Mastering SvelteKit: A Comprehensive Guide

This book provides an in-depth exploration of SvelteKit, a powerful framework for building fast, dynamic web applications. Covering core concepts, advanced techniques, and best practices, it is designed to help developers of all levels become proficient in leveraging SvelteKit for their projects.

Table of Contents

Getting Started

Core Concepts

Build and Deploy

Advanced

SvelteKit Documentation

Introduction

On this page

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 documentation regarding project types.

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 helps you build web apps while following modern best practices and providing solutions to common development challenges. It offers everything from basic functionalities — like a router that updates your UI when a link is clicked — to more advanced capabilities. Its extensive list of features includes build optimizations to load only the minimal required code; offline support; preloading pages before user navigation; configurable rendering to handle different parts of your app on the server via SSR, in the browser through client-side rendering, or at build-time with prerendering; image optimization; and much more. Building an app with all the modern best practices is fiendishly complicated, but 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).

Edit this page on GitHub
llms.txt

Navigation

- Creating a project

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Project structure • Docs • Svelte

Project structure

On this page

A typical SvelteKit project looks like this:

```plaintext
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
| └ service-worker.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 `eslint.config.js` and so on, if you chose those options when running `npx sv create`).

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.)

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 `npx sv create`, 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`) configures TypeScript, if you added typechecking during `npx sv create`. 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`).

Project types Web standards

Web standards

On this page

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 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`:

```
import { json } from '@sveltejs/kit';

export function GET({ request }) {
  // retrieve a specific header
  const userAgent = request.headers.get('user-agent');

  // set a header on the response
  return json({ userAgent }, {
    headers: {
      'x-custom-header': 'potato'
    }
  });
}
```

FormData

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

```
import { json } from '@sveltejs/kit';

export async function POST({ request }) {
  const body = await request.formData();

  // log all fields
  // (see console usage below)
  console.log(...body);
  // create a JSON Response using a header we received
  return json({ 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:

```
const foo = (new URL('https://example.com?foo=bar')).searchParams.get('foo'); // 'bar'
```

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:

```
const uuid = `${crypto.randomUUID()}-${crypto.randomUUID()}-${crypto.randomUUID()}-${c
```

Available only in secure contexts.

Edit this page on GitHub | llms.txt

Previous | Next

- Project structure
- Routing

Hello

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

SvelteKit - Core Concepts: Page options

Page options

By default, SvelteKit renders each component on the server first and sends it as HTML, then re-renders it in the browser for interactivity (*hydration*). Components must be able to run in both environments. SvelteKit initializes a *router* for subsequent navigations.

You can control these options per page by exporting from `+page.js` or `+page.server.js`, or for groups via shared `+layout.js` or `+layout.server.js`. To set app-wide defaults, export from the root layout.

These options can be mixed and matched—e.g., prerender marketing pages, server-render dynamic pages for SEO, or turn admin sections into SPAs.

prerender

Routes that can be generated as static HTML at build time are *prerendered*.

Set prerender flag

```
export const prerender: true;
```

Prerender with auto

Set `prerender` to `'auto'` to prerender selectively while allowing dynamic or long-tail pages to be server-rendered:

```
export const prerender: 'auto';
```

Prerender server routes

`+server.js` files inherit the `prerender` setting. If a route (or a parent page) exports `prerender = true`, it is prerendered unless explicitly set to false.

When not to prerender

Prerendering requires identical content for all users. Personalized or dynamic content based on `url.searchParams` or non-static data is unsuitable. Actions requiring server POST handling also cannot be prerendered.

Route conflicts

Prerendering writes files; having both a directory and a file with the same name causes conflicts (`foo/` vs `foo.json`). Using file extensions (e.g., `foo.json`) avoids this. Static pages use `index.html`.

Troubleshooting

Routes marked as prerenderable but not crawled will cause errors. Fix by ensuring links to the route are discoverable or explicitly listed in `config.kit.prerender.entries`. Routes with `export const prerender = true` but not reached won't be prerendered.

entries

SvelteKit discovers pages to prerender starting from entry points. Non-dynamic routes are default entry points, e.g., `/`, `/blog`.

