SVELTEKIT Docs - English

Table of contents

Introduction

Before we begin

> If you're new to Svelte or SvelteKit we recommend checking out the interactive tutorial.
>
> If you get stuck, reach out for help in the Discord chatroom.

What is SvelteKit?

SvelteKit is a framework for rapidly developing robust, performant web applications using Svelte. If you're coming from React, SvelteKit is similar to Next. If you're coming from Vue, SvelteKit is similar to Nuxt.

To learn more about the kinds of applications you can build with SvelteKit, see the FAQ.

What is Svelte?

In short, Svelte is a way of writing user interface components — like a navigation bar, comment section, or contact form — that users see and interact with in their browsers. The Svelte compiler converts your components to JavaScript that can be run to render the HTML for the page and to CSS that styles the page. You don't need to know Svelte to understand the rest of this guide, but it will help. If you'd like to learn more, check out the Svelte tutorial.

SvelteKit vs Svelte

Svelte renders UI components. You can compose these components and render an entire page with just Svelte, but you need more than just Svelte to write an entire app.

SvelteKit provides basic functionality like a router — which updates the UI when a link is clicked — and server-side rendering (SSR). But beyond that, building an app with all the modern best practices is fiendishly complicated. Those practices include build optimizations, so that you load only the minimal required code; offline support; preloading pages before the user initiates navigation; configurable rendering that allows you to render different parts of your app on the server with SSR, in the browser client-side rendering, or at build-time with prerendering; and many other things. SvelteKit does all the boring stuff for you so that you can get on with the creative part.

It reflects changes to your code in the browser instantly to provide a lightning-fast and feature-rich development experience by leveraging Vite with a Svelte plugin to do Hot Module Replacement (HMR).

Go to TOC

Creating a project

The easiest way to start building a SvelteKit app is to run `npm create`:

```
npm create svelte@latest my-app
cd my-app
npm install
npm run dev
```

The first command will scaffold a new project in the `my-app` directory asking you if you'd like to set up some basic tooling such as TypeScript. See integrations for pointers on setting up additional tooling. The subsequent commands will then install its dependencies and start a server on localhost:5173.

There are two basic concepts:

- Each page of your app is a Svelte component
- You create pages by adding files to the `src/routes` directory of your project. These will be server-rendered so that a user's first visit to your app is as fast as possible, then a client-side app takes over

Try editing the files to get a feel for how everything works.

Editor setup

We recommend using Visual Studio Code (aka VS Code) with the Svelte extension, but support also exists for numerous other editors.

Go to TOC

Project structure

A typical SvelteKit project looks like this:

```
my-project/
├ src/
| ├ lib/
| | ├ server/
| | | └ [your server-only lib files]
| | └ [your lib files]
| ├ params/
| | └ [your param matchers]
| ├ routes/
| | └ [your routes]
| ├ app.html
| ├ error.html
| ├ hooks.client.js
| └ hooks.server.js
├ static/
| └ [your static assets]
├ tests/
| └ [your tests]
├ package.json
├ svelte.config.js
├ tsconfig.json
└ vite.config.js
```

You'll also find common files like `.gitignore` and `.npmrc` (and `.prettierrc` and `.eslintrc.cjs` and so on, if you chose those options when running `npm create svelte@latest`).

Project files

src

The `src` directory contains the meat of your project. Everything except `src/routes` and `src/app.html` is optional.

- `lib` contains your library code (utilities and components), which can be imported via the `$lib` alias, or packaged up for distribution using `svelte-package`
  - `server` contains your server-only library code. It can be imported by using the `$lib/server` alias. SvelteKit will prevent you from importing these in client code.
- `params` contains any param matchers your app needs
- `routes` contains the routes of your application. You can also colocate other components that are only used within a single route here
- `app.html` is your page template — an HTML document containing the following placeholders:
  - `%sveltekit.head%` — `<link>` and `<script>` elements needed by the app, plus any `<svelte:head>` content

  - `%sveltekit.body%` — the markup for a rendered page. This should live inside a `<div>` or other element, rather than directly inside `<body>`, to prevent bugs caused by browser extensions injecting elements that are then destroyed by the hydration process. SvelteKit will warn you in development if this is not the case
  - `%sveltekit.assets%` — either `paths.assets`, if specified, or a relative path to `paths.base`
  - `%sveltekit.nonce%` — a CSP nonce for manually included links and scripts, if used
  - `%sveltekit.env.[NAME]%` - this will be replaced at render time with the `[NAME]` environment variable, which must begin with the `publicPrefix` (usually `PUBLIC_`). It will fallback to `''` if not matched.
- `error.html` is the page that is rendered when everything else fails. It can contain the following placeholders:
  - `%sveltekit.status%` — the HTTP status
  - `%sveltekit.error.message%` — the error message
- `hooks.client.js` contains your client hooks
- `hooks.server.js` contains your server hooks
- `service-worker.js` contains your service worker

(Whether the project contains `.js` or `.ts` files depends on whether you opt to use TypeScript when you create your project. You can switch between JavaScript and TypeScript in the documentation using the toggle at the bottom of this page.)

If you added Vitest when you set up your project, your unit tests will live in the `src` directory with a `.test.js` extension.

static

Any static assets that should be served as-is, like `robots.txt` or `favicon.png`, go in here.

tests

If you added Playwright for browser testing when you set up your project, the tests will live in this directory.

package.json

Your `package.json` file must include `@sveltejs/kit`, `svelte` and `vite` as `devDependencies`.

When you create a project with `npm create svelte@latest`, you'll also notice that `package.json` includes `"type": "module"`. This means that `.js` files are interpreted as native JavaScript modules with `import` and `export` keywords. Legacy CommonJS files need a `.cjs` file extension.

svelte.config.js

This file contains your Svelte and SvelteKit configuration.

tsconfig.json

This file (or `jsconfig.json`, if you prefer type-checked `.js` files over `.ts` files) configures TypeScript, if you added typechecking during `npm create svelte@latest`. Since SvelteKit relies on certain configuration being set a specific way, it generates its own `.svelte-kit/tsconfig.json` file which your own config `extends`.

vite.config.js

A SvelteKit project is really just a Vite project that uses the `@sveltejs/kit/vite` plugin, along with any other Vite configuration.

Other files

.svelte-kit

As you develop and build your project, SvelteKit will generate files in a `.svelte-kit` directory (configurable as `outDir`). You can ignore its contents, and delete them at any time (they will be regenerated when you next `dev` or `build`).

Go to TOC

Web standards

Throughout this documentation, you'll see references to the standard Web APIs that SvelteKit builds on top of. Rather than reinventing the wheel, we *use the platform*, which means your existing web development skills are applicable to SvelteKit. Conversely, time spent learning SvelteKit will help you be a better web developer elsewhere.

These APIs are available in all modern browsers and in many non-browser environments like Cloudflare Workers, Deno and Vercel Edge Functions. During development, and in adapters for Node-based environments (including AWS Lambda), they're made available via polyfills where necessary (for now, that is — Node is rapidly adding support for more web standards).

In particular, you'll get comfortable with the following:

Fetch APIs

SvelteKit uses `fetch` for getting data from the network. It's available in hooks and server routes as well as in the browser.

> A special version of `fetch` is available in `load` functions, server hooks and API routes for invoking endpoints directly during server-side rendering, without making an HTTP call, while preserving credentials. (To make credentialled fetches in server-side code outside `load`, you must explicitly pass `cookie` and/or `authorization` headers.) It also allows you to make relative requests, whereas server-side `fetch` normally requires a fully qualified URL.

Besides `fetch` itself, the Fetch API includes the following interfaces:

Request

An instance of `Request` is accessible in hooks and server routes as `event.request`. It contains useful methods like `request.json()` and `request.formData()` for getting data that was posted to an endpoint.

Response

An instance of `Response` is returned from `await fetch(...)` and handlers in `+server.js` files. Fundamentally, a SvelteKit app is a machine for turning a `Request` into a `Response`.

Headers

The `Headers` interface allows you to read incoming `request.headers` and set outgoing `response.headers`. For example, you can get the `request.headers` as shown below, and use the `json` convenience function to send modified `response.headers`:

```
// @errors: 2461
/// file: src/routes/what-is-my-user-agent/+server.js
import { json } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export function GET({ request }) {
    // log all headers
    console.log(...request.headers);

    // create a JSON Response using a header we received
    return json({
        // retrieve a specific header
        userAgent: request.headers.get('user-agent')
    }, {
        // set a header on the response
        headers: { 'x-custom-header': 'potato' }
    });
}
```

FormData

When dealing with HTML native form submissions you'll be working with `FormData` objects.

```
// @errors: 2461
/// file: src/routes/hello/+server.js
import { json } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export async function POST(event) {
    const body = await event.request.formData();

    // log all fields
    console.log([...body]);

    return json({
        // get a specific field's value
        name: body.get('name') ?? 'world'
    });
}
```

Stream APIs

Most of the time, your endpoints will return complete data, as in the `userAgent` example above. Sometimes, you may need to return a response that's too large to fit in memory in one go, or is delivered in chunks, and for this the platform provides streams — ReadableStream, WritableStream and TransformStream.

URL APIs

URLs are represented by the `URL` interface, which includes useful properties like `origin` and `pathname` (and, in the browser, `hash`). This interface shows up in various places — `event.url` in hooks and server routes, `$page.url` in pages, `from` and `to` in `beforeNavigate` and `afterNavigate` and so on.

URLSearchParams

Wherever you encounter a URL, you can access query parameters via `url.searchParams`, which is an instance of `URLSearchParams`:

```ts
// @filename: ambient.d.ts
declare global {
    const url: URL;
}

export {};

// @filename: index.js
// cut---
const foo = url.searchParams.get('foo');
```

Web Crypto

The Web Crypto API is made available via the `crypto` global. It's used internally for Content Security Policy headers, but you can also use it for things like generating UUIDs:

```ts
const uuid = crypto.randomUUID();
```

Go to TOC

Routing

At the heart of SvelteKit is a *filesystem-based router*. The routes of your app — i.e. the URL paths that users can access — are defined by the directories in your codebase:

- `src/routes` is the root route
- `src/routes/about` creates an `/about` route
- `src/routes/blog/[slug]` creates a route with a *parameter*, `slug`, that can be used to load data dynamically when a user requests a page like `/blog/hello-world`

> You can change `src/routes` to a different directory by editing the project config.

Each route directory contains one or more *route files*, which can be identified by their `+` prefix.

+page

+page.svelte

A `+page.svelte` component defines a page of your app. By default, pages are rendered both on the server (SSR) for the initial request and in the browser (CSR) for subsequent navigation.

```
<! file: src/routes/+page.svelte --->
<h1>Hello and welcome to my site!</h1>
<a href="/about">About my site</a>
```

```
<! file: src/routes/about/+page.svelte --->
<h1>About this site</h1>
<p>TODO...</p>
<a href="/">Home</a>
```

```
<! file: src/routes/blog/[slug]/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;
</script>

<h1>{data.title}</h1>
<div>{@html data.content}</div>
```

> Note that SvelteKit uses `<a>` elements to navigate between routes, rather than a framework-specific `<Link>` component.

+page.js

Often, a page will need to load some data before it can be rendered. For this, we add a `+page.js` module that exports a `load` function:

```
/// file: src/routes/blog/[slug]/+page.js
import { error } from '@sveltejs/kit';

/** @type {import('./$types').PageLoad} */
export function load({ params }) {
    if (params.slug === 'hello-world') {
        return {
            # 'Hello world!',
            content: 'Welcome to our blog. Lorem ipsum dolor sit amet...'
        };
    }

    throw error(404, 'Not found');
}
```

This function runs alongside `+page.svelte`, which means it runs on the server during server-side rendering and in the browser during client-side navigation. See `load` for full details of the API.

As well as `load`, `+page.js` can export values that configure the page's behaviour:

- `export const prerender = true` or `false` or `'auto'`
- `export const ssr = true` or `false`
- `export const csr = true` or `false`

You can find more information about these in page options.

+page.server.js

If your `load` function can only run on the server — for example, if it needs to fetch data from a database or you need to access private environment variables like API keys — then you can rename `+page.js` to `+page.server.js` and change the `PageLoad` type to `PageServerLoad`.

```
/// file: src/routes/blog/[slug]/+page.server.js

// @filename: ambient.d.ts
declare global {
    const getPostFromDatabase: (slug: string) => {
        # string;
        content: string;
    }
}

export {};

// @filename: index.js
// cut---
import { error } from '@sveltejs/kit';

/** @type {import('./$types').PageServerLoad} */
```

```
export async function load({ params }) {
    const post = await getPostFromDatabase(params.slug);

    if (post) {
        return post;
    }

    throw error(404, 'Not found');
}
```

During client-side navigation, SvelteKit will load this data from the server, which means that the returned value must be serializable using devalue. See `load` for full details of the API.

Like `+page.js`, `+page.server.js` can export page options — `prerender`, `ssr` and `csr`.

A `+page.server.js` file can also export *actions*. If `load` lets you read data from the server, `actions` let you write data *to* the server using the `<form>` element. To learn how to use them, see the form actions section.

+error

If an error occurs during `load`, SvelteKit will render a default error page. You can customise this error page on a per-route basis by adding an `+error.svelte` file:

```
<! file: src/routes/blog/[slug]/+error.svelte --->
<script>
    import { page } from '$app/stores';
</script>

<h1>{$page.status}: {$page.error.message}</h1>
```

SvelteKit will 'walk up the tree' looking for the closest error boundary — if the file above didn't exist it would try `src/routes/blog/+error.svelte` and then `src/routes/+error.svelte` before rendering the default error page. If *that* fails (or if the error was thrown from the `load` function of the root `+layout`, which sits 'above' the root `+error`), SvelteKit will bail out and render a static fallback error page, which you can customise by creating a `src/error.html` file.

If the error occurs inside a `load` function in `+layout(.server).js`, the closest error boundary in the tree is an `+error.svelte` file *above* that layout (not next to it).

If no route can be found (404), `src/routes/+error.svelte` (or the default error page, if that file does not exist) will be used.

> `+error.svelte` is *not* used when an error occurs inside `handle` or a +server.js request handler.

You can read more about error handling here.

+layout

So far, we've treated pages as entirely standalone components — upon navigation, the existing `+page.svelte` component will be destroyed, and a new one will take its place.

But in many apps, there are elements that should be visible on *every* page, such as top-level navigation or a footer. Instead of repeating them in every `+page.svelte`, we can put them in *layouts*.

+layout.svelte

To create a layout that applies to every page, make a file called `src/routes/+layout.svelte`. The default layout (the one that SvelteKit uses if you don't bring your own) looks like this...

```
<slot></slot>
```

...but we can add whatever markup, styles and behaviour we want. The only requirement is that the component includes a `<slot>` for the page content. For example, let's add a nav bar:

```
/// file: src/routes/+layout.svelte
<nav>
    <a href="/">Home</a>
    <a href="/about">About</a>
    <a href="/settings">Settings</a>
</nav>

<slot></slot>
```

If we create pages for `/`, `/about` and `/settings`...

```
/// file: src/routes/+page.svelte
<h1>Home</h1>
```

```
/// file: src/routes/about/+page.svelte
<h1>About</h1>
```

```
/// file: src/routes/settings/+page.svelte
<h1>Settings</h1>
```

...the nav will always be visible, and clicking between the three pages will only result in the `<h1>` being replaced.

Layouts can be *nested*. Suppose we don't just have a single `/settings` page, but instead have nested pages like `/settings/profile` and `/settings/notifications` with a shared submenu (for a real-life example, see github.com/settings).

We can create a layout that only applies to pages below `/settings` (while inheriting the root layout with the top-level nav):

```
<! file: src/routes/settings/+layout.svelte --->
<script>
    /** @type {import('./$types').LayoutData} */
```

```
    export let data;
</script>

<h1>Settings</h1>

<div class="submenu">
    {#each data.sections as section}
        <a href="/settings/{section.slug}">{section.title}</a>
    {/each}
</div>

<slot></slot>
```

By default, each layout inherits the layout above it. Sometimes that isn't what you want - in this case, advanced layouts can help you.

+layout.js

Just like `+page.svelte` loading data from `+page.js`, your `+layout.svelte` component can get data from a `load` function in `+layout.js`.

```
/// file: src/routes/settings/+layout.js
/** @type {import('./$types').LayoutLoad} */
export function load() {
    return {
        sections: [
            { slug: 'profile', # 'Profile' },
            { slug: 'notifications', # 'Notifications' }
        ]
    };
}
```

If a `+layout.js` exports page options — `prerender`, `ssr` and `csr` — they will be used as defaults for child pages.

Data returned from a layout's `load` function is also available to all its child pages:

```
<! file: src/routes/settings/profile/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;

    console.log(data.sections); // [{ slug: 'profile', # 'Profile' }, ...]
</script>
```

> Often, layout data is unchanged when navigating between pages. SvelteKit will intelligently rerun `load` functions when necessary.

+layout.server.js

To run your layout's `load` function on the server, move it to `+layout.server.js`, and change the `LayoutLoad` type to `LayoutServerLoad`.

Like `+layout.js`, `+layout.server.js` can export page options — `prerender`, `ssr` and `csr`.

+server

As well as pages, you can define routes with a `+server.js` file (sometimes referred to as an 'API route' or an 'endpoint'), which gives you full control over the response. Your `+server.js` file exports functions corresponding to HTTP verbs like `GET`, `POST`, `PATCH`, `PUT`, `DELETE`, `OPTIONS`, and `HEAD` that take a `RequestEvent` argument and return a `Response` object.

For example we could create an `/api/random-number` route with a `GET` handler:

```
/// file: src/routes/api/random-number/+server.js
import { error } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export function GET({ url }) {
    const min = Number(url.searchParams.get('min') ?? '0');
    const max = Number(url.searchParams.get('max') ?? '1');

    const d = max - min;

    if (isNaN(d) || d < 0) {
        throw error(400, 'min and max must be numbers, and min must be less than
max');
    }

    const random = min + Math.random() * d;

    return new Response(String(random));
}
```

The first argument to `Response` can be a `ReadableStream`, making it possible to stream large amounts of data or create server-sent events (unless deploying to platforms that buffer responses, like AWS Lambda).

You can use the `error`, `redirect` and `json` methods from `@sveltejs/kit` for convenience (but you don't have to).

If an error is thrown (either `throw error(...)` or an unexpected error), the response will be a JSON representation of the error or a fallback error page — which can be customised via `src/error.html` — depending on the `Accept` header. The `+error.svelte` component will *not* be rendered in this case. You can read more about error handling here.

When creating an `OPTIONS` handler, note that Vite will inject `Access-Control-Allow-Origin` and `Access-Control-Allow-Methods` headers — these will not be present in production unless you add them.

Receiving data

By exporting `POST` / `PUT` / `PATCH` / `DELETE` / `OPTIONS` / `HEAD` handlers, `+server.js` files can be used to create a complete API:

```
<! file: src/routes/add/+page.svelte --->
<script>
    let a = 0;
    let b = 0;
    let total = 0;

    async function add() {
        const response = await fetch('/api/add', {
            method: 'POST',
            body: JSON.stringify({ a, b }),
            headers: {
                'content-type': 'application/json'
            }
        });

        total = await response.json();
    }
</script>

<input type="number" bind:value={a}> +
<input type="number" bind:value={b}> =
{total}

<button on:click={add}>Calculate</button>
```

```
/// file: src/routes/api/add/+server.js
import { json } from '@sveltejs/kit';

/** @type {import('./$types').RequestHandler} */
export async function POST({ request }) {
    const { a, b } = await request.json();
    return json(a + b);
}
```

In general, form actions are a better way to submit data from the browser to the server.

Content negotiation

`+server.js` files can be placed in the same directory as `+page` files, allowing the same route to be either a page or an API endpoint. To determine which, SvelteKit applies the following rules:

- `PUT` / `PATCH` / `DELETE` / `OPTIONS` requests are always handled by `+server.js` since they do not apply to pages
- `GET` / `POST` / `HEAD` requests are treated as page requests if the `accept` header prioritises `text/html` (in other words, it's a browser page request), else they are handled by `+server.js`.
- Responses to `GET` requests will include a `Vary: Accept` header, so that proxies and browsers cache HTML and JSON responses separately.

$types

Throughout the examples above, we've been importing types from a `$types.d.ts` file. This is a file SvelteKit creates for you in a hidden directory if you're using TypeScript (or JavaScript with JSDoc type annotations) to give you type safety when working with your root files.

For example, annotating `export let data` with `PageData` (or `LayoutData`, for a `+layout.svelte` file) tells TypeScript that the type of `data` is whatever was returned from `load`:

```
<! file: src/routes/blog/[slug]/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;
</script>
```

In turn, annotating the `load` function with `PageLoad`, `PageServerLoad`, `LayoutLoad` or `LayoutServerLoad` (for `+page.js`, `+page.server.js`, `+layout.js` and `+layout.server.js` respectively) ensures that `params` and the return value are correctly typed.

If you're using VS Code or any IDE that supports the language server protocol and TypeScript plugins then you can omit these types *entirely*! Svelte's IDE tooling will insert the correct types for you, so you'll get type checking without writing them yourself. It also works with our command line tool `svelte-check`.

You can read more about omitting `$types` in our blog post about it.

Other files

Any other files inside a route directory are ignored by SvelteKit. This means you can colocate components and utility modules with the routes that need them.

If components and modules are needed by multiple routes, it's a good idea to put them in `$lib`.

Further reading

- Tutorial: Routing
- Tutorial: API routes
- Docs: Advanced routing

Go to TOC

Loading data

Before a `+page.svelte` component (and its containing `+layout.svelte` components) can be rendered, we often need to get some data. This is done by defining `load` functions.

Page data

A `+page.svelte` file can have a sibling `+page.js` that exports a `load` function, the return value of which is available to the page via the `data` prop:

```
/// file: src/routes/blog/[slug]/+page.js
/** @type {import('./$types').PageLoad} */
export function load({ params }) {
    return {
        post: {
            # `Title for ${params.slug} goes here`,
            content: `Content for ${params.slug} goes here`
        }
    };
}
```

```
<! file: src/routes/blog/[slug]/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;
</script>

<h1>{data.post.title}</h1>
<div>{@html data.post.content}</div>
```

Thanks to the generated `$types` module, we get full type safety.

A `load` function in a `+page.js` file runs both on the server and in the browser (unless combined with `export const ssr = false`, in which case it will only run in the browser). If your `load` function should *always* run on the server (because it uses private environment variables, for example, or accesses a database) then it would go in a `+page.server.js` instead.

A more realistic version of your blog post's `load` function, that only runs on the server and pulls data from a database, might look like this:

```
/// file: src/routes/blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getPost(slug: string): Promise<{ # string, content: string }>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').PageServerLoad} */
export async function load({ params }) {
```

```
    return {
        post: await db.getPost(params.slug)
    };
}
```

Notice that the type changed from `PageLoad` to `PageServerLoad`, because server `load` functions can access additional arguments. To understand when to use `+page.js` and when to use `+page.server.js`, see Universal vs server.

