WICKED FAST WEBSITES

How to build WordPress sites that load in under 1 second.

CHRIS FERDINANDI

Wicked Fast Websites

How to build WordPress sites that load in under 1 second.

By Chris Ferdinandi

Go Make Things, LLC

v2.0.1

Why web performance matters

I'm going to assume that if you're reading this book, you already care about web performance. But you may find yourself in a situation where you need to convince a client or a manager that this is worth spending time and money on.

With that in mind, let's take a minute to look at some data that explains why web performance matters.

Fractions of a second

A few year's ago, Google ran an interesting experiment [1]. Based on analyst research, they increased the number of search results displayed on a single page from 10 to 30. This added about 500ms—that's just half a second—of extra load time to the page.

The result? Search traffic, and subsequently ad revenues, decreased by 20-percent.

Amazon ran a similar experiment, and found that delays of as little as one-tenth of a second could reduce revenue by 1-percent[2]. One-percent may not sound like a big deal, but for a company with Amazon's volume, it's huge. One-percent of their Q3 2015 sales is $254 million.

Firefox reduced their page load time by 2.2 seconds and saw downloads increase by 15.4-percent[3].

Sites are getting bigger

In 2010, the average webpage was 600kb in size. At the beginning of 2013, that number had doubled to 1.2mb. At the beginning of 2016, the average website had ballooned to 2.2mb in size[4]. The average website has doubled in size every three

years for the last six years.

For a while, this exponential growth wasn't a problem. We took for granted that both computers and bandwidth got faster and more reliable every year.

And then mobile happened

People are accessing the web from devices with varying levels of computing power and bandwidth. Desktops and laptops, phones and tablets, TVs, watches, video game consoles.

And for a growing number of people, mobile isn't just one way they access the web—it's the way they access the web.

More than half of all Google searches[5] happen on a mobile device. Sixty-five percent of emails are opened on a smartphone or tablet[6]. More than half of all Facebook users visit the site exclusively on their mobile device[7].

Harvard Business Review reported[8] from 2012 reported that almost a third of all Americans used a mobile device as the primary way they access the internet.

Bandwidth varies wildly

I've heard people argue that it doesn't matter. 4G LTE, they say, is as fast as broadband wifi. And they're right. LTE is awesome!

Unless you live in a city with poor coverage, or a tall building blocks your cell signal. Or there's a bad storm that disrupts cell signal and knocks you down to a 3g connection.

And that's in North America, which has a good mobile infrastructure. In remote and developing nations, an EDGE network can be the norm.

Greater Performance Expectations

Amazingly, despite the varied power and bandwidth of mobile devices, users expect websites to be faster on mobile than on desktops.

Forty-percent of visitors abandon a website that takes more than 3-seconds to load[9].

Mobile performance is so important that Google factors it into their page rankings. Slower sites rank lower than fast ones.

We're in the middle of a perfect storm

1. Websites are larger.
2. Devices are more varied and less predictable.
3. Performance expectations are higher than ever.

Fast websites make more money. The rest of this book will be focused on what you can do to build WordPress sites that load insanely fast.

Measuring performance

When it comes to web performance, how fast is fast enough? **One second.**

One second is roughly the limit to keep a visitor's flow of thought on the task at hand uninterrupted[10].

Sound impossible? It's not, but that's because there's a catch.

Total load time and page weight aren't the right metrics

Historically, developers have looked at things like total load time and page weight as the key metrics for performance. While those are important, there are two that matter a lot more:

1. **Time to First Byte**, which tells you how quickly your server is sending data back to the browser.
2. **Time to Start Render**, which is how quickly the browser begins displaying content to your visitor.

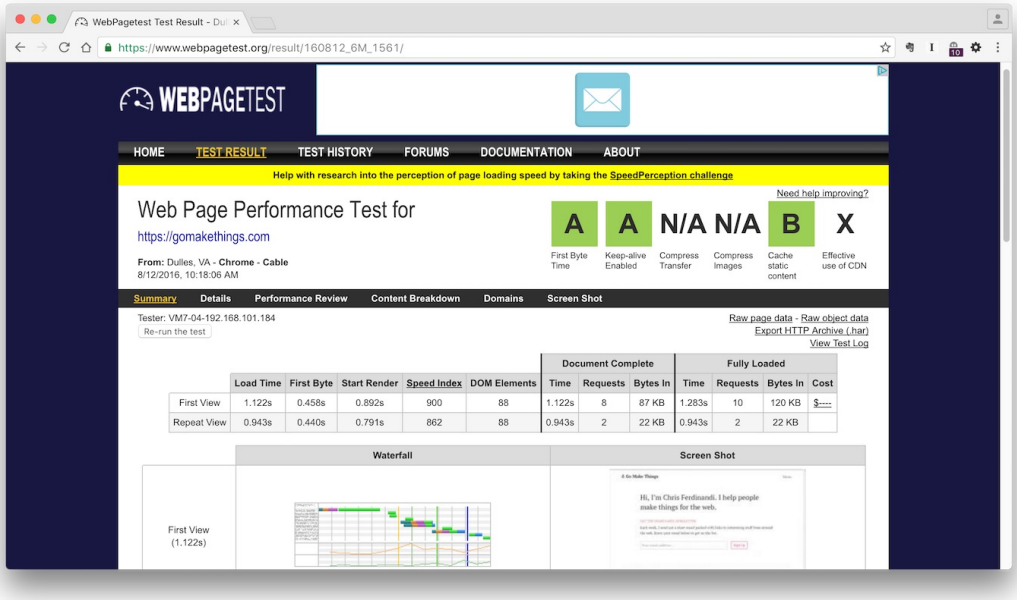
Your whole site doesn't need to load in one second for it to feel fast. It just needs to start displaying content in that amount of time.

Perceived performance is more important than *actual performance*.

Tools for measuring performance

My favorite tool for measuring web performance is WebPagetest.org[11]. Originally built by AOL, it was opened source in 2008 and is now maintained by Google.

WebPagetest provides a simple, web-based GUI that measures a ton of performance metrics about your site, but the two you care about the most are, of course, *Time to First Byte* and *Start Render Time*.



A report from WebPagetest.org

Let's use WebPagetest to get some baselines for your site. Visit WebPagetest, pop in your URL, and run it with all of the default settings. When that's done, run it again, but expand out the Advanced Settings and change the Connection to 3g. When that's done, run it a third time with an EDGE connection.

We want to get a baseline for your site on a variety of different connections. Sites can sometimes do quite well on cable but fail horribly on slower connections.

Target baselines

Time to First Byte

You ideally want this to be 100ms - 300ms, but if you're running on inexpensive,

shared hosting, it can closer to 500ms or 600ms.

If that's that case, don't freak out. You may want to consider switching web hosts. There may also be some simple changes you can make to your WordPress setup to improve this number. We're actually going to talk about some of this in a later lesson.

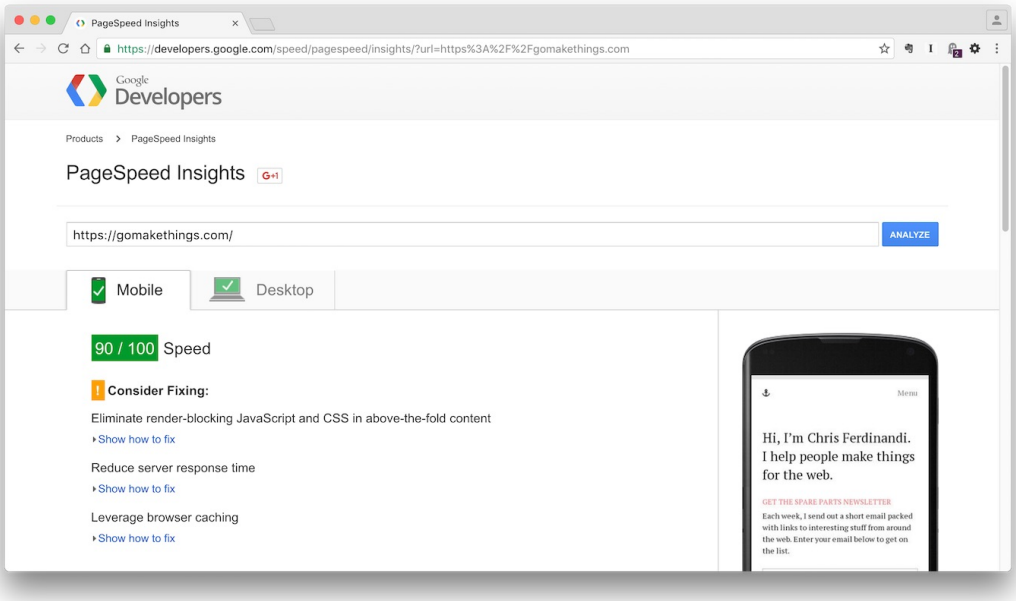
Start Render Time

- On a cable connection, you want this number to be 1,000ms (1 second) or lower.
- On a 3g connection, you want this number to be 3,000ms (3 seconds) or lower.
- On an EDGE network, you want this number to be 5,000ms (5 seconds) or lower.

If your site doesn't hit these numbers today, that's ok. We're just getting a baseline so we can measure our improvement. These are the targets we'd like to hit after we're done making all of our changes.

Google PageSpeed Insights

Another often recommended tool is Google PageSpeed Insights[12]. It provides a simple 0-100 ranking of how performant your site is.



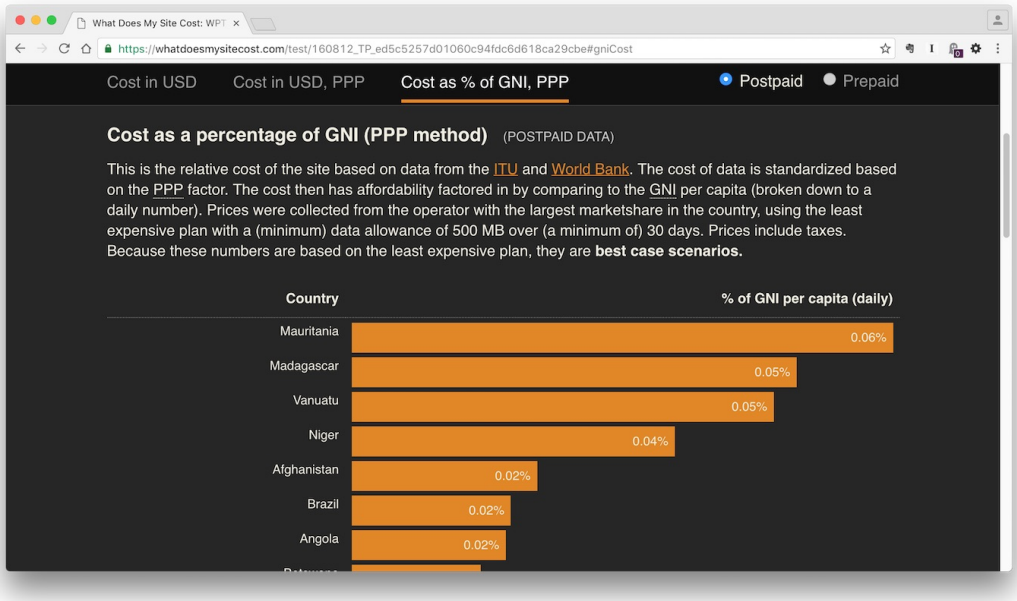
A report from Google PageSpeed Insights

I'm actually not a big fan of this tool for measuring performance. My site today is faster than it's ever been. It has a start render time of just over a second on initial load, and under a second on subsequent visits. My Google PageSpeed score is also the worst it's ever been.

Google PageSpeed Insights measures indicators of performance rather than actual performance. That makes a great tool for identifying potential bottlenecks, but not a great way to measure your actual performance.

What does my site cost?

Another really interesting tool for measuring performance is What does my site cost?[13].



A report from "What does my site cost?"

Built by Tim Kadlec [14], it mashes up data about your site from WebPageTest.org with average data costs around the globe[15] and financial metrics from the World Bank[16] to provide you with a few different views of what your site costs to visitors around the world.

When lobbying for performance, this is nice, tangible metric to share with clients.

Combine files

It's actually often faster for a browser to download one 150kb file than it is to download three 50kb files, even though the total file weight is the same.

DNS lookups, HTTP headers, redirects, 404s… there's a lot of places things can go wrong. Each HTTP request adds additional load time. How much time? From a now defunct article by Google:

> Every time a client sends an HTTP request, it has to send all associated cookies that have been set for that domain and path along with it. Most users have asymmetric Internet connections: upload-to-download bandwidth ratios are commonly in the range of 1:4 to 1:20. This means that a 500-byte HTTP header request could take the equivalent time to upload as 10 KB of HTTP response data takes to download. The factor is actually even higher because HTTP request headers are sent uncompressed. In other words, for requests for small objects (say, less than 10 KB, the typical size of a compressed image), the data sent in a request header can account for the majority of the response time.

