SiteGround

sitepoint

JUMP START WEB PERFORMANCE

BY CRAIG BUCKLER

LEARN THE SECRETS OF BLAZING FAST WEBSITES

Jump Start Web Performance

- **Product Manager:** Simon Mackie
- **Technical Editor:** James Hibbard
- **English Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

Level 1, 110 Johnston St, Fitzroy
VIC Australia 3065
Web: www.sitepoint.com
Email: books@sitepoint.com
ISBN 978-1-925836-33-2 (ebook)
Printed and bound in the United States of America

About Craig Buckler

Craig is a freelance developer, author, and speaker who never shuts up about the web.

He started coding in the 1980s when applications had to squeeze into a few kilobytes of RAM. His passion for the Web was ignited in the mid 1990s when 28K modems were typical and 100KB pages were considered extravagant.

Over the past decade, Craig has written 1,200 tutorials for SitePoint as web standards evolved. Despite living in a technically wondrous future, he has never forgotten what could be achieved with modest resources.

About SitePoint

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

Table of Contents

Preface

Despite working on the web every day, few developers have a good word to say about the monster they've created. Achingly slow sites with annoying overlays, cookie agreements, instant notifications, and obtrusive ads litter the web landscape.

While there may be some excuses for complex web applications, there's little justification for sluggish content-based and ecommerce sites. People are notoriously impatient, and an unresponsive site receives fewer visitors and conversions.

This book provides advice, tips, and best practice for improving website performance.

Who Should Read This Book?

The performance options described in the following chapters range from quick, five-minute configuration changes to major website overhauls. We primarily concentrate on front-end activities and server configurations to optimize the code delivered to a browser.

Some back-end tips are provided, but this is often specific to your application, framework, database, and usage patterns. Server-side performance can often be improved with additional or more powerful computing resources.

Ideally, everyone involved in a project would consider performance from the start. Somewhat understandably, that rarely occurs, because no one can appreciate the speed of a website or application before it's been created. Many of the tips can therefore be applied after your project has been delivered.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout

this book to signify different types of information. Look out for the following items.

Code Samples

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

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

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

```
function animate() {
  ⋮
  new_variable = "Hello";
}
```

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

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

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

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

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

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

Supplementary Materials

- https://www.sitepoint.com/community/ are SitePoint's forums, for help on any tricky problems.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

Chapter

1

Web Performance Matters

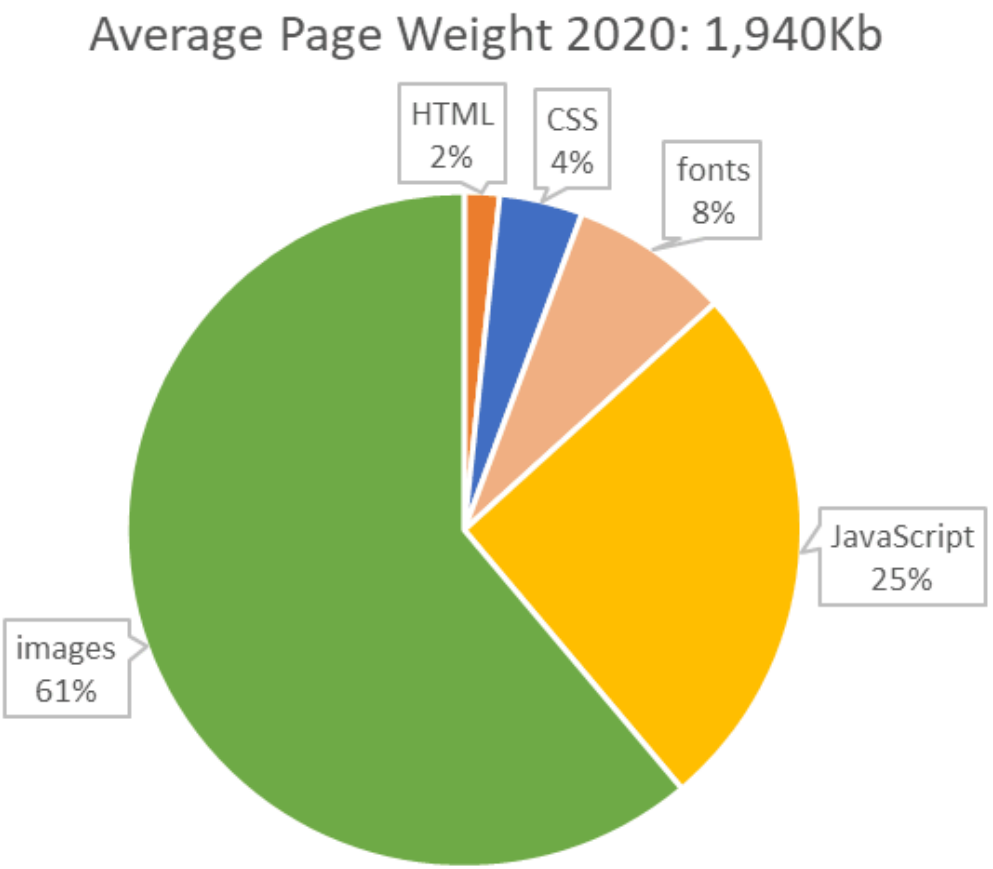
The Cost of Poor Performance

Web obesity, slow downloads, and poor performance hit everyone—site users, online business owners, and even those who've never accessed the Web.

User Costs

At the start of 2020, the average web page comprises:

- 27KB of HTML content
- 64KB of CSS split over seven style sheets
- four fonts, totaling 122KB
- 410KB of JavaScript in 20 source files
- 31 images, requiring 980KB of bandwidth (*a third of these are off screen and may never be viewed!*)



1-1. Average page weight in 2020

The total: 1,940KB of data made over 74 HTTP requests, which takes seven seconds to fully appear on the average user's desktop worldwide. This increases to a frustrating 20 seconds on mobile devices. (Source: HTTP Archive, which analyzes five million popular content websites.)

Downloading this web page on a typical mobile phone costs US users $0.20. Those browsing in Vanuatu, Mauritania, and Madagascar pay more than 1% of their daily income for the privilege of viewing a single page—despite it containing a mere 27KB of potentially readable content. (Source: whatdoesmysitecost.com.)

Business Costs

Slow, bloated pages are bad for business:

1. The larger the page download, the slower the user experience, and the less likely that person will consider making a purchase or returning.
2. 55% of visitors use a mobile device. These have more limited capabilities and may be connected to a slower network, which exacerbates the problem. (Source: statcounter.com.)
3. Google's page speed algorithms downgrade slower sites, which harms search engine optimization efforts.
4. More data results in higher hosting, storage, and bandwidth costs.
5. The larger your codebase, the longer it takes to update and maintain.

Environmental Costs

The Internet consumes 420TWh—*or up to 10%*—of the world's electricity consumption. This accounts for 4% of global greenhouse gas emissions, which is comparable to the aviation industry. Taking the web infrastructure and traffic into account, a single page load is estimated to emit 1.3g of CO^2. (Source: websitecarbon.com.)

While the Web has reduced energy use by providing a virtual alternative to travel and postage, those 1MB hero images still have an environmental impact.

The Reason for the Woeful Web

How have badly performing sites become ubiquitous when they cost more money to run, receive fewer visitors, and decrease conversions?

The main reason: *performance is a lower priority compared to other features*.

It's easy to add more stuff. Optimizing or removing unnecessary junk is more difficult. We fear breaking the site or visitor usage patterns, so it becomes easier

to make excuses for not addressing performance.

Excuse #1: "We Don't Have a Performance Problem!"

Are you using the latest PC or smartphone on a fast network? Try a mid-range, two-year-old device. Try limiting bandwidth to your country's average speed. Try using your site on a VPN or hotel Wi-Fi.

Excuse #2: "Our Users Never Complain?"

Possibly because they abandon your site and never return. Few people bother to make a complaint when competing content and services are a few clicks away.

Excuse #3: "Our Users Have High-end Devices"

This presumption becomes a self-fulfilling prophecy when a site can only be viewed by those with a recent device on a fast connection.

Would your revenues increase if more people could access your service? Are you considering explosive web growth in markets such as Asia, Africa, and South America, where smartphone and network capacity may be more limited?

Excuse #4: "Our Customers Use Modern Browsers"

There's a common myth that 1% of users disable JavaScript or block other browser features such as images or CSS—for example, those using screen readers. This could be a considerable number, yet it's used as an excuse to discriminate against certain groups in order to make development easier and avoid addressing performance on less capable devices.

In reality, it's not 1% of *users* blocking assets, but 1% of *visits*. Every user will eventually encounter a situation where something breaks, such as when:

- one or more files fail to download
- firewalls or ISPs block certain assets
- a screen reader or older device is used

- the browser doesn't support specific features
- a browser extension blocks, breaks, or modifies code
- the browser disables JavaScript on slow connections

It *is* possible to build robust, high-performance applications that can cope with these situations. Does yours?

Excuse #5: "We'll Address Performance Later"

> *Premature optimization is the root of all evil.*

This quote is attributed to Donald Knuth, from his paper "Structured Programming with `go to` Statements". It relates to programmers wasting time on efficiencies that aren't an immediate problem—such as a small start-up trying to ensure their application scales to millions of users.

The full quote in context:

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

Before you reach millions of users, you need to ensure the first few dozen people want to use your product. Back-end server or database inefficiencies are unlikely to be a major issue in the early days.

However, front-end performance can make or break an application. It could be part of your "critical 3%", and it's easier to address optimization from the start.

Excuse #6: "Some Systems Require More Bandwidth and Processing"

Complex web apps such as Gmail, maps, social networks, games, and image galleries will require more bandwidth and processing capacity than content websites. Performance remains a critical issue, but a higher page weight and slower load time can be expected.

However, the HTTP Archive crawls articles and online shops. It's not looking at web applications. The average 2MB website page weight is the equivalent of half of all Shakespeare's plays, or the 1993 disk-based distribution of DOOM—*on a single page often containing only a few paragraphs of content*. There's little excuse for not addressing performance.

Excuse #7: "Expanding Page Weight is the Price of Progress"

This may be true for some edge cases. However, developers strived to keep pages under 100KB during the dial-up days of the late 1990s. Has web content become 20x better or faster since that time?

Excuse #8: "Slimming Pages Means Dumbing Down, with Fewer Features and Effects"

Performance can often be improved with minimal effort and no loss of functionality.

You can do more with less as web browsers evolve. Consider the CSS3 `border-radius` property: adding rounded corners now requires a few bytes of code compared to the multiple image shenanigans of a decade ago.

Excuse #9: "Improving Performance Increases Complications and Maintenance"

Removing unused or unnecessary images, videos, fonts, CSS, and JavaScript will simplify your site. It should result in fewer complications and less maintenance.

Excuse #10: "Our Client is Happy!"

Clients employ you for your expertise, and pitching optimization as a selling point will differentiate your business from others.

Web performance is an essential part of a web developer's job. A little effort can reap considerable rewards for everyone:

- less code is required
- users receive a slicker experience
- search engine rankings improve
- conversions increase
- hosting costs decrease

No one will criticize you for building a super-fast, responsive site!

Where do I Start?

The next chapter introduces tools to help you identify issues. This is followed by a delicious buffet of food-inspired chapters:

Chapter 3: Quick Snacks A selection of simple, practical, cost-effective performance solutions that can be implemented on any site in minutes.

Chapter 4: Simple Recipes Some more complex development options that may take a few hours or days to implement but could have a larger positive impact.

Chapter 5: Life-changing Diets More radical development considerations and techniques that are best adopted from the start of your project.

Page weight reduction and optimization tips are generally grouped into similar concepts, with the easiest or most beneficial covered first.

Those attempting to improve an existing site should read each chapter in order. Those starting a new project may benefit from reading the chapters in reverse order, since more radical approaches then become viable.

Chapter

2

Testing Tools

Admitting your site has a performance problem is the first step on the road to recovery! This chapter provides a list of testing tools to help you understand issues using real data, showing how:

1. the largest assets can be discovered
2. the slowest network responses can be identified
3. the reasons for poor browser performance can be diagnosed

It may be necessary to run tests a few times to establish a measurable performance baseline. The same tests can then be rerun to evaluate performance improvements—*or deteriorations*—after code has been updated.

Create a Test Plan

You should test your websites and applications for defects, ease of use, accessibility, quality assurance, and other factors. Evaluating performance is no different, and it's best to follow a plan that lays out:

1. who's responsible for running tests
2. what tools and settings will be used for each test
3. how results will be recorded and fed back into the development process

Use performance analysis tools manually at first to understand the reports and determine optimization priorities.

As your processes evolve, it *may* become possible to automate these tests so developers are warned about potential problems and perhaps blocked from committing poorly optimized code. Chapter 5 describes several options for improving your workflow with build processes and performance budgets.

Automated vs Manual Testing

Automated tests are never a substitute for manual user testing! Tests are good at repeating operations to report faults, but they're unlikely to discover issues you weren't expecting.

It's possible to build a wholly unusable site that fully passes automated testing. For example, a button could trigger a fast change to the page that shows skeleton content, while the real results take an hour to appear!

Identify Performance Bottlenecks

Knowing you have a performance problem is one thing. Finding and fixing the causes is another matter.

The first step is to identify whether the fault occurs server-side or client-side. A slow network response, either on the initial page load or during an Ajax request, will normally indicate a server issue. Database queries are often the culprit, but you'll need to prove that! Existing tools and logs can help, but it may be necessary to output diagnostic information to a file in a similar way to other debugging activities.

Client-side issues can be diagnosed using browser developer tools, as described below. Performance is affected when the browser has considerable work to do—such as a long-running JavaScript function, a DOM update that causes the page to re-layout, or CSS changes that affect many elements. The tips in Chapters 3, 4, and 5 provide solutions to typical problems.

Performance Tool Concepts

Most of the tools described in this chapter diagnose a particular "page" in your site within the context of a web browser. They primarily analyze front-end performance, although a back-end server or database could be to blame for a large or slow response. (Chapter 4 provides further information about potential back-end issues and database tools.)

The Browser Rendering Process

When a site or app is first accessed by a user, the following steps occur:

1. The browser makes an HTTP request for a specific URL. Under the hood, several network processes are taking place to resolve the domain name to an IP address and route the request to a server.

2. The server receives and parses the request. It will reference a specific URL and may have data appended as a query string, in the HTTP header, or message body. It returns a response which, in this case, we'll presume is HTML content.

3. The browser starts to receive HTML data, which it parses. The document may reference further assets, such as images, fonts, style sheets, and JavaScript, which trigger additional HTTP requests to the same or another server.

4. Eventually, the browser has enough information to start the rendering process. Behind the scenes, it has started to build the HTML DOM (document object model) which defines the page in a hierarchical tree structure. Style calculations also determine which CSS rules apply to each DOM node, and a CSSOM (CSS object model) is created for JavaScript interaction.

5. The browser initiates the **layout** (or **reflow**) phase. This calculates the dimensions of each element and how it affects the size or positioning of elements around it.

6. The layout is followed by a **paint** phase. This draws the visual parts of each element onto separate layers—that is, text, colors, images, borders, shadows, and so on.

7. Finally, a **composite** phase draws each layer to the screen in the correct order.

The page is now in an initial viewable state. During or after the render, JavaScript can run to make further HTTP requests (Ajax or WebSocket calls), perform calculations, update the DOM, or apply CSS rules. This could trigger further layout, paint, and/or composite phases.

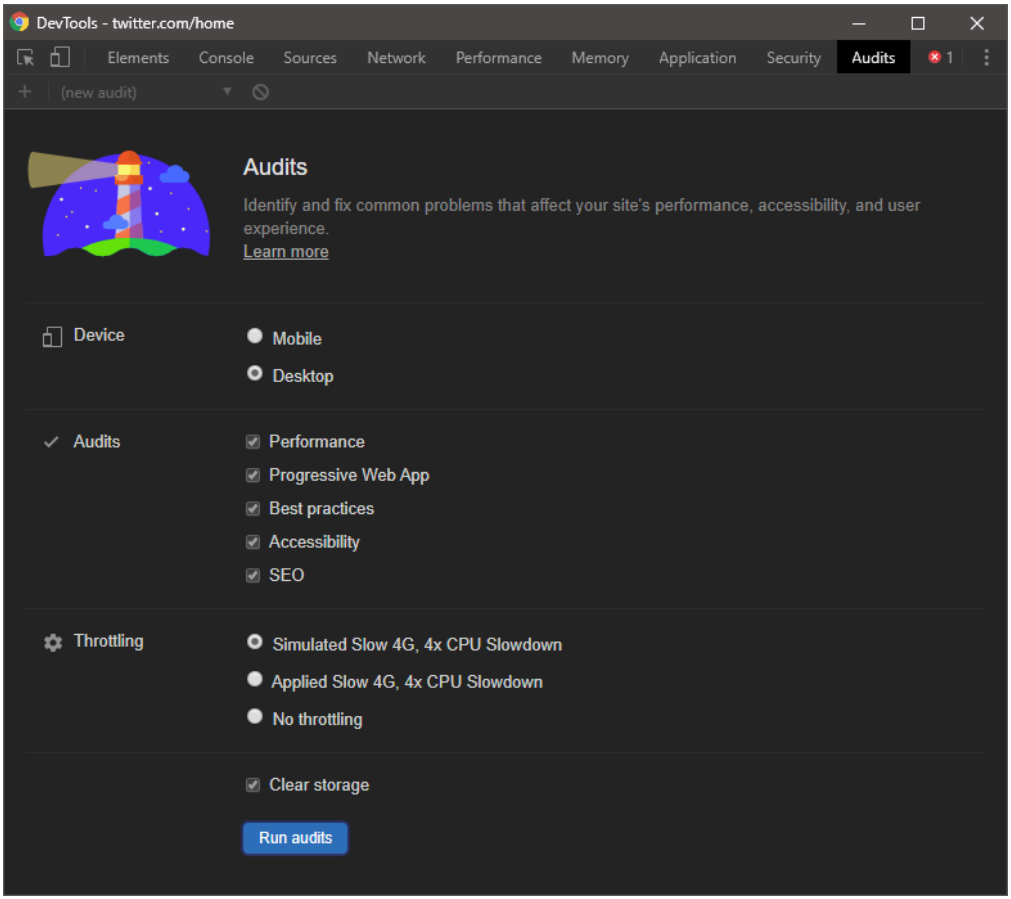
Most tools make reference to these phases, while also introducing their own metrics that often combine two or more stages. In essence, the fewer steps you require, the better the performance. Taking steps to minimize HTTP requests and reduce browser processing will result in a snappier user experience.