Layout data

Your `+layout.svelte` files can also load data, via `+layout.js` or `+layout.server.js`.

```
/// file: src/routes/blog/[slug]/+layout.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getPostSummaries(): Promise<Array<{ # string, slug: string }>>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').LayoutServerLoad} */
export async function load() {
    return {
        posts: await db.getPostSummaries()
    };
}
```

```
<! file: src/routes/blog/[slug]/+layout.svelte --->
<script>
    /** @type {import('./$types').LayoutData} */
    export let data;
</script>

<main>
    <!-- +page.svelte is rendered in this <slot> -->
    <slot />
</main>

<aside>
    <h2>More posts</h2>
    <ul>
        {#each data.posts as post}
            <li>
                <a href="/blog/{post.slug}">
                    {post.title}
                </a>
            </li>
        {/each}
    </ul>
</aside>
```

Data returned from layout `load` functions is available to child `+layout.svelte` components and the `+page.svelte` component as well as the layout that it 'belongs' to.

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
+	import { page } from '$app/stores';

	/** @type {import('./$types').PageData} */
	export let data;

+	// we can access `data.posts` because it's returned from
+	// the parent layout `load` function
+	$: index = data.posts.findIndex(post => post.slug === $page.params.slug);
+	$: next = data.posts[index - 1];
</script>

<h1>{data.post.title}</h1>
<div>{@html data.post.content}</div>

+{#if next}
+	<p>Next post: <a href="/blog/{next.slug}">{next.title}</a></p>
+{/if}
```

If multiple `load` functions return data with the same key, the last one 'wins' — the result of a layout `load` returning `{ a: 1, b: 2 }` and a page `load` returning `{ b: 3, c: 4 }` would be `{ a: 1, b: 3, c: 4 }`.

$page.data

The `+page.svelte` component, and each `+layout.svelte` component above it, has access to its own data plus all the data from its parents.

In some cases, we might need the opposite — a parent layout might need to access page data or data from a child layout. For example, the root layout might want to access a `title` property returned from a `load` function in `+page.js` or `+page.server.js`. This can be done with `$page.data`:

```
<! file: src/routes/+layout.svelte --->
<script>
	import { page } from '$app/stores';
</script>

<svelte:head>
	<title>{$page.data.title}</title>
</svelte:head>
```

Type information for `$page.data` is provided by `App.PageData`.

Universal vs server

As we've seen, there are two types of `load` function:

- `+page.js` and `+layout.js` files export *universal* `load` functions that run both on the server and in the browser

- `+page.server.js` and `+layout.server.js` files export *server* `load` functions that only run server-side

Conceptually, they're the same thing, but there are some important differences to be aware of.

When does which load function run?

Server `load` functions *always* run on the server.

By default, universal `load` functions run on the server during SSR when the user first visits your page. They will then run again during hydration, reusing any responses from fetch requests. All subsequent invocations of universal `load` functions happen in the browser. You can customize the behavior through page options. If you disable server side rendering, you'll get an SPA and universal `load` functions *always* run on the client.

A `load` function is invoked at runtime, unless you prerender the page — in that case, it's invoked at build time.

Input

Both universal and server `load` functions have access to properties describing the request (`params`, `route` and `url`) and various functions (`fetch`, `setHeaders`, `parent` and `depends`). These are described in the following sections.

Server `load` functions are called with a `ServerLoadEvent`, which inherits `clientAddress`, `cookies`, `locals`, `platform` and `request` from `RequestEvent`.

Universal `load` functions are called with a `LoadEvent`, which has a `data` property. If you have `load` functions in both `+page.js` and `+page.server.js` (or `+layout.js` and `+layout.server.js`), the return value of the server `load` function is the `data` property of the universal `load` function's argument.

Output

A universal `load` function can return an object containing any values, including things like custom classes and component constructors.

A server `load` function must return data that can be serialized with devalue — anything that can be represented as JSON plus things like `BigInt`, `Date`, `Map`, `Set` and `RegExp`, or repeated/cyclical references — so that it can be transported over the network. Your data can include promises, in which case it will be streamed to browsers.

When to use which

Server `load` functions are convenient when you need to access data directly from a database or filesystem, or need to use private environment variables.

Universal `load` functions are useful when you need to `fetch` data from an external API and don't need private credentials, since SvelteKit can get the data directly from the API rather than going via your server. They are also useful when you need to return something that can't be serialized, such as a Svelte component constructor.

In rare cases, you might need to use both together — for example, you might need to return an instance of a custom class that was initialised with data from your server.

Using URL data

Often the `load` function depends on the URL in one way or another. For this, the `load` function provides you with `url`, `route` and `params`.

url

An instance of `URL`, containing properties like the `origin`, `hostname`, `pathname` and `searchParams` (which contains the parsed query string as a `URLSearchParams` object). `url.hash` cannot be accessed during `load`, since it is unavailable on the server.

> In some environments this is derived from request headers during server-side rendering. If you're using adapter-node, for example, you may need to configure the adapter in order for the URL to be correct.

route

Contains the name of the current route directory, relative to `src/routes`:

```
/// file: src/routes/a/[b]/[...c]/+page.js
/** @type {import('./$types').PageLoad} */
export function load({ route }) {
    console.log(route.id); // '/a/[b]/[...c]'
}
```

params

`params` is derived from `url.pathname` and `route.id`.

Given a `route.id` of `/a/[b]/[...c]` and a `url.pathname` of `/a/x/y/z`, the `params` object would look like this:

```
{
    "b": "x",
    "c": "y/z"
}
```

Making fetch requests

To get data from an external API or a `+server.js` handler, you can use the provided `fetch` function, which behaves identically to the native `fetch` web API with a few additional features:

- It can be used to make credentialed requests on the server, as it inherits the `cookie` and `authorization` headers for the page request.
- It can make relative requests on the server (ordinarily, `fetch` requires a URL with an origin when used in a server context).
- Internal requests (e.g. for `+server.js` routes) go directly to the handler function when running on the server, without the overhead of an HTTP call.
- During server-side rendering, the response will be captured and inlined into the rendered HTML by hooking into the `text` and `json` methods of the `Response` object. Note that headers will *not* be serialized, unless explicitly included via `filterSerializedResponseHeaders`.
- During hydration, the response will be read from the HTML, guaranteeing consistency and preventing an additional network request - if you received a warning in your browser console when using the browser `fetch` instead of the `load` `fetch`, this is why.

```
/// file: src/routes/items/[id]/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ fetch, params }) {
    const res = await fetch(`/api/items/${params.id}`);
    const item = await res.json();

    return { item };
}
```

Cookies

A server `load` function can get and set `cookies`.

```
/// file: src/routes/+layout.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getUser(sessionid: string | undefined): Promise<{ name:
string, avatar: string }>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').LayoutServerLoad} */
export async function load({ cookies }) {
    const sessionid = cookies.get('sessionid');

    return {
        user: await db.getUser(sessionid)
    };
}
```

Cookies will only be passed through the provided `fetch` function if the target host is the same as the SvelteKit application or a more specific subdomain of it.

For example, if SvelteKit is serving my.domain.com:

- domain.com WILL NOT receive cookies
- my.domain.com WILL receive cookies
- api.domain.dom WILL NOT receive cookies
- sub.my.domain.com WILL receive cookies

Other cookies will not be passed when `credentials: 'include'` is set, because SvelteKit does not know which domain which cookie belongs to (the browser does not pass this information along), so it's not safe to forward any of them. Use the handleFetch hook to work around it.

> When setting cookies, be aware of the `path` property. By default, the `path` of a cookie is the current pathname. If you for example set a cookie at page `admin/user`, the cookie will only be available within the `admin` pages by default. In most cases you likely want to set `path` to `'/'` to make the cookie available throughout your app.

Headers

Both server and universal `load` functions have access to a `setHeaders` function that, when running on the server, can set headers for the response. (When running in the browser, `setHeaders` has no effect.) This is useful if you want the page to be cached, for example:

```
// @errors: 2322 1360
/// file: src/routes/products/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ fetch, setHeaders }) {
    const url = `https://cms.example.com/products.json`;
    const response = await fetch(url);

    // cache the page for the same length of time
    // as the underlying data
    setHeaders({
        age: response.headers.get('age'),
        'cache-control': response.headers.get('cache-control')
    });

    return response.json();
}
```

Setting the same header multiple times (even in separate `load` functions) is an error — you can only set a given header once. You cannot add a `set-cookie` header with `setHeaders` — use `cookies.set(name, value, options)` instead.

Using parent data

Occasionally it's useful for a `load` function to access data from a parent `load` function, which can be done with `await parent()`:

```
/// file: src/routes/+layout.js
/** @type {import('./$types').LayoutLoad} */
export function load() {
    return { a: 1 };
}
```

```
/// file: src/routes/abc/+layout.js
/** @type {import('./$types').LayoutLoad} */
export async function load({ parent }) {
    const { a } = await parent();
    return { b: a + 1 };
}
```

```
/// file: src/routes/abc/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ parent }) {
    const { a, b } = await parent();
    return { c: a + b };
}
```

```
<! file: src/routes/abc/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;
</script>

<!-- renders `1 + 2 = 3` -->
<p>{data.a} + {data.b} = {data.c}</p>
```

> Notice that the `load` function in `+page.js` receives the merged data from both layout `load` functions, not just the immediate parent.

Inside `+page.server.js` and `+layout.server.js`, `parent` returns data from parent `+layout.server.js` files.

In `+page.js` or `+layout.js` it will return data from parent `+layout.js` files. However, a missing `+layout.js` is treated as a `({ data }) => data` function, meaning that it will also return data from parent `+layout.server.js` files that are not 'shadowed' by a `+layout.js` file

Take care not to introduce waterfalls when using `await parent()`. Here, for example, `getData(params)` does not depend on the result of calling `parent()`, so we should call it first to avoid a delayed render.

```
/// file: +page.js
/** @type {import('./$types').PageLoad} */
export async function load({ params, parent }) {
-   const parentData = await parent();
    const data = await getData(params);
```

```
+   const parentData = await parent();

    return {
        ...data
        meta: { ...parentData.meta, ...data.meta }
    };
}
```

Errors

If an error is thrown during `load`, the nearest `+error.svelte` will be rendered. For *expected* errors, use the `error` helper from `@sveltejs/kit` to specify the HTTP status code and an optional message:

```
/// file: src/routes/admin/+layout.server.js
// @filename: ambient.d.ts
declare namespace App {
    interface Locals {
        user?: {
            name: string;
            isAdmin: boolean;
        }
    }
}

// @filename: index.js
// cut---
import { error } from '@sveltejs/kit';

/** @type {import('./$types').LayoutServerLoad} */
export function load({ locals }) {
    if (!locals.user) {
        throw error(401, 'not logged in');
    }

    if (!locals.user.isAdmin) {
        throw error(403, 'not an admin');
    }
}
```

If an *unexpected* error is thrown, SvelteKit will invoke `handleError` and treat it as a 500 Internal Error.

Redirects

To redirect users, use the `redirect` helper from `@sveltejs/kit` to specify the location to which they should be redirected alongside a `3xx` status code.

```
/// file: src/routes/user/+layout.server.js
// @filename: ambient.d.ts
declare namespace App {
    interface Locals {
        user?: {
            name: string;
        }
    }
}

// @filename: index.js
```

```
// cut---
import { redirect } from '@sveltejs/kit';

/** @type {import('./$types').LayoutServerLoad} */
export function load({ locals }) {
    if (!locals.user) {
        throw redirect(307, '/login');
    }
}
```

Make sure you're not catching the thrown redirect, which would prevent SvelteKit from handling it.

In the browser, you can also navigate programmatically outside of a `load` function using `goto` from `$app.navigation`.

Streaming with promises

Promises at the *top level* of the returned object will be awaited, making it easy to return multiple promises without creating a waterfall. When using a server `load`, *nested* promises will be streamed to the browser as they resolve. This is useful if you have slow, non-essential data, since you can start rendering the page before all the data is available:

```
/// file: src/routes/+page.server.js
/** @type {import('./$types').PageServerLoad} */
export function load() {
    return {
        one: Promise.resolve(1),
        two: Promise.resolve(2),
        streamed: {
            three: new Promise((fulfil) => {
                setTimeout(() => {
                    fulfil(3)
                }, 1000);
            })
        }
    };
}
```

This is useful for creating skeleton loading states, for example:

```
<! file: src/routes/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;
</script>

<p>
    one: {data.one}
</p>
<p>
    two: {data.two}
</p>
<p>
    three:
```

```
    {#await data.streamed.three}
        Loading...
    {:then value}
        {value}
    {:catch error}
        {error.message}
    {/await}
</p>
```

On platforms that do not support streaming, such as AWS Lambda, responses will be buffered. This means the page will only render once all promises resolve. If you are using a proxy (e.g. NGINX), make sure it does not buffer responses from the proxied server.

Streaming data will only work when JavaScript is enabled. You should avoid returning nested promises from a universal `load` function if the page is server rendered, as these are *not* streamed — instead, the promise is recreated when the function reruns in the browser.

The headers and status code of a response cannot be changed once the response has started streaming, therefore you cannot `setHeaders` or throw redirects inside a streamed promise.

Parallel loading

When rendering (or navigating to) a page, SvelteKit runs all `load` functions concurrently, avoiding a waterfall of requests. During client-side navigation, the result of calling multiple server `load` functions are grouped into a single response. Once all `load` functions have returned, the page is rendered.

Rerunning load functions

SvelteKit tracks the dependencies of each `load` function to avoid rerunning it unnecessarily during navigation.

For example, given a pair of `load` functions like these...

```
/// file: src/routes/blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getPost(slug: string): Promise<{ # string, content: string }>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').PageServerLoad} */
export async function load({ params }) {
```

```
    return {
        post: await db.getPost(params.slug)
    };
}
```

```
/// file: src/routes/blog/[slug]/+layout.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getPostSummaries(): Promise<Array<{ # string, slug: string }>>
}

// @filename: index.js
// cut---
import * as db from '$lib/server/database';

/** @type {import('./$types').LayoutServerLoad} */
export async function load() {
    return {
        posts: await db.getPostSummaries()
    };
}
```

...the one in `+page.server.js` will rerun if we navigate from `/blog/trying-the-raw-meat-diet` to `/blog/i-regret-my-choices` because `params.slug` has changed. The one in `+layout.server.js` will not, because the data is still valid. In other words, we won't call `db.getPostSummaries()` a second time.

A `load` function that calls `await parent()` will also rerun if a parent `load` function is rerun.

Dependency tracking does not apply *after* the `load` function has returned — for example, accessing `params.x` inside a nested promise will not cause the function to rerun when `params.x` changes. (Don't worry, you'll get a warning in development if you accidentally do this.) Instead, access the parameter in the main body of your `load` function.

Manual invalidation

You can also rerun `load` functions that apply to the current page using `invalidate(url)`, which reruns all `load` functions that depend on `url`, and `invalidateAll()`, which reruns every `load` function. Server load functions will never automatically depend on a fetched `url` to avoid leaking secrets to the client.

A `load` function depends on `url` if it calls `fetch(url)` or `depends(url)`. Note that `url` can be a custom identifier that starts with `[a-z]:`:

```
/// file: src/routes/random-number/+page.js
/** @type {import('./$types').PageLoad} */
export async function load({ fetch, depends }) {
    // load reruns when `invalidate('https://api.example.com/random-number')` is
called...
    const response = await fetch('https://api.example.com/random-number');

    // ...or when `invalidate('app:random')` is called
    depends('app:random');

    return {
```

```
        number: await response.json()
    };
}
```

```
<! file: src/routes/random-number/+page.svelte --->
<script>
    import { invalidate, invalidateAll } from '$app/navigation';

    /** @type {import('./$types').PageData} */
    export let data;

    function rerunLoadFunction() {
        // any of these will cause the `load` function to rerun
        invalidate('app:random');
        invalidate('https://api.example.com/random-number');
        invalidate(url => url.href.includes('random-number'));
        invalidateAll();
    }
</script>

<p>random number: {data.number}</p>
<button on:click={rerunLoadFunction}>Update random number</button>
```

When do load functions rerun?

To summarize, a `load` function will rerun in the following situations:

- It references a property of `params` whose value has changed
- It references a property of `url` (such as `url.pathname` or `url.search`) whose value has changed. Properties in `request.url` are *not* tracked
- It calls `await parent()` and a parent `load` function reran
- It declared a dependency on a specific URL via `fetch` (universal load only) or `depends`, and that URL was marked invalid with `invalidate(url)`
- All active `load` functions were forcibly rerun with `invalidateAll()`

`params` and `url` can change in response to a `<a href="..">` link click, a `<form>` interaction, a `goto` invocation, or a `redirect`.

Note that rerunning a `load` function will update the `data` prop inside the corresponding `+layout.svelte` or `+page.svelte`; it does *not* cause the component to be recreated. As a result, internal state is preserved. If this isn't what you want, you can reset whatever you need to reset inside an `afterNavigate` callback, and/or wrap your component in a `{#key ...}` block.

Further reading

- Tutorial: Loading data
- Tutorial: Errors and redirects
- Tutorial: Advanced loading

Go to TOC

Form actions

A `+page.server.js` file can export *actions*, which allow you to `POST` data to the server using the `<form>` element.

When using `<form>`, client-side JavaScript is optional, but you can easily *progressively enhance* your form interactions with JavaScript to provide the best user experience.

Default actions

In the simplest case, a page declares a `default` action:

```
/// file: src/routes/login/+page.server.js
/** @type {import('./$types').Actions} */
export const actions = {
    default: async (event) => {
        // TODO log the user in
    }
};
```

To invoke this action from the `/login` page, just add a `<form>` — no JavaScript needed:

```
<! file: src/routes/login/+page.svelte --->
<form method="POST">
    <label>
        Email
        <input name="email" type="email">
    </label>
    <label>
        Password
        <input name="password" type="password">
    </label>
    <button>Log in</button>
</form>
```

If someone were to click the button, the browser would send the form data via `POST` request to the server, running the default action.

> Actions always use `POST` requests, since `GET` requests should never have side-effects.

We can also invoke the action from other pages (for example if there's a login widget in the nav in the root layout) by adding the `action` attribute, pointing to the page:

```
/// file: src/routes/+layout.svelte
<form method="POST" action="/login">
    <!-- content -->
</form>
```

Named actions

Instead of one `default` action, a page can have as many named actions as it needs:

```
/// file: src/routes/login/+page.server.js
/** @type {import('./$types').Actions} */
export const actions = {
-   default: async (event) => {
+   login: async (event) => {
        // TODO log the user in
    },
+   register: async (event) => {
+       // TODO register the user
+   }
};
```

To invoke a named action, add a query parameter with the name prefixed by a `/` character:

```
<! file: src/routes/login/+page.svelte --->
<form method="POST" action="?/register">
```

```
<! file: src/routes/+layout.svelte --->
<form method="POST" action="/login?/register">
```

As well as the `action` attribute, we can use the `formaction` attribute on a button to `POST` the same form data to a different action than the parent `<form>`:

```
/// file: src/routes/login/+page.svelte
-<form method="POST">
+<form method="POST" action="?/login">
    <label>
        Email
        <input name="email" type="email">
    </label>
    <label>
        Password
        <input name="password" type="password">
    </label>
    <button>Log in</button>
+   <button formaction="?/register">Register</button>
</form>
```

> We can't have default actions next to named actions, because if you POST to a named action without a redirect, the query parameter is persisted in the URL, which means the next default POST would go through the named action from before.

Anatomy of an action

Each action receives a `RequestEvent` object, allowing you to read the data with `request.formData()`. After processing the request (for example, logging the user in by setting a cookie), the action can respond with data that will be available through the `form` property on the corresponding page and through `$page.form` app-wide until the next update.

```
// @errors: 2304
/// file: src/routes/login/+page.server.js
/** @type {import('./$types').PageServerLoad} */
export async function load({ cookies }) {
    const user = await db.getUserFromSession(cookies.get('sessionid'));
    return { user };
}

/** @type {import('./$types').Actions} */
export const actions = {
    login: async ({ cookies, request }) => {
        const data = await request.formData();
        const email = data.get('email');
        const password = data.get('password');

        const user = await db.getUser(email);
        cookies.set('sessionid', await db.createSession(user));

        return { success: true };
    },
    register: async (event) => {
        // TODO register the user
    }
};
```

```
<! file: src/routes/login/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;

    /** @type {import('./$types').ActionData} */
    export let form;
</script>

{#if form?.success}
    <!-- this message is ephemeral; it exists because the page was rendered in
           response to a form submission. it will vanish if the user reloads -->
    <p>Successfully logged in! Welcome back, {data.user.name}</p>
{/if}
```

Validation errors

If the request couldn't be processed because of invalid data, you can return validation errors — along with the previously submitted form values — back to the user so that they can try again. The `fail` function lets you return an HTTP status code (typically 400 or 422, in the case of validation errors) along with the data. The status code is available through `$page.status` and the data through `form`:

```
/// file: src/routes/login/+page.server.js
+import { fail } from '@sveltejs/kit';

/** @type {import('./$types').Actions} */
export const actions = {
    login: async ({ cookies, request }) => {
        const data = await request.formData();
        const email = data.get('email');
        const password = data.get('password');

+       if (!email) {
+           return fail(400, { email, missing: true });
+       }

        const user = await db.getUser(email);

+       if (!user || user.password !== hash(password)) {
+           return fail(400, { email, incorrect: true });
+       }

        cookies.set('sessionid', await db.createSession(user));

        return { success: true };
    },
    register: async (event) => {
        // TODO register the user
    }
};
```

Note that as a precaution, we only return the email back to the page — not the password.

