Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT • GETTING STARTED

Introduction

On this page
Introduction
Before we begin
What is SvelteKit?
What is Svelte?
SvelteKit vs Svelte

Before we begin

If you're new to Svelte or SvelteKit we recommend checking out the interactive tutorial.

If you get stuck, reach out for help in the Discord chatroom.

What is SvelteKit?

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

To learn more about the kinds of applications you can build with SvelteKit, see the documentation regarding project types.

What is Svelte?

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

SvelteKit vs Svelte

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

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

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

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$app/environment

```
import { browser, building, dev, version } from '$app/environment';
```

On this page
$app/environment
browser
building
dev
version

browser

`true` if the app is running in the browser.

```
const browser: boolean;
```

building

SvelteKit analyses your app during the `build` step by running it. During this process, `building` is `true`. This also applies during prerendering.

```
const building: boolean;
```

dev

Whether the dev server is running. This is not guaranteed to correspond to `NODE_ENV` or `MODE`.

```
const dev: boolean;
```

version

The value of `config.kit.version.name`.

```
const version: string;
```

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$app/forms

```
import { applyAction, deserialize, enhance } from '$app/forms';
```

applyAction

This action updates the `form` property of the current page with the given data and updates `page.status`. In case of an error, it redirects to the nearest error page.

```
function applyAction<
  Success extends Record<string, unknown> | undefined,
  Failure extends Record<string, unknown> | undefined
>(
  result: import('@sveltejs/kit').ActionResult<
    Success,
    Failure
  >
): Promise<void>;
```

deserialize

Use this function to deserialize the response from a form submission. Usage:

```
import { deserialize } from '$app/forms';

async function handleSubmit(event) {
  const response = await fetch('/form?/action', {
    method: 'POST',
    body: new FormData(event.target)
  });

  const result = deserialize(await response.text());
  // ...
}
```

```
function deserialize<
  Success extends Record<string, unknown> | undefined,
  Failure extends Record<string, unknown> | undefined
>(
  result: string
): import('@sveltejs/kit').ActionResult<Success, Failure>;
```

enhance

This action enhances a `<form>` element that otherwise would work without JavaScript.

The `submit` function is called upon submission with the given FormData and the `action` that should be triggered. If `cancel` is called, the form will not be submitted. You can use the abort `controller` to cancel the submission in case another one starts. If a function is returned, that function is called with the response from the server. If nothing is returned, the fallback will be used.

If this function or its return value isn't set, it

- falls back to updating the `form` prop with the returned data if the action is on the same page as the form
- updates `page.status`
- resets the `<form>` element and invalidates all data in case of successful submission with no redirect response
- redirects in case of a redirect response
- redirects to the nearest error page in case of an unexpected error

If you provide a custom function with a callback and want to use the default behavior, invoke `update` in your callback. It accepts an options object

- `reset: false` if you don't want the `<form>` values to be reset after a successful submission
- `invalidateAll: false` if you don't want the action to call `invalidateAll` after submission

```
function enhance<
  Success extends Record<string, unknown> | undefined,
  Failure extends Record<string, unknown> | undefined
>(
  form_element: HTMLFormElement,
  submit?: import('@sveltejs/kit').SubmitFunction<
    Success,
    Failure
  >
): {
  destroy(): void;
};
```

On this page
$app/forms
applyAction
deserialize
enhance

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$app/navigation

```
import {
  afterNavigate,
  beforeNavigate,
  disableScrollHandling,
  goto,
  invalidate,
  invalidateAll,
  onNavigate,
  preloadCode,
  preloadData,
  pushState,
  replaceState
} from '$app/navigation';
```

afterNavigate

A lifecycle function that runs the supplied `callback` when the current component mounts, and also whenever we navigate to a URL.

`afterNavigate` must be called during a component initialization. It remains active as long as the component is mounted.

```
function afterNavigate(
  callback: (
    navigation: import('@sveltejs/kit').AfterNavigate
  ) => void
): void;
```

beforeNavigate

A navigation interceptor that triggers before we navigate to a URL, whether by clicking a link, calling `goto(...)`, or using the browser back/forward controls.