And because browsers only download two files at a time (except JavaScript files, which block all other downloads), downloads create bottlenecks in the rendering process.

This is addressed in HTTP2[17], but browser support isn't broad enough that you can bank on it yet.

Combining Files

Combining similar file types together——a process known as concatenation——improves web performance.

Combine your scripts into just a few logical groups. For example, an `svg.js` file to

detect SVG support, and a `canvas.js` file to detect canvas support, might get grouped into a single `detects.js` file.

Similarly, `houdini.js`, `smooth-scroll.js`, and `fluidVids.js`, a set of DOM manipulation scripts, might get grouped into `main.js`.

To keep stylesheet size down, I often see people do something like this:

```
<link rel="stylesheet" href="style.css">
<link rel="stylesheet" media="(min-width: 20em)" href="phone.css">
<link rel="stylesheet" media="(min-width: 40em)" href="tablet.css"
>
<link rel="stylesheet" media="(min-width: 60em)" href="desk.css">
```

Here, the `style.css` file contains base styles, with additional stylesheets containing styles specific to smaller or larger viewports.

The problem is, browsers download all of these stylesheets regardless. The reason they do this is because if a browser window is resized or a device is rotated and this triggers a different stylesheet, the browser doesn't want to have to wait for another file to download before re-rendering the content.

So… it downloads all of the files ahead of time.

A better approach is to put all of your styles in a single stylesheet and use media queries to override styles within that file.

```
<link rel="stylesheet" href="style.css">
```

```
.base-styles { … }
@media (min-width: 20em) { ... }
@media (min-width: 40em) { ... }
@media (min-width: 60em) { ... }
```

Don't think you can cheat by using `@import`, either. It still triggers an additional HTTP request.

Tools and Techniques

GUIs

- CodeKit for Mac[18]
- Prepos for Windows[19]

Command Line

- Gulp[20]
- Grunt[21]

Plugins

- MinQueue[22]

Other

- Manually

Concatenate plugin files

One often repeated piece of advice I see for improving web performance is to minimize the number of plugins you use.

Plugins, conventional wisdom says, increase the load on your server and database and slow down your site. It's even in the WordPress Codex[23].

It's also 100% wrong.

Plugins aren't the problem

I run multiple WordPress sites that run 20, even 30 plugins. They all start rendering content in around 1 second (slightly more on initial load and under that on subsequent page views).

Plugins are not bad for performance. Badly written plugins are.

And in our case, one of the things we should really be concerned about is plugins that load multiple scripts and styles on the front end. This is very common.

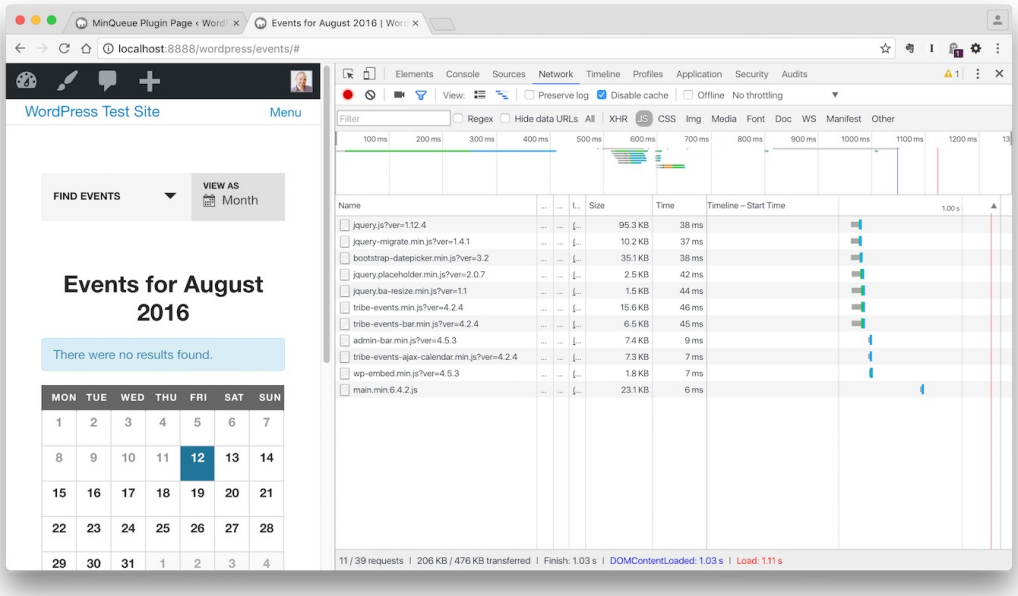
Fortunately, one of the tools we talked about in the last chapter, MinQueue[24], is an excellent tool for addressing this issue.

MinQueue hasn't been updated in a few years, but it still works perfectly and doesn't generate any errors. And it's still the best plugin I've found for this sort of thing, so I feel comfortable using and recommending it.

Getting a baseline

Visit any page on your site, but ideally one that seems to run a bit slow or where you know plugin files might be getting loaded (for example, a store if you're using

an ecommerce plugin or a calendar if you're using an events plugin).



There are 13 different scripts being loaded on this page

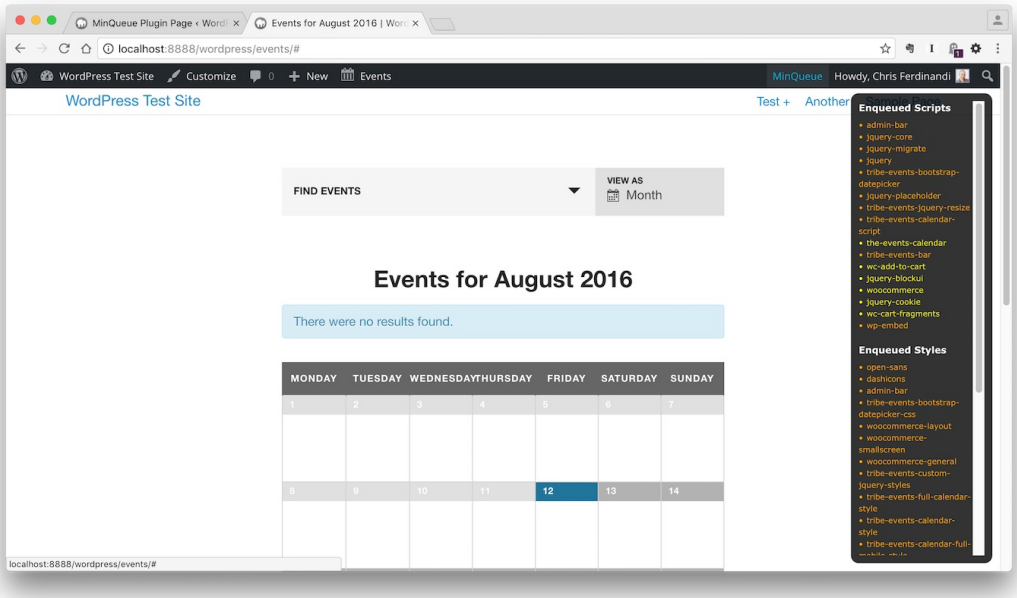
Open up developer tools and click on the "Network" tab. Now reload the page, and watch how many CSS and JavaScript files are loaded. This is our baseline.

How to use MinQueue

First, install and activate the plugin.

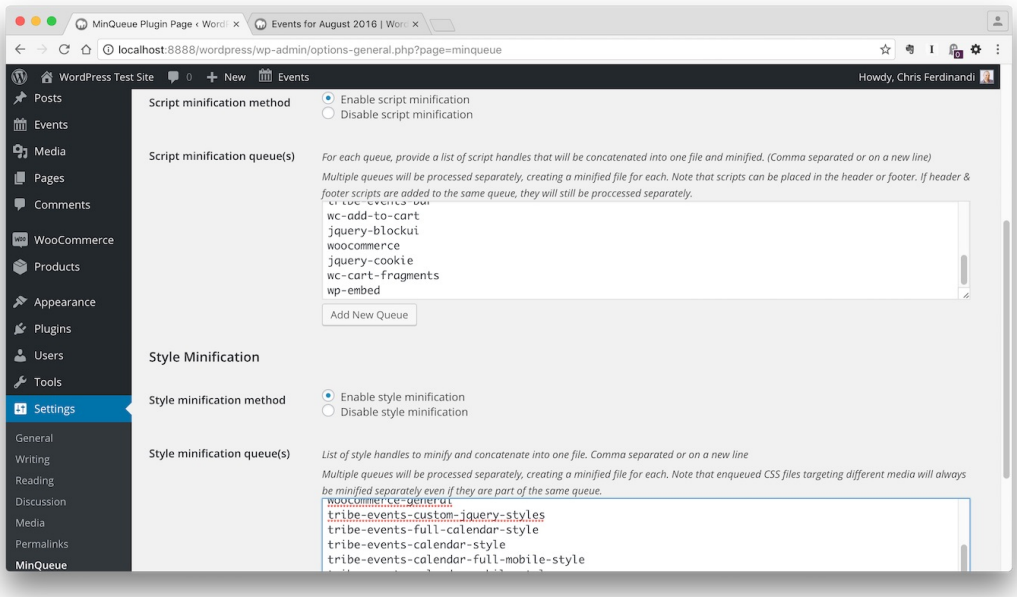
In the Dashboard, go to `Settings` > `MinQueue`, and check, "Enable the helper in the front end of the site." This provides you with a list of all the styles and scripts being loaded on any given page. Make sure you have the WordPress toolbar enabled on the front end (under `Users` > `Your Profile`) or the helper won't show up.

In a new tab, open up the page you tested for your baseline. Click `MinQueue` from the admin toolbar. A popup will appear with a list of "Enqueued Scripts" and "Enqueued Styles," color coded by whether they're loaded in the header or the footer. Under "Scripts," highlight and copy the entire list.



The MinQueue Helper

Back in your MinQueue settings, click the “Enable Script Minification” radio button, and paste in the entire list you copied from the front end. It doesn’t matter where the scripts load currently. MinQueue will figure it out.



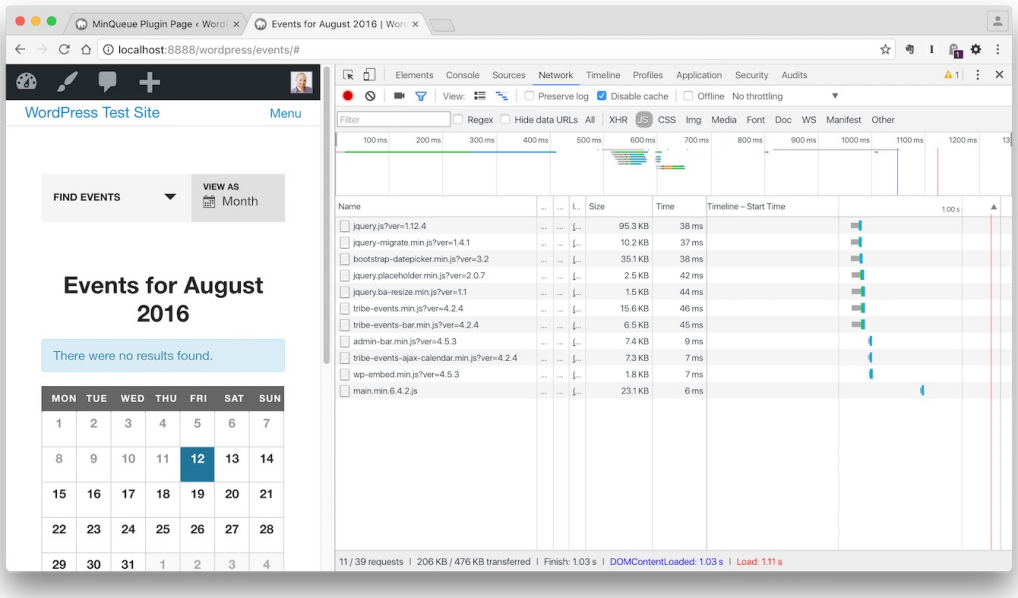
Adding files to MinQueue in the dashboard

Now repeat the process with the “Styles.” Copy the list, check the radio button to “Enable Style Minification,” and paste the list in. Scroll to the bottom of the page and click “Save.”

Checking that it’s working

Go back to the page you used for a baseline. With the “Network” tab open, reload the page again.

The number of styles and scripts should have gone down, and you should see CSS and JavaScript files that start with the name `minqueue-` and have a random string after them. These are your concatenated files.