Google Lighthouse/Chrome Audits

Lighthouse is an open-source tool that helps evaluate the performance and quality of your page or app. You can access it in the following ways:

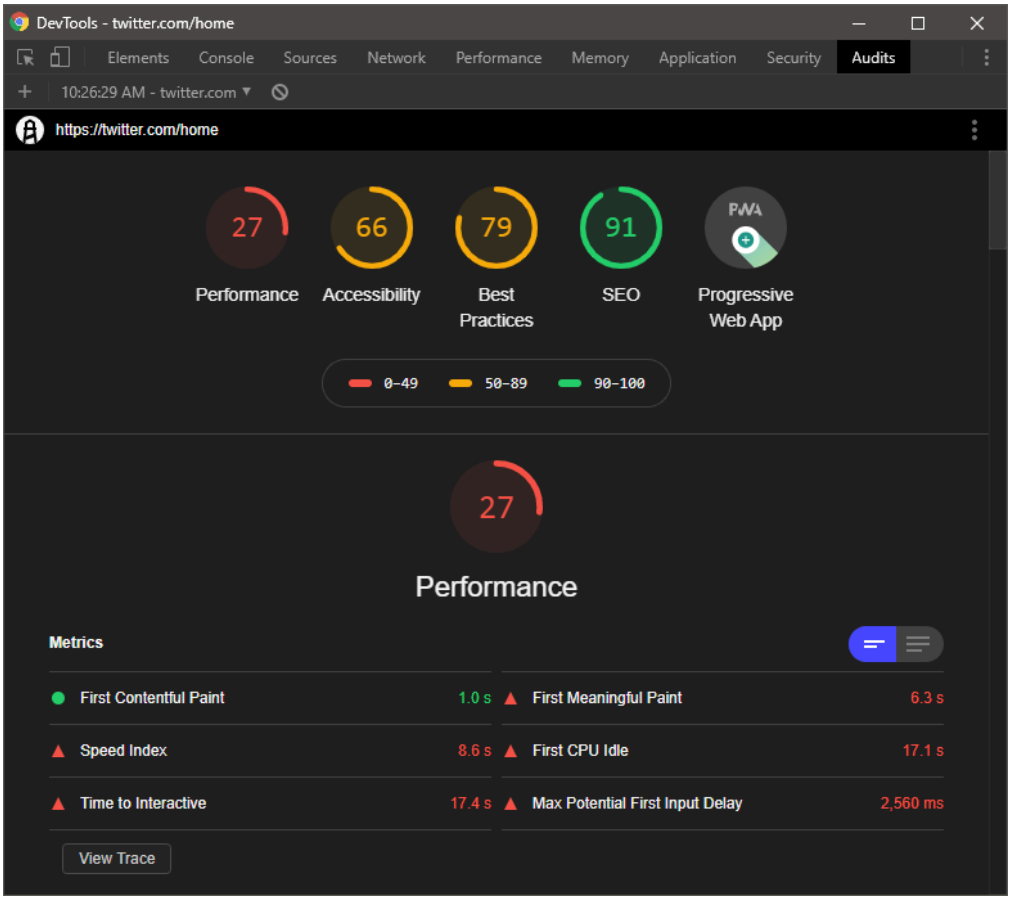
1. from within Chrome's DevTools
2. as the online web.dev or PageSpeed Insights tools
3. as a Node.js module, through which command-line and automated tests can be executed

It's easiest to start with Chrome's DevTools. Navigate to any page in Chrome, press `Ctrl|Cmd` + `Shift` + `I` or `F12` to open the DevTools panel, and click the **Audits** tab:



2-1. The Lighthouse Audits tab

Select the device, audit types, network speed, and check **Clear storage** to ensure there's no influence from browser caching. The results screen appears shortly after clicking **Run audits**:



2-2. Lighthouse audit results

Browser Extensions

Browser extensions can affect results, but Chrome will warn you about potential issues. It may be necessary to run tests in an Incognito window since it disables extensions by default.

The Performance, Accessibility, Best Practices, and SEO scores provide a quick-view percentage result, which can be clicked for more information.

Performance information includes:

Performance Metric	Description
First Contentful Paint	the time when the first text or image is painted
First Meaningful Paint	when the primary content is visible
Speed Index	how quickly the contents of a page are visibly populated
First CPU Idle	the time when the main thread is able to handle input
Time to Interactive	the time taken for the page to become fully interactive
Max Potential First Input Delay	the time when the browser is able to respond to interaction

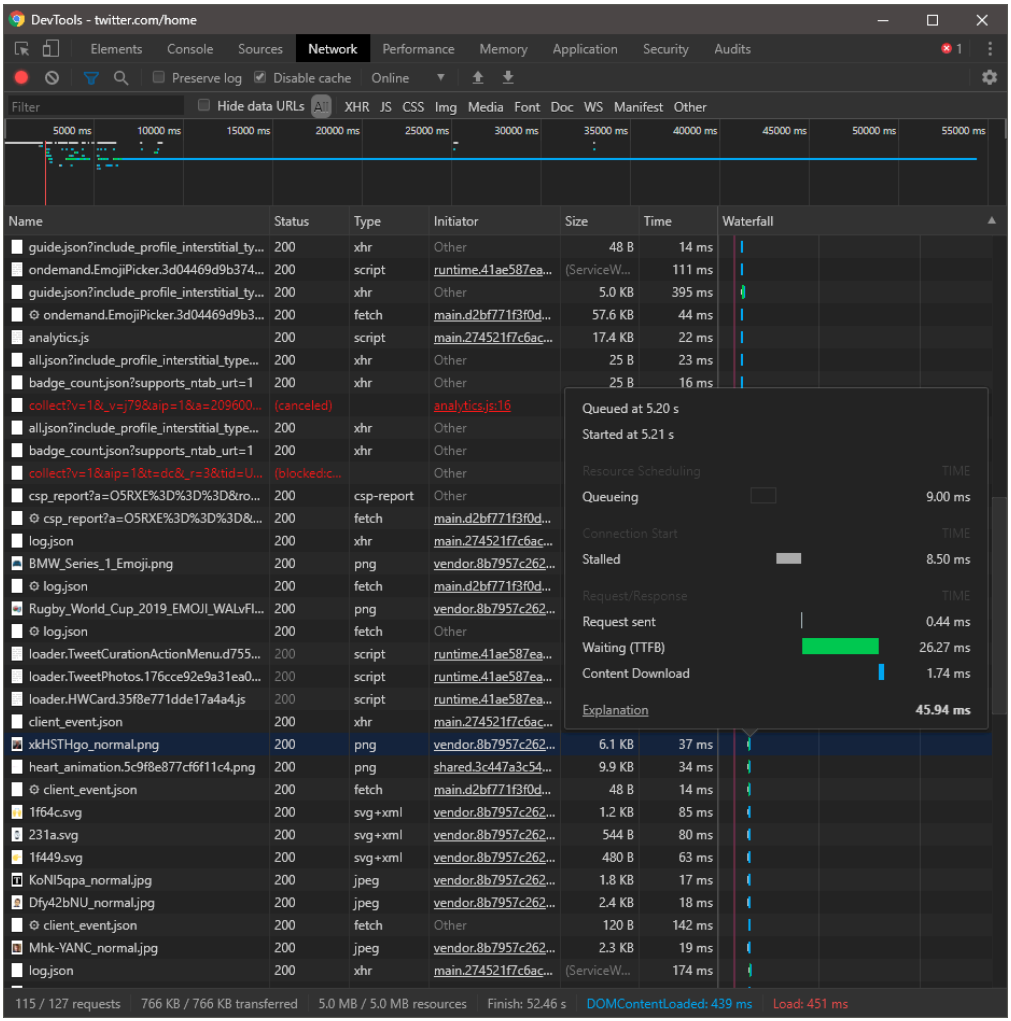
The lower the figures, the better the page performance. This is followed by an **Opportunity** section, which suggests potential improvements and estimated savings.

Progressive Web Apps

The PWA section in Chapter 4 describes the benefits of progressive web app technologies, which allow a web application to be installed and cached, and to work offline.

DevTools' Network Panel

The developer tools in most browsers provide a **Network** panel that shows a log of all network activity during page load and any subsequent file, Ajax, or WebSocket data flows.



2-3. DevTools' Network panel

The status bar at the bottom summarizes the number of requests, total data transfer (possibly compressed), the total size of all uncompressed resources, the total download time, and the time when the document DOMContentLoaded and window load events were triggered.

Further options are provided at the top:

- **Preserve log**: don't clear the log between page loads

- **Disable cache**: load all files from the network to make a better assessment of first-time page access
- **Throttle network speed**: select or define download speed profiles

Assets can be displayed, hidden, or reordered by clicking a table heading. Ordering by size or download time will help find the largest or most costly resources.

The **Filter** box allows you to search for specific assets or enter criteria such as:

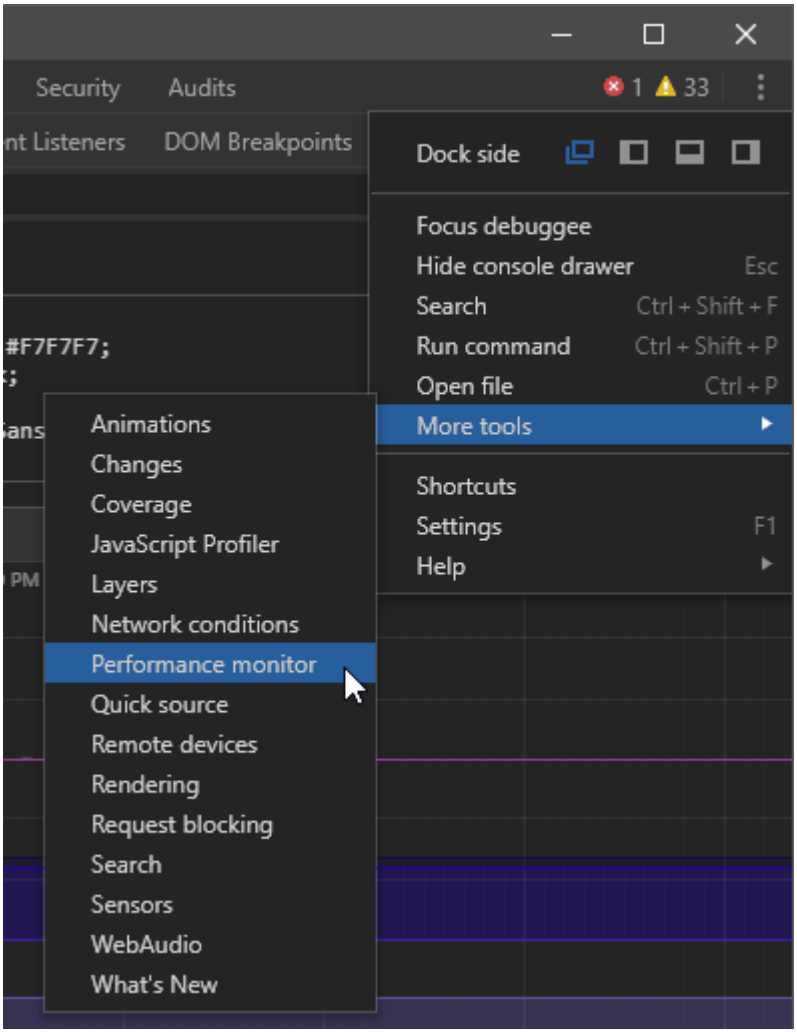
- `is:running` : show any incomplete or unresponsive requests
- `larger-than:S` : limit to files larger than `S` , which can be expressed as bytes (`10000`) Kilobytes (`1000k`), or megabytes (`1M`)
- `-larger-than:S` : limit to files smaller than `S`
- `-domain:*.yourdomain.com` : show third-party requests that aren't from your primary domain

Assets from Other Domains

Most sites request assets from other domains, such as CDNs (content delivery networks), fonts repositories, analytics trackers, advertising networks, social media share buttons, and so on. While useful, those resources can have a negative impact on performance, privacy, and security. Refer to Chapter 4 for further information.

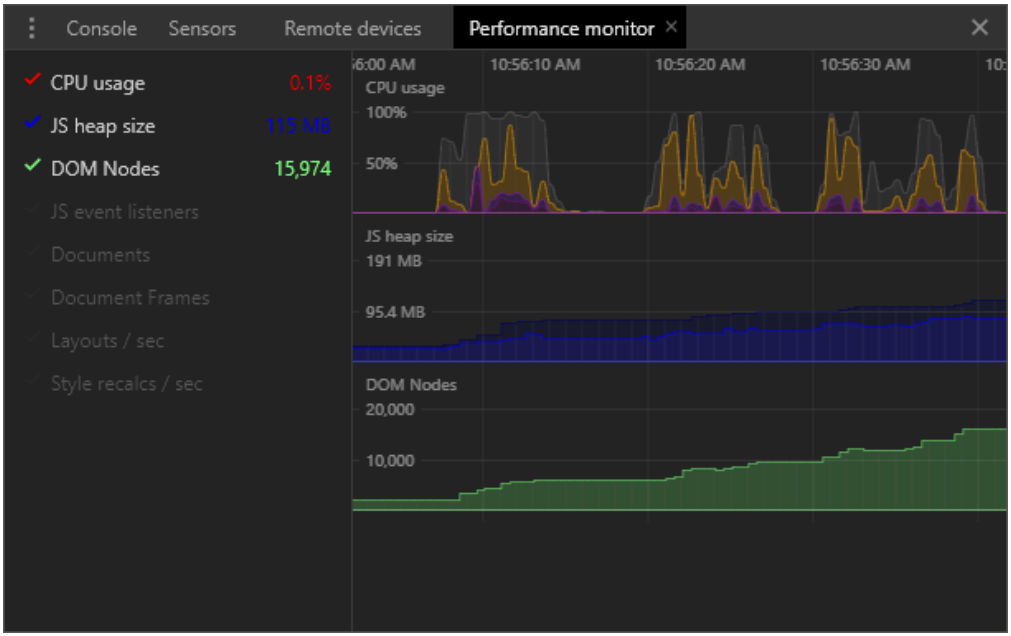
Chrome's Performance Monitor

Chrome's new **Performance Monitor** can be accessed from the DevTools' **More tools** sub-menu (although this may vary across Chrome versions).



2-4. Starting the Performance Monitor

It appears in the lower Console drawer panel, and charts are updated in real time as you use a page.



2-5. The Chrome DevTools' Performance Monitor

Monitors can be displayed and hidden by clicking the heading in the left:

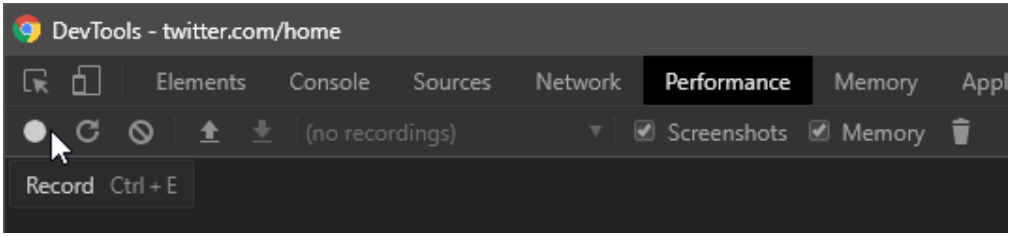
Performance Monitor	Description
CPU usage	processor utilization from 0% to 100%
JS heap size	memory required for JavaScript objects
DOM Nodes	the number of elements in the HTML document
JS event listeners	the number of registered JavaScript event listeners
Documents	the number of document resources including the page, CSS, JS, etc.
Document Frames	the number of frames, iframes, and worker scripts
Layouts / sec	the rate at which the browser has to re-layout the DOM
Style recalcs / sec	the rate at which the browser has to recalculate styles

The Performance Monitor could be used to discover unusual spikes in activity—such as rising memory use or layout recalculations when an element has been added to the page. Further investigation can then be carried out in the **Performance Panel**.

Developer Tools' Performance Panel

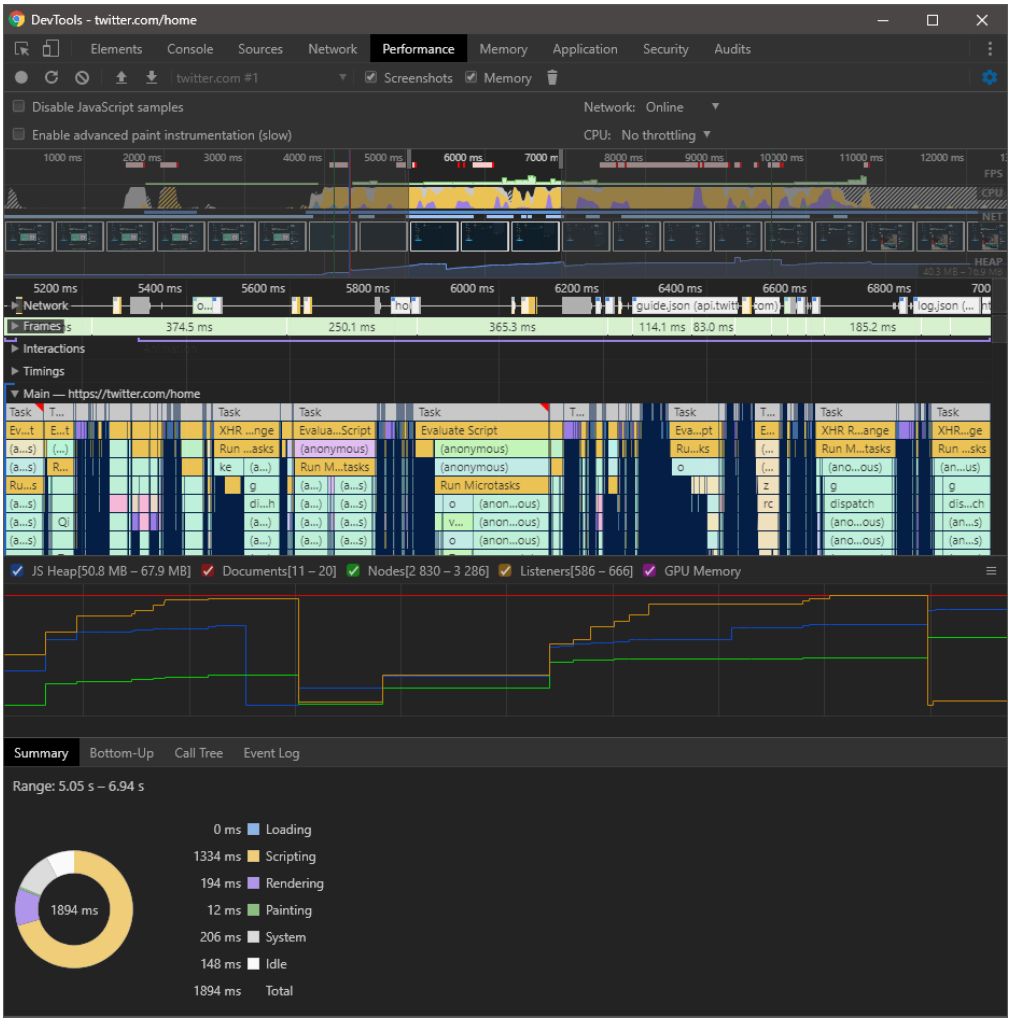
The developer tools provided in Chrome, Firefox, Safari, and Edge provide a **Performance** tab that allows you to record a snapshot of browser activity when particular actions are made. Unlike the Performance Monitor, you must record a profile before it can be analyzed.

To use Chrome's version, open DevTools and navigate to the **Performance** pane. The **Settings** cog allows you to select network and CPU throttling options before clicking the **Record** icon.



2-6. DevTools' record performance button

Load or use your site as required, then hit **Stop** to generate the performance report.



2-7. A DevTools Performance report

The report can be daunting, but it can be saved and reloaded later for further analysis.

The top timeline chart shows frames per second, CPU usage, network usage, screenshots, and the heap memory size. An area can be selected with the mouse to focus on a specific point.

Result panes can be expanded and collapsed. Versions of Chrome differ, but

panes may include:

- **Network**: loading times for individual files
- **Frames**: screenshots at points on the timeline
- **Interactions**: input and animation timings
- **Timings**: events such as DOMContentLoaded and the First Meaningful Paint
- **Main**: thread activities such as function calls and event handlers

These are followed by a chart showing the JavaScript memory heap, number of documents, number of nodes, event listeners, and GPU memory usage.

The final **Summary** panel changes as you click items in the upper panes. The breakdown may include function call and event details as well as timings where appropriate:

- **Loading**: time to load assets from the network
- **Scripting**: JavaScript execution resulting in visual changes (which can also include CSS animations and transitions)
- **Rendering**: the browser process of calculating which CSS rules apply and how layout is affected
- **Painting**: the browser process of filling in pixels and drawing layers in the correct order
- **System**: other browser activities
- **Idle**: no activity

The panel can typically be used to discover inefficient activities, including:

- expensive event handlers, such as those attached to scroll or mouseover actions
- long-running JavaScript functions
- slow or badly throttled network requests
- a continually rising JavaScript Heap, which could result from memory leaks or poor garbage collection
- style changes that affect many DOM elements
- animations that incur frequent layout changes

Chapters 4 and 5 provide common solutions to these issues.