Calling `cancel()` will prevent the navigation from completing. If `navigation.type === 'leave'` — meaning the user is navigating away from the app (or closing the tab) — calling `cancel` will trigger the native browser unload confirmation dialog. In this case, the navigation may or may not be cancelled depending on the user's response.

When a navigation isn't to a SvelteKit-owned route (and therefore controlled by SvelteKit's client-side router), `navigation.to.route.id` will be `null`.

If the navigation will (if not cancelled) cause the document to unload — in other words `'leave'` navigations and `'link'` navigations where `navigation.to.route === null` — `navigation.willUnload` is `true`.

`beforeNavigate` must be called during a component initialization. It remains active as long as the component is mounted.

```
function beforeNavigate(
  callback: (
    navigation: import('@sveltejs/kit').BeforeNavigate
  ) => void
): void;
```

disableScrollHandling

If called when the page is being updated following a navigation (in `onMount` or `afterNavigate` or an action, for example), this disables SvelteKit's built-in scroll handling. This is generally discouraged, since it breaks user expectations.

```
function disableScrollHandling(): void;
```

goto

Allows you to navigate programmatically to a given route, with options such as keeping the current element focused. Returns a Promise that resolves when SvelteKit navigates (or fails to navigate, in which case the promise rejects) to the specified `url`.

For external URLs, use `window.location = url` instead of calling `goto(url)`.

```
function goto(
  url: string | URL,
  opts?:
    | {
        replaceState?: boolean | undefined;
        noScroll?: boolean | undefined;
        keepFocus?: boolean | undefined;
        invalidateAll?: boolean | undefined;
        invalidate?:
          | (string | URL | ((url: URL) => boolean))[]
          | undefined;
        state?: App.PageState | undefined;
      }
    | undefined
): Promise<void>;
```

invalidate

Causes any `load` functions belonging to the currently active page to re-run if they depend on the `url` in question, via `fetch` or `depends`. Returns a `Promise` that resolves when the page is subsequently updated.

If the argument is given as a `string` or `URL`, it must resolve to the same URL that was passed to `fetch` or `depends` (including query parameters). To create a custom identifier, use a string beginning with `[a-z]+:` (e.g. `custom:state`) — this is a valid URL.

The `function` argument can be used define a custom predicate. It receives the full `URL` and causes `load` to rerun if `true` is returned. This can be useful if you want to invalidate based on a pattern instead of a exact match.

```
// Example: Match '/path' regardless of the query parameters
import { invalidate } from '$app/navigation';

invalidate(url) => url.pathname === '/path');
```

```
function invalidate(
  resource: string | URL | ((url: URL) => boolean)
): Promise<void>;
```

invalidateAll

Causes all `load` functions belonging to the currently active page to re-run. Returns a `Promise` that resolves when the page is subsequently updated.

```
function invalidateAll(): Promise<void>;
```

onNavigate

A lifecycle function that runs the supplied `callback` immediately before we navigate to a new URL except during full-page navigations.

If you return a `Promise`, SvelteKit will wait for it to resolve before completing the navigation. This allows you to — for example — use `document.startViewTransition`. Avoid promises that are slow to resolve, since navigation will appear stalled to the user.

If a function (or a `Promise` that resolves to a function) is returned from the callback, it will be called once the DOM has updated.

`onNavigate` must be called during a component initialization. It remains active as long as the component is mounted.

```
function onNavigate(
  callback: (
    navigation: import('@sveltejs/kit').OnNavigate
  ) => MaybePromise<(() => void) | void>
): void;
```

preloadCode

Programmatically imports the code for routes that haven't yet been fetched. Typically, you might call this to speed up subsequent navigation.

You can specify routes by any matching pathname such as `/about` (to match `src/routes/about/+page.svelte`) or `/blog/*` (to match `src/routes/blog/[slug]/+page.svelte`).