Now there are only 3 scripts being loaded. That’s a huge difference!

You may find that not all files were concatenated. Some plugins that do the same thing as MinQueue are a lot more aggressive about this, but every single other one I tested ended up breaking my site or plugin functionality as a result.

MinQueue is a lot smarter about how it does things, making sure that dependencies and conditional loading are all respected. This is why it's my go-to concatenation plugin.

Concatenate with CodeKit

In the Codekit Boilerplate, there are two directories:

1. `js` for our JavaScript files.
2. `sass` for our CSS files.

Sass

We're using Sass to let us compile a bunch of modular files into a single file. You don't actually need to use Sass's advanced features to take advantage of this. All you need to do is change the file name extension from `.css` to `.scss` on your CSS files. This converts them to Sass files (CSS is valid Sass).

In the `components` subdirectory, the files are all prefixed with an underscore. This tells our Sass compiler not to compile these files directly, since we're referencing them somewhere.

That somewhere else is in our `main.scss` file. In this file, you'll see several lines of `@import "components/filename"`. This tells our compiler to fetch the contents of that file and drop it in. You'll notice that both the underscore and the extension are missing from the filename. Sass just figures it out.

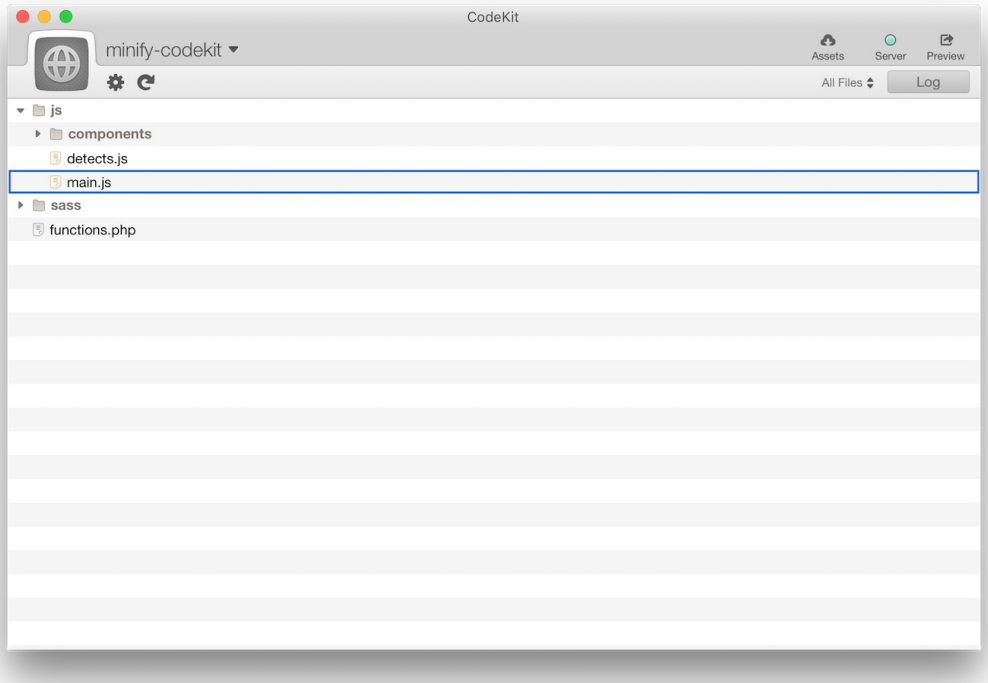
JavaScript

In our `js` directory, there are a few files and a `components` directory. The `components` directory contains all of the modular files we want to combine into a one file. The two primary files in the `js` directory—`detects.js` and `main.js`—are where we import those files into.

In those files, you'll see a few lines of `// @codekit-prepend "components/filename.js"` or `// @codekit-append "components/filename.js"`. These comments tell CodeKit to import the referenced file before or after the content of the file we're in. We're just using these files as placeholders for important content, but you can include code in them if you want.

Using CodeKit

In CodeKit, go to `File` > `Add Project`, locate your project on your computer, and click `Add`. You can also just drag-and-drop the project folder into the CodeKit window.

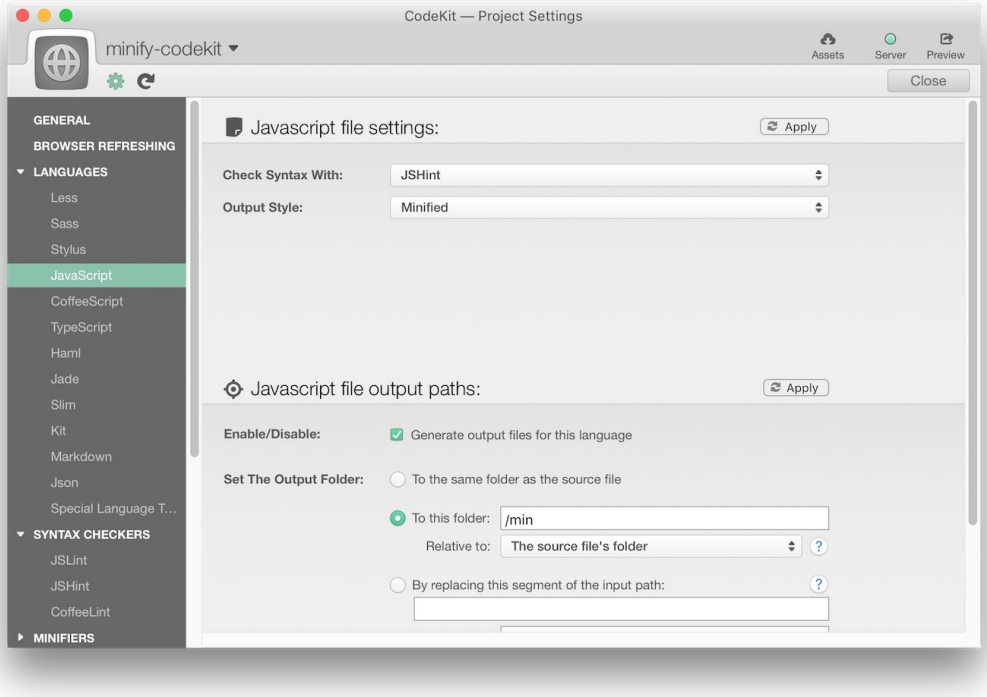


The CodeKit project window

There are a few ways to compile your code:

1. In the CodeKit window, right click the JavaScript file you want to compile and click `Process 'filename.js'`.
2. In the CodeKit window, right click the Sass file you want to compile and click `Compile 'filename.scss'`.
3. Any time you make a modification to one of your files, CodeKit will detect it and recompile your files.

CSS is compiled into the `css` directory, while JavaScript files are compiled into the `min` subdirectory under `js`.



The CodeKit JavaScript settings window

By default, JavaScript is minified. You can change this by clicking on the gear icon and selecting `JavaScript` from the `Languages` dropdown. Choose `Non-minified` from the output style dropdown.

Setting up Gulp

Note: *This is intended for people who are completely new to Gulp or command line. If that's not you, feel free to skip to the next chapter.*

First, we need to install a few files.

1. NodeJS[25] - NodeJS is what powers Gulp. The site will autodetect your operating system. Download and run the installer.
2. Gulp[26] - Click "Docs" and find "Getting Started." The first step, "Install gulp globally," is the one we want. Copy-and-paste the command there into Terminal without the leading `$:` `npm install --global gulp-cli`. Then click `return` on your keyboard.

Installing dependencies

Once that's done, we're good to go. Now, we want to navigate into our project folder in Terminal. This really tripped me up when I first started working with command line, but there's a simple trick that makes it really easy.

1. Type `cd` for "change directory" (don't forget the space afterwards).
2. Locate your project folder, and drag-and-drop it into the Terminal window. This automatically adds the file path for you.
3. Click `return` on your keyboard.

You're now in your project folder. Now we want to install of the dependencies we need to run our Gulp build.

1. Type `npm install`.
2. Click `return` on your keyboard.

Note: *You don't need to run* `npm install`*. I've included these files for you, but wanted you to know how to do it in case you want to get more involved in Gulp.*

There's now a `node_modules` directory in our project. This is where all of the packages and scripts that run our Gulp build live.

Now we can type `gulp` in Terminal and hit `return`, and our Gulp build will run.

Under the hood

There are two files that power our Gulp build.

1. `package.json`
2. `gulpfile.js`

`package.json`

`package.json` contains all of the information about our project—the name, version number, description, and so on. It also contains a list of our dependencies and the version number we should use. When we run `npm install`, it grabs all of the packages from this list.

`gulpfile.js`

`gulpfile.js` is where we tell Gulp exactly what to do when it runs.

At the top are a bunch of variables where we `require` packages. This is where we identify all of the packages we'll be using to do various things in our Gulp tasks.

I've also created a `paths` variable. This is where I specify where files are located and where they should go after they're compiled and processed.

The `banner` variable is where we create dynamically generated headers to get added to the top of our compiled files. This pulls information about the project from our `package.json` file, so it can be easily reused across projects.

The file also includes various tasks for building our scripts and styles, deleting our `dist` folder on each run so that we start clean, and listening for changes when we run `gulp watch`.

At the very end of our file are the task runners. This is where we pull together all of the build tasks into simple commands that can be run in Terminal.

Concatenate with Gulp

Unlike GUIs like CodeKit, Gulp can be configured in a seemingly endless number of combinations. For this course, I've put together a simple Gulp Boilerplate that we can use to concatenate our JavaScript and CSS.

Obviously if you're comfortable with Gulp or command line, go ahead and configure your own setup to best fit your needs.

In the Gulp Boilerplate, the `src` directory contains two subdirectories:

1. `js` for our JavaScript files.
2. `sass` for our CSS files.

Sass

We're using Sass to let us compile a bunch of modular files into a single file. You don't actually need to use Sass's advanced features to take advantage of this. All you need to do is change the file name extension from `.css` to `.scss` on your CSS files. This converts them to Sass files (CSS is valid Sass).

In the `components` subdirectory, the files are all prefixed with an underscore. This tells our Sass compiler not to compile these files directly, since we're referencing them somewhere.

That somewhere else is in our `main.scss` file. In this file, you'll see several lines of `@import "components/filename"`. This tells our compiler to fetch the contents of that file and drop it in. You'll notice that both the underscore and the extension are missing from the filename. Sass just figures it out.

JavaScript

In our `js` directory, we have two subdirectories:

1. `detects`
2. `main`

The files in `detects` will be compiled into `detects.js`, and the files in `main` will be compiled into `main.js`. The way this Gulp build is setup, you can rename those folders or add others. The files in the subdirectories always compile into a file named after the folder the files are in.

Using Gulp

1. Open up your terminal window.
2. Go into the project folder.
3. Run `gulp`, the default Gulp command.

There should now be a new `dist` directory in the project, with `css` and `js` subdirectories. Those subdirectories should now contain concatenated versions of your JavaScript and CSS files.

You can also run the `gulp watch` command. When this is running, anytime you make changes to your JS or CSS, Gulp will rerun the build and reconcatenate your files. Type `control+c` into the Terminal window to stop this from running.

Social Sharing Buttons

As we learned in the lesson on concatenation, it's actually faster to load a single 300kb file than it is to load three 100kb files, even though the total file weight is the same.

One unexpected way you can run into trouble here is with social sharing buttons. The social sharing buttons provided by sites like Facebook, Twitter, and Google+ all load a bunch of additional files behind the scenes and you might not even realize it.

I did a quick test, and here's what I found:

- Twitter: 5 files
- Google+: 11 files
- Facebook: 10 files (and 15 seconds to load them all)

Not only are these social sharing buttons slowing your sites down, they're not even that effective. One study found[27] that on both mobile and desktop visitors click these buttons less than one-percent of the time. On mobile, it's less than half a percent.

Using old-fashioned HTML

If you can, I would avoid social sharing buttons altogether. But if you need to include them, use static HTML instead of script-powered buttons.

Instead of this:

```
<a href="https://twitter.com/share" class="twitter-share-button" d
ata-via="ChrisFerdinandi">Tweet</a>

<script>!function(d,s,id){var js,fjs=d.getElementsByTagName(s)[0],
p=/^http:/.test(d.location)?'http':'https';if(!d.getElementById(id
)){js=d.createElement(s);js.id=id;js.src=p+'://platform.twitter.co
m/widgets.js';fjs.parentNode.insertBefore(js,fjs);}}(document, 'sc
ript', 'twitter-wjs');</script>
```

Do this:

```
<a target="_blank" href="https://twitter.com/intent/tweet?text=YOU
R-TITLE&url=YOUR-URL&via=TWITTER-HANDLE">
    Tweet
</a>
```

This simple link opens in a new window, and the query string provides information to the site about the type of information you're trying to share.

