SVELTEKIT • APPENDIX

Migrating to SvelteKit v2

ON THIS PAGE

Upgrading from SvelteKit version 1 to version 2 should be mostly seamless. There are a few breaking changes to note, which are listed here. You can use `npx sv migrate sveltekit-2` to migrate some of these changes automatically.

We highly recommend upgrading to the most recent 1.x version before upgrading to 2.0, so that you can take advantage of targeted deprecation warnings. We also recommend updating to Svelte 4 first: Later versions of SvelteKit 1.x support it, and SvelteKit 2.0 requires it.

redirect and error are no longer thrown by you

Previously, you had to `throw` the values returned from `error(...)` and `redirect(...)` yourself. In SvelteKit 2 this is no longer the case — calling the functions is sufficient.

```
import { error } from '@sveltejs/kit'

// ...
throw error(500, 'something went wrong');
error(500, 'something went wrong');
```

`svelte-migrate` will do these changes automatically for you.

If the error or redirect is thrown inside a `try {...}` block (hint: don't do this!), you can distinguish them from unexpected errors using `isHttpError` and `isRedirect` imported from `@sveltejs/kit`.

When receiving a `Set-Cookie` header that doesn't specify a `path`, browsers will set the cookie path to the parent of the resource in question. This behaviour isn't particularly helpful or intuitive, and frequently results in bugs because the developer expected the cookie to apply to the domain as a whole.

As of SvelteKit 2.0, you need to set a `path` when calling `cookies.set(...)`, `cookies.delete(...)` or `cookies.serialize(...)` so that there's no ambiguity. Most of the time, you probably want to use `path: '/'`, but you can set it to whatever you like, including relative paths — `''` means 'the current path', `'.'` means 'the current directory'.

```
/** @type {import('./$types').PageServerLoad} */
export function load({ cookies }) {
  cookies.set(name, value, { path: '/' });
  return { response }
}
```

`svelte-migrate` will add comments highlighting the locations that need to be adjusted.

Top-level promises are no longer awaited

In SvelteKit version 1, if the top-level properties of the object returned from a `load` function were promises, they were automatically awaited. With the introduction of streaming this behavior became a bit awkward as it forces you to nest your streamed data one level deep.

As of version 2, SvelteKit no longer differentiates between top-level and non-top-level promises. To get back the blocking behavior, use `await` (with `Promise.all` to prevent waterfalls, where appropriate):

```
// If you have a single promise
/** @type {import('./$types').PageServerLoad} */
export async function load({ fetch }) {
  const response = await fetch(url).then(r => r.json())
```

```js
// If you have multiple promises
/** @type {import('./$types').PageServerLoad} */
export async function load({ fetch }) {
  const a = fetch(url1).then(r => r.json());
  const b = fetch(url2).then(r => r.json());
  const [a, b] = await Promise.all([
    fetch(url1).then(r => r.json()),
    fetch(url2).then(r => r.json()),
  ]);
  return { a, b };
}
```

goto(...) changes

`goto(...)` no longer accepts external URLs. To navigate to an external URL, use `window.location.href = url`. The `state` object now determines `$page.state` and must adhere to the `App.PageState` interface, if declared. See shallow routing for more details.

paths are now relative by default

In SvelteKit 1, `%sveltekit.assets%` in your `app.html` was replaced with a relative path by default (i.e. `.` or `..` or `../..` etc, depending on the path being rendered) during server-side rendering unless the `paths.relative` config option was explicitly set to `false`. The same was true for `base` and `assets` imported from `$app/paths`, but only if the `paths.relative` option was explicitly set to `true`.

This inconsistency is fixed in version 2. Paths are either always relative or always absolute, depending on the value of `paths.relative`. It defaults to `true` as this results in more portable apps: if the `base` is something other than the app expected (as is the case when viewed on the Internet Archive, for example) or unknown at build time (as is the case when deploying to IPFS and so on), fewer things are likely to break.

Previously it was possible to track URLs from `fetch` es on the server in order to rerun load functions. This poses a possible security risk (private URLs leaking), and as such it was behind the `dangerZone.trackServerFetches` setting, which is now removed.

preloadCode arguments must be prefixed with base

SvelteKit exposes two functions, `preloadCode` and `preloadData` , for programmatically loading the code and data associated with a particular path. In version 1, there was a subtle inconsistency — the path passed to `preloadCode` did not need to be prefixed with the `base` path (if set), while the path passed to `preloadData` did.

This is fixed in SvelteKit 2 — in both cases, the path should be prefixed with `base` if it is set.

Additionally, `preloadCode` now takes a single argument rather than *n* arguments.

resolvePath has been removed

SvelteKit 1 included a function called `resolvePath` which allows you to resolve a route ID (like `/blog/[slug]`) and a set of parameters (like `{ slug: 'hello' }`) to a pathname. Unfortunately the return value didn't include the `base` path, limiting its usefulness in cases where `base` was set.