Unlike `preloadData`, this won't call `load` functions. Returns a Promise that resolves when the modules have been imported.

```
function preloadCode(pathname: string): Promise<void>;
```

preloadData

Programmatically preloads the given page, which means

1. ensuring that the code for the page is loaded, and
2. calling the page's load function with the appropriate options.

This is the same behaviour that SvelteKit triggers when the user taps or mouses over an `<a>` element with `data-sveltekit-preload-data`. If the next navigation is to `href`, the values returned from load will be used, making navigation instantaneous. Returns a Promise that resolves with the result of running the new route's `load` functions once the preload is complete.

```
function preloadData(href: string): Promise<
    | {
        type: 'loaded';
        status: number;
        data: Record<string, any>;
      }
    | {
        type: 'redirect';
        location: string;
      }
>;
```

pushState

Programmatically create a new history entry with the given `page.state`. To use the current URL, you can pass `''` as the first argument. Used for shallow routing.

```
function pushState(
  url: string | URL,
  state: App.PageState
): void;
```

replaceState

Programmatically replace the current history entry with the given `page.state`. To use the current URL, you can pass `''` as the first argument. Used for shallow routing.

```
function replaceState(
  url: string | URL,
  state: App.PageState
): void;
```

On this page
$app/navigation
afterNavigate
beforeNavigate
disableScrollHandling
goto
invalidate
invalidateAll
onNavigate
preloadCode
preloadData
pushState
replaceState

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$app/paths

```
import { assets, base, resolveRoute } from '$app/paths';
```

On this page
$app/paths
assets
base
resolveRoute

assets

An absolute path that matches `config.kit.paths.assets`.

If a value for `config.kit.paths.assets` is specified, it will be replaced with `'/_svelte_kit_assets'` during `vite dev` or `vite preview`, since the assets don't yet live at their eventual URL.

```
let assets:
  | ''
  | `https://${string}`
  | `http://${string}`
  | '/_svelte_kit_assets';
```

base

A string that matches `config.kit.paths.base`.

Example usage: `<a href="{base}/your-page">Link</a>`

```
let base: '' | `/${string}`;
```

resolveRoute

Populate a route ID with params to resolve a pathname.

```
import { resolveRoute } from '$app/paths';

resolveRoute(
  `/blog/[slug]/[...somethingElse]`,
  {
    slug: 'hello-world',
    somethingElse: 'something/else'
  }
); // `/blog/hello-world/something/else`
```

```
function resolveRoute(
  id: string,
  params: Record<string, string | undefined>
): string;
```

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$app/server

On this page
$app/server
getRequestEvent
read

```
import { getRequestEvent, read } from '$app/server';
```

getRequestEvent

AVAILABLE SINCE 2.20.0

Returns the current `RequestEvent`. Can be used inside `handle`, `load` and actions (and functions called by them).

In environments without `AsyncLocalStorage`, this must be called synchronously (i.e. not after an `await`).

```
function getRequestEvent(): RequestEvent<
  Partial<Record<string, string>>,
  string | null
>;
```

read

AVAILABLE SINCE 2.4.0

Read the contents of an imported asset from the filesystem

```
import { read } from '$app/server';
import somefile from './somefile.txt';

const asset = read(somefile);
const text = await asset.text();
```

```
function read(asset: string): Response;
```

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$app/state

On this page
$app/state
navigating
page
updated

SvelteKit makes three read-only state objects available via the `$app/state` module — `page`, `navigating` and `updated`.