DevTools' Console Logs

Performance monitoring can help discover problems when specific actions are performed in a site or app. However, it may become necessary to profile JavaScript execution by logging messages to the console when events occur. Modern browsers support various Performance Timing APIs that can help to analyze code.

performance.now()

`performance.now()` returns the elapsed time in milliseconds since the page was loaded. Unlike `Date.now()`, it returns a floating-point number representing fractions of a millisecond:

```
let t0 = performance.now();
doSomething();
let t1 = performance.now();
console.log(`doSomething() executed in ${ t1 - t0 }ms`);
```

Performance Marks and Measures

`performance.now()` can become arduous to manage as an application grows. The API also allows you to **mark** when an event occurs and **measure** the time elapsed between two marks. A mark is defined by passing a name string to `performance.mark()`:

```
performance.mark('script:start');

performance.mark('doSomething1:start');
doSomething1();
performance.mark('doSomething1:end');

performance.mark('doSomething2:start');
doSomething2();
```

```
performance.mark('doSomething2:end');

performance.mark('script:end');
```

Each `mark()` call adds a `PerformanceMark` object to an array, which defines the `name` and `startTime`.

Navigation and Resource Entries

The array is likely to contain other automatically generated browser entries for navigation and resource timings.

A mark can be cleared with `performance.clearMarks(markName)`. All marks are cleared when no name is passed.

The elapsed time between two marks can be calculated by creating a `performance.measure()` by passing the measure name, start mark, and end mark:

```
performance.measure('doSomething1', 'doSomething1:start', 'doSomething1:end');
performance.measure('script', 'script:start', 'doSomething1:end');
```

Omitting the start mark measures from the moment the page loaded. Omitting the end mark measures to the current time. Each `measure()` call adds a `PerformanceEntry` object to the same array, which defines the `name`, `startTime`, and `duration`.

A measure can be cleared with `performance.clearMeasures(measureName)`. All measures are cleared when no name is passed.

Marks and measures can be accessed with:

1. `performance.getEntriesByType(type)`, where `type` is either `"mark"` or `"measure"`
2. `performance.getEntriesByName(name)`, where `name` is a mark or measure

name

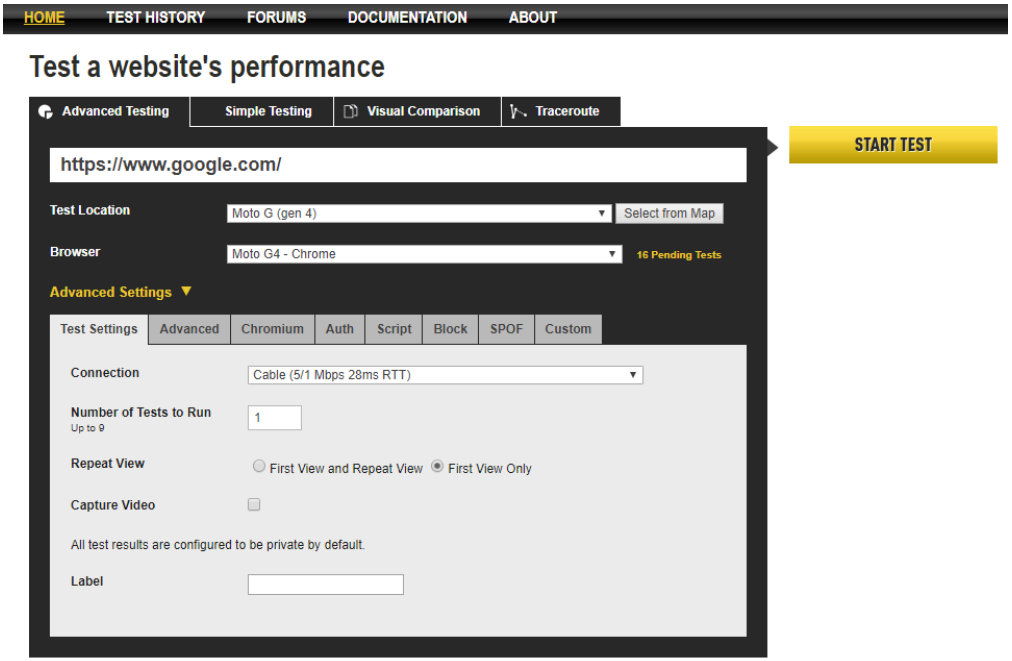
3 *performance.getEntries()*, to access all items in the array.

The name and duration of every measure can therefore be output:

```
performance.getEntriesByType('measure')
  .forEach(m => console.log(`${m.name}: ${m.duration}ms`));
```

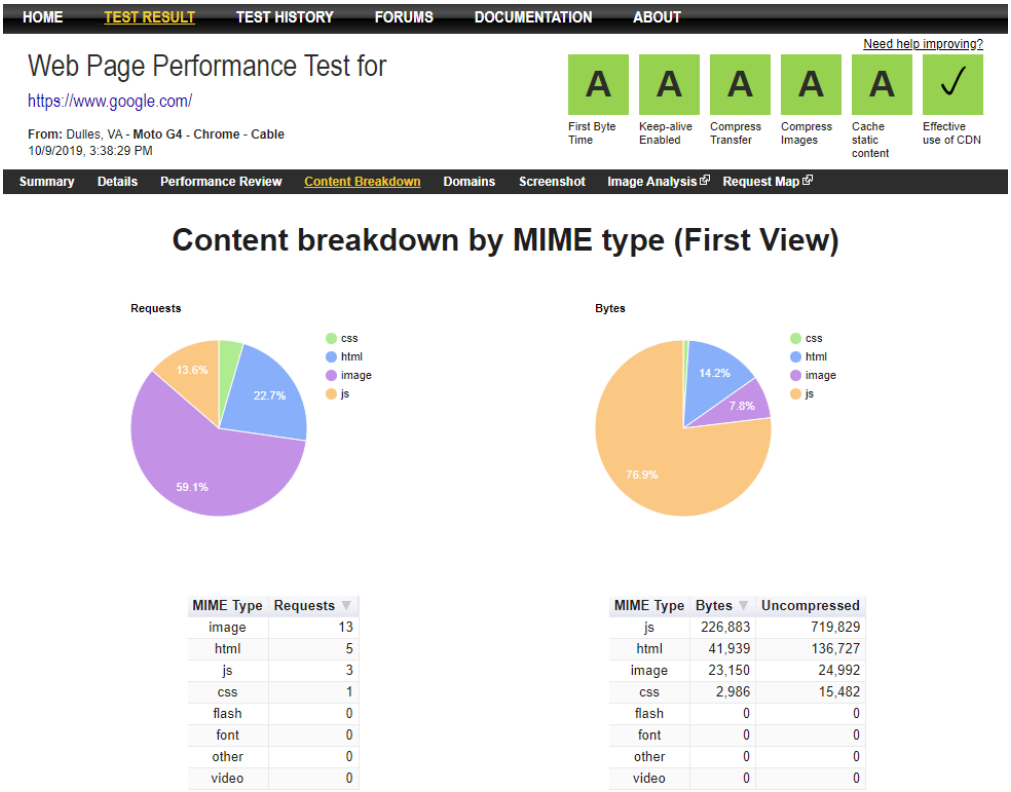
WebPageTest.org

Despite its retro look, WebPageTest.org reports performance information from global locations using emulated devices with a range of settings.



2-8. WebPageTest.org

DevTool-like reports take a few minutes to generate.



2-9. WebPageTest.org report

An A (good) to F (bad) report is shown at the top:

- **First byte time**: the time taken for the first byte to be received
- **Keep-alive enabled**: are persistent HTTP connections used?
- **Compress transfer**: are assets gzip compressed?
- **Compress images**: can images be compressed further?
- **Cache static content**: does the site leverage browser caching?
- **Effective use of a CDN**: are content delivery networks used?

Content Delivery Networks

The benefits of using a CDN are described in Chapter 3.

The **Summary** and **Details** panels provide a table and waterfall chart showing network metrics, where lower figures indicate better performance:

Performance Metric	Description
Load Time	the time taken from the initial request to the browser load event
First Byte	the time taken for the first byte to be received
Start Render	the time taken for the browser to start rendering content on the page
Speed Index	the average time at which visible parts of the page are displayed (more information)
DOM Elements	the number of DOM elements
Document Complete	a set of metrics relating to the DOMContentLoaded event, when the HTML has fully loaded but other assets such as images and fonts may still be in progress
Fully Loaded	a set of metrics relating to the window load event when all page assets have been downloaded and rendered

The **Performance Review** panel shows information about effective use of file compression, browser caching, and CDN usage.

The **Content Breakdown** panel provides a list of assets by type (HTML, CSS, JavaScript, images, and others) with requests, size, network response times, and rendering events.

The **Domains** panel shows the number and size of requests made to the page and third-party domains such as CDNs, font repositories, analytics, advertisers, social media widgets, and so on.

The **Processing Breakdown** panel shows the time taken for browser processes

including loading, scripting, layout, and painting.

The **Screenshot** panel shows frames and/or a video at specific points, such as the time the first content appeared, when a hero image was shown, and the fully loaded view.

Other panels provide links to additional services such as image optimizers and domain request maps.

For more information, refer to:

- WebPageTest documentation
- Lean Websites, Chapter 3

WebPageTest API

WebPageTest offers a REST API to programmatically automate tests or obtain results during a build process.

Tests are limited to 200 page loads per day. Repeated views to evaluate cached loading factors also count as a page load. Developers must apply for an API key to get started.

Tests are initiated by requesting a URL. In this example, `mysite.com` tests are run twice:

```
http://www.webpagetest.org/runtest.php?k=my-api-key&url=mysite.com&runs=2&f=json
```

A successful request returns a JSON response with a test ID and a set of result URLs:

```
{
  "data": {
    "testId": "1234567890",
    "ownerKey": "ceb6128dbf05e09a1969ad",
```

```
    "jsonUrl": "https://www.webpagetest.org/jsonResult.php?test=1234567890",
    "xmlUrl": "https://www.webpagetest.org/xmlResult/1234567890/",
    "userUrl": "https://www.webpagetest.org/result/1234567890/",
    "summaryCSV": "https://www.webpagetest.org/result/1234567890/page_data.csv",
    "detailCSV": "https://www.webpagetest.org/result/1234567890/requests.csv"
  }
}
```

The status of a test can be checked by passing the `testId` to the `testStatus.php` URL. An HTTP 200 is returned when the result URLs are ready:

```
http://www.webpagetest.org/testStatus.php?k=my-api-key&f=json&test=1234567890
```

The completed results are stored for up to 30 days and can be accessed from the URLs shown in the initial `runtest.php` request.

A Node.js webpagetest module is available to help with API processing.

More Performance Assessment Tools

A variety of other free tools, browser extensions, and commercial services (often with free tiers or trials) are available, which can provide performance monitoring, usage tracking, and improvement advice:

- sitespeed.io
- webhint
- Pingdom Website Speed Test
- GMetrix
- Uptrends
- Sentry.io

Finally, don't forget to test your system using a combination of real devices, operating systems, input types, and browsers on a range of network speeds. Consult your analytics reports to help determine what systems and connections your visitors typically use.

Chapter

3

Quick Snacks

This chapter lists a selection of tasty yet simple, practical, and cost-effective performance solutions that can be implemented on any site within a few minutes. It's best to read the following sections in order, since they start with simpler hosting and server settings before moving on to quick image, CSS, and JavaScript improvements.

Consider Your Hosting Plan

Hosting will affect the performance of your website or application. The range of choices and prices may be bewildering, but hosting services are segregated into four primary types.

Shared Hosting

Your website is hosted on a physical server alongside hundreds, if not thousands, of other sites. Each customer shares resources, so disk space, RAM, CPU time, and other facilities may be limited to ensure the server remains responsive.

Pros:

- They're inexpensive, with services starting from a few dollars per month.
- They offer fully managed backups, security, maintenance, and upgrades.
- They may provide simple, one-click installations for CMS, forum, wiki and other applications.
- They sometimes offer specialist expertise, such as WordPress management.
- Technical support is often included in the price.

Cons:

- Servers may be over-sold, in which case performance and page load times can suffer.
- A problem with another site can affect yours—such as high traffic, denial of service attacks, and so on.
- Hardware failures can take your site offline for a considerable period.
- It can be difficult to scale anything other than disk space or bandwidth.
- Support expertise and response times will vary from host to host.

Dedicated Server Hosting

Your website is hosted on a physical server (or servers) which you own. The hardware is exclusively used by you, so it can be configured to your exact requirements.

Pros:

- You get fast performance, with little possibility of other sites affecting speed.
- You have access to further resources such as RAM, disks, CPUs, and other servers can be added.

Cons:

- Unless the server is fully managed, skilled technical staff will be required to maintain server updates, backups, security, and so on.
- Hardware failures can still occur.
- They're expensive—typically a few hundred dollars per month.

Virtual Private Server (VPS) Hosting

A VPS is your own remote virtual machine. It *feels* like a dedicated server, but is effectively a software emulation running on one or more real servers.

Pros:

- A VPS is quicker to set up and more affordable than a dedicated server.
- It's fast, flexible, and easier to scale.
- It's more robust: your server is just data that can be moved to or run from other hardware.

Cons:

- Resources are still shared and the host's network can be swamped.
- A VPS still requires skilled technical staff to maintain server updates, backups, security, and so on (unless you pay extra for a "managed" VPS).

Cloud Hosting

Cloud hosting comes in many different guises, but it normally abstracts the hardware infrastructure into a set of services that can be accessed on demand. For example, your web application may require server-side processing, a database, and file store services, which are provisioned separately. You could opt to use serverless functions, which implement an application as a series of small micro-services rather than as a single monolithic program.

Pros:

- It's robust and reliable.
- It offers scalability and flexibility, as resources can be instantly scaled up when demand increases.
- It's cost efficient, as you only pay for what you use.

Cons:

- Cloud hosting has a steep learning curve, with each host offering a different service, concepts, and terminology.
- Costs can be extremely difficult to determine up front.
- There's vendor lock-in, as it can be more difficult to move away from bespoke services.
- Support can be a costly extra service.

Switch to a More Appropriate Hosting Option

Performance will be affected as your site increases in popularity. Shared hosting plans are especially susceptible to surges in demand, such as links from Reddit, Hacker News, and so on. Your site could be blocked as soon as it reaches processing, storage, or bandwidth thresholds.

Contact your host first, as they may be able to suggest options for switching your site to another service or platform. For example, SiteGround provides standard web hosting accounts where you can manually install a WordPress CMS. However, they also offer fully managed WordPress hosting in the US, UK, Europe,

and Australia. Your site can be transferred for free to take advantage of application-specific performance optimizations such as caching, compression, minification, image optimization, and lazy loading.

However, be aware that you'll always be limited by your host's infrastructure and location. For example, a European company should consider a US-based host if the majority of their clients are based in North America.

CDNs to the Rescue

The next section describes content delivery networks, which can offset the effects of physical server locations.

Scale Resources

Most hosting services provide scaling options to increase processor, memory, disk space, or bandwidth capacity. This is usually simpler with cloud and VPS hosting, which have less reliance on physical hardware. It may even be possible to move resources to alternative geographical locations.

An additional monthly fee for extra resources will normally apply, although dedicated server providers can make a one-off charge. Some cloud services scale and charge according to demand, although there are often fixed-price elements such as disk space.

Switch Hosts

Switching hosts can have a positive impact on performance if your new host has a faster infrastructure based in a location closer to users. Improved hosting is unlikely to solve all your performance problems, but it can be a cost-effective solution for back-end speed issues.

What to Look for in a New Host

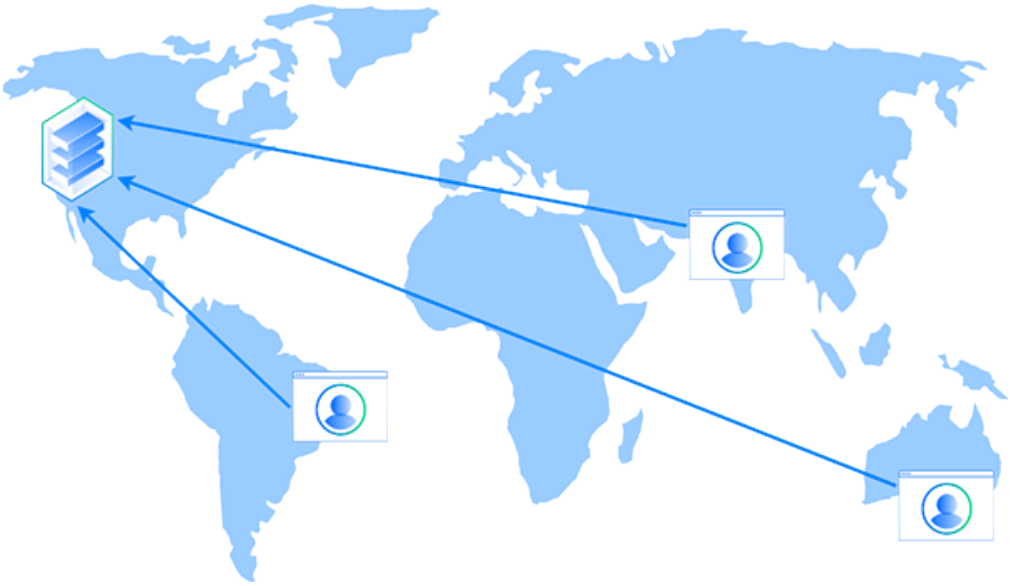
These are some performance-related features to look for in a new host:

- a global infrastructure, or data centers geographically close to your main users
- specialist support for applications you're using, such as WordPress
- HTTP/2 and simple SSL certificate installation
- domain nameserver and DNS configuration
- gzip and Brotli compression enabled
- options for automatic minification, caching, and image optimization
- attack detection and prevention
- usage and speed reports
- automated issue alerts
- good, independent reviews
- a knowledgeable and fast support service

Don't be swayed by worthless 99.9% up-time claims. Reputable hosts are reliable and those that aren't won't provide proof or guarantees. Besides, it still equates to nine hours of downtime per year, which will inevitably coincide with your product launch!

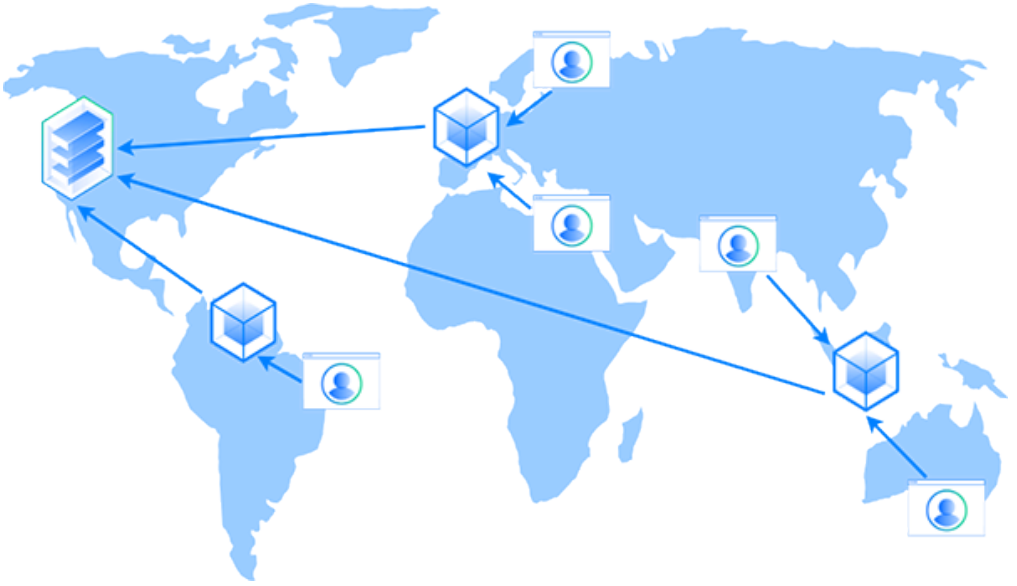
Use a Content Delivery Network

A content delivery network can provide a performance boost to any website by distributing the load and serving assets from locations geographically closer to users. If your site is hosted on servers in California, for example, and you don't have a CDN, all users must connect to the server in California directly:



3-1. No CDN

Using a CDN service allows a user from Australia to access assets from closer servers hosted in Melbourne:



3-2. Active CDN

The response is faster, users are happier, and your hosting requirements may be reduced.

