To paraphrase Tolstoy, all fast websites are alike, but all slow websites are slow in different ways.

This document is a brief guide to improve your Gatsby site's performance.

Part 1: Creating a process for performance improvements

Because websites are hard to make fast and easy to make slow, performance improvement isn't just something that takes place at one point in time. Instead, performance is a system that you put in place to maintain and extend performance gains over time.

Step 1: Choose a testing tool

In order to get a baseline, you need to decide what you consider the "source of truth".

Lighthouse / PageSpeed Insights and Webpagetest are the most common website performance testing tools in the Gatsby community. Lighthouse tends to be seen as more "canonical". Webpagetest tends to be seen as more precise.

These tools measure what are known as "Core Web Vitals", which measure both time to page load and time to page interactivity. Google's official page has more detail on what these metrics are.

Step 2: Set up performance monitoring

Site performance monitoring becomes increasingly important the more long-lasting your project is. You can put a lot of effort into performance work -- but then ship a performance regression that wipes out all your hard work!

There are three options for this:

- Gatsby Cloud has Lighthouse performance reports built into its CI/CD. Every time someone opens a pull request or merges into master, a Lighthouse report will be run and the performance score displayed.
- Use a third-party service that offers performance monitoring, such as Calibre or Speedcurve
- Integrate Webpagetest or Lighthouse reports into your CI/CD process, via eg webhooks or GitHub Actions.

For additional precision, run a test multiple times and take the median result.

Step 3: Quantify the impact of each change you make

While you're doing performance improvement work, it's important to understand the impact of each change or set of changes. You may find that one change gives you a 2-point Lighthouse improvement and another gives you a 20-point improvement.

There's an easy way to measure this:

1. Write a pull request with each change or set of changes, using a service like Gatsby Cloud or Netlify that generates deploy previews per-PR.
2. Run Lighthouse tests against the deploy preview for that branch and the deploy preview for master. Do not compare the live site to a deploy preview since the CDN setup may differ.
3. Calculate the difference in Lighthouse scores.
4. If you don't see a significant difference from a change, consider batching similar changes together until the difference is noticeable and thinking about the changes as a group.

Performance work can be surprisingly nonlinear in impact.

Part 2: Implement Improvements.

When you run a test in your testing tool of choice, it will give you a number of recommendations. These can be bucketed into five core categories:

- Blocking calls & third-party scripts.
- JavaScript bundle size.
- Stylesheets and font files.
- Images and other media.
- Resource requests & CDN caching configuration.

Every site is different, and the recommendations will give some guidance as to where the highest effort to impact ratio is on your site.

Address third-party-script impact

Various types of calls made in your HTML, like calls to external font files, will block page load or page interactivity in different ways. In addition, third-party scripts can execute "eagerly", often delaying page load while they do so.

Step 1: Remove unneeded and high-cost, low-value scripts

For each script, it can be helpful to understand the business purpose, relative importance, and who's using the data. This will allow you to identify and remove any unused scripts.

There may also be relatively unimportant scripts that have a high performance costs; these are also good candidates for removal.

Step 2: lazy load or inline scripts

One of the lowest-hanging fruits is to set your scripts to load lazily rather than "eagerly" (the default). Any `<script>` tags being embedded manually can be set to `<script async>`.

For slightly more effort, you can get additional performance gains; rather than loading third-party scripts from external sources, you can inline scripts in your code to reduce the cost of a network call.

There are a number of places to put an inlined script, depending whether you need it to execute immediately upon loading or can defer execution.

- *No deferring*: This is a good default. Put the script in onPreRenderHTML to have it added to your document tree. You can place it lower in your DOM to have parsed and evaluated later.
- *Some deferring*: You can place the script in onClientEntry to have it execute after page load, but before the browser renders the page.
- *More deferring*: You can place the script in onInitialClientRender to have it execute after the browser renders the page.

Note that if you are already using html.js, you should modify that file to include your snippet instead of using `onPreRenderHTML`. This will have the same behavior.

Reduce your JavaScript bundle cost

Among all assets, JavaScript can be uniquely costly to your performance. This is due to three main reasons:

- Like other assets, it needs to be loaded into your browser.
- Unlike most other assets, it's code. That means it needs to be parsed by your browser, which can "block" other work from happening.
- With third-party npm modules, it's easy to accidentally add a lot of JavaScript you don't actually need.