This module was added in 2.12. If you're using an earlier version of SvelteKit, use `$app/stores` instead.

```
import { navigating, page, updated } from '$app/state';
```

navigating

A read-only object representing an in-progress navigation, with `from`, `to`, `type` and (if `type === 'popstate'`) `delta` properties. Values are `null` when no navigation is occurring, or during server rendering.

```
const navigating:
  | import('@sveltejs/kit').Navigation
  | {
      from: null;
      to: null;
      type: null;
      willUnload: null;
      delta: null;
      complete: null;
    };
```

page

A read-only reactive object with information about the current page, serving several use cases:

- retrieving the combined `data` of all pages/layouts anywhere in your component tree (also see loading data)
- retrieving the current value of the `form` prop anywhere in your component tree (also see form actions)
- retrieving the page state that was set through `goto`, `pushState` or `replaceState` (also see goto and shallow routing)
- retrieving metadata such as the URL you're on, the current route and its parameters, and whether or not there was an error

+layout.svelte JS TS

```
<script lang="ts">
  import { page } from '$app/state';
</script>

<p>Currently at {page.url.pathname}</p>

{#if page.error}
  <span class="red">Problem detected</span>
{:else}
  <span class="small">All systems operational</span>
{/if}
```

Changes to `page` are available exclusively with runes. (The legacy reactivity syntax will not reflect any changes)

+page.svelte JS TS

```
<script lang="ts">
  import { page } from '$app/state';
  const id = $derived(page.params.id); // This will correctly update id for usage on thi
  $: badId = page.params.id; // Do not use; will never update after initial load
</script>
```

On the server, values can only be read during rendering (in other words *not* in e.g. `load` functions). In the browser, the values can be read at any time.

```
const page: import('@sveltejs/kit').Page;
```

updated

A read-only reactive value that's initially `false`. If `version.pollInterval` is a non-zero value, SvelteKit will poll for new versions of the app and update `current` to `true` when it detects one. `updated.check()` will force an immediate check, regardless of polling.

```
const updated: {
  get current(): boolean;
  check(): Promise<boolean>;
};
```

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$app/stores

On this page
$app/stores
getStores
navigating
page
updated

This module contains store-based equivalents of the exports from `$app/state`. If you're using SvelteKit 2.12 or later, use that module instead.

```
import { getStores, navigating, page, updated } from '$app/stores';
```

getStores

```
function getStores(): {
  page: typeof page;

  navigating: typeof navigating;

  updated: typeof updated;
};
```

navigating

DEPRECATED Use `navigating` from `$app/state` instead (requires Svelte 5, see docs for more info)

A readable store. When navigating starts, its value is a `Navigation` object with `from`, `to`, `type` and (if `type === 'popstate'`) `delta` properties. When navigating finishes, its value reverts to `null`.

On the server, this store can only be subscribed to during component initialization. In the browser, it can be subscribed to at any time.

```
const navigating: import('svelte/store').Readable<
  import('@sveltejs/kit').Navigation | null
>;
```

page

DEPRECATED Use `page` from `$app/state` instead (requires Svelte 5, see docs for more info)

A readable store whose value contains page data.

On the server, this store can only be subscribed to during component initialization. In the browser, it can be subscribed to at any time.

```
const page: import('svelte/store').Readable<
  import('@sveltejs/kit').Page
>;
```

updated

DEPRECATED Use `updated` from `$app/state` instead (requires Svelte 5, see docs for more info)

A readable store whose initial value is `false`. If `version.pollInterval` is a non-zero value, SvelteKit will poll for new versions of the app and update the store value to `true` when it detects one. `updated.check()` will force an immediate check, regardless of polling.

On the server, this store can only be subscribed to during component initialization. In the browser, it can be subscribed to at any time.

```
const updated: import('svelte/store').Readable<boolean> & {
  check(): Promise<boolean>;
};
```

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$env/dynamic/private

On this page
$env/dynamic/private

This module provides access to runtime environment variables, as defined by the platform you're running on. For example if you're using `adapter-node` (or running `vite preview`), this is equivalent to `process.env`. This module only includes variables that *do not* begin with `config.kit.env.publicPrefix` *and do* start with `config.kit.env.privatePrefix` (if configured).

This module cannot be imported into client-side code.