You may already be using a CDN for static assets such as CSS or JavaScript frameworks. For example, jQuery can be referenced at `https://code.jquery.com/jquery-3.4.1.slim.min.js` in a `script` tag. This type of CDN offers several benefits:

- Files are available on fast servers replicated across the globe.
- Files are hosted on a domain other than your website's, which increases the number of concurrent assets the browser can download.
- Files may already be cached in the user's browser, since many sites may reference that URL.

Unfortunately, simple static-file CDNs can be frustrating to manage, and third-party scripts have performance, privacy, and security risks (see Chapter 4).

More recently, CDN services have appeared that automatically proxy requests to your site. This usually requires your domain to point at the CDN's name servers or set specific DNS records. The benefits of this include:

- a faster, high-capacity infrastructure with more efficient delivery from many locations around the globe
- high availability: a CDN can continue to deliver cached files even when a host's server fails
- improved SEO: Google rewards sites with faster response times
- cheaper costs: adopting a CDN is likely to cost less than scaling server resources

CDNs may provide additional services regardless of your host's server facilities and limitations, such as:

- SSL certificates for HTTPS encryption
- load balancing, data compression, and the HTTP/2 protocol for faster transmission
- automatic file minification, image optimization, video transcoding, and email

obfuscation
- attack detection and distributed denial of service (DDoS) prevention
- access blocking to specific IP addresses, countries, etc.
- server-based visitor analytics that don't rely on client-side JavaScript
- custom error pages, redirects, authentication, AMP site generation, serverless APIs, and more

Many services offer free plans or time-limited trials so you can assess performance with the tools mentioned in Chapter 2 before making a commitment. Popular options include:

- Akamai
- Alibaba Cloud CDN
- Azure (Microsoft)
- BelugaCDN
- BunnyCDN
- CacheFly
- CDN.net
- CDN77
- CDNetworks
- Cloud CDN (Google)
- Cloudflare
- CloudFront (Amazon)
- Edgecast (Verizon)
- Fastly
- G-Core Labs CDN
- Hostry
- Imperva
- KeyCDN
- Limelight
- Medianova
- StackPath

Use Image and Video CDNs

Specialist image and video CDNs can be used in addition to or instead of a

standard CDN. Popular options include:

- Cloudimage
- Cloudinary
- ImageEngine
- imgix
- piio
- imagekit.io
- pixboost
- Uploadcare

The main benefits are described in the following sections.

Asset Management

Image CDNs allow you to upload original images—perhaps directly from users—where they can be stored, optimized, and managed via a user interface or API.

Optimal Formatting and Compression

Regardless of the media uploaded, an image CDN can serve the file in the most optimal format. For example, you could upload a JPG image but have it served to Chrome and Firefox users in the more efficient WebP format. Browsers without WebP support would receive the next most appropriate image format.

Video can also be transcoded using a range of alternative codecs so all popular browsers are supported.

Art Direction, Sizing, and Effects

Some image CDNs offer an API that allows you to crop, resize, transform, or apply filters without affecting the original image. For example, Cloudinary's URL-based API allows an image of a person to be cropped to 400px around any detected face, resized to 200px, and served in the most appropriate format:

```
https://res.cloudinary.com/demo/image/upload/w_400,h_400,g_face,r_max/w_200/
↪f_auto/portrait
```

Activate Server Compression

Assets can be compressed on a web server prior to transmission, then uncompressed on the browser. For text-based files such as HTML, SVG, CSS, and JavaScript, this can often reduce bandwidth by 60% or more. According to W3Techs 2019 reports, compression was not activated on one in five websites.

Compression Won't Fix Bloated Code

Compression reduces network transfer times, but the file must still be uncompressed and parsed when it reaches the browser. Bloated code may arrive sooner, but it won't magically become more efficient!

Most good web hosts enable compression by default, or do the work for you. Gzip compression can be activated on all popular web servers including Apache, NGINX, IIS, and Express.js compression middleware.

Brotli is a more modern compression algorithm that reduces file sizes further. It can be enabled on all popular web servers alongside gzip, which must be provided for IE and older browsers that don't support newer standards.

CDNs and Asset Compression

A CDN can implement asset compression even if it's not enabled on your primary server.

Activate HTTP/2

HTTP/2 improves upon the HTTP transmission protocol originally devised by Sir Tim Berners-Lee when he invented the Web in 1989. HTTP/2 reduces latency by:

- sending data in a binary rather than text format
- compressing HTTP headers
- sending more than one file on the same TCP connection
- implementing Server Push, which can send a file before it has been requested

All popular servers and CDNs support HTTP/2 but fall back to HTTP/1.1 for older browsers. More recently, HTTP/3 has been announced, which will further optimize performance. Browser and server support will increase over the coming years.

Leverage Browser Caching

When a browser downloads an asset from a URL, it stores that file locally so it can be referenced and used again. Infrequently-changing files such as images, CSS, and JavaScript are therefore downloaded once and used across multiple pages on a site. Without caching, the Web would be considerably slower and more unreliable.

The server should set appropriate Expires headers, Last-Modified dates, and/or adopt ETag hashes in the HTTP header. Most servers should have reasonable defaults, but you can set custom options. For example, in an Apache `.htaccess` file you can do this:

```
<IfModule mod_expires.c>

ExpiresActive On

# Expire images after one year
<FilesMatch "\.(jpg|jpeg|png|gif|svg|ico)$">
ExpiresDefault "access plus 1 year"
</FilesMatch>

# Expire CSS and JavaScript after one month
ExpiresByType text/css "access plus 1 month"
ExpiresByType text/javascript "access plus 1 month"

# default expiry to one week
ExpiresDefault "access plus 1 week"
```

```
</IfModule>
```

Enable CMS Page Caching

By default, content management systems such as WordPress construct and return a page on every user visit:

1. The URL is examined.
2. The appropriate content is extracted from the database.
3. The content is inserted into template code and returned to the user.

This process is repeated every time—*even when the same page content is seen by all site visitors.*

Fortunately, caching plugins are available, which store the generated HTML after the first visit so all subsequent visitor requests receive the same page. The cached page is then invalidated when content is changed or after a specific time has elapsed. CMS caching can have a dramatic effect on performance and site reliability.

Popular caching plugins for WordPress include:

- LiteSpeed Cache
- W3 Total Cache
- WP Fastest Cache
- WP-Optimize
- WP Super Cache

CDN vs Plugin Caching

A CDN can also cache the HTML page, but most CMS plugins perform other optimizations such as cleaning databases, minifying code, adding HTTP expiry headers, and so on.

Are Videos Necessary?

Videos can offer an engaging experience. Few do, yet they have a higher bandwidth and performance cost than any other web asset. Here are some recommendations:

1. Do you really need to show that tedious CEO presentation to every visitor? Remove all media assets where possible.
2. Ensure the video is as short as possible, removing scenes where practical.
3. Transcode the video into multiple formats using the minimum dimensions with optimal compression. Many video CDNs and services will handle this for you.
4. Only play the video on demand—not as the user accesses the page (see Chapter 4).

Check Your Primary Images

While images don't have the same processing and rendering overheads as HTML, CSS and JavaScript, they usually account for a large proportion of page weight and perceived performance.

Examine your regularly used images, such as those appearing in headers, footers, home page hero blocks. The following tips can dramatically reduce file sizes, although using an image CDN can do some of the hard work for you.

Resize Large Bitmaps

An entry-level smartphone or digital camera takes multi-megapixel images that

can't be displayed in full on the largest screens. Few sites require images of more than 1,600 pixels in width or height.

Resizing has a dramatic effect on image files, since halving the dimensions reduces the size by 75%. You may also be able to crop areas that aren't normally shown or contain large blocks of single colors.

Choose an Appropriate Image Format

Choosing the correct format will radically reduce image file sizes. In general:

1. The JPG/JPEG format is best for photographs with intricate details.
2. The PNG format is best for logos, diagrams, and charts with solid blocks of color. The 8-bit 256-color format will normally result in smaller files if you don't require 24-bit true-color or alpha transparency.

You should also consider:

1. **SVG**: Scalable Vector Graphics define lines, paths, and shapes in XML rather than individual pixels. They're best suited to logos and diagrams, since they can be scaled to any size without loss of quality.
2. **GIF**: these can be animated and sometimes result in smaller files than similar 8-bit PNGs.
3. **WebP**: this format can compress any type of image, but is not currently supported in IE, Safari, and older browsers.

New image formats such as HEIC and AVIF may become viable in future.

Avoid Base64 Encoding

Images can be encoded into a base64 string within a data URI defined in an HTML `<img>` tag or CSS `background` property:

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

This reduces the number of HTTP requests, but it rarely boosts performance:

1. Base64 encoding is typically 30% larger than the binary equivalent.
2. The browser must parse the string before an image can be used.
3. Altering an image invalidates the whole (cached) HTML or CSS file.

Only consider base64 encoding if:

- it's a practical option for your application—such as when images are generated
- the encoded string is very small—perhaps not much longer than a URL

There may also be a case for reusable SVG icons defined as CSS `background-image` properties. For example:

```
.mysvgbackground {
  background-image: url('data:image/svg+xml;utf8,<svg xmlns="http://www.w3.org/2000/
  ↪svg" viewBox="0 0 800 600"><circle cx="400" cy="300" r="50" stroke-width="5"
  ↪stroke="#f00" fill="#ff0" /></svg>');
}
```

Compress Images Effectively

Image tools can reduce file sizes by stripping metadata, simplifying details, and increasing compression factors. Ideally, image compression will be handled automatically in a build process using options such as imagemin and svgo (see Chapter 5), but several online tools are available for one-off tasks:

- Compressor.io: online, all image types
- jpeg.io: online, any format to JPG
- RIOT: Windows application, bitmaps

- ShrinkMe: online, all
- Squoosh: online, all
- SVGOMG: online, SVGs
- TinyPNG / TinyJPG: online, bitmaps

Concatenate and Minify CSS

Multiple style sheets can be loading using HTML `<link>` elements and CSS `@import` at-rules.

Loading separate files is usually inefficient, because each `@import` blocks the browser's rendering process; the imported file could have further nested `@import` rules. Performance can therefore be improved using:

1. **Concatenation**: all partials are combined into a single large file in the necessary source order.
2. **Minification**: unnecessary comments, whitespace, and characters are removed to minimize the file size.

A build process (see Chapter 5) or pre-processor can automate CSS concatenation and minification, but online tools are also available:

- CSS Minifier
- minifier.org
- CSS Compressor
- CSS Minify
- Online Compressor

HTTP/2 and Multiple Files

HTTP/2 lessens the need for file concatenation because it reduces the overhead of transmitting multiple files:

- pipelining allows the server to send responses in any order
- multiplexing permits any number of request and response messages on the same TCP connection at the same time
- the server can use Server Push to send assets before they're requested

In theory, separate files may be a benefit on large, regularly updated applications, because just the modified assets can be sent. However, testing is recommended. There are unlikely to be many downsides of concatenation and minification.

Concatenate and Minify JavaScript

Application code is normally split into multiple files with related or self-contained functionality. This makes development more practical: files are easier to understand, each can be tested individually, and reuse in other projects is easier. However, dependencies must be declared in some way to ensure script A is loaded before it's referenced in script B.

Multiple JavaScript files can be loaded in a single web page using:

1. more than one HTML `<script>` element defined in dependency order
2. ES6 modules, which `import` dependencies when they're required within a script
3. older run-time module loaders such as RequireJS, which provide dependency management in ES5 and below

Like CSS, JavaScript benefits from concatenation and minification: dependencies can be determined at build time, a single HTTP request is required, and the download file is smaller. Some minification tools can also optimize code for improved performance.

A build process using modules such as Babel, rollup.js, or preprocess can manage dependencies and create a single JavaScript file that's minified with terser (see Chapter 5). Alternatively, the process can be handled manually to improve performance on an existing site:

- JSCompress
- minifier.org
- Minify your JavaScript
- Online Compressor
- Packer

Pre-minified Third-party Code

Third-party JavaScript frameworks and libraries often provide pre-minified versions of the source code. Consult the documentation or look for file names containing `min`, such as `jquery-3.4.1.min.js`.

Minify HTML

HTML can also be minified to remove comments, white space, and even unnecessary quotes around attributes. HTML code is often smaller than CSS and JavaScript, so performance gains will be less noticeable, but minification can be simple with a CMS plugin, framework module, or build system (refer to Chapter 5).

Load JavaScript at the End of the Page

When the browser encounters a `<script>` tag in the HTML, it halts all other operations while it downloads and parses the code. This is known as a **render-blocking** process.

It's normally more effective to place `<script>` tags at the bottom of the page before the closing `<\body>` tag. This improves page performance, since the content is viewable before an attempt is made to process JavaScript.

Two attributes can be added to a `<script>` tag to ensure JavaScript is loaded in the background without blocking the render process:

1. `defer` : the script is executed when the DOM is ready and shortly before the DOMContentLoaded event. All deferred scripts are run in the order they're referenced on the page.
2. `async` : the script is executed once it has downloaded. This could occur at any point during or after the page has loaded, so it can't have other script dependencies.

Both attributes are well supported across modern browsers, but they're not suited to all scripts. For example, deferred scripts run when the DOM is ready, but this can occur before the CSS Object Model has been parsed. A script that analyses applied CSS colors can therefore fail randomly. Scripts placed at the bottom of the page are never affected by this issue, and preloading may help (discussed next).

Loading CSS

CSS is also render-blocking. However, if it were loaded at the end of the page, the browser would show unstyled HTML as the page loaded, then re-layout the content after the CSS had been parsed. This looks somewhat ugly and has a negative effect on performance. An alternative option is "critical CSS", as described in Chapter 4.

Preload Assets

The HTML `<link>` tag has a `preload` attribute. This specifies resources the page requires so downloading can start immediately rather than waiting for its reference in the HTML.

For example, a page with a `<script>` tag just before the closing `</body>` can be preloaded in the HTML `<head>` :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My page</title>

  <!-- preload script -->
  <link rel="preload" href="script.js" as="script" />

  <link rel="stylesheet" href="styles.css" />
</head>
<body>

  <h1>My page</h1>
  <p>Lots of content...</p>

  <!-- load script (may be ready) -->
  <script src="script.js"></script>

</body>
</html>
```

The larger the HTML page, the greater the preloading benefit.

The optional `as` attribute allows the browser to prioritize and cache assets more effectively. That is, a script is downloaded before a video because it's more critical to the page's operation. Permitted values are:

- `audio` : an audio file used in an `<audio>` element
- `document` : an HTML document embedded in a `<frame>` or `<iframe>`
- `embed` : a resource embedded inside an `<embed>` element
- `fetch` : a URL required by an Ajax fetch or XHR request
- `font` : a font file
- `image` : an image file
- `object` : a resource embedded in an `<object>` element
- `script` : a JavaScript file
- `style` : a CSS style sheet
- `track` : a video subtitle WebVTT file
- `video` : a video file used in a `<video>` element

- `worker`: a JavaScript web worker or shared worker

A further optional `type` attribute defines the resource's MIME type, so the browser can make further optimizations or avoid downloading unsupported assets. For example:

```
<link rel="preload" href="video.mp4" as="video" type="video/mp4" />
```

Other Attributes and an API

The `prefetch` attribute is similar to `preload`, except that it's intended to fetch resources that will be in the next navigation/page load. Browsers give `prefetch` a lower priority.

Similarly, `prerender` can be used to render a specified web page in the background. Since this potentially wastes bandwidth, browsers often limit processing and memory use.

The `dns-prefetch` attribute can be used to resolve a domain name to an IP address before resources are requested, while `preconnect` establishes a connection to a server.

Finally, look out for Portals, which renders a page in the background and allows the user to instantly navigate to it. The technology is new and yet to be fully implemented in any browser.

Remove Unused Assets

Features will be added and dropped as your website or application evolves. Unfortunately, it's easy to leave stray, unused resources lingering in the codebase, which negatively affects performance. Easier assets to remove include:

- CMS plugins. Disable or delete CMS (WordPress) plugins you're no longer using.

- Unused fonts, weights, and styles. Try removing suspicious fonts and retest the site. This can improve page weight by several hundred kilobytes, and critical fonts will normally fall back to reasonable alternatives.
- Unused CSS and JavaScript. Check that any removed HTML components have their associated styles and functionality deleted.
- Duplicate dependencies. Make sure you aren't including similar assets more than once. This can be an issue in a CMS where different plugins use slightly different versions of Bootstrap or jQuery.

Code for Specific Pages

Consider removing code from pages that don't use a particular feature. For example, if a page doesn't require a carousel, its styles and functionality could be omitted.

This is less beneficial when concatenated CSS and JavaScript files have been downloaded and cached in the browser. These would contain unnecessary code for that page, but providing a smaller alternative would incur a further download. Analyze your code to determine what could offer the best performance:

1. A single, concatenated file served to every page that can be cached on first use.
2. Multiple source files served over HTTP/2.
3. A compromise solution, such as core styles loaded on every page, plus article styles used on news pages, pagination styles used on search results, and so on.

Assess Analytics Performance

Site owners should be given an easy way to measure page views, journey flows, and feature usage. Tracking via server access logs is possible, but richer statistics are normally available using client-side systems such as Google Analytics. These systems may be free to use, but can have a negative impact on page performance. Try temporarily removing all analytics code from your site to assess

the speed gains.

Fortunately, it is possible to retain your statistics and improve performance:

1. Use a single analytics provider. Using more than one will adversely affect speed and give mismatching reports. Traffic analysis is based on a stack of assumptions; it's often impossible to compare results.

2. Test analytics systems to determine which offers the best performance for the information provided. Google alternatives include Matomo, Clicky, Heap, FoxMetrics, and Woopra. Some can be hosted locally to improve performance further.

3. Consider alternative code. For example, minimalanalytics.com removes lesser-used features to provide a 1.5KB, Analytics-compatible script compared to Google's 73KB original (although it will be more stable).

4. Load analytics scripts after all other JavaScript functionality has completed. Rather than placing code in the `<head>`, the analytics scripts could be the last in the page or loaded after a timeout:

```
<script>
// load Google analytics after one second
setTimeout(() => {

  let
    uaId = 'UA-12345678-9', 'UA-12345678-9'// Analytics ID
    script = document.createElement('script');

  script.src = 'https://www.googletagmanager.com/gtag/js?id=' + uaId;
  script.async = 1;
  script.onload = function() {

    // initialize AnalyticsaLayer = window.dataLayer || [];
    window.gtag = function() { dataLayer.push(arguments); }
    gtag('js', new Date());
    gtag('config', uaId);

  };
```

```
  document.head.appendChild(script);

}, 1000);
'js'</script>
```

Something More Substantial?

Tasty snacks may satisfy hunger for a while, but you'll soon be ravenous again! The next chapter provides more substantial, performance-improving recipes.

Chapter

4

Simple Recipes

The tips provided in this chapter will require a little more effort, but the performance results on new and existing sites may be more dramatic. Some of the simplest but most effective database optimizations are tackled first before delving into images, media, fonts, CSS animations, and some controversial topics.

Optimize Your Database

Database access is often the biggest processing bottleneck on the server. Optimizing front-end performance may be futile if your database is struggling to cope with user demand.

There are a vast array of database types, but common performance solutions are described in the following sections.

Use a Query Analyzer

Most databases provide tools that describe how a query has been processed. These can identify missing indexes or other performance issues. Many SQL and NoSQL databases offer an `EXPLAIN` clause or option:

- MySQL EXPLAIN
- PostgreSQL EXPLAIN
- SQLite EXPLAIN
- SQL Server EXPLAIN
- Oracle EXPLAIN PLAN
- MongoDB .explain()
- Couchbase EXPLAIN

The output can be complex, verbose, and beyond the scope of this book. Consult the documentation and look for tools that can help understand the issues.

The database logs can usually be tuned to record long-running queries, and you may find open-source or commercial products to help optimize data.

Create Indexes

Many database performance issues will be solved with an index. An **index** works identically to those in a book: it allows a database to quickly jump to a record given a list of items defined in a specific order.

Consider a `user` table containing an ID (number), name, email, and hashed password. The ID is likely to be the primary key and the table is ordered by that value. A query for a specific ID is fast, because the database can use an efficient searching algorithm. For example, it can start at the middle record and, if its ID is higher, it knows the record must be in the first half of the table.

However, a login form requesting a user's email address and password must query by that email. Those will be randomly ordered in the `user` table, so the database has to check every record until it locates a match. The larger the table, the slower the query. An index can define a list of emails in alphabetical order (or any order that's practical). The searching algorithm can then use that index to locate a record by email, just as fast as searching by ID.

Indexes should therefore be considered on any field commonly used in search queries (typically `WHERE` or `JOIN` clauses in an SQL `SELECT`). It's tempting to add indexes for every field, but the more you create, the more space is required, and the slower write operations become, as all indexes must be updated.

Simplify Queries

The less work the database has to do, the faster a result will be returned. Examine your codebase for complex or multiple dependent queries, especially those that are generated or contain sub-queries. It will usually be possible to make the search more efficient.

For example, consider a query that retrieves the top five selling books. In MySQL-compatible SQL:

```
SELECT title, author_id FROM book ORDER BY sales DESC LIMIT 5;
```

The results contain an `author_id` reference, so five further queries are made to fetch author names. For example:

```
SELECT firstname, lastname FROM author WHERE author_id = 14;
SELECT firstname, lastname FROM author WHERE author_id = 52;
SELECT firstname, lastname FROM author WHERE author_id = 50;
SELECT firstname, lastname FROM author WHERE author_id = 22;
SELECT firstname, lastname FROM author WHERE author_id = 20;
```

This is known as the **N+1 problem**: a large set of queries must be made for the parent records and each result.

A more efficient option would be to fetch all the authors in a single query:

```
SELECT firstname, lastname FROM author WHERE author_id IN (14,52,50,22,20);
```

Performance can be improved further with a single query that can be optimized by the database:

```
SELECT book.title, author.firstname, author.lastname
FROM book
LEFT JOIN author ON book.author_id = author.id
ORDER BY sales DESC LIMIT 5;
```

Create Additional Database Connections

Many web applications create a single database connection object that's used for all queries and updates. Unfortunately, some databases queue all incoming requests from a single connection and process them in order. If one user runs a complex operation that takes 20 seconds to complete, every other user will have to wait at least 20 seconds for their operation to be processed.

Connection queuing issues will be more evident on continually running applications such those implemented in Node.js. PHP applications are usually served by a web server, which creates separate threads with new connection objects on every request, although pooling solutions may be in place.

To prevent database request queuing problems, consider these options:

1. creating single-use connection objects for queries that could take time
2. creating multiple connection objects for specific uses or which can be used in request order

However, be wary of creating too many in-memory connection objects, which could lead to stability issues.

Consider a Server or Memory Upgrade

Databases work more effectively when they have plenty of RAM. RAM allows the system to optimize frequently used queries and cache results in memory for fast access.

Alternatively, you could consider using either:

1. a separate database server
2. multiple servers that either share processing or shard data into smaller silos
3. a third-party database provider that handles the hard work for you

Cache Results

It may not be necessary to perform queries every time a user requests a resource. Consider a statistical dashboard displaying various charts that are computationally expensive to create. The data could be fetched once from the database, cached in memory or a file, then returned on every subsequent request. The charts would be updated either when:

1. data has changed
2. a specific time has elapsed (such as ten minutes)
3. a combination of factors is satisfied (such as when data has changed and

it's at least five minutes since the last calculation)

Solutions such as Redis and memcached are often used for caching purposes.

Use Background Processing

Consider a web application where a user can upload multiple images. These have metadata extracted, are resized, and have filters applied before data is stored in various tables.

Rather than doing all this work in the web application at the point the request is made, the server could return a result immediately and offload processing to one or more background tasks. The application will feel more responsive, even though the final results may take a short while to appear.

Use Alternative Data Systems

Examine alternative systems such as Elasticsearch, which provides faster, richer, and more appropriate search results than standard, full-text database queries. Background processes could populate Elasticsearch indexes, which are then used for search queries. While this now means you have two database systems to manage and optimize, it could reduce bottlenecks and improve functionality.

Remove or Optimize Social Media Buttons

Social media sharing buttons are regularly added to websites to improve engagement and publicize content on other platforms:

4-1. Facebook like and share buttons

Those innocent buttons have a high cost: *Facebook's share button downloads 786KB of code (216KB gzipped)*. Twitter adds a further 151KB (52KB) and LinkedIn 182KB (55KB). Adding a few buttons considerably increases page weight, and processing a megabyte or two of JavaScript has a detrimental effect on performance—especially on mobile devices. That could be the start of your

problems, for various reasons listed below.

- The code is not sitting idle. Regardless of whether or not someone clicks a button, your visitors are being monitored across your site and others.
- You may be liable for the use—*or misuse*—of personal data. The European Court of Justice ruled in 2019 that sites voluntarily sharing visitor information with a social network are considered joint data controllers.
- Third-party JavaScript is a security risk (see the next section).
- Supporting every social media platform is impossible. You're likely to miss options, and some services don't provide sharing facilities.
- Site engagement can be reduced if your visitors are tempted to stay on the social network.

The risks are high, given just 0.2% of visitors use the buttons. (Sources: GOV.UK and Moovweb.)

If your site owners understand the hazards but still want to keep the buttons, there's a couple of options for retaining sharing without adversely affecting performance, privacy, and security.

Use URL-based Share Links

Any page can be shared on Facebook with a link like this:

```
https://www.facebook.com/sharer/sharer.php?u=${url}
```

Likewise for Twitter:

```
https://twitter.com/intent/tweet?url=${url}&text=${title}
```

And LinkedIn:

```
https://www.linkedin.com/shareArticle?mini=true&url=${url}&title=${title}
```

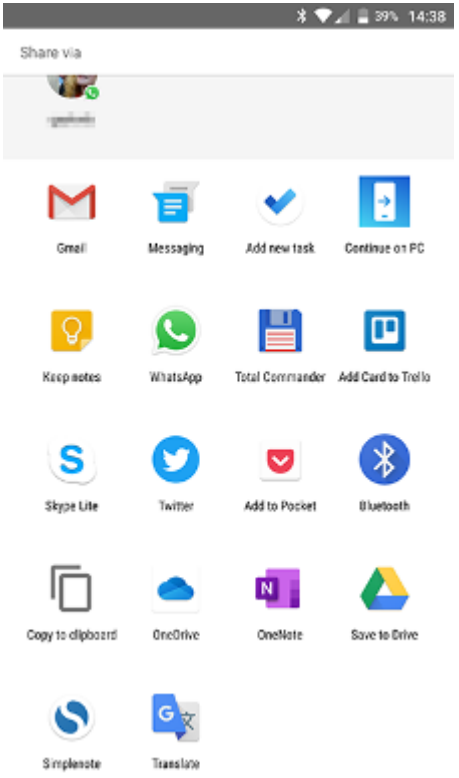
In these examples, `${url}` is the page URL and `${title}` is the title (perhaps

the text contained in the page's `<title>` tag).

Most social networks offer similar URL-based APIs. They're lightweight and only activate when a user chooses to engage with the platform. You can implement these in standard `<a>` tags and, if necessary, intercept the click with JavaScript to open the link in a new window.

Use the Web Share API

Visitors can use their browser's **Share** facility to post URLs to social media apps as well as email, messaging, Pocket, WhatsApp, and more.



4-2. Browser share options

The option is normally provided on mobile browsers, but it may not be obvious to users. Progressive Web Apps (see Chapter 5) can also hide the browser interface.

Fortunately, the Web Share API was introduced in Chrome 76 on Android, Safari 12.3 on iOS, and Safari 12.1 on macOS. The API hands information to the host operating system, which knows which apps support sharing.

The sharing UI can be shown in response to a user click. The following JavaScript checks whether the Web Share API is supported, then adds a button click handler that passes a `ShareData` object to `navigator.share()` :

```
// is the Web Share API supported?
if ( navigator.share ) {

  // share button click handler
  document.getElementById('share').addEventListener('click', () => {

    'share'// share page information
    navigator.share({
      url: 'https://example.com/',
      title: 'My example page',
      text: 'An example page implementing the Web Share API.'
    });

  });

}
```

The `ShareData` object contains:

- `url` : the URL being shared (an empty string denotes the current page)
- `title` : the document title (perhaps the page's HTML `<title>` string)
- `text` : arbitrary body text (perhaps the page's `description` meta tag)

Unlike with share buttons, it's possible to share a page `#target` such as an individual section or comment rather than the primary URL.

`navigator.share()` returns a Promise so `.then()` and `.catch()` blocks can be used if you need to perform other actions or react to failures.

Be Wary of Third-party Scripts

Analytics systems, advertising platforms, social media buttons, and custom widgets often require you to add a third-party `<script>` (from another domain). Those scripts may be huge or grow without you realizing.

Third-party scripts also run with the same site-wide rights and permissions as your own code. As well as hindering performance, they can track users, upload data elsewhere, change your content, redirect to other pages, trigger ecommerce transactions, auto-click advertisements, or perform any other malicious actions.

Your performance, privacy, and security is only as good as the weakest provider. Ensure third-party scripts:

- are delivered over HTTPS to eliminate man-in-the-middle attacks
- use `<script crossorigin="anonymous">` to ensure there's no exchange of user credentials via cookies or other technologies
- set a `<script> integrity` attribute with a file hash to reject any script that's been changed by the provider (refer to Subresource Integrity on MDN)

Ideally, move the script to your domain or remove it entirely.

Third-party Script Used to Target Site

British Airways was fined US$232 million in 2018 when 500,000 customers had their names, email addresses, and full credit card information stolen during website transactions. The attack originated from a third-party script that was modified to target BA, possibly without the knowledge or consent of its supplier.

Use Responsive Images

The `<img>` tag offers optional `srcset` and `size` attributes which are well-supported in most browsers (except IE). These allow specific images to be

requested according to the size of the element and pixel density.

CSS Resolution

Modern smartphones offer screens with very high native resolutions, known as HiDPI or Retina displays. Each pixel is almost invisible to the naked eye, so the browser implements a **CSS resolution** such as 360x760px, where the native resolution could be 1440x3040px. The **display density** is therefore 4x, and a single CSS pixel will be using 4x4 (16) physical pixels.

Given a 100x100px space, it's usually optimal to load a 100x100px image. However, the image quality can look comparatively poor on a 4x display density, so a 400x400px image could be preferable. The `srcset` attribute can define appropriate images in a standard `<img>` tag:

```
<img width="100" height="100"
     alt="responsive image"
     src="img-100.jpg"
     srcset="img-100.jpg 1x,
             img-200.jpg 2x,
             img-300.jpg 3x,
             img-400.jpg 4x" />
```

The browser will select and download the most appropriate image for the display density. This ensures the best image quality without end users having to download unnecessarily large images on all devices.

The image referenced in the `src` attribute is used when the browser doesn't support `srcset`.

image-set() and Media Queries

The CSS `image-set()` function offers similar options for background images, but support is currently limited.

An alternative that works in most browsers is the CSS `resolution` media query, although the code is more verbose.

Alternatively, you can target images based on the rendered width of the `<img>` element:

```
<img alt="responsive image"
     src="small.jpg"
     srcset="small.jpg 400w
             large.jpg 800w" />
```

The `w` unit defines the image file's actual **width** in pixels. Don't use `px` as you would normally expect.

`small.jpg` is used when the viewport is below 400px, but `large.jpg` is used on screens where the CSS or physical pixels exceed 400px.

This example is only practical when the image is the full width of the viewport. The `sizes` attribute defines the size of the image in relation to the viewport so the width can be calculated:

```
<img alt="responsive image"
     src="small.jpg"
     srcset="small.jpg 200w
             large.jpg 400w"
     sizes="50vw" />
```

The image width is `50vw` —half the viewport. `small.jpg` is used when the image width is `200px` or less (the viewport is therefore less than `400px`), but `large.jpg` is used when the image width is greater.

The `sizes` attribute can contain complex media queries and a final fallback size to determine the image width in multiple viewport dimensions. For example:

```
<img alt="responsive image"
     sizes="(max-width: 299px) 100vw,
            (min-width: 300px) and (max-width: 799px) calc(100vw - 60px),
            50vw" />
```

The Bandwidth Cost of Larger Images

A 400x400px image could have a file size 16x greater than its 100x100px equivalent. It requires considerably more bandwidth, which could lead to a poor experience on a mobile network.

Presume smaller images are 20KB and the larger version is 200KB. Each page contains five images and 1,000 page views are made per day. The daily bandwidth saved by using smaller images is 900MB—*or 330GB per year*.

A compromise—perhaps 200x200px—could look reasonable without adversely affecting performance.

Define Responsive Image Aspect Ratios

Since the advent of responsive web design, developers have been advised not to set `width` and `height` attributes on `<img>` tags. The CSS then sets `width: 100%` or `max-width: 100%` to ensure the image is sized to the width of its container or the maximum dimensions of the image accordingly.

The technique has an unfortunate side-effect: *when images start to load, the page must reflow to allocate space*. You'll often experience this on mobile devices, where the text you're reading suddenly moves off-screen because an image suddenly appears further up the page.

An aspect ratio defines the relationship between the height and width, so it becomes possible to calculate the size when only one dimension is known. From

Firefox 71 and Chrome 79, the browser parses `<img>` `width` and `height` attributes to calculate the aspect ratio. The appropriate space can then be reserved so reflows aren't required:

```
<!-- image has a 4:3 aspect ratio -->
<img src="image.jpg" width="400" height="300" alt="my image" />
```

Choosing Height and Width

Any appropriate width or height can be used to set the aspect ratio, since it will be resized using CSS. For example, `width="4" height="3"`. That said, it's best to set a reasonable size to ensure the image is visible in very old browsers, or when CSS fails to load or is disabled.

The following CSS ensures the image uses the full width of its container and sets a height according to the aspect ratio:

```
img {
  width: 100%;
  height: auto; /* this is essential */
}
```

The browser will reserve appropriate space on the page so re-flows become unnecessary. Browsers that don't calculate the aspect ratio won't reserve any space, but there are no downsides. The image will remain responsive.

HTML and CSS Proposals for Defining Aspect Ratios

There are also proposals to define aspect ratios using an HTML `intrinsicsize="400x300"` attribute or a CSS `aspect-ratio: 4/3` property. These would provide alternative options for avoiding reflows, so keep an eye on new browser releases!

Implement Art Direction

The HTML `<picture>` element is similar to `<audio>` and `<video>` in that it will request one of its child elements according to browser support and conditions. For example, it can be used to load a smaller WebP image or fall back to a standard JPG:

```
<picture>
  <source type="image/webp" srcset="image.webp" />
  <img src="image.jpg" alt="JPG image" />
</picture>
```

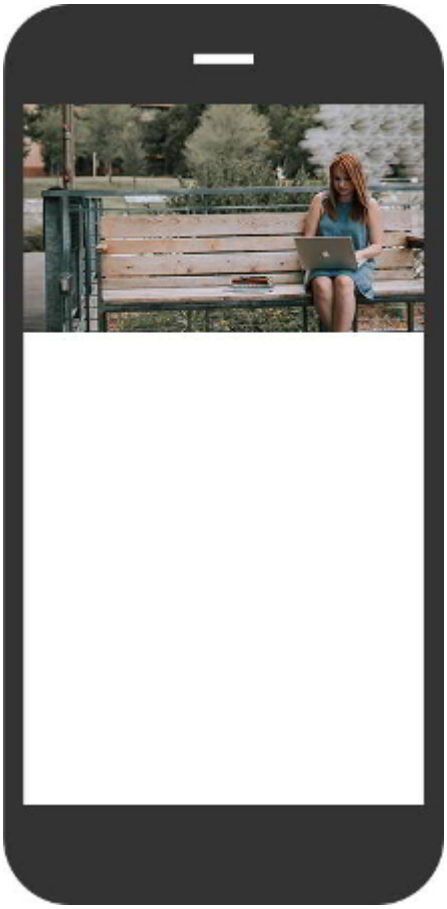
CDN and Server-side Solutions

Some image CDNs and server-side solutions can deliver the most optimum image based on the HTTP request, so just an `<img>` tag would be required.

The `<picture>` element can also be used for art direction. Different images are requested according to the dimensions and orientation of a device. Consider the following hero photograph:

4-3. A landscape image

The landscape image looks reasonable on a typical desktop monitor, but detail is lost on smaller devices held in portrait orientation. It would also become difficult to overlay text in the smaller space.

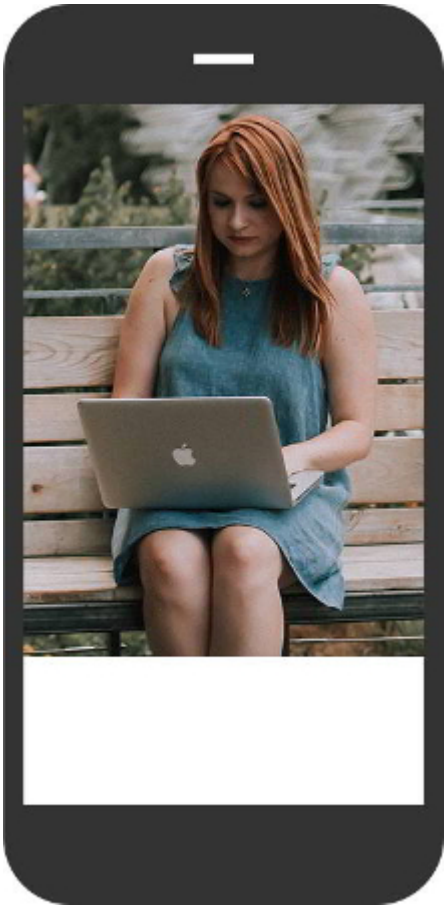


4-4. A smartphone landscape image

Using art direction, we can serve a more appropriate image showing the main subject with less background detail:

4-5. A portrait image

This looks better on a smartphone held in portrait orientation and, in this case, the file size is 65% smaller (59KB compared to 168KB):



4-6. A smartphone portrait image

The `<source>` items in a `<picture>` element can set media queries to determine which image is requested. For example, use `landscape.jpg` when the viewport width is greater than the height, or fall back to `portrait.jpg` otherwise:

```
<picture>
  <source srcset="landscape.jpg"
          media="(min-aspect-ratio:1/1)" />
  <img src="portrait.jpg" alt="portrait image" />
</picture>
```

Any number of `<source>` images can be defined with differing media queries. Each is processed in the specified order until a match is found. A default `<img>`

should always be set as a fallback when no match is available, or for older browsers that don't support `<picture>`.

Lazy Load Images and Iframes

The average web page requests almost 1MB of images. Half of all websites load significantly more! These images (and embedded `<iframe>` elements) download regardless of whether they're viewed or not. A large off-screen image requires bandwidth and processing even when the user clicks a link at the top of the page and never scrolls down.

Load times, bandwidth, and device requirements can be reduced by lazy loading images and iframes when they're scrolled into the viewport. Chrome 76 and above support native lazy loading with the new `loading` attribute:

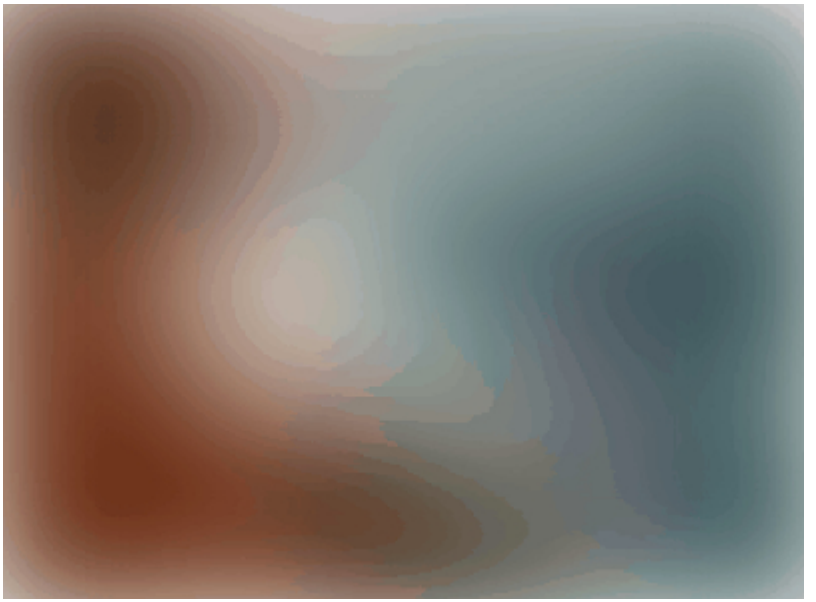
```
<img src="image.png" loading="lazy" alt="lazy load" />
<iframe src="https://site.com/" loading="lazy"></iframe>
```

The following values can be set:

- `auto` : the browser's default behavior (identical to not using the attribute)
- `lazy` : defer loading until the resource reaches a distance from the viewport
- `eager` : load the resource immediately

The *distance from the viewport* can vary according to the type of resource, the network connection, and whether Lite mode/Save-Data is enabled. (Lite mode/Save-Data is covered later in this chapter.)

Native lazy loading is new, so non-Chrome and older browsers require JavaScript-based solutions such as progressive-image.js. These analyze scroll and resize events or use the Intersection Observer API to determine when an element is in view. As well as supporting more browsers, they can also implement attractive loading effects.



4-7. The initial, low-resolution image loaded in the browser

4-8. The full image in view

Play Audio and Video on Demand

Auto-playing media saps bandwidth, degrades performance, and is unlikely to be appreciated by users. Modern browsers will also block or silence auto-playing by default.

In most cases, it's preferable to show a thumbnail image—perhaps with a *play* icon overlay—which the user can click to start the media. Both the `<video>` and `<audio>` elements support this feature with the following attributes:

- `autoplay="false"` to stop auto-playing
- `preload="none"` to prevent media preloading or `preload="metadata"` to fetch meta data such as the video duration
- `poster="image.jpg"` to show a thumbnail image
- `controls="true"` to enable native playback controls

Here's an example:

```
<video controls="true"
       autoplay="false"
        preload="metadata"
         poster="videothumb.jpg">
  <source src="video.mp4" type="video/mp4">
  <source src="video.webm" type="video/webm">
</video>
```

Alternatively, a JavaScript solution could be implemented that replaces a (lazy loaded) `<img>` with appropriate `<video>` or `<audio>` elements when clicked. The solution could also work for third-party video providers such as YouTube and Vimeo, which provide custom video players.

Replace Images with CSS3 Effects

The days of slicing and dicing images in a graphic package to create custom fonts, rounded corners, shadows, linear gradients, and transparency effects have long gone. CSS3 options such as web fonts, border-radius, text-shadow, box-

shadow, color gradients, and opacity are quicker to implement, easier to change, and require far fewer bytes than images.

An element, image, or background image can be manipulated using CSS3 effects rather than having to create multiple variations. For example:

- The clip-path and mask properties can partially or fully hide parts of an image or element to create non-rectangular shapes.
- The shape-outside, shape-margin, and shape-image-threshold properties can be used to define non-rectangular text flows around or within an element.
- The transform property can rotate, scale, and skew an element.
- The filter property offers possibilities such as blurring, brightness, contrast, hue rotation, inversion, saturation, grayscale, sepia, opacity, and shadows.
- Both background-blend-mode and mix-blend-mode control how backgrounds and images blend with each other in a similar way to Photoshop layers. Options include normal, multiply, screen, overlay, darken, lighten, color-dodge, color-burn, hard-light, soft-light, difference, exclusion, hue, saturation, color, and luminosity.

CSS3 Effects Can Be Costly

CSS shadows, gradients, and filters may be costly during browser repaints. Use the effects sparingly and test their impact on scrolling and animation performance.

Use SVGs Effectively

Scalable Vector Graphics define points, lines, and shapes as vectors in XML. Unlike bitmaps, SVG images can be scaled to any dimensions without increasing the file size or losing quality. This makes them ideal for logos, charts, and diagrams.

It's possible to create and manipulate SVGs manually, on the server, or in client-side JavaScript. However, more complex images will require a graphics package such as Adobe Illustrator, Affinity Designer, Inkscape, or SVG edit, followed by an

optimization clean-up in svgo or SVGOMG.

There are three primary ways to add an SVG to a web page. Choose the most appropriate option for each graphic you're using.

1. Add SVGs Using an <img> Tag

The SVG acts like any normal image: it can be cached and reused on other pages.

For security reasons, browsers will disable embedded scripts, links, and other types of interactivity. Some browsers won't apply style sheet rules defined in a separate CSS file.

The lesser-used `<object>`, `<embed>`, and `<iframe>` elements can circumvent these restrictions, but the browser treats the image as another document, so performance could be affected.

2. Add SVGs as CSS Background Images

An SVG can be referenced as a URL in a background image:

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

It can also be embedded inline:

```
.mysvgbackground {
  background-image: url('data:image/svg+xml;utf8,<svg xmlns="http://www.w3.org/2000/
  ↪svg" viewBox="0 0 800 600"><circle cx="400" cy="300" r="50" stroke-width="5"
  ↪stroke="#f00" fill="#ff0" /></svg>');
}
```

Like `<img>`, the browser will block embedded scripts, links, and other SVG interactions, but backgrounds can be useful for regularly used icons.

Inline Data for Larger Images

Inline data should be avoided for larger images, especially when regular changes will invalidate the whole style sheet cached in the browser.

3. Embed SVGs into the Page

An SVG can be embedded directly into the HTML:

```
<body>
  <svg class="mysvg" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 800 600">
    <circle cx="400" cy="300" r="50" />
  <svg>
</body>
```

The SVG nodes become part of the DOM and can be styled or animated directly using CSS:

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

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

This reduces SVG code weight by reusing CSS styles, and it offers additional flexibility such as alternative colors, hover effects, animation of specific elements, and so on.

Unfortunately, the SVG must be embedded into every page where it's required. This will increase HTML weight, so embedding is generally best for small or infrequently used SVGs.

Consider Image Sprites

Often-used images can be packaged into a single **sprite** file so individual items can be accessed in CSS. This is an old optimization technique, but it continues to offer advantages:

1. A single HTTP request is required for many images (although this is less beneficial with HTTP/2).
2. A single image will normally result in a smaller overall file size than the total weight of the individual images.
3. All referenced images appear instantly after the sprite has loaded.

The following image defines five 64x64px icons in a single 320x64px 24-bit PNG:

4-9. A browser icon sprite

Background position offsets are then defined in CSS:

```
.sprite {
  width: 64px;
  padding: 64px 0 10px 0;
  text-align: center;
  background: url("browser-sprite.png") 0 0 no-repeat;
}

.sprite.edge    { background-position: -64px  0; }
.sprite.firefox { background-position: -128px 0; }
.sprite.opera   { background-position: -192px 0; }
.sprite.safari  { background-position: -256px 0; }
```

Individual images can then be referenced in HTML using class names:

```
<div class="sprite chrome">Chrome</div>
<div class="sprite edge">Edge</div>
<div class="sprite firefox">Firefox</div>
<div class="sprite opera">Opera</div>
<div class="sprite safari">Safari</div>
```

The result:

4-10. The rendered sprite icons

Image sprites can be generated in a graphics package, using tools such as SpriteCow or Instant Sprite, or in your build process.

Consider OS Fonts

It's possible to add dozens of fonts to a page … *but that doesn't mean you should!*

1. Designers recommend using fonts sparingly, with one or two typefaces per document.
2. A custom font typically requires a few hundred kilobytes of data. The more you add, the larger the page weight, and the worse the performance.
3. The days of every site using standard OS fonts are over. *Perhaps Helvetica, Times New Roman, or Comic Sans would look good on your site?!*

Using an OS font provides a noticeable performance boost; there's no download delay or flash of unstyled or invisible text.

Each platform supplies different default fonts, but fallbacks can be specified as well as the generic font family names of `serif`, `sans-serif`, `monospace`, `cursive`, `fantasy`, and `system-ui`. For example:

```
body {
  font-family: Arial, Helvetica, sans-serif;
}
```

Web apps may also feel more native if they use a standard system font. The following stack implemented on GitHub.com targets system fonts available across all popular platforms:

```
body {
  font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica,
  Arial, sans-serif, "Apple Color Emoji", "Segoe UI Emoji", "Segoe UI Symbol";
}
```

Similar variations are used by Medium.com and the WordPress administration panels:

```
body {
  font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Oxygen-Sans, U
  buntu, Cantarell, "Helvetica Neue", sans-serif;
}
```

Alternatively, the CSS `@font-face local()` function can be used to locate a font on the user's system first, but load from a URL when it can't be found:

```
@font-face {
  font-family: MyHelvetica;
  src: local("Helvetica Neue"),
       local("HelveticaNeue"),
       url("/fonts/Helvetica-webfont.woff2") format("woff2"),
       url("/fonts/Helvetica-webfont.woff") format("woff");
}
```

An OS font should be your first choice if it closely matches branding requirements.

Embed Web Fonts with <link>

Many designers will be horrified by the suggestion of using OS fonts, so web font use is inevitable. The most popular option is to use a repository that serves fonts from a CDN. Popular options include:

- Google Fonts: fonts.google.com
- Font Library: fontlibrary.org
- Adobe Edge: edgewebfonts.adobe.com

Where possible, load fonts using a `<link>` in your HTML `<head>`. For example:

```
<link href="https://fonts.googleapis.com/css?family=Open+Sans" rel="stylesheet">
```

This downloads the font in parallel with other fonts and style sheets.

A CSS `@import` method may be offered by the repository, but this blocks processing of the style sheet until the font has been downloaded and parsed.

Limit Font Styles and Text

Only request the fonts, weights, and styles you require—*and definitely remove any fonts you aren't using!*

Here's an example of two Google Font URLs:

- https://fonts.googleapis.com/css?family=Inconsolata:500,700
- https://fonts.googleapis.com/css?family=Roboto:bolditalic

Both fonts can be contained in a single URL:

- https://fonts.googleapis.com/css?family=Inconsolata:500,700|Roboto:bolditalic

In some cases, you may only need specific characters—perhaps for a regularly used title or logo. The text "Hello" requires just four characters from a specific font:

- https://fonts.googleapis.com/css?family=Inconsolata&text=Helo

Finally, you could benefit from hosting the fonts locally or using more popular fonts that have a higher chance of being pre-cached in the user's browser.

Use a Good Font-loading Strategy

A web font can take several seconds to download. The browser will choose one of two options:

1. Show a **flash of unstyled text** (FOUT). The first available font fallback is used immediately. It's replaced by the web font once it's loaded. This process is used by IE, Edge 18 and below, and older editions of Firefox and Opera.
2. Show a **flash of invisible text** (FOIT). No text is displayed until the web font has loaded. This process is used in all modern browsers, which typically wait three seconds before reverting to a fallback.

Either option can be jarring and affect perceived performance.

The CSS font-display property allows you to define the font-handling process. The options are:

- `auto`: the browser's default behavior (usually FOIT).
- `block`: effectively FOIT. The text may be invisible for up to three seconds. There's no font swap, but text can't be read immediately.
- `swap`: effectively FOUT. The first fallback is used until the web font is available. Text can be read immediately, but the font swap effect may be jarring if not managed effectively.
- `fallback`: a compromise between FOIT and FOUT. Text is invisible for a short period (typically 100ms) then the first fallback is used until the web font is available. Text is readable as the page loads, but the font swap can still be problematic.
- `optional`: the same as `fallback`, except no font swapping occurs. The web font will only be used if it's available within the initial period. The first page view is likely to show a fallback font while the web font is downloaded and cached. Subsequent page views will use the web font.

Similar Web and OS Fonts

`optional` could be a reasonable choice if the web and OS fallback fonts are similar, but if that's the case, using an OS font throughout would offer better performance!

Example CSS:

```
@font-face {
  font-family: 'mytypeface';
  src: url('mytypeface-webfont.woff2') format('woff2'),
       url('mytypeface-webfont.woff') format('woff');
  font-weight: 500;
  font-style: normal;
  font-display: swap;
```

```
}
```

Google Fonts also provides a `display` URL query string parameter. For example:

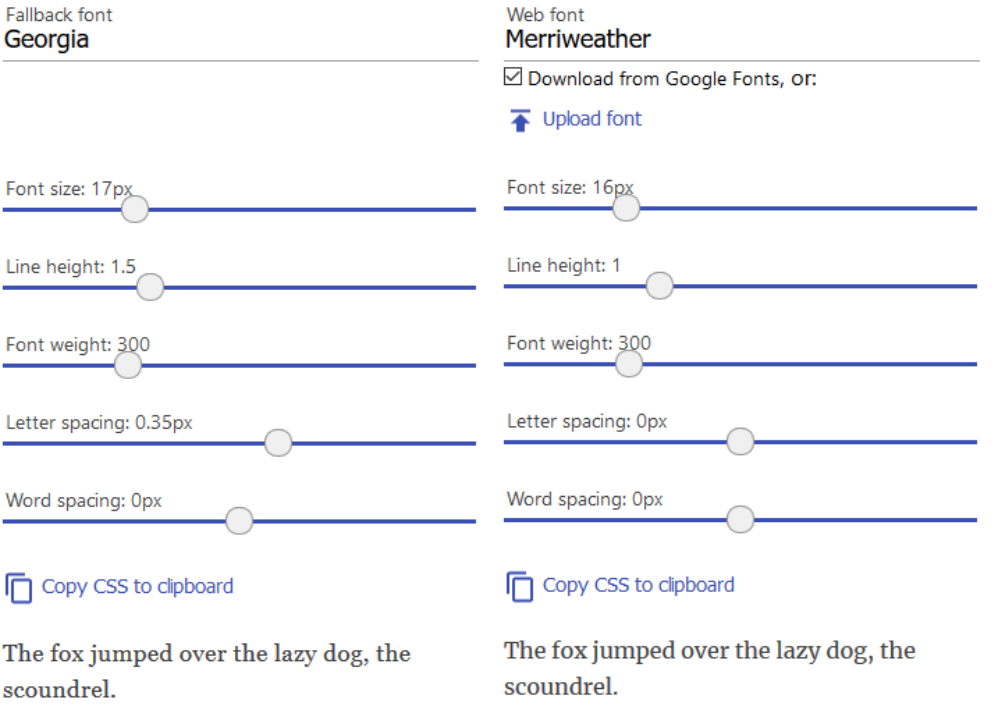
- https://fonts.googleapis.com/css?family=Inconsolata:500,700&display=swap

Settings for Specific Text Types

Different text blocks could use different `font-display` settings. For example, body text could use `swap` (FOUT) so it can be read immediately, while menus and heading text use `block` (FOIT).

A pragmatic compromise could be considered, which uses a fallback font with similar weights, line heights, and spacing to the web font. `font-display: swap` (FOUT) can then be used, but the replacement effect is less noticeable.

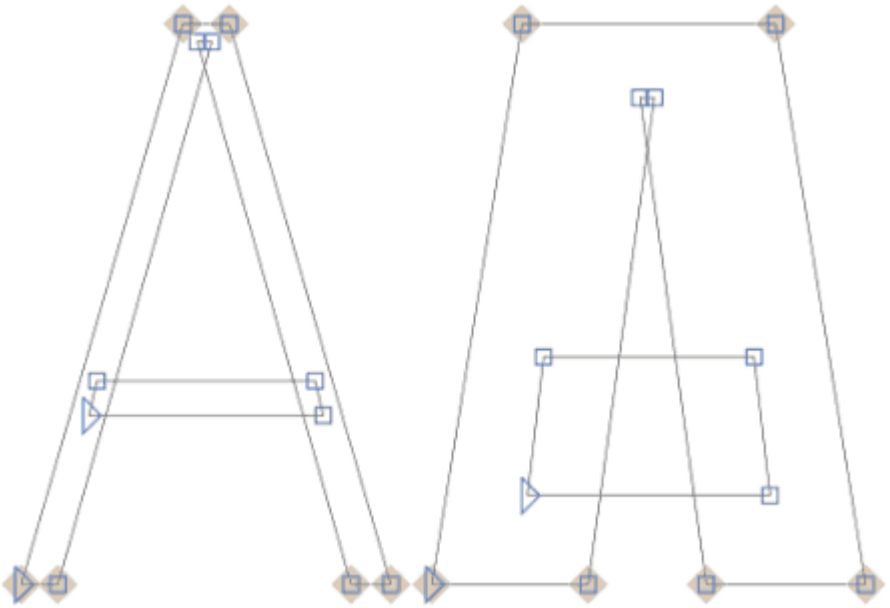
A tool such as Font Style Matcher can be used to find suitable fallback parameters.



4-11. Font Style Matcher

Consider Variable Fonts

OpenType 1.8 introduced **variable fonts**, and they're supported in most browsers (except IE). Rather than creating multiple files for each variation of the same typeface, a font is defined with minimum and maximum vector limits along an axis.



4-12. Variable font axis definitions

Any weight between the two extremes can be interpolated. A single variable font can therefore be used instead of several variations in order to reduce page weight and improve performance.

Open-source and commercial variable fonts can be found at sites including:

- Variable Fonts
- Axis Praxis
- Font Playground
- Recursive—a revolutionary font that includes monospace and casual settings

These can then be loaded using `@font-face` with a woff2-variations format and the allowable ranges. For example:

```
@font-face {
  font-family: 'VariableFont';
  src: 'variablefont.woff2' format('woff2-variations');
  font-weight: 200 800;
```

```
  font-stretch: 75% 125%;
  font-style: oblique 0deg 20deg;
}
```

Browser support for variable fonts can be tested using `@supports` with `font-variation-settings`:

```
body {
  font-family: sans-serif;
}

@supports (font-variation-settings: 'wght' 500) {

  body {
    font-family: 'VariableFont';
  }

}
```

Aspects of the typeface can then be adjusted in CSS, including the weight (typically 0 to 1000):

```
font-weight: 500;
/* or */
font-variation-settings: 'wght' 500;
```

Also width—*or stretch*—can be adjusted to produce condensed and extended variations (100% is normally the default, with lower values creating narrower fonts and higher values creating wider fonts):

```
font-stretch: 80%;
/* or */
font-variation-settings: 'wdth' 80;
```

Whether or not italics are required can also be set (either on or off, since italics are often defined as a different character set):

```
font-style: italic;
/* or */
font-variation-settings: 'ital' 1;
```

Also slant—*or oblique*—can be adjusted, which modifies the axis in a different way from italic (typically between 0 and 20 degrees):

```
font-style: oblique 10deg;
/* or */
font-variation-settings: 'slnt' 10;
```

The shorthand font-variation-settings property allows multiple font aspects to be set:

```
font-variation-settings: 'wght' 300, 'wdth' 100, 'slnt' 0;
```

OS Fonts as Variable Font Fallback

It's possible to download a single variable font but retain multiple fonts for older browsers. Unfortunately, modern browsers will download every font specified, which negates any performance benefit. It's therefore preferable to use an OS font as the fallback.

Use Modern CSS3 Layouts

For many years, it's been necessary to use CSS floats to lay out pages. The technique was always a hack and required considerable code, along with endless margin/padding tweaking to make the layout work. Even then, floats break at smaller screen sizes unless media queries are used.

Floats are no longer necessary:

- **Flexbox** should be used for one-dimensional layouts, which (can) wrap to the next row according to the widths of each block. It's ideal for menus, image galleries, cards, etc. Flexbox is supported by most browsers including IE10+.

- **Grid** is for two-dimensional layouts with explicit rows and columns. It's ideal for page layouts. Grid is supported by most browsers, although IE10/11 use an older version of the standard.

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

Fallbacks for Older Browsers

It's possible to use float-based fallbacks for older browsers. However, it's often better to use a simpler, single-column layout rather than trying to emulate what you achieved using Flexbox or Grid. Pixel perfection is futile!

Remove Unused CSS

The smaller your style sheet, the quicker it will download, the sooner it will parse, and the faster your page will become.

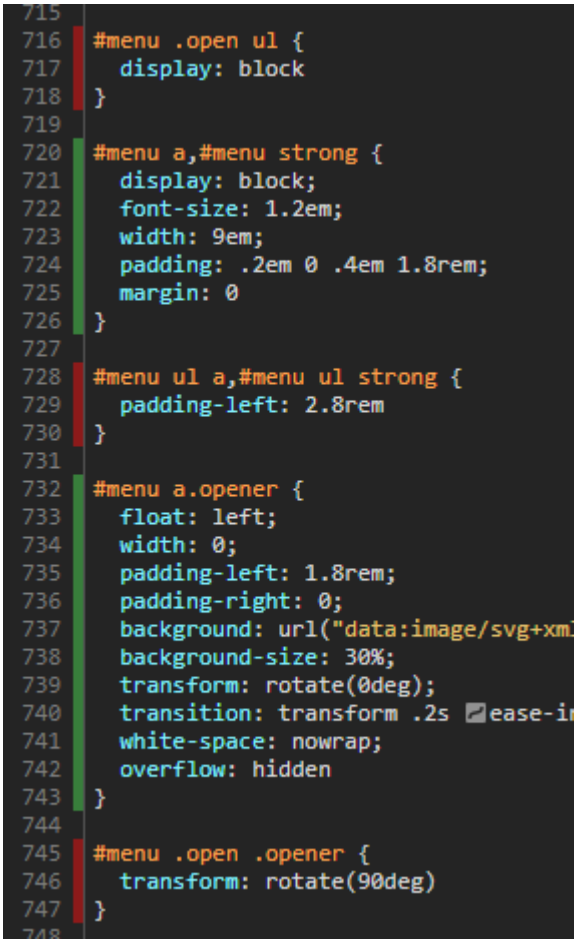
We all start with good intentions, but CSS can bloat over time as the number of features increases. It's easier to retain old, unnecessary code than remove it and risk breaking something. Those using a CSS framework such as Bootstrap may find they're only using a fraction of the facilities.

CSS removal recommendations:

1. Organize CSS into smaller files (partials) with clear responsibilities (which can be concatenated into a single file at build time). It's easier to remove a carousel widget if the CSS is clearly defined in `widgets/_carousel.css`.
2. Consider naming methodologies such as BEM to aid the development of discrete components.
3. Avoid deeply nested Sass/pre-processor declarations. The expanded code can become unexpectedly large.

4. Avoid using `!important` to override the cascade.
5. Avoid inline styles in HTML.

Chrome's **Coverage** panel helps locate unused CSS and JavaScript code. Select **Coverage** from the DevTools **More tools** sub-menu, then hit the record button and browse your application. Click any file to open its source. Unused code is highlighted in red in the line number gutter.



4-13. Chrome code coverage

Coverage for Single Pages Only

Chrome doesn't remember used/unused code as you navigate to new pages! The Coverage panel is only practical for single-page applications.

The following tools provide options to analyze HTML and CSS usage either at build time or by crawling URLs so that redundant code can be identified. Note that some configuration will be required to ensure styles triggered by JavaScript and user interactions are whitelisted.

- PurifyCSS (there's also an online version)
- PurgeCSS
- UnCSS
- UnusedCSS

Alternatively, a visual regression system such as Percy could be used to compare old and new screenshots.

Those preferring a manual—*and considerably more hardcore*—process could add an invisible background image to suspicious selectors. For example:

```
/* check usage */
.amiused1 {
  color: #abc;
  background-image: url(/used.png?.amiused1/);
}

#another .suspect {
  color: #123;
  background-image: url(/used.png?#another-.suspect/);
}
```

Either selector can be removed if no reference to their background image appears in server logs over a reasonable usage period.

Be Wary of Expensive CSS Properties

Not all CSS properties are created equally. Those that take longer to paint than others include:

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

This does't mean you shouldn't use them, but be wary of applying expensive effects to hundreds of elements, as it will affect rendering and scrolling performance.

Keeping Selectors Simple

Try to simplify CSS selectors where possible. CSS performance improvements may be negligible, but simpler selectors are easier to maintain, reduce page weight, and have a better chance of working in older browsers.

Embrace CSS3 Animations

Native CSS3 transitions and animations will always be faster and require less code than JavaScript-powered equivalents. It shouldn't be necessary to add a library or framework for typical fade, show, hide, and move effects. Very old browsers may not support the properties, but CSS degrades gracefully, and users will rarely know they're missing anything.

JavaScript animations should only be considered when fine-grained control is required—such as for HTML5 games, interactive charts, `<canvas>` manipulation, and so on.

Avoid Animating Expensive Properties

Once the browser has parsed the HTML document and styles, it renders elements in three stages:

1. **Layout**: the calculation of how much space an element requires and how it affects elements around it
2. **Paint**: the filling of pixels with color
3. **Composite**: the drawing of layers in the correct order when they overlap

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

1. opacity and/or
2. transform to translate (move), scale, skew, or rotate an element (the original space the element used is not altered so the layout is not affected)

Browsers often use the hardware-accelerated GPU to render these effects in their own layer. If neither property is ideal for your animation, consider taking the element out of the page flow with `position: absolute;` or similar to avoid complex layout changes.

Indicate Which Elements Will Animate

The **will-change** property allows CSS authors to indicate how an element will be animated so the browser can make performance optimizations in advance—for example, to declare that an element will have a transform applied:

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

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

- Only use `will-change` as a last resort to fix animation issues. *It should not be used to anticipate performance problems*.
- Don't apply it to too many elements.
- Give it sufficient time to work. Don't begin animations immediately.

Use CSS Containment

CSS Containment is a new (experimental) feature that indicates when an element's subtree is independent from the rest of the page. This can improve rendering performance during animations or when elements are added, modified, or removed from the DOM. The new CSS `contain` property accepts one or more of the following values in a space-separated list:

- `none` : containment is not applied.
- `layout` : the internal layout of the element is isolated from the rest of the page. Its content cannot have any effect on ancestor elements.
- `paint` : children of the element will not be displayed outside its boundary. Any overflows will not be visible (similar to `overflow: hidden;`).
- `size` : the size of the element can be determined without checking its children. The dimensions are independent of the content.
- `style` : counters and quotes cannot appear outside the element. (*This value may be dropped from the specification.*)

Two special values are also available:

- `strict` : all containment rules except `style` are applied. This is equivalent to `contain: layout paint size;` .
- `content` : all containment rules except `size` and `style` are applied. This is equivalent to `contain: layout paint;` .

Imagine you have a page with an unordered `<ul>` list containing one thousand child `<li>` list elements. If you change the contents of a single item that has `contain: strict;` applied, the browser won't attempt to recalculate the size or position of that item, others in the list, or any other elements on the page.

Check the Save-Data Header

The `Save-Data` field is an HTTP request header indicating that reduced data usage is preferred. It's named **Lite mode** in Chrome and can be enabled or disabled by the user.

When enabled, the `Save-Data` header is sent with every browser request. For example:

```
GET /image.jpg HTTP/1.0
Host: example.com
Save-Data: on
```

A server can respond accordingly when `Save-Data` is detected. For example, it can respond by:

- reducing the volume of HTML content—such as returning 100 rows of table data rather than 500
- providing low-resolution versions of an image even when high-resolution options are requested
- removing non-essential JavaScript such as trackers or advertising scripts

To ensure the minimal content is not cached and reused after the user disables `Save-Data`, the server should set the following header in the HTTP response:

```
Vary: Accept-Encoding, Save-Data
```

The `Save-Data` header can also be detected using client-side JavaScript:

```
if ('connection' in navigator && navigator.connection.saveData) {
  // Save-Data enabled
}
```

An optimum solution could presume data-saving by default, but add a `full-data` class to the HTML element when the header is *not* enabled:

```
if ('connection' in navigator && !navigator.connection.saveData) {
  document.documentElement.classList.add('full-data');
}
```

CSS and JavaScript components could then react accordingly. For example:

```
header {
  background-image: url("low-res-hero.jpg");
}

.full-data header {
  background-image: url("high-res-hero.jpg");
}
```

Adopt Progressive Web App Technologies

Progressive web apps (PWAs) can enhance performance by caching essential files locally. They're usually more responsive than standard web apps and can even be faster than native apps.

PWAs comprise a mixture of technologies that make web apps function like native mobile apps and overcome the constraints imposed by web-only and native-only solutions:

1. The app requires a single codebase developed with open, standard W3C web technologies.
2. Users can discover and install a PWA from the Web. There's no need to abide with app store rules or fees.
3. PWAs can work offline and update automatically.

Most tutorials describe how to build a native-looking, single-page, mobile-like app. However, any site can benefit from PWA technologies and be working within a few hours. There are three essential requirements ...

1. Enable HTTPS

PWAs require an HTTPS connection, although Chrome, for example, permits an HTTP `localhost` or `127.x.x.x` address during testing.

2. Create a Web App Manifest

The web app manifest provides information about your application, such as the name, description, and images. These are used by the OS to configure home screen icons, splash pages, and viewport settings.

The manifest is a JSON text file in the root of your app. It must be served with a `Content-Type: application/manifest+json` or `Content-Type: application/json` HTTP header:

```
{
  "lang"             : "en-US",
  "dir"              : "ltr",
  "name"             : "Standard Name",
  "short_name"       : "Short Name",
  "description"      : "A description of the site/app",
  "scope"            : "/",
  "start_url"        : "/",
  "display"          : "minimal-ui",
  "theme_color"      : "#fff",
  "background_color" : "#fff",
  "icons": [
    {
      "src"          : "https://site.com/icon-076.png",
      "sizes"        : "76x76",
      "type"         : "image/png"
    },
    {
      "src"          : "https://site.com/icon-192.png",
      "sizes"        : "192x192",
      "type"         : "image/png"
    },
    {
      "src"          : "https://site.com/icon-512.png",
      "sizes"        : "512x512",
```

```
      "type"          : "image/png"
    }
  ]
}
```

A list of manifest properties can be found on MDN, or you can use the Generate Web Manifest tool.

A link to the manifest file is required in the `<head>` of all your pages:

```
<link rel="manifest" href="/app.webmanifest">
```

3. Create a Service Worker

Service workers are programmable proxies that can intercept and respond to network requests. They're a single JavaScript file that resides in the application root.

Your page JavaScript must check for service worker support and register the file:

```
if ('serviceWorker' in navigator) {

  // register service worker
  navigator.serviceWorker.register('/service-worker.js');

}
```

`service-worker.js` then triggers and reacts to events, including:

- `install` when the app is first run. This can be used to cache regularly used files.
- `fetch` when a network request is made. This can return a cached file or make further network requests.

```
const
```

```
  staticCacheName = 'cache-v1';
  filesToCache = [
    '/',
    'style/main.css',
    'js/main.js',
    'images/hero.jpg'
  ];

// install event: cache regularly used files
self.addEventListener('install', event => {

  event.waitUntil(
    caches.open(staticCacheName)
    .then(cache => {
      return cache.addAll(filesToCache);
    })
  );

});

'install'// fetch event: serve files from cache or network
self.addEventListener('fetch', event => {

  event.respondWith(
    caches.match(event.request)
    .then(response => {
      if (response) {
        return response; // from cache
      }
      return fetch(event.request); // from network
    })
    .catch(error => {})
  );

});
```

This example doesn't update cached files or attempt to cache further requests, but it illustrates the basics of progressive web apps. Further PWA tutorials can be found at:

- Web Fundamentals: Progressive Web Apps
- MDN Progressive Web Apps

- Retrofit Your Website as a Progressive Web App

Power Down Inactive Tabs

Although we're mostly concerned with page performance when a user interacts with our site, we should also be responsible when the tab is inactive.

Browsers normally throttle events such as requestAnimationFrame, intervals, and timeouts on inactive tabs, but we can take this further to auto-pause and resume games, animations, video playback, Ajax polling, WebSocket handling, background loading, notifications, and so on. The less work an inactive tab does, the longer the smartphone battery will last, and the more likely the user can return to your site!

The Page Visibility API can be used to detect whether or not a tab is active and trigger an event when visibility changes. The following code adds a `tab-active` class to the `<html>` element when the tab is being viewed:

```
console.log('tab is', isTabActive() ? 'active' : 'not active');

document.addEventListener('visibilitychange', isTabActive);

function isTabActive() {

  if (document.visibilityState === 'visible') {
    // tab is active
    document.documentElement.classList.add('tab-active');
    return true;
  }
  else {
    // tab is inactive
    document.documentElement.classList.remove('tab-active');
    return false;
  }

});
```

CSS could then be used to start or stop animations. For example:

```
.myelement {
  animation: something 3s linear 1s infinite alternate;
  animation-play-state: paused;
}

.tab-active .myelement {
  animation-play-state: running;
}
```

Other Throttling Techniques

Similar throttling techniques could be used with these tools:

- The NetworkInformation API, to determine when connection speeds could affect performance. The API is experimental, has limited support, and may not be accurate.
- The Battery Status API, to detect when device power falls below a specific threshold. While this may be implemented in some browsers, the API was dropped as a web standard owing to privacy concerns. An individual could be identified and tracked by their fairly unique battery status.
- The Ambient Light API, to determine whether a device is being used in strong or dim lighting and modify the theme accordingly—such as increased contrast in strong sunlight and dimmer colors in darker situations. The API is experimental and has limited support.

Consider Inlining Critical CSS

Google page analysis tools often make a suggestion to "inline critical CSS" or "reduce render-blocking style sheets". Loading a CSS file blocks rendering, so performance can be improved by:

1. Extracting the styles used to render elements above the fold. Tools such as critical and criticalCSS can help.
2. Inlining those styles in a `<style>` element in the HTML `<head>`.
3. Loading the main CSS file asynchronously using JavaScript at the bottom of the page, or perhaps once the DOM is ready.

The technique noticeably improves performance—even on a fast connection—and will boost audit scores. It could benefit progressive web or single-page apps with consistent interfaces, but may be more difficult to manage on other sites:

- The "fold" is different on every device.
- Many sites have a variety of page layouts. Each could require different critical CSS, so a build tool becomes essential.
- Critical CSS tools can struggle with specific frameworks, HTML generated by client-side code, or dynamic, event-driven changes.
- The technique mostly benefits the user's first page load. CSS is cached for subsequent pages so additional inlined styles will increase page weight.

Provide Accelerated Mobile Pages (AMP)

The AMP project was announced in October 2015. A collaboration between Google and more than 30 news publishers aimed to improve mobile web performance. **AMP** is an open-source web component framework which claims that "you can easily create user-first websites, stories, emails, and ads."

AMP requires you to publish existing or original content as an AMP HTML page. AMP HTML is a subset of HTML5, providing a limited set of web components, styles, images, videos, and advertisements. Features and styling are purposely restricted, and you can't add custom JavaScript. Most AMP pages are served from Google's AMP cache—a proxy-based CDN that assigns a Google-specific URL to the page. This ensures optimal delivery using Google's global network.

AMP is fast, so it's mentioned in this book. However, while the project may have started with noble aims, AMP has been criticized for serving Google more than publishers and users, for reasons such as this:

- AMP is not necessarily faster or more efficient than your own optimized website.
- Unless you go AMP-only, you must duplicate existing content pages.
- Alternative URLs may be confusing to users.
- Google gains control of your content, visitors, and data.

- AMP could be considered a closed alternative to the open web.

Google wants the Web to be faster, yet AMP pages receive preferential treatment in mobile search results even when the original site is more efficient.

Ultimately, the decision is yours. There are WordPress plugins and CDNs such as Cloudflare that can automatically create AMP pages from your content but, for many sites, AMP will require further development effort. Those pages may receive additional publicity, but whether it's you or Google who benefits is another matter.

AMP development guides:

- amp.dev
- Convert HTML to AMP
- AMP development tools
- Official AMP Plugin for WordPress

AMP criticisms:

- Google's AMP HTML
- AMPersand
- The meaning of AMP
- The Two Faces of AMP
- Google AMP Can Go To Hell
- Kill Google AMP before it kills the web

Feeling Full Yet?

We may have lost weight, but the only way to guarantee long-term benefits is to change our development attitude! The next chapter provides life-changing diets.

Chapter

5

Life-Changing Diets

The performance techniques described in this chapter are more radical and could be difficult to apply to an existing project. Fortunately, there are no such limitations when embarking on a new site or app, so we look deeper into CMS issues, JavaScript optimization, DOM handling, server-side rendering, static site generators (SSGs), and development processes.

Evaluate CMS Templates and Plugins

Content management systems such as WordPress don't generate bloated, badly performing pages … *until you start adding stuff!*

Free or commercial templates make financial sense. Why employ a developer when an off-the-shelf solution does everything you need for a few dollars? Unfortunately, there's a hidden cost. Generic templates must sell hundreds of copies—if not thousands—to recoup the development effort. To attract buyers, the developers bundle every conceivable feature. Your site may only use a fraction of those facilities, but they can still be present in the code, so the download weight and processing are affected.

Similarly, be wary about plugins, since their quality and effectiveness vary. The best plugins can improve performance by optimizing database tables, caching data, and removing redundant code. The worst will duplicate assets, make convoluted configuration changes (such as `.htaccess` files), add unnecessary bloat, and affect responsiveness even though they're inactive on a particular page.

Always evaluate page cost and performance when considering new templates and plugins. Where possible, choose more lightweight options, even if the purchase price is higher.

Reduce Client-side Code

Blindingly obvious statement alert: *smaller files results in faster pages.*

Not all assets are created equal, though. 500KB of image data has a relatively low performance hit, since it's downloaded once, cached in the browser, and

positioned on the page. The same quantity of HTML, CSS, or JavaScript has a far bigger impact, because it must be downloaded, parsed, and processed.

Ideally, the number of HTML DOM nodes should be reduced to a minimum. A shallower tree depth means rendering and reflows are performed more effectively. Modern layout tools such as Flexbox and Grid allow you to remove wrapper elements that may have been necessary in float-based designs. Keep the document small and look out for signs of DIVitis!

Similarly, the fewer CSS rules you require, the quicker a document can be rendered. Look our for complex selectors, especially when using preprocessors such as Sass, which expand deeply nested rule sets. Check your compiled style sheet output to ensure it's as efficient as is practical.

Try to embrace the CSS cascade rather than working against it! A little understanding can reduce code and improve performance. For example, you can set default fonts, colors, sizes, tables, grids, and form fields that are universally applied but can be tweaked for individual components.

Also be wary of using CSS resets, which means having to re-apply default styling to every element. CSS normalization, such as Normalize.css, *could* be a better alternative, since it makes browsers render more consistently. That said, default styling between browsers is closer than ever.

Optimize JavaScript Code

HTML is a robust technology; even the oldest browsers without HTML5 support will show content. Similarly, CSS can fail to download or have coding errors, but the page remains viewable. By contrast, JavaScript is fragile and computationally expensive. A single error, unsupported command, or long-running task can prevent further code from running.

It's difficult to recommend JavaScript optimizations, since all applications will be different, but there are a few general tips that could improve performance. That said, be wary of micro-optimizations, which may shave a few milliseconds but aren't called frequently enough to make a difference. Use your browser's

developer tools to check whether any gains have been achieved.

Use JavaScript Sparingly

If a browser can do something in HTML and/or CSS alone, that should be your preferred option. You can still apply progressive enhancements where necessary (discussed below).

Modern browsers have implemented many regularly used features that previously required scripting, such as form validation, field auto-complete, animations, video, expanding text, modal dialogs, and more. There will be challenges—ask anyone who's ever tried styling a `<select>` drop-down—but using a native feature will always be faster and use less code.

Consider the choice of using an HTML `<button>` verses a `<div>` as a form submit. The HTML code starts in a similar way:

```
<button>submit</button>
```

Styling a DIV in CSS may be easier:

```
<div class="button">submit</div>
```

However, the HTML `<button>`:

1. offers default styling to look like an OS button
2. works on all browsers even when CSS or JavaScript fails
3. works immediately, as the page loads and before JavaScript has started executing
4. will automatically submit its parent `<form>` (if validity checks pass)
5. can be operated with a mouse, touch screen, keyboard, or any other input device

6. can receive focus, and accepts keypress shortcuts
7. requires no ARIA roles or other accessibility assistance

A button that's simulated in CSS and JavaScript requires significant effort, and it will never function as effectively as the native HTML alternative.

Avoid Long-running Tasks

Long-running tasks often trigger *unresponsive browser* messages, which prompt the user to halt JavaScript execution. Complex processing is best handled by a **Web Worker**, which allows a script to run in a background thread.

Web Worker scripts are limited. They can't interact with the page DOM, and must communicate with the main script using a message API, but they're able to perform Ajax requests and launch their own child workers.

Bind Events Sparingly

Applications can have dozens of event handlers. A handler function is registered to an event when it's triggered on a specific DOM element—such as running the `doSomething()` function when a click is detected on the `myElement` node:

```
myElement.addEventListener('click', doSomething);
```

Each bound event has a performance hit. Ideally, you should only add events you require, return from handler functions quickly, and unbind using removeEventListener when an event is no longer necessary.

Also be wary of quick-firing events such as `mousemove` and `scroll`, which can trigger rapid and wasteful rerunning of handler functions. One way around this is to use throttling to ensure an event is called no more than once every `N` milliseconds. For example:

```
// thottle event to delay ms
```

```
function eventThrottle(element, event, callback, delay = 300) {

  let throttle;
  element.addEventListener(event, (e) => {

    throttle = throttle || setTimeout(() => {
      throttle = null;
      callback(e);
    }, delay);

  }, false);

}

// call windowScrollHandler no more than once every 300ms
eventThrottle(window, 'scroll', windowScrollHandler);
```

Alternatively, **debouncing** can be used to ensure a handler is only called after the event has stopped being triggered for `N` milliseconds:

```
// debounce event until it no longer occurs for delay ms
function eventDebounce(element, event, callback, delay = 300) {

  let debounce;
  element.addEventListener(event, (e) => {
    clearTimeout(debounce);
    debounce = setTimeout(() => callback(e), delay);
  }, false);

}

// call windowScrollHandler when at least 300ms has elapsed since the last event
eventThrottle(window, 'scroll', windowScrollHandler);
```

Finally, remember to make effective use of event delegation. For example, presume you have an HTML `<table>` with thousands of cells and want to react to a `<td>` being clicked. Attaching an event to each cell requires significant processing and would need to be reapplied if the table changed. Instead, you can attach a single event handler to the `<table>` element and examine the target. For example:

```
// handle a click on any <td> element
document.getElementById('mytable').addEventListener('click', (e) => {

  let t = e.target.closest('td');
  if (!t) return;

  console.log('clicked cell', t);

});
```

Analyze Modified Code

It's rare to encounter code that hasn't been modified before it reaches the browser!

- Minifiers attempt optimizations such as rearranging lines or expanding loops.
- Transpilers such as Babel convert ES6 to ES5 so the code runs in older browsers.
- Compilers such as TypeScript, CoffeeScript, and Flow convert alternative or superset syntaxes to JavaScript.
- Projects such as Blazor convert C# to WebAssembly—a low-level, assembly-like language that offers near-native OS performance in JavaScript engines.

All offer stability and performance benefits, but check that the conversion is optimal and that it's not unnecessarily importing several kilobytes of transpiler library code. For example, consider the following 32-byte ES6 for...of loop:

```
for (let p of n) console.log(p);
```

This results in 598 bytes of Babel-transpiled code. Each additional loop adds a similar quantity of code, and none will execute in IE11—*which partly defeats the point of transpiling!* Options to consider:

- Use ES5 or more transpiler-efficient ES6 code to achieve the same result.
- Use differential loading to serve ES6 module-based code to modern browsers and larger transpiled scripts to older applications.

- Drop support for browsers without ES6 support (primarily IE). Your site or application can remain usable if you adopt server-side rendering and progressive enhancement techniques.

Modify the DOM Effectively

Some modern JavaScript frameworks implement a virtual DOM. As you change page elements, the virtual DOM works out what's been altered and determines how and when to make modifications. Ultimately, it must still change the real DOM, and you can make similar optimizations to improve performance without the additional overhead of virtual DOM calculations.

Cache Regularly Used Nodes

Regularly used DOM nodes should be stored as JavaScript variables so they don't need to be re-fetched. The DOM references are retained even when other tree nodes are modified:

```
const
  main    = document.getElementsByTagName('main')[0],
  heading = main.querySelector('h1'),
  tables  = main.getElementsByTagName('table');
```

Search from Any Node

Rather than searching the whole tree from `document`, many DOM methods allow you to start from any node. The example above searches for the first heading and tables within the `<main>` element.

`querySelector()` and `querySelectorAll()` can find elements using jQuery-like CSS selectors. They're usually slower than `getElementById()`, `getElementsByTagName()` and `getElementsByClassName()`, although the speed difference is unlikely to affect most applications.

Running Benchmarks

Tools such as jsPerf.com provide a way to create code snippets and run benchmarks on any browser to prove the efficiency—or inefficiency—of alternative functions.

`getElementsByTagName()` and `getElementsByClassName()` also return *live* HTMLCollections, which update automatically as the DOM is modified—so that it's not necessary to rerun the query.

Minimize Reflows

When an element is added, modified, or removed from a page, it can trigger a cascade of layout changes to surrounding elements. For example, increasing a width by 1px could result in a neighboring element wrapping to the next line, which pushes all subsequent content down the page. It's therefore more efficient to make changes that can't impact the layout. For example:

- use opacity and/or transform to translate (move), scale, or rotate an element
- limit the scope of the reflow by changing elements low in the DOM tree (those without deeply nested children)
- update elements in their own `position: absolute;` or `position: fixed;` layer
- modify hidden elements (`display: none;`), then show them after the change has been applied

Batch-update Styles

The following example could cause three reflows:

```
let myelement = document.getElementById('myelement');
myelement.width = '100px';
myelement.height = '200px';
myelement.style.margin = '10px';
```

Performance can be improved by appending a class:

```
let myelement = document.getElementById('myelement');
myelement.classList.add('newstyles');
```

This applies CSS properties in one reflow operation:

```
.newstyles {
  width: 100px;
  height: 200px;
  margin: 10px;
}
```

Batch-update Elements

Try to minimize the number of times you interact with the DOM. An empty DocumentFragment can be used to build elements in memory before applying those changes to the page. For example, you can create an unordered list with three items like so:

```
// create list
let
  frag = document.createDocumentFragment(),
  ul = frag.appendChild( document.createElement('ul') );

for (let i = 1; i <= 3; i++) {
  let li = ul.appendChild( document.createElement('li') );
  li.textContent = 'item ' + i;
}

// append list to the DOM
document.body.appendChild(frag);
```

The DOM is only modified on the last line.

Use requestAnimationFrame

The window.requestAnimationFrame() method calls a function just before the

browser performs the next repaint—normally once every sixtieth of a second (approximately every 17ms, presuming no other render-blocking processes are occurring). It's normally used for animating frames in HTML5 games, although running it before any DOM update will be beneficial. For example:

```
function updateDOM() {
  let p = document.createElement('p');
  p.textContent = 'new element';
  document.body.appendChild( p );
}

requestAnimationFrame( updateDOM );
```

Consider Progressive Rendering

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

```
<head>

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

</head>
<body>

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

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

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

```
  </main>

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

</body>
```

Each `<link>` still blocks rendering, but for a shorter time, because the file is smaller. The page is usable sooner, since each component renders in sequence; the top of the page can be viewed while remaining content loads. A similar approach is often adopted by Web Components, which encapsulate CSS within the code.

The technique can be less practical in templates where the content dictates the layout (Flexbox and tables), since reflows are triggered more frequently as the page loads. Grid-based page layouts are generally more suitable.

There's some variation in how browsers treat progressive rendering, but the worst-case scenario is that the browser blocks rendering until all discovered CSS files have loaded. That's no worse than loading each in the `<head>`.

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

Use Server-side Rendering

Which process is quicker?

Process 1 (typically used by JavaScript frameworks):

1. Request a URL.
2. Respond with a (mostly) empty HTML file.
3. Download and execute JavaScript.
4. Use Ajax or similar techniques to fetch content according to the URL.

5. Load the content into the page body.

Process 2 (old-school method):

1. Request a URL.
2. Respond with the full HTML.

Server-side rendering is always quicker for the initial page load.

Loading a second page can be faster in Process 1, since it's able to start at step 4. Assets such as style sheets, JavaScript, and images may already be available and parsed. Unfortunately, a large proportion of visitors may only view a single page, and the payload is higher because a larger quantity of JavaScript is necessary.

This is a better-performing process:

1. Request a URL.
2. Load HTML directly from the server into the browser.
3. Download and execute JavaScript. Some *rehydration* may be necessary to initiate components with HTML data.
4. Use Ajax or similar techniques to fetch and populate content according to URL navigation changes.

This can be more difficult to manage, since not all JavaScript frameworks provide server-based rendering capabilities using Node.js, PHP, Ruby, Python, and so on.

Do You Need a JavaScript or CSS Framework?

A CSS and/or JavaScript framework can provide a good development structure for teams working on larger sites or applications. However, most are general-purpose tools: they provide a range of features you may not need or may have to adapt. Optimizing performance is often difficult because the core code isn't under your control.

While a framework is certainly useful for prototyping, always question whether it's necessary for the final site or application. How much weight does it add? Will it improve performance? Can it be updated easily? What happens when it's eventually abandoned?

Invest time in researching the choices. Without investigation, every application looks like a nail to developers who understand a specific hammer. You should certainly avoid using more than one framework—with the possible exception of server-side options, or compilers such as Svelte, which remove themselves from production code.

Even once you settle on a chosen framework, there may be modular or lightweight alternatives such as Preact instead of React, or bling.js instead of jQuery.

Ultimately, the most efficient and adaptable framework will be one written specifically for your application.

Use a Static Site Generator

Most people start web development by creating (static) HTML, CSS, and possibly JavaScript files. The resulting assets can be hosted anywhere and are fast because they don't use server- or client-side processing.

The main downside is content management: adding a new page could involve changing hard-coded navigation menus on every page in the site. At this point, developers often turn to server-side languages or a database-driven CMS, both of which have their own set of challenges.

What if you could create a fast, static site but make cross-site changes programmatically when something is added or removed? That's exactly what a **static site generator** (SSG) does. It takes content—typically defined in markdown files—and builds a set of static web pages. The build-time process can construct menus, import images, generate styles, and so on, and can be rerun when anything changes. The resulting site is decoupled from a server and is often referred to as using a **JAMstack**: JavaScript, APIs, and markup.

Most SSGs build a set of folder-based HTML files with associated assets that can be uploaded to any web server capable of serving static content. The Ruby-based Jekyll was one of the first SSGs, but StaticGen.com lists dozens of alternatives for a range of languages. Options such as Gatsby also create React-based JavaScript applications rather than HTML files. (*Whether or not that's a benefit is another matter!)*

A static site can offer the best site performance, since it's rendered once, then delivered to all users as is. There are no server-side dependencies, reliability is improved, version control is easy, and security issues can be eradicated.

There are some downsides:

- configuration and setup takes time and is more difficult than a CMS
- SSGs are rarely suitable for non-technical editors
- there's no concept of user roles or permission rights
- site consistency can be more difficult to enforce, as editors can add any client-side code
- the rebuild process can be slow, especially on larger sites

SSGs are ideal for sites that change relatively infrequently, but many of the issues can be overcome by importing data from a headless CMS or automating the build process.

Use a Build System

Even the most conscientious developer can forget to minimize a CSS file, optimize an image, or remove debugging `console` statements. Whatever technology you use to create a site or app, a **build process** can automate mundane tasks to ensure there are no oversights. Additionally, they can run tests, verify code, and deploy to staging or live servers.

Creating a build process can take a day or two, but it should save time over the long term. Popular generic build tools include Gulp.js, Grunt.js, Broccoli.js, and Brunch, which allow you to define and run tasks manually or when files are changed.

Alternatively, you could opt for web-specific module bundlers such as webpack or Parcel, which *understand* HTML, CSS, and JavaScript so they can parse and build optimized code, through operations like these:

- dead asset elimination
- code splitting and dependency handling
- ES6 to ES5 transpiling
- minification
- source map generation
- cache-busting
- live reloading
- enforcing performance budgets (discussed below)

Module bundlers often promise zero configuration … *although the reality may be somewhat different!*

A few tips to get started:

- Choose a build system and stick with it for a while.
- Automate the most frustrating tasks first.
- Try not to overcomplicate your build process. Spend an hour or two creating an initial setup, then evolve it over time.
- Do as much during the build process as possible. For example, an HTML template could be partially constructed from known data and partials rather than parsing everything at render time.

Further reading:

- A Guide to Using npm as a Build Tool
- An Introduction to Gulp.js
- A Beginner's Guide to Webpack
- A Beginner's Guide to Parcel

Use Progressive Enhancement

Progressive enhancement is a development approach rather than a technology.

Each site or app feature starts with a baseline minimum viable implementation—perhaps an HTML-only solution. Enhancements are then added progressively when they're supported by the user's device. Consider a simple search box:

1. The base solution is an HTML `<input type="search" />` field which, when a string is entered, triggers a new page load showing search results.
2. HTML5 constraint validation can be applied to ensure searching only occurs when a minimum of three characters has been entered.
3. CSS styles are applied, showing basic formatting such as fonts, colors, borders, etc.
4. When the field has focus, CSS animations could enlarge the field, show a submit button, etc.
5. JavaScript could show suggestions as the user types characters.
6. JavaScript could show a simple list of search results without the user having to leave the current page.
7. PWA service workers could be used to cache suggestions and search results for later use.

Where necessary, the code tests that a feature is supported before attempting the enhancement. For example, suggestions could be implemented when JavaScript is running, events are supported, and the HTML5 `<datalist>` element is available.

Adding Missing Features with Polyfills

It's often possible to use a *polyfill* to add a missing feature to browsers without native support. This can range from additional prototypes, such as the String.padStart() method, through to full APIs, such as one to provide geolocation support using IP lookups.

Polyfill.io provides a custom set of polyfills. However, be wary about the performance cost of attempting to polyfill everything. It may be preferable to offer IE users a fast, rudimentary feature than a slow, fully polyfilled experience.

In the search box example above, progressive enhancement offers the following benefits:

- The search box is device agnostic and works in all browsers—old, current, and those released tomorrow.
- Assuming the HTML loads, the search box is always operational. This includes the period before CSS and/or JavaScript is downloaded and parsed. In performance terms, the feature is responsive immediately.
- The user gets the best possible experience their device can handle. Performance isn't affected when an enhancement can't be added.
- The search box is fault-tolerant: any enhancement can work or fail without breaking the system. It doesn't matter whether CSS and/or JavaScript are blocked, are slow to arrive, or fail to download.
- It's the responsible option, and doesn't require more development effort in most situations.

The approach has no downsides. Progressive enhancement only breaks when:

1. It isn't considered from the start. It may be difficult to retrospectively enhance a feature that already requires a high base-level of CSS and JavaScript.
2. You try to support all browsers equally. It's futile to expect a decade-old version of IE to behave the same as a modern application. Progressive

enhancement means you never need to worry about old browsers. Their users may not receive the best experience, but the feature remains usable.

Adopt a Performance Budget

A performance budget imposes a limit on related metrics. Typical options include:

- quantity-based limits, such as the maximum number of fonts, images, scripts, etc.
- time-based limits, such as the first meaningful paint or interactive times
- rule-based limits, such as a minimum performance and accessibility score in Lighthouse audits

You should experiment and discuss options with stakeholders to establish baseline criteria, such as:

- the total size of a page must not exceed 500KB
- a single image must be no more than 150KB
- the home page must deliver less than 100KB of JavaScript
- all pages must be readable within five seconds on a mid-range mobile device operating on an average 3G connection

Ideally, these criteria can be added to your build process. Tools such as the Lighthouse module and file size plugins can report—*and potentially block*—any deviation from the budget. Exceeding the budget means you must either:

1. optimize an existing feature/asset
2. lazy load an existing feature/asset on demand
3. remove an existing feature/asset
4. reject the new feature/asset

The limitations can help teams prioritize features. Increasing the budget should always be tougher than implementing another solution! For example, a budget increase must be discussed, justified, and agreed to by a two-thirds majority at a

monthly progress meeting!

Performance budget tools:

- performancebudget.io: estimate file sizes according to download timings
- bundlesize: calculate file sizes during the build process
- SpeedCurve: track real-world performance (commercial)

Create a Style Guide

A style guide is a set of agreed brand, content, design, and coding standards for teams generally working on large codebases developed over a long period. A good style guide promotes consistency and illustrates how developers should approach solutions. Front-end components can be demonstrated with example code that shows styling, animation, functionality, and restrictions. The benefits include:

- new team members can become productive quickly
- components are reused: developers are less likely to introduce their own HTML, CSS, and JavaScript
- it becomes easier to update, maintain, and improve component performance when the same code is used throughout
- code can be tested and quality assurance becomes simpler
- users receive a consistent UI experience

A style guide can be as rigid or as flexible as you require. It's often best to develop it as a set of HTML pages that can demonstrate code and be updated quickly. Example documents are available from styleguides.io.

Simplify and Streamline

Performance problems often start because stakeholders equate more features with more customers. This is rarely the case; most people prefer simplicity. They're not using your site/app on a daily basis and just want to get a task done quickly and easily.

Average page weight reached 2MB because developers let it happen. We're under pressure to deliver more in a shorter time, but are we doing the job effectively when it results in a slow, clunky application no one wants to use? Few clients will understand the intricacies of web performance, so it's our responsibility to use efficient coding practices and to highlight potential pitfalls in layman's terms.

1. Be wary of the performance cost of any added features.
2. Use analytics to monitor and identify little-used features.
3. Fully remove unnecessary features or replace them with sleeker, lightweight alternatives.

Look after the bytes and the megabytes will take care of themselves!

Learn to Love the Web

The Web evolved from a document publishing platform to an application delivery system that revolutionized the way we distribute and use software. Unfortunately, this has resulted in an alarming tendency to over-engineer solutions when simpler options could be more effective. Rather than choose a native HTML control, we import the latest JavaScript module. Instead of adding a few styles, we copy vast quantities of CSS from Stack Overflow and Bootstrap.

If there's one piece of advice to take away from this book, it's *learn the basics*. Somewhat contradictorily, HTML and CSS are either disregarded as too simplistic to warrant respect or considered impenetrable technologies that must be fixed using JavaScript. Yet they're the fundamental building blocks of the Web:

- HTML5 has around 120 elements. Half of those will rarely be used, but there's usually a better alternative to `<div>` and `<span>`.
- There are almost 400 CSS3 properties and more are being added. No one could name them all, but they're modularized. The foundations can be learned quickly, but experimentation and experience is required to understand the concepts.

Learning HTML and CSS will make you a better web developer and advance your JavaScript skills. A little knowledge will considerably improve your application's performance.

Chapter

6

Check, Please!

I hope you're feeling full after your extensive buffet of performance delicacies. Not all dishes will have been to your taste, but you should have found a few recipes to try.

The main reason we don't have a fast and responsive web is because we let it become slow and bloated. Performance is rarely given equal priority with other features. In most cases, it's never even acknowledged until someone complains about speed. Like SEO or usability, it's possible to improve performance toward the end of a project—but it's far more effective to implement it from the start.

Software development is a complex process, and it will always be possible to make optimizations. However, if you target the big, easy wins first, the tougher refactoring or rewrites will become less necessary. Ideally, you should consider performance every time you add a feature or asset to your site or application. It will make you a more conscientious developer and will win the respect of your peers and users.

Let's strive to build a better web!

Further reading:

1. MDN web performance
2. The Cost of JavaScript
3. The Ethics of Web Performance