If you're interested in this approach, my Social Sharing project on GitHub[28] documents the markup you need to create sharing links for many of today's most popular sites, and also includes styles for lightweight branded buttons.

Loading CSS Better

Your CSS doesn't have to go in your `style.css` file. In fact, your CSS *shouldn't* go in your `style.css` file.

You still want to include a `style.css` file, but we just use it to hold our theme header information. All of our CSS will go in another file.

```
/**
 * Theme Name: my-project
 * Theme URI: https://github.com/myusername/my-project
 * Description: A description of my project
 * Version: 1.0.0
 * Author: My Name
 * Author URI: http://myurl.com
 * License: MIT
 */
```

Why? We want more control over how, when, and where our CSS loads.

CSS blocks rendering

Your stylesheet is necessary for rendering content properly, but while it's being downloaded and parsed, it also blocks any rendering from happening.

To get around this challenge, an emerging technique recommended by folks like Google[29] and Filament Group[30] is to inline your critical path CSS.

Yes, that's right. Inline your CSS. It works like this:

- Extract the styles that apply to above-the-fold content and load them inline in the `<head>`.

- Load your full stylesheet asynchronously so that the rest of your page can continue downloading and rendering.

Sounds wacky, but it makes a big difference, particularly on larger sites with big stylesheets.

How to decide what to inline

When I mentioned this technique to a few folks on Twitter, the most common question I got was how I decided what to inline.

If you use a JS task runner like Gulp[31] or Grunt[32], there are a few plugins that you can use to automate this: Critical by Addy Osmani[33] and Critical CSS by Filament Group[34].

There is also an online generator[35] if command line isn't thing.

I don't use any of these.

I tried using them, and found that they often left out styles important for rendering my layout on smaller or taller viewports (as in, my iPhone), so I create my critical CSS manually.

The magic number

The magic number you should care about is 14kb.

That's (give or take) how much data a server sends per round trip when the browser makes a request for a web page. You want your above-the-fold content—and its associated styles and scripts—–to weigh 14kb or less so that the browser can start rendering it as soon as that first packet of data is received.

How to inline and async your CSS

There's no browser-native way to asynchronously load CSS.

Fortunately, Filament Group created loadCSS[36], a lightweight JavaScript plugin that loads CSS asynchronously. Include loadCSS inline so that you don't have to wait for it download before running.

```html

```
<style>
    /* Your inline CSS... */
</style>
<script>
    function loadCSS(){ ... }
    loadCSS('/path/to/your/full.css');
</script>
```

You should also add a `<noscript>` fallback in the footer (again, to prevent render blocking) for browsers that don't support JavaScript or have it turned off.

```
<noscript>
    <link rel="stylesheet" type="text/css" href="/path/to/your/full.css">
</noscript>
```

The WordPress Way

Here's how you do that the WordPress way. In your `functions.php` file, you'll typically have a function you use to register and load your scripts:
```

```
// Load theme styles
function load_theme_styles() {
    wp_enqueue_style( 'theme-styles', get_template_directory_uri()
 . '/path/to/your/main.css', null, null, 'all' );
}
add_action('wp_enqueue_scripts', 'load_theme_styles');
```

Instead of doing that, we're going to hook into the `wp_head` action to inject some content inline:

```
// Load styles inline in the header
function load_critical_css_inline_header() {
    ?>
        <style type="text/css">
            /**
             * Your Critical Path CSS
             */
        </style>
        <script>
            function loadCSS(){ … }
            loadCSS( "<?php echo get_template_directory_uri() . '/
path/to/your/main.css'; ?>" );
        </script>
    <?php
}
add_action('wp_head', 'load_critical_css_inline_header', 30);
```

If your server supports `file_get_contents`, you can even use that to avoid having to copy and paste the contents of your critical path CSS and loadCSS into your `functions.php` file:

```
// Load styles inline in the header
function load_critical_css_inline_header() {
    ?>
        <style type="text/css">
            <?php echo file_get_contents( get_template_directory_u
ri() . '/path/to/your/critical.css' ); ?>
        </style>
        <script>
            <?php echo file_get_contents( get_template_directory_u
ri() . '/path/to/your/loadCSS.js' ); ?>
            loadCSS( "<?php echo get_template_directory_uri() . '/
path/to/your/main.css'; ?>" );
        </script>
    <?php
}
add_action('wp_head', 'load_critical_css_inline_header', 30);
```

You also want to hook into the `wp_footer` action to inline a fallback link:

```
// Load style fallbacks inline in the footer
function load_fallback_css_footer() {
    ?>
        <noscript>
            <link rel="stylesheet" type="text/css" href="<?php ech
o get_template_directory_uri() . '/path/to/your/main.css'; ?>">
        </noscript>
    <?php
}
add_action('wp_footer', 'load_fallback_css_footer', 30);
```

What about browser caching?

Using this approach means that the browser is no longer able to take advantage of having your stylesheet cached for reuse on subsequent pages and visits.

Fortunately, there is a relatively easy way to get around this, developed by the wonderfully talented folks at Filament Group: set a cookie when the full stylesheet is loaded asynchronously, and then check for that cookie on subsequent page visits. If it's there, skip the critical CSS inlining and just load the stylesheet via a traditional `<link>` element using `wp_enqueue_style`.

Setting the cookie with loadCSS

Setting the cookie requires one additional, super lightweight script from the Filament Group, onloadCSS, that runs a callback when the CSS file is loaded. onloadCSS comes bundled with loadCSS.

```
function loadCSS () { ... }
function onloadCSS () { ... }

var stylesheet = loadCSS( "<?php echo get_template_directory_uri()
 . '/path/to/your/main.css'; ?>" );

onloadCSS( stylesheet, function() {
    var expires = new Date(+new Date + (7 * 24 * 60 * 60 * 1000)).
toUTCString();
    document.cookie = 'fullCSS=true; expires=' + expires;
});
```

The WordPress Way

```
// Load theme styles
function load_theme_styles() {
    if ( isset($_COOKIE['fullCSS']) && $_COOKIE['fullCSS'] === 'tr
ue' ) {
        wp_enqueue_style( 'theme-styles', get_template_directory_u
ri() . '/path/to/your/main.css', null, null, 'all' );
    }
}
add_action('wp_enqueue_scripts', 'load_theme_styles');

// Load styles inline in the header
function load_critical_css_inline_header() {
    if ( !isset($_COOKIE['fullCSS']) || $_COOKIE['fullCSS'] !== 't
rue' ) :
    ?>
        <style type="text/css">
            <?php echo file_get_contents( get_template_directory_u
ri() . '/path/to/your/critical.css' ); ?>
        </style>
        <script>           <?php echo file_get_contents( get_templ
ate_directory_uri() . '/path/to/your/loadCSS.js' ); ?>       <?php
 echo file_get_contents( get_template_directory_uri() . '/dist/js/
onloadCSS.js' ); ?>
            var stylesheet = loadCSS( "<?php echo get_template_dir
ectory_uri() . '/path/to/your/main.css'; ?>" );
            onloadCSS( stylesheet, function() { // Set cookie });
        </script>
    <?php
    endif;
}
add_action('wp_head', 'load_critical_css_inline_header', 30);

// Load styles fallback inline in the footer
```

```
// Load styles fallback inline in the footer
function load_fallback_css_footer() {
    if ( !isset($_COOKIE['fullCSS']) || $_COOKIE['fullCSS'] !== 'true' ) :
    ?>
        <noscript>
            <link rel="stylesheet" type="text/css" href="<?php echo get_template_directory_uri() . '/path/to/your/main.css'; ?>">
        </noscript>
    <?php
    endif;
}
add_action('wp_footer', 'load_fallback_css_footer', 30);
```

Critical CSS with CodeKit

In our project folder, the `css` directory contains our `main.css` file. This file is created from the `main.scss` file in `src` > `sass`. This file tells Sass to import files from the `components` subdirectory.

Make a copy of `main.scss` and rename is `critical.scss`.

This approach requires us to know what's in all of our stylesheet components and which ones affect elements that are rendered above-the-fold. As long as you know that, this is relatively easy to do.

In my case, I only need:

- Normalize
- Grid
- Typography
- Overrides

I can delete everything else and save the file. CodeKit will run and compile a new `critical.css` file for me that's about half the size of my `main.css` file.

Critical CSS with Gulp

In our project folder, the `css` subdirectory under `dist` contains our `main.css` file. This file is created from the `main.scss` file in `src` > `sass`. This file tells Sass to import files from the `components` subdirectory.

Make a copy of `main.scss` and rename is `critical.scss`.

This approach requires us to know what's in all of our stylesheet components and which ones affect elements that are rendered above-the-fold. As long as you know that, this is relatively easy to do.

In my case, I only need:

- Normalize
- Grid
- Typography
- Overrides

I can delete everything else and save the file.

When you're done, go into the project directory in Terminal and run `gulp`. You should see a new `critical.css` file in `dist` > `css`.

Loading JavaScript Better

Like your CSS, JavaScript also blocks content rendering. Unlike CSS, it also stops all other files from downloading until it has been downloaded and parsed.

Normally, a browser will download two assets from the same source at once. This is important because it means that one large file won't become a bottleneck for other files. However, because JavaScript often makes additions or changes to the DOM, the browser stops everything and waits until it's parsed before continuing.

That's smart, but it also means that your scripts can have both real and perceived negative impacts on performance. So what can you do about it?

Be smart about script location in the markup

Feature detection scripts may affect how CSS renders content, so they should go in the `<head>`. Ideally they're small and lightweight, and you'll inline them so the browser doesn't have to wait for a file to download before running them.

Everything else should go in the footer.

What about `async` and `defer`?

`async` and `defer` are attributes you can add to your `<script>` tag:

```
<script src="path/to/your/main.js" async></script>
<script src="path/to/your/main.js" defer></script>
```

`async` tells the browser that it shouldn't block rendering or prevent other files from downloading while the script itself downloads. `defer` tells the browser to wait until everything else is loaded before downloading the file.

`defer` is well supported, but useless. It does the same thing as loading your scripts in the footer, so you might as well just put them there in the first place.

`async` is awesome, but has less support. On unsupported browsers, the script will still block rendering. And since most of your DOM manipulation scripts need to wait until the rest of the content has loaded and parsed anyways, you might as well just load your scripts on the footer.

The WordPress Way

In WordPress, we use the `wp_enqueue_script` function to load scripts. It accepts a handful of arguments.

The very last one is `$in_footer`. Setting it to `false` loads the script in the header. Setting it to `true` loads the script in the footer.

```
// Load theme scripts
function load_theme_scripts() {

    // Feature detects: loads in header
    wp_enqueue_script( 'theme-detects', get_template_directory_uri
() . '/path/to/your/detects.js', null, null, false );

    // Main scripts: loads in footer
    wp_enqueue_script( 'theme-scripts', get_template_directory_uri
() . '/path/to/your/main.js', null, null, true );

}
add_action('wp_enqueue_scripts', 'load_theme_scripts');
```

The Magic Number

Remember, 14kb is the magic number-the approximate amount of data sent in one roundtrip HTTP request.

If your feature detection scripts are small and fall below that number, it might be worth inlining them in the header instead of putting them in an external file. The avoids a download-blocking external JavaScript file.

If your server supports it, you can use the `file_get_contents` function instead of copy/pasting content into your `functions.php` file. That way, if you ever update your scripts the new content is automatically added.

```
// Load scripts inline in the header
function load_scripts_inline_header() {
    ?>
        <script>
            <?php echo file_get_contents( get_template_directory_u
ri() . '/path/to/your/detects.js' ); ?>
        </script>
    <?php
}
add_action('wp_head', 'load_scripts_inline_header', 30);
```

It's worth noting that this is *not* the WordPress way of doing things, but I think it's worth doing for the improved performance benefits.

Fixing Fonts

Loading web fonts often results in a Flash of Invisible Text (FOIT) that leaves the page unusable until it loads.

Ilya Grigorik, a web performance engineer at Google, reports:[37]

> 29% of page loads on Chrome for Android displayed blank text: the user agent knew the text it needed to paint, but was blocked from doing so due to the unavailable font resource. In the median case the blank text time was ~350 ms, ~750 ms for the 75th percentile, and a scary ~2300 ms for the 95th.

Even the median value of 350ms is an unacceptable performance hit. Fortunately, there's a simple fix.

A better way

A better approach is to have a system font load immediately, and then when your web font is available, switch over to that. It results in a repaint, but your content is immediately available to visitors.

To achieve this, we're going to use the same approach we use to for loading CSS better: `loadCSS.js`[38].

However, just loading asynchronously isn't enough. You'll still get a FOIT doing that—it will just happen a second or three later.