Dynamic environment variables cannot be used during prerendering.

```
import { env } from '$env/dynamic/private';
console.log(env.DEPLOYMENT_SPECIFIC_VARIABLE);
```

> In `dev`, `$env/dynamic` always includes environment variables from `.env`. In `prod`, this behavior will depend on your adapter.

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$env/dynamic/public

On this page
$env/dynamic/public

Similar to `$env/dynamic/private`, but only includes variables that begin with `config.kit.env.publicPrefix` (which defaults to `PUBLIC_`), and can therefore safely be exposed to client-side code.

Note that public dynamic environment variables must all be sent from the server to the client, causing larger network requests — when possible, use `$env/static/public` instead.

Dynamic environment variables cannot be used during prerendering.

```
import { env } from '$env/dynamic/public';
console.log(env.PUBLIC_DEPLOYMENT_SPECIFIC_VARIABLE);
```

Edit this page on GitHub

PREVIOUS
$env/dynamic/private

NEXT
$env/static/private

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$env/static/private

On this page
$env/static/private

Environment variables loaded by Vite from `.env` files and `process.env`. Like `$env/dynamic/private`, this module cannot be imported into client-side code. This module only includes variables that *do not* begin with `config.kit.env.publicPrefix` *and do* start with `config.kit.env.privatePrefix` (if configured).

Unlike `$env/dynamic/private`, the values exported from this module are statically injected into your bundle at build time, enabling optimisations like dead code elimination.

```
import { API_KEY } from '$env/static/private';
```

Note that all environment variables referenced in your code should be declared (for example in an `.env` file), even if they don't have a value until the app is deployed:

```
MY_FEATURE_FLAG=""
```

You can override `.env` values from the command line like so:

```
MY_FEATURE_FLAG="enabled" npm run dev
```

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$env/static/public

On this page
$env/static/public

Similar to `$env/static/private`, except that it only includes environment variables that begin with `config.kit.env.publicPrefix` (which defaults to `PUBLIC_`), and can therefore safely be exposed to client-side code.

Values are replaced statically at build time.

```
import { PUBLIC_BASE_URL } from '$env/static/public';
```

Edit this page on GitHub

PREVIOUS
$env/static/private

NEXT
$lib

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$lib

SvelteKit automatically makes files under `src/lib` available using the `$lib` import alias. You can change which directory this alias points to in your config file.

src/lib/Component.svelte

```
A reusable component
```

src/routes/+page.svelte JS TS

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

<Component />
```

Edit this page on GitHub

PREVIOUS
$env/static/public

NEXT
$service-worker

On this page
$lib

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

$service-worker

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

This module is only available to service workers.

On this page
$service-worker
base
build
files
prerendered
version

base

The `base` path of the deployment. Typically this is equivalent to `config.kit.paths.base`, but it is calculated from `location.pathname` meaning that it will continue to work correctly if the site is deployed to a subdirectory. Note that there is a `base` but no `assets`, since service workers cannot be used if `config.kit.paths.assets` is specified.

```
const base: string;
```

build

An array of URL strings representing the files generated by Vite, suitable for caching with `cache.addAll(build)`. During development, this is an empty array.

```
const build: string[];
```

files

An array of URL strings representing the files in your static directory, or whatever directory is specified by `config.kit.files.assets`. You can customize which files are included from `static` directory using `config.kit.serviceWorker.files`

```
const files: string[];
```

prerendered

An array of pathnames corresponding to prerendered pages and endpoints. During development, this is an empty array.

```
const prerendered: string[];
```

version

See `config.kit.version`. It's useful for generating unique cache names inside your service worker, so that a later deployment of your app can invalidate old caches.