You can specify explicit entries, especially for dynamic routes, with an `entries` function:

```
export function entries() {
  return [
    { slug: 'hello-world' },
    { slug: 'another-blog-post' }
  ];
}
export const prerender: true;
```

This function can be async, e.g., fetching post slugs from a CMS.

ssr

Server-side rendering (*SSR*) renders pages on the server before sending to the client. Set `ssr = false` to disable SSR and render only an empty shell, useful for purely client-side pages.

```
export const ssr: false;
```

Note: If `ssr` and `csr` are both false, nothing is rendered.

csr

Client-side rendering (*CSR*) hydrates server-rendered HTML into an interactive page. Setting `csr = false` disables CSR, making pages purely static HTML:

```
export const csr: false;
```

Disabling CSR:

- Removes widget JavaScript
- Web page works with HTML and CSS only
- Links perform full page reload
- HMR is disabled during development

Enable during development:

```
import { dev } from '$app/environment';

export const csr: boolean = dev;
```

trailingSlash

Controls trailing slash behavior in URLs, affecting file generation and SEO.

Options:

- `'never'` (default): URLs like `/about` become `/about.html`
- `'always'`: `/about` redirects to `/about/` and creates `/about/index.html`
- `'ignore'`: preserves URL as-is

Set in `+layout.js`:

```
export const trailingSlash = 'always'; // or 'never' / 'ignore'
```

config

Configuration object for deployment platforms; merges per-route configurations.

Example:

```
/** @type {import('some-adapter').Config} */
export const config = {
  runtime: 'edge'
};
```

Additionally, configuration objects are merged at the top level. For example:

```
export const config = {
  runtime: 'edge',
  regions: ['us1', 'us2'],
  foo: {
    baz: true
  }
};
```

These settings override defaults in layout or page configs.

Further reading

- Tutorial: Page options

State management

On this page

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:

```
// +page.server
let user: any;

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

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

        // NEVER DO THIS!
        let user: any = {
            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 or global state inside a `load`

function so that you can use the value in your components:

```
// +page.svelte
import { user } from '$lib/user';

/** @type {import('./$types').PageLoad} */
export async function load(event) {
    // This is buggy!
    const wordCount = data.content.split(' ').length;
    const estimatedReadingTime = wordCount / 250;
}
```

Instead, navigate your data by returning it from `load` and passing it into your components:

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

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

Then, pass it around to components or access via `page.data`. Avoid side-effects in `load` to make your applications easier to reason about.

Using state and stores with context

We can use Svelte's context API (`setContext` , `getContext`) to share state or stores across components without resorting to global variables. For example:

In a layout:

```
<script>
  import { setContext } from 'svelte';

  /** @type {import('./$types').LayoutProps} */
  let { data } = $props;

  // Pass a function referencing our state to the context
  setContext('user', () => data.user);
</script>
```

In a page:

```
<script>
  import { getContext } from 'svelte';

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

<p>Welcome {user().name}</p>
```

> We're passing a function into `setContext` to keep reactivity across boundaries. Read more about it here.

Legacy mode

You can also use stores from `svelte/store`, but with Svelte 5, it's recommended to adopt universal reactivity instead.

Note: Updating context-based state in deeper pages or components during SSR will not affect the parent, as the parent has already been rendered by the time the state updates. On the client, the value propagates and triggers updates up the hierarchy. To avoid visible 'flashing', it's advisable to pass state down rather than up.

Component and page state is preserved

When navigating within your application, SvelteKit reuses existing layout and page components. For example:

```
// src/routes/blog/[slug]/+page.svelte
/** @type {import('./$types').PageProps} */
let { data } = $props;

