here's what one looks like:

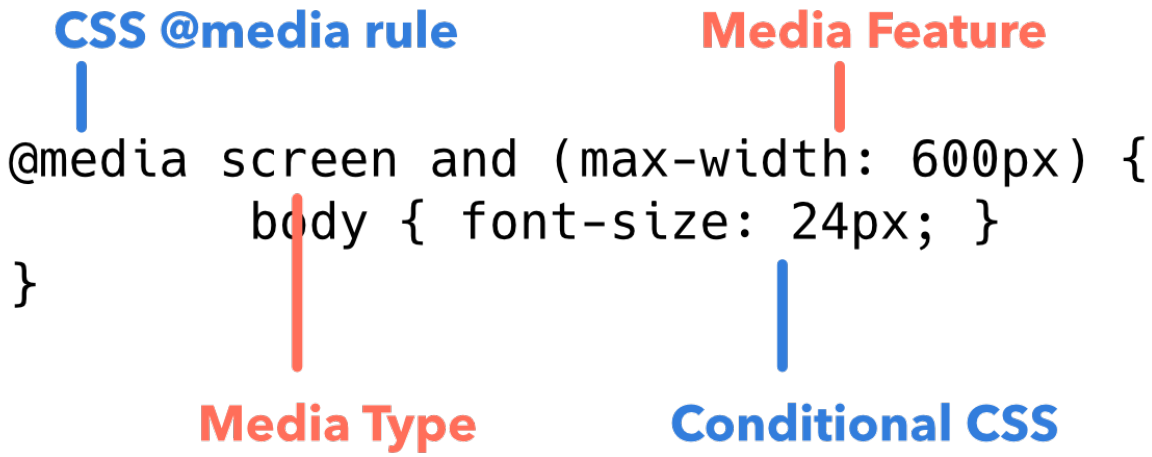
@media screen and (max-width: 600px) {}

Media queries will always live in a style block in the head of our emails.

Let's break that down.

We always start the media query by using the **`@media`**, or at-rule, followed by a media type, in this case **`screen`**. The media type can accept one of four values: **`all`**, **`print`**, **`screen`**, and **`speech`**. Since we're dealing with designing for devices with a screen, we'll almost always use the **`screen`** media type. If you think your subscribers are likely to print your email, you can use styles declared in the **`print`** media query to adapt your email layout for friendly printing.

Next, we can declare any number of media features. Media features allow us to test for certain conditions. In the example above, we are checking to see if the width of the document window is 600 pixels or below. If it is, then the media query is evaluated to be true. When true, any CSS written inside the curly braces of the media query will be applied to our document.



Once understood, media queries become extraordinarily powerful. They allow us to move beyond the desktop and target mobile clients, feeding those clients styles that improve our campaigns on various screen sizes. You can even use media queries to target specific email clients and browsers, which can be helpful when progressively enhancing your email campaigns with more advanced techniques that don't work everywhere.

When it comes to targeting different devices, there are a number of media features which are useful. While there are a dozen or so universally supported media features (and a hell of a lot more browser-specific media features), most email designers rely on the **`width`** and **`device-width`** media features for targeting devices of certain sizes. Both can be prefixed with either **`min-`** or **`max-`**, which allow you to target ranges of screen sizes.

Using **`min-`** and **`max-`** depends on your approach to email design.

There are two concepts at work: building mobile-first or building desktop-first. Using **`min-`** assumes that you are building mobile-first i.e. your inline styles are meant for mobile devices and anything within the media query is used to enhance the design in desktop environments. Using **`max-`** implies the opposite. Due to the limited support of media queries in email clients, especially desktop clients, it's typically safer to rely on **`max-`** and build desktop-first. Then, within our media queries, we can use styles to enhance our campaigns for mobile users.

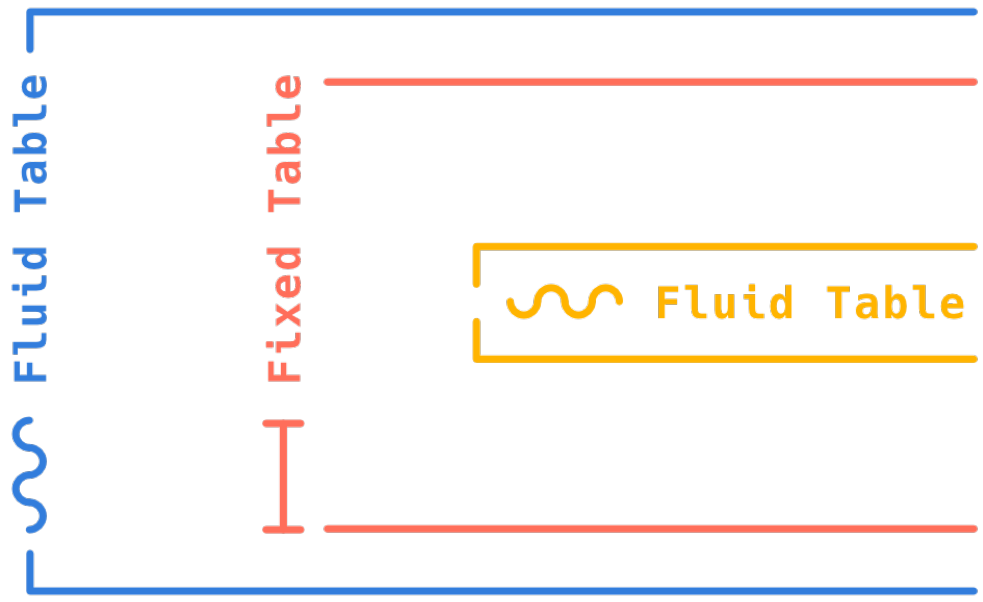
To keep things simple when starting out, media queries in this book will take the following form:

```
@media screen and (max-width: 600px) {
  ...STYLES HERE...
}
```

We use **`max-width`** with a fairly generic value of 600 pixels. Any devices with a screen narrower than 600 pixels will then see and render our defined styles. This basic media query is all we need to get started with implementing the next two components of responsive emails: fluid layouts and fluid images.

Fluid Layouts

If you recall from Chapter 1, we used nested tables in a very specific manner when building our email structure. We had a fluid outer table followed by a fixed-width table to constrain our email content's dimensions, inside of which we again used a fluid table to house any actual content.



The outermost and innermost tables are already fluid, leaving us one, fixed-width table to worry about. On mobile, we want to be able to target any fixed-width tables and force them to be fluid. To accomplish this, we rely on the class HTML attribute to provide a hook with which to target that table. We give the table a class name for referencing later:

```
<table border="0" cellpadding="0" cellspacing="0" width="100%">
  <tr>
    <td>
      <table border="0" cellpadding="0" cellspacing="0"
width="600" class="fluid-table">
        <tr>
          <td>
            <table border="0" cellpadding="0" cellspacing="0"
width="100%">
              <tr>
                <td>
                ...CONTENT...
                </td>
              </tr>
            </table>
          </td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

Here, we are calling that table **`fluid-table`** since, well, we want that table to be fluid. You can name your classes whatever you want. I prefer using class names that are descriptive of their purpose, so that it's easy to revisit a template and immediately know what is going on in the code. I

also prefer to hyphenate my class names. Some people use underscores or camelCasing, but since we're dealing with CSS, in which properties are hyphenated (e.g. **`font-style`** and **`background-color`**), I like to keep things consistent. I'm admittedly not the best at always following this convention, but I'm trying to get better.

Now that we have that class applied to our fixed-width table, we can select that class in our media queries and force it to be a fluid, 100% wide table on anything below 600 pixels wide.

```
@media screen and (max-width: 600px) {
    .fluid-table { width: 100% !important; }
}
```

Since we're overriding an inline attribute, we use the **`!important`** declaration to force the width. Now, our table flows nicely between screen sizes, getting us one step closer to responsive emails.

Fluid Images

Fortunately for us, we're already using responsive images by default. Back in the chapter on images, we applied the width attribute to our img tag, along with both the width and max-width CSS properties to create fluid images.

However, if we wanted to further target our images, we can use the exact same technique that we just applied to our tables. Simply apply a class to

the image, then target that image in your media query to override any inline styling.

There is something to be said for art direction and responsive images. This approach doesn't take into account the content of an image. However, scaling an image down sometimes leads to problems with the information not being easily readable in the smaller image.

Think about an image with text—when scaled down too far, the text becomes unreadable. In these cases, it can be useful to swap out images instead. The typical approach is to have two images in the email and selectively hide and show those images based on screen size. Alternatively, you could use the **`srcset`** attribute to declare multiple images and image sizes in a single **`img`** tag.

```
<img src="small.jpg" srcset="medium.jpg 1000w, large.jpg 2000w"
alt="">
```

In the example above, taken from CSS-Tricks, you can see both the **`src`** and **`srcset`**, or source set, attributes defining multiple images. Source set also needs width dimensions applied with those images to know which breakpoints to use for which images. This is a very handy way to handle responsive images, but doesn't work in most email clients.

Adjusting Other Elements

While using media queries to make our tables and images fluid technically gets us to responsive emails (in most clients), there are a lot of times where selectively adjusting properties of an element can be beneficial on mobile. In these cases, we use the same approach as before: target an element with a class and override any properties using CSS.

There are a number of ways to improve the mobile experience. In the following sections, we'll look at three of the most common and helpful.

Increasing Text Size

Trying to read small text on a mobile device sucks. It strains the eyes and frustrates users, leading them to close, delete, and even unsubscribe from your campaigns. If you are smart, you are using a large baseline font-size for any text in your email. Somewhere around 14-18 pixels is a good place to start. Even on email clients that don't understand media queries and just scale an email down, this text is still readable. But, it's sometimes a good idea to bump things up even more.

Let's say we have some copy in a paragraph.

```
<p style="margin: 0 0 20px 0">This is some copy in a
paragraph.</p>
```

Using our same approach, we can target that paragrah with a class name, in this case, **mobile-copy**.

```
<p class="mobile-copy" style="font-size: 16px; line-height:
18px; margin: 0 0 20px 0">This is some copy in a paragraph.</p>
```

Then, in our media query, we adjust both the **font-size** and **line-height** to our liking.

```
@media screen and (max-width: 600px) {
    .mobile-copy {
        font-size: 20px !important;
        line-height: 32px !important;
     }
}
```

Now, our text is comfortable to read on both desktop and mobile screens. The same can be applied to headings, buttons, disclaimers—pretty much anything. It's a good idea to keep your class names specific enough to tell you what they are doing, but general enough to be applied to multiple elements in a design. I like using classes like **mobile-copy**, **mobile-heading**, or **mobile-button-text**, but to each their own. If you want to keep your code really clean, and are using semantic elements, you can target the elements themselves. Just replace the class names in your media query with whatever element you are targeting, like **h1**, **h2**, **p**, and so on.

Increasing White Space

Similar to text size, white space often needs adjusting on mobile devices. The same table cell padding on desktop rarely works perfectly on smaller screens. Once again, targeting and overriding works like a charm.

Since we use **`padding`** on table cells, we can target those with appropriate class names:

```
<td align="center" class="mobile-pad" style="padding: 20px 10px 20px 10px">
```

Then, we adjust that padding to taste:

```
@media screen and (max-width: 600px) {
    .mobile-pad {
        padding: 40px 20px 40px 20px !important;
     }
}
```

Using this approach allows you to fine-tune your design and make it not only more functional, but more beautiful on mobile devices, too.

Hiding Content

Sometimes, you need to hide something on a smaller screen. Without getting into too much of a philosophical debate on whether or not you

should hide something, let's get to the practicalities of how to actually do the hiding.

A good example of this might be visible preheader text at the top of an email. It's not vital information and can take up valuable screen space (even though the fold in email does not exist) on mobile devices. So, for email clients that support media queries, we can hide it. Since we encapsulate all of our content in its own table cell, we can use a generic class name of hide which is then applied to any table cell that needs to be hidden.

Then, in our CSS, we use the **`display`** property to accomplish the hiding:

```
@media screen and (max-width: 600px) {
    .hide {
        display:none !important;
    }
}
```

While this works beautifully in a lot of email clients, some of them will still display that content. You definitely run the risk of having subscribers seeing something out of context, which is why I recommend only hiding content that is secondary in nature. This same method can work for the 'art directed images' problem described above, but that is even more volatile, since you run the risk of having doubled up images for some subscribers. Just test things out, hack around if needed, and see what works for you audience and campaigns.

Responsive Support

You may have noticed that I make a few caveats when talking about responsive email design. Using media queries and CSS targeting works in a lot of cases, but will absolutely fail in some email clients. So, where does support currently stand?

At the time of writing, support for the techniques described above in major clients looks like the below (the key being media query support):

Email Client	@media Support
Android/Samsung	✔
iOS Apple Mail	✔
Windows Phone 8.1	✔
Gmail App	✔
Gmail	✔
AOL Mail	✔
Yahoo! Mail	✔
Outlook.com/Office 365	✗
Outlook 2007-2016	✗
Outlook iOS/Mac	✔
Apple Mail	✔

As you can see, there are a few omissions. Still, these techniques are pretty widely supported, especially when you consider worldwide email client market share. What techniques you use should absolutely be determined by where your audience opens your emails. If you notice that a lot of your subscribers open on iOS Mail.app, then you can safely use the responsive techniques above (along with plenty of fancy CSS, too). If you're not sure where your subscribers are opening, using something like **Litmus Email Analytics** is a must.

But, what can we do for email clients that don't support this traditional responsive approach? We'll see in the next chapter.

Chapter 8

Different Layout Approaches

Chapter 8

Different Layout Approaches

While the traditional responsive approach we saw in the last chapter works well across *most* email clients, and is easy to learn—especially if you're coming from a web background—it might not always be the right choice of technique for building your campaigns. Over the years, a number of different techniques have been developed to address various issues with email clients.

Although I'm going to go over four different layout approaches in this chapter, I'll only look at the first two in-depth. The other two approaches are interesting but, in my opinion, aren't worth the effort to use in most cases. That could change in the future as email clients evolve but, for now, we'll just learn the basics of those approaches.

Now, onto the first two: hybrid coding and the table of your dreams…

The Hybrid Coding Approach

The hybrid coding approach is an alternate way to set up the structure of your campaigns that works well even in the absence of media query support. It was largely pioneered by **Fabio Carneiro** and **Nicole Merlin**, and popularized by **Mike Ragan**, three email developers that deserve a

beer or cup of coffee if you ever run into them. It's sometimes referred to as the "spongy" approach or the "ghost tables" technique. Whatever you call it, here's how it works.

All Fluid Layout

The crux of the hybrid coding approach is that it uses entirely fluid tables when building the initial structure. It gets rid of the fixed-width container table that we're used to in the traditional responsive approach. Since everything is fluid by default, we don't need to change width values using media queries, and things just naturally work in clients that don't support media queries. However, this leads to the problem of constraining the width, since we don't want insanely wide emails on large screens.

An example of that table structure can be found below.

```
<table border="0" cellpadding="0" cellspacing="0" width="100%"
role="presentation">
  <tr>
    <td>
      <table border="0" cellpadding="0" cellspacing="0"
width="100%" role="presentation">
      </table>
    </td>
  </tr>
</Table>
```

Since that container table has **`width="100%"`** applied, it will flow to fill the available space in the email client viewport. Let's see if we can fix that.

Constraining Widths

To constrain a table width, the hybrid coding approach uses the **max-width** CSS property, which allows you to constrain an element to a specified width while keeping it fluid below that width. This property is applied as an inline style on the table. For our purposes, we'll constrain the table to a maximum width of 600 pixels.

```
<table border="0" cellpadding="0" cellspacing="0" width="100%"
role="presentation">
  <tr>
    <td>
      <table border="0" cellpadding="0" cellspacing="0"
width="100%" role="presentation" style="max-width: 600px;">
      </table>
    </td>
  </tr>
</table>
```

Now, in most clients, that table will be properly displayed at 600 pixels. On screens that fall below 600 pixels wide, then it will take up one hundred percent of the viewport and flow to fill the screen. All without media queries.

Dealing With Outlook

I wish this was all that was needed to get emails working everywhere, but we still have one major problem: Microsoft Outlook. None of the desktop

Outlook clients (except Outlook for Mac, which is essentially a wrapper for WebKit and supports most HTML/CSS) support the **`max-width`** property, which leaves us in a pickle.

Fortunately, we can easily target Outlook clients and provide code which is supported.

Microsoft Conditional Comments

Starting with Outlook 2007, Microsoft updated their email clients to use the Microsoft Word rendering engine for displaying content. While this has benefits for Microsoft Office users and people in the Windows world, it was a disaster for email marketers. Outlook previously used Internet Explorer (which was *built* to render HTML and CSS) to render emails but, with Word, there was a massive drop in support for HTML and CSS. Word was never built to properly display HTML and CSS, and we've been left picking up the pieces ever since.

Although the change to Word created a lot of problems, there was one major trick that helped ease the pain for email developers. Microsoft Office supports targeting through code by using conditional comments. Using HTML comments, we can add a bit of text to tell Word to read the content inside of those comments and use it in our emails.

Here's what that looks like:

```
<!--[if gte mso 12]>
```

```
    Only Outlook 2007+ will understand and display this content.
<![endif]-->
```

We use the same format as any HTML comments, but within that we add square brackets surrounding some conditions. We use the **`if`** and **`endif`** logic to make Outlook evaluate the statement that follows. The **`gte`** part stands for "greater than or equal to", **`mso`** means "Microsoft Office", and the **12** is equal to the Outlook 2007 version of the email client. There are a number of values you can test against to target specific versions of Outlook if needed.

- **lt** - less than
- **lte** - less than or equal to
- **gt** - greater than
- **gte** - greater than or equal to
- **Outlook 2000** - Version 9
- **Outlook 2002** - Version 10
- **Outlook 2003** - Version 11
- **Outlook 2007** - Version 12
- **Outlook 2010** - Version 14
- **Outlook 2013** - Version 15
- **Outlook 2016** - Version 16

Knowing all of this, we can now put it to work to help us constrain the widths of our fluid tables in Outlook, giving us the last piece of the puzzle for hybrid emails.

Within our fluid tables, we can provide additional, fixed-width tables that only Outlook will see. These are commonly called “ghost tables” since no other email clients will see them. The **width** of both the table and table cell will be the same as that used in the **max-width** attribute on the fluid tables.

```
<table border="0" cellpadding="0" cellspacing="0" width="100%"
role="presentation">
      <tr>
        <td align="center">
          <!--[if (gte mso 9)|(IE)]>
          <table align="center" border="0" cellspacing="0"
cellpadding="0" width="600">
          <tr>
          <td align="center" valign="top" width="600">
          <![endif]-->
          <table border="0" cellpadding="0" cellspacing="0"
width="100%" role="presentation" style="max-width: 600px;">
            <!-- FEATURED ARTICLE -->
            <tr>
              <td style="padding: 20x 10px 20px 10px;">
                <table border="0" cellpadding="0"
cellspacing="0" width="100%" role="presentation">
                  <tr>
                    <td>
                      <h1>Headline</h1>
                    </td>
                  </tr>
                  <tr>
                    <td>
                      <img alt="hero image" src="http://
placehold.it/1200x600" width="600" border="0" style="display:
block; max-width: 100%; min-width: 100px; width: 100%;">
                    </td>
```

```
                </tr>
                <tr>
                  <td>
                    <p>Lorem, ipsum dolor sit amet consectetur
adipisicing elit. Laudantium dicta ducimus quibusdam, enim
fugiat magnam impedit nulla distinctio excepturi. Molestiae modi
quas aut totam similique suscipit autem deleniti eaque
necessitatibus.</p>
                  </td>
                </tr>
              </table>
            </td>
          </tr>
          <!-- /FEATURED ARTICLE -->
        </table>
        <!--[if (gte mso 9)|(IE)]>
        </td>
        </tr>
        </table>
        <![endif]-->
      </td>
    </tr>
  </table>
```

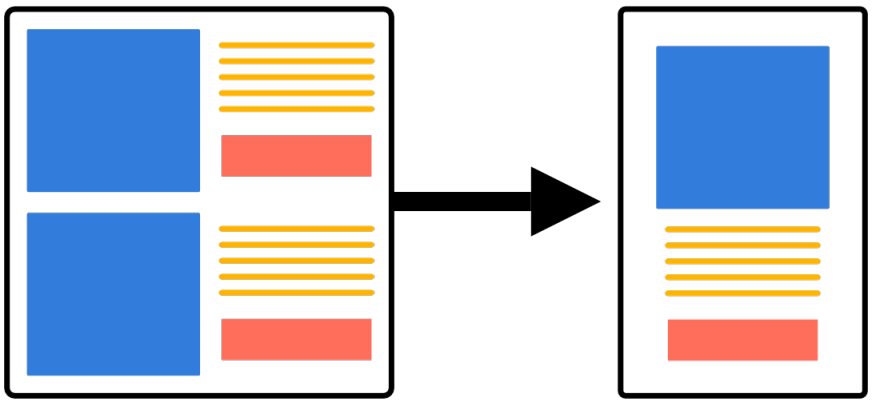
View on CodePen

Now, when are emails are viewed in Outlook, they are properly constrained and display perfectly (at least as perfectly as an email can look in a Microsoft client).

Multiple Columns in Hybrid

Again, the hybrid coding approach is a fantastic way to create responsive emails even when media queries are not supported. However, things can get a bit more complicated when you start introducing multiple columns to your layout.

Since we can't rely on media queries to change the declared width of the columns like in the traditional responsive approach, we have to used fixed widths to keep those columns a particular size and then rely on the fact that those columns will naturally stack on smaller screens.



In code, that seems like it would be easy, but we do run into issues of centering and alignment of the columns. There are a few tricks to getting columns centering properly on mobile, mainly by wrapping column

tables in a **`div`** set to **`display: inline-block;`** with the **`max-width`** and **`width`** set to the same values as the table within. I'm not too keen to get into that long of a code sample here, but you can check out a two- and three-column example of a hybrid email **here** and **here**, respectively.

The major downside with multiple columns in the hybrid approach is that, since those columns are declared as fixed-width elements, they don't grow to fill the screen on mobile devices. You can adapt those columns to become fluid using media queries but, again, it's not supported everywhere.

Although it's not as clean as the traditional responsive approach, it does work everywhere and you get used to the structure and mechanics of hybrid code the more you use it.

The Table of Your Dreams

My second favorite coding approach for emails was first written about by **Mark Robbins**, another email developer that deserves a drink. In **this blog post**, he describes a way to make a simple, responsive email using a single table with three columns. The table is set up so that the center column has a specified width attribute, with the other two having none.

```
<table width="100%" border="0" cellspacing="0" cellpadding="0"
role="presentation" style="table-layout: fixed;”>
  <tr>
    <td style="font-size:0px"> </td>
    <td align="center" width="600" bgcolor="white">
```

```
        <h1>
            New Table Structure
        </h1>
    </td>
    <td style="font-size:0px"> </td>
  </tr>
</table>
```

View on CodePen

You can see that both of the outer columns have a non-breaking space (**` `**) within, which ensures that those cells won't collapse. This effectively forces email clients to behave like web browsers when you apply the **`margin: 0 auto;`** rule to an element, centering that element on the page. Since we're only using the **`width`** attribute set to 600 pixels on that center table, when the screen is smaller that 600 pixels, the content will become fluid to fill those dimensions.

Mark notes that, using this approach, you can still add content in those outer tables to create some interesting designs. Even simple things like adding a **`bgcolor`** to those cells allow for great design options. And, by stacking this table structure and adding multiple outer columns, you can offset the center content, which adds to the creative possibilities.

What I love about this approach is the simplicity of the code. It doesn't use media queries, it uses only a single **`width`** attribute, and—if your layout is relatively simple at least—it doesn't rely on complex nesting or Microsoft ghost tables. If your email is a simple, single-column affair (which I recommend striving for), this is the approach I would

recommend. You can always progressively enhance it, too, by adding in media queries to adjust styles where supported.

More Complicated Stuff

Again, there are other layout options that have been developed over the past few years. While I won't take much time going over them here, I will describe them briefly and point you to some resources if you want to dig deeper. I want to reiterate that these are all absolutely fascinating techniques and deserve consideration. That said, I think the techniques already described in this book are better options for most people and have the benefit of being easier to implement and update over time.

The Fab Four Technique

The Fab Four technique was developed by **Remí Parmentier**, the great detective of the email world. Like the hybrid coding approach, it was developed to address the lack of media query support in some email clients. The Fab Four mentioned in the technique are:

- The CSS **`calc()`** function
- The CSS **`width`** property
- The CSS **`min-width`** property
- The CSS **`max-width`** property

Using these four elements, you can set up a cascading set of rules that determine the width of elements. This allows us to address one of the problems with the hybrid approach, where columns were of a fixed size and often looked awkward on smaller screens.

The width cascade works when the three CSS width properties are used together. You just have to remember these rules:

- If the **`width`** is greater than **`max-width`**, **`max-width`** wins and is applied
- If the **`min-width`** is greater than either **`max-width`** or **`width`**, **`min-width`** wins and is applied

Combined with the **`calc()`** function, which allows us to use some CSS arithmetic to determine the value of the **`width`** property, the Fab Four technique allows for some really cool responsive emails. The main problem with this approach is that the **`calc()`** function isn't supported in some major clients, specifically Lotus Notes (but who cares?), some versions of Outlook, the Outlook web app, and all versions of Yahoo! Mail.

There are some workarounds, but I'll let Remí explain those in his **excellent writeup** of the technique.

Mobile-First

The next method is **Stig Morten Myre's** mobile-first approach. Like the Fab Four approach, it also uses the CSS **`calc()`** function to calculate widths. However, it does so not make use of the **`min-`** or **`max-width`** properties. So, instead of acting like a fluid container, the email switches between two fixed-width sizes: one for mobile and one for desktop.

On top of that, Stig uses a different structure for the email that is more similar to hybrid coding. For the base, he uses a series of nested **`div`** elements with CSS classes to target those elements. In the **`head`** of the email, he then uses a few properties to set up the layout.

The divs are set to either **`display: table;`** or **`display: table-cell;`** depending on their function. These properties allow for elements other than actual table elements to function as tables. Combined with the **`width`** property on those elements, you can set up a table-like structure with actual tables.

However, you probably guessed that this won't work in Outlook. To handle display in Outlook, Stig turns to our old friend: the ghost table. He wraps the layout where necessary with Microsoft conditional tables to a fixed-width layout that works in Outlook.

There's a lot going on in the mobile-first method, but it's definitely worth reading in **Stig's post on Medium**.

All of these layout techniques have advantages and disadvantages. As I mentioned, I'm a big fan of the traditional responsive approach, hybrid coding, and the table of your dreams. But you shouldn't let me make decisions for you. Explore the other approaches described above, research others you stumble across online, and try to come up with your own.

Chapter 9

Animation, Effects, and Interactivity

Chapter 9

Animation, Effects, and Interactivity

So far, we've looked at how to build the typical, static (but responsive!) email campaign. While this will be *most* of what you build throughout your career, sometimes we need something atypical. Something to set our emails apart from a very crowded inbox. That's where adding animation, interactivity, and dynamic content come into play.

In this chapter, we'll look at a few ways to add movement and interactivity to an HTML email campaign. These techniques range from the basic (animated GIFs and simple CSS effects) to the complex (CSS keyframe animations and the checkbox hack). After figuring out how to pull off some cool tricks in our campaigns, we'll see how to make some of that content dynamic in the next chapter. Let's get started.

Animated GIFs

One of the simplest ways to add movement and animation to an email campaign is by using animated GIFs. As we saw in Chapter 5, GIFs are widely supported across email clients and are one of the image formats that supports animation. They are commonplace on the web, so sourcing

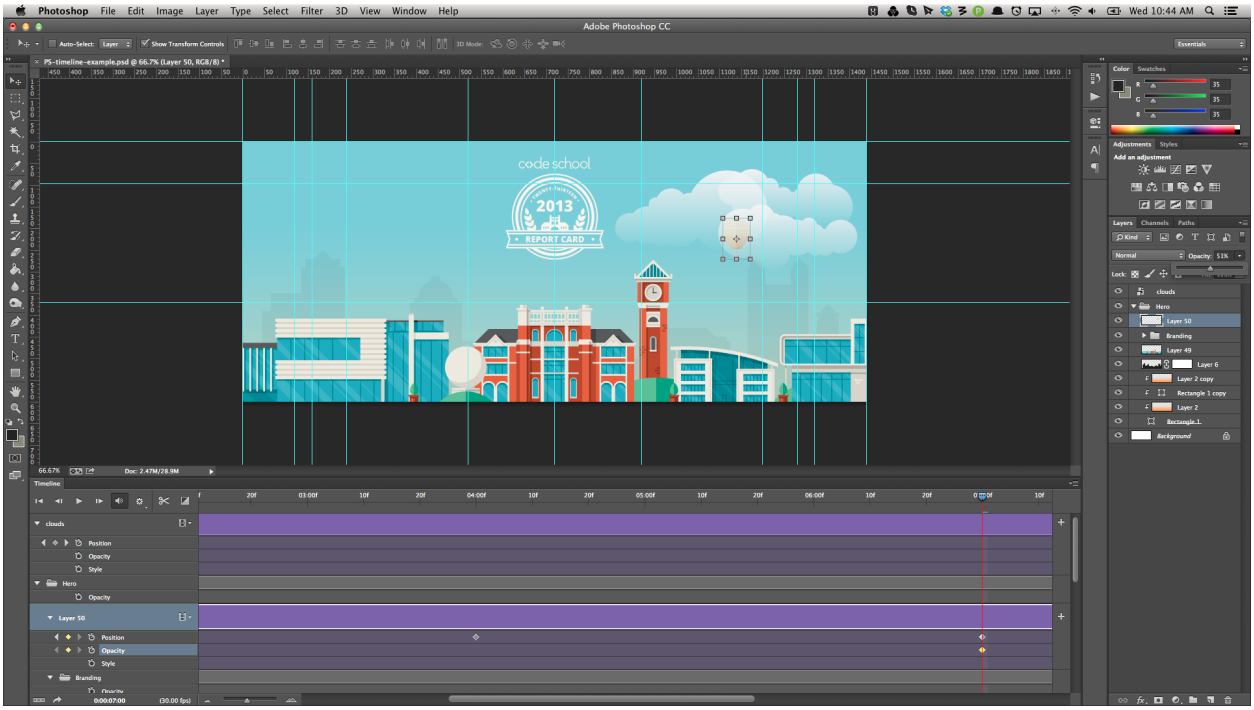
GIFs is relatively easy. Although creating them from scratch can be more time-consuming, they are still one of the easier options to implement in email.

How Animated GIFs Work

In essence, animated GIFs work like a flip book. Within the GIF file, a series of frames (the pages in a flip book) are saved, each one different from the last (in theory, at least). Where supported, an email client or browser will flip through those frames at a relatively high speed. Just like in a flip book, when those frames are flipped through, it creates an animation and tricks the eye into seeing movement on screen. It's the exact same principle as film—we're essentially creating little movies within this single image file.

There are a lot of ways to create GIFs and even more tools out there to help you during the process. However, for the types of GIFs you see in emails, there are two major applications used to create GIFs: Adobe Photoshop and Adobe After Effects.

If you're not familiar with Adobe Photoshop, it's an image editor and graphics creation tool used by countless creators around the world. While most are familiar with using Photoshop to adjust a photograph, create a logo, or mock up an email or website, Photoshop includes a **Timeline** tool that allows you to create frame-by-frame animations.

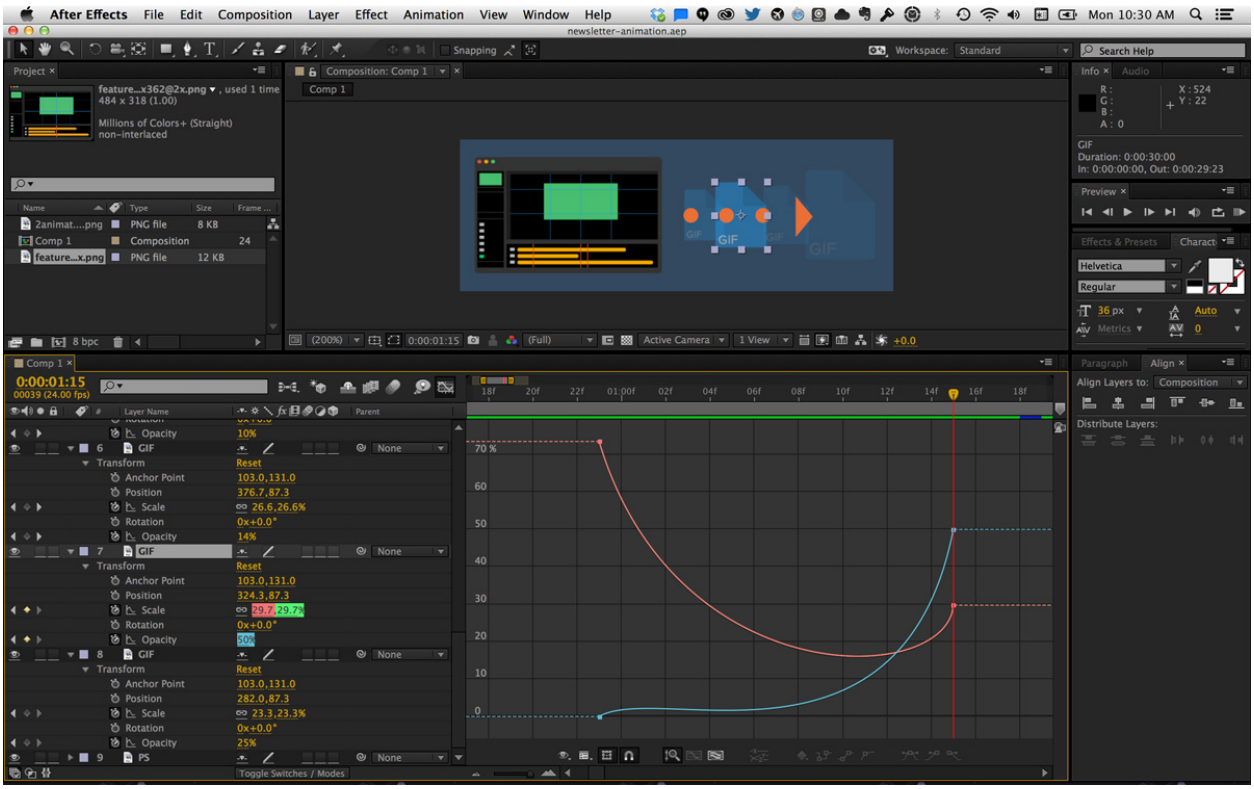


Dan Denney's Photoshop Timeline setup. ***Source***

Using Photoshop's Timeline, you can create individual frames declared within Photoshop's Layers. When a frame is highlighted on the Timeline, any visible layer will be included within that frame. I really like this approach. It allows you to pretty quickly build up an animation just by toggling layers on and off. If you want something always visible within the animation, just keep that layer always toggled on.

It can be somewhat time consuming building animations in this way, but the level of control is fantastic. Plus, by building in Photoshop, you can more thoroughly optimize your animation to keep the file size down, something we'll touch on in just a bit.

Adobe After Effects is a tool for creating motion graphics. Since the entire application is geared towards making animations and manipulating video, After Effects is a great tool for more complicated animations.



It works on a similar principle as Photoshop's Timeline, but is a bit more complex. But, if you're comfortable with After Effects (or don't mind learning a new tool), it opens up a lot of possibilities. Being able to fine-tune animations using curves, add effects to graphics and text, and quickly implement advanced tricks with After Effects' animation presets can be very rewarding.

The one problem with using After Effects is that you will still need to make a trip through Photoshop. After Effects exports videos or image sequences, which you will then need to make into a GIF. In this case, you would import either the video or the image sequence into Photoshop, which will automatically add it to the Timeline. I would highly recommend fine-tuning the imported sequence as much as possible to reduce file size, since this process can yield really large files.

There are other tools out there to help make animated GIFs, but I'll leave it to you to hunt them down and try them out.

Including Animated GIFs in an Email

When it comes to actually getting our animated GIFs into an email, things are extraordinarily easy. Since GIFs are just another image file format, we will use the exact same method we are used to:

```
<img src="http://site.com/path/to/animation.gif" alt="Some Descriptive Text" width="600" border="0" style="display: block; color: #888888; font-family: sans-serif; font-size: 24px; width: 100%; max-width: 100%;" />
```

And with that, we have a responsive-by-default, animated GIF ready to delight our subscribers!

Some Considerations

Animated GIFs are not without their drawbacks. Although they are still the most reliable way to include animation in an email campaign, they aren't supported everywhere. Some email clients will display the GIF, but won't animated it—only the first frame of the animation will be shown. It's usually a good idea to make sure the first frame includes any necessary content for getting your message across to subscribers in these cases.

The other major problem with animated GIFs is that they tend to be very large files. If you aren't optimizing your GIFs, then you can easily create bloated files that weigh 2mb or more. I've seen animated GIFs in emails that are upwards of 6mb in size. For desktop clients, this is rarely an issue. But as more people view emails on their mobile devices, this can create a couple of problems.

The first problem is that the large files are slow to load, which creates a poor user experience. You want to get your point across using a clever, beautifully animated illustration but instead stick your users with a slow, jittering effect that reflects poorly on your brand.

The second problem is that these large files can eat into users' data plans. As most of us know, cellular data plans are very expensive. Forcing our users to download large files is forcing them to use up valuable data on an email campaign, something nearly everyone is loathe to do.

Subscribers would much prefer using that data catching up with friends, browsing online, or watching a video. I'd hate to be the one to push them over their data limit and force them to incur any fines.

So, it behooves us to always optimize the hell out of all of our images, especially animated GIFs. Optimization is a bit outside the scope of this book, but a quick search online will bubble up some interesting articles on the subject. I also touched on optimizing animated GIFs in **this blog post** on the Litmus blog.

CSS Effects and Animations

Another option for creating effects and animation in email is to use actual code. Over the past few years, CSS has come a long way with effects and animation. Techniques that previously relied on Photoshopped images, Flash, or JavaScript can now be accomplished using a handful off CSS properties. In this section, we'll look at some of those properties and how they can be put to use in HTML emails.

Simple Effects

Before we dive into actual CSS animations, I want to talk a bit about using CSS to achieve some simple, but useful effects. CSS has a number of properties that can be put to use to add a layer of visual enhancement to elements in an email.

There are properties that allow us to further style elements like buttons:

- **box-shadow**
- **text-shadow**
- **skew()**

And also properties that allow us to add basic animations to elements:

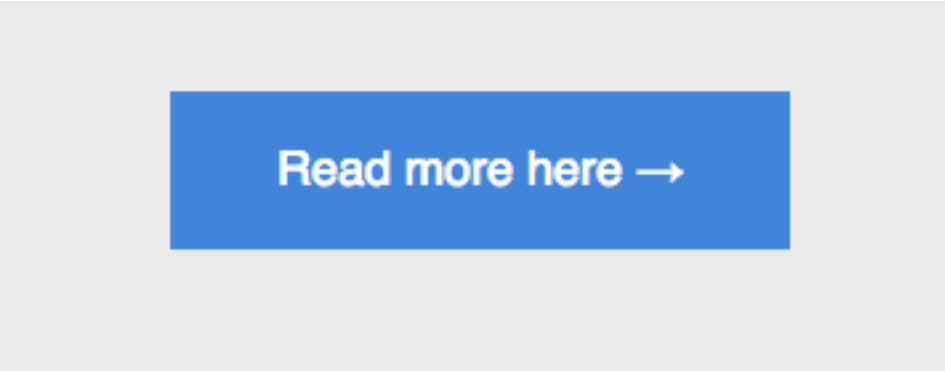
- **:hover**
- **transition**
- **transform**
- **translate()**
- **scale()**
- **rotate()**

Let's take a look at how these work.

When it comes to just manipulating styles with CSS, **box-shadow**, **text-shadow**, and **skew()** can be very effective in email clients that support them. Let's take a look at a simple bulletproof button:

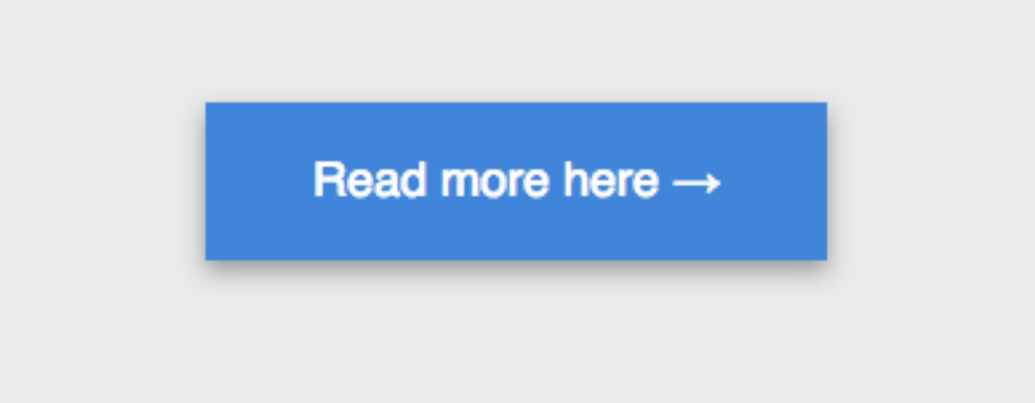
```
<table border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td style="padding: 20px;">
            <a href="https://thebetter.email" target="_blank"
style="font-size: 18px; font-family: sans-serif; color: #ffffff;
text-decoration: none; border-radius: 8px; -webkit-border-
radius: 8px; background-color: #357edd; border-top: 20px solid
#357edd; border-bottom: 18px solid #357edd; border-right: 40px
solid #357edd; border-left: 40px solid #357edd; display: inline-
block;">Read more here &rarr;</a>
```

```
        </td>
    </tr>
</table>
```



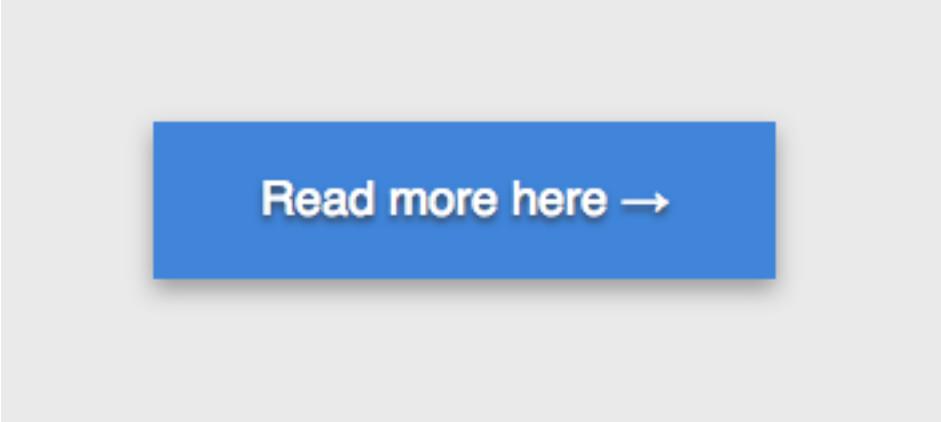
By applying **box-shadow** to that button, we can add a simple drop shadow beneath the button. This can add a level of dimension to the button and visually call it out to subscribers, making them more likely to press that button.

```
<table border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td style="padding: 20px;">
            <a href="https://thebetter.email" target="_blank"
style="box-shadow: 0px 4px 8px #999999; font-size: 18px; font-
family: sans-serif; color: #ffffff; text-decoration: none;
background-color: #357edd; border-top: 20px solid #357edd;
border-bottom: 18px solid #357edd; border-right: 40px solid
#357edd; border-left: 40px solid #357edd; display: inline-
block;">Read more here &rarr;</a>
        </td>
    </tr>
</table>
```



Likewise, we could add the **`text-shadow`** property to the link and improve the styling there, too. This can be especially useful when using light text on a colored background, as it makes that text pop out from the background and can improve readability.

```
<table border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td style="padding: 20px;">
            <a href="https://thebetter.email" target="_blank"
style="box-shadow: 0px 4px 8px #999999; font-size: 18px; font-
family: sans-serif; color: #ffffff; text-decoration: none;
background-color: #357edd; border-top: 20px solid #357edd;
border-bottom: 18px solid #357edd; border-right: 40px solid
#357edd; border-left: 40px solid #357edd; display: inline-block;
text-shadow: 0px 2px 2px #333333;">Read more here &rarr;</a>
        </td>
    </tr>
</table>
```



Finally, we could use the CSS **skew( )** function to shear that element and create in interesting, modern effect. This gets us a lot closer to certain styles that are increasingly used on websites, allowing us to maintain consistent styles across channels.

```
<table border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td style="padding: 60px 0;">
            <a href="https://thebetter.email" target="_blank"
style="font-size: 18px; font-family: sans-serif; color: #ffffff;
text-decoration: none; background-color: #357edd; border-top:
20px solid #357edd; border-bottom: 18px solid #357edd; border-
right: 40px solid #357edd; border-left: 40px solid #357edd;
display: inline-block; box-shadow: 8px 8px 0px #000000; text-
shadow: 0px 2px 2px #333333; transform: skew(-10deg,
-6deg);">Read more here &rarr;</a>
        </td>
    </tr>
</table>
```

The important thing to note with these styles is that they allow us to create beautiful, engaging elements in an email without relying on images. The same effects can be achieved with image-based buttons, but as we saw in Chapter 4, image-based buttons have significant drawbacks. Although these styles don't work in all clients, they fall back gracefully and don't cause any problematic artifacts when they aren't supported. They simply don't display. This is a fair tradeoff for improving the accessibility and usability of our emails.

Perhaps more important, by relying on code-based buttons and elements, we can do some lightweight animation with the other CSS properties mentioned above.

Revisiting our first button from the examples above, we can add target the button in the head of our document and add a **`:hover`** pseudo class to change the look of that button when a user hovers over it.

```
.button:hover {
    background-color: gold !important;
    border-top: 20px solid gold !important;
    border-right: 40px solid gold !important;
    border-bottom: 20px solid gold !important;
    border-left: 40px solid gold !important;
    color: black !important;
}
```

In the code above, we're simply changing the background and text colors of the button to further show subscribers that they can interact with it. One of my favorite CSS properties to add to links is the **transition** property, which allows you to animate between two states of an element.

```
.button {
    transition: all 0.2s ease-in-out;
}

.button:hover {
    background-color: gold !important;
    border-top: 20px solid gold !important;
    border-right: 40px solid gold !important;
```

```
    border-bottom: 20px solid gold !important;
    border-left: 40px solid gold !important;
    color: black !important;
}
```

Looking at the **`transition`** property, you can see a few values declared. In order, they are **`transition-property`**, **`transition-duration`**, **`transition-timing-function`**, and **`transition-delay`**. All of those could be declared as individual CSS properties on that link element, but the shorthand **`transition`** property helps keep our code clean. Here's what each of those does:

- **`transition-property`** sets the name of the CSS property you want transitioned. This can be set to **`none`**, **`all`**, or a specific CSS property like **`color`**.
- **`transition-duration`** sets the length of the transition animation in seconds or milliseconds.
- **`transition-timing-function`** declares how the animation accelerates over that period of time. You can use keywords like **`ease`**, **`ease-in`**, **`ease-out`**, and **`ease-in-out`** to help smooth out that animation.
- **`transition-delay`** allows you to set a delay on the start of the transition in seconds or milliseconds.

Knowing that we can use **`transition`** to animate elements, we can then apply the **`transform`** property to do some cool stuff. The **`transform`** property allows you to change the position, size, and rotation of an

element by using the following functions: **`translate()`**, **`scale()`**, **`rotate()`**.

The **`translate()`** function lets you move the position of an element. If we wanted to have a button move up on hover—giving it a little nudge animation—we could apply the following:

```
.button:hover {
    background-color: gold !important;
    border-top: 20px solid gold !important;
    border-right: 40px solid gold !important;
    border-bottom: 20px solid gold !important;
    border-left: 40px solid gold !important;
    color: black !important;
    transform: translate(0px, -10px);
}
```

The **`scale()`** function lets you scale the size of an element. If we wanted a button to grow larger on hover, this would work:

```
.button:hover {
    background-color: gold !important;
    border-top: 20px solid gold !important;
    border-right: 40px solid gold !important;
    border-bottom: 20px solid gold !important;
    border-left: 40px solid gold !important;
    color: black !important;
    transform: scale(1.2);
}
```

Finally, the **`rotate()`** function allows you to rotate the element around a point of origin. If, for some reason, you wanted your button to spin around on hover, you could use this:

```
.button:hover {
    background-color: gold !important;
    border-top: 20px solid gold !important;
    border-right: 40px solid gold !important;
    border-bottom: 20px solid gold !important;
    border-left: 40px solid gold !important;
    color: black !important;
    transform: rotate(360deg);
}
```

There are other **`transform`** functions available, like **`perspective()`**, but I'll leave those to you to explore.

Like most things in email, these properties and functions aren't supported everywhere. Even something as basic as the **`:hover`** pseudo class isn't supported across all email clients. Still, they are a fantastic way to progressively enhance your email campaigns and add some lightweight animations for your subscribers to enjoy.

Keyframe Animations

When we want to get more serious about animation in email and create more complex visual effects, CSS keyframe animations are the way to do it. Keyframe animations work very similarly to an animated GIF or Photoshop's Timeline. Each animation in CSS requires two things: the

@keyframes rule and the **animation** property used on the element you want to animate.

The **@keyframes** rule is how you declare the frames, or steps, in your animation. A simple animation might look like this:

```
@keyframes square {
  25% { transform: translate(100px, 0px) scale(0.5); }
  50% { transform: translate(100px, 100px); }
  75% { transform: translate(0px, 100px) scale(0.5); }
  100% { transform: translate(0px, 0px); }
}
```

You can see that, like a media query, we open up the keyframes with the **@keyframes** rule. We then give it a name for identifying that animation. Within the curly brackets, you set up the individual steps using either the **from** and **to** keywords or percentages. Within those steps, you declare which CSS properties you want updated and their new values.

In the example above, we have an animation called **square**. There are four steps that animate a box using the **transform** property and both the **translate()** and **scale()** functions. It's called **square** since the animation will translate the box around a square path on screen. At the **25%** and **75%** steps, the box is scaled down to half its size.

In that same example, I'm using the following code for that square:

```
<table align="center" border="0" cellpadding="0" cellspacing="0"
role="presentation" width="100%">
```

```
    <tr>
        <td align="left" style="padding: 50px;">
            <div style="background-color: #357edd; border-radius:
4px; height: 100px; width: 100px;"></div>
        </td>
    </tr>
</table>
```

There's nothing too fancy going on here: just a simple **`div`** that uses inline styles to create a 100x100 pixel, blue box with rounded corners. If you're paying close attention, though, you'll probably notice that there is nothing connecting the **`div`** in our HTML to the **`@keyframes`** animation in our CSS. So how do we actually get an element to animate?

Enter the **`animation`** property. The animation property is how we tie elements to their animations. It is always applied to the element that you want animated, in this case our div.

```
div { animation: 2s infinite square; }
```

View on CodePen

This is a relatively simple example of the animation property. In order, we are declaring the **`animation-duration`**, **`animation-iteration-count`**, and **`animation-name`**. The duration tells the browser or client how long an animation should last, the iteration count says how many times it should repeat (in this case, infinitely), and the name is how we reference the **`@keyframes`** rule. There are other values you can set in the animation property, corresponding to individual CSS animation

properties, but those are better referenced in **Mozilla's Developer Network documentation**.

There are a couple of things to keep in mind when using CSS keyframe animations.

First, you need to specify unique names for your **`@keyframe`** rules. If you have duplicate animations with different properties being animated, the last **`@keyframes`** rule in your CSS is what is used. The same goes for duplicate steps within an animation. It's just like in regular CSS, the cascade dictates that if there are multiple declarations or styles written, the last one wins (all other things being equal).

Finally, and perhaps more importantly when it comes to email, the **`! important`** declaration is ignored within the **`@keyframes`** rule. The **`! important`** declaration is commonly used to override inline styles within an email but, in this case, we want to avoid using it when declaring steps in the animation. Don't worry though, in clients where keyframe animations are supported, things tend to work out just fine.

Although the example above is a basic one, I hope it helped explain how keyframe animations work in CSS. There are a ton of resources online for CSS animations. If you're just looking to add some effects to your CTAs, I'd recommend checking out **Hover.css** for inspiration. If you *really* want to dig into CSS animation, I highly recommend following **Rachel Nabors** on Twitter, checking our **her course** on the subject, or subscribing to her

own **email newsletter** on animation. She also has a **book**, which I hear is excellent but haven't picked up… yet.

Interactivity in Email

The hot trend of the last year or two has been interactivity. Interactive emails allow subscribers to trigger different states within in email, all in the inbox. This can be to trigger things like animations or to selectively display and hide different bits of content in the email. The best way to wrap your head around interactive email is by seeing a few examples. Check these out, I'll wait:

- **American Express Advent Calendar** by Cyril Gross
- **Christmas Tree Decorator** by Kristian Robinson
- **Sonic The Hedgehog** by Kristian Robinson
- **Litmus Builder** Tour by Kevin Mandeville
- **Anything** by Mark Robbins

While all of these emails vary in their design and what elements are interactive, they all utilize the same technique: the checkbox hack. In this section, we'll be building a fairly simple slide carousel, with three tabs that show and hide content based on which tab is selected. But first, let's see how the checkbox hack works.

The Checkbox Hack

The checkbox hack is a technique wherein you use HTML **`input`** elements to trigger different states and actions within an email. Let's look at the **`input`** element to orient ourselves:

```
<input type="radio" name="slides" id="slide1" checked>
<input type="radio" name="slides" id="slide2">
<input type="radio" name="slides" id="slide3”>
```

The **`input`** element has a few attributes which we'll make use of. The **`type`** attribute declares what kind of input it is. For our purposes, we'll almost always use the value of **`radio`** or **`checkbox`**. The **`name`** attribute names that input for when you're submitting forms but, for our purposes, prevents duplicate content within the email. The **`id`** attribute is used to give our inputs unique identifiers for referencing later. And the **`checked`** attribute, which doesn't require a value, is used to set an initial state for the input.

You may be asking why, when the technique is called the *checkbox* hack, we are using the **`radio`** value. While **`radio`** and **`checkbox`** work in the same way (in that they toggle between an on or off state), they differ in that **`checkbox`** allows for multiple inputs to be checked, whereas **`radio`** only allows for a single input at a time to be checked. In most cases, you're going to toggle between two states, or hide and show content in a

specific container. So the **radio** input prevents any weird issues with duplicate content trying to display at the same time.

Along with our three input elements, we will include corresponding labels for each input:

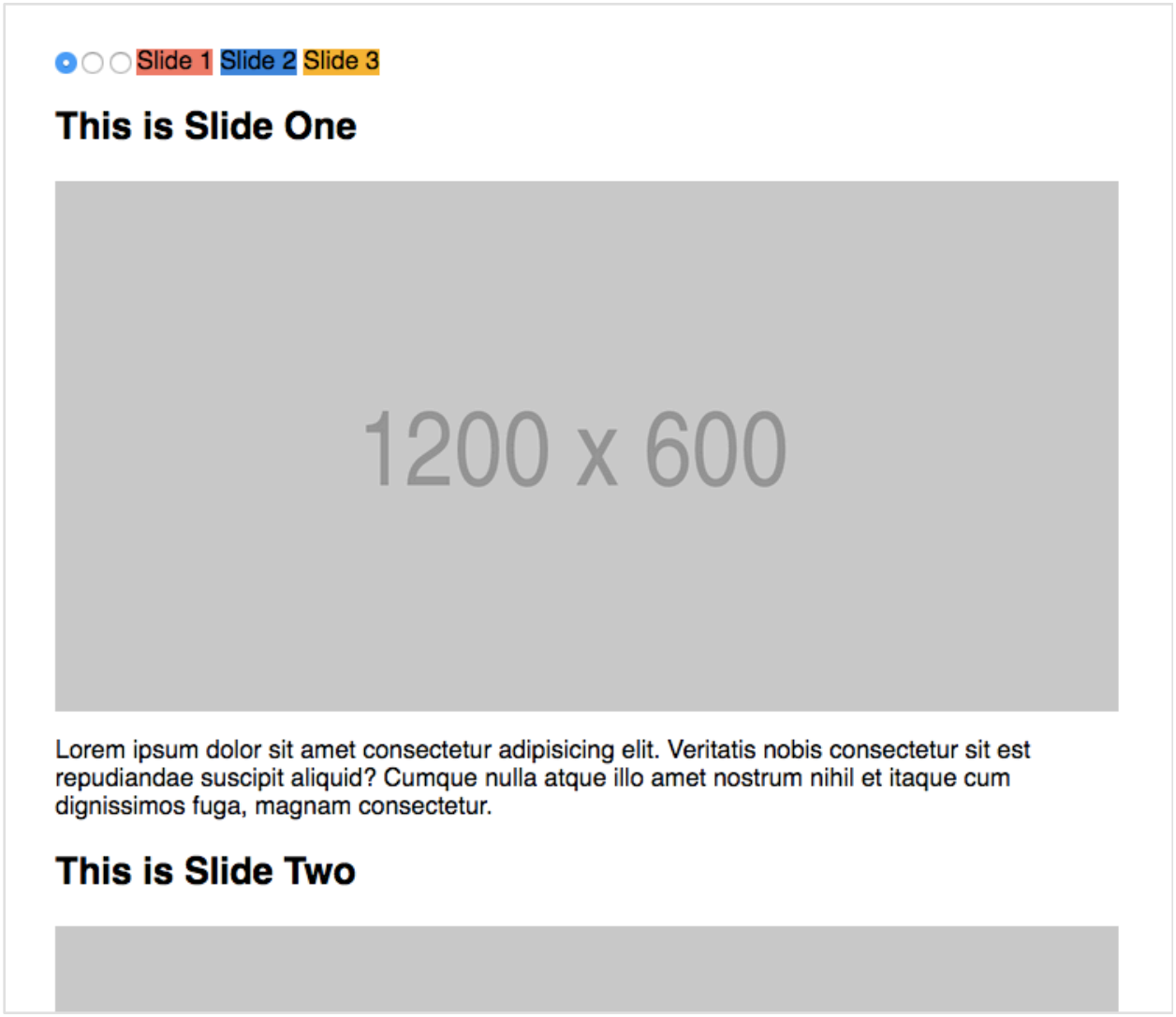
```
<label for="slide1" style="background-color: #ff725c;">Slide 1</
label>
<label for="slide2" style="background-color: #357edd;">Slide 2</
label>
<label for="slide3" style="background-color: #ffb700;">Slide 3</
label>
```

Each **label** element uses the **for** attribute to reference which **input** it goes with. These are populated with the **id** values that we set on our inputs. I also included a background color for each, which goes along with the background colors of each slide. This is purely for styling, not functionality.

The labels will act as the buttons or links to the slide content. When someone presses a label, it will trigger the state of the associated input. Before we move on, though, we need to create content for the slides. We'll contain the content for each slide in a separate **div**, with a **class** of both **slide** and **slideX**—the X equal to the number slide. Here's a base:

```
<div class="slide slide1"></div>
<div class="slide slide2"></div>
<div class="slide slide3"></div>
```

You can populate those slides with pretty much whatever content you want. In **this example**, I've included an heading, image, and paragraph of text. Without worrying about hiding anything (that will come next), our slide carousel looks something like this:



Obviously, that's not what we want. We need our slides to display individually and we'll need to hide those radio inputs. Hiding the inputs is relatively easy. We'll simply target all inputs and set them to **`display: none;`**:

```
input { display: none; }
```

We'll also want to style the labels themselves a bit so that they look more like tabs. So let's add this:

```
label {
  color: #ffffff;
  cursor: pointer;
  display: block;
  float: left;
  padding:1em 0;
  text-align:center;
  width: 200px;
}
```

If we were to preview this now, the slides themselves will still look like garbage. We can take care of that with the following:

```
.slide {
  background-color: #000000;
  clear: both;
  color: #ffffff;
  display: none;
  padding: 50px;
  text-align: center;
  width: 500px;
```

```
}

.slide1 { background-color: #ff725c; }
.slide2 { background-color: #357edd; }
.slide3 { background-color: #ffb700; }
```

I've applied a background color to each individual slide to correspond to the background color of it's matching label. By targeting the **`slide`** class, I've given the slides some structure. If you're previewing this in your browser, you're probably wondering why the slides aren't displaying. What gives?

Looking more closely at that code above, you can see that, just like the inputs, I've used **`display: none;`** to hide those slides. That takes care of the stacking issue we saw in that first screenshot, but how can we get our slides to display one-at-a-time in our email?

That's where this whole checkbox hack comes into play. If you remember, the first of the three inputs had the **`checked`** attribute set. What we need to do now is change the **`display`** property of the slides from **`none`** to **`block`** whenever the corresponding **`label`**—and in turn, **`input`**—has the **`checked`** attribute set. This is accomplished by using the **`:checked`** pseudo class and the general sibling selector (**`~`**) to target the appropriate **`div`** based on its **`class`**. That sounds confusing but is clarified in the code:

```
#slide1:checked ~ .slide1,
#slide2:checked ~ .slide2,
#slide3:checked ~ .slide3 {
```

```
  display: block;
}
```

View on CodePen

When the **`input`** with **`id="slide1"`** is **`checked`**, the **`div`** with **`class="slide slide1"`** is set to **`display: block;`**. The same display switching happens when one of the other labels is pressed, triggering the **`checked`** attribute on that **`input`**. Finally, since we're use radio inputs –and not checkboxes–only one of those can be checked at a time, preventing multiple slides from displaying at the same time. It cleans up nicely and works a treat.

One *very* important thing to note is that, when it comes to the order and nesting of your HTML, your **`label`**, **`input`**, and **`div`** tags need to exist at the same level for this to work. Since we're using the general sibling selector, those elements need to be actual general siblings. If your content is nested within another **`div`** or **`table`**, your interactive functionality will break.

That's kind of a lot to take in, so let's break that down into abstracted steps you can follow for pretty much any interactive email:

1. Create inputs with specific **`id`** attributes for targeting those inputs
2. Create labels to use as buttons with **`for`** attributes referencing the **`id`** of the inputs
3. Create **`div`** containers to house the content you want to show and hide
4. Use CSS to hide the inputs
5. Use CSS to hide the containers and content
6. Style your labels and content as needed
7. Use the **`:checked`** pseudo class and general sibling selector to display content when an input is toggled
8. Test, test, test

Things can definitely get complicated depending on what you're making interactive, but this is the foundation on which all interactive emails are built.

Support and Fallbacks

The thing to remember about interactive emails is that they absolutely do not work everywhere. Even taking email clients out of the equation, not all email service providers support sending interactive emails. The biggest bottleneck for ESPs is MailChimp, which has a massive user base. MailChimp will strip form elements (inputs and labels), rendering that code useless.

When it comes to email clients, quite a few won't support the techniques described above. Perhaps most importantly, Outlook 2007-2016, Outlook.com, and Office 365 won't work. Other clients like Yahoo! And AOL (Oath?) have only limited support. And, keep in mind that email clients (especially webmail clients) are in a near constant state of development, so support can change any day.

All that being said, you shouldn't let that dissuade you from creating and sending interactive email campaigns. Apple Mail on both desktop and iOS support all of the techniques described here. **According to Litmus**, which tracks over a billion opens a month, Apple Mail collectively accounts for roughly 48% of the email client market share. That's a huge audience that can potentially interact with your emails.

The important thing to keep in mind is that you should be targeting those email clients and providing proper fallbacks for clients that don't support interactive emails. Fortunately for us, targeting Apple Mail is relatively

easy. Since those clients use the WebKit rendering engine, we can use the following media query for targeting:

```
@media screen and (-webkit-min-device-pixel-ratio: 0) { }
```

Then, within that media query, we can include all of the CSS we used above for the interactive email. This takes care of hiding a lot of styling and the functionality, but we would still need to hide the HTML from clients, too. For hiding content across clients, you can use the following:

```
<div style="display: none; max-height: 0; overflow: hidden;"></
div>
```

It's the same code that powers our hidden preview text. You may run into an issue with Outlook selectively not hiding content, in which case you can wrap that content in a conditional comment and apply the mso-hide: all; rule inline on an element, like so:

```
<!--[if !mso]><!-->
<table align="center" border="0" cellpadding="0" cellspacing="0"
role="presentation" width="100%">
    <tr>
        <td align="left" style="display: none; mso-hide: all;
max-height: 0; overflow: hidden;">
            CONTENT GOES HERE
        </td>
    </tr>
</table>
<!--<![endif]-->
```

With your interactive elements hidden, you just need to ensure that you provide a decent fallback. In most cases this amounts to still displaying the content (like our slides, just not the inputs and labels) and stacking that content into a more traditional email design. Think of it like you would any progressive enhancement: you create the base layout, then enhance that layout with extra styling and interactivity in clients that support it.

There are a lot of nuances to interactive email and a lot of different techniques you can put to use. If you want to take things further, check out these resources online:

- Justin Khoo's **Interactive Email Examples**
- Justin Khoo's **CSS Support Chart**
- Mark Robbins' **Modern CSS and Interactive Email Presentation**
- Mark Robbins' **Punched Card Coding Article**
- Kristian Robinson's **CodePen Account**
- Rebel's **Blog**
- Litmus' **Blog**

And thank you to the folks above (and the many more) that have pioneered these techniques and taught it to the rest of us.

Chapter 10

Different Development Workflows

Chapter 10

Different Development Workflows

We've looked at a lot of concepts and code over the last nine chapters. One of the last few things I want to address in this book are different development workflows. So far, we've only looked at hand-coding email campaigns. Although this works for a lot of people (me being one of them!), there is an entire world of workflows out there that could help improve your productivity and decrease your build times, allowing you to focus on what's truly important in an email: the content.

In order of increasing complexity, here are some different development workflows you should consider in your own process.

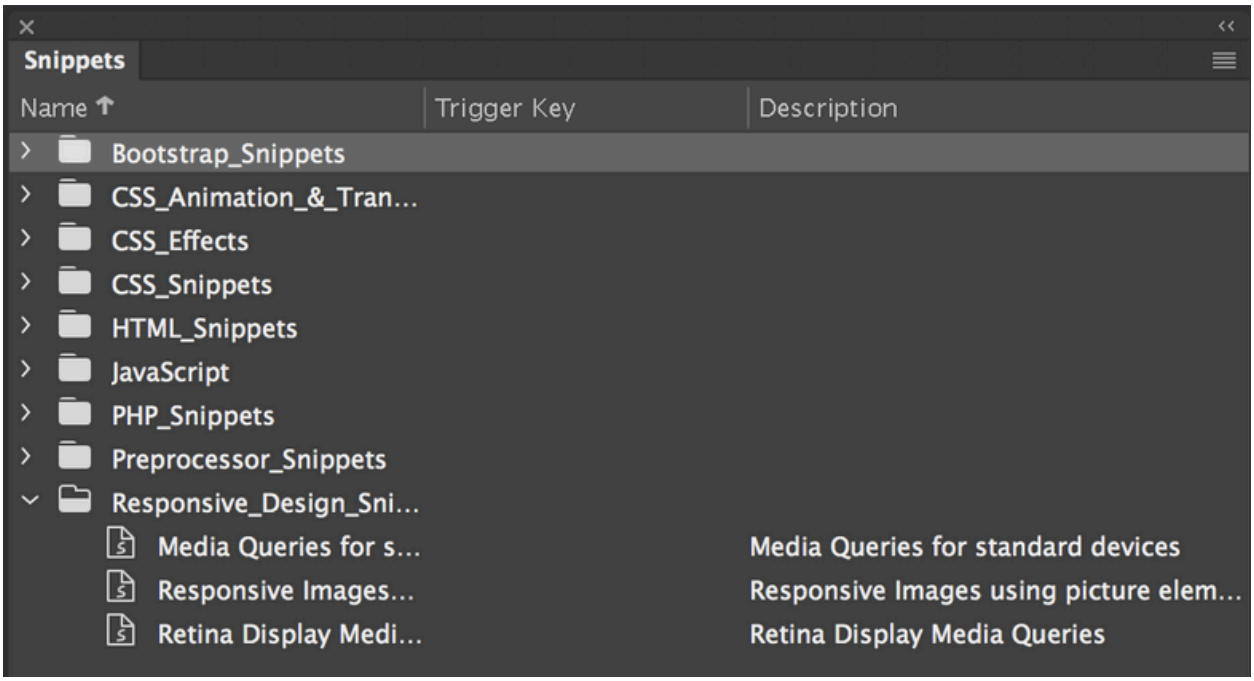
Old School Development

What I would call old school development is what you've been exposed to already. It's largely hand-coding HTML and CSS in your text editor of choice, with the files saved locally on your computer. This has worked well for a lot of people for a very long time. If this is what you're comfortable with, then I don't think there's much need to blow up your process.

Hand-coding definitely allows for the most control over your code. You can update, format, and build your emails however the hell you want, without being subject to the whims of someone else's tools. It may potentially be slower, but for individual developers, smaller projects, and even smaller teams, I think it's definitely the way to go.

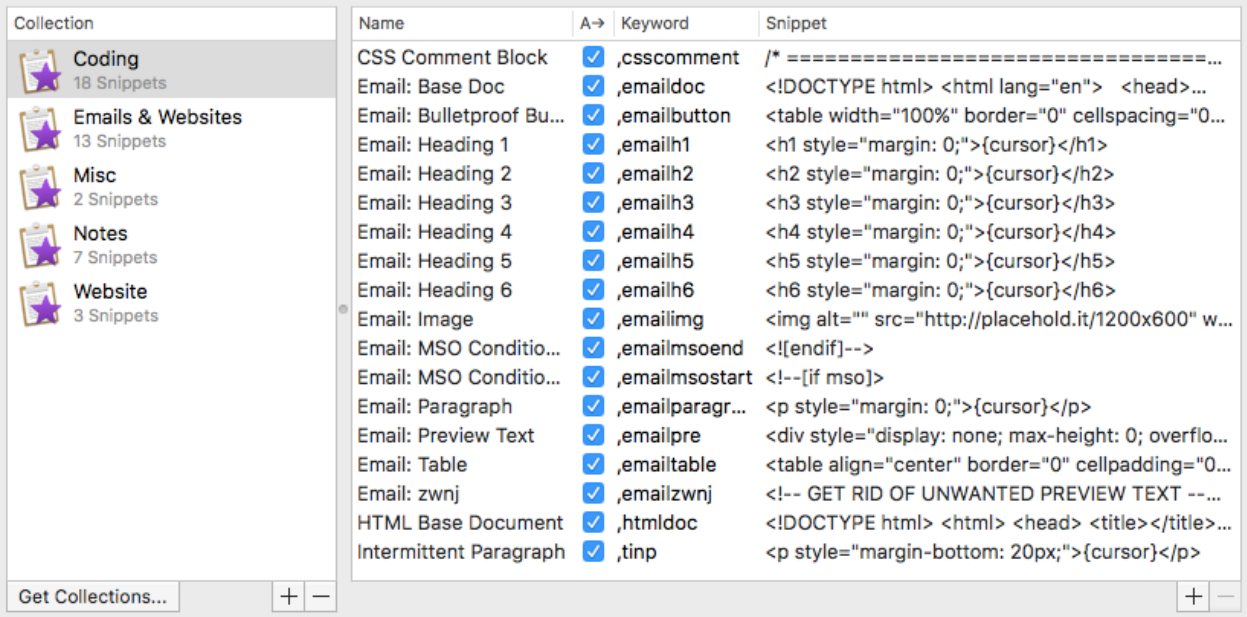
That being said, there are some tools that could make your old school development workflow a little better. My two favorites are using code snippets and relying on a tool called Emmet.

Code snippets are simply saved pieces of code that you can trigger within your text editor. Most text editors allow you to declare your own snippets. For example, Dreamweaver has an entire Snippets panel within the application. It comes with a lot of prebuilt code snippets and allows you to create your own using a graphical interface. If you're a Dreamweaver user, **this video** might help you out.



Sublime Text (and other pure text editors) allows you to declare snippets in a **simplified XML file**. It's not as nice of a process as in Dreamweaver, but you get the benefit of being able to tap into a vast community of developers, plugins, and modern tools, instead. Plus, most modern text editors are a hell of a lot faster than Dreamweaver ¯_(ツ)_/¯

Personally, I use a tool on my Mac called **Alfred** to store and trigger all of my coding snippets. Within Alfred, I have a few folders for different uses, including a "Coding" folder that stores my code snippets.



I can quickly add new snippets, set up triggers for them, and even place my cursor anywhere in the snippet after it is triggered. An example

snippet is a heading. By typing **`,emailh1`** in my text editor, the following is automatically written out:

```
<h1 style="margin: 0;">Cursor is automatically placed here.</h1>
```

Snippets are a wildly useful tool and greatly speed up my coding. I highly suggest setting up some of your own.

You won't want to set up snippets for absolutely everything, though. Snippets are great for commonly used, larger pieces of code but, for simpler bits of code or quickly building up code structures, I'd suggest using a tool like **Emmet**. Emmet (formerly Zen Coding) is a plugin for most modern text editors that offers a similar functionality to snippets, but with a lot more power. Once Emmet is installed, I can type **`p`**, press the **`tab`** button, and have Emmet automatically insert:

```
<p></p>
```

This is a simple example. Emmet has operators that allow you to do amazing things really quickly. Let's say that I wanted to add a table with three rows. Each row needs to contain a table cell with an **`h2`**, **`img`**, and **`p`** tag inside. Instead of writing that all out by hand (and wasting a lot of valuable time), I can write the following in Emmet:

```
table>tr*3>td>h2+img+p
```

Emmet then spits out:

```
<table>
    <tr>
        <td>
            <h2></h2>
            <img src="" alt=""/>
            <p></p>
        </td>
    </tr>
    <tr>
        <td>
            <h2></h2>
            <img src="" alt=""/>
            <p></p>
        </td>
    </tr>
    <tr>
        <td>
            <h2></h2>
            <img src="" alt=""/>
            <p></p>
        </td>
    </tr>
</table>
```

That's pretty damned amazing, but that's just the **tip of the iceberg**.

Along with code snippets and Emmet, I find it valuable to add some sort of version control system to your old school workflow (or any workflow, for that matter). Version control systems allow you to keep track of the changes you make to a file by "checking in" files to the system. This can save your ass, as coding mistakes are bound to happen. It also helps when you're trying to change a lot of code to test out a new idea or fix a

problem but you want to have a backup of your code if you need it later. Instead of manually copying and pasting files or code between files, and naming those files something unique every time, you can "branch" your code within your version control system and make all the changes you want without having to worry about losing older versions.

Two of the most popular version control tools are **GitHub** and **GitLab**. Both are free to start using and offer nearly identical features, but have their own advantages.

GitHub has a massive community using its service and is built with social components baked in, making it easy to find, fork, and use code that others have written. If you want to work with a team, get access to some more advanced features, or have your code kept private from others, you'll need to pay for your account. At this point, it's almost required to have a GitHub account if you're a coder of any sort. The community has flocked to the service and a lot of employers look at your GitHub account as a measure of your skills.

GitLab definitely has less of a community around it, but has some cool features that still make it worth checking out. My favorite is that you can have private code repositories on their free plan. Actually, you get a ton of amazing features on their free plan. They also have great build tools, which they call their Pipeline, which can be useful if you start digging into more complex web projects. I'm actually an avid GitLab user and prefer it over GitHub.

Regardless of which service you use, a version control system is a great way to back up your code, quickly iterate on ideas, and share code with others.

Email Frameworks

The next step up on the workflow ladder is using an email framework for coding your emails. Email frameworks are prebuilt collections of code that allow you to write your emails in an abstracted coding language that is then compiled into proper email code. This is nice in that you don't have to remember all of the intricacies of coding an email to build one. You can safely ignore a lot of the bugs and hacks we've gone through in this book and focus more on the content of your campaign.

My favorite current email framework is **MJML** from the folks at **Mailjet**. MJML, which stands for Mailjet Markup Language, was designed in-house by their team of experienced email developers. It does require you to install it, which can be tricky depending on your familiarity (or lack thereof) of the command line. Once it is installed, you can write in their markup language to generate your emails:

```
<mjml>
  <mj-body>
    <mj-container>

      <!-- Company Header -->
      <mj-section background-color="#f0f0f0"></mj-section>
```

```
      <!-- Image Header -->
      <mj-section background-color="#f0f0f0"></mj-section>

      <!-- Introduction Text -->
      <mj-section background-color="#fafafa"></mj-section>

      <!-- 2 columns section -->
      <mj-section background-color="white"></mj-section>

      <!-- Icons -->
      <mj-section background-color="#fbfbfb"></mj-section>

      <!-- Social icons -->
      <mj-section background-color="#f0f0f0"></mj-section>

    </mj-container>
  </mj-body>
</mjml>
```

The following will create a skeleton for your email. To add something like a two-column section, you could use something like this (taken from MJML's documentation):

```
<!-- Side image -->
<mj-section background-color="white">

  <!-- Left image -->
  <mj-column>
    <mj-image width="200"
              src="https://designspell.files.wordpress.com/
2012/01/sciolino-paris-bw.jpg" />
  </mj-column>

  <!-- right paragraph -->
  <mj-column>
```

```
    <mj-text font-style="italic"
             font-size="20"
             font-family="Helvetica Neue"
             color="#626262">
        Find amazing places
      </mj-text>

      <mj-text color="#525252">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Proin rutrum enim eget magna efficitur, eu semper augue semper.
Aliquam erat volutpat. Cras id dui lectus. Vestibulum sed
finibus lectus.</mj-text>

  </mj-column>
</mj-section>
```

While you're still writing code, it tends to be a lot cleaner than the typical HTML and CSS you would write for an email. Once you have your MJML code written, MJML's build tool will generate the HTML and CSS you need for you, which you can then take and upload to your email service provider.

MJML has a ton of useful components built in, making it easy to build out most common email designs. Even better, you can build your own components to make customizing your templates even quicker. MJML also has great tooling around it, with plugins for popular editors, and a wonderful, active community. You should definitely check it out at **MJML.io**.

Another interesting choice is **Foundation for Emails** by **Zurb**. Foundation offers much of the same functionality as MJML but has a bit

of a different feel to it. Your code is more or less straight HTML, but you use classes that invoke Foundation's CSS to style your email. Foundation's build tool then allows you to inline that CSS to generate a send-worthy campaign.

Foundation does have its own templating language, similar to MJML's **<mj-section>** type tags. It's called Inky. If anything, it's actually a bit easier to understand than MJML's version. Here's what a single-column section would look like:

```
<container>
  <row>
    <columns>Put content in me!</columns>
  </row>
</container>
```

There is a lot more to Foundation, which you can read about in their **documentation**.

Again, I like both frameworks but prefer MJML. It feels like it's built by people more experienced with HTML email and has a really great community of folks to help out if you run into any issues.

Email frameworks can be a fantastic way to speed up your process and can be wildly helpful if you're working on a larger project or larger team. They help get everyone on the same page and following a well-defined methodology for creating emails, which can help when you have more

than one or two people working on emails that need to maintain a consistent set of standards.

My main complaint with email frameworks is that they abstract away from the email itself. It's harder to fully understand what's going on in your email code since it's hidden away. If you're not already familiar with coding HTML emails and you need to poke around in the generated code from your framework (which can easily happen when troubleshooting problems) then you're going to get lost and have a rough time. They also tend to be a bit harder to customize. Each email framework comes with its own assumptions about how an email should be built (and to a lesser extent, how it should look). If you need to create emails with different visual looks to them, it might be easier to hand-code a bespoke template instead.

Build Tools

If you want to get more control over your development workflow, but want some of the advantages of using an email framework, you may want to look into using a build tool or task runner. Build tools are programs that allow you to automate tasks. For email, this might include the following:

- Automatically inlining CSS
- Build templates from different partial files
- Automatically build localized versions of emails

- Build different variations of emails based on specified variables

There are a lot of build tools out there, but the two major players are **Grunt** and **Gulp**. Both are based on the JavaScript programming language and need to be installed through the command line. Once installed, you still have to do a bit of work for setting up the build tool. In the case of Grunt, you'll need to create a new Grunt project, which includes two files: **`package.json`** and **`Gruntfile.js`**. The JSON package is used to store information about your project, including which Grunt plugins are required to complete any tasks. There are a massive number of plugins in the Grunt ecosystem (at the time of writing 6,280), so you can find a plugin to handle pretty much anything you may need to do. The Gruntfile is used to actually load your plugins and then define any custom tasks you need performed.

Getting into the weeds is most definitely outside the scope of this book, but you can find out more about both tools using the links listed earlier.

Build tools really start to shine when you start working on larger email projects. For example, let's say that you work for an international retailer that sends dozens of different templates that all need to adhere to brand guidelines (which may change depending on a specific region) and that need to be localized for a region's language. Using other workflows, that would take *a lot* of manual work and be absurdly time-consuming. Using a build tool, you could set up the following:

- Individual partial files that control components like headers, footers, product sections, CTAs, etc.
- Variables that control all aspects of those designs, including styling like colors, logos, and fonts as well as individual pieces of copy for those components
- A translation table for all of your copy, typically a CSV file

Then, using the build tool, you can automatically combine all of those elements into the required emails and even test them and upload them to your email service provider, ready to send.

What's more is that you have full control over the HTML and CSS. Since you're writing it all yourself, you can code using whatever techniques you want and know that you can (relatively easily) dig in and troubleshoot or update that code whenever you need to.

The major drawback of using a build tool is that it takes a fair amount of initial work to get running. You need to install the tools and then spend a good amount of time getting familiar with them, researching plugins that you may need to use, then set up and test your build process to make sure it does what you need. Most also assume some familiarity with a programming language like JavaScript, which can be intimidating for some.

For individual developers working on smaller projects or bespoke templates, I don't think they are worth the effort. For everyone else, though, they are definitely worth trying out.

Commercial Tools

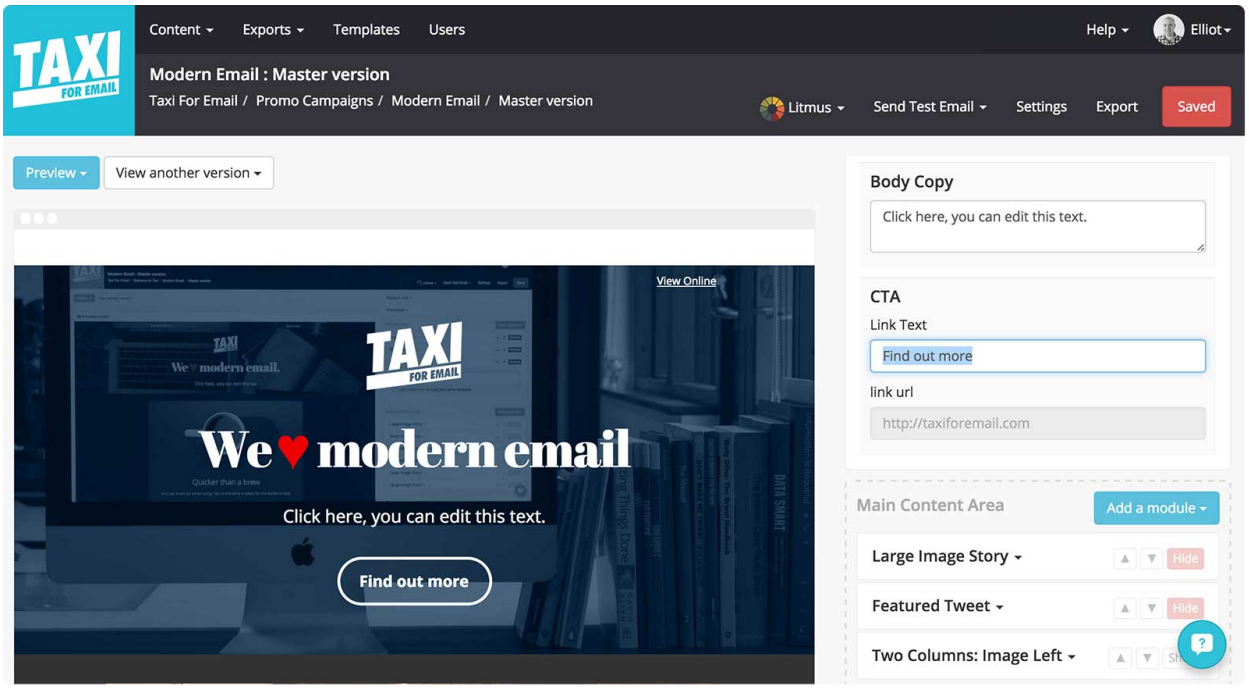
The last level of workflows that I want to touch on are using commercial tools. In particular, two commercial tools: Litmus Builder and Taxi for Email.

Litmus Builder (full disclosure in case you haven't figured it out yet: I work for and use Litmus tools on the regular) is an online code editor and testing tool built by **Litmus**. For all intents and purposes, it is a complete build tool for email development. But, it doesn't require all of the setup and maintenance of a build tool like Grunt or Gulp. You could easily use Builder as a simple code editor and work in it old school-style, but you could greatly improve your workflow by relying on its more powerful features.

Litmus Builder includes support for partials and code snippets, has a custom-built CSS inliner specifically for email, and has tools to help you quickly navigate and test your emails in over 70 different email clients. Hell, you can even automatically add tracking parameters to all of your links and sync your emails (including images with updated paths) to your ESP.

There's a lot to Builder, but this **series of articles** on the Litmus Blog (penned by yours truly and my good friend Jaina!) will help get you up to speed.

Taxi for Email, created by the folks over at **Action Rocket**, is a little different from Builder but can be just as beneficial to use. It acts kind of like a content management system for email campaigns, and decouples the content of the email from the design and code underneath. While it's not a build tool, it can help with the email creation process by allowing you to build out and upload components that marketers can then use to build emails without having to mess with code.



This can be a huge win for teams that divide the development and copywriting, strategy, and marketing people. Email developers can build components however they want, upload them to Taxi, then give marketers access to Taxi's drag-and-drop interface to build out individual

campaigns. Marketers don't have to touch the code and Taxi won't mess with your code, ensuring that your well-tested, robust components won't break. Like Litmus Builder, Taxi also offers connectors that allow you to sync your campaigns and assets with your ESP.

Both tools address different concerns in the email creation process but, like the other tools mentioned in this chapter, can greatly improve your productivity and efficiency. The major drawback is that both tools cost money, which could be a blocker for some.

There are a lot of other tools and workflows out there. It seems that nearly every team has their own way of doing things. A great resource for exploring how other people create emails is Litmus' **2017 State of Email Workflows report** (which is free BTW). Regardless of what you ultimately end up implementing, spending time on refining your workflow will pay dividends in the long run.

Chapter 11

Troubleshooting Emails

Chapter 11

Troubleshooting Emails

While most of us would love to spend our time coming up with and implementing beautiful email designs, a large part of your time working in email will be spent on troubleshooting problems.

As we've discussed before, email clients are a fickle bunch. Varying levels of support for HTML and CSS, combined with strange, often undocumented bugs, lead to lots of problems in email campaigns. Our most important job as email designers and developers, then, is to seek out those bugs, understand that support, and figure out solutions to any problems we encounter.

That may seem like a daunting task, but I've worked out a framework to help you troubleshoot and fix any problems that may arise. This framework (I use that term loosely) was developed this past summer for a series of workshops at **Litmus Live**. With the help of my friend and colleague, **Jaina Mistry**, I put together a set of steps and tips to help troubleshoot any email problems.

Troubleshooting Step-by-Step

I've broken troubleshooting down to four main steps:

1. Testing an email
2. Checking images
3. Checking code
4. Getting help

These four steps will allow you to troubleshoot (and hopefully fix) any potential problems you may encounter. Let's see how the work together.

Testing an Email

The first part of troubleshooting is actually identifying problems. At no point should you ever send an email campaign without first testing to see how it displays in different email clients and on different devices.

There are a few ways to go about testing. Let's look at the pros and cons of each.

Sending a Test Campaign

The first—and easiest way—to test HTML emails is by sending yourself a test of that campaign. By using your email service provider or a one-off sending tool (like **PutsMail**), you can quickly see how your email looks

right in your own inbox. Most ESPs allow you to set up a list of email addresses for specific purposes and then send test emails to that list. When you're developing your email, you can load your campaign into your ESP and then send a test email to yourself, checking it in whatever email clients you have on your test list.

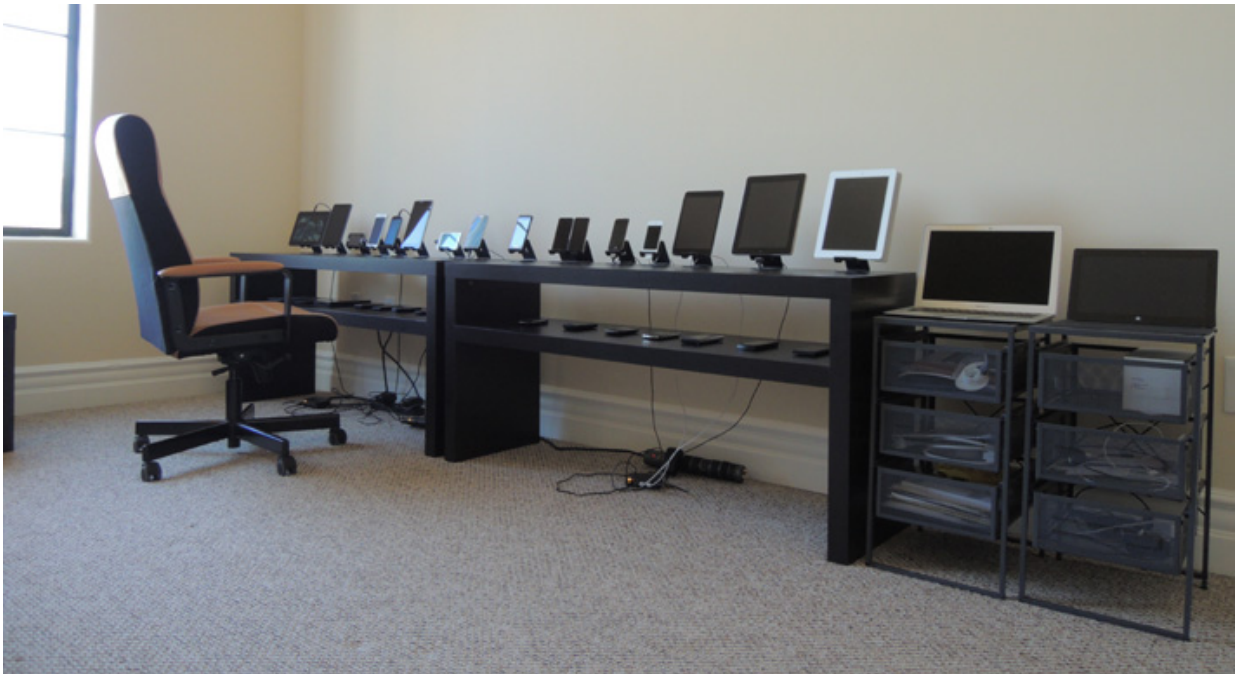
If you don't have access to your ESP (common in larger email teams), then you can use a service like **PutsMail** or **Litmus Builder** to quickly send yourself a test campaign.

Whatever method you use, you'll want to make sure that you're sending tests to as many email clients and services as you can get your hands on. This is relatively easy for webmail clients, since services like Gmail, Yahoo, and Outlook.com allow you to set up free email accounts. But for desktop and mobile clients, it's harder to set up multiple testing accounts since some clients and apps require specific operating systems or devices to run on.

Still, sending yourself a test campaign and seeing how your email looks in real, live email clients is the best way to understand exactly what your subscribers are seeing. And, if you're building and sending interactive or dynamic campaigns, you **have** to test those features in live clients to ensure they are working properly.

Testing on Real Devices

Most beginning developers will start testing by sending themselves campaigns and checking to see how they render in a handful of email accounts. But, you can take that a step further by setting up a device lab to test on as many devices and in as many email clients as possible. This is similar to the last approach in that you will be sending yourself test campaigns either through your ESP or something like PutsMail, but this approach focuses on setting up individual, real devices and machines to see how your email renders in any environment.



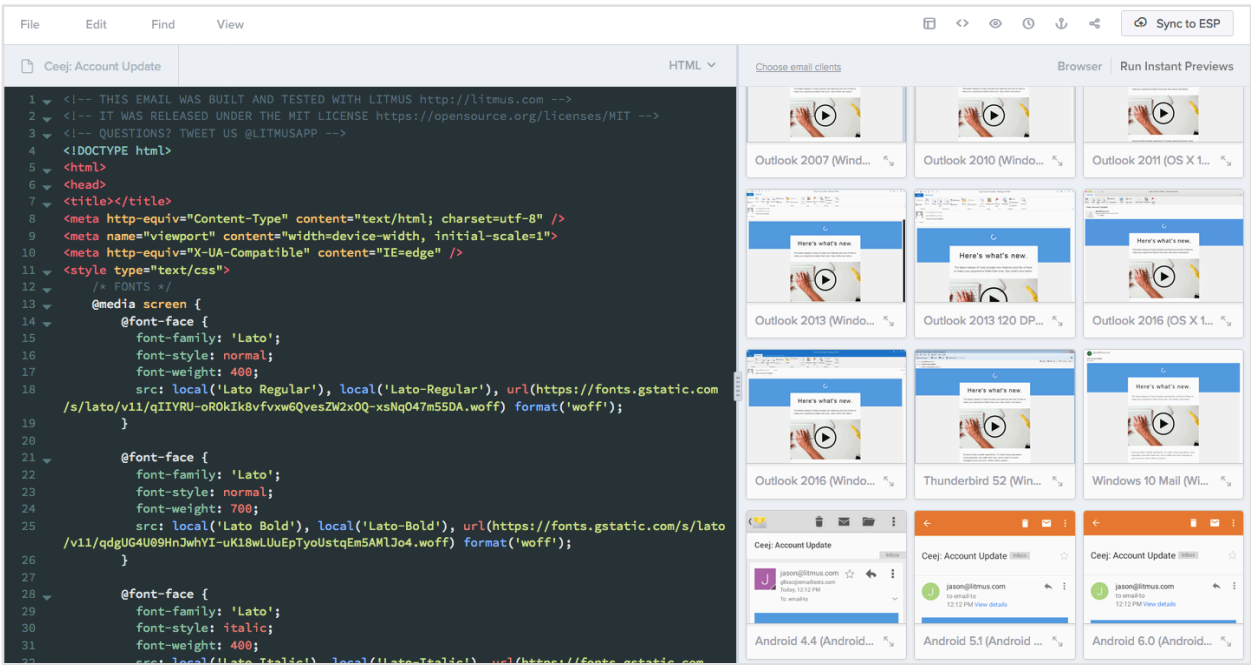
In the photograph above, you can see a real device lab from the email agency **StyleCampaign**. StyleCampaign maintains dozens of real devices, all running various email clients and operating systems, to allow them to test their campaigns in real-world conditions. This is the ultimate method of testing, as it gives you the most realistic experience compared to what your subscribers see. And, like I mentioned before, if you're sending interactive or dynamic emails, this is the only way to truly test that functionality.

The problem with testing on real devices is that it can get very expensive and time-consuming to set up and maintain those devices. While device labs can be impressive to clients (and necessary in some instances), most email developers can use more affordable options to test their campaigns…

Using a Testing Service

The best option for most email designers and developers to test their campaigns is to use a testing service. There are a few online services that allow you to automatically test your campaign in dozens of email clients, but my favorite is **Litmus**. While I actually work for Litmus, I was a Litmus customer and user for about three years prior to joining the company. During that time, and since then, I've used Litmus to vastly speed up the development and testing process.

Testing services like Litmus allow you to either send a campaign to the service or upload your code for testing. Once they have the code, they run it through dozens of email clients, take screenshots of your email in those clients, and then display the results. You can then look through those screenshots to identify any rendering problems that you need to fix.



Testing services used to be quite slow, but in recent years have drastically improved. Litmus now allows you to test in (as of this writing) over 70 different email clients, with screenshots returned in mere seconds.

Although testing services cost money (typically a monthly or annual fee), they obviate the need to manually sign in and check multiple email accounts or maintain an expensive device lab.

Regardless of what approach you use to test your email campaigns, seeing how they display across different email clients and devices is the first step in troubleshooting any problems.

Checking Images

Before you start messing with your code after encountering a problem, I recommend you first look at any images in your campaign to see if there's a problem. A lot of times, new email developers mistake a problem with one of their images for a coding problem and waste a lot of time trying to figure out what is usually a quick fix. With that in mind, here's what you should check first if your email campaign is looking off in any email clients.

Image Paths

As we discussed in the earlier chapter on images, all images in an email require the use of *absolute paths*. All of your images need to be hosted on a publicly accessible server, with the **`src`** attribute in the **`img`** tag pointing to the URL of the image. Email clients don't have a concept of relative paths, so this won't work:

```
<img src="/path/to/img.jpg">
```

This is an extremely common mistake developers make, especially when building emails locally on their machine. Most developers create a folder for an email, in which they save the email HTML along with images for the campaign. By calling those images using a relative path, developers can quickly update those images in their campaign until the design is perfect. But, they sometimes forget to upload those same images to a server and update the paths in the HTML, causing display issues when testing their campaigns.

Double-checking that you're using absolute paths—and your images are hosted on a publicly accessible server—is always my first stop for troubleshooting. I've been doing this for the better part of a decade, and I still make this mistake constantly.

Unsupported File Types

Again, just another reminder to use supported image file types in your campaigns. Most email clients support a limited set of image file types, namely PNG, JPEG, and GIF. Occasionally, design or marketing teams, or outside agencies, deliver assets in a different file format and we forget to convert it to a usable one. Then, when our campaigns fail to render properly, we scramble to identify the issue.

Just double-checking that you're using a supported file type, or that you fallback to a supported format when using something more experimental like SVG, can save you a lot of time when troubleshooting.

Checking Code

If you're confident that your images aren't the problem, the next step is to locate problems within your HTML or CSS. This can sometimes be harder than you would expect, especially considering that HTML and CSS that works in one email client won't work in another.

My first step in checking code is usually isolating the problematic code. Knowing where to check in the code gets you halfway there, then it's just a matter of fixing things. There are a few tricks and tools you can use to isolate the problem areas in your code. Let's see how they work.

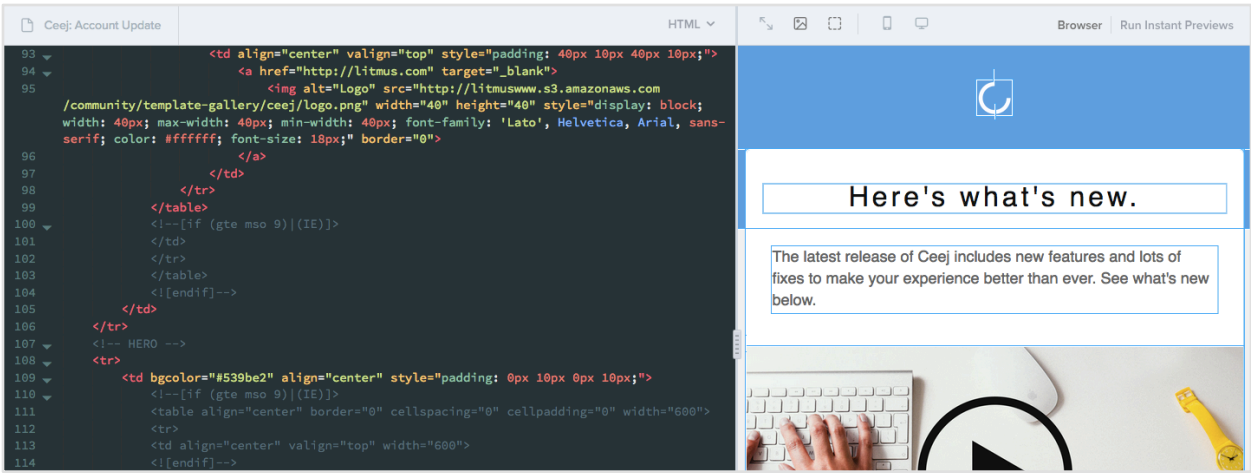
Outlining Code Components

The easiest way to hone in on a problem is to visually outline the different parts of your email. Then, once those parts are outlined, identify which table, row, or cell is causing the issue and look at that bit of code in isolation.

One way to outline parts of your email is to add borders to your tables and table cells. If you recall from chapter 2, we always override default styling on tables by applying a **`border="0"`** attribute to the **`table`** tag. By switching that to something other than zero, you can quickly view those individual components to zero in on the problem. An easy way to do this is to pop open the HTML of your email in your favorite text editor

and go choose “Find and Replace” in the menu. Then you can simply find all instances of **`border=“0”`** and replace them with **`border=“1”`**. Once you figure out your problem, just swap that back to zero and you’re good to go.

Another option is utilizing built in tools within your text editor. Both Adobe Dreamweaver and Litmus Builder have “design” or “grid” views. When triggered, these tools will automatically overlay lines around individual tables and cells within your email.



Both of these options allow you to see exactly where your problems are and can be indispensable for email designers and developers.

Isolating Components

Once you've outlined and zeroed in on the problem, it's often beneficial to isolate that component and work on fixing it away from the rest of the email. We've previously talked about the benefits of using modular design and discrete components for building campaigns. These modules really start to shine when it comes to troubleshooting. Since those components are largely decoupled from each other, you can remove the problematic one and test it in isolation.

Simply copy and paste that module into its own HTML file and work on fixing any problems. Then you can test those fixes on your own devices or using a service like Litmus to ensure that everything renders properly across different email clients. Once you're confident that the module is fixed, you can reintroduce it to the rest of your email and test it as a whole.

That two step process—outlining and isolating components—is where nearly all of my troubleshooting starts.

Common Code Problems

When you are looking at your code, there are a few things you should look for before digging in much deeper. HTML, while forgiving on the

web, can be fickle in the email world. It's for that reason that I recommend reviewing your code for any of the following:

- Missing HTML tags
- Missing delimiters
- Typos
- Unsupported HTML and CSS

Missing HTML tags are a very common problem. Most of us work under very tight timelines and we can often forget to close out an HTML tag. Especially if you're using an older text editor or hand-coding your email, missing tags are a common cause of rendering issues. There are a few ways to avoid missing HTMl tags.

The first is by double-checking your code after writing it. As you become more familiar with coding, you'll get a better feel for how your code should look and be able to more easily spot missing HTML tags.

The second is by relying on your text editor or plugins for your text editor to identify missing tags for you. Most modern text editors either support highlighting missing tags by default or can be extended with plugins or add-ons to allow for the same functionality. This is my preferred method, since it's completely automated. I *believe* that the following all support highlighting missing tags by default:

- Sublime Text
- Atom

- Visual Studio Code
- Dreamweaver
- Coda

If you run into an issue where you don't see missing tags highlighted, you may need to check your application's settings. If all else fails, try searching online for a plugin to extend your text editor.

Finally, you could also use a tool like the **W3C Markup Validation** Service to identify problematic HTML for you. The W3C Markup Validation Service allows you to upload HTML that is then scanned for issues. It will highlight specific pieces of code that could cause problems, even going so far as to tell you what line that code is on. It's wildly handy but comes with one major caveat:

HTML emails use a lot of deprecated code and hacks that will be flagged by the W3C Validation Service. While it may seem like you have a lot of errors and warnings in your code, you can generally ignore most of them. Just pay attention to errors like "unclosed element".

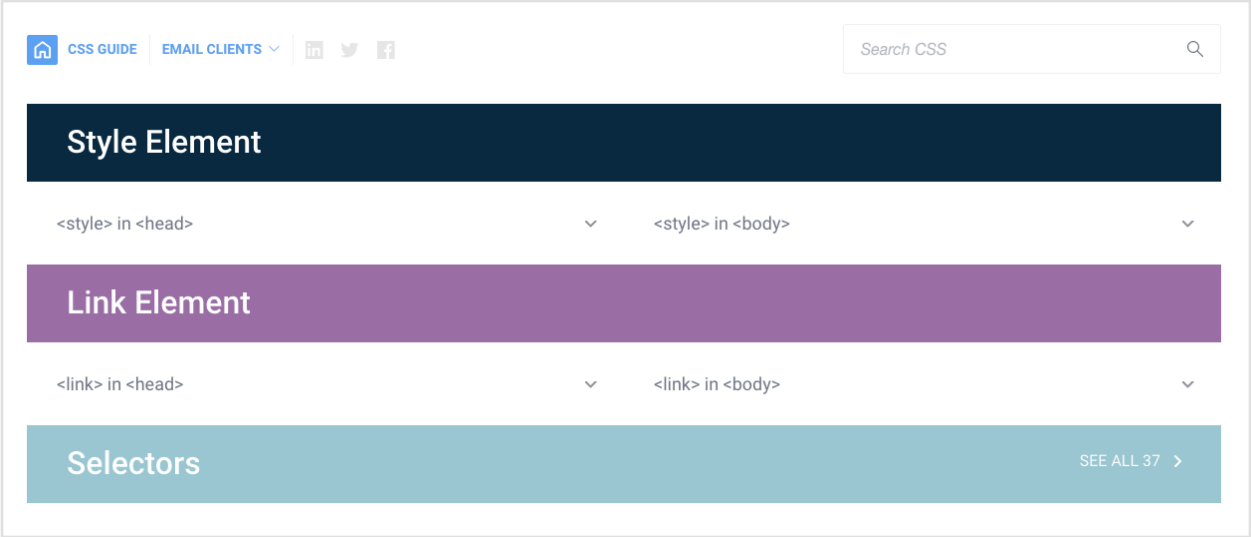
Missing delimiters can be a problem as well. Delimiters are characters like **"**, **;**, **<**, **>**, **{**, and **}** that are used to open and close sections of tags, properties, and other code. Again, when hand-coding especially, it's easy to miss some of these and they could cause major rendering issues. Using the same methods as mentioned in the previous paragraphs is the best way to check for missing delimiters.

Typos can be a little trickier to account for. They are relatively common even amongst the best coders and, unfortunately, not all text editors can pick up on and highlight them. A keen eye is your best bet for keeping track of them.

Finally, the biggest problem you'll run into with coding emails is a shear **lack of support for HTML and CSS** across clients. As we've seen before, email clients and their rendering engines are a far cry from current web browsers. The web standards movement of the mid-2000s flew right by email client vendors, so we're stuck picking up the pieces and making the best of a shitty situation.

Understanding which clients support what HTML and CSS is the best way to become a better email developer. Although a lot of that understanding comes with practice and time in the industry, there's one killer tool at our disposal for checking support.

Campaign Monitor's Ultimate Guide to CSS is a massive guide to most CSS properties and features and their support across the most popular email clients in use today.



Campaign Monitor's CSS Guide.

It has been around for years, but received a huge update in September of 2017 and now tests 278 different CSS properties and features across the following 35 email clients:

Desktop	Webmail	Mobile
AOL Desktop	AOL Mail	Android 4.2.2 Mail
IBM (Lotus) Notes 9	Gmail	Alto on Android
Outlook 2007-2016	Outlook.com	Blackberry
Outlook for Mac	G Suite	Gmail on Android IMAP
Apple Mail 10	Google Inbox	Android 4.4.4 Mail
Outlook 2000-2003	Yahoo! Mail	Alto on iOS
Outlook Express		Gmail on Android
Postbox		Gmail on iOS

that you can dive in and find exactly what you're looking for. Get familiar with this tool, it will be your savior countless times throughout your career.

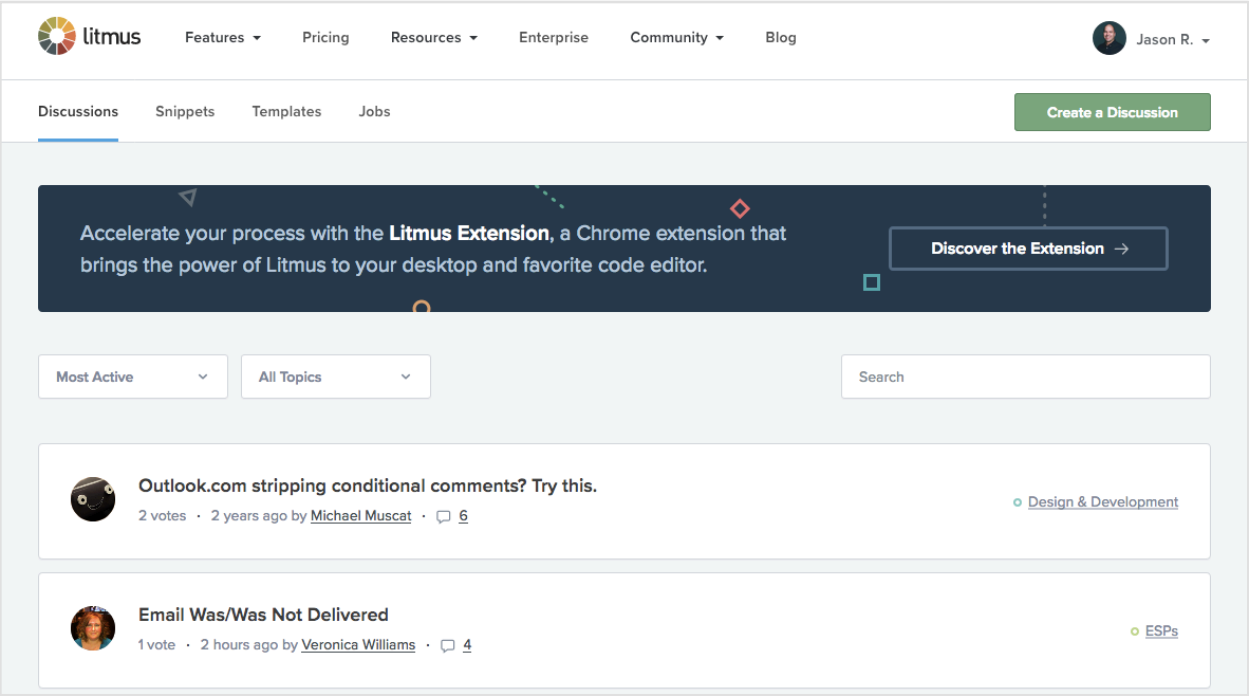
Getting Help

Sometimes, in spite of our knowledge, skills, and best intentions, we still can't figure out a problem on our own. This is where asking for help can save your tail. There are a lot of places and people to turn to in the email world, but the following are the ones I constantly turn to and believe are the best of the bunch.

The Litmus Community

The Litmus Community, founded in 2013, consists of thousands of email marketers, designers, and developers of all skill levels sharing knowledge, posting resources, and helping troubleshoot even the trickiest problems. There are thousands of discussions on all sorts of email topics, code snippets to use in your own campaigns, free, fully responsive templates on which you can build, and even a jobs board specifically for email professionals.

I may be biased since I helped build and grow the Litmus Community, but it truly is a special place. Some of the smarted email pros in the world hang out here and all of them are eager to help troubleshoot issues or answer questions.



You can sign up for free and start posting discussions. If you are looking for a specific issue, I'd recommend searching the forums first. Chances are good that someone else has had the same problem before and a fix has already been posted.

Twitter

Twitter is another great resource for email folks. People in email tend to be very vocal and active on Twitter and are always up for a good discussion on damned near any topic. Some good hashtags to follow include:

- #emaildesign
- #emaildevelopment
- #emailmarketing
- #litmuslive
- #emailgeeks

#EmailGeeks

Speaking of #emailgeeks, there is an entire Slack group devoted to them. Created by Viktor Edvardsson, the **#emailgeeks group is free to join** and—at the time of writing—consists of a little over 1,500 extraordinarily friendly, intelligent, and helpful email folks. Plus, there are dedicated channels discussing things like code, design, accessibility, automation, jobs, deliverability, and more.

Stack Overflow

A much larger resource is Stack Overflow, a gigantic community of designers and developers posting questions on all things related to the web and programming. While the HTML and CSS tags can be helpful, you'll find both the **HTML-Email** and **HTML-Templates** tags a bit more focused and relevant. My one word of caution is that a lot of questions may have outdated replies or information, or could be answered by developers without a strong knowledge of the realities of coding HTML email campaigns.

The Better Email Resources

Finally, a shout-out for my own resources. I maintain a **resources page on The Better Email** that acts as my personal, email-related bookmarking service. There is information on everything from tutorials and articles to code frameworks, tools, and even inspirational examples of emails. I also have a directory of some of the major thought leaders in the industry.

I try to keep it as up-to-date as possible. You should definitely **check it out**.

Again, there is a lot that can go wrong in an email campaign. Hopefully these steps will help you troubleshoot any problems you may encounter. And remember, if all else fails, just ask someone for help. Chances are good that the problem's been encountered before, so there's almost certainly a solution out in the wild. You just need to find it.

Chapter 12

Questioning Best Practices

Chapter 12

Questioning Best Practices

I want to end this book with a quick rant about best practices in email design and development. What we've looked at over the course of eleven chapters are what most would term best practices in email. Best practices are well-tested, generally agreed upon techniques for getting work done with a minimal amount of fuss.

While the techniques in this book will get you going, not all of them should be applied all of the time. And they should never stand in the way of trying something new and different.

If email developers stuck to the best practices of even six years ago, we wouldn't have interactive emails. If you go back about ten years, you wouldn't really have hybrid coding or bulletproof buttons.

There are a lot of different ways to accomplish the same tasks in email design. I know of a lot of them, but I am 100% sure that I don't know them all. People are constantly taking ideas from the web and programming worlds and applying them to emails.

I encourage you, in the strongest terms possible, to do the same. Experiment and never stop learning.

If you're looking for some ways to experiment, a few options to look into come to mind…

The holy grail of email development is **getting rid of tables**. Tables are still absolutely necessary in 99% of all emails, due solely to Microsoft Outlook. In the past year or so, some progress has been made on getting rid of tables. My own email newsletters use a **nearly table-free** approach. The only table is a single, conditional one for Outlook. Mark Robbins has made **significant progress** on getting rid of even that. There's still work to be done, but that work is flying in the face of best practices.

Another avenue for experimentation is using new page layout techniques like **CSS grid**. CSS grid is an entirely new specification for coding layouts that gets away from tables and things like floats. It only recently saw adoption in the web world when most major browsers started supporting in back in March of 2017. While it doesn't work everywhere, both **Litmus** and **Action Rocket** have put it to use in live campaigns that are sent to thousands of subscribers.

Mark Robbins and the crew at Rebel have taken interactive email so far that they now have a viable **shopping cart solution** for the inbox. You can literally browse products, select options and sizes, and pay for your clothes right from an HTML email campaign. They even have an API that allows you to **build that functionality** into your own campaigns. All because they didn't stick to best practices.

A lot of people joke that building emails is like coding a website back in 1999, long before the web standards revolution. I've joked about it many times (and sometimes still do), but we all need to move beyond that kind of thinking. It's only holding us and our industry back.

There is so much we can do in email these days it's insane. And there is so much happening in the web and design worlds that we can use as inspiration for pushing the envelope in email. We just need to move beyond best practices and experiment.

At the Litmus Live conference this past year, Kevin Mandeville had a great talk on a future-forward approach to email. He addressed a lot of myths prevalent in the email community, and dispelled them all. I think he summed it up best when he said:

> *Code like it's 2017. Fallback like it's 1999.*

I just want you to keep that in mind when you're creating your own emails. Follow best practices when you need to, but stay curious and try new things. I have my own little motto that I try to stick to:

> *Try. Make. Learn. Repeat.*

I hope that you try to do the same. Good luck.

About the Author

Jason Rodriguez is a writer and designer from Michigan who helps people better understand the web and email. He's written a few books on email design and marketing, contributed to industry publications like A List Apart and CSS-Tricks, and frequently speaks at industry events.

More at **https://thebetter.email**