```
const version: string;
```

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT · REFERENCES

@sveltejs/kit

```
import {
  Server,
  VERSION,
  error,
  fail,
  isActionFailure,
  isHttpError,
  isRedirect,
  json,
  normalizeUrl,
  redirect,
  text
} from '@sveltejs/kit';
```

Server

```
class Server {…}
```

```
constructor(manifest: SSRManifest);
```

```
init(options: ServerInitOptions): Promise<void>;
```

```
respond(request: Request, options: RequestOptions): Promise<Response>;
```

VERSION

```
const VERSION: string;
```

error

Throws an error with a HTTP status code and an optional message. When called during request handling, this will cause SvelteKit to return an error response without invoking `handleError`. Make sure you're not catching the thrown error, which would prevent SvelteKit from handling it.

```
function error(status: number, body: App.Error): never;
```

```
function error(
  status: number,
  body? : {
    message: string;
  } extends App.Error
    ? App.Error | string | undefined
    : never
); never;
```

fail

Create an `ActionFailure` object.

```
function fail(status: number): ActionFailure<undefined>;
```

```
function fail(
  T extends Record<string, unknown> | undefined = undefined
,<status: number, data: T): ActionFailure<T>;
```

isActionFailure

Checks whether this is an action failure thrown by `fail`.

```
function isActionFailure(e: unknown): e is ActionFailure;
```

isHttpError

Checks whether this is an error thrown by `error`.

```
function isHttpError<T extends number>(
  e: unknown,
  status? : T | undefined
): e is HttpError_1 & {
  status: T extends undefined ? never : T;
};
```

isRedirect

Checks whether this is a redirect thrown by `redirect`.

```
function isRedirect(e: unknown): e is Redirect_1;
```

json

Create a JSON `Response` object from the supplied data.

```
function json(
  data: any,
  init? : ResponseInit | undefined
): Response;
```

normalizeUrl

AVAILABLE SINCE 2.18.0

Strips possible SvelteKit-internal suffixes and trailing slashes from the URL pathname. Returns the normalized URL as well as a method for adding the potential suffix back based on a new pathname (possibly including search) or URL.

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

const { url, denormalize } = normalizeUrl('/blog/post/__data.json');
console.log(url.pathname); // /blog/post
console.log(denormalize('/blog/post/a')); // /blog/post/a/__data.json
```

```
function normalizeUrl(url: URL | string): {
  url: URL;
  wasNormalized: boolean;
  denormalize: (url?: string | URL) => URL;
};
```

redirect

Redirect a request. When called during request handling, SvelteKit will return a redirect response. Make sure you're not catching the thrown redirect, which would prevent SvelteKit from handling it.

Most common status codes:

- `303 See Other` : redirect as a GET request (often used after a form POST request)
- `307 Temporary Redirect` : redirect will keep the request method
- `308 Permanent Redirect` : redirect will keep the request method, SEO will be transferred to the new page

See all redirect status codes

```
function redirect(
  status:
    | 300
    | 301
    | 302
    | 303
    | 304
    | 305
    | 306
    | 307
    | 308
    | (3 & number),
  location: string | URL
); never;
```

text

Create a `Response` object from the supplied body.

```
function text(
  body: string,
  init? : ResponseInit | undefined
); Response;
```

Action

Shape of a form action method that is part of `export const actions = {…}` in `+page.server.js`. See form actions for more information.

```
type Action<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  OutputData extends Record<string, any> | void = Record<
    string,
    any
  > | void,
  RouteId extends string | null = string | null
> = (
  event: RequestEvent<Params, RouteId>
) => MaybePromise<OutputData>;
```

ActionFailure

```
interface ActionFailure<
  T extends Record<string, unknown> | undefined = undefined
> {…}
```

```
status: number;
```

```
data: T;
```

```
[uniqueSymbol]: true;
```

ActionResult

When calling a form action via fetch, the response will be one of these shapes.

```
<form method="post" use:enhance={() => {
  return ({ result }) => {
    // result is of type ActionResult
  };
}}>
```

```
type ActionResult<
  Success extends
    | Record<string, unknown>
    | undefined = Record<string, any>,
  Failure extends
    | Record<string, unknown>
    | undefined = Record<string, any>
> =
  | { type: 'success'; status: number; data?: Success }
  | { type: 'failure'; status: number; data?: