SVELTEKIT Docs - English

Table of contents

Introduction

Before we begin

SvelteKit is in release candidate phase for 1.0 while we address reported issues and add polish. If you get stuck, reach out for help in the Discord chatroom.

See the migration guides for help upgrading from Sapper.

What is SvelteKit?

SvelteKit is a framework for building extremely high-performance web apps.

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; prefetching pages before the user initiates navigation; and configurable rendering that allows you to render your app on the server or in the browser at runtime or at build-time. SvelteKit does all the boring stuff for you so that you can get on with the creative part.

It uses Vite with a Svelte plugin to provide a lightning-fast and feature-rich development experience with Hot Module Replacement (HMR), where changes to your code are reflected in the browser instantly.

You don't need to know Svelte to understand the rest of this guide, but it will help. In short, it's a UI framework that compiles your components to highly optimized vanilla JavaScript. Read the introduction to Svelte blog post and the Svelte tutorial to learn more.

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 the FAQ 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 – you may not need to bother reading the rest of this guide!

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.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.

- `lib` contains your library code, 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
- `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
- `error.html` (optional) 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.js` (optional) contains your application's hooks
- `service-worker.js` (optional) contains your service worker

You can use `.ts` files instead of `.js` files, if using TypeScript.

static

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

tests

If you chose to add tests to your project during `npm create svelte@latest`, they 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 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`:

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

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

    return json({
        // retrieve a specific header
        userAgent: event.request.headers.get('user-agent')
    });
}
```

FormData

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

```js
// @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`:

```
// @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:

```
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` (or `+page.ts`, if you're TypeScript-inclined) module that exports a `load` function:

```js
/// 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`.

```js
/// 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 `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.

> `+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 next 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 re-run `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 (or `+server.ts`) exports functions corresponding to HTTP verbs like `GET`, `POST`, `PATCH`, `PUT` and `DELETE` 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.

Receiving data

By exporting `POST` / `PUT` / `PATCH` / `DELETE` 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` requests are always handled by `+server.js` since they do not apply to pages
- `GET` / `POST` 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`

$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.

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`.

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` (or `+page.ts`) 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. If your `load` function should *always* run on the server (because it uses private environment variables, for example, or accesses a database) then you can put it 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-only `load` functions can access additional arguments. To understand when to use `+page.js` and when to use `+page.server.js`, see Shared 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 here -->
    <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`.

Shared vs server

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

- `+page.js` and `+layout.js` files export `load` functions that are *shared* between server and browser
- `+page.server.js` and `+layout.server.js` files export `load` functions that are *server-only*

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

Input

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

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

Shared `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-only `load` function is the `data` property of the shared `load` function's argument.

Output

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

A server-only `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.

When to use which

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

Shared `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`:

```js
/// 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:

```json
{
    "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 direct 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. Note that headers will *not* be serialized, unless explicitly included via `filterSerializedResponseHeaders`. Then, during hydration, the response will be read from the HTML, guaranteeing consistency and preventing an additional network request - if you got a warning in your browser console when using the browser `fetch` instead of the `load` `fetch`, this is why.

```js
/// 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 will only be passed through if the target host is the same as the SvelteKit application or a more specific subdomain of it.

Cookies and headers

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

```js
/// 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)
    };
}
```

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.

Both server-only and shared `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:

```js
// @errors: 2322
/// 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 };
}
```

```
<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.

```js
/// 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');
    }
}
```

Promise unwrapping

Top-level promises will be awaited, which makes it easy to return multiple promises without creating a waterfall:

```js
/// file: src/routes/+page.server.js
/** @type {import('./$types').PageServerLoad} */
export function load() {
    return {
        a: Promise.resolve('a'),
        b: Promise.resolve('b'),
        c: {
            value: Promise.resolve('c')
        }
    };
}
```

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

    console.log(data.a); // 'a'
    console.log(data.b); // 'b'
    console.log(data.c.value); // `Promise {...}`
</script>
```

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-only `load` functions are grouped into a single response. Once all `load` functions have returned, the page is rendered.

Invalidation

SvelteKit tracks the dependencies of each `load` function to avoid re-running 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 re-run 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 re-run if a parent `load` function is re-run.

Manual invalidation

You can also re-run `load` functions that apply to the current page using `invalidate(url)`, which re-runs all `load` functions that depend on `url`, and `invalidateAll()`, which re-runs every `load` function.

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 { invalidateAll } from '$app/navigation';

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

    function rerunLoadFunction() {
        // any of these will cause the `load` function to re-run
        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>
```

To summarize, a `load` function will re-run 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
- It calls `await parent()` and a parent `load` function re-ran
- It declared a dependency on a specific URL via `fetch` or `depends`, and that URL was marked invalid with `invalidate(url)`
- All active `load` functions were forcibly re-run with `invalidateAll()`

Note that re-running 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.

Shared state

In many server environments, a single instance of your app will serve multiple users. For that reason, per-request or per-user state must not be stored in shared variables outside your `load` functions, but should instead be stored in `event.locals`.

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">
    <input name="email" type="email">
    <input name="password" type="password">
    <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">
    <input name="email" type="email">
    <input name="password" type="password">
    <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: 2339 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 };
}
```

```js
/** @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
    }
};
```

```svelte
/// 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 `invalid` 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`:

```diff
// @errors: 2339 2304
/// file: src/routes/login/+page.server.js
+import { invalid } 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 invalid(400, { email, missing: true });
+       }

        const user = await db.getUser(email);

+       if (!user || user.password !== hash(password)) {
+           return invalid(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">
-   <input name="email" type="email">
+   {#if form?.missing}<p class="error">The email field is required</p>{/if}
+   {#if form?.incorrect}<p class="error">Invalid credentials!</p>{/if}
+   <input name="email" type="email" value={form?.email ?? ''}>

    <input name="password" type="password">
    <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`:

```
// @errors: 2339 2304
/// file: src/routes/login/+page.server.js
+import { invalid, 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 invalid(400, { email, missing: true });
        }

        if (user.password !== hash(password)) {
            return invalid(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 re-run 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.
- 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

To customise the behaviour, you can provide a function 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={({ form, data, action, cancel }) => {
        // `form` is the `<form>` element
        // `data` is its `FormData` object
        // `action` is the URL to which the form is posted
        // `cancel()` will prevent the submission

        return async ({ result, update }) => {
            // `result` is an `ActionResult` object
            // `update` is a function which triggers the 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`:

```
<script>
+   import { enhance, applyAction } from '$app/forms';

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

<form
    method="POST"
    use:enhance={({ form, data, action, cancel }) => {
        // `form` is the `<form>` element
        // `data` is its `FormData` object
        // `action` is the URL to which the form is posted
        // `cancel()` will prevent the submission

        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`, `invalid` — 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`

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') {
            // re-run 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'
+   }
});
```

Go to TOC

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.

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 the `entries` option in the prerender configuration.

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...

```js
/// 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.

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`.

Note that this will disable client-side routing for any navigation from this page, regardless of whether the router is already active.

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`. 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

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).

```js
/// 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:

```js
/// 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.

```js
/// 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.

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.

By default, projects are configured to use `@sveltejs/adapter-auto`, which detects your production environment and selects the appropriate adapter where possible. If your platform isn't (yet) supported, you may need to install a custom adapter or write one.

> See the adapter-auto README for information on adding support for new environments.

Supported environments

SvelteKit offers a number of officially supported adapters.

You can deploy to the following platforms with the default adapter, `adapter-auto`:

- Cloudflare Pages via `adapter-cloudflare`
- Netlify via `adapter-netlify`
- Vercel via `adapter-vercel`

Node.js

To create a simple Node server, install the `@sveltejs/adapter-node` package and update your `svelte.config.js`:

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

With this, `vite build` will generate a self-contained Node app inside the `build` directory. You can pass options to adapters, such as customising the output directory:

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

export default {
    kit: {
-       adapter: adapter()
+       adapter: adapter({ out: 'my-output-directory' })
    }
};
```

Static sites

Most adapters will generate static HTML for any prerenderable pages of your site. In some cases, your entire app might be prerenderable, in which case you can use `@sveltejs/adapter-static` to generate static HTML for *all* your pages. A fully static site can be hosted on a wide variety of platforms, including static hosts like GitHub Pages.

```diff
/// file: svelte.config.js
-import adapter from '@sveltejs/adapter-auto';
+import adapter from '@sveltejs/adapter-static';
```

You can also use `adapter-static` to generate single-page apps (SPAs) by specifying a fallback page and disabling SSR.

> You must ensure `trailingSlash` 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'` to create `/a/index.html` instead.

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.

Community adapters

Additional community-provided adapters exist for other platforms. After installing the relevant adapter with your package manager, update your `svelte.config.js`:

```diff
/// file: svelte.config.js
-import adapter from '@sveltejs/adapter-auto';
+import adapter from 'svelte-adapter-[x]';
```

Writing custom adapters

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`:

```js
// @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;
}
```

The types for `Adapter` and its parameters are available in types/index.d.ts.

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/install-fetch` helper for platforms that can use `node-fetch`
- 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]`.

> The adapter API may change before 1.0.

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
```

```js
/// 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...

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

...and augmenting your routes:

```diff
-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.

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]]/foo/+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]]/foo/+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);
}
```

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-only `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 add call multiple `handle` functions 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` function that runs on the server (or during pre-rendering).

Or 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`:

```
// @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; code: string }` and returning it accordingly from the `handleError` functions:

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

```
/// file: src/hooks.server.js
// @errors: 2322 2571
// @filename: ambient.d.ts
const Sentry: any;

// @filename: index.js
// cut---
/** @type {import('@sveltejs/kit').HandleServerError} */
export function handleError({ error, event }) {
	// example integration with https://sentry.io/
	Sentry.captureException(error, { event });

	return {
		message: 'Whoops!',
		code: error.code ?? 'UNKNOWN'
	};
}
```

```
/// file: src/hooks.client.js
// @errors: 2322 2571
// @filename: ambient.d.ts
```

```js
const Sentry: any;

// @filename: index.js
// cut---
/** @type {import('@sveltejs/kit').HandleClientError} */
export function handleError({ error, event }) {
    // example integration with https://sentry.io/
    Sentry.captureException(error, { event });

    return {
        message: 'Whoops!',
        code: error.code ?? 'UNKNOWN'
    };
}
```

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

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 below.

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 2571
// @filename: ambient.d.ts
const Sentry: any;

// @filename: index.js
// cut---
/** @type {import('@sveltejs/kit').HandleServerError} */
export function handleError({ error, event }) {
    // example integration with https://sentry.io/
    Sentry.captureException(error, { event });

    return {
        message: 'Whoops!',
        code: error.code ?? 'UNKNOWN'
    };
}
```

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.

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
namespace App {
	interface Error {
		code: string;
		id: string;
	}
}
```

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

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.

data-sveltekit-prefetch

To get a head start on importing the code and fetching the page's data, use the `data-sveltekit-prefetch` attribute, which will start loading everything as soon as the user hovers over the link (on a desktop) or touches it (on mobile), rather than waiting for the `click` event to trigger navigation. Typically, this buys us an extra couple of hundred milliseconds, which is the difference between a user interface that feels laggy, and one that feels snappy.

To apply this behaviour across the board, add the attribute to a parent element (or even the `<body>` in your `src/app.html`):

```
/// file: src/routes/+layout.svelte
<main data-sveltekit-prefetch>
    <slot />
</main>
```

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

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-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 `"off"` value:

```
<div data-sveltekit-prefetch>
    <!-- these links will be prefetched -->
    <a href="/a">a</a>
    <a href="/b">b</a>
    <a href="/c">c</a>

    <div data-sveltekit-prefetch="off">
        <!-- these links will NOT be prefetched -->
        <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:

```
<div data-sveltekit-reload={shouldReload ? '' : 'off'}>
```

This works because in HTML, `<element attribute>` is equivalent to `<element attribute="">`

Go to TOC

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.ts`, or `src/service-worker/index.js`, etc) it will be built with Vite and automatically registered. You can disable automatic registration if you need to register the service worker with your own logic (e.g. prompt user for update, configure periodic updates, use `workbox`, etc).

> You can change the location of your service worker and disable automatic registration in your project configuration.

Inside the service worker you have access to the `$service-worker` module. If your Vite config specifies `define`, this will be applied to service workers as well as your server/client builds.

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'
});
```<span style='float: footnote;'><a href="../../index.html#toc">Go to TOC</a>
</span>
~~~

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
/// 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.

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 the `experimental.useVitePreprocess` option useful:

```
// svelte.config.js
export default {
    vitePlugin: {
        experimental: {
            useVitePreprocess: true
        }
    }
};
```

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

Packaging

`svelte-package` is currently experimental. Non-backward compatible changes may occur in any future release.

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. `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 `package` directory (which can be configured) containing the following:

- All the files in `src/lib`, unless you configure custom `include`/`exclude` options. 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.
- A `package.json` copied from the project root with all fields except `"scripts"`, `"publishConfig.directory"` and `"publishConfig.linkDirectory"`. The `"dependencies"` field is included, which means you should add packages that you only need for your documentation or demo site to `"devDependencies"`. A `"type": "module"` and an `"exports"` field will be added if it's not defined in the original file.

The `"exports"` field contains the package's entry points. By default, all files in `src/lib` will be treated as an entry point unless they start with (or live in a directory that starts with) an underscore, but you can configure this behaviour. If you have a `src/lib/index.js` or `src/lib/index.svelte` file, it will be treated as the package root.

For example, if you had a `src/lib/Foo.svelte` component and a `src/lib/index.js` module that re-exported it, a consumer of your library could do either of the following:

```
// @filename: ambient.d.ts
declare module 'your-library';

// @filename: index.js
// cut---
import { Foo } from 'your-library';
```

```
// @filename: ambient.d.ts
declare module 'your-library/Foo.svelte';

// @filename: index.js
// cut---
import Foo from 'your-library/Foo.svelte';
```

> 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.meta.env.SSR` to make the library available to all Vite-based projects or better yet use Node conditional exports to make it work for all bundlers. 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.

Options

`svelte-package` accepts the following options:

- `-w` / `--watch` — watch files in `src/lib` for changes and rebuild the package

Publishing

To publish the generated package:

```
npm publish ./package
```

The `./package` above is referring to the directory name generated, change accordingly if you configure a custom `package.dir`.

Caveats

All relative file imports need to be fully specified, adhering to Node's ESM algorithm. This means you cannot import the file `src/lib/something/index.js` like `import { something } from './something`, instead you need to import it like this: `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 isn't under our control, the TypeScript team has made that decision). Setting `"moduleResolution": "NodeNext"` in your `tsconfig.json` or `jsconfig.json` will help you with this.

This is a relatively experimental feature and is not yet fully implemented. 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. If you want to customize this behavior, you can implement custom focus management logic using 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 `inline-StyleThreshold` option...

```js
/// 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` ...

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

...and transforming the HTML using `transformPageChunk` along with `transform` imported from `@sveltejs/amp` :

```js
import * as amp from '@sveltejs/amp';

/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    let buffer = '';
    return resolve(event, {
        transformPageChunk: ({ html, done }) => {
            buffer += html;
            if (done) return amp.transform(html);
        }
    });
}
```

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. All values are optional. The complete list of options, with defaults, is shown here:

```
/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
    // options passed to svelte.compile (https://svelte.dev/docs#compile-time-
svelte-compile)
    compilerOptions: {},

    // an array of file extensions that should be treated as Svelte components
    extensions: ['.svelte'],

    kit: {
        adapter: undefined,
        alias: {},
        appDir: '_app',
        csp: {
            mode: 'auto',
            directives: {
                'default-src': undefined
                // ...
            }
        },
        csrf: {
            checkOrigin: true
        },
        env: {
            dir: process.cwd(),
            publicPrefix: 'PUBLIC_'
        },
        files: {
            assets: 'static',
            hooks: {
                client: 'src/hooks.client',
                server: 'src/hooks.server'
            },
            lib: 'src/lib',
            params: 'src/params',
            routes: 'src/routes',
            serviceWorker: 'src/service-worker',
            appTemplate: 'src/app.html',
            errorTemplate: 'src/error.html'
        },
        inlineStyleThreshold: 0,
        moduleExtensions: ['.js', '.ts'],
        outDir: '.svelte-kit',
        paths: {
            assets: '',
            base: ''
        },
        prerender: {
            concurrency: 1,
            crawl: true,
            entries: ['*'],
            handleHttpError: 'fail',
            handleMissingId: 'fail',
            origin: 'http://sveltekit-prerender'
        },
        serviceWorker: {
            register: true,
            files: (filepath) => !/\.DS_Store/.test(filepath)
        },
        version: {
            name: Date.now().toString(),
            pollInterval: 0
        }
    },

    // options passed to @sveltejs/package
    package: {
        source: 'value of kit.files.lib, if available, else src/lib',
        dir: 'package',
        emitTypes: true,
        // excludes all .d.ts and files starting with _ as the name
        exports: (filepath) => !/^_|\/_|\.d\.ts$/.test(filepath),
        files: () => true
    },

    // options passed to svelte.preprocess (https://svelte.dev/docs#compile-time-
svelte-preprocess)
    preprocess: null
};

export default config;
```

adapter

Run when executing `vite build` and determines how the output is converted for different platforms. See Adapters.

alias

An object containing zero or more aliases used to replace values in `import` statements. These aliases are automatically passed to Vite and TypeScript.

```
/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        alias: {
            // this will match a file
            'my-file': 'path/to/my-file.js',

            // this will match a directory and its contents
            // (`my-directory/x` resolves to `path/to/my-directory/x`)
            'my-directory': 'path/to/my-directory',

            // an alias ending /* will only match
            // the contents of a directory, not the directory itself
            'my-directory/*': 'path/to/my-directory/*'
        }
    }
};
```

> The built-in `$lib` alias is controlled by `config.kit.files.lib` as it is used for packaging.

> You will need to run `npm run dev` to have SvelteKit automatically generate the required alias configuration in `jsconfig.json` or `tsconfig.json`.

appDir

The directory relative to `paths.assets` where the built JS and CSS (and imported assets) are served from. (The filenames therein contain content-based hashes, meaning they can be cached indefinitely). Must not start or end with `/`.

csp

An object containing zero or more of the following values:

- `mode` — 'hash', 'nonce' or 'auto'
- `directives` — an object of `[directive]: value[]` pairs
- `reportOnly` — an object of `[directive]: value[]` pairs for CSP report-only mode

Content Security Policy configuration. CSP helps to protect your users against cross-site scripting (XSS) attacks, by limiting the places resources can be loaded from. For example, a configuration like this...

```js
/// file: svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        csp: {
            directives: {
                'script-src': ['self']
            },
            reportOnly: {
                'script-src': ['self']
            }
        }
    }
};

export default config;
```

...would prevent scripts loading from external sites. SvelteKit will augment the specified directives with nonces or hashes (depending on `mode`) for any inline styles and scripts it generates.

To add a nonce for scripts and links manually included in `app.html`, you may use the placeholder `%sveltekit.nonce%` (for example `<script nonce="%sveltekit.nonce%">`).

When pages are prerendered, the CSP header is added via a `<meta http-equiv>` tag (note that in this case, `frame-ancestors`, `report-uri` and `sandbox` directives will be ignored).

When `mode` is `'auto'`, SvelteKit will use nonces for dynamically rendered pages and hashes for pre-rendered pages. Using nonces with prerendered pages is insecure and therefore forbidden.

Note that most Svelte transitions work by creating an inline `<style>` element. If you use these in your app, you must either leave the `style-src` directive unspecified or add `unsafe-inline`.

csrf

Protection against cross-site request forgery attacks:

- `checkOrigin` — if `true`, SvelteKit will check the incoming `origin` header for `POST` form submissions and verify that it matches the server's origin

To allow people to make `POST` form submissions to your app from other origins, you will need to disable this option. Be careful!

env

Environment variable configuration:

- `dir` — the directory to search for `.env` files.
- `publicPrefix` — a prefix that signals that an environment variable is safe to expose to client-side code. See `$env/static/public` and `$env/dynamic/public`. Note that Vite's `envPrefix` must be set separately if you are using Vite's environment variable handling - though use of that feature should generally be unnecessary.

files

An object containing zero or more of the following `string` values:

- `assets` — a place to put static files that should have stable URLs and undergo no processing, such as `favicon.ico` or `manifest.json`
- `hooks` — the location of your client and server hooks (see Hooks)
- `lib` — your app's internal library, accessible throughout the codebase as `$lib`
- `params` — a directory containing parameter matchers
- `routes` — the files that define the structure of your app (see Routing)
- `serviceWorker` — the location of your service worker's entry point (see Service workers)
- `template` — the location of the template for HTML responses

inlineStyleThreshold

Inline CSS inside a `<style>` block at the head of the HTML. This option is a number that specifies the maximum length of a CSS file to be inlined. All CSS files needed for the page and smaller than this value are merged and inlined in a `<style>` block.

> This results in fewer initial requests and can improve your First Contentful Paint score. However, it generates larger HTML output and reduces the effectiveness of browser caches. Use it advisedly.

moduleExtensions

An array of file extensions that SvelteKit will treat as modules. Files with extensions that match neither `config.extensions` nor `config.kit.moduleExtensions` will be ignored by the router.

outDir

The directory that SvelteKit writes files to during `dev` and `build`. You should exclude this directory from version control.

package

Options related to creating a package.

- `source` - library directory
- `dir` - output directory
- `emitTypes` - by default, `svelte-package` will automatically generate types for your package in the form of `.d.ts` files. While generating types is configurable, we believe it is best for the ecosystem quality to generate types, always. Please make sure you have a good reason when setting it to `false` (for example when you want to provide handwritten type definitions instead)
- `exports` - a function with the type of `(filepath: string) => boolean`. When `true`, the filepath will be included in the `exports` field of the `package.json`. Any existing values in the `package.json` source will be merged with values from the original `exports` field taking precedence
- `files` - a function with the type of `(filepath: string) => boolean`. When `true`, the file will be processed and copied over to the final output folder, specified in `dir`

For advanced `filepath` matching, you can use `exports` and `files` options in conjunction with a globbing library:

```
// @filename: ambient.d.ts
declare module 'micromatch';

/// file: svelte.config.js
// @filename: index.js
// cut---
import mm from 'micromatch';
```

```js
/** @type {import('@sveltejs/kit').Config} */
const config = {
    package: {
        exports: (filepath) => {
            if (filepath.endsWith('.d.ts')) return false;
            return mm.isMatch(filepath, ['!**/_*', '!**/internal/**']);
        },
        files: mm.matcher('!**/build.*')
    }
};

export default config;
```

paths

An object containing zero or more of the following `string` values:

- `assets` — an absolute path that your app's files are served from. This is useful if your files are served from a storage bucket of some kind
- `base` — a root-relative path that must start, but not end with `/` (e.g. `/base-path`), unless it is the empty string. This specifies where your app is served from and allows the app to live on a non-root path. Note that you need to prepend all your root-relative links with the base value or they will point to the root of your domain, not your `base` (this is how the browser works). You can use `base` from `$app/paths` for that: `<a href="{base}/your-page">Link</a>`. If you find yourself writing this often, it may make sense to extract this into a reusable component.

prerender

See Prerendering. An object containing zero or more of the following:

- `concurrency` — how many pages can be prerendered simultaneously. JS is single-threaded, but in cases where prerendering performance is network-bound (for example loading content from a remote CMS) this can speed things up by processing other tasks while waiting on the network response

- `crawl` — determines whether SvelteKit should find pages to prerender by following links from the seed page(s)

- `entries` — an array of pages to prerender, or start crawling from (if `crawl: true`). The `*` string includes all non-dynamic routes (i.e. pages with no `[parameters]`, because SvelteKit doesn't know what value the parameters should have)

- `handleHttpError`

  - `'fail'` — (default) fails the build when a routing error is encountered when following a link

  - `'ignore'` - silently ignore the failure and continue

  - `'warn'` — continue, but print a warning

  - `(details) => void` — a custom error handler that takes a `details` object with `status`, `path`, `referrer`, `referenceType` and `message` properties. If you `throw` from this function, the build will fail

```
/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        prerender: {
            handleHttpError: ({ path, referrer, message }) => {
                // ignore deliberate link to shiny 404 page
                if (path === '/not-found' && referrer === '/blog/how-we-built-
our-404-page') {
                    return;
                }

                // otherwise fail the build
                throw new Error(message);
            }
        }
    }
};
```

- `handleMissingId`
  - `'fail'` — (default) fails the build when a prerendered page links to another prerendered page with a `#` fragment that doesn't correspond to an `id`
  - `'ignore'` - silently ignore the failure and continue
  - `'warn'` — continue, but print a warning
  - `(details) => void` — a custom error handler that takes a `details` object with `path`, `id`, `referrers` and `message` properties. If you `throw` from this function, the build will fail
- `origin` — the value of `url.origin` during prerendering; useful if it is included in rendered content

serviceWorker

An object containing zero or more of the following values:

- `register` - if set to `false`, will disable automatic service worker registration
- `files` - a function with the type of `(filepath: string) => boolean`. When `true`, the given file will be available in `$service-worker.files`, otherwise it will be excluded.

version

An object containing zero or more of the following values:

- `name` - current app version string
- `pollInterval` - interval in milliseconds to poll for version changes

Client-side navigation can be buggy if you deploy a new version of your app while people are using it. If the code for the new page is already loaded, it may have stale content; if it isn't, the app's route manifest may point to a JavaScript file that no longer exists. SvelteKit solves this problem by falling back to traditional full-page navigation if it detects that a new version has been deployed, using the `name` specified here (which defaults to a timestamp of the build).

If you set `pollInterval` to a non-zero value, SvelteKit will poll for new versions in the background and set the value of the `updated` store to `true` when it detects one.

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 generated files for your project such as types and a `tsconfig.json`. 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.

EXPORTS

Go to TOC

Types

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
/** @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"
    }
}
```

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 identical imports 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-prefetch`
- `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 the FAQ for detailed information about integrations.

HTML minifier

Sapper includes `html-minifier` by default. SvelteKit does not include this, but it can be added as a hook:

```
// @filename: ambient.d.ts
/// <reference types="@sveltejs/kit" />
declare module 'html-minifier';

// @filename: index.js
```

```js
// cut---
import { minify } from 'html-minifier';
import { prerendering } 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 }) {
    const response = await resolve(event);

    if (prerendering && response.headers.get('content-type') === 'text/html') {
        return new Response(minify(await response.text(), minification_options), {
            status: response.status,
            headers: response.headers
        });
    }

    return response;
}
```

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.

Integrations

`svelte-preprocess` automatically transforms the code in your Svelte templates to provide support for TypeScript, PostCSS, scss/sass, Less, and many other technologies (except CoffeeScript which is not supported by SvelteKit). The first step of setting it up is to add `svelte-preprocess` to your `svelte.config.js`. It is provided by the template if you're using TypeScript whereas JavaScript users will need to add it. After that, you will often only need to install the corresponding library such as `npm install -D sass` or `npm install -D less`. See the `svelte-preprocess` docs for more details.

Svelte Adders allow you to setup many different complex integrations like Tailwind, PostCSS, 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.

The SvelteKit FAQ also has a section on integrations, which may be helpful if you run into any issues.

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.

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).

CSR and SPA

Client-side rendering (CSR) is the generation of the page contents in the web browser using JavaScript. 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.

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.

SSG

Static Site Generation (SSG) is a term that refers to a site where every page is prerendered. This is what SvelteKit's `adapter-static` does. 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.

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.

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.

Go to TOC

Go to TOC

Other resources

Please see the Svelte FAQ and `vite-plugin-svelte` FAQ as well for the answers to questions deriving from those libraries.

Go to TOC

SvelteKit is not 1.0 yet. Should I use it?

SvelteKit is currently in release candidate phase. We expect minimal to no breaking changes and for development to be focused on bug fixes.

Go to TOC

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 unexpected 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.

Go to TOC

Go to TOC

I'm having trouble using an adapter.

Please make sure the version of the adapter specified in your `package.json` is `"next"`.

Go to TOC

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:

```js
/// file: svelte.config.js
// @filename: index.js
/// <reference types="@types/node" />
import { URL } from 'url';
// cut---
import { readFileSync } from 'fs';
import { fileURLToPath } from 'url';

const file = fileURLToPath(new URL('package.json', import.meta.url));
const json = readFileSync(file, 'utf8');
const pkg = JSON.parse(json);
```

Go to TOC

How do I fix the error I'm getting trying to include a package?

Vite's SSR support has become fairly stable since Vite 2.7. Most issues related to including a library are due to incorrect packaging.

Libraries work best with Vite when they distribute an ESM version and you may wish to suggest this to library authors. 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 entirely as ESM and have a `svelte` field defining the entry point.

It is encouraged to make sure the dependencies of external Svelte components provide an ESM version. However, in order to handle CommonJS dependencies `vite-plugin-svelte` will look for any CJS dependencies of external Svelte components and ask Vite to pre-bundle them by automatically adding them to Vite's `optimizeDeps.include` which will use `esbuild` to convert them to ESM. A side effect of this approach is that it takes longer to load the initial page. If this becomes noticable, try setting experimental.prebundleSvelteLibraries: true in `svelte.config.js`. Note that this option is experimental.

If you are still encountering issues we recommend checking the list of known Vite issues most commonly affecting SvelteKit users and 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.

Go to TOC

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 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:

```
// @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.

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:

```js
/// <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:

```js
// @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:

```js
// @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.

```js
// @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 with Yarn?

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

Colophon

This book is created by using the following sources:

- Sveltekit - English
- GitHub source: sveltejs/kit/documentation
- Created: 2022-11-28
- 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