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

Because the component isn't recreated, lifecycle methods like `onMount` and `onDestroy` do not rerun, and calculations like `estimatedReadingTime` won't be recalculated automatically. To keep reactive data, use Svelte reactivity:

```
<script>
  /** @type {import('./$types').PageProps} */
  let { data } = $props;

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

This approach preserves state across navigation without unnecessary recomputation.

Storing state in the URL

State that should survive reloads or affect SSR, such as filters or sorting rules, can be stored in URL search parameters (e.g., `?sort=price&order=ascending`). These can be set via `<a href="...">`, `<form action="...">`, or programmatically using `goto('?key=value')`. Inside `load` functions, access these with the `url` parameter, and in components via `page.url.searchParams`.

Storing ephemeral state in snapshots

Transient UI state (e.g., whether an accordion is open) that can be rebuilt on page reload doesn't need to be stored persistently. SvelteKit provides snapshots, which allow associating component state with a history entry, useful for preserving UI state across navigations without persistent storage.

Previous | Next

Page options | Building your app

Building your app • Docs • Svelte

Building your app

On this page

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';


if (!building) {
  import setupMyDatabase from '$lib/server/database';
  setupMyDatabase();
}
```

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

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.

Edit this page on GitHub llms.txt

previous next
State management Adapters

Hello

Writing adapters

On this page

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.

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

```
/** @param {AdapterSpecificOptions} options */
export default function (options: any) {
  /** @type {import('@sveltejs/kit').Adapter} */
  const adapter: Adapter = {
    /** The name of the adapter, using for logging. Will typically correspond to the p
    name: 'adapter-package-name',

    /**
     * This function is called after SvelteKit has built your app.
     * @param {Builder} builder An object provided by SvelteKit that contains methods
     */
    async adapt(builder: Builder) {
      // adapter implementation
    },

    /**
     * Creates an `Emulator`, which allows the adapter to influence the environment
     * during dev, build and prerendering
     */
    emulate() {
      return {
        async platform(
          details?: { config: any; prerender: PrerenderOption }
        ): Promise<App.Platform> {
          // the returned object becomes `event.platform` during dev, build and
          // preview. Its shape is that of `App.Platform`
        }
      };
    },

    /**
     * Checks called during dev and build to determine whether specific features will
     */
    supports?: {
      read?: (details: { config: any; route: { id: string } }) => boolean;
    }
  };

  return adapter;
}
```

Of these, `name` and `adapt` are required. `emulate` and `supports` are optional.

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
  - Exposes 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]`.

Edit this page on GitHub
llms.txt

Previous Next
Deploy | Advanced routing

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Single-page apps

On this page

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:

```
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:

```
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.

> Note that the fallback page will always contain absolute asset paths (i.e., beginning with `/` rather than `.`) regardless of the value of paths.relative, since it is used to respond to requests for arbitrary paths.

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:

```
// src/routes/my-prerendered-page/+page.js
export const prerender = true;
export const ssr = true;
```

Edit this page on GitHub
llms.txt

---

previous | **next**
Static site generation | Cloudflare

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Not found!

If you were expecting to find something here, please drop by the Discord chatroom and let us know, or raise an issue on GitHub. Thanks!

Advanced routing - Docs - Svelte

On this page

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/main/documentation/docs/04-advanced-routing.md` would result in the following parameters being available to the page:

```
{
  "org": "sveltejs",
  "repo": "kit",
  "branch": "main",
  "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
```

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

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

  function error(status, body) {
    // Throws an error with a HTTP status code and an optional message.
    // When called during request handling, this will cause SvelteKit to
    // return an error response without invoking handleError.
    // Make sure you're not catching the thrown error, which would prevent SvelteKit f
    throw error(status, body);
  }

  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/fruits/[page]` would match `/fruits/apple`, but it would also match `/fruits/rocketship`. To ensure route parameters are well-formed, you can add a matcher.

```
// src/params/fruit.js
/**
 * @param {string} param
 * @return {param is ('apple' | 'orange')}
 * @satisfies {import('@sveltejs/kit').ParamMatcher}
 */
export function match(param) {
  return param === 'apple' || param === 'orange';
}
```

And augment your routes:

```
src/routes/fruits/[page=fruit]
```

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 matching (e.g., `[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. For example, `x/[[y]]/z` is equivalent to `x/z` for sorting purposes.
- Ties are resolved alphabetically

This ordering means 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 meanings in URLs, and `[ ] ( )` characters also have special meanings to SvelteKit, so these can't be used directly as part of your route.

To include these characters, use hexadecimal escape sequences of the form `[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 file at:

```
src/routes/smileys/[x+3a]-[x+29]/+page.svelte
```

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

```
':'.charCodeAt(0).toString(16); // '3a'
```

Unicode escape sequences can also be used, formatted as `[u+nnnn]`, where `nnnn` is a valid Unicode code point between `0000` and `10ffff`.

```
// Example: 😜
src/routes/[u+d83e][u+dd2a]/+page.svelte
// or directly with emoji
src/routes/😜/+page.svelte
```

> 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**. Sometimes, you may want different layout groups.

(group)

For example, routes on `/dashboard` or `/item` might share a layout, while marketing pages like `/about` or `/testimonials` use another. You can group routes with a directory name wrapped in parentheses. These do not affect the URL pathname of contained routes:

```
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)` if needed.

Breaking out of layouts

The root layout applies to every page unless omitted. To override for specific pages, place entire app inside one or more groups except for routes that should not inherit them.

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

+page@

Pages can break out of the current layout hierarchy using `@` to specify which layout to inherit from:

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

Possible options:

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

+layout@

Similarly, layouts can break out of their parent layout with `@`, resetting the hierarchy for all children. For example:

```
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   // skips `(app)/+layout.svelte`
│ └ +layout.svelte
└ +layout.svelte
```

When to use layout groups

Not all cases require layout grouping. You can use composition, reusable load functions, or conditionals inside components for flexibility.

An example layout that rewinds to the root layout:

```
<script>
  import ReusableLayout from '$lib/ReusableLayout.svelte';
  let { data, children } = $props();
</script>

<ReusableLayout {data}>
```

```
  {@render children()}
</ReusableLayout>
```

`+page@` to break out of layout hierarchy

Suppose, inside `(app)` group, you have `/item/[id]/embed`:

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

Options:

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

`+layout@`

Layouts can also break out similarly to pages, e.g.,

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

This resets the hierarchy for child routes.

When to use layout groups

Use them when they simplify your structure or reuse. Avoid overly complex nesting to keep your project manageable.

Advanced layouts example

In `src/routes/nested/route/+layout@.svelte`, you can import components:

```
<script>
  import ReusableLayout from '$lib/ReusableLayout.svelte';
  let { data, children } = $props();
</script>

<ReusableLayout {data}>
  {@render children()}
</ReusableLayout>
```

or with TypeScript:

```
<script lang="ts">
  import ReusableLayout from '$lib/ReusableLayout.svelte';
```

```
  let { data, children } = $props();
</script>
<ReusableLayout {data}>
  {@render children()}
</ReusableLayout>
```

`reusableLoad` function example

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

/** @type {import('./$types').PageLoad} */
export function load(event) {
  return reusableLoad(event);
  function error(status, body) {
    throw error(status, body);
  }
}

// Or with additional logic
import { reusableLoad } from '$lib/reusable-load-function';

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

Further reading

- Tutorial: Advanced Routing

Edit this page on GitHub

- llms.txt

previous next
Writing adapters Hooks

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

Errors

On this page

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.

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

/** @type {import('./$types').PageServerLoad} */
export async function load({ params }) {
  const post = await getPost(params.slug);
  if (!post) {
    throw error(404, {
      message: 'Not found'
    });
  }

  return {
    post
  };
}
```

This throws an exception that SvelteKit catches, causing it 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(...)`.

You can also pass a string as the second argument for convenience:

```
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 which becomes `$page.error`.

Responses

If an error occurs inside `handle` or inside a 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 customize 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 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'):

```
declare global {
  namespace App {
    interface Error {
      /**
       * Defines the common shape of expected and unexpected errors. Expected errors a
```

```
       * Unexpected errors are handled by the `handleError` hooks which should return
       */
      "code": string;
      "id": string;
    }
  }
}

export {};
```

This interface always includes a `message: string` property.

Further reading

- Tutorial: Errors and redirects
- Tutorial: Hooks

Link options

On this page

Overview

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 (not an external link), SvelteKit navigates to the new page by importing its code and calling any `load` functions needed to fetch data.

You can customize link behavior using `data-sveltekit-*` attributes. These can be applied directly to the `<a>` element or to a parent element.

These options also apply to `<form>` elements with a `<a href="form-actions#GET-vs-POST">method="GET"</a>`.

data-sveltekit-preload-data

Before the browser recognizes a click on a link, it can detect if the user has hovered the mouse over it (desktop) or triggered a `touchstart` or `mousedown`. This suggests an impending `click`.

SvelteKit can prefetch the code and data, giving a smoother UI. Control this with `data-sveltekit-preload-data`, which can be:

- `"hover"`: preload on hover (default)
- `"tap"`: preload on tap or click

The default `<body>` has `data-sveltekit-preload-data="hover"`, preloading links on hover. To disable preloading on hover, set `data-sveltekit-preload-data="tap"`, for example:

```
<a data-sveltekit-preload-data="tap" href="/stonks">Get current stonk values</a>
```

You can also programmatically invoke `preloadData` from `$app/navigation`.

Data preloading respects the `navigator.connection.saveData` setting, which if `true`, disables preloading.

data-sveltekit-preload-code

Preloading code can improve performance. It can be triggered eagerly or lazily with:

- `"eager"`: preload immediately

- `"viewport"`: preload when in viewport
- `"hover"`: preload on hover (only code)
- `"tap"`: preload on tap or click (only code)

Note: `viewport` and `eager` apply only to links already in DOM at navigation start. They will not trigger for dynamically added links.

This attribute only works if it's more eager than any `data-sveltekit-preload-data` present.

Preloading is disabled if reduced data usage is detected (`navigator.connection.saveData`).

data-sveltekit-reload

Add this attribute to compel a full-page reload instead of SvelteKit navigation:

```
<a data-sveltekit-reload href="/path">Path</a>
```

Links with `rel="external"` will also use full reload, and such links are ignored during prerendering.

data-sveltekit-replacestate

Use this attribute to replace the current history entry instead of pushing a new one:

```
<a data-sveltekit-replacestate href="/path">Path</a>
```

data-sveltekit-keepfocus

Prevent focus from resetting after navigation, useful for forms:

```
<form data-sveltekit-keepfocus>
  <input type="text" name="query" />
</form>
```

Note: Avoid using on links; focus should typically move naturally after navigation. Use on elements that remain after navigation; otherwise, focus may be lost.

data-sveltekit-noscroll

Disable scroll position reset after navigation:

```
<a href="path" data-sveltekit-noscroll>Path</a>
```

This preserves current scroll position during navigation.

Disabling options

To disable any of these options within a container:

```
<div data-sveltekit-preload-data="false">
  <!-- links here will not preload data -->
  <a href="/a">a</a>
  <a href="/b">b</a>
  <a href="/c">c</a>
</div>
```

You can also conditionally apply attributes:

```
<div data-sveltekit-preload-data={condition ? 'hover' : false}>
  <!-- links here -->
</div>
```

Edit this page on GitHub
llms.txt

Previous | **Next**
Errors | Service workers

Service workers

On this page

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) {
  window.addEventListener('load', () => {
    navigator.serviceWorker
      .register('./path/to/service-worker.js')
      .then(registration => {
        // Registration was successful
        console.log('ServiceWorker registration successful with scope: ', registration
      })
      .catch(error => {
        // registration failed
        console.log('ServiceWorker registration failed:', error);
      });
  });
}
```

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:

```
/// <reference types="@sveltejs/kit" />

import { build, files, version } from '$service-worker';
```

```
// Create a unique cache name for this deployment
const CACHE = `cache-${version}`;

// Define assets to cache
const ASSETS = [
  ...build, // the files generated by Vite
  ...files  // the static assets in your static directory
];

// Install event
self.addEventListener('install', (event) => {
  // Create a new cache and add all files to it
  event.waitUntil(
    caches.open(CACHE).then((cache) => {
      return cache.addAll(ASSETS);
    })
  );
});

// Activate event
self.addEventListener('activate', (event) => {
  // Remove previous cached data
  event.waitUntil(
    (async () => {
      const cacheNames = await caches.keys();
      await Promise.all(
        cacheNames.map((key) => {
          if (key !== CACHE) {
            return caches.delete(key);
          }
        })
      );
    })()
  );
});

// Fetch event
self.addEventListener('fetch', (event) => {
  // Ignore POST requests etc.
  if (event.request.method !== 'GET') return;

  event.respondWith(
    (async () => {
      const cache = await caches.open(CACHE);
      const cachedResponse = await cache.match(event.request);

      if (cachedResponse) {
        return cachedResponse;
      }

      try {
        const response = await fetch(event.request);
        // Save to cache if response is OK
        if (response.status === 200) {
```

```
          cache.put(event.request, response.clone());
        }
        return response;
      } catch (err) {
        // Optionally, fallback response or error handling
        throw err;
      }
    })()
  );
});
```

> **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';

if (dev) {
  navigator.serviceWorker.register('/service-worker.js', { type: 'module' });
} else {
  navigator.serviceWorker.register('/service-worker.js');
}
```

> `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));
```

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. Use `sw` instead of `self` in the rest of your code. The reference to the SvelteKit types ensures that the `$service-worker` import has proper type definitions.

Other solutions

SvelteKit's service worker implementation is designed to be easy to work with and is probably a good solution for most users. However, outside of SvelteKit, many PWA applications leverage the Workbox library. If you're used to using Workbox, you may prefer the Vite PWA plugin.

References

For more general information on service workers, we recommend the MDN web docs on using Service Workers.

Edit this page on GitHub | llms.txt

---

previous **next**
Link options Server-only modules

Server-only modules

On this page

Introduction

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 can only be imported into modules that only run on the server, such as hooks or `+page.server.js`.

Server-only utilities

The `$app/server` module, which contains a read function for reading assets from the filesystem, can likewise only be imported by code that runs on the server.

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)...

```
// Example: export in $lib/server/secrets.js
export const atlantisCoordinates = [/* redacted */];
```

```
// Example: import in src/routes/utils.js
export { atlantisCoordinates } from '$lib/server/secrets.js';
```

```
<!-- Example: usage in 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

Previous | **Next**

- Service workers
- Snapshots

Hello

SvelteKit: Shallow routing

On this page

Overview

As you navigate around a SvelteKit app, you create *history entries*. Clicking the back and forward buttons traverses through this list of entries, re-running any `load` functions and replacing page components as necessary.

Sometimes, it's useful to create history entries *without* navigating. For example, you might want to show a modal dialog that the user can dismiss by navigating back. This is particularly valuable on mobile devices, where swipe gestures are often more natural than interacting directly with the UI. In these cases, a modal that is *not* associated with a history entry can be a source of frustration, as a user may swipe backwards in an attempt to dismiss it and find themselves on the wrong page.

SvelteKit makes this possible with the `pushState` and `replaceState` functions, which allow you to associate state with a history entry without navigating.

Example: implementing a history-driven modal

```
<script>
  import { pushState } from '$app/navigation';
  import { page } from '$app/states';
  import Modal from './Modal.svelte';

  function showModal() {
    pushState('', {
      showModal: true
    });
  }
</script>

{#if page.state.showModal}
  <Modal onclose={() => history.back()} />
{/if}
```

```
<script lang="ts">
  import { pushState } from '$app/navigation';
  import { page } from '$app/states';
  import Modal from './Modal.svelte';

  function showModal() {
    pushState('', {
      showModal: true
    });
  }
```

```
</script>

{#if page.state.showModal}
  <Modal onclose={() => history.back()} />
{/if}
```

The modal can be dismissed by navigating back (unsetting `page.state.showModal`) or by interacting with it in a way that causes the `close` callback to run, which will navigate back programmatically.

API

The first argument to `pushState` is the URL, relative to the current URL. To stay on the current URL, use `''`.

The second argument is the new page state, which can be accessed via the page object as `page.state`. You can make page state type-safe by declaring an `App.PageState` interface (usually in `src/app.d.ts`).

To set page state without creating a new history entry, use `replaceState` instead of `pushState`.

> **Legacy mode**
> `page.state` from `$app/state` was added in SvelteKit 2.12. If you're using an earlier version or are using Svelte 4, use `$page.state` from `$app/stores` instead.

Loading data for a route

When shallow routing, you may want to render another `+page.svelte` inside the current page. For example, clicking on a photo thumbnail could pop up the detail view without navigating to the photo page.

For this to work, you need to load the data that the `+page.svelte` expects. A convenient way is to use `preloadData` inside the click handler of an `<a>` element. If the element (or a parent) uses data-sveltekit-preload-data, the data will have already been requested, and `preloadData` will reuse that request.

Example:

```
<script>
  import { preloadData, pushState, goto } from '$app/navigation';
  import { page } from '$app/states';
  import Modal from './Modal.svelte';
  import PhotoPage from './[id]/+page.svelte';

  let { data } = $props;
</script>

{#each data.thumbnails as thumbnail}
  <a
    href="/photos/{thumbnail.id}"
    onclick={async (e) => {
      if (innerWidth < 640 || e.shiftKey || e.metaKey || e.ctrlKey) return;

      // prevent navigation
      e.preventDefault();

      const { href } = e.currentTarget;

      // run `load` functions (or rather, get the result of the `load` functions)
```

```
      const result = await preloadData(href);

      if (result.type === 'loaded' && result.status === 200) {
        pushState(href, { selected: result.data });
      } else {
        // fallback: navigate normally
        goto(href);
      }
    }}
  >
    <img alt={thumbnail.alt} src={thumbnail.src} />
  </a>
{/each}

{#if page.state.selected}
  <Modal onclose={() => history.back()}>
    <!-- pass page data to the +page.svelte component, just like SvelteKit would on na
    <PhotoPage data={page.state.selected} />
  </Modal>
{/if}
```

Caveats

During server-side rendering, `page.state` is always an empty object. The same is true for the first page the user lands on — if the user reloads the page or returns from another document, state will *not* be applied until they navigate.

Shallow routing is a feature that requires JavaScript to work. Be mindful when using it and try to think of sensible fallback behavior in case JavaScript isn't available.

Edit this page on GitHub
llms.txt

Previous | **Next**
Snapshots | Packaging

Packaging

On this page

Introduction

You can use SvelteKit to build apps as well as component libraries, using the `@sveltejs/package` package (`npx sv create` 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 it is strongly recommended — people using your library might use TypeScript, which requires 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 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:

name

This is the name of your package. It will be available for others to install using that name, and visible on npm.

```
{
  "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 that 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.), add them to an `.npmignore` file. This results 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 `npx sv create`, it's set to a single export, the package root:

```
{
  "exports": {
    ".": {
      "types": "./dist/index.d.ts",
      "svelte": "./dist/index.js"
    }
  }
}
```

This informs bundlers and tooling that your package only has one entry point, the root, and everything should be imported through that:

```
import { Something } from 'your-library';
```

The `types` and `svelte` keys are export conditions. They tell tooling what file to import when looking 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 recognizes this as a Svelte component library. For libraries that do not export Svelte components but still work in non-Svelte projects, you can replace this

with `default`.

Previous versions of `@sveltejs/package` added a `package.json` export. That is no longer part of the template because all tooling now can handle a `package.json` without an explicit export.

You can customize the `exports` to your needs. For example, exposing a specific component directly:

```
{
  "exports": {
    "./Foo.svelte": {
      "types": "./dist/Foo.svelte.d.ts",
      "svelte": "./dist/Foo.svelte"
    }
  }
}
```

Consumers can then import directly:

```
import Foo from 'your-library/Foo.svelte';
```

Beware that doing this requires additional care if you provide type definitions. More here.

Each key in the `exports` map is the path users will use to import from your package, with values being either the file path to import or a map of conditions with respective paths.

Read more about here.

svelte

This is a legacy field that recognized Svelte component libraries. It's no longer needed when using the `svelte` export condition, but for backward compatibility it can stay. It should point to your root entry point:

```
{
  "svelte": "./dist/index.js"
}
```

sideEffects

The `sideEffects` field helps bundlers determine if a module contains side effects. Modules are considered to have side effects if they modify global variables or prototypes when imported.

Setting this field correctly improves tree-shaking and results in smaller bundles. By default, newly created projects mark scripts as side-effect-free.

Example:

```
{
  "sideEffects": ["**/*.css"]
}
```

If your package has files with side effects, list them:

```
{
  "sideEffects": [
    "**/*.css",
    "./dist/sideEffectfulFile.js"
  ]
}
```

TypeScript

Even if you don't use TypeScript, providing type definitions helps users with better IntelliSense.

By default, `@sveltejs/package` auto-generates type definitions for JavaScript, TypeScript, and Svelte files, with the correct `types` condition set in the exports map.

If you expose additional modules (like `your-library/foo`), you need to ensure type definitions are correctly resolved. TypeScript may not correctly resolve types for exports that don't follow the default structure.

Options include:

- Requiring users to set `moduleResolution` to `bundler`, `node16`, or `nodenext`.
- Using the `typesVersions` field in `package.json`, which remaps types based on TypeScript version:

```
{
  "exports": {
    "./foo": {
      "types": "./dist/foo.d.ts",
      "svelte": "./dist/foo.js"
    }
  },
  "typesVersions": {
    ">4.0": {
      "foo": ["./dist/foo.d.ts"]
    }
  }
}
```

More here.

Best practices

Avoid using SvelteKit-specific modules like `$app/environment` in packages unless intended solely for SvelteKit projects. Use alternatives like `esm-env`.

Pass configuration or environment details as props instead of relying on `$app` stores, to improve testability and reusability.

Ensure you add aliases via `svelte.config.js`, not `vite.config.js` or `tsconfig.json`, so they are processed correctly by `svelte-package`.

Update your package version carefully, especially when removing export paths or changing conditions, as these are breaking changes.

Source maps

Generate declaration maps by setting `"declarationMap": true` in `tsconfig.json`. This assists editors like VS Code in linking back to source files for features like "Go to Definition."

Include your source files in `files`, e.g., `src/lib`, so relative paths lead to actual source files on disk:

```
{
  "files": [
    "dist",
```

```
    "src/lib"
  ]
}
```

Options

`@sveltejs/package` accepts options:

- `-w` / `--watch` : watch `src/lib` for changes and rebuild
- `-i` / `--input` : input directory (defaults to `src/lib`)
- `-o` / `--output` : output directory (defaults to `dist`)
- `-t` / `--types` : generate type definitions (defaults to true)
- `--tsconfig` : path to `tsconfig.json` or `jsconfig.json`

Publishing

To publish the package:

```
npm publish
```

Caveats

All relative imports must be fully specified with extensions, e.g.,

```
import { something } from './something/index.js';
```

for TypeScript, the same applies; use `.js` extension, not `.ts` . Setting `"moduleResolution": "NodeNext"` in `tsconfig.json` helps.

All files except Svelte and transpiled TypeScript will be copied as-is during build.