As such, SvelteKit 2 replaces `resolvePath` with a (slightly better named) function called `resolveRoute` , which is imported from `$app/paths` and which takes `base` into account.

```
import { resolvePath } from '@sveltejs/kit';
import { base } from '$app/paths';
import { resolveRoute } from '$app/paths';

const path = base + resolvePath('/blog/[slug]', { slug });
const path = resolveRoute('/blog/[slug]', { slug });
```

result with `base`, you need to remove that yourself.

Improved error handling

Errors are handled inconsistently in SvelteKit 1. Some errors trigger the `handleError` hook but there is no good way to discern their status (for example, the only way to tell a 404 from a 500 is by seeing if `event.route.id` is `null`), while others (such as 405 errors for `POST` requests to pages without actions) don't trigger `handleError` at all, but should. In the latter case, the resulting `$page.error` will deviate from the `App.Error` type, if it is specified.

SvelteKit 2 cleans this up by calling `handleError` hooks with two new properties: `status` and `message`. For errors thrown from your code (or library code called by your code) the status will be `500` and the message will be `Internal Error`. While `error.message` may contain sensitive information that should not be exposed to users, `message` is safe.

Dynamic environment variables cannot be used during prerendering

The `$env/dynamic/public` and `$env/dynamic/private` modules provide access to *run time* environment variables, as opposed to the *build time* environment variables exposed by `$env/static/public` and `$env/static/private`.

During prerendering in SvelteKit 1, they are one and the same. As such, prerendered pages that make use of 'dynamic' environment variables are really 'baking in' build time values, which is incorrect. Worse, `$env/dynamic/public` is populated in the browser with these stale values if the user happens to land on a prerendered page before navigating to dynamically-rendered pages.

Because of this, dynamic environment variables can no longer be read during prerendering in SvelteKit 2 — you should use the `static` modules instead. If the user lands on a prerendered page, SvelteKit will request up-to-date values for `$env/dynamic/public` from

from use:enhance callbacks

If you provide a callback to `use:enhance`, it will be called with an object containing various useful properties.

In SvelteKit 1, those properties included `form` and `data`. These were deprecated some time ago in favour of `formElement` and `formData`, and have been removed altogether in SvelteKit 2.

Forms containing file inputs must use multipart/form-data

If a form contains an `<input type="file">` but does not have an `enctype="multipart/form-data"` attribute, non-JS submissions will omit the file. SvelteKit 2 will throw an error if it encounters a form like this during a `use:enhance` submission to ensure that your forms work correctly when JavaScript is not present.

Generated tsconfig.json is more strict

Previously, the generated `tsconfig.json` was trying its best to still produce a somewhat valid config when your `tsconfig.json` included `paths` or `baseUrl`. In SvelteKit 2, the validation is more strict and will warn when you use either `paths` or `baseUrl` in your `tsconfig.json`. These settings are used to generate path aliases and you should use the `alias` config option in your `svelte.config.js` instead, to also create a corresponding alias for the bundler.

getRequest no longer throws errors

The `@sveltejs/kit/node` module exports helper functions for use in Node environments, including `getRequest` which turns a Node `ClientRequest` into a standard `Request` object specified size limit. In SvelteKit 2, the error will not be thrown until later, when the request body (if any) is being read. This enables better diagnostics and simpler code.

vitePreprocess is no longer exported from @sveltejs/kit/vite

Since `@sveltejs/vite-plugin-svelte` is now a peer dependency, SvelteKit 2 no longer re-exports `vitePreprocess`. You should import it directly from `@sveltejs/vite-plugin-svelte`.

Updated dependency requirements

SvelteKit 2 requires Node `18.13` or higher, and the following minimum dependency versions:

- `svelte@4`
- `vite@5`
- `typescript@5`
- `@sveltejs/vite-plugin-svelte@3` (this is now required as a `peerDependency` of SvelteKit — previously it was directly depended upon)
- `@sveltejs/adapter-cloudflare@3` (if you're using these adapters)
- `@sveltejs/adapter-cloudflare-workers@2`
- `@sveltejs/adapter-netlify@3`
- `@sveltejs/adapter-node@2`
- `@sveltejs/adapter-static@3`
- `@sveltejs/adapter-vercel@4`

`svelte-migrate` will update your `package.json` for you `tsconfig.json` extends from) now uses `"moduleResolution": "bundler"` (which is recommended by the TypeScript team, as it properly resolves types from packages with an `exports` map in package.json) and `verbatimModuleSyntax` (which replaces the existing `importsNotUsedAsValues` and `preserveValueImports` flags — if you have those in your `tsconfig.json`, remove them. `svelte-migrate` will do this for you).

Edit this page on GitHub

PREVIOUS
Breakpoint Debugging

NEXT
Migrating from Sapper