To reduce your JavaScript bundle size, follow the steps below.

Step 1: Profile your bundle

The first step to fix this is to figure out what's going on. Use `gatsby-plugin-webpack-bundle-analyser-v2` and the *experimental* plugin `gatsby-plugin-perf-budgets` to profile your bundle. When you add this tool to your `gatsby-config.js`, you can analyze the bundle size on each page in your application.

Chunk naming patterns help you break down a page into three categories:

- application-level chunks (shared with all pages),
- template-level chunks (shared with all pages of the same template),
- page-level data imports

Visuals illustrating this breakdown: all chunks on a page, only application-level chunks, only template-level chunks.

Step 2: Remove unneeded third-party imports from application-level chunks

Start by auditing your application-level chunks, especially `commons.js` -- the bundle that is shared by all components.

Start by inspecting third-party package size. Anything over 50kb, and certainly 100kb, is worth examining whether it's needed. Some common culprits include: Moment.js, Lodash, Material UI, but you'll want to inspect your individual libraries.

To prevent large imports from recurring, consider using a tool that displays the size of library imports you're pulling in. The Import Cost extension for Visual Studio Code and BundlePhobia are good resources.

In addition, eyeball all the medium-sized packages (10-50kb). If it doesn't look like there's a good reason for that particular package to be in the commons, carefully audit your import structure.

One edge case: If you're importing Redux globally, Redux can pull in data bundles that don't seem to be related. If you're only using Redux on a page or two, you may want to check the impact by removing it entirely and running Lighthouse.

Step 3: Audit your application-level chunks for components and data that don't need to be on every page

One common challenge is inadvertently pulling in more code or data than you intend to your commons bundle.

For example, let's say you have a header that imports a JSON object in order to check a key / value pair on that object. Whether that JSON object is 5kb or 50kb, you've now imported it into every page on your site!

There's a couple ways to detect this:

- *Notice components and data that don't seem to be needed on every page.* If you're using v2 of Gatsby, certain methods of importing can cause code to get bundled on pages it doesn't belong on. Try replacing indirect import statements like `import { myIcon } from './icons/index.js'` with direct imports like `import { myIcon } from './icons/my-icon.js'`.
- *Watch for unexpectedly large data imports.* If you notice large JSON objects, and you do need the data (or some portion of it), there are a couple options.
    - If you only need a small portion of that data, consider getting it a different way. Split up the JSON file, query it via GraphQL, or import it in `gatsby-node.js` and pass through only the subset of data you need.

- If you need the data, but not right away (perhaps it's lower in the page, or being used by an event handler), you might consider switching to asynchronously fetching it.

Step 4: On critical paths, identify unneeded code & components

Your site likely has critical paths on your site that are very important to business success. This might be your home page, a signup page, a product template, and so on.

Inspect critical paths' template-level chunks for large third-party libraries and unneeded components. Repeat the process from steps two and three to identify optimization opportunities.

Step 5: On critical paths, lazy-load below-the-fold components

Gatsby's default behavior is to bundle the entire page together. However, there may be some components that don't need to be loaded right away. Perhaps your home page is quite long and you're willing to defer loading elements farther down on the page if it makes the initial page load faster.

One way you can do this is to lazy-load below-the-fold components using `loadable-components`. `loadable-components` is the recommended lazy-loading solution for all server-side-rendered React applications, including Gatsby websites.

We recommend you use the gatsby plugin to install `loadable-components`. This plugin ensures that all components are still server-rendered for performance benefits.

Step 6: Consider using the Preact plugin

Preact is a UI library that works similarly to React, but is much smaller (~3kb compressed as opposed to ~40kb). gatsby-plugin-preact is a drop-in plugin that will render your site in Preact instead of React, cutting 35-40kb of JavaScript from your bundle.

This step can make sense if the `framework.js` bundle is a large part of your overall bundle size, and want to further optimize.

Using Preact is an advanced way to decrease bundle size. Note that in certain, occasional edge cases this can create ill-documented, odd user interactions. We do not recommend this for sites with complex UI logic, like a SaaS app. After installing `gatsby-plugin-preact`, you'll need to use Preact Developer Tools instead of React Developer Tools to inspect your component behavior.

Styling & Fonts

When working with CSS and fonts, there are certain additional patterns you'll need to follow or tools to use in order to optimize performance.

Step 1: Check for globally bundled CSS

Without properly scoped and imported CSS you can end up with a large bundle with all your CSS getting pulled in on every page. What you want, instead, is a small bundle pulling in only needed CSS.

You can use the Coverage drawer in Chrome's Dev Tools to detect the proportion of unused CSS on each page.

Gatsby inlines CSS into the initial HTML file. So when running Coverage look for the line with the current page's UR and take note of the % of bytes that are unused and the page size.

A "good" scenario is having between 20-40% of unused code -- this includes, for example CSS to define responsive layouts that isn't evaluated on desktop format. A "bad" scenario is 80-90% unused code and a 1mb file.

You can dig deeper by scroll through the page on the usage drawer to look at the unused CSS and get a sense of whether it is needed for that page. To fix these issues, look at moving to a modular CSS solution like CSS modules.

Step 2: If you're using a CSS-in-JS library, use the Gatsby plugin

If you're using a CSS-in-JS library like styled-components or emotion, use the relevant plugin: gatsby-plugin-styled-components or gatsby-plugin-emotion.

These plugins server-side render the styles; otherwise, the output HTML will intersperse `<style>` tags with HTML elements, which can cause the browser to perform costly layout reflow.

Step 3: Optimize fonts

Font files can usually be reduced in size significantly. If your font file is over 50kb, it's too large. In addition, fonts block page load, so it's important to think about reducing network calls.

- **Prefer `woff2`. Don't use `ttf`.** `woff2` is a compressed font format, supported by browsers used by over 95% of Internet users. A few legacy browsers need `woff`.Like using `avif` and `webp` instead of `png` and jpg, using the correct format can significantly cut down the amount of data sent over the network.
- **Self-host rather than installing from an external CDN.** Having the font file available locally will save a trip over the network and reduce blocking time.
- **Use Latin font subsets only** (if creating a Latin-language site). It's common to accidentally include font extensions (Greek, Cyrillic, Devnagari, Chinese) when typically you only need the Latin base set. The Google Webfonts Helper app can help you do this with free fonts.

Font optimizations are usually small, but easy performance wins.

Images & Media

Media files are often the largest files on a site, and so can delay page load significantly while they are pulled over the network, especially if their location is not well-defined.

Gatsby Plugin Image is our approach to optimizing image loading performance. It does three basic things:

1. It delays non-essential work for images not above the fold to avoid resource congestion.
2. It provides a placeholder during image fetch.
3. It minimizes image file size to reduce request roundtrip time.

The `gatsby-plugin-image` documentation is fairly exhaustive, ranging from why image optimization is important to how to implement Gatsby Plugin Image.

Implementing Gatsby Image is typically the bulk of image- and media-related performance optimization.

Resource Requests & CDN Configuration

Part of the work in loading a Gatsby site is minimizing the time to transport bits over the network. There's a number of ways to do this.

- *Load critical assets from your main domain where possible*. Some people use another domain for their images. This can have a 300ms delay when it comes to LCP compared to loading it from the main CDN. This is sometimes necessitated by company policies; try to avoid it if possible.

- *Preconnect to subdomains* using gatsby-plugin-preconnect. The impact of this is very site-specific, and while it's usually positive, we've seen this actually slow down page loads on occasion, so you'll want to test this.

- *Utilize Gatsby Link*. Gatsby Link is our approach to optimizing the intra-site navigation experience. We pre-load linked pages on your site so that transitioning between pages is smooth and seamless. Here's a guide to using Gatsby Link.
- *Implement proper CDN caching policies.* Configuration differs on a per-CDN basis, but since Gatsby generates unique bundle names via hashes, bundles can be cached indefinitely since they are immutable.

Additional Resources

- We did a "talk" version of this documentation at GatsbyConf 2021: "The Anatomy of A Performance Audit" (video,slides)
- Smashing Magazine's Frontend Performance Checklist is an in-depth, more generalized guide to performance optimization. It isn't specific to Gatsby, so some of the things it mentions Gatsby will already do for you.
- We've written additional material in the past, for example Kyle Mathews's deep dive blog, Dustin Schau's deep dive blog, and the Gatsby Guide.
- The Gatsby team is available to engage with teams looking to optimize performance through the Gatsby Concierge Service