```
/// file: src/routes/login/+page.svelte
<form method="POST" action="?/login">
+   {#if form?.missing}<p class="error">The email field is required</p>{/if}
+   {#if form?.incorrect}<p class="error">Invalid credentials!</p>{/if}
    <label>
        Email
-       <input name="email" type="email">
+       <input name="email" type="email" value={form?.email ?? ''}>
    </label>
    <label>
        Password
        <input name="password" type="password">
    </label>
    <button>Log in</button>
    <button formaction="?/register">Register</button>
</form>
```

The returned data must be serializable as JSON. Beyond that, the structure is entirely up to you. For example, if you had multiple forms on the page, you could distinguish which `<form>` the returned `form` data referred to with an `id` property or similar.

Redirects

Redirects (and errors) work exactly the same as in `load`:

```
/// file: src/routes/login/+page.server.js
+import { fail, redirect } from '@sveltejs/kit';

/** @type {import('./$types').Actions} */
export const actions = {
+   login: async ({ cookies, request, url }) => {
        const data = await request.formData();
        const email = data.get('email');
        const password = data.get('password');

        const user = await db.getUser(email);
        if (!user) {
            return fail(400, { email, missing: true });
        }

        if (user.password !== hash(password)) {
            return fail(400, { email, incorrect: true });
        }

        cookies.set('sessionid', await db.createSession(user));

+       if (url.searchParams.has('redirectTo')) {
+           throw redirect(303, url.searchParams.get('redirectTo'));
+       }

        return { success: true };
    },
    register: async (event) => {
        // TODO register the user
    }
};
```

Loading data

After an action runs, the page will be re-rendered (unless a redirect or an unexpected error occurs), with the action's return value available to the page as the `form` prop. This means that your page's `load` functions will run after the action completes.

Note that `handle` runs before the action is invoked, and does not rerun before the `load` functions. This means that if, for example, you use `handle` to populate `event.locals` based on a cookie, you must update `event.locals` when you set or delete the cookie in an action:

```
/// file: src/hooks.server.js
// @filename: ambient.d.ts
declare namespace App {
    interface Locals {
        user: {
            name: string;
        } | null
    }
}

// @filename: global.d.ts
declare global {
    function getUser(sessionid: string | undefined): {
        name: string;
    };
}
```

```
export {};

// @filename: index.js
// cut---
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    event.locals.user = await getUser(event.cookies.get('sessionid'));
    return resolve(event);
}
```

```
/// file: src/routes/account/+page.server.js
// @filename: ambient.d.ts
declare namespace App {
    interface Locals {
        user: {
            name: string;
        } | null
    }
}

// @filename: index.js
// cut---
/** @type {import('./$types').PageServerLoad} */
export function load(event) {
    return {
        user: event.locals.user
    };
}

/** @type {import('./$types').Actions} */
export const actions = {
    logout: async (event) => {
        event.cookies.delete('sessionid');
        event.locals.user = null;
    }
};
```

Progressive enhancement

In the preceding sections we built a `/login` action that works without client-side JavaScript — not a `fetch` in sight. That's great, but when JavaScript *is* available we can progressively enhance our form interactions to provide a better user experience.

use:enhance

The easiest way to progressively enhance a form is to add the `use:enhance` action:

```
/// file: src/routes/login/+page.svelte
<script>
+   import { enhance } from '$app/forms';

    /** @type {import('./$types').ActionData} */
    export let form;
</script>

+<form method="POST" use:enhance>
```

> Yes, it's a little confusing that the `enhance` action and `<form action>` are both called 'action'. These docs are action-packed. Sorry.

Without an argument, `use:enhance` will emulate the browser-native behaviour, just without the full-page reloads. It will:

- update the `form` property, `$page.form` and `$page.status` on a successful or invalid response, but only if the action is on the same page you're submitting from. So for example if your form looks like `<form action="/somewhere/else" ..>`, `form` and `$page` will *not* be updated. This is because in the native form submission case you would be redirected to the page the action is on. If you want to have them updated either way, use `applyAction`
- reset the `<form>` element and invalidate all data using `invalidateAll` on a successful response
- call `goto` on a redirect response
- render the nearest `+error` boundary if an error occurs
- reset focus to the appropriate element

To customise the behaviour, you can provide a `SubmitFunction` that runs immediately before the form is submitted, and (optionally) returns a callback that runs with the `ActionResult`. Note that if you return a callback, the default behavior mentioned above is not triggered. To get it back, call `update`.

```
<form
    method="POST"
    use:enhance={({ formElement, formData, action, cancel, submitter }) => {
        // `formElement` is this `<form>` element
        // `formData` is its `FormData` object that's about to be submitted
        // `action` is the URL to which the form is posted
        // calling `cancel()` will prevent the submission
        // `submitter` is the `HTMLElement` that caused the form to be submitted

        return async ({ result, update }) => {
            // `result` is an `ActionResult` object
            // `update` is a function which triggers the default logic that would
be triggered if this callback wasn't set
        };
    }}
>
```

You can use these functions to show and hide loading UI, and so on.

applyAction

If you provide your own callbacks, you may need to reproduce part of the default `use:enhance` behaviour, such as showing the nearest `+error` boundary. Most of the time, calling `update` passed to the callback is enough. If you need more customization you can do so with `applyAction`:

```
/// file: src/routes/login/+page.svelte
<script>
+   import { enhance, applyAction } from '$app/forms';
```

```
    /** @type {import('./$types').ActionData} */
    export let form;
</script>

<form
    method="POST"
    use:enhance={({ formElement, formData, action, cancel }) => {

        return async ({ result }) => {
            // `result` is an `ActionResult` object
+           if (result.type === 'error') {
+               await applyAction(result);
+           }
        };
    }}
>
```

The behaviour of `applyAction(result)` depends on `result.type`:

- `success`, `failure` — sets `$page.status` to `result.status` and updates `form` and `$page.form` to `result.data` (regardless of where you are submitting from, in contrast to `update` from `enhance`)
- `redirect` — calls `goto(result.location)`
- `error` — renders the nearest `+error` boundary with `result.error`

In all cases, focus will be reset.

Custom event listener

We can also implement progressive enhancement ourselves, without `use:enhance`, with a normal event listener on the `<form>`:

```
<! file: src/routes/login/+page.svelte --->
<script>
    import { invalidateAll, goto } from '$app/navigation';
    import { applyAction, deserialize } from '$app/forms';

    /** @type {import('./$types').ActionData} */
    export let form;

    /** @type {any} */
    let error;

    async function handleSubmit(event) {
        const data = new FormData(this);

        const response = await fetch(this.action, {
            method: 'POST',
            body: data
        });

        /** @type {import('@sveltejs/kit').ActionResult} */
        const result = deserialize(await response.text());

        if (result.type === 'success') {
            // rerun all `load` functions, following the successful update
            await invalidateAll();
        }
```

```
        applyAction(result);
    }
</script>

<form method="POST" on:submit|preventDefault={handleSubmit}>
    <!-- content -->
</form>
```

Note that you need to `deserialize` the response before processing it further using the corresponding method from `$app/forms`. `JSON.parse()` isn't enough because form actions - like `load` functions - also support returning `Date` or `BigInt` objects.

If you have a `+server.js` alongside your `+page.server.js`, `fetch` requests will be routed there by default. To `POST` to an action in `+page.server.js` instead, use the custom `x-sveltekit-action` header:

```
const response = await fetch(this.action, {
    method: 'POST',
    body: data,
+   headers: {
+       'x-sveltekit-action': 'true'
+   }
});
```

Alternatives

Form actions are the preferred way to send data to the server, since they can be progressively enhanced, but you can also use `+server.js` files to expose (for example) a JSON API. Here's how such an interaction could look like:

```
<! file: send-message/+page.svelte --->
<script>
    function rerun() {
        fetch('/api/ci', {
            method: 'POST'
        });
    }
</script>

<button on:click={rerun}>Rerun CI</button>
```

```
// @errors: 2355 1360 2322
/// file: api/ci/+server.js

/** @type {import('./$types').RequestHandler} */
export function POST() {
    // do something
}
```

GET vs POST

As we've seen, to invoke a form action you must use `method="POST"`.

Some forms don't need to `POST` data to the server — search inputs, for example. For these you can use `method="GET"` (or, equivalently, no `method` at all), and SvelteKit will treat them like `<a>` elements, using the client-side router instead of a full page navigation:

```
<form action="/search">
    <label>
        Search
        <input name="q">
    </label>
</form>
```

Submitting this form will navigate to `/search?q=...` and invoke your load function but will not invoke an action. As with `<a>` elements, you can set the `data-sveltekit-reload`, `data-sveltekit-replaces-tate`, `data-sveltekit-keepfocus` and `data-sveltekit-noscroll` attributes on the `<form>` to control the router's behaviour.

Further reading

- Tutorial: Forms

Go to TOC

Page options

By default, SvelteKit will render (or prerender) any component first on the server and send it to the client as HTML. It will then render the component again in the browser to make it interactive in a process called **hydration**. For this reason, you need to ensure that components can run in both places. SvelteKit will then initialize a **router** that takes over subsequent navigations.

You can control each of these on a page-by-page basis by exporting options from `+page.js` or `+page.server.js`, or for groups of pages using a shared `+layout.js` or `+layout.server.js`. To define an option for the whole app, export it from the root layout. Child layouts and pages override values set in parent layouts, so — for example — you can enable prerendering for your entire app then disable it for pages that need to be dynamically rendered.

You can mix and match these options in different areas of your app. For example you could prerender your marketing page for maximum speed, server-render your dynamic pages for SEO and accessibility and turn your admin section into an SPA by rendering it on the client only. This makes SvelteKit very versatile.

prerender

It's likely that at least some routes of your app can be represented as a simple HTML file generated at build time. These routes can be *prerendered*.

```
/// file: +page.js/+page.server.js/+server.js
export const prerender = true;
```

Alternatively, you can set `export const prerender = true` in your root `+layout.js` or `+layout.server.js` and prerender everything except pages that are explicitly marked as *not* prerenderable:

```
/// file: +page.js/+page.server.js/+server.js
export const prerender = false;
```

Routes with `prerender = true` will be excluded from manifests used for dynamic SSR, making your server (or serverless/edge functions) smaller. In some cases you might want to prerender a route but also include it in the manifest (for example, with a route like `/blog/[slug]` where you want to prerender your most recent/popular content but server-render the long tail) — for these cases, there's a third option, 'auto':

```
/// file: +page.js/+page.server.js/+server.js
export const prerender = 'auto';
```

If your entire app is suitable for prerendering, you can use `adapter-static`, which will output files suitable for use with any static webserver.

The prerenderer will start at the root of your app and generate files for any prerenderable pages or `+server.js` routes it finds. Each page is scanned for `<a>` elements that point to other pages that are candidates for prerendering — because of this, you generally don't need to specify which pages should be accessed. If you *do* need to specify which pages should be accessed by the prerenderer, you can do so with `config.kit.prerender.entries`, or by exporting an `entries` function from your dynamic route.

While prerendering, the value of `building` imported from `$app/environment` will be `true`.

Prerendering server routes

Unlike the other page options, `prerender` also applies to `+server.js` files. These files are *not* affected from layouts, but will inherit default values from the pages that fetch data from them, if any. For example if a `+page.js` contains this `load` function...

```
/// file: +page.js
export const prerender = true;

/** @type {import('./$types').PageLoad} */
export async function load({ fetch }) {
    const res = await fetch('/my-server-route.json');
    return await res.json();
}
```

...then `src/routes/my-server-route.json/+server.js` will be treated as prerenderable if it doesn't contain its own `export const prerender = false`.

When not to prerender

The basic rule is this: for a page to be prerenderable, any two users hitting it directly must get the same content from the server.

> Not all pages are suitable for prerendering. Any content that is prerendered will be seen by all users. You can of course fetch personalized data in `onMount` in a prerendered page, but this may result in a poorer user experience since it will involve blank initial content or loading indicators.

Note that you can still prerender pages that load data based on the page's parameters, such as a `src/routes/blog/[slug]/+page.svelte` route.

Accessing `url.searchParams` during prerendering is forbidden. If you need to use it, ensure you are only doing so in the browser (for example in `onMount`).

Pages with actions cannot be prerendered, because a server must be able to handle the action `POST` requests.

Prerender and ssr

If you set the ssr option to `false`, each request will result in the same empty HTML shell. Since this would result in unnecessary work, SvelteKit defaults to prerendering any pages it finds where `prerender` is not explicitly set to `false`.

Route conflicts

Because prerendering writes to the filesystem, it isn't possible to have two endpoints that would cause a directory and a file to have the same name. For example, `src/routes/foo/+server.js` and `src/routes/foo/bar/+server.js` would try to create `foo` and `foo/bar`, which is impossible.

For that reason among others, it's recommended that you always include a file extension — `src/routes/foo.json/+server.js` and `src/routes/foo/bar.json/+server.js` would result in `foo.json` and `foo/bar.json` files living harmoniously side-by-side.

For *pages*, we skirt around this problem by writing `foo/index.html` instead of `foo`.

Troubleshooting

If you encounter an error like 'The following routes were marked as prerenderable, but were not prerendered' it's because the route in question (or a parent layout, if it's a page) has `export const prerender = true` but the page wasn't actually prerendered, because it wasn't reached by the prerendering crawler.

Since these routes cannot be dynamically server-rendered, this will cause errors when people try to access the route in question. There are two ways to fix it:

- Ensure that SvelteKit can find the route by following links from `config.kit.prerender.entries` or the `entries` page option. Add links to dynamic routes (i.e. pages with `[parameters]`) to this option if they are not found through crawling the other entry points, else they are not prerendered because SvelteKit doesn't know what value the parameters should have. Pages not marked as prerenderable will be ignored and their links to other pages will not be crawled, even if some of them would be prerenderable.
- Change `export const prerender = true` to `export const prerender = 'auto'`. Routes with `'auto'` can be dynamically server rendered

entries

SvelteKit will discover pages to prerender automatically, by starting at *entry points* and crawling them. By default, all your non-dynamic routes are considered entry points — for example, if you have these routes...

```
/             # non-dynamic
/blog         # non-dynamic
/blog/[slug]  # dynamic, because of `[slug]`
```

...SvelteKit will prerender `/` and `/blog`, and in the process discover links like `<a href="/blog/hello-world">` which give it new pages to prerender.

Most of the time, that's enough. In some situations, links to pages like `/blog/hello-world` might not exist (or might not exist on prerendered pages), in which case we need to tell SvelteKit about their existence.

This can be done with `config.kit.prerender.entries`, or by exporting an `entries` function from a `+page.js`, a `+page.server.js` or a `+server.js` belonging to a dynamic route:

```
/// file: src/routes/blog/[slug]/+page.server.js
/** @type {import('./$types').EntryGenerator} */
export function entries() {
    return [
        { slug: 'hello-world' },
        { slug: 'another-blog-post' }
    ];
}

export const prerender = true;
```

`entries` can be an `async` function, allowing you to (for example) retrieve a list of posts from a CMS or database, in the example above.

ssr

Normally, SvelteKit renders your page on the server first and sends that HTML to the client where it's hydrated. If you set `ssr` to `false`, it renders an empty 'shell' page instead. This is useful if your page is unable to be rendered on the server (because you use browser-only globals like `document` for example), but in most situations it's not recommended (see appendix).

```
/// file: +page.js
export const ssr = false;
```

If you add `export const ssr = false` to your root `+layout.js`, your entire app will only be rendered on the client — which essentially means you turn your app into an SPA.

csr

Ordinarily, SvelteKit hydrates your server-rendered HTML into an interactive client-side-rendered (CSR) page. Some pages don't require JavaScript at all — many blog posts and 'about' pages fall into this category. In these cases you can disable CSR:

```
/// file: +page.js
export const csr = false;
```

> If both `ssr` and `csr` are `false`, nothing will be rendered!

trailingSlash

By default, SvelteKit will remove trailing slashes from URLs — if you visit `/about/`, it will respond with a redirect to `/about`. You can change this behaviour with the `trailingSlash` option, which can be one of `'never'` (the default), `'always'`, or `'ignore'`.

As with other page options, you can export this value from a `+layout.js` or a `+layout.server.js` and it will apply to all child pages. You can also export the configuration from `+server.js` files.

```
/// file: src/routes/+layout.js
export const trailingSlash = 'always';
```

This option also affects prerendering. If `trailingSlash` is `always`, a route like `/about` will result in an `about/index.html` file, otherwise it will create `about.html`, mirroring static webserver conventions.

> Ignoring trailing slashes is not recommended — the semantics of relative paths differ between the two cases (`./y` from `/x` is `/y`, but from `/x/` is `/x/y`), and `/x` and `/x/` are treated as separate URLs which is harmful to SEO.

config

With the concept of adapters, SvelteKit is able to run on a variety of platforms. Each of these might have specific configuration to further tweak the deployment — for example on Vercel you could choose to deploy some parts of your app on the edge and others on serverless environments.

`config` is an object with key-value pairs at the top level. Beyond that, the concrete shape is dependent on the adapter you're using. Every adapter should provide a `Config` interface to import for type safety. Consult the documentation of your adapter for more information.

```
// @filename: ambient.d.ts
declare module 'some-adapter' {
    export interface Config { runtime: string }
}

// @filename: index.js
// cut---
/// file: src/routes/+page.js
/** @type {import('some-adapter').Config} */
export const config = {
    runtime: 'edge'
};
```

`config` objects are merged at the top level (but *not* deeper levels). This means you don't need to repeat all the values in a `+page.js` if you want to only override some of the values in the upper `+layout.js`. For example this layout configuration...

```
/// file: src/routes/+layout.js
export const config = {
    runtime: 'edge',
    regions: 'all',
    foo: {
        bar: true
    }
}
```

...is overridden by this page configuration...

```
/// file: src/routes/+page.js
export const config = {
    regions: ['us1', 'us2'],
    foo: {
        baz: true
    }
}
```

...which results in the config value `{ runtime: 'edge', regions: ['us1', 'us2'], foo: { baz: true } }` for that page.

Further reading

- Tutorial: Page options

Go to TOC

State management

If you're used to building client-only apps, state management in an app that spans server and client might seem intimidating. This section provides tips for avoiding some common gotchas.

Avoid shared state on the server

Browsers are *stateful* — state is stored in memory as the user interacts with the application. Servers, on the other hand, are *stateless* — the content of the response is determined entirely by the content of the request.

Conceptually, that is. In reality, servers are often long-lived and shared by multiple users. For that reason it's important not to store data in shared variables. For example, consider this code:

```
// @errors: 7034 7005
/// file: +page.server.js
let user;

/** @type {import('./$types').PageServerLoad} */
export function load() {
    return { user };
}

/** @type {import('./$types').Actions} */
export const actions = {
    default: async ({ request }) => {
        const data = await request.formData();

        // NEVER DO THIS!
        user = {
            name: data.get('name'),
            embarrassingSecret: data.get('secret')
        };
    }
}
```

The `user` variable is shared by everyone who connects to this server. If Alice submitted an embarrassing secret, and Bob visited the page after her, Bob would know Alice's secret. In addition, when Alice returns to the site later in the day, the server may have restarted, losing her data.

Instead, you should *authenticate* the user using `cookies` and persist the data to a database.

No side-effects in load

For the same reason, your `load` functions should be *pure* — no side-effects (except maybe the occasional `console.log(...)`). For example, you might be tempted to write to a store inside a `load` function so that you can use the store value in your components:

```
/// file: +page.js
// @filename: ambient.d.ts
declare module '$lib/user' {
```

```
    export const user: { set: (value: any) => void };
}

// @filename: index.js
// cut---
import { user } from '$lib/user';

/** @type {import('./$types').PageLoad} */
export async function load({ fetch }) {
    const response = await fetch('/api/user');

    // NEVER DO THIS!
    user.set(await response.json());
}
```

As with the previous example, this puts one user's information in a place that is shared by *all* users. Instead, just return the data...

```
/// file: +page.js
export async function load({ fetch }) {
    const response = await fetch('/api/user');

+   return {
+       user: await response.json()
+   };
}
```

...and pass it around to the components that need it, or use `$page.data`.

If you're not using SSR, then there's no risk of accidentally exposing one user's data to another. But you should still avoid side-effects in your `load` functions — your application will be much easier to reason about without them.

Using stores with context

You might wonder how we're able to use `$page.data` and other app stores if we can't use our own stores. The answer is that app stores on the server use Svelte's context API — the store is attached to the component tree with `setContext`, and when you subscribe you retrieve it with `getContext`. We can do the same thing with our own stores:

```
<! file: src/routes/+layout.svelte --->
<script>
    import { setContext } from 'svelte';
    import { writable } from 'svelte/store';

    /** @type {import('./$types').LayoutData} */
    export let data;

    // Create a store and update it when necessary...
    const user = writable();
    $: user.set(data.user);

    // ...and add it to the context for child components to access
    setContext('user', user);
</script>
```

```
<! file: src/routes/user/+page.svelte --->
<script>
    import { getContext } from 'svelte';

    // Retrieve user store from context
    const user = getContext('user');
</script>

<p>Welcome {$user.name}</p>
```

Updating the context-based store value in deeper-level pages or components will not affect the value in the parent component when the page is rendered via SSR: The parent component has already been rendered by the time the store value is updated. To avoid values 'flashing' during state updates during hydration, it is generally recommended to pass state down into components rather than up.

If you're not using SSR (and can guarantee that you won't need to use SSR in future) then you can safely keep state in a shared module, without using the context API.

Component state is preserved

When you navigate around your application, SvelteKit reuses existing layout and page components. For example, if you have a route like this...

```
<! file: src/routes/blog/[slug]/+page.svelte --->
<script>
    /** @type {import('./$types').PageData} */
    export let data;

    // THIS CODE IS BUGGY!
    const wordCount = data.content.split(' ').length;
    const estimatedReadingTime = wordCount / 250;
</script>

<header>
    <h1>{data.title}</h1>
    <p>Reading time: {Math.round(estimatedReadingTime)} minutes</p>
</header>

<div>{@html data.content}</div>
```

...then navigating from `/blog/my-short-post` to `/blog/my-long-post` won't cause the component to be destroyed and recreated. The `data` prop (and by extension `data.title` and `data.content`) will change, but because the code isn't rerunning, `estimatedReadingTime` won't be recalculated.

Instead, we need to make the value *reactive*:

```
/// file: src/routes/blog/[slug]/+page.svelte
<script>
    /** @type {import('./$types').PageData} */
    export let data;

+   $: wordCount = data.content.split(' ').length;
+   $: estimatedReadingTime = wordCount / 250;
</script>
```

Reusing components like this means that things like sidebar scroll state are preserved, and you can easily animate between changing values. However, if you do need to completely destroy and remount a component on navigation, you can use this pattern:

```
{#key $page.url.pathname}
    <BlogPost title={data.title} content={data.title} />
{/key}
```

Storing state in the URL

If you have state that should survive a reload and/or affect SSR, such as filters or sorting rules on a table, URL search parameters (like `?sort=price&order=ascending`) are a good place to put them. You can put them in `<a href="...">` or `<form action="...">` attributes, or set them programmatically via `goto('?key=value')`. They can be accessed inside `load` functions via the `url` parameter, and inside components via `$page.url.searchParams`.

Storing ephemeral state in snapshots

Some UI state, such as 'is the accordion open?', is disposable — if the user navigates away or refreshes the page, it doesn't matter if the state is lost. In some cases, you *do* want the data to persist if the user navigates to a different page and comes back, but storing the state in the URL or in a database would be overkill. For this, SvelteKit provides snapshots, which let you associate component state with a history entry.

Go to TOC

Building your app

Building a SvelteKit app happens in two stages, which both happen when you run `vite build` (usually via `npm run build`).

Firstly, Vite creates an optimized production build of your server code, your browser code, and your service worker (if you have one). Prerendering is executed at this stage, if appropriate.

Secondly, an *adapter* takes this production build and tunes it for your target environment — more on this on the following pages.

During the build

SvelteKit will load your `+page/layout(.server).js` files (and all files they import) for analysis during the build. Any code that should *not* be executed at this stage must check that `building` from `$app/environment` is `false`:

```
+import { building } from '$app/environment';
import { setupMyDatabase } from '$lib/server/database';

+if (!building) {
    setupMyDatabase();
+}

export function load() {
    // ...
}
```

Preview your app

After building, you can view your production build locally with `vite preview` (via `npm run preview`). Note that this will run the app in Node, and so is not a perfect reproduction of your deployed app — adapter-specific adjustments like the `platform` object do not apply to previews.

Go to TOC

Adapters

Before you can deploy your SvelteKit app, you need to *adapt* it for your deployment target. Adapters are small plugins that take the built app as input and generate output for deployment.

Official adapters exist for a variety of platforms — these are documented on the following pages:

- `@sveltejs/adapter-cloudflare` for Cloudflare Pages
- `@sveltejs/adapter-cloudflare-workers` for Cloudflare Workers
- `@sveltejs/adapter-netlify` for Netlify
- `@sveltejs/adapter-node` for Node servers
- `@sveltejs/adapter-static` for static site generation (SSG)
- `@sveltejs/adapter-vercel` for Vercel

Additional community-provided adapters exist for other platforms.

Using adapters

Your adapter is specified in `svelte.config.js`:

```
/// file: svelte.config.js
// @filename: ambient.d.ts
declare module 'svelte-adapter-foo' {
    const adapter: (opts: any) => import('@sveltejs/kit').Adapter;
    export default adapter;
}

// @filename: index.js
// cut---
import adapter from 'svelte-adapter-foo';

/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        adapter: adapter({
            // adapter options go here
        })
    }
};

export default config;
```

Platform-specific context

Some adapters may have access to additional information about the request. For example, Cloudflare Workers can access an `env` object containing KV namespaces etc. This can be passed to the `RequestEvent` used in hooks and server routes as the `platform` property — consult each adapter's documentation to learn more.

Go to TOC

Zero-config deployments

When you create a new SvelteKit project with `npm create svelte@latest`, it installs `adapter-auto` by default. This adapter automatically installs and uses the correct adapter for supported environments when you deploy:

- `@sveltejs/adapter-cloudflare` for Cloudflare Pages
- `@sveltejs/adapter-netlify` for Netlify
- `@sveltejs/adapter-vercel` for Vercel
- `svelte-adapter-azure-swa` for Azure Static Web Apps
- `svelte-kit-sst` for AWS via SST

It's recommended to install the appropriate adapter to your `devDependencies` once you've settled on a target environment, since this will add the adapter to your lockfile and slightly improve install times on CI.

Environment-specific configuration

To add configuration options, such as `{ edge: true }` in `adapter-vercel` and `adapter-netlify`, you must install the underlying adapter — `adapter-auto` does not take any options.

Adding community adapters

You can add zero-config support for additional adapters by editing adapters.js and opening a pull request.

Go to TOC

Node servers

To generate a standalone Node server, use `adapter-node`.

Usage

Install with `npm i -D @sveltejs/adapter-node`, then add the adapter to your `svelte.config.js`:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-node';

export default {
    kit: {
        adapter: adapter()
    }
};
```

Deploying

First, build your app with `npm run build`. This will create the production server in the output directory specified in the adapter options, defaulting to `build`.