We also need a way to detect when the font is fully loaded. Fortunately, Font Face Observer from Bram Stein[39] does just that!

We'll load the font asynchronously, and use Font Face Observer to detect when it's loaded. When it loads, we'll add a class to the `<html>` element that we can hook into for styling.

```
body {
    font-family: Georgia, serif;
}

.fonts-loaded body {
    font-family: 'PT Serif', serif;
}
```

The WordPress Way

Traditionally, you would load your web font using the `wp_enqueue_style` function:

```
// Load theme fonts
function load_theme_fonts() {
    wp_enqueue_style( 'theme-fonts', 'https://fonts.googleapis.com
/css?family=Open+Sans', null, null, 'all' );
}
add_action('wp_enqueue_scripts', 'load_theme_fonts');
```

Instead, we want to hook into the `wp_head` action to inline loadCSS:

```
// Load fonts inline in the header
function load_fonts_inline_header() {
    ?>
        <script>
            // Inline our JS files
            function loadCSS(){ ... };
            // Also inline all of the contents from fontfaceobserv
er.js

            // Async load the CSS file
            loadCSS('//fonts.googleapis.com/css?family=PT+Serif:40
0,400italic,700,700italic');

            // When the font loads, as the .fonts-loaded class to
the <html> element
            var font = new FontFaceObserver('PT Serif');
            font.load().then(function () {
                document.documentElement.className += ' fonts-load
ed';
            });
        </script>
    <?php
}
add_action('wp_head', 'load_fonts_inline_header', 30);
```

This is technically not the “WordPress Way” of doing things. The Codex stress using `wp_enqueue_style` so that stylesheets are registered and accessible to plugins and other functions. But I think the performance benefits far outweigh the cost of deviating from the best practice here.

What about browser caching?

CSS files loaded from places like Typekit and Google Web Fonts get cached for about a week.

Once they're stored in the browser cache, we don't need to use the technique described above. We can load them the traditional way to avoid the font switch that happens once the font CSS loads asynchronously.

To make this happen, we'll set a cookie with Font Face Observer when the font loads. In our `functions.php` file, we'll use a different approach depending on whether or not that cookie exists.

```
// If cookie set, load font traditional way
function load_theme_fonts() {
    if ( isset($_COOKIE['fontsLoaded']) && $_COOKIE['fontsLoaded']
 === 'true' ) {
        wp_enqueue_style( 'theme-fonts', 'https://fonts.googleapis
.com/css?family=Open+Sans', null, null, 'all' );
    }
}
add_action('wp_enqueue_scripts', 'load_theme_fonts');


// Otherwise, load the font async
function load_theme_font_async() {
    ?>
        <script>
            <?php if ( !isset($_COOKIE['fontsLoaded']) || $_COOKIE
['fontsLoaded'] !== 'true' ) : ?>
                // Inline our JS files
                function loadCSS(){ ... };
                // Also inline all of the contents from fontfaceob
server.js
```

```
server.js

                // Async load the CSS file
                loadCSS('//fonts.googleapis.com/css?family=PT+Serif:400,400italic,700,700italic');

                // When the font loads, as the .fonts-loaded class to the <html> element
                var font = new FontFaceObserver('PT Serif');
                font.load().then(function () {
                    var expires = new Date(+new Date() + (7 * 24 * 60 * 60 * 1000)).toUTCString();
                    document.cookie = 'fontsLoaded=true; expires=' + expires;
                    document.documentElement.className += ' fonts-loaded';
                });
            <?php endif; ?>
        </script>
    <?php
}
add_action('wp_head', 'load_theme_font_async', 30);
```

Lightning Fast Tapping

Many mobile browsers introduce a 300ms delay when a user taps on a link or button. From Google:[40]

> Mobile browsers will wait approximately 300ms from the time that you tap the button to fire the click event. The reason for this is that the browser is waiting to see if you are actually performing a double tap.

This is why web apps feel *so* much slower than native apps. Don't believe me? Check out this video from Jake Archibald.[41]

The delay made a lot of sense when smartphones were first introduced and most websites weren't mobile-friendly. When viewing a desktop-oriented site on a small screen, double-tapping was a fast and easy way to zoom in and out of content areas.

While there are still far too many sites today that aren't mobile-friendly, many are, and waiting for a double tap on those sites doesn't make sense.

Set a viewport width

Setting a `<meta>` tag that defines the viewport width will remove the delay on both Chrome for Android (since version 32) and Safari on iOS (since version 9.1).

```
<meta name="viewport" content="width=device-width">
```

Mobile Safari also provides a way to remove the delay on an element-by-element basis. Jeremy Keith from Clearleft recommends using the follow CSS:

```
/**
 * Remove the tap delay in webkit
 */
a,
button,
input,
select,
textarea,
label,
summary {
    touch-action: manipulation;
}
```

Remove White Space

Minification is the process of removing spaces, line breaks and comments from your CSS and JavaScript. Though it might not seem like a big deal, removing all of that unused whitespace can have a big impact on the overall size of your files.

I ran a quick test and and minifying my stylesheet reduced its size by 75-percent.

There's just one problem

Minified files are essentially unusable if you need to make updates or changes. Everything is on a single line, and with JavaScript files, your intuitive variables are often renamed to single letters for even better size reduction.

Minified CSS

Keep an un-minified version of your file that you use for development. Save your minified version as a separate file with a `.min` suffix. This also makes debugging easier during the development phase.

```
main.css
main.min.css

main.js
main.min.js
```

The WordPress Way

In your `functions.php` file, update your `wp_enqueue_script` function with the new filename. This:

```
// Load theme files
function load_theme_files() {

    // Feature detects
    wp_enqueue_script( 'theme-detects', get_template_directory_uri
() . '/path/to/your/detects.js', null, null, false );

    // Main scripts
    wp_enqueue_script( 'theme-scripts', get_template_directory_uri
() . '/path/to/your/main.js', null, null, true );

}
add_action('wp_enqueue_scripts', 'load_theme_files');
```

Becomes this:

```
// Load theme files
function load_theme_files() {

    // Feature detects
    wp_enqueue_script( 'theme-detects', get_template_directory_uri
() . '/path/to/your/detects.min.js', null, null, false );

    // Main scripts
    wp_enqueue_script( 'theme-scripts', get_template_directory_uri
() . '/path/to/your/main.min.js', null, null, true );

}
add_action('wp_enqueue_scripts', 'load_theme_files');
```

Tools and Techniques

Removing all of the whitespace from your files by hand would be insane. There are a bunch of tools that make it easier:

GUIs

- CodeKit for Mac[42]
- Prepos for Windows[43]
- Google PageSpeed Insights[44]

Command Line

- Gulp[45]
- Grunt[46]

Plugins

- MinQueue[47]

Minify with CodeKit

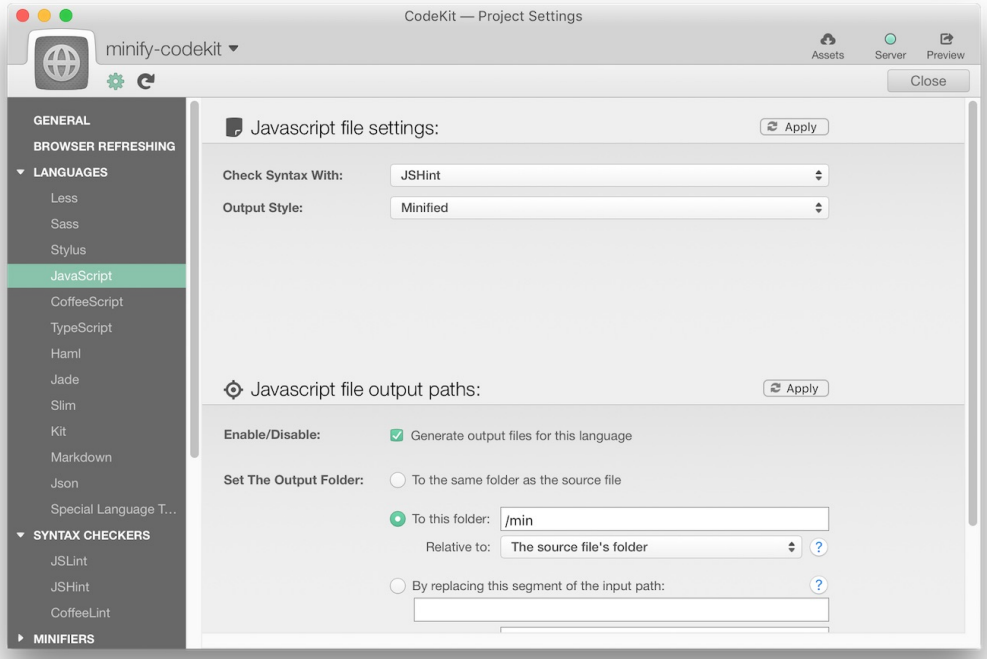
In the dummy project, there are two directories:

1. `js` for our JavaScript files.
2. `sass` for our CSS files.

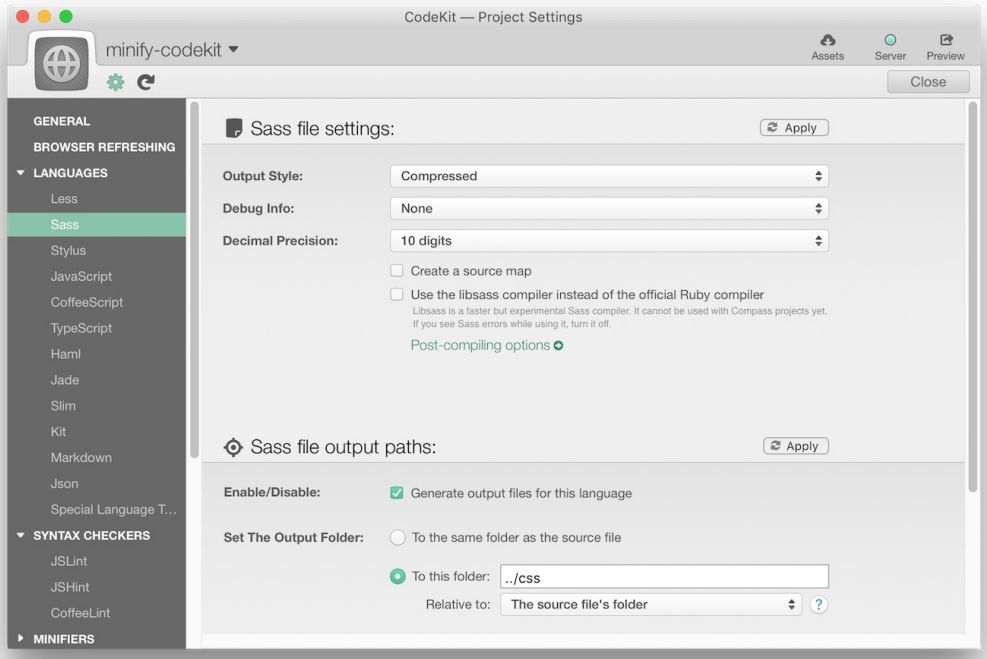
Sass

We're using Sass to let us compile a bunch of modular files into a single file. You don't actually need to use Sass's advanced features to take advantage of this. All you need to do is change the file name extension form `.css` to `.scss` on your CSS files. This converts them to Sass files (CSS is valid Sass).

Using CodeKit



Minifying JavaScript with CodeKit



Minifying CSS with CodeKit

1. Click the gear icon
2. Select `JavaScript` from the `Languages` dropdown. Choose `Minified` from the output style dropdown (this is the default).
3. Select `Sass` from the `Languages` dropdown. Choose `Compressed` from the output style dropdown.

Your files will now be minified when they compile. In my tests, files were about 50% - 75% smaller after minification.

Minify with Gulp

In the dummy project, the `src` directory contains two subdirectories:

1. `js` for our JavaScript files.
2. `sass` for our CSS files.

Sass

We're using Sass to let us compile a bunch of modular files into a single file. You don't actually need to use Sass's advanced features to take advantage of this. All you need to do is change the file name extension form `.css` to `.scss` on your CSS files. This converts them to Sass files (CSS is valid Sass).

Doing the heavy lifting

Let's open up `gulpfile.js` and take a look at a the Gulp modules that are doing the minification.

- Gulp Uglify is a JavaScript minification plugin.
- Gulp CSS Nano is a CSS minification plugin.
- Gulp Rename will rename our files. This let's us create both minified and unminified versions of our file with a single build.

The `banner` variable includes a smaller version of our header for inclusion in our minified files.

In our `build:scripts` task, we're:

1. Renaming our file with a `.min` suffix.
2. Passing it through Gulp Uglify.

3. Adding the minified banner.

In our `build:styles` task, we're:

1. Renaming our file with a `.min` suffix.
2. Passing it through Gulp CSS Nano.
3. Adding the minified banner.

In both cases, the full banner is removed as part of the minification process, so you don't need to worry about two banners showing up in the same file.

Using Gulp

Open up the project folder in Terminal and run `gulp`. You should now see both minified and unminified versions of our files in the `dist` subdirectory. In my tests, files were about 50% - 75% smaller after minification.

Minify with Google PageSpeed Insights

1. Visit Google PageSpeed Insights[48].
2. Paste in your URL.
3. Click `Analyze`.

There will be a handful of recommendations, but if you scroll down, you'll see "Minify Javascript" and "Minify CSS." If you expand those, they'll even show you how much you'll reduce your file size by.

The reason they know this is because Google actually minifies your files for you and compares them to your unminified versions.

You can download the minified versions by scrolling down to "Download optimized image, JavaScript, and CSS resources for this page." Click the link to save a zipped version of your files.

Google is a bit more conservative with their minification than Gulp or CodeKit, so their files aren't quite as small. But if you don't have access to a GUI and aren't comfortable using command line, this is a great alternative.

Smarter Images

Images now account for almost two-thirds of the total weight of an average web page[49]. They are an important part of an engaging web experience, but we can be smarter about how we use them.

The convergence of high-density displays and mobile, low-bandwidth computing puts two competing interests at odds. Your site performs better-particularly on spotty mobile networks—when images are smaller, but looks worse when viewed on high—resolution screens. And some of the highest resolution screens are found on tablets and smartphones, which often have unpredictable internet connections.

How do you balance these competing interests?

Picking the Right Format

Different image formats work better for different types of graphics.

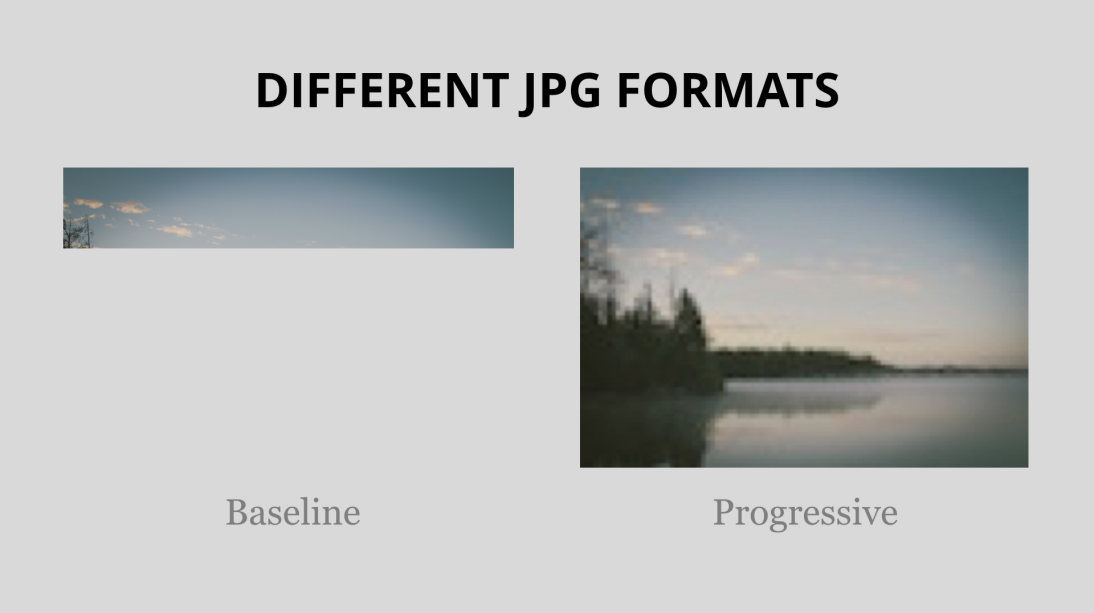
PNG is a lossless image format, so it keeps graphics sharp and crisp (as opposed to the lossy JPG format). For icons and simple images with clean lines, they can actually be more lightweight than JPGs.

But for photos and images with lots of visual noise, JPGs will be much smaller in size with comparable quality.

Picking the right JPG Format

The JPG actually has multiple formats. The most common on the web is the baseline JPG. Baseline JPGs start rendering at the top, and build down as they go.

An alternative format that was popular a decade ago and is seeing a bit of a comeback is the progressive JPG. Progressive JPGs build in layers. Initially, the full image in low resolution is displayed, and as the image renders, it becomes increasingly crisp and clear.



Baseline vs. Progressive JPGs

While progressive JPGs are typically a little smaller in size than baseline JPGs, their real advantage is that they appear faster to the user because they display more of an image at once. And on smaller screens, the lack of clarity on initial renders may not even be as noticeable.

While all browsers display progressive JPGs, some browsers do a better job than others[50]. For "non-supporting" browsers, the entire progressive JPG needs to download before it can be displayed, resulting in a worse experience than a baseline JPG.

Tools and techniques for creating progressive JPGs

- ImageOptim[51] is a Mac-only desktop app
- b64.io[52] is a web app

Smush your images

The metadata that photographs include——timestamps, color profiles, and such——can add quite a bit of weight. Smushing is the process of removing that metadata, and it can reduce the size of an image by more than 25-percent.

Both ImageOptim and b64.io also smush PNGs, JPGs, and GIFs.

Compress your JPGs

A high-quality photo can weigh upwards of 1mb or more. By compressing photos, you can reduce them down to a fraction of their original size while maintaining image quality.

A massive reduction in file size from compression

A JPG compression rate of 70 is considered high-quality for the web. WordPress uses a compression rate of 90 when it resizes your images.

This means that you could optimize a photo, upload to WordPress, and come out with a photo that's smaller in dimension but larger in weight than the one you

uploaded.

In version 4.5, WordPress increased their compression rate to 82, which is better, but still not aggressive enough for the web.

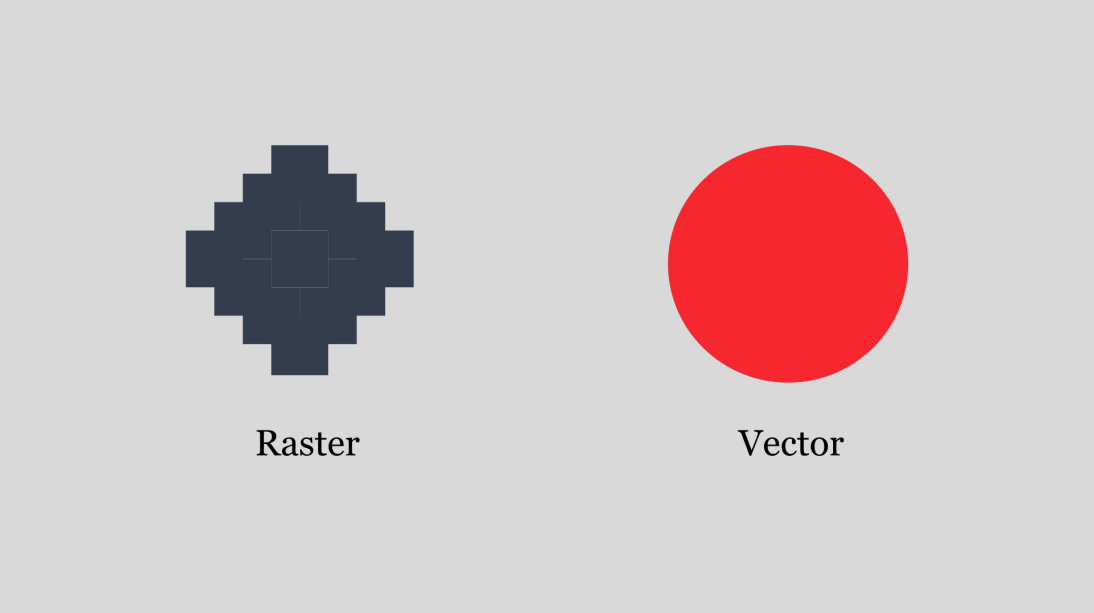
Image Compress and Sharpen[53] is a plugin I maintain on GitHub that lets you adjust the default compression rate in WordPress. It also converts JPGs to the progressive format.

If you'd like to recompress a bunch of existing images on your site, you can install one of the many thumbnail recreation plugins that exist. They'll recreate all of your resized images with the new compression settings.

Using SVGs

JPGs, PNGs, and GIFs are raster images. They have specific pixels assigned to specific locations that render images at a specific size.

SVGs, or scalable vector graphics, use geometric shapes and mathematical relationships to render images. As a result, you can scale them up or down in size infinitely and they never lose their crispness or degrade in quality.



For detail-heavy graphics, like photographs, they can end up being much larger than a high-resolution JPG, but they're perfect for icons and simple images. You can also style them with CSS, which makes it easy to do things like changes colors, add hover effects, and more.

How to use SVGs

There are many ways to use SVGs, including simply including them as the src in an `<img>` element.

```
<img src="twitter.svg">
```

Because SVGs are really markup files, my preferred method is to copy-and-paste the contents directly into my HTML. This gives me more flexibility in styling them, and eliminates an HTTP request.

```
<svg viewBox="0 0 320 320">...</svg>
```

You can also add classes to control styling with CSS.

```
<svg class="icon" viewBox="0 0 320 320">
    ...
</svg>
```

Browser Support

SVG is well supported in IE 9 and up, all modern browsers, and Opera Mini.

I like to include a simple feature text inline in the `<head>` to check for browser support. This adds an `.svg` class to the `<html>` element that you can hook into for styling and fallback content.

```
// SVG feature detection
var isSVG = !!document.createElementNS &&
            !!document.createElementNS('http://www.w3.org/2000/svg
', 'svg').createSVGRect;

// If SVG is supported, add `.svg` class to <html> element
if ( isSVG ) {
    document.documentElement.className += ' svg';
}
```

In unsupported browsers, SVGs will display an empty content area equivalent to the size of the SVG. I use a little CSS to prevent this from happening:

```css
/**
 * Hide icons by default to prevent blank spaces
 * in unsupported browsers
 */
.icon {
    display: inline-block;
    fill: currentColor;
    height: 0;
    width: 0;
}

/**
 * Display icons when browser supports SVG.
 * Inherit height, width, and color.
 */
.svg .icon {
    height: 1em;
    width: auto;
}
```

I also include a class for fallback text that's only displayed when SVGs aren't supported and for people using assistive devices like screen readers.

```
/**
 * Hide fallback text if browser supports SVG
 */
.svg .icon-fallback-text {
    border: 0;
    clip: rect(0 0 0 0);
    height: 1px;
    margin: -1px;
    overflow: hidden;
    padding: 0;
    position: absolute;
    width: 1px;
}
```

In the example below, visitors on browsers that support SVG will see a Twitter icon, while unsupported browsers and screen readers will display “Follow me on Twitter” as fallback text:

```
<svg xmlns="http://www.w3.org/2000/svg" class="icon" viewBox="0 0
32 32"><path d="M32 6.076c-1.177.522-2.443.875-3.77 1.034 1.354-.8
13 2.395-2.1 2.886-3.632-1.27.752-2.674 1.3-4.17 1.593C25.75 3.796
 24.044 3 22.157 3c-3.627 0-6.566 2.94-6.566 6.565 0 .515.058 1.01
6.17 1.496-5.456-.275-10.294-2.89-13.532-6.86-.565.97-.89 2.096-.8
9 3.3 0 2.278 1.16 4.287 2.922 5.465-1.076-.034-2.088-.33-2.974-.8
2v.082c0 3.18 2.262 5.834 5.265 6.437-.55.15-1.13.23-1.73.23-.422
0-.833-.04-1.234-.118.835 2.608 3.26 4.506 6.133 4.56-2.248 1.76-5
.08 2.81-8.155 2.81-.53 0-1.052-.032-1.566-.093 2.904 1.863 6.355
2.95 10.063 2.95 12.076 0 18.68-10.004 18.68-18.68 0-.285-.007-.56
8-.02-.85C30.006 8.55 31.12 7.394 32 6.077z"/></svg>
<span class="icon-fallback-text">Follow me on Twitter</span>
```

The WordPress Way

The WordPress Media Uploader does not allow SVGs, and TinyMCE can strip them out, too.

WordPress SVG[54] is a plugin I maintain on GitHub that enables SVG uploads in the WordPress Media Uploader. It also adds a shortcode you can use to inline your uploaded SVG files and pass in classes.

Optimize your SVGs

SVGs often have a lot of junk and metadata in them, put there by your graphic design software. Optimizing them can reduce their overall size by 50-percent.

SVGO[55] is a great cross-platform tool for optimizing your SVGs. If you're not a fan of command line, Jake Archibald created SVGOMG[56], a GUI for SVGO.

Want to learn more?

This lesson only scratched the surface of what you can do with SVGs.

If you want to learn more (and you absolutely should!), Sara Soueidan[57] is hands-down the leading authority on SVG in our industry. She's constantly putting out great tutorials and pushing the boundaries of what you can do with them.

Responsive Images

A common technique for making images responsive is to give them a `max-width` of `100%`:

```
img {
    max-width: 100%;
    height: auto;
}
```

But why would you send the same image to a five-year old feature phone or a tiny smart watch that you would to a modern laptop attached to 32-inch, high-definition monitor?

Ideally, you would send different images to different viewports.

Background images

You can load viewport-aware background images using media queries. You can even use different images based on screen density[58].

```
.hero {
    background-image: url("/path/to/the/image-small.jpg");
}

@media (min-width: 30em) {
      .hero {
            background-image: url("/path/to/the/image-medium.jpg")
;
      }
}

@media (min-width: 60em),
       (min-width: 30em) and (-webkit-min-device-pixel-ratio: 2),
       (min-width: 30em) and (min-resolution: 192dpi) {
      .hero {
            background-image: url("/path/to/the/image-large.jpg");
      }
}
```

Most browsers will only load the image required for the current viewport size. If a visitor resizes their browser or rotates their device, and this triggers a new break point, a subsequent image download will occur, but the benefits of using smaller sizes outweigh the edge case of a potential second download.

Of course, you're not always going to use a background image.

The `<picture>` element and `srcset`

Being able to specify which version of an image to show at which breakpoint sounds great. And in many cases, it is (think art direction across a variety of viewports).

But it also involves a lot of math and decision making on your part that really should be offloaded to the browser. Rather than figuring which image to send and when, imagine if you could just tell a browser the sizes it has to choose from and let it figure out which image works. This opens up the door for some interesting opportunities:

- Browsers could automatically figure out which image would look best given the device's screen resolution.
- Visitors could choose to prioritize performance over aesthetics, forgoing, for example, the high-resolution images when in low bandwidth conditions or if trying to conserve data.
- Browsers could determine that grabbing the highest required resolution for an image and downsizing on orientation change makes more sense than a second download (or vice-versa).

Today, you as the developer are making decisions that really belong in the hands of the visitor and their browser. Fortunately, a pair of solutions are in the works.

<picture>

The `<picture>` element let's you serve different images based on breakpoints. We provide an image, and the media query conditions under which to use it.

```
<picture>
    <source srcset="path/to/image-xlarge.jpg"
media="(min-width: 60em)">
    <source srcset="path/to/image-large.jpg"
media="(min-width: 40em)">
    <img srcset="path/to/image-standard.jpg">
</picture>
```

This is great, but we're still making decisions that belong with the browser and the visitor who's using it.

`srcset`

The `srcset` attribute allows you to provide a few image choices, and share some information with the browser that helps it decide which image to choose. It can be used with a standard `<img>` element, or with the `<picture>` element.

```
<img src="path/to/image-small.jpg"
     srcset="path/to/image-large.jpg 1024w,
             path/to/image-medium.jpg 640w,
             path/to/image-small.jpg 320w"
     sizes="(min-width: 40em) 33.3vw,
            100vw"
>
```

Here, we provide a list of images and their pixel width in the `srcset` attribute. We also use the `sizes` attribute to pass in information about our layout, telling the browser what our breakpoints are and how wide the content area is at those breakpoints.

The browser uses that information to pick the image that makes the most sense. Since it's still just an `img` tag, we also include a `src` attribute that serves a fallback for browseres that don't support `srcset`.

Browser Support

Currently, support for srcset isn't great[59], and support for the element is even worse[60], but support will get better in the near future.

However, the Picturefill polyfill from Filament Group[61] let's you use both methods today. When support improves in the future, you can simply remove the polyfill.

To learn more about these approaches and how to use them, check out "Srcset and sizes" by Eric Portis[62].

The markup kind of sucks

The biggest issue with these responsive image solutions is how cumbersome the markup is. We went from the simple image tag:

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

To this:

```
<img src="path/to/image-small.jpg"
     srcset="path/to/image-large.jpg 1024w,
             path/to/image-medium.jpg 640w,
             path/to/image-small.jpg 320w"
     sizes="(min-width: 40em) 33.3vw,
            100vw"
>
```

Here's the good news: WordPress makes this insanely easy.

The WordPress Way

WordPress creates responsive images automatically since version 4.4. Any image you add to the content editor gets converted into a responsive image using `srcset`. WordPress also loads the Picturefill polyfill.

If you're using an older version of WordPress, install the RICG Responsive Images Plugin[63]. This plugin was forked into the WordPress core in version 4.4, and does the exact same thing that new version of WordPress do today.

Gzip

Your server can actually compress your website files——a process known as gzipping ——before sending them to the browser. This results in about a 70-percent reduction in website size[64].

Gzipping requires a server configuration. Some web hosts do this automatically, while others require you to activate it.

You can test your site to see if it's being gzipped using gzipWTF[65]. It will tell you which of your files are being gzipped and which ones are not.

Note: *Third-party scripts and styles may not be gzipped, and unforunately, there's not much you can do to change that.*

Enabling gzip

If your site isn't already enabled, you can activate it in your `.htaccess` file.

This is located in the root directory of WordPress, the same place where your `wp-config.php` file is. If you're on a Mac, this file might not be visible. Files that start with a `.` are hidden, so we need to display hidden files to see it.

Displaying hidden files on a Mac

Open up your Terminal app, paste in the following code, and hit return:

```
defaults write com.apple.finder AppleShowAllFiles YES
```

Next, hold down the Option key and right click your Finder icon in the dock. The click `Relaunch`. Now your hidden files should be visible.

If at some point in the future you no longer want to see hidden files, repeat this process, but use this code in Terminal instead:

```
defaults write com.apple.finder AppleShowAllFiles NO
```

Modifying `.htaccess`

Your `.htaccess` file will contain some WordPress-specific stuff near the top:

```
# BEGIN WordPress
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteBase /wordpress/
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /wordpress/index.php [L]
</IfModule>

# END WordPress
```

We'll be adding code after the `#END WordPress` comment.

Pull up the `.htacess` file from HTML5 Boilerplate[66] and do a search for `Compression`. You want to copy-and-paste in everything between the opening and closing `<IfModule mod_deflate.c>` tags.

Different servers are configured differently. This contains a set of checks and commands to enable gzip based on different configurations.

Once you're done, retest your site in gzipWTF to check that it's working. All of your files should now be gzipped.

Expires Headers

Expires headers tell browsers to keep static assets stored locally so that the visitor's browser doesn't have to re-download them every time they visit your site.

While things like markup and RSS feeds should be refreshed on each page load, files like CSS, JavaScript, and images may not change for months or even years. You can ask browsers to hold on to these types of files for longer periods of time.

Enabling Expires Headers

This is also something that's done using the `.htaccess` file. We will again be using the code in HTML5 Boilerplate's server configuration file[67]. Search for `Expires headers`.

This specifies how long the browser should retain various file formats.

Copy-and-paste the code between the opening and closing `<IfModule mod_expires.c>` tags into your `.htacess` file after the `#END WordPress` comment.

When you're done, run your page through Google PageSpeed Insights [68] to check that it's working. If it is, you shouldn't see "Leverage Browser Caching" under things to be fixed.

Cache-Busting

One problem with this approach: if you update a file, the visitor may already have an older version stored locally and won't load the new file with your changes. Fortunately, there's a really simple fix: giving a file a different name forces the browser to re-download it.

Instead of using a generic file name like this

```
main.min.js
```

Append a version number to the filename like this:

```
main.min.1.0.0.js
```

This is something that you can do manually, or you can use a tool like Gulp to automate it for you.

Cache-Busting Manually

1. Open up `functions.php`
2. Locate the function where you're loading the script or style.
3. Update the filename with the version number suffix.

Cache-Busting with Gulp

I can update my version number in a single place—my `package.json` file—and have it automatically update all of my filenames, the theme header in my `style.css` file, and references to all of my files in my `functions.php` file.

How this works

In our `gulpfile.js`, the `theme` key under the `path` variable points to an empty `style.css` file in our `src` directory. We're going to grab that file and dynamically add our theme header with our version number.

Similarly, there's a `banner` key for `theme` as well. This is our dynamic theme header, and it pulls in information—including the version number—from our `package.json` file.

`gulpfile.js` tasks

In `build:scripts` and `build:styles`, the `.min` suffix that's added to our minified files is appended with `. + package.version`. This grabs the version number from `package.json` and adds it to our filename. Anytime you update the version number, the filename changes to match.

The `build:theme` task grabs that empty `style.css` file in `src`, adds our dynamic theme banner, and renders the new file in the primary directory.

`functions.php`

The `wp_get_theme()` function gets a bunch of info about our WordPress theme. We want to set it to a variable—`$theme`—and use it in our filenames.

Where we load and inline files in `functions.php`, append the filename with `$theme->get( 'Version' )`. This gets the version number from our theme header in `style.css` and adds it to the filename.

```
function load_theme_files() {
    $theme = wp_get_theme();
    wp_enqueue_style( 'theme-styles', get_template_directory_uri()
 . '/path/to/your/main.min.' . $theme->get( 'Version' ) . '.css',
null, null, 'all' );
}
add_action( 'wp_enqueue_scripts', 'load_theme_files' );
```

Putting it all together

Now, all you have to do is update your version number in `package.json` and run `gulp`. All of your files will be renamed, your theme will get an updated header, and references to your files will update to match the new file names.

Cache-Busting: A Hybrid Approach

If command line isn't your preferred way of doing development, but you'd still like to automate your cache-busting implementation, there's actually a hybrid approach you can use that reduces the amount of work you have to do by quite a bit.

In the theme header of your `style.css` file, make sure that the `version` for your theme matches the version suffix for your updated files.

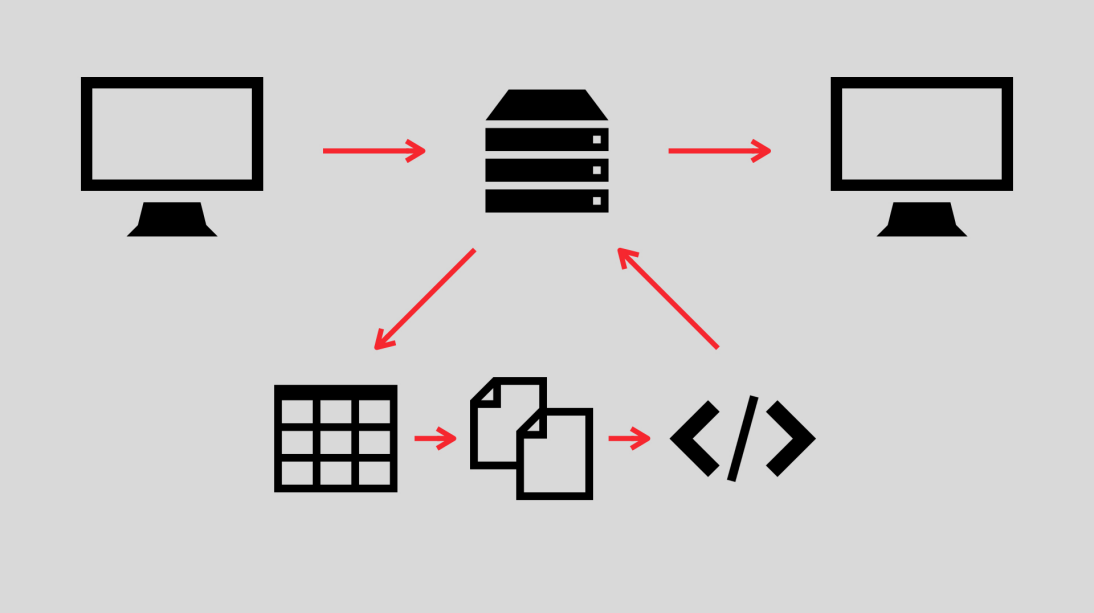
Now, open up `functions.php`. The `wp_get_theme()` function gets a bunch of theme about our WordPress theme. We want to set it to a variable—`$theme`—and use it in our filenames.

Where we load and inline files in `functions.php`, append the filename with `$theme->get( 'Version' )`. This gets the version number from our theme header in `style.css` and adds it to the filename.

```
function load_theme_files() {
    $theme = wp_get_theme();
    wp_enqueue_style( 'theme-styles', get_template_directory_uri()
 . '/path/to/your/main.min.' . $theme->get( 'Version' ) . '.css',
null, null, 'all' );
}
add_action( 'wp_enqueue_scripts', 'load_theme_files' );
```

Now, anytime you change your files, update the filename and version number in `style.css`, and your new files will automatically load. Just make sure you add the same version suffix to all files.

Pre-Compile Static HTML



When a browser requests a page from your site, your server:

1. Grabs content from the database.
2. Gets the template files from your theme.
3. Compiles the database content and template files into HTML.
4. Sends this static HTML file to the browser.

This happens every single time someone visits any page on your site. If someone visits your homepage, the visits another page, then comes back to your homepage 30 seconds later, that page is still recompiled, even though it's unlikely anything changed.

This process is slow and increases your time to first byte. Particularly on inexpensive, shared hosting, it can add quite a bit of latency to your site.

Tell WordPress to compile HTML files ahead of time

Fortunately, you can tell WordPress to pre-compile static HTML files ahead of time.

This is something that a plugin will handle for you. You don't even need to think about it.

Two often recommended plugins are W3 Total Cache[69] and WP Super Cache[70]. I personally find both of these plugins far too complicated.

My favorite plugin for this is Comet Cache[71]. I've used it on every site I've built for the last two years. While it does have some advanced settings, the only thing you need to do is activate it and check "Enable Comet Cache." The plugin handles the rest, and it just works.

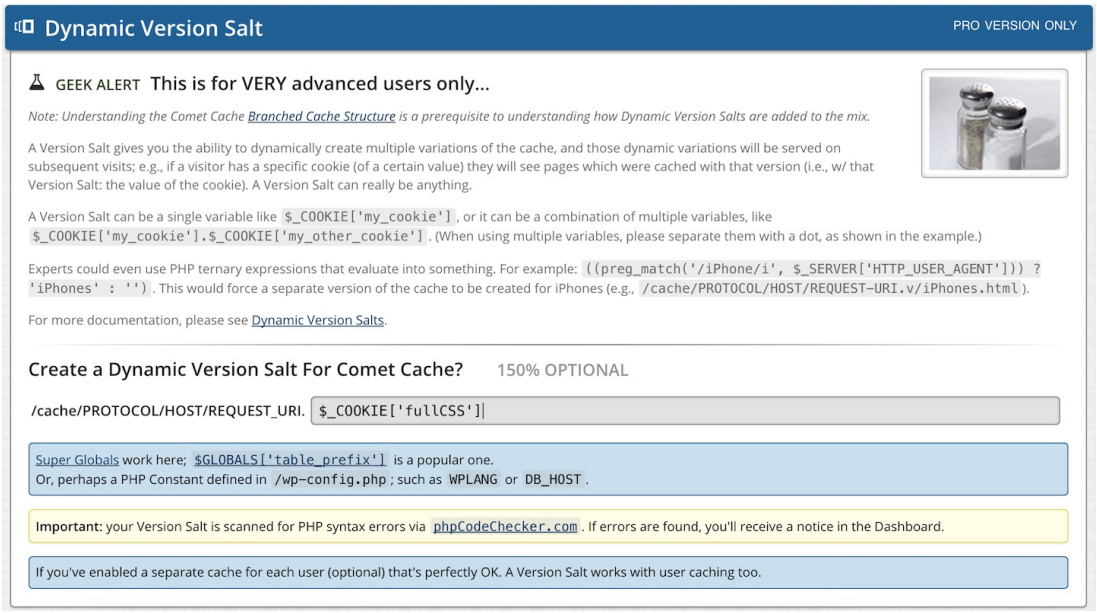
Caching and critical path CSS

When you're a caching plugin, the server sends the same pre-compiled HTML file to everyone. If you're using the critical path CSS technique, the server never gets a chance to check for the cookie that says the CSS file was loaded, so it will load asynchronously for everyone. The same is true of the asynchronous font loading technique.

```
// Load theme styles
function load_theme_styles() {
    if ( isset($_COOKIE['fullCSS']) && $_COOKIE['fullCSS'] === 'tr
ue' ) {
        wp_enqueue_style( 'theme-styles', get_template_directory_u
ri() . '/path/to/your/main.css', null, null, 'all' );
    }
}
add_action('wp_enqueue_scripts', 'load_theme_styles');

// If cookie set, load font traditional way
function keel_load_theme_files() {
    if ( isset($_COOKIE['fontsLoaded']) && $_COOKIE['fontsLoaded']
 === 'true' ) {
        wp_enqueue_style( 'keel-theme-fonts', '//fonts.googleapis.
com/css?family=PT+Serif:400,400italic,700,700italic', null, null,
'all' );
    }
}
add_action('wp_enqueue_scripts', 'keel_load_theme_files');
```

Fortunately, Comet Cache provides a way for you to create different pre-compiled versions based on whether or not a cookie is set.



Adding dynamic caching via the Comet Cache GUI

If you pay for the pro version, this is as simple as adding the cookie to the GUI in the WordPress Dashboard. You can also do this manually (for free) with some PHP and an FTP client.

Creating a dynamic version salt

In your `wp-content` directory, we want to add a new file, `critical-path-salt.php` to the `ac-plugins` sub-directory (if that folder doesn't exist, create it):

`wp-content/ac-plugins/critical-path-salt.php`

In the `critical-path-salt.php` file, we want to do two things:

1. Modify the `$version_salt` variable based on whether or not our cookie exists.
2. Add a filter against Comet Cache's salting function to update our variable.

```
// Prevent people from accessing this file directly
if ( !defined( 'WPINC' ) ) {
```

```
if ( !defined( 'WPINC' ) ) {
    exit( 'Do NOT access this file directly: '.basename(__FILE__)
);
}

/*
 * Dynamically generate a new version salt based on the cookie
 */
function critical_css_salt_shaker( $version_salt ) {

    // Check for fullCSS cookie
    if ( isset($_COOKIE['fullCSS']) && $_COOKIE['fullCSS'] === 'tr
ue' ) {
        $version_salt .= 'fullcss'; // Give users with cached CSS
files their own variation of the cache.
    } else {
        $version_salt .= 'inlinecss'; // A default group for all o
thers.
    }

    // Check for fontsLoaded cookie
    if ( isset($_COOKIE['fontsLoaded']) && $_COOKIE['fontsLoaded']
 === 'true' ) {
        $version_salt .= 'fontsloaded'; // Give users with cached
CSS files their own variation of the cache.
    } else {
        $version_salt .= 'fontsnotloaded'; // A default group for
all others.
    }

    return $version_salt;

}
```

```php
/**
 * Add a new version salt
 */
function critical_css_salt_plugin() {
	$ac = $GLOBALS['comet_cache_advanced_cache'];
	$ac->addFilter('comet_cache_version_salt', 'critical_css_salt_shaker');
}
critical_css_salt_plugin();
```

Putting it all together

Let's quickly recap all of the things we learned in this course:

1. Combine files
2. Replace social sharing buttons
3. Inline Critical Path CSS
4. Load JS in the footer
5. Remove whitespace
6. Load fonts asynchronously
7. Pick the right image format
8. Smush and compress images
9. Switch to SVG
10. Use responsive images
11. Remove the tap delay
12. Enable gzip
13. Enable expires headers
14. Use a caching plugin

Remeasure your site performance

Once you've finished making your performance improvements, lets measure your site's performance in WebPagetest[72] again to see how you've improved.

As a reminder, here are our Start Render targets:

- On a cable connection, you want this number to be 1,000ms (1 second) or lower.
- On a 3g connection, you want this number to be 3,000ms (3 seconds) or lower.
- On an EDGE network, you want this number to be 5,000ms (5 seconds) or lower.

If you take nothing else away from this course, I hope you'll remember that perceived performance matters more than actual performance.

About the Author

Hi, I'm Chris Ferdinandi. I help people make things for the web.

I love pirates, puppies, and Pixar movies, and live near horse farms in rural Massachusetts. I run Go Make Things with Bailey Puppy, a lab-mix from Tennessee.

You can find me:

- On my website at GoMakeThings.com.
- By email at chris@gomakethings.com.
- On Twitter at @ChrisFerdinandi.

1. https://blog.kissmetrics.com/speed-is-a-killer/↩
2. http://www.slideshare.net/stubbornella/designing-fast-websites-presentation↩
3. https://blog.mozilla.org/metrics/2010/04/05/firefox-page-load-speed-%E2%80%93-part-ii/↩
4. http://httparchive.org/interesting.php↩
5. http://searchengineland.com/its-official-google-says-more-searches-now-on-mobile-than-on-desktop-220369↩
6. http://venturebeat.com/2014/01/22/65-of-all-email-gets-opened-first-on-a-mobile-device-and-thats-great-news-for-marketers/↩
7. http://venturebeat.com/2016/01/27/over-half-of-facebook-users-access-the-service-only-on-mobile/↩
8. http://blogs.hbr.org/cs/2013/05/the_rise_of_the_mobile-only_us.html↩
9. https://blog.kissmetrics.com/loading-time/). After five seconds, [that number skyrockets to 74-percent](http://bradfrost.com/blog/post/performance-as-design/. After five seconds, that number skyrockets to 74-percent↩
10. http://www.nngroup.com/articles/response-times-3-important-limits/↩
11. http://www.webpagetest.org/↩
12. https://developers.google.com/speed/pagespeed/insights/↩
13. https://whatdoesmysitecost.com/↩
14. http://timkadlec.com/↩
15. http://www.itu.int/en/Pages/default.aspx↩
16. http://data.worldbank.org/↩

17. https://www.youtube.com/watch?v=fJ0C4zN5uOQ↩
18. https://incident57.com/codekit/↩
19. https://prepros.io/↩
20. http://gulpjs.com/↩
21. http://gruntjs.com/↩
22. https://wordpress.org/plugins/minqueue/↩
23. https://codex.wordpress.org/WordPress_Optimization/WordPress_Performance↩
24. https://wordpress.org/plugins/minqueue/↩
25. https://nodejs.org↩
26. http://gulpjs.com/↩
27. http://moovweb.com/blog/anyone-use-social-sharing-buttons-mobile/↩
28. https://github.com/cferdinandi/social-sharing↩
29. https://developers.google.com/speed/pagespeed/service/PrioritizeCriticalCss↩
30. http://www.filamentgroup.com/lab/performance-rwd.html↩
31. http://gulpjs.com↩
32. http://gruntjs.com↩
33. https://github.com/addyosmani/critical↩
34. https://github.com/filamentgroup/criticalCSS↩
35. http://jonassebastianohlsson.com/criticalpathcssgenerator/↩
36. https://github.com/filamentgroup/loadCSS↩
37. https://www.igvita.com/2015/04/10/fixing-the-blank-text-problem/↩

38. https://github.com/filamentgroup/loadCSS↩

39. https://github.com/bramstein/fontfaceobserver↩

40. https://developers.google.com/mobile/articles/fast_buttons↩

41. https://www.youtube.com/watch?v=AjUpiwvIa5A↩

42. https://incident57.com/codekit/↩

43. https://prepros.io/↩

44. https://developers.google.com/speed/pagespeed/insights/↩

45. http://gulpjs.com/↩

46. http://gruntjs.com/↩

47. https://wordpress.org/plugins/minqueue/↩

48. https://developers.google.com/speed/pagespeed/insights/↩

49. http://httparchive.org/interesting.php#bytesperpage↩

50. http://calendar.perfplanet.com/2012/progressive-jpegs-a-new-best-practice/↩

51. https://imageoptim.com/↩

52. http://b64.io↩

53. https://github.com/cferdinandi/gmt-image-compress-and-sharpen↩

54. https://github.com/cferdinandi/gmt-wordpress-svg↩

55. https://github.com/svg/svgo↩

56. https://jakearchibald.github.io/svgomg/↩

57. https://sarasoueidan.com/↩

58. https://css-tricks.com/snippets/css/retina-display-media-query/↩

59. http://caniuse.com/#feat=picture↩

60. http://caniuse.com/#feat=picture↩

61. http://scottjehl.github.io/picturefill/↩

62. https://ericportis.com/posts/2014/srcset-sizes/↩

63. https://wordpress.org/plugins/ricg-responsive-images/↩

64. https://developer.yahoo.com/performance/rules.html↩

65. http://gzipwtf.com/↩

66. https://github.com/h5bp/html5-boilerplate/blob/master/dist/.htaccess↩

67. https://github.com/h5bp/html5-boilerplate/blob/master/dist/.htaccess↩

68. https://developers.google.com/speed/pagespeed/insights/↩

69. https://wordpress.org/plugins/w3-total-cache/↩

70. https://wordpress.org/plugins/wp-super-cache/↩

71. https://wordpress.org/plugins/comet-cache/↩

72. http://www.webpagetest.org/↩