SVELTEKIT • BEST PRACTICES

Performance

ON THIS PAGE

Out of the box, SvelteKit does a lot of work to make your applications as performant as possible:

- Code-splitting, so that only the code you need for the current page is loaded
- Asset preloading, so that 'waterfalls' (of files requesting other files) are prevented
- File hashing, so that your assets can be cached forever
- Request coalescing, so that data fetched from separate server `load` functions is grouped into a single HTTP request
- Parallel loading, so that separate universal `load` functions fetch data simultaneously
- Data inlining, so that requests made with `fetch` during server rendering can be replayed in the browser without issuing a new request
- Conservative invalidation, so that `load` functions are only re-run when necessary
- Prerendering (configurable on a per-route basis, if necessary) so that pages without dynamic data can be served instantaneously
- Link preloading, so that data and code requirements for a client-side navigation are eagerly anticipated

Nevertheless, we can't (yet) eliminate all sources of slowness. To eke out maximum performance, you should be mindful of the following tips.

Diagnosing issues

ways to understand the performance characteristics of a site that is already deployed to the internet.

Your browser also includes useful developer tools for analysing your site, whether deployed or running locally:

Chrome - Lighthouse, Network, and Performance devtools

Edge - Lighthouse, Network, and Performance devtools

Firefox - Network and Performance devtools

Safari - enhancing the performance of your webpage

Note that your site running locally in `dev` mode will exhibit different behaviour than your production app, so you should do performance testing in preview mode after building.

Instrumenting

If you see in the network tab of your browser that an API call is taking a long time and you'd like to understand why, you may consider instrumenting your backend with a tool like OpenTelemetry or Server-Timing headers.

Optimizing assets

Images

Reducing the size of image files is often one of the most impactful changes you can make to a site's performance. Svelte provides the `@sveltejs/enhanced-img` package, detailed on the images page, for making this easier. Additionally, Lighthouse is useful for identifying the worst offenders.

Videos

Docs

friendly formats such as `.webm` or `.mp4`.

You can lazy-load videos located below the fold with `preload="none"` (though note that this will slow down playback when the user *does* initiate it).

Strip the audio track out of muted videos using a tool like FFmpeg.

Fonts

SvelteKit automatically preloads critical `.js` and `.css` files when the user visits a page, but it does *not* preload fonts by default, since this may cause unnecessary files (such as font weights that are referenced by your CSS but not actually used on the current page) to be downloaded. Having said that, preloading fonts correctly can make a big difference to how fast your site feels. In your `handle` hook, you can call `resolve` with a `preload` filter that includes your fonts.

You can reduce the size of font files by subsetting your fonts.

Reducing code size

Svelte version

We recommend running the latest version of Svelte. Svelte 5 is smaller and faster than Svelte 4, which is smaller and faster than Svelte 3.

Packages

`rollup-plugin-visualizer` can be helpful for identifying which packages are contributing the most to the size of your site. You may also find opportunities to remove code by manually inspecting the build output (use `build: { minify: false }` in your Vite config to make the output readable, but remember to undo that before deploying your app), or via the network tab of your browser's devtools.

Try to minimize the number of third-party scripts running in the browser. For example, instead of using JavaScript-based analytics consider using server-side implementations, such as those offered by many platforms with SvelteKit adapters including Cloudflare, Netlify, and Vercel.

To run third party scripts in a web worker (which avoids blocking the main thread), use Partytown's SvelteKit integration.

Selective loading

Code imported with static `import` declarations will be automatically bundled with the rest of your page. If there is a piece of code you need only when some condition is met, use the dynamic `import(...)` form to selectively lazy-load the component.

Navigation

Preloading

You can speed up client-side navigations by eagerly preloading the necessary code and data, using link options. This is configured by default on the `<body>` element when you create a new SvelteKit app.

Non-essential data

For slow-loading data that isn't needed immediately, the object returned from your `load` function can contain promises rather than the data itself. For server `load` functions, this will cause the data to stream in after the navigation (or initial page load).

Preventing waterfalls

CSS which requests a background image and web font. SvelteKit will largely solve this class of problems for you by adding `modulepreload` tags or headers, but you should view the network tab in your devtools to check whether additional resources need to be preloaded. Pay special attention to this if you use web fonts since they need to be handled manually.

If a universal `load` function makes an API call to fetch the current user, then uses the details from that response to fetch a list of saved items, and then uses *that* response to fetch the details for each item, the browser will end up making multiple sequential requests. This is deadly for performance, especially for users that are physically located far from your backend. Avoid this issue by using server `load` functions where possible.

Server `load` functions are also not immune to waterfalls (though they are much less costly since they rarely involve roundtrips with high latency). For example if you query a database to get the current user and then use that data to make a second query for a list of saved items, it will typically be more performant to issue a single query with a database join.

Hosting

Your frontend should be located in the same data center as your backend to minimize latency. For sites with no central backend, many SvelteKit adapters support deploying to the *edge*, which means handling each user's requests from a nearby server. This can reduce load times significantly. Some adapters even support configuring deployment on a per-route basis. You should also consider serving images from a CDN (which are typically edge networks) — the hosts for many SvelteKit adapters will do this automatically.

Ensure your host uses HTTP/2 or newer. Vite's code splitting creates numerous small files for improved cacheability, which results in excellent performance, but this does assume that your files can be loaded in parallel with HTTP/2.

performant web app. You should be able to apply information from general performance resources such as Core Web Vitals to any web experience you build.

Edit this page on GitHub

PREVIOUS
Auth

NEXT
Images