You will need the output directory, the project's `package.json`, and the production dependencies in `node_modules` to run the application. Production dependencies can be generated by copying the `package.json` and `package-lock.json` and then running `npm ci --omit dev` (you can skip this step if your app doesn't have any dependencies). You can then start your app with this command:

```
node build
```

Development dependencies will be bundled into your app using Rollup. To control whether a given package is bundled or externalised, place it in `devDependencies` or `dependencies` respectively in your `package.json`.

Environment variables

In `dev` and `preview`, SvelteKit will read environment variables from your `.env` file (or `.env.local`, or `.env.[mode]`, as determined by Vite.)

In production, `.env` files are *not* automatically loaded. To do so, install `dotenv` in your project...

```
npm install dotenv
```

...and invoke it before running the built app:

```
-node build
+node -r dotenv/config build
```

`PORT`, `HOST` and `SOCKET_PATH`

By default, the server will accept connections on `0.0.0.0` using port 3000. These can be customised with the `PORT` and `HOST` environment variables:

```
HOST=127.0.0.1 PORT=4000 node build
```

Alternatively, the server can be configured to accept connections on a specified socket path. When this is done using the `SOCKET_PATH` environment variable, the `HOST` and `PORT` environment variables will be disregarded.

```
SOCKET_PATH=/tmp/socket node build
```

`ORIGIN`, `PROTOCOL_HEADER` and `HOST_HEADER`

HTTP doesn't give SvelteKit a reliable way to know the URL that is currently being requested. The simplest way to tell SvelteKit where the app is being served is to set the `ORIGIN` environment variable:

```
ORIGIN=https://my.site node build

# or e.g. for local previewing and testing
ORIGIN=http://localhost:3000 node build
```

With this, a request for the `/stuff` pathname will correctly resolve to `https://my.site/stuff`. Alternatively, you can specify headers that tell SvelteKit about the request protocol and host, from which it can construct the origin URL:

```
PROTOCOL_HEADER=x-forwarded-proto HOST_HEADER=x-forwarded-host node build
```

> `x-forwarded-proto` and `x-forwarded-host` are de facto standard headers that forward the original protocol and host if you're using a reverse proxy (think load balancers and CDNs). You should only set these variables if your server is behind a trusted reverse proxy; otherwise, it'd be possible for clients to spoof these headers.

If `adapter-node` can't correctly determine the URL of your deployment, you may experience this error when using form actions:

> Cross-site POST form submissions are forbidden

`ADDRESS_HEADER` and `XFF_DEPTH`

The RequestEvent object passed to hooks and endpoints includes an `event.getClientAddress()` function that returns the client's IP address. By default this is the connecting `remoteAddress`. If your server is behind one or more proxies (such as a load balancer), this value will contain the innermost proxy's IP address rather than the client's, so we need to specify an `ADDRESS_HEADER` to read the address from:

```
ADDRESS_HEADER=True-Client-IP node build
```

> Headers can easily be spoofed. As with `PROTOCOL_HEADER` and `HOST_HEADER`, you should know what you're doing before setting these.

If the `ADDRESS_HEADER` is `X-Forwarded-For`, the header value will contain a comma-separated list of IP addresses. The `XFF_DEPTH` environment variable should specify how many trusted proxies sit in front of your server. E.g. if there are three trusted proxies, proxy 3 will forward the addresses of the original connection and the first two proxies:

```
<client address>, <proxy 1 address>, <proxy 2 address>
```

Some guides will tell you to read the left-most address, but this leaves you vulnerable to spoofing:

```
<spoofed address>, <client address>, <proxy 1 address>, <proxy 2 address>
```

We instead read from the *right*, accounting for the number of trusted proxies. In this case, we would use `XFF_DEPTH=3`.

> If you need to read the left-most address instead (and don't care about spoofing) — for example, to offer a geolocation service, where it's more important for the IP address to be *real* than *trusted*, you can do so by inspecting the `x-forwarded-for` header within your app.

`BODY_SIZE_LIMIT`

The maximum request body size to accept in bytes including while streaming. Defaults to 512kb. You can disable this option with a value of 0 and implement a custom check in `handle` if you need something more advanced.

Options

The adapter can be configured with various options:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-node';
```

```
export default {
    kit: {
        adapter: adapter({
            // default options are shown
            out: 'build',
            precompress: false,
            envPrefix: '',
            polyfill: true
        })
    }
};
```

out

The directory to build the server to. It defaults to `build` — i.e. `node build` would start the server locally after it has been created.

precompress

Enables precompressing using gzip and brotli for assets and prerendered pages. It defaults to `false`.

envPrefix

If you need to change the name of the environment variables used to configure the deployment (for example, to deconflict with environment variables you don't control), you can specify a prefix:

```
envPrefix: 'MY_CUSTOM_';
```

```
MY_CUSTOM_HOST=127.0.0.1 \
MY_CUSTOM_PORT=4000 \
MY_CUSTOM_ORIGIN=https://my.site \
node build
```

polyfill

Controls whether your build will load polyfills for missing modules. It defaults to `true`, and should only be disabled when using Node 18.11 or greater.

Note: to use Node's built-in `crypto` global with Node 18 you will need to use the `--experimental-global-webcrypto` flag. This flag is not required with Node 20.

Custom server

The adapter creates two files in your build directory — `index.js` and `handler.js`. Running `index.js` — e.g. `node build`, if you use the default build directory — will start a server on the configured port.

Alternatively, you can import the `handler.js` file, which exports a handler suitable for use with Express, Connect or Polka (or even just the built-in `http.createServer`) and set up your own server:

```
// @errors: 2307 7006
/// file: my-server.js
import { handler } from './build/handler.js';
import express from 'express';

const app = express();

// add a route that lives separately from the SvelteKit app
app.get('/healthcheck', (req, res) => {
    res.end('ok');
});

// let SvelteKit handle everything else, including serving prerendered pages and
static assets
app.use(handler);

app.listen(3000, () => {
    console.log('listening on port 3000');
});
```

Troubleshooting

Is there a hook for cleaning up before the server exits?

There's nothing built-in to SvelteKit for this, because such a cleanup hook depends highly on the execution environment you're on. For Node, you can use its built-in `process.on(..)` to implement a callback that runs before the server exits:

```
// @errors: 2304 2580
function shutdownGracefully() {
    // anything you need to clean up manually goes in here
    db.shutdown();
}

process.on('SIGINT', shutdownGracefully);
process.on('SIGTERM', shutdownGracefully);
```

Go to TOC

Static site generation

To use SvelteKit as a static site generator (SSG), use `adapter-static`.

This will prerender your entire site as a collection of static files. If you'd like to prerender only some pages and dynamically server-render others, you will need to use a different adapter together with the `prerender` option.

Usage

Install with `npm i -D @sveltejs/adapter-static`, then add the adapter to your `svelte.config.js`:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-static';

export default {
    kit: {
        adapter: adapter({
            // default options are shown. On some platforms
            // these options are set automatically — see below
            pages: 'build',
            assets: 'build',
            fallback: undefined,
            precompress: false,
            strict: true
        })
    }
};
```

...and add the `prerender` option to your root layout:

```
/// file: src/routes/+layout.js
// This can be false if you're using a fallback (i.e. SPA mode)
export const prerender = true;
```

> You must ensure SvelteKit's `trailingSlash` option is set appropriately for your environment. If your host does not render `/a.html` upon receiving a request for `/a` then you will need to set `trailingSlash: 'always'` in your root layout to create `/a/index.html` instead.

Zero-config support

Some platforms have zero-config support (more to come in future):

- Vercel

On these platforms, you should omit the adapter options so that `adapter-static` can provide the optimal configuration:

```
/// file: svelte.config.js
export default {
    kit: {
-       adapter: adapter({...})
+       adapter: adapter()
    }
};
```

Options

pages

The directory to write prerendered pages to. It defaults to `build`.

assets

The directory to write static assets (the contents of `static`, plus client-side JS and CSS generated by SvelteKit) to. Ordinarily this should be the same as `pages`, and it will default to whatever the value of `pages` is, but in rare circumstances you might need to output pages and assets to separate locations.

fallback

Specify a fallback page for SPA mode, e.g. `index.html` or `200.html` or `404.html`.

precompress

If `true`, precompresses files with brotli and gzip. This will generate `.br` and `.gz` files.

strict

By default, `adapter-static` checks that either all pages and endpoints (if any) of your app were prerendered, or you have the `fallback` option set. This check exists to prevent you from accidentally publishing an app where some parts of it are not accessible, because they are not contained in the final output. If you know this is ok (for example when a certain page only exists conditionally), you can set `strict` to `false` to turn off this check.

GitHub Pages

When building for GitHub Pages, make sure to update `paths.base` to match your repo name, since the site will be served from https://your-username.github.io/your-repo-name rather than from the root.

You will have to prevent GitHub's provided Jekyll from managing your site by putting an empty `.nojekyll` file in your `static` folder.

A config for GitHub Pages might look like the following:

```
// @errors: 2307 2322
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-static';

const dev = process.argv.includes('dev');

/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        adapter: adapter(),
        paths: {
            base: dev ? '' : process.env.BASE_PATH,
        }
    }
};
```

You can use GitHub actions to automatically deploy your site to GitHub Pages when you make a change. Here's an example workflow:

```
/// file: .github/workflows/deploy.yml
name: Deploy to GitHub Pages

on:
  push:
    branches: 'main'

jobs:
  build_site:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      # If you're using pnpm, add this step then change the commands and cache key
below to use `pnpm`
      # - name: Install pnpm
      #   uses: pnpm/action-setup@v2
      #   with:
      #     version: 8

      - name: Install Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 18
          cache: npm

      - name: Install dependencies
        run: npm install

      - name: build
        env:
          BASE_PATH: '/your-repo-name'
        run: |
          npm run build
          touch build/.nojekyll

      - name: Upload Artifacts
        uses: actions/upload-pages-artifact@v1
        with:
          # this should match the `pages` option in your adapter-static options
          path: 'build/'
```

```yaml
  deploy:
    needs: build_site
    runs-on: ubuntu-latest

    permissions:
      pages: write
      id-token: write

    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}

    steps:
      - name: Deploy
        id: deployment
        uses: actions/deploy-pages@v1
```

Go to TOC

Single-page apps

You can turn any SvelteKit app, using any adapter, into a fully client-rendered single-page app (SPA) by disabling SSR at the root layout:

```
/// file: src/routes/+layout.js
export const ssr = false;
```

> In most situations this is not recommended: it harms SEO, tends to slow down perceived performance, and makes your app inaccessible to users if JavaScript fails or is disabled (which happens more often than you probably think).

If you don't have any server-side logic (i.e. `+page.server.js`, `+layout.server.js` or `+server.js` files) you can use `adapter-static` to create your SPA by adding a *fallback page*.

Usage

Install with `npm i -D @sveltejs/adapter-static`, then add the adapter to your `svelte.config.js` with the following options:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-static';

export default {
    kit: {
        adapter: adapter({
            fallback: '200.html' // may differ from host to host
        })
    }
};
```

The `fallback` page is an HTML page created by SvelteKit from your page template (e.g. `app.html`) that loads your app and navigates to the correct route. For example Surge, a static web host, lets you add a `200.html` file that will handle any requests that don't correspond to static assets or prerendered pages.

On some hosts it may be `index.html` or something else entirely — consult your platform's documentation.

Apache

To run an SPA on Apache, you should add a `static/.htaccess` file to route requests to the fallback page:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /
    RewriteRule ^200\.html$ - [L]
    RewriteCond %{REQUEST_FILENAME} !-f
```

```
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule . /200.html [L]
</IfModule>
```

Prerendering individual pages

If you want certain pages to be prerendered, you can re-enable `ssr` alongside `prerender` for just those parts of your app:

```
/// file: src/routes/my-prerendered-page/+page.js
export const prerender = true;
export const ssr = true;
```<span style='float: footnote;'><a href="../../index.html#toc">Go to TOC</a>

Cloudflare Pages

To deploy to Cloudflare Pages, use `adapter-cloudflare`.

This adapter will be installed by default when you use `adapter-auto`. If you plan on staying with Cloudflare Pages you can switch from `adapter-auto` to using this adapter directly so that type declarations will be automatically applied and you can set Cloudflare-specific options.

Comparisons

- `adapter-cloudflare` – supports all SvelteKit features; builds for Cloudflare Pages
- `adapter-cloudflare-workers` – supports all SvelteKit features; builds for Cloudflare Workers
- `adapter-static` – only produces client-side static assets; compatible with Cloudflare Pages

Usage

Install with `npm i -D @sveltejs/adapter-cloudflare`, then add the adapter to your `svelte.config.js`:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-cloudflare';

export default {
    kit: {
        adapter: adapter({
            // See below for an explanation of these options
            routes: {
                include: ['/*'],
                exclude: ['<all>']
            }
        })
    }
};
```

Options

The `routes` option allows you to customise the `_routes.json` file generated by `adapter-cloudflare`.

- `include` defines routes that will invoke a function, and defaults to `['/*']`
- `exclude` defines routes that will *not* invoke a function — this is a faster and cheaper way to serve your app's static assets. This array can include the following special values:
  - `<build>` contains your app's build artifacts (the files generated by Vite)
  - `<files>` contains the contents of your `static` directory
  - `<prerendered>` contains a list of prerendered pages
  - `<all>` (the default) contains all of the above

You can have up to 100 `include` and `exclude` rules combined. Generally you can omit the `routes` options, but if (for example) your `<prerendered>` paths exceed that limit, you may find it helpful to manually create an `exclude` list that includes `'/articles/*'` instead of the auto-generated `['/articles/foo', '/articles/bar', '/articles/baz', ...]`.

Deployment

Please follow the Get Started Guide for Cloudflare Pages to begin.

When configuring your project settings, you must use the following settings:

- **Framework preset** – SvelteKit
- **Build command** – `npm run build` or `vite build`
- **Build output directory** – `.svelte-kit/cloudflare`

Bindings

The `env` object contains your project's bindings, which consist of KV/DO namespaces, etc. It is passed to SvelteKit via the `platform` property, along with `context` and `caches`, meaning that you can access it in hooks and endpoints:

```
// @errors: 7031
export async function POST({ request, platform }) {
    const x = platform.env.YOUR_DURABLE_OBJECT_NAMESPACE.idFromName('x');
}
```

> SvelteKit's built-in `$env` module should be preferred for environment variables.

To make these types available to your app, reference them in your `src/app.d.ts`:

```
/// file: src/app.d.ts
declare global {
    namespace App {
        interface Platform {
+           env?: {
+               YOUR_KV_NAMESPACE: KVNamespace;
+               YOUR_DURABLE_OBJECT_NAMESPACE: DurableObjectNamespace;
+           };
        }
    }
}

export {};
```

Testing Locally

`platform.env` is only available in the final build and not in dev mode. For testing the build, you can use wrangler **version 3**. Once you have built your site, run `wrangler pages dev .svelte-kit/cloudflare`. Ensure you have your bindings in your `wrangler.toml`.

Notes

Functions contained in the `/functions` directory at the project's root will *not* be included in the deployment, which is compiled to a single `_worker.js` file. Functions should be implemented as server endpoints in your SvelteKit app.

The `_headers` and `_redirects` files specific to Cloudflare Pages can be used for static asset responses (like images) by putting them into the `/static` folder.

However, they will have no effect on responses dynamically rendered by SvelteKit, which should return custom headers or redirect responses from server endpoints or with the `handle` hook.

Troubleshooting

Further reading

You may wish to refer to Cloudflare's documentation for deploying a SvelteKit site.

Accessing the file system

You can't access the file system through methods like `fs.readFileSync` in Serverless/Edge environments. If you need to access files that way, do that during building the app through prerendering. If you have a blog for example and don't want to manage your content through a CMS, then you need to prerender the content (or prerender the endpoint from which you get it) and redeploy your blog everytime you add new content.

Go to TOC

Cloudflare Workers

To deploy to Cloudflare Workers, use `adapter-cloudflare-workers`.

> Unless you have a specific reason to use `adapter-cloudflare-workers`, it's recommended that you use `adapter-cloudflare` instead. Both adapters have equivalent functionality, but Cloudflare Pages offers features like GitHub integration with automatic builds and deploys, preview deployments, instant rollback and so on.

Usage

Install with `npm i -D @sveltejs/adapter-cloudflare-workers`, then add the adapter to your `svelte.config.js`:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-cloudflare-workers';

export default {
    kit: {
        adapter: adapter()
    }
};
```

Basic Configuration

This adapter expects to find a wrangler.toml file in the project root. It should look something like this:

```
/// file: wrangler.toml
name = "<your-service-name>"
account_id = "<your-account-id>"

main = "./.cloudflare/worker.js"
site.bucket = "./.cloudflare/public"

build.command = "npm run build"

compatibility_date = "2021-11-12"
workers_dev = true
```

`<your-service-name>` can be anything. `<your-account-id>` can be found by logging into your Cloudflare dashboard and grabbing it from the end of the URL:

```
https://dash.cloudflare.com/<your-account-id>
```

> You should add the `.cloudflare` directory (or whichever directories you specified for `main` and `site.bucket`) to your `.gitignore`.

You will need to install wrangler and log in, if you haven't already:

```
npm i -g wrangler
wrangler login
```

Then, you can build your app and deploy it:

```
wrangler publish
```

Custom config

If you would like to use a config file other than `wrangler.toml`, you can do like so:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-cloudflare-workers';

export default {
    kit: {
        adapter: adapter({ config: '<your-wrangler-name>.toml' })
    }
};
```

Bindings

The `env` object contains your project's bindings, which consist of KV/DO namespaces, etc. It is passed to SvelteKit via the `platform` property, along with `context` and `caches`, meaning that you can access it in hooks and endpoints:

```
// @errors: 7031
export async function POST({ request, platform }) {
    const x = platform.env.YOUR_DURABLE_OBJECT_NAMESPACE.idFromName('x');
}
```

> SvelteKit's built-in `$env` module should be preferred for environment variables.

To make these types available to your app, reference them in your `src/app.d.ts`:

```
/// file: src/app.d.ts
declare global {
    namespace App {
        interface Platform {
+           env?: {
+               YOUR_KV_NAMESPACE: KVNamespace;
+               YOUR_DURABLE_OBJECT_NAMESPACE: DurableObjectNamespace;
```

```
+            };
         }
     }
}

export {};
```

Testing Locally

`platform.env` is only available in the final build and not in dev mode. For testing the build, you can use wrangler. Once you have built your site, run `wrangler dev`. Ensure you have your bindings in your `wrangler.toml`. Wrangler version 3 is recommended.

Troubleshooting

Worker size limits

When deploying to workers, the server generated by SvelteKit is bundled into a single file. Wrangler will fail to publish your worker if it exceeds the size limits after minification. You're unlikely to hit this limit usually, but some large libraries can cause this to happen. In that case, you can try to reduce the size of your worker by only importing such libraries on the client side. See the FAQ for more information.

Accessing the file system

You can't access the file system through methods like `fs.readFileSync` in Serverless/Edge environments. If you need to access files that way, do that during building the app through prerendering. If you have a blog for example and don't want to manage your content through a CMS, then you need to prerender the content (or prerender the endpoint from which you get it) and redeploy your blog everytime you add new content.

Go to TOC

Netlify

To deploy to Netlify, use `adapter-netlify`.

This adapter will be installed by default when you use `adapter-auto`, but adding it to your project allows you to specify Netlify-specific options.

Usage

Install with `npm i -D @sveltejs/adapter-netlify`, then add the adapter to your `svelte.config.js`:

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-netlify';

export default {
    kit: {
        // default options are shown
        adapter: adapter({
            // if true, will create a Netlify Edge Function rather
            // than using standard Node-based functions
            edge: false,

            // if true, will split your app into multiple functions
            // instead of creating a single one for the entire app.
            // if `edge` is true, this option cannot be used
            split: false
        })
    }
};
```

Then, make sure you have a netlify.toml file in the project root. This will determine where to write static assets based on the `build.publish` settings, as per this sample configuration:

```
[build]
    command = "npm run build"
    publish = "build"
```

If the `netlify.toml` file or the `build.publish` value is missing, a default value of `"build"` will be used. Note that if you have set the publish directory in the Netlify UI to something else then you will need to set it in `netlify.toml` too, or use the default value of `"build"`.

Node version

New projects will use Node 16 by default. However, if you're upgrading a project you created a while ago it may be stuck on an older version. See the Netlify docs for details on manually specifying Node 16 or newer.

Netlify Edge Functions (beta)

SvelteKit supports the beta release of Netlify Edge Functions. If you pass the option `edge: true` to the `adapter` function, server-side rendering will happen in a Deno-based edge function that's deployed close to the site visitor. If set to `false` (the default), the site will deploy to standard Node-based Netlify Functions.

```
// @errors: 2307
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-netlify';

export default {
    kit: {
        adapter: adapter({
            // will create a Netlify Edge Function using Deno-based
            // rather than using standard Node-based functions
            edge: true
        })
    }
};
```

Netlify alternatives to SvelteKit functionality

You may build your app using functionality provided directly by SvelteKit without relying on any Netlify functionality. Using the SvelteKit versions of these features will allow them to be used in dev mode, tested with integration tests, and to work with other adapters should you ever decide to switch away from Netlify. However, in some scenarios you may find it beneficial to use the Netlify versions of these features. One example would be if you're migrating an app that's already hosted on Netlify to SvelteKit.

Redirect rules

During compilation, redirect rules are automatically appended to your `_redirects` file. (If it doesn't exist yet, it will be created.) That means:

- `[[redirects]]` in `netlify.toml` will never match as `_redirects` has a higher priority. So always put your rules in the `_redirects` file.
- `_redirects` shouldn't have any custom "catch all" rules such as `/* /foobar/:splat`. Otherwise the automatically appended rule will never be applied as Netlify is only processing the first matching rule.

Netlify Forms

1. Create your Netlify HTML form as described here, e.g. as `/routes/contact/+page.svelte`. (Don't forget to add the hidden `form-name` input element!)
2. Netlify's build bot parses your HTML files at deploy time, which means your form must be prerendered as HTML. You can either add `export const prerender = true` to your `contact.svelte` to prerender just that page or set the `kit.prerender.force: true` option to prerender all pages.
3. If your Netlify form has a custom success message like `<form netlify ... action="/success">` then ensure the corresponding `/routes/success/+page.svelte` exists and is prerendered.

Netlify Functions

With this adapter, SvelteKit endpoints are hosted as Netlify Functions. Netlify function handlers have additional context, including Netlify Identity information. You can access this context via the `event.platform.context` field inside your hooks and `+page.server` or `+layout.server` endpoints. These are serverless functions when the `edge` property is `false` in the adapter config or edge functions when it is `true`.

```
// @errors: 2705 7006
/// file: +page.server.js
export const load = async (event) => {
    const context = event.platform.context;
    console.log(context); // shows up in your functions log in the Netlify app
};
```

Additionally, you can add your own Netlify functions by creating a directory for them and adding the configuration to your `netlify.toml` file. For example:

```
[build]
    command = "npm run build"
    publish = "build"

[functions]
    directory = "functions"
```

Troubleshooting

Accessing the file system

You can't access the file system through methods like `fs.readFileSync` in Serverless/Edge environments. If you need to access files that way, do that during building the app through prerendering. If you have a blog for example and don't want to manage your content through a CMS, then you need to prerender the content (or prerender the endpoint from which you get it) and redeploy your blog everytime you add new content.

Go to TOC

Vercel

To deploy to Vercel, use `adapter-vercel`.

This adapter will be installed by default when you use `adapter-auto`, but adding it to your project allows you to specify Vercel-specific options.

Usage

Install with `npm i -D @sveltejs/adapter-vercel`, then add the adapter to your `svelte.config.js`:

```
// @errors: 2307 2345
/// file: svelte.config.js
import adapter from '@sveltejs/adapter-vercel';

export default {
    kit: {
        adapter: adapter({
            // see the 'Deployment configuration' section below
        })
    }
};
```

Deployment configuration

To control how your routes are deployed to Vercel as functions, you can specify deployment configuration, either through the option shown above or with `export const config` inside `+server.js`, `+page(.server).js` and `+layout(.server).js` files.

For example you could deploy some parts of your app as Edge Functions...

```
/// file: about/+page.js
/** @type {import('@sveltejs/adapter-vercel').Config} */
export const config = {
    runtime: 'edge'
};
```

...and others as Serverless Functions (note that by specifying `config` inside a layout, it applies to all child pages):

```
/// file: admin/+layout.js
/** @type {import('@sveltejs/adapter-vercel').Config} */
export const config = {
    runtime: 'nodejs18.x'
};
```

The following options apply to all functions:

- `runtime`: `'edge'`, `'nodejs16.x'` or `'nodejs18.x'`. By default, the adapter will select `'nodejs16.x'` or `'nodejs18.x'` depending on the Node version your project is configured to use on the Vercel dashboard
- `regions`: an array of edge network regions (defaulting to `["iad1"]` for serverless functions) or `'all'` if `runtime` is `edge` (its default). Note that multiple regions for serverless functions are only supported on Enterprise plans
- `split`: if `true`, causes a route to be deployed as an individual function. If `split` is set to `true` at the adapter level, all routes will be deployed as individual functions

Additionally, the following option applies to edge functions:

- `external`: an array of dependencies that esbuild should treat as external when bundling functions. This should only be used to exclude optional dependencies that will not run outside Node

And the following option apply to serverless functions:

- `memory`: the amount of memory available to the function. Defaults to `1024` Mb, and can be decreased to `128` Mb or increased in 64Mb increments up to `3008` Mb on Pro or Enterprise accounts
- `maxDuration`: maximum execution duration of the function. Defaults to `10` seconds for Hobby accounts, `60` for Pro and `900` for Enterprise
- `isr`: configuration Incremental Static Regeneration, described below

If your functions need to access data in a specific region, it's recommended that they be deployed in the same region (or close to it) for optimal performance.

Incremental Static Regeneration

Vercel supports Incremental Static Regeneration (ISR), which provides the performance and cost advantages of prerendered content with the flexibility of dynamically rendered content.

To add ISR to a route, include the `isr` property in your `config` object:

```
/// file: blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$env/static/private' {
    export const BYPASS_TOKEN: string;
}

// @filename: index.js
// cut---
import { BYPASS_TOKEN } from '$env/static/private';

export const config = {
    isr: {
        // Expiration time (in seconds) before the cached asset will be re-
generated by invoking the Serverless Function.
        // Setting the value to `false` means it will never expire.
        expiration: 60,

        // Random token that can be provided in the URL to bypass the cached
version of the asset, by requesting the asset
        // with a __prerender_bypass=<token> cookie.
```

```
        //
        // Making a `GET` or `HEAD` request with `x-prerender-revalidate: <token>`
will force the asset to be re-validated.
        bypassToken: BYPASS_TOKEN,

        // List of valid query parameters. Other parameters (such as utm tracking
codes) will be ignored,
        // ensuring that they do not result in content being regenerated
unnecessarily
        allowQuery: ['search']
    }
};
```

The `expiration` property is required; all others are optional.

Environment variables

Vercel makes a set of deployment-specific environment variables available. Like other environment variables, these are accessible from `$env/static/private` and `$env/dynamic/private` (sometimes — more on that later), and inaccessible from their public counterparts. To access one of these variables from the client:

```
// @errors: 2305
/// file: +layout.server.js
import { VERCEL_COMMIT_REF } from '$env/static/private';

/** @type {import('./$types').LayoutServerLoad} */
export function load() {
    return {
        deploymentGitBranch: VERCEL_COMMIT_REF
    };
}
```

```
<! file: +layout.svelte --->
<script>
    /** @type {import('./$types').LayoutServerData} */
    export let data;
</script>

<p>This staging environment was deployed from {data.deploymentGitBranch}.</p>
```

Since all of these variables are unchanged between build time and run time when building on Vercel, we recommend using `$env/static/private` — which will statically replace the variables, enabling optimisations like dead code elimination — rather than `$env/dynamic/private`.

Notes

Vercel functions

If you have Vercel functions contained in the `api` directory at the project's root, any requests for `/api/*` will *not* be handled by SvelteKit. You should implement these as API routes in your SvelteKit app instead, unless you need to use a non-JavaScript language in which case you will need to ensure that you don't have any `/api/*` routes in your SvelteKit app.

Node version

Projects created before a certain date will default to using Node 14, while SvelteKit requires Node 16 or later. You can change the Node version in your project settings.

Troubleshooting

Accessing the file system

You can't access the file system through methods like `fs.readFileSync` in Serverless/Edge environments. If you need to access files that way, do that during building the app through prerendering. If you have a blog for example and don't want to manage your content through a CMS, then you need to prerender the content (or prerender the endpoint from which you get it) and redeploy your blog everytime you add new content.

Go to TOC

Writing adapters

If an adapter for your preferred environment doesn't yet exist, you can build your own. We recommend looking at the source for an adapter to a platform similar to yours and copying it as a starting point.

Adapters packages must implement the following API, which creates an `Adapter`:

```
// @filename: ambient.d.ts
type AdapterSpecificOptions = any;

// @filename: index.js
// cut---
/** @param {AdapterSpecificOptions} options */
export default function (options) {
    /** @type {import('@sveltejs/kit').Adapter} */
    const adapter = {
        name: 'adapter-package-name',
        async adapt(builder) {
            // adapter implementation
        }
    };

    return adapter;
}
```

Within the `adapt` method, there are a number of things that an adapter should do:

- Clear out the build directory
- Write SvelteKit output with `builder.writeClient`, `builder.writeServer`, and `builder.writePrerendered`
- Output code that:
  - Imports `Server` from `${builder.getServerDirectory()}/index.js`
  - Instantiates the app with a manifest generated with `builder.generateManifest({ relativePath })`
  - Listens for requests from the platform, converts them to a standard Request if necessary, calls the `server.respond(request, { getClientAddress })` function to generate a Response and responds with it
  - expose any platform-specific information to SvelteKit via the `platform` option passed to `server.respond`
  - Globally shims `fetch` to work on the target platform, if necessary. SvelteKit provides a `@sveltejs/kit/node/polyfills` helper for platforms that can use `undici`
- Bundle the output to avoid needing to install dependencies on the target platform, if necessary
- Put the user's static files and the generated JS/CSS in the correct location for the target platform

Where possible, we recommend putting the adapter output under the `build/` directory with any intermediate output placed under `.svelte-kit/[adapter-name]`.

Go to TOC

Advanced routing

Rest parameters

If the number of route segments is unknown, you can use rest syntax — for example you might implement GitHub's file viewer like so...

```
/[org]/[repo]/tree/[branch]/[...file]
```

...in which case a request for `/sveltejs/kit/tree/master/documentation/docs/04-advanced-routing.md` would result in the following parameters being available to the page:

```
// @noErrors
{
    org: 'sveltejs',
    repo: 'kit',
    branch: 'master',
    file: 'documentation/docs/04-advanced-routing.md'
}
```

> `src/routes/a/[...rest]/z/+page.svelte` will match `/a/z` (i.e. there's no parameter at all) as well as `/a/b/z` and `/a/b/c/z` and so on. Make sure you check that the value of the rest parameter is valid, for example using a matcher.

404 pages

Rest parameters also allow you to render custom 404s. Given these routes...

```
src/routes/
├ marx-brothers/
|  ├ chico/
|  ├ harpo/
|  ├ groucho/
|  └ +error.svelte
└ +error.svelte
```

...the `marx-brothers/+error.svelte` file will *not* be rendered if you visit `/marx-brothers/karl`, because no route was matched. If you want to render the nested error page, you should create a route that matches any `/marx-brothers/*` request, and return a 404 from it:

```
src/routes/
├ marx-brothers/
+|  ├ [...path]/
|  ├ chico/
|  ├ harpo/
|  ├ groucho/
|  └ +error.svelte
└ +error.svelte
```

```
/// file: src/routes/marx-brothers/[...path]/+page.js
import { error } from '@sveltejs/kit';

/** @type {import('./$types').PageLoad} */
export function load(event) {
    throw error(404, 'Not Found');
}
```

If you don't handle 404 cases, they will appear in `handleError`

Optional parameters

A route like `[lang]/home` contains a parameter named `lang` which is required. Sometimes it's beneficial to make these parameters optional, so that in this example both `home` and `en/home` point to the same page. You can do that by wrapping the parameter in another bracket pair: `[[lang]]/home`

Note that an optional route parameter cannot follow a rest parameter (`[...rest]/[[optional]]`), since parameters are matched 'greedily' and the optional parameter would always be unused.

Matching

A route like `src/routes/archive/[page]` would match `/archive/3`, but it would also match `/archive/potato`. We don't want that. You can ensure that route parameters are well-formed by adding a *matcher* — which takes the parameter string (`"3"` or `"potato"`) and returns `true` if it is valid — to your `params` directory...

```
/// file: src/params/integer.js
/** @type {import('@sveltejs/kit').ParamMatcher} */
export function match(param) {
    return /^\d+$/.test(param);
}
```

...and augmenting your routes:

```
-src/routes/archive/[page]
+src/routes/archive/[page=integer]
```

If the pathname doesn't match, SvelteKit will try to match other routes (using the sort order specified below), before eventually returning a 404.

Each module in the `params` directory corresponds to a matcher, with the exception of `*.test.js` and `*.spec.js` files which may be used to unit test your matchers.

Matchers run both on the server and in the browser.

Sorting

It's possible for multiple routes to match a given path. For example each of these routes would match `/foo-abc`:

```
src/routes/[...catchall]/+page.svelte
src/routes/[[a=x]]/+page.svelte
src/routes/[b]/+page.svelte
src/routes/foo-[c]/+page.svelte
src/routes/foo-abc/+page.svelte
```

SvelteKit needs to know which route is being requested. To do so, it sorts them according to the following rules...

- More specific routes are higher priority (e.g. a route with no parameters is more specific than a route with one dynamic parameter, and so on)
- Parameters with matchers (`[name=type]`) are higher priority than those without (`[name]`)
- `[[optional]]` and `[...rest]` parameters are ignored unless they are the final part of the route, in which case they are treated with lowest priority. In other words `x/[[y]]/z` is treated equivalently to `x/z` for the purposes of sorting
- Ties are resolved alphabetically

...resulting in this ordering, meaning that `/foo-abc` will invoke `src/routes/foo-abc/+page.svelte`, and `/foo-def` will invoke `src/routes/foo-[c]/+page.svelte` rather than less specific routes:

```
src/routes/foo-abc/+page.svelte
src/routes/foo-[c]/+page.svelte
src/routes/[[a=x]]/+page.svelte
src/routes/[b]/+page.svelte
src/routes/[...catchall]/+page.svelte
```

Encoding

Some characters can't be used on the filesystem — `/` on Linux and Mac, `\ / : * ? " < > |` on Windows. The `#` and `%` characters have special meaning in URLs, and the `[ ] ( )` characters have special meaning to SvelteKit, so these also can't be used directly as part of your route.

To use these characters in your routes, you can use hexadecimal escape sequences, which have the format `[x+nn]` where `nn` is a hexadecimal character code:

- `\` — `[x+5c]`
- `/` — `[x+2f]`
- `:` — `[x+3a]`
- `*` — `[x+2a]`
- `?` — `[x+3f]`
- `"` — `[x+22]`
- `<` — `[x+3c]`
- `>` — `[x+3e]`

- `|` — `[x+7c]`
- `#` — `[x+23]`
- `%` — `[x+25]`
- `[` — `[x+5b]`
- `]` — `[x+5d]`
- `(` — `[x+28]`
- `)` — `[x+29]`

For example, to create a `/smileys/:-)` route, you would create a `src/routes/smileys/[x+3a]-[x+29]/+page.svelte` file.

You can determine the hexadecimal code for a character with JavaScript:

```
':'.charCodeAt(0).toString(16); // '3a', hence '[x+3a]'
```

You can also use Unicode escape sequences. Generally you won't need to as you can use the unencoded character directly, but if — for some reason — you can't have a filename with an emoji in it, for example, then you can use the escaped characters. In other words, these are equivalent:

```
src/routes/[u+d83e][u+dd2a]/+page.svelte
src/routes/🤪/+page.svelte
```

The format for a Unicode escape sequence is `[u+nnnn]` where `nnnn` is a valid value between `0000` and `10ffff`. (Unlike JavaScript string escaping, there's no need to use surrogate pairs to represent code points above `ffff`.) To learn more about Unicode encodings, consult Programming with Unicode.

> Since TypeScript struggles with directories with a leading `.` character, you may find it useful to encode these characters when creating e.g. `.well-known` routes: `src/routes/[x+2e]well-known/...`

Advanced layouts

By default, the *layout hierarchy* mirrors the *route hierarchy*. In some cases, that might not be what you want.

(group)

Perhaps you have some routes that are 'app' routes that should have one layout (e.g. `/dashboard` or `/item`), and others that are 'marketing' routes that should have a different layout (`/blog` or `/testimonials`). We can group these routes with a directory whose name is wrapped in parentheses — unlike normal directories, `(app)` and `(marketing)` do not affect the URL pathname of the routes inside them:

```
src/routes/
+| (app)/
| ├ dashboard/
```

```
|  ├ item/
|  └ +layout.svelte
+|  (marketing)/
|  ├ about/
|  ├ testimonials/
|  └ +layout.svelte
├ admin/
└ +layout.svelte
```

You can also put a `+page` directly inside a `(group)`, for example if `/` should be an `(app)` or a `(marketing)` page.

Breaking out of layouts

The root layout applies to every page of your app — if omitted, it defaults to `<slot />`. If you want some pages to have a different layout hierarchy than the rest, then you can put your entire app inside one or more groups *except* the routes that should not inherit the common layouts.

In the example above, the `/admin` route does not inherit either the `(app)` or `(marketing)` layouts.

+page@

Pages can break out of the current layout hierarchy on a route-by-route basis. Suppose we have an `/item/[id]/embed` route inside the `(app)` group from the previous example:

```
src/routes/
├ (app)/
|  ├ item/
|  |  ├ [id]/
|  |  |  ├ embed/
+|  |  |  |  └ +page.svelte
|  |  |  └ +layout.svelte
|  |  └ +layout.svelte
|  └ +layout.svelte
└ +layout.svelte
```

Ordinarily, this would inherit the root layout, the `(app)` layout, the `item` layout and the `[id]` layout. We can reset to one of those layouts by appending `@` followed by the segment name — or, for the root layout, the empty string. In this example, we can choose from the following options:

- `+page@[id].svelte` - inherits from `src/routes/(app)/item/[id]/+layout.svelte`
- `+page@item.svelte` - inherits from `src/routes/(app)/item/+layout.svelte`
- `+page@(app).svelte` - inherits from `src/routes/(app)/+layout.svelte`
- `+page@.svelte` - inherits from `src/routes/+layout.svelte`

```
src/routes/
├ (app)/
|  ├ item/
|  |  ├ [id]/
|  |  |  ├ embed/
+|  |  |  |  └ +page@(app).svelte
|  |  |  └ +layout.svelte
```

```
| | └ +layout.svelte
| └ +layout.svelte
└ +layout.svelte
```

+layout@

Like pages, layouts can *themselves* break out of their parent layout hierarchy, using the same technique. For example, a `+layout@.svelte` component would reset the hierarchy for all its child routes.

```
src/routes/
├ (app)/
| ├ item/
| | ├ [id]/
| | | ├ embed/
| | | | └ +page.svelte  // uses (app)/item/[id]/+layout.svelte
| | | ├ +layout.svelte  // inherits from (app)/item/+layout@.svelte
| | | └ +page.svelte    // uses (app)/item/+layout@.svelte
| | └ +layout@.svelte   // inherits from root layout, skipping
(app)/+layout.svelte
| └ +layout.svelte
└ +layout.svelte
```

When to use layout groups

Not all use cases are suited for layout grouping, nor should you feel compelled to use them. It might be that your use case would result in complex `(group)` nesting, or that you don't want to introduce a `(group)` for a single outlier. It's perfectly fine to use other means such as composition (reusable `load` functions or Svelte components) or if-statements to achieve what you want. The following example shows a layout that rewinds to the root layout and reuses components and functions that other layouts can also use:

```
<! file: src/routes/nested/route/+layout@.svelte --->
<script>
    import ReusableLayout from '$lib/ReusableLayout.svelte';
    export let data;
</script>

<ReusableLayout {data}>
    <slot />
</ReusableLayout>
```

```
/// file: src/routes/nested/route/+layout.js
// @filename: ambient.d.ts
declare module "$lib/reusable-load-function" {
    export function reusableLoad(event: import('@sveltejs/kit').LoadEvent):
Promise<Record<string, any>>;
}
// @filename: index.js
// cut---
import { reusableLoad } from '$lib/reusable-load-function';

/** @type {import('./$types').PageLoad} */
export function load(event) {
    // Add additional logic here, if needed
    return reusableLoad(event);
}
```

Further reading

- Tutorial: Advanced Routing

Go to TOC

Hooks

'Hooks' are app-wide functions you declare that SvelteKit will call in response to specific events, giving you fine-grained control over the framework's behaviour.

There are two hooks files, both optional:

- `src/hooks.server.js` — your app's server hooks
- `src/hooks.client.js` — your app's client hooks

Code in these modules will run when the application starts up, making them useful for initializing database clients and so on.

> You can configure the location of these files with `config.kit.files.hooks`.

Server hooks

The following hooks can be added to `src/hooks.server.js`:

handle

This function runs every time the SvelteKit server receives a request — whether that happens while the app is running, or during prerendering — and determines the response. It receives an `event` object representing the request and a function called `resolve`, which renders the route and generates a `Response`. This allows you to modify response headers or bodies, or bypass SvelteKit entirely (for implementing routes programmatically, for example).

```
/// file: src/hooks.server.js
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    if (event.url.pathname.startsWith('/custom')) {
        return new Response('custom response');
    }

    const response = await resolve(event);
    return response;
}
```

> Requests for static assets — which includes pages that were already prerendered — are *not* handled by SvelteKit.

If unimplemented, defaults to `({ event, resolve }) => resolve(event)`. To add custom data to the request, which is passed to handlers in `+server.js` and server `load` functions, populate the `event.locals` object, as shown below.

```
/// file: src/hooks.server.js
// @filename: ambient.d.ts
type User = {
    name: string;
}

declare namespace App {
    interface Locals {
        user: User;
    }
}

const getUserInformation: (cookie: string | void) => Promise<User>;

// @filename: index.js
// cut---
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    event.locals.user = await getUserInformation(event.cookies.get('sessionid'));

    const response = await resolve(event);
    response.headers.set('x-custom-header', 'potato');

    return response;
}
```

You can define multiple `handle` functions and execute them with the `sequence` helper function.

`resolve` also supports a second, optional parameter that gives you more control over how the response will be rendered. That parameter is an object that can have the following fields:

- `transformPageChunk(opts: { html: string, done: boolean }): MaybePromise<string | undefined>` — applies custom transforms to HTML. If `done` is true, it's the final chunk. Chunks are not guaranteed to be well-formed HTML (they could include an element's opening tag but not its closing tag, for example) but they will always be split at sensible boundaries such as `%sveltekit.head%` or layout/page components.
- `filterSerializedResponseHeaders(name: string, value: string): boolean` — determines which headers should be included in serialized responses when a `load` function loads a resource with `fetch`. By default, none will be included.
- `preload(input: { type: 'js' | 'css' | 'font' | 'asset', path: string }): boolean` — determines what files should be added to the `<head>` tag to preload it. The method is called with each file that was found at build time while constructing the code chunks — so if you for example have `import './styles.css'` in your `+page.svelte`, `preload` will be called with the resolved path to that CSS file when visiting that page. Note that in dev mode `preload` is *not* called, since it depends on analysis that happens at build time. Preloading can improve performance by downloading assets sooner, but it can also hurt if too much is downloaded unnecessarily. By default, `js` and `css` files will be preloaded. `asset` files are not preloaded at all currently, but we may add this later after evaluating feedback.

```
/// file: src/hooks.server.js
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    const response = await resolve(event, {
        transformPageChunk: ({ html }) => html.replace('old', 'new'),
        filterSerializedResponseHeaders: (name) => name.startsWith('x-'),
        preload: ({ type, path }) => type === 'js' || path.includes('/important/')
    });

    return response;
}
```

Note that `resolve(...)` will never throw an error, it will always return a `Promise<Response>` with the appropriate status code. If an error is thrown elsewhere during `handle`, it is treated as fatal, and SvelteKit will respond with a JSON representation of the error or a fallback error page — which can be customised via `src/error.html` — depending on the `Accept` header. You can read more about error handling here.

handleFetch

This function allows you to modify (or replace) a `fetch` request that happens inside a `load` or `action` function that runs on the server (or during pre-rendering).

For example, your `load` function might make a request to a public URL like `https://api.yourapp.com` when the user performs a client-side navigation to the respective page, but during SSR it might make sense to hit the API directly (bypassing whatever proxies and load balancers sit between it and the public internet).

```
/// file: src/hooks.server.js
/** @type {import('@sveltejs/kit').HandleFetch} */
export async function handleFetch({ request, fetch }) {
    if (request.url.startsWith('https://api.yourapp.com/')) {
        // clone the original request, but change the URL
        request = new Request(
            request.url.replace('https://api.yourapp.com/',
'http://localhost:9999/'),
            request
        );
    }

    return fetch(request);
}
```

Credentials

For same-origin requests, SvelteKit's `fetch` implementation will forward `cookie` and `authorization` headers unless the `credentials` option is set to `"omit"`.

For cross-origin requests, `cookie` will be included if the request URL belongs to a subdomain of the app — for example if your app is on `my-domain.com`, and your API is on `api.my-domain.com`, cookies will be included in the request.

If your app and your API are on sibling subdomains — `www.my-domain.com` and `api.my-domain.com` for example — then a cookie belonging to a common parent domain like `my-domain.com` will *not* be included, because SvelteKit has no way to know which domain the cookie belongs to. In these cases you will need to manually include the cookie using `handleFetch`:

```
/// file: src/hooks.server.js
// @errors: 2345
/** @type {import('@sveltejs/kit').HandleFetch} */
export async function handleFetch({ event, request, fetch }) {
    if (request.url.startsWith('https://api.my-domain.com/')) {
        request.headers.set('cookie', event.request.headers.get('cookie'));
    }

    return fetch(request);
}
```

Shared hooks

The following can be added to `src/hooks.server.js` *and* `src/hooks.client.js`:

handleError

If an unexpected error is thrown during loading or rendering, this function will be called with the `error` and the `event`. This allows for two things:

- you can log the error
- you can generate a custom representation of the error that is safe to show to users, omitting sensitive details like messages and stack traces. The returned value becomes the value of `$page.error`. It defaults to `{ message: 'Not Found' }` in case of a 404 (you can detect them through `event.route.id` being `null`) and to `{ message: 'Internal Error' }` for everything else. To make this type-safe, you can customize the expected shape by declaring an `App.Error` interface (which must include `message: string`, to guarantee sensible fallback behavior).

The following code shows an example of typing the error shape as `{ message: string; errorId: string }` and returning it accordingly from the `handleError` functions:

```
/// file: src/app.d.ts
declare global {
    namespace App {
        interface Error {
            message: string;
            errorId: string;
        }
    }
}

export {};
```

```
/// file: src/hooks.server.js
// @errors: 2322
// @filename: ambient.d.ts
declare module '@sentry/node' {
```

```
    export const init: (opts: any) => void;
    export const captureException: (error: any, opts: any) => void;
}

// @filename: index.js
// cut---
import * as Sentry from '@sentry/node';
import crypto from 'crypto';

Sentry.init({/*...*/})

/** @type {import('@sveltejs/kit').HandleServerError} */
export async function handleError({ error, event }) {
    const errorId = crypto.randomUUID();
    // example integration with https://sentry.io/
    Sentry.captureException(error, { extra: { event, errorId } });

    return {
        message: 'Whoops!',
        errorId
    };
}
```

```
/// file: src/hooks.client.js
// @errors: 2322
// @filename: ambient.d.ts
declare module '@sentry/svelte' {
    export const init: (opts: any) => void;
    export const captureException: (error: any, opts: any) => void;
}

// @filename: index.js
// cut---
import * as Sentry from '@sentry/svelte';

Sentry.init({/*...*/})

/** @type {import('@sveltejs/kit').HandleClientError} */
export async function handleError({ error, event }) {
    const errorId = crypto.randomUUID();
    // example integration with https://sentry.io/
    Sentry.captureException(error, { extra: { event, errorId } });

    return {
        message: 'Whoops!',
        errorId
    };
}
```

In `src/hooks.client.js`, the type of `handleError` is `HandleClientError` instead of `HandleServerError`, and `event` is a `NavigationEvent` rather than a `RequestEvent`.

This function is not called for *expected* errors (those thrown with the `error` function imported from `@sveltejs/kit`).

During development, if an error occurs because of a syntax error in your Svelte code, the passed in error has a `frame` property appended highlighting the location of the error.

> Make sure that `handleError` *never* throws an error

Further reading

- Tutorial: Hooks

Go to TOC

Errors

Errors are an inevitable fact of software development. SvelteKit handles errors differently depending on where they occur, what kind of errors they are, and the nature of the incoming request.

Error objects

SvelteKit distinguishes between expected and unexpected errors, both of which are represented as simple `{ message: string }` objects by default.

You can add additional properties, like a `code` or a tracking `id`, as shown in the examples below. (When using TypeScript this requires you to redefine the `Error` type as described in type safety).

Expected errors

An *expected* error is one created with the `error` helper imported from `@sveltejs/kit`:

```
/// file: src/routes/blog/[slug]/+page.server.js
// @filename: ambient.d.ts
declare module '$lib/server/database' {
    export function getPost(slug: string): Promise<{ # string, content: string } |
undefined>
}

// @filename: index.js
// cut---
import { error } from '@sveltejs/kit';
import * as db from '$lib/server/database';

/** @type {import('./$types').PageServerLoad} */
export async function load({ params }) {
    const post = await db.getPost(params.slug);

    if (!post) {
        throw error(404, {
            message: 'Not found'
        });
    }

    return { post };
}
```

This tells SvelteKit to set the response status code to 404 and render an `+error.svelte` component, where `$page.error` is the object provided as the second argument to `error(...)`.

```
<! file: src/routes/+error.svelte --->
<script>
    import { page } from '$app/stores';
</script>

<h1>{$page.error.message}</h1>
```

You can add extra properties to the error object if needed...

```
throw error(404, {
    message: 'Not found',
+   code: 'NOT_FOUND'
});
```

...otherwise, for convenience, you can pass a string as the second argument:

```
-throw error(404, { message: 'Not found' });
+throw error(404, 'Not found');
```

Unexpected errors

An *unexpected* error is any other exception that occurs while handling a request. Since these can contain sensitive information, unexpected error messages and stack traces are not exposed to users.

By default, unexpected errors are printed to the console (or, in production, your server logs), while the error that is exposed to the user has a generic shape:

```
{ "message": "Internal Error" }
```

Unexpected errors will go through the `handleError` hook, where you can add your own error handling — for example, sending errors to a reporting service, or returning a custom error object.

```
/// file: src/hooks.server.js
// @errors: 2322 1360 2571 2339
// @filename: ambient.d.ts
declare module '@sentry/node' {
    export const init: (opts: any) => void;
    export const captureException: (error: any, opts: any) => void;
}

// @filename: index.js
// cut---
import * as Sentry from '@sentry/node';

Sentry.init({/*...*/})

/** @type {import('@sveltejs/kit').HandleServerError} */
export function handleError({ error, event }) {
    // example integration with https://sentry.io/
    Sentry.captureException(error, { extra: { event } });

    return {
        message: 'Whoops!',
        code: error?.code ?? 'UNKNOWN'
    };
}
```

Make sure that `handleError` *never* throws an error

Responses

If an error occurs inside `handle` or inside a `+server.js` request handler, SvelteKit will respond with either a fallback error page or a JSON representation of the error object, depending on the request's `Accept` headers.

You can customise the fallback error page by adding a `src/error.html` file:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>%sveltekit.error.message%</title>
    </head>
    <body>
        <h1>My custom error page</h1>
        <p>Status: %sveltekit.status%</p>
        <p>Message: %sveltekit.error.message%</p>
    </body>
</html>
```

SvelteKit will replace `%sveltekit.status%` and `%sveltekit.error.message%` with their corresponding values.

If the error instead occurs inside a `load` function while rendering a page, SvelteKit will render the `+error.svelte` component nearest to where the error occurred. If the error occurs inside a `load` function in `+layout(.server).js`, the closest error boundary in the tree is an `+error.svelte` file *above* that layout (not next to it).

The exception is when the error occurs inside the root `+layout.js` or `+layout.server.js`, since the root layout would ordinarily *contain* the `+error.svelte` component. In this case, SvelteKit uses the fallback error page.

Type safety

If you're using TypeScript and need to customize the shape of errors, you can do so by declaring an `App.Error` interface in your app (by convention, in `src/app.d.ts`, though it can live anywhere that TypeScript can 'see'):

```
/// file: src/app.d.ts
declare global {
    namespace App {
        interface Error {
+           code: string;
+           id: string;
        }
    }
}

export {};
```

This interface always includes a `message: string` property.

Further reading

- Tutorial: Errors and redirects
- Tutorial: Hooks

Go to TOC

Link options

In SvelteKit, `<a>` elements (rather than framework-specific `<Link>` components) are used to navigate between the routes of your app. If the user clicks on a link whose `href` is 'owned' by the app (as opposed to, say, a link to an external site) then SvelteKit will navigate to the new page by importing its code and then calling any `load` functions it needs to fetch data.

You can customise the behaviour of links with `data-sveltekit-*` attributes. These can be applied to the `<a>` itself, or to a parent element.

These options also apply to `<form>` elements with `method="GET"`.

data-sveltekit-preload-data

Before the browser registers that the user has clicked on a link, we can detect that they've hovered the mouse over it (on desktop) or that a `touchstart` or `mousedown` event was triggered. In both cases, we can make an educated guess that a `click` event is coming.

SvelteKit can use this information to get a head start on importing the code and fetching the page's data, which can give us an extra couple of hundred milliseconds — the difference between a user interface that feels laggy and one that feels snappy.

We can control this behaviour with the `data-sveltekit-preload-data` attribute, which can have one of two values:

- `"hover"` means that preloading will start if the mouse comes to a rest over a link. On mobile, preloading begins on `touchstart`
- `"tap"` means that preloading will start as soon as a `touchstart` or `mousedown` event is registered

The default project template has a `data-sveltekit-preload-data="hover"` attribute applied to the `<body>` element in `src/app.html`, meaning that every link is preloaded on hover by default:

```
<body data-sveltekit-preload-data="hover">
    <div style="display: contents">%sveltekit.body%</div>
</body>
```

Sometimes, calling `load` when the user hovers over a link might be undesirable, either because it's likely to result in false positives (a click needn't follow a hover) or because data is updating very quickly and a delay could mean staleness.

In these cases, you can specify the `"tap"` value, which causes SvelteKit to call `load` only when the user taps or clicks on a link:

```
<a data-sveltekit-preload-data="tap" href="/stonks">
    Get current stonk values
</a>
```

> You can also programmatically invoke `preloadData` from `$app/navigation`.

Data will never be preloaded if the user has chosen reduced data usage, meaning `navigator.connection.saveData` is `true`.

data-sveltekit-preload-code

Even in cases where you don't want to preload *data* for a link, it can be beneficial to preload the *code*. The `data-sveltekit-preload-code` attribute works similarly to `data-sveltekit-preload-data`, except that it can take one of four values, in decreasing 'eagerness':

- `"eager"` means that links will be preloaded straight away
- `"viewport"` means that links will be preloaded once they enter the viewport
- `"hover"` - as above, except that only code is preloaded
- `"tap"` - as above, except that only code is preloaded

Note that `viewport` and `eager` only apply to links that are present in the DOM immediately following navigation — if a link is added later (in an `{#if ...}` block, for example) it will not be preloaded until triggered by `hover` or `tap`. This is to avoid performance pitfalls resulting from aggressively observing the DOM for changes.

> Since preloading code is a prerequisite for preloading data, this attribute will only have an effect if it specifies a more eager value than any `data-sveltekit-preload-data` attribute that is present.

As with `data-sveltekit-preload-data`, this attribute will be ignored if the user has chosen reduced data usage.

data-sveltekit-reload

Occasionally, we need to tell SvelteKit not to handle a link, but allow the browser to handle it. Adding a `data-sveltekit-reload` attribute to a link...

```
<a data-sveltekit-reload href="/path">Path</a>
```

...will cause a full-page navigation when the link is clicked.

Links with a `rel="external"` attribute will receive the same treatment. In addition, they will be ignored during prerendering.

data-sveltekit-replacestate

Sometimes you don't want navigation to create a new entry in the browser's session history. Adding a `data-sveltekit-replacestate` attribute to a link...

```
<a data-sveltekit-replacestate href="/path">Path</a>
```

...will replace the current `history` entry rather than creating a new one with `pushState` when the link is clicked.

data-sveltekit-keepfocus

Sometimes you don't want focus to be reset after navigation. For example, maybe you have a search form that submits as the user is typing, and you want to keep focus on the text input. Adding a `data-sveltekit-keepfocus` attribute to it...

```
<form data-sveltekit-keepfocus>
    <input type="text" name="query">
</form>
```

...will cause the currently focused element to retain focus after navigation. In general, avoid using this attribute on links, since the focused element would be the `<a>` tag (and not a previously focused element) and screen reader and other assistive technology users often expect focus to be moved after a navigation. You should also only use this attribute on elements that still exist after navigation. If the element no longer exists, the user's focus will be lost, making for a confusing experience for assistive technology users.

data-sveltekit-noscroll

When navigating to internal links, SvelteKit mirrors the browser's default navigation behaviour: it will change the scroll position to 0,0 so that the user is at the very top left of the page (unless the link includes a `#hash`, in which case it will scroll to the element with a matching ID).

In certain cases, you may wish to disable this behaviour. Adding a `data-sveltekit-noscroll` attribute to a link...

```
<a href="path" data-sveltekit-noscroll>Path</a>
```

...will prevent scrolling after the link is clicked.

Disabling options

To disable any of these options inside an element where they have been enabled, use the `"false"` value:

```
<div data-sveltekit-preload-data>
    <!-- these links will be preloaded -->
    <a href="/a">a</a>
    <a href="/b">b</a>
    <a href="/c">c</a>
</div>
```

```
    <div data-sveltekit-preload-data="false">
        <!-- these links will NOT be preloaded -->
        <a href="/d">d</a>
        <a href="/e">e</a>
        <a href="/f">f</a>
    </div>
</div>
```

To apply an attribute to an element conditionally, do this (`"true"` and `"false"` are both accepted values):

```
<div data-sveltekit-reload={shouldReload}>
```<span style='float: footnote;'><a href="../../index.html#toc">Go to TOC</a>

```
```

Service workers

Service workers act as proxy servers that handle network requests inside your app. This makes it possible to make your app work offline, but even if you don't need offline support (or can't realistically implement it because of the type of app you're building), it's often worth using service workers to speed up navigation by precaching your built JS and CSS.

In SvelteKit, if you have a `src/service-worker.js` file (or `src/service-worker/index.js`) it will be bundled and automatically registered. You can change the location of your service worker if you need to.

You can disable automatic registration if you need to register the service worker with your own logic or use another solution. The default registration looks something like this:

```
if ('serviceWorker' in navigator) {
    addEventListener('load', function () {
        navigator.serviceWorker.register('./path/to/service-worker.js');
    });
}
```

Inside the service worker

Inside the service worker you have access to the `$service-worker` module, which provides you with the paths to all static assets, build files and prerendered pages. You're also provided with an app version string, which you can use for creating a unique cache name, and the deployment's `base` path. If your Vite config specifies `define` (used for global variable replacements), this will be applied to service workers as well as your server/client builds.

The following example caches the built app and any files in `static` eagerly, and caches all other requests as they happen. This would make each page work offline once visited.

```
// @errors: 2339
/// <reference types="@sveltejs/kit" />
import { build, files, version } from '$service-worker';

// Create a unique cache name for this deployment
const CACHE = `cache-${version}`;

const ASSETS = [
    ...build, // the app itself
    ...files  // everything in `static`
];

self.addEventListener('install', (event) => {
    // Create a new cache and add all files to it
    async function addFilesToCache() {
        const cache = await caches.open(CACHE);
        await cache.addAll(ASSETS);
    }

    event.waitUntil(addFilesToCache());
});
```

```
self.addEventListener('activate', (event) => {
    // Remove previous cached data from disk
    async function deleteOldCaches() {
        for (const key of await caches.keys()) {
            if (key !== CACHE) await caches.delete(key);
        }
    }

    event.waitUntil(deleteOldCaches());
});

self.addEventListener('fetch', (event) => {
    // ignore POST requests etc
    if (event.request.method !== 'GET') return;

    async function respond() {
        const url = new URL(event.request.url);
        const cache = await caches.open(CACHE);

        // `build`/`files` can always be served from the cache
        if (ASSETS.includes(url.pathname)) {
            return cache.match(url.pathname);
        }

        // for everything else, try the network first, but
        // fall back to the cache if we're offline
        try {
            const response = await fetch(event.request);

            if (response.status === 200) {
                cache.put(event.request, response.clone());
            }

            return response;
        } catch {
            return cache.match(event.request);
        }
    }

    event.respondWith(respond());
});
```

Be careful when caching! In some cases, stale data might be worse than data that's unavailable while offline. Since browsers will empty caches if they get too full, you should also be careful about caching large assets like video files.

During development

The service worker is bundled for production, but not during development. For that reason, only browsers that support modules in service workers will be able to use them at dev time. If you are manually registering your service worker, you will need to pass the `{ type: 'module' }` option in development:

```
import { dev } from '$app/environment';

navigator.serviceWorker.register('/service-worker.js', {
    type: dev ? 'module' : 'classic'
});
```

`build` and `prerendered` are empty arrays during development

Type safety

Setting up proper types for service workers requires some manual setup. Inside your `service-worker.js`, add the following to the top of your file:

```
/// <reference types="@sveltejs/kit" />
/// <reference no-default-lib="true"/>
/// <reference lib="esnext" />
/// <reference lib="webworker" />

const sw = /** @type {ServiceWorkerGlobalScope} */ (/** @type {unknown} */
(self));
```

```
/// <reference types="@sveltejs/kit" />
/// <reference no-default-lib="true"/>
/// <reference lib="esnext" />
/// <reference lib="webworker" />

const sw = self as unknown as ServiceWorkerGlobalScope;
```

This disables access to DOM typings like `HTMLElement` which are not available inside a service worker and instantiates the correct globals. The reassignment of `self` to `sw` allows you to type cast it in the process (there are a couple of ways to do this, but the easiest that requires no additional files). Use `sw` instead of `self` in the rest of the file. The reference to the SvelteKit types ensures that the `$service-worker` import has proper type definitions.

Other solutions

SvelteKit's service worker implementation is deliberately low-level. If you need a more full-flegded but also more opinionated solution, we recommend looking at solutions like Vite PWA plugin, which uses Workbox. For more general information on service workers, we recommend the MDN web docs.

Go to TOC

Server-only modules

Like a good friend, SvelteKit keeps your secrets. When writing your backend and frontend in the same repository, it can be easy to accidentally import sensitive data into your front-end code (environment variables containing API keys, for example). SvelteKit provides a way to prevent this entirely: server-only modules.

Private environment variables

The `$env/static/private` and `$env/dynamic/private` modules, which are covered in the modules section, can only be imported into modules that only run on the server, such as `hooks.server.js` or `+page.server.js`.

Your modules

You can make your own modules server-only in two ways:

- adding `.server` to the filename, e.g. `secrets.server.js`
- placing them in `$lib/server`, e.g. `$lib/server/secrets.js`

How it works

Any time you have public-facing code that imports server-only code (whether directly or indirectly)...

```
// @errors: 7005
/// file: $lib/server/secrets.js
export const atlantisCoordinates = [/* redacted */];
```

```
// @errors: 2307 7006 7005
/// file: src/routes/utils.js
export { atlantisCoordinates } from '$lib/server/secrets.js';

export const add = (a, b) => a + b;
```

```
/// file: src/routes/+page.svelte
<script>
    import { add } from './utils.js';
</script>
```

...SvelteKit will error:

```
Cannot import $lib/server/secrets.js into public-facing code:
- src/routes/+page.svelte
    - src/routes/utils.js
        - $lib/server/secrets.js
```

Even though the public-facing code — `src/routes/+page.svelte` — only uses the `add` export and not the secret `atlantisCoordinates` export, the secret code could end up in JavaScript that the browser downloads, and so the import chain is considered unsafe.

This feature also works with dynamic imports, even interpolated ones like `await import(`./${foo}.js`)`, with one small caveat: during development, if there are two or more dynamic imports between the public-facing code and the server-only module, the illegal import will not be detected the first time the code is loaded.

> Unit testing frameworks like Vitest do not distinguish between server-only and public-facing code. For this reason, illegal import detection is disabled when running tests, as determined by `process.env.TEST === 'true'`.

Further reading

- Tutorial: Environment variables

Go to TOC

Asset handling

Caching and inlining

Vite will automatically process imported assets for improved performance. Hashes will be added to the file-names so that they can be cached and assets smaller than `assetsInlineLimit` will be inlined.

```
<script>
    import logo from '$lib/assets/logo.png';
</script>

<img alt="The project logo" src={logo} />
```

If you prefer to reference assets directly in the markup, you can use a preprocessor such as svelte-preprocess-import-assets.

For assets included via the CSS `url()` function, you may find `vitePreprocess` useful.

Transforming

You may wish to transform your images to output compressed image formats such as `.webp` or `.avif`, responsive images with different sizes for different devices, or images with the EXIF data stripped for privacy. For images that are included statically, you may use a Vite plugin such as vite-imagetools. You may also consider a CDN, which can serve the appropriate transformed image based on the `Accept` HTTP header and query string parameters.

Go to TOC

Snapshots

Ephemeral DOM state — like scroll positions on sidebars, the content of `<input>` elements and so on — is discarded when you navigate from one page to another.

For example, if the user fills out a form but clicks a link before submitting, then hits the browser's back button, the values they filled in will be lost. In cases where it's valuable to preserve that input, you can take a *snapshot* of DOM state, which can then be restored if the user navigates back.

To do this, export a `snapshot` object with `capture` and `restore` methods from a `+page.svelte` or `+layout.svelte`:

```
<! file: +page.svelte --->
<script>
    let comment = '';

    /** @type {import('./$types').Snapshot<string>} */
    export const snapshot = {
        capture: () => comment,
        restore: (value) => comment = value
    };
</script>

<form method="POST">
    <label for="comment">Comment</label>
    <textarea id="comment" bind:value={comment} />
    <button>Post comment</button>
</form>
```

When you navigate away from this page, the `capture` function is called immediately before the page updates, and the returned value is associated with the current entry in the browser's history stack. If you navigate back, the `restore` function is called with the stored value as soon as the page is updated.

The data must be serializable as JSON so that it can be persisted to `sessionStorage`. This allows the state to be restored when the page is reloaded, or when the user navigates back from a different site.

> Avoid returning very large objects from `capture` — once captured, objects will be retained in memory for the duration of the session, and in extreme cases may be too large to persist to `sessionStorage`.

Go to TOC

Packaging

You can use SvelteKit to build apps as well as component libraries, using the `@sveltejs/package` package (`npm create svelte` has an option to set this up for you).

When you're creating an app, the contents of `src/routes` is the public-facing stuff; `src/lib` contains your app's internal library.

A component library has the exact same structure as a SvelteKit app, except that `src/lib` is the public-facing bit, and your root `package.json` is used to publish the package. `src/routes` might be a documentation or demo site that accompanies the library, or it might just be a sandbox you use during development.

Running the `svelte-package` command from `@sveltejs/package` will take the contents of `src/lib` and generate a `dist` directory (which can be configured) containing the following:

- All the files in `src/lib`. Svelte components will be preprocessed, TypeScript files will be transpiled to JavaScript.
- Type definitions (`d.ts` files) which are generated for Svelte, JavaScript and TypeScript files. You need to install `typescript >= 4.0.0` for this. Type definitions are placed next to their implementation, hand-written `d.ts` files are copied over as is. You can disable generation, but we strongly recommend against it — people using your library might use TypeScript, for which they require these type definition files.

> `@sveltejs/package` version 1 generated a `package.json`. This is no longer the case and it will now use the `package.json` from your project and validate that it is correct instead. If you're still on version 1, see this PR for migration instructions.

Anatomy of a package.json

Since you're now building a library for public use, the contents of your `package.json` will become more important. Through it, you configure the entry points of your package, which files are published to npm, and which dependencies your library has. Let's go through the most important fields one by one.

name

This is the name of your package. It will be available for others to install using that name, and visible on `https://npmjs.com/package/<name>`.

```
{
    "name": "your-library"
}
```

Read more about it here.

license

Every package should have a license field so people know how they are allowed to use it. A very popular license which is also very permissive in terms of distribution and reuse without warranty is `MIT`.

```
{
    "license": "MIT"
}
```

Read more about it here. Note that you should also include a `LICENSE` file in your package.

files

This tells npm which files it will pack up and upload to npm. It should contain your output folder (`dist` by default). Your `package.json` and `README` and `LICENSE` will always be included, so you don't need to specify them.

```
{
    "files": ["dist"]
}
```

To exclude unnecessary files (such as unit tests, or modules that are only imported from `src/routes` etc) you can add them to an `.npmignore` file. This will result in smaller packages that are faster to install.

Read more about it here.

exports

The `"exports"` field contains the package's entry points. If you set up a new library project through `npm create svelte@latest`, it's set to a single export, the package root:

```
{
    "exports": {
        ".": {
            "types": "./dist/index.d.ts",
            "svelte": "./dist/index.js"
        }
    }
}
```

This tells bundlers and tooling that your package only has one entry point, the root, and everything should be imported through that, like this:

```
// @errors: 2307
import { Something } from 'your-library';
```

The `types` and `svelte` keys are export conditions. They tell tooling what file to import when they look up the `your-library` import:

- TypeScript sees the `types` condition and looks up the type definition file. If you don't publish type definitions, omit this condition.
- Svelte-aware tooling sees the `svelte` condition and knows this is a Svelte component library. If you publish a library that does not export any Svelte components and that could also work in non-Svelte projects (for example a Svelte store library), you can replace this condition with `default`.

> Previous versions of `@sveltejs/package` also added a `package.json` export. This is no longer part of the template because all tooling can now deal with a `package.json` not being explicitly exported.

You can adjust `exports` to your liking and provide more entry points. For example, if instead of a `src/lib/index.js` file that re-exported components you wanted to expose a `src/lib/Foo.svelte` component directly, you could create the following export map...

```
{
    "exports": {
        "./Foo.svelte": {
            "types": "./dist/Foo.svelte.d.ts",
            "svelte": "./dist/Foo.svelte"
        }
    }
}
```

...and a consumer of your library could import the component like so:

```
// @filename: ambient.d.ts
declare module 'your-library/Foo.svelte';

// @filename: index.js
// cut---
import Foo from 'your-library/Foo.svelte';
```

> Beware that doing this will need additional care if you provide type definitions. Read more about the caveat here

In general, each key of the exports map is the path the user will have to use to import something from your package, and the value is the path to the file that will be imported or a map of export conditions which in turn contains these file paths.

Read more about `exports` here.

svelte

This is a legacy field that enabled tooling to recognise Svelte component libraries. It's no longer necessary when using the `svelte` export condition, but for backwards compatibility with outdated tooling that doesn't yet know about export conditions it's good to keep it around. It should point towards your root entry point.

```
{
    "svelte": "./dist/index.js"
}
```

TypeScript

You should ship type definitions for your library even if you don't use TypeScript yourself so that people who do get proper intellisense when using your library. `@sveltejs/package` makes the process of generating types mostly opaque to you. By default, when packaging your library, type definitions are auto-generated for JavaScript, TypeScript and Svelte files. All you need to ensure is that the `types` condition in the exports map points to the correct files. When initialising a library project through `npm create svelte@latest`, this is automatically setup for the root export.

If you have something else than a root export however — for example providing a `your-library/foo` import — you need to take additional care for providing type definitions. Unfortunately, TypeScript by default will *not* resolve the `types` condition for an export like `{ "./foo": { "types": "./dist/foo.d.ts", ... }}`. Instead, it will search for a `foo.d.ts` relative to the root of your library (i.e. `your-library/foo.d.ts` instead of `your-library/dist/foo.d.ts`). To fix this, you have two options:

The first option is to require people using your library to set the `moduleResolution` option in their `tsconfig.json` (or `jsconfig.json`) to `bundler` (available since TypeScript 5, the best and recommended option in the future), `node16` or `nodenext`. This opts TypeScript into actually looking at the exports map and resolving the types correctly.

The second option is to (ab)use the `typesVersions` feature from TypeScript to wire up the types. This is a field inside `package.json` TypeScript uses to check for different type definitions depending on the TypeScript version, and also contains a path mapping feature for that. We leverage that path mapping feature to get what we want. For the mentioned `foo` export above, the corresponding `typesVersions` looks like this:

```
{
    "exports": {
        "./foo": {
            "types": "./dist/foo.d.ts",
            "svelte": "./dist/foo.js"
        }
    },
    "typesVersions": {
        ">4.0": {
            "foo": ["./dist/foo.d.ts"]
        }
    }
}
```

`>4.0` tells TypeScript to check the inner map if the used TypeScript version is greater than 4 (which should in practice always be true). The inner map tells TypeScript that the typings for `your-library/foo` are found within `./dist/foo.d.ts`, which essentially replicates the `exports` condition. You also have `*` as a

wildcard at your disposal to make many type definitions at once available without repeating yourself. Note that if you opt into `typesVersions` you have to declare all type imports through it, including the root import (which is defined as `"index.d.ts": [..]`).

You can read more about that feature here.

Best practices

You should avoid using SvelteKit-specific modules like `$app` in your packages unless you intend for them to only be consumable by other SvelteKit projects. E.g. rather than using `import { browser } from '$app/environment'` you could use `import { BROWSER } from 'esm-env'` (see esm-env docs). You may also wish to pass in things like the current URL or a navigation action as a prop rather than relying directly on `$app/stores`, `$app/navigation`, etc. Writing your app in this more generic fashion will also make it easier to setup tools for testing, UI demos and so on.

Ensure that you add aliases via `svelte.config.js` (not `vite.config.js` or `tsconfig.json`), so that they are processed by `svelte-package`.

You should think carefully about whether or not the changes you make to your package are a bug fix, a new feature, or a breaking change, and update the package version accordingly. Note that if you remove any paths from `exports` or any `export` conditions inside them from your existing library, that should be regarded as a breaking change.

```
{
    "exports": {
        ".": {
            "types": "./dist/index.d.ts",
// changing `svelte` to `default` is a breaking change:
-           "svelte": "./dist/index.js"
+           "default": "./dist/index.js"
        },
// removing this is a breaking change:
-       "./foo": {
-           "types": "./dist/foo.d.ts",
-           "svelte": "./dist/foo.js",
-           "default": "./dist/foo.js"
-       },
// adding this is ok:
+       "./bar": {
+           "types": "./dist/bar.d.ts",
+           "svelte": "./dist/bar.js",
+           "default": "./dist/bar.js"
+       }
    }
}
```

Options

`svelte-package` accepts the following options:

- `-w` / `--watch` — watch files in `src/lib` for changes and rebuild the package
- `-i` / `--input` — the input directory which contains all the files of the package. Defaults to `src/lib`

- `-o` / `--o` — the output directory where the processed files are written to. Your `package.json`'s `exports` should point to files inside there, and the `files` array should include that folder. Defaults to `dist`
- `-t` / `--types` — whether or not to create type definitions (`d.ts` files). We strongly recommend doing this as it fosters ecosystem library quality. Defaults to `true`

Publishing

To publish the generated package:

```
npm publish
```

Caveats

All relative file imports need to be fully specified, adhering to Node's ESM algorithm. This means that for a file like `src/lib/something/index.js`, you must include the filename with the extension:

```
-import { something } from './something';
+import { something } from './something/index.js';
```

If you are using TypeScript, you need to import `.ts` files the same way, but using a `.js` file ending, *not* a `.ts` file ending. (This is a TypeScript design decision outside our control.) Setting `"moduleResolution": "NodeNext"` in your `tsconfig.json` or `jsconfig.json` will help you with this.

All files except Svelte files (preprocessed) and TypeScript files (transpiled to JavaScript) are copied across as-is.

Go to TOC

Accessibility

SvelteKit strives to provide an accessible platform for your app by default. Svelte's compile-time accessibility checks will also apply to any SvelteKit application you build.

Here's how SvelteKit's built-in accessibility features work and what you need to do to help these features to work as well as possible. Keep in mind that while SvelteKit provides an accessible foundation, you are still responsible for making sure your application code is accessible. If you're new to accessibility, see the "further reading" section of this guide for additional resources.

We recognize that accessibility can be hard to get right. If you want to suggest improvements to how SvelteKit handles accessibility, please open a GitHub issue.

Route announcements

In traditional server-rendered applications, every navigation (e.g. clicking on an `<a>` tag) triggers a full page reload. When this happens, screen readers and other assistive technology will read out the new page's title so that users understand that the page has changed.

Since navigation between pages in SvelteKit happens without reloading the page (known as client-side routing), SvelteKit injects a live region onto the page that will read out the new page name after each navigation. This determines the page name to announce by inspecting the `<title>` element.

Because of this behavior, every page in your app should have a unique, descriptive title. In SvelteKit, you can do this by placing a `<svelte:head>` element on each page:

```
<! file: src/routes/+page.svelte --->
<svelte:head>
    <title>Todo List</title>
</svelte:head>
```

This will allow screen readers and other assistive technology to identify the new page after a navigation occurs. Providing a descriptive title is also important for SEO.

Focus management

In traditional server-rendered applications, every navigation will reset focus to the top of the page. This ensures that people browsing the web with a keyboard or screen reader will start interacting with the page from the beginning.

To simulate this behavior during client-side routing, SvelteKit focuses the `<body>` element after each navigation and enhanced form submission. There is one exception - if an element with the `autofocus` attribute is present, SvelteKit will focus that element instead. Make sure to consider the implications for assistive technology when using that attribute.

If you want to customize SvelteKit's focus management, you can use the `afterNavigate` hook:

```
/// <reference types="@sveltejs/kit" />
// cut---
import { afterNavigate } from '$app/navigation';

afterNavigate(() => {
    /** @type {HTMLElement | null} */
    const to_focus = document.querySelector('.focus-me');
    to_focus?.focus();
});
```

You can also programmatically navigate to a different page using the `goto` function. By default, this will have the same client-side routing behavior as clicking on a link. However, `goto` also accepts a `keepFocus` option that will preserve the currently-focused element instead of resetting focus. If you enable this option, make sure the currently-focused element still exists on the page after navigation. If the element no longer exists, the user's focus will be lost, making for a confusing experience for assistive technology users.

The "lang" attribute

By default, SvelteKit's page template sets the default language of the document to English. If your content is not in English, you should update the `<html>` element in `src/app.html` to have the correct `lang` attribute. This will ensure that any assistive technology reading the document uses the correct pronunciation. For example, if your content is in German, you should update `app.html` to the following:

```
/// file: src/app.html
<html lang="de">
```

If your content is available in multiple languages, you should set the `lang` attribute based on the language of the current page. You can do this with SvelteKit's handle hook:

```
/// file: src/app.html
<html lang="%lang%">
```

```
/// file: src/hooks.server.js
/**
 * @param {import('@sveltejs/kit').RequestEvent} event
 */
function get_lang(event) {
    return 'en';
}
// cut---
/** @type {import('@sveltejs/kit').Handle} */
export function handle({ event, resolve }) {
    return resolve(event, {
        transformPageChunk: ({ html }) => html.replace('%lang%', get_lang(event))
    });
}
```

Further reading

For the most part, building an accessible SvelteKit app is the same as building an accessible web app. You should be able to apply information from the following general accessibility resources to any web experience you build:

- MDN Web Docs: Accessibility
- The A11y Project
- How to Meet WCAG (Quick Reference)

Go to TOC

SEO

The most important aspect of SEO is to create high-quality content that is widely linked to from around the web. However, there are a few technical considerations for building sites that rank well.

Out of the box

SSR

While search engines have got better in recent years at indexing content that was rendered with client-side JavaScript, server-side rendered content is indexed more frequently and reliably. SvelteKit employs SSR by default, and while you can disable it in `handle`, you should leave it on unless you have a good reason not to.

> SvelteKit's rendering is highly configurable and you can implement dynamic rendering if necessary. It's not generally recommended, since SSR has other benefits beyond SEO.

Performance

Signals such as Core Web Vitals impact search engine ranking. Because Svelte and SvelteKit introduce minimal overhead, it's easier to build high performance sites. You can test your site's performance using Google's PageSpeed Insights or Lighthouse.

Normalized URLs

SvelteKit redirects pathnames with trailing slashes to ones without (or vice versa depending on your configuration), as duplicate URLs are bad for SEO.

Manual setup

<title> and <meta>

Every page should have well-written and unique `<title>` and `<meta name="description">` elements inside a `<svelte:head>`. Guidance on how to write descriptive titles and descriptions, along with other suggestions on making content understandable by search engines, can be found on Google's Lighthouse SEO audits documentation.

> A common pattern is to return SEO-related `data` from page `load` functions, then use it (as `$page.data`) in a `<svelte:head>` in your root layout.

Structured data

Structured data helps search engines understand the content of a page. If you're using structured data alongside `svelte-preprocess`, you will need to explicitly preserve `ld+json` data (this may change in future):

```
/// file: svelte.config.js
// @filename: ambient.d.ts
declare module 'svelte-preprocess';

// @filename: index.js
// cut---
import preprocess from 'svelte-preprocess';

/** @type {import('@sveltejs/kit').Config} */
const config = {
    preprocess: preprocess({
        preserve: ['ld+json']
        // ...
    })
};

export default config;
```

Sitemaps

Sitemaps help search engines prioritize pages within your site, particularly when you have a large amount of content. You can create a sitemap dynamically using an endpoint:

```
/// file: src/routes/sitemap.xml/+server.js
export async function GET() {
    return new Response(
        `
        <?xml version="1.0" encoding="UTF-8" ?>
        <urlset
            xmlns="https://www.sitemaps.org/schemas/sitemap/0.9"
            xmlns:xhtml="https://www.w3.org/1999/xhtml"
            xmlns:mobile="https://www.google.com/schemas/sitemap-mobile/1.0"
            xmlns:news="https://www.google.com/schemas/sitemap-news/0.9"
            xmlns:image="https://www.google.com/schemas/sitemap-image/1.1"
            xmlns:video="https://www.google.com/schemas/sitemap-video/1.1"
        >
            <!-- <url> elements go here -->
        </urlset>`.trim(),
        {
            headers: {
                'Content-Type': 'application/xml'
            }
        }
    );
}
```

AMP

An unfortunate reality of modern web development is that it is sometimes necessary to create an Accelerated Mobile Pages (AMP) version of your site. In SvelteKit this can be done by setting the `inlineStyleThreshold` option...

```
/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        // since <link rel="stylesheet"> isn't
        // allowed, inline all styles
        inlineStyleThreshold: Infinity
    }
};

export default config;
```

...disabling `csr` in your root `+layout.js` / `+layout.server.js`...

```
/// file: src/routes/+layout.server.js
export const csr = false;
```

...adding `amp` to your `app.html`

```
<html amp>
...
```

...and transforming the HTML using `transformPageChunk` along with `transform` imported from `@sveltejs/amp`:

```
/// file: src/hooks.server.js
import * as amp from '@sveltejs/amp';

/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    let buffer = '';
    return await resolve(event, {
        transformPageChunk: ({ html, done }) => {
            buffer += html;
            if (done) return amp.transform(buffer);
        }
    });
}
```

To prevent shipping any unused CSS as a result of transforming the page to amp, we can use `dropcss`:

```
/// file: src/hooks.server.js
// @errors: 2307
import * as amp from '@sveltejs/amp';
import dropcss from 'dropcss';

/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    let buffer = '';
```

```
    return await resolve(event, {
        transformPageChunk: ({ html, done }) => {
            buffer += html;

            if (done) {
                let css = '';
                const markup = amp
                    .transform(buffer)
                    .replace('⚡', 'amp') // dropcss can't handle this character
                    .replace(/<style amp-custom([^>]*?)>([^]+?)<\/style>/, (match,
attributes, contents) => {
                        css = contents;
                        return `<style amp-custom${attributes}></style>`;
                    });

                css = dropcss({ css, html: markup }).css;
                return markup.replace('</style>', `${css}</style>`);
            }
        }
    });
}
```

It's a good idea to use the `handle` hook to validate the transformed HTML using `amphtml-validator`, but only if you're prerendering pages since it's very slow.

Go to TOC

Configuration

Your project's configuration lives in a `svelte.config.js` file at the root of your project. As well as SvelteKit, this config object is used by other tooling that integrates with Svelte such as editor extensions.

```
/// file: svelte.config.js
// @filename: ambient.d.ts
declare module '@sveltejs/adapter-auto' {
    const plugin: () => import('@sveltejs/kit').Adapter;
    export default plugin;
}

// @filename: index.js
// cut---
import adapter from '@sveltejs/adapter-auto';

/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        adapter: adapter()
    }
};

export default config;
```

TYPES: @sveltejs/kit#Config

The `kit` property configures SvelteKit, and can have the following properties:

EXPANDED_TYPES: @sveltejs/kit#KitConfig

Go to TOC

Command Line Interface

SvelteKit projects use Vite, meaning you'll mostly use its CLI (albeit via `npm run dev/build/preview` scripts):

- `vite dev` — start a development server
- `vite build` — build a production version of your app
- `vite preview` — run the production version locally

However SvelteKit includes its own CLI for initialising your project:

svelte-kit sync

`svelte-kit sync` creates the `tsconfig.json` and all generated types (which you can import as `./$types` inside routing files) for your project. When you create a new project, it is listed as the `prepare` script and will be run automatically as part of the npm lifecycle, so you should not ordinarily have to run this command.

Go to TOC

Modules

SvelteKit makes a number of modules available to your application.

MODULES

Go to TOC

Types

Public types

The following types can be imported from `@sveltejs/kit`:

TYPES: @sveltejs/kit

Private types

The following are referenced by the public types documented above, but cannot be imported directly:

TYPES: Private types

Generated types

The `RequestHandler` and `Load` types both accept a `Params` argument allowing you to type the `params` object. For example this endpoint expects `foo`, `bar` and `baz` params:

```
/// file: src/routes/[foo]/[bar]/[baz]/+page.server.js
// @errors: 2355 2322 1360
/** @type {import('@sveltejs/kit').RequestHandler<{
    foo: string;
    bar: string;
    baz: string
  }>} */
export async function GET({ params }) {
    // ...
}
```

Needless to say, this is cumbersome to write out, and less portable (if you were to rename the `[foo]` directory to `[qux]`, the type would no longer reflect reality).

To solve this problem, SvelteKit generates `.d.ts` files for each of your endpoints and pages:

```
/// file: .svelte-kit/types/src/routes/[foo]/[bar]/[baz]/$types.d.ts
/// link: false
import type * as Kit from '@sveltejs/kit';

type RouteParams = {
    foo: string;
    bar: string;
    baz: string;
}
```

```
export type PageServerLoad = Kit.ServerLoad<RouteParams>;
export type PageLoad = Kit.Load<RouteParams>;
```

These files can be imported into your endpoints and pages as siblings, thanks to the `rootDirs` option in your TypeScript configuration:

```
/// file: src/routes/[foo]/[bar]/[baz]/+page.server.js
// @filename: $types.d.ts
import type * as Kit from '@sveltejs/kit';

type RouteParams = {
    foo: string;
    bar: string;
    baz: string;
}

export type PageServerLoad = Kit.ServerLoad<RouteParams>;

// @filename: index.js
// @errors: 2355
// cut---
/** @type {import('./$types').PageServerLoad} */
export async function GET({ params }) {
    // ...
}
```

```
/// file: src/routes/[foo]/[bar]/[baz]/+page.js
// @filename: $types.d.ts
import type * as Kit from '@sveltejs/kit';

type RouteParams = {
    foo: string;
    bar: string;
    baz: string;
}

export type PageLoad = Kit.Load<RouteParams>;

// @filename: index.js
// @errors: 2355
// cut---
/** @type {import('./$types').PageLoad} */
export async function load({ params, fetch }) {
    // ...
}
```

For this to work, your own `tsconfig.json` or `jsconfig.json` should extend from the generated `.svelte-kit/tsconfig.json` (where `.svelte-kit` is your `outDir`):

```
{ "extends": "./.svelte-kit/tsconfig.json" }
```

Default tsconfig.json

The generated `.svelte-kit/tsconfig.json` file contains a mixture of options. Some are generated programmatically based on your project configuration, and should generally not be overridden without good reason:

```
/// file: .svelte-kit/tsconfig.json
{
	"compilerOptions": {
		"baseUrl": "..",
		"paths": {
			"$lib": "src/lib",
			"$lib/*": "src/lib/*"
		},
		"rootDirs": ["..", "./types"]
	},
	"include": ["../src/**/*.js", "../src/**/*.ts", "../src/**/*.svelte"],
	"exclude": ["../node_modules/**", "./**"]
}
```

Others are required for SvelteKit to work properly, and should also be left untouched unless you know what you're doing:

```
/// file: .svelte-kit/tsconfig.json
{
	"compilerOptions": {
		// this ensures that types are explicitly
		// imported with `import type`, which is
		// necessary as svelte-preprocess cannot
		// otherwise compile components correctly
		"importsNotUsedAsValues": "error",

		// Vite compiles one TypeScript module
		// at a time, rather than compiling
		// the entire module graph
		"isolatedModules": true,

		// TypeScript cannot 'see' when you
		// use an imported value in your
		// markup, so we need this
		"preserveValueImports": true,

		// This ensures both `vite build`
		// and `svelte-package` work correctly
		"lib": ["esnext", "DOM", "DOM.Iterable"],
		"moduleResolution": "node",
		"module": "esnext",
		"target": "esnext"
	}
}
```

App

TYPES: App

Go to TOC

Frequently asked questions

Other resources

Please see the Svelte FAQ and `vite-plugin-svelte` FAQ as well for the answers to questions deriving from those libraries.

What can I make with SvelteKit?

SvelteKit can be used to create most kinds of applications. Out of the box, SvelteKit supports many features including:

- Dynamic page content with load functions and API routes.
- SEO-friendly dynamic content with server-side rendering (SSR).
- User-friendly progressively-enhanced interactive pages with SSR and Form Actions.
- Static pages with prerendering.

SvelteKit can also be deployed to a wide spectrum of hosted architectures via adapters. In cases where SSR is used (or server-side logic is added without prerendering), those functions will be adapted to the target backend. Some examples include:

- Self-hosted dynamic web applications with a Node.js backend.
- Serverless web applications with backend loaders and APIs deployed as remote functions. See zero-config deployments for popular deployment options.
- Static pre-rendered sites such as a blog or multi-page site hosted on a CDN or static host. Statically-generated sites are shipped without a backend.
- Single-page Applications (SPAs) with client-side routing and rendering for API-driven dynamic content. SPAs are shipped without a backend and are not server-rendered. This option is commonly chosen when bundling SvelteKit with an app written in PHP, .Net, Java, C, Golang, Rust, etc.
- A mix of the above; some routes can be static, and some routes can use backend functions to fetch dynamic information. This can be configured with page options that includes the option to opt out of SSR.

In order to support SSR, a JS backend — such as Node.js or Deno-based server, serverless function, or edge function — is required.

It is also possible to write custom adapters or leverage community adapters to deploy SvelteKit to more platforms such as specialized server environments, browser extensions, or native applications. See integrations for more examples and integrations.

How do I use HMR with SvelteKit?

SvelteKit has HMR enabled by default powered by svelte-hmr. If you saw Rich's presentation at the 2020 Svelte Summit, you may have seen a more powerful-looking version of HMR presented. This demo had `svelte-hmr`'s `preserveLocalState` flag on. This flag is now off by default because it may lead to unex-

pected behaviour and edge cases. But don't worry, you are still getting HMR with SvelteKit! If you'd like to preserve local state you can use the `@hmr:keep` or `@hmr:keep-all` directives as documented on the svelte-hmr page.

How do I include details from package.json in my application?

You cannot directly require JSON files, since SvelteKit expects `svelte.config.js` to be an ES module. If you'd like to include your application's version number or other information from `package.json` in your application, you can load JSON like so:

```
/// file: svelte.config.js
// @filename: index.js
/// <reference types="@types/node" />
import { URL } from 'node:url';
// cut---
import { readFileSync } from 'node:fs';
import { fileURLToPath } from 'node:url';

const path = fileURLToPath(new URL('package.json', import.meta.url));
const pkg = JSON.parse(readFileSync(path, 'utf8'));
```

How do I fix the error I'm getting trying to include a package?

Most issues related to including a library are due to incorrect packaging. You can check if a library's packaging is compatible with Node.js by entering it into the publint website.

Here are a few things to keep in mind when checking if a library is packaged correctly:

- `exports` takes precedence over the other entry point fields such as `main` and `module`. Adding an `exports` field may not be backwards-compatible as it prevents deep imports.
- ESM files should end with `.mjs` unless `"type": "module"` is set in which any case CommonJS files should end with `.cjs`.
- `main` should be defined if `exports` is not. It should be either a CommonJS or ESM file and adhere to the previous bullet. If a `module` field is defined, it should refer to an ESM file.
- Svelte components should be distributed as uncompiled `.svelte` files with any JS in the package written as ESM only. Custom script and style languages, like TypeScript and SCSS, should be preprocessed as vanilla JS and CSS respectively. We recommend using `svelte-package` for packaging Svelte libraries, which will do this for you.

Libraries work best in the browser with Vite when they distribute an ESM version, especially if they are dependencies of a Svelte component library. You may wish to suggest to library authors that they provide an ESM version. However, CommonJS (CJS) dependencies should work as well since, by default, `vite-plugin-svelte` will ask Vite to pre-bundle them using `esbuild` to convert them to ESM.

If you are still encountering issues we recommend searching both the Vite issue tracker and the issue tracker of the library in question. Sometimes issues can be worked around by fiddling with the `optimizeDeps` or `ssr` config values though we recommend this as only a short-term workaround in favor of fixing the library in question.

How do I use X with SvelteKit?

Make sure you've read the documentation section on integrations. If you're still having trouble, solutions to common issues are listed below.

How do I setup a database?

Put the code to query your database in a server route - don't query the database in .svelte files. You can create a `db.js` or similar that sets up a connection immediately and makes the client accessible throughout the app as a singleton. You can execute any one-time setup code in `hooks.js` and import your database helpers into any endpoint that needs them.

How do I use a client-side only library that depends on `document` or `window`?

If you need access to the `document` or `window` variables or otherwise need code to run only on the client-side you can wrap it in a `browser` check:

```
/// <reference types="@sveltejs/kit" />
// cut---
import { browser } from '$app/environment';

if (browser) {
    // client-only code here
}
```

You can also run code in `onMount` if you'd like to run it after the component has been first rendered to the DOM:

```
// @filename: ambient.d.ts
// @lib: ES2015
declare module 'some-browser-only-library';

// @filename: index.js
// cut---
import { onMount } from 'svelte';

onMount(async () => {
    const { method } = await import('some-browser-only-library');
    method('hello world');
});
```

If the library you'd like to use is side-effect free you can also statically import it and it will be tree-shaken out in the server-side build where `onMount` will be automatically replaced with a no-op:

```
// @filename: ambient.d.ts
// @lib: ES2015
declare module 'some-browser-only-library';

// @filename: index.js
// cut---
import { onMount } from 'svelte';
import { method } from 'some-browser-only-library';

onMount(() => {
    method('hello world');
});
```

Otherwise, if the library has side effects and you'd still prefer to use static imports, check out vite-plugin-iso-import to support the `?client` import suffix. The import will be stripped out in SSR builds. However, note that you will lose the ability to use VS Code Intellisense if you use this method.

```
// @filename: ambient.d.ts
// @lib: ES2015
declare module 'some-browser-only-library?client';

// @filename: index.js
// cut---
import { onMount } from 'svelte';
import { method } from 'some-browser-only-library?client';

onMount(() => {
    method('hello world');
});
```

How do I use a different backend API server?

You can use `event.fetch` to request data from an external API server, but be aware that you would need to deal with CORS, which will result in complications such as generally requiring requests to be preflighted resulting in higher latency. Requests to a separate subdomain may also increase latency due to an additional DNS lookup, TLS setup, etc. If you wish to use this method, you may find `handleFetch` helpful.

Another approach is to set up a proxy to bypass CORS headaches. In production, you would rewrite a path like `/api` to the API server; for local development, use Vite's `server.proxy` option.

How to setup rewrites in production will depend on your deployment platform. If rewrites aren't an option, you could alternatively add an API route:

```
/// file: src/routes/api/[...path]/+server.js
/** @type {import('./$types').RequestHandler} */
export function GET({ params, url }) {
    return fetch(`https://my-api-server.com/${params.path + url.search}`);
}
```

(Note that you may also need to proxy `POST`/`PATCH` etc requests, and forward `request.headers`, depending on your needs.)

How do I use middleware?

`adapter-node` builds a middleware that you can use with your own server for production mode. In dev, you can add middleware to Vite by using a Vite plugin. For example:

```
// @errors: 2322
// @filename: ambient.d.ts
declare module '@sveltejs/kit/vite'; // TODO this feels unnecessary, why can't it 'see' the declarations?

// @filename: index.js
// cut---
import { sveltekit } from '@sveltejs/kit/vite';

/** @type {import('vite').Plugin} */
const myPlugin = {
    name: 'log-request-middleware',
    configureServer(server) {
        server.middlewares.use((req, res, next) => {
            console.log(`Got request ${req.url}`);
            next();
        });
    }
};

/** @type {import('vite').UserConfig} */
const config = {
    plugins: [myPlugin, sveltekit()]
};

export default config;
```

See Vite's `configureServer` docs for more details including how to control ordering.

Does it work with Yarn 2?

Sort of. The Plug'n'Play feature, aka 'pnp', is broken (it deviates from the Node module resolution algorithm, and doesn't yet work with native JavaScript modules which SvelteKit — along with an increasing number of packages — uses). You can use `nodeLinker: 'node-modules'` in your `.yarnrc.yml` file to disable pnp, but it's probably easier to just use npm or pnpm, which is similarly fast and efficient but without the compatibility headaches.

How do I use with Yarn 3?

Currently ESM Support within the latest Yarn (version 3) is considered experimental.

The below seems to work although your results may vary.

First create a new application:

```
yarn create svelte myapp
cd myapp
```

And enable Yarn Berry:

```
yarn set version berry
yarn install
```

Yarn 3 global cache

One of the more interesting features of Yarn Berry is the ability to have a single global cache for packages, instead of having multiple copies for each project on the disk. However, setting `enableGlobalCache` to true causes building to fail, so it is recommended to add the following to the `.yarnrc.yml` file:

```
nodeLinker: node-modules
```

This will cause packages to be downloaded into a local node_modules directory but avoids the above problem and is your best bet for using version 3 of Yarn at this point in time.

Go to TOC

Integrations

Preprocessors

Preprocessors transform your `.svelte` files before passing them to the compiler. For example, if your `.svelte` file uses TypeScript and PostCSS, it must first be transformed into JavaScript and CSS so that the Svelte compiler can handle it. There are many available preprocessors. The Svelte team maintains two official ones discussed below.

`vitePreprocess`

`vite-plugin-svelte` offers a `vitePreprocess` feature which utilizes Vite for preprocessing. It is capable of handling the language flavors Vite handles: TypeScript, PostCSS, SCSS, Less, Stylus, and SugarSS. For convenience, it is re-exported from the `@sveltejs/kit/vite` package. If you set your project up with TypeScript it will be included by default:

```
// svelte.config.js
import { vitePreprocess } from '@sveltejs/kit/vite';

export default {
  preprocess: [vitePreprocess()]
};
```

`svelte-preprocess`

`svelte-preprocess` has some additional functionality not found in `vitePreprocess` such as support for Pug, Babel, and global styles. However, `vitePreprocess` may be faster and require less configuration, so it is used by default. Note that CoffeeScript is not supported by SvelteKit.

You will need to install `svelte-preprocess` with `npm install --save-dev svelte-preprocess` and add it to your `svelte.config.js`. After that, you will often need to install the corresponding library such as `npm install -D sass` or `npm install -D less`.

Adders

Svelte Adders allow you to setup many different complex integrations like Tailwind, PostCSS, Storybook, Firebase, GraphQL, mdsvex, and more with a single command. Please see sveltesociety.dev for a full listing of templates, components, and tools available for use with Svelte and SvelteKit.

Integration FAQs

The SvelteKit FAQ has a how to do X with SvelteKit, which may be helpful if you still have questions.

Go to TOC

Migrating from Sapper

rank: 1

SvelteKit is the successor to Sapper and shares many elements of its design.

If you have an existing Sapper app that you plan to migrate to SvelteKit, there are a number of changes you will need to make. You may find it helpful to view some examples while migrating.

package.json

type: "module"

Add `"type": "module"` to your `package.json`. You can do this step separately from the rest as part of an incremental migration if you are using Sapper 0.29.3 or newer.

dependencies

Remove `polka` or `express`, if you're using one of those, and any middleware such as `sirv` or `compression`.

devDependencies

Remove `sapper` from your `devDependencies` and replace it with `@sveltejs/kit` and whichever adapter you plan to use (see next section).

scripts

Any scripts that reference `sapper` should be updated:

- `sapper build` should become `vite build` using the Node adapter
- `sapper export` should become `vite build` using the static adapter
- `sapper dev` should become `vite dev`
- `node __sapper__/build` should become `node build`

Project files

The bulk of your app, in `src/routes`, can be left where it is, but several project files will need to be moved or updated.

Configuration

Your `webpack.config.js` or `rollup.config.js` should be replaced with a `svelte.config.js`, as documented here. Svelte preprocessor options should be moved to `config.preprocess`.

You will need to add an adapter. `sapper build` is roughly equivalent to adapter-node while `sapper export` is roughly equivalent to adapter-static, though you might prefer to use an adapter designed for the platform you're deploying to.

If you were using plugins for filetypes that are not automatically handled by Vite, you will need to find Vite equivalents and add them to the Vite config.

src/client.js

This file has no equivalent in SvelteKit. Any custom logic (beyond `sapper.start(...)`) should be expressed in your `+layout.svelte` file, inside an `onMount` callback.

src/server.js

When using `adapter-node` the equivalent is a custom server. Otherwise, this file has no direct equivalent, since SvelteKit apps can run in serverless environments.

src/service-worker.js

Most imports from `@sapper/service-worker` have equivalents in `$service-worker`:

- `files` is unchanged
- `routes` has been removed
- `shell` is now `build`
- `timestamp` is now `version`

src/template.html

The `src/template.html` file should be renamed `src/app.html`.

Remove `%sapper.base%`, `%sapper.scripts%` and `%sapper.styles%`. Replace `%sapper.head%` with `%sveltekit.head%` and `%sapper.html%` with `%sveltekit.body%`. The `<div id="sapper">` is no longer necessary.

src/node_modules

A common pattern in Sapper apps is to put your internal library in a directory inside `src/node_modules`. This doesn't work with Vite, so we use `src/lib` instead.

Pages and layouts

Renamed files

Routes now are made up of the folder name exclusively to remove ambiguity, the folder names leading up to a `+page.svelte` correspond to the route. See the routing docs for an overview. The following shows a old/new comparison:

Old	New
routes/about/index.svelte	routes/about/+page.svelte
routes/about.svelte	routes/about/+page.svelte

Your custom error page component should be renamed from `_error.svelte` to `+error.svelte`. Any `_layout.svelte` files should likewise be renamed `+layout.svelte`. Any other files are ignored.

Imports

The `goto`, `prefetch` and `prefetchRoutes` imports from `@sapper/app` should be replaced with `goto`, `preloadData` and `preloadCode` imports respectively from `$app/navigation`.

The `stores` import from `@sapper/app` should be replaced — see the Stores section below.

Any files you previously imported from directories in `src/node_modules` will need to be replaced with `$lib` imports.

Preload

As before, pages and layouts can export a function that allows data to be loaded before rendering takes place.

This function has been renamed from `preload` to `load`, it now lives in a `+page.js` (or `+layout.js`) next to its `+page.svelte` (or `+layout.svelte`), and its API has changed. Instead of two arguments — `page` and `session` — there is a single `event` argument.

There is no more `this` object, and consequently no `this.fetch`, `this.error` or `this.redirect`. Instead, you can get `fetch` from the input methods, and both `error` and `redirect` are now thrown.

Stores

In Sapper, you would get references to provided stores like so:

```
// @filename: ambient.d.ts
declare module '@sapper/app';

// @filename: index.js
```

```
// cut---
import { stores } from '@sapper/app';
const { preloading, page, session } = stores();
```

The `page` store still exists; `preloading` has been replaced with a `navigating` store that contains `from` and `to` properties. `page` now has `url` and `params` properties, but no `path` or `query`.

You access them differently in SvelteKit. `stores` is now `getStores`, but in most cases it is unnecessary since you can import `navigating`, and `page` directly from `$app/stores`.

Routing

Regex routes are no longer supported. Instead, use advanced route matching.

Segments

Previously, layout components received a `segment` prop indicating the child segment. This has been removed; you should use the more flexible `$page.url.pathname` value to derive the segment you're interested in.

URLs

In Sapper, all relative URLs were resolved against the base URL — usually `/`, unless the `basepath` option was used — rather than against the current page.

This caused problems and is no longer the case in SvelteKit. Instead, relative URLs are resolved against the current page (or the destination page, for `fetch` URLs in `load` functions) instead. In most cases, it's easier to use root-relative (i.e. starts with `/`) URLs, since their meaning is not context-dependent.

<a> attributes

- `sapper:prefetch` is now `data-sveltekit-preload-data`
- `sapper:noscroll` is now `data-sveltekit-noscroll`

Endpoints

In Sapper, server routes received the `req` and `res` objects exposed by Node's `http` module (or the augmented versions provided by frameworks like Polka and Express).

SvelteKit is designed to be agnostic as to where the app is running — it could be running on a Node server, but could equally be running on a serverless platform or in a Cloudflare Worker. For that reason, you no longer interact directly with `req` and `res`. Your endpoints will need to be updated to match the new signature.

To support this environment-agnostic behavior, `fetch` is now available in the global context, so you don't need to import `node-fetch`, `cross-fetch`, or similar server-side fetch implementations in order to use it.

Integrations

See integrations for detailed information about integrations.

HTML minifier

Sapper includes `html-minifier` by default. SvelteKit does not include this, but you can add it as a prod dependency and then use it through a hook:

```
// @filename: ambient.d.ts
/// <reference types="@sveltejs/kit" />
declare module 'html-minifier';

// @filename: index.js
// cut---
import { minify } from 'html-minifier';
import { building } from '$app/environment';

const minification_options = {
    collapseBooleanAttributes: true,
    collapseWhitespace: true,
    conservativeCollapse: true,
    decodeEntities: true,
    html5: true,
    ignoreCustomComments: [/^#/],
    minifyCSS: true,
    minifyJS: false,
    removeAttributeQuotes: true,
    removeComments: false, // some hydration code needs comments, so leave them in
    removeOptionalTags: true,
    removeRedundantAttributes: true,
    removeScriptTypeAttributes: true,
    removeStyleLinkTypeAttributes: true,
    sortAttributes: true,
    sortClassName: true
};

/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    let page = '';

    return resolve(event, {
        transformPageChunk: ({ html, done }) => {
            page += html;
            if (done) {
                return building ? minify(page, minification_options) : page;
            }
        }
    });
}
```

Note that `prerendering` is `false` when using `vite preview` to test the production build of the site, so to verify the results of minifying, you'll need to inspect the built HTML files directly.

Go to TOC

Additional resources

FAQs

Please see the SvelteKit FAQ for solutions to common issues and helpful tips and tricks.

The Svelte FAQ and `vite-plugin-svelte` FAQ may also be helpful for questions deriving from those libraries.

Examples

We've written and published a few different SvelteKit sites as examples:

- `sveltejs/realworld` contains an example blog site
- The `sites/kit.svelte.dev` directory contains the code for this site
- `sveltejs/sites` contains the code for svelte.dev and for a HackerNews clone

SvelteKit users have also published plenty of examples on GitHub, under the #sveltekit and #sveltekit-template topics, as well as on the Svelte Society site. Note that these have not been vetted by the maintainers and may not be up to date.

Support

You can ask for help on Discord and StackOverflow. Please first search for information related to your issue in the FAQ, Google or another search engine, issue tracker, and Discord chat history in order to be respectful of others' time. There are many more people asking questions than answering them, so this will help in allowing the community to grow in a scalable fashion.

Go to TOC

Glossary

The core of SvelteKit provides a highly configurable rendering engine. This section describes some of the terms used when discussing rendering. A reference for setting these options is provided in the documentation above.

CSR

Client-side rendering (CSR) is the generation of the page contents in the web browser using JavaScript.

In SvelteKit, client-side rendering will be used by default, but you can turn off JavaScript with the `csr = false` page option.

Hydration

Svelte components store some state and update the DOM when the state is updated. When fetching data during SSR, by default SvelteKit will store this data and transmit it to the client along with the server-rendered HTML. The components can then be initialized on the client with that data without having to call the same API endpoints again. Svelte will then check that the DOM is in the expected state and attach event listeners in a process called hydration. Once the components are fully hydrated, they can react to changes to their properties just like any newly created Svelte component.

In SvelteKit, pages will be hydrated by default, but you can turn off JavaScript with the `csr = false` page option.

Prerendering

Prerendering means computing the contents of a page at build time and saving the HTML for display. This approach has the same benefits as traditional server-rendered pages, but avoids recomputing the page for each visitor and so scales nearly for free as the number of visitors increases. The tradeoff is that the build process is more expensive and prerendered content can only be updated by building and deploying a new version of the application.

Not all pages can be prerendered. The basic rule is this: for content to be prerenderable, any two users hitting it directly must get the same content from the server, and the page must not contain actions. Note that you can still prerender content that is loaded based on the page's parameters as long as all users will be seeing the same prerendered content.

Pre-rendered pages are not limited to static content. You can build personalized pages if user-specific data is fetched and rendered client-side. This is subject to the caveat that you will experience the downsides of not doing SSR for that content as discussed above.

In SvelteKit, you can control prerendering with the `prerender` page option and `prerender` config in `svelte.config.js`.

Routing

By default, when you navigate to a new page (by clicking on a link or using the browser's forward or back buttons), SvelteKit will intercept the attempted navigation and handle it instead of allowing the browser to send a request to the server for the destination page. SvelteKit will then update the displayed contents on the client by rendering the component for the new page, which in turn can make calls to the necessary API endpoints. This process of updating the page on the client in response to attempted navigation is called client-side routing.

In SvelteKit, client-side routing will be used by default, but you can skip it with `data-sveltekit-reload`.

SPA

A single-page app (SPA) is an application in which all requests to the server load a single HTML file which then does client-side rendering of the requested contents based on the requested URL. All navigation is handled on the client-side in a process called client-side routing with per-page contents being updated and common layout elements remaining largely unchanged. SPAs do not provide SSR, which has the shortcoming described above. However, some applications are not greatly impacted by these shortcomings such as a complex business application behind a login where SEO would not be important and it is known that users will be accessing the application from a consistent computing environment.

In SvelteKit, you can build an SPA with `adapter-static`.

SSG

Static Site Generation (SSG) is a term that refers to a site where every page is prerendered. SvelteKit was not built to do only static site generation like some tools and so may not scale as well to efficiently render a very large number of pages as tools built specifically for that purpose. However, in contrast to most purpose-built SSGs, SvelteKit does nicely allow for mixing and matching different rendering types on different pages. One benefit of fully prerendering a site is that you do not need to maintain or pay for servers to perform SSR. Once generated, the site can be served from CDNs, leading to great "time to first byte" performance. This delivery model is often referred to as JAMstack.

In SvelteKit, you can do static site generation by using `adapter-static` or by configuring every page to be prerendered using the `prerender` page option or `prerender` config in `svelte.config.js`.

SSR

Server-side rendering (SSR) is the generation of the page contents on the server. SSR is generally preferred for SEO. While some search engines can index content that is dynamically generated on the client-side it may take longer even in these cases. It also tends to improve perceived performance and makes your app accessible to users if JavaScript fails or is disabled (which happens more often than you probably think).

In SvelteKit, pages are server-side rendered by default. You can disable SSR with the `ssr` page option.

Go to TOC

Colophon

This book is created by using the following sources:

- Sveltekit - English
- GitHub source: sveltejs/kit/documentation
- Created: 2023-08-25
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- Viewer: https://read-html-download-pdf.vercel.app/
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf
- Viewer repo: https://github.com/shinokada/read-html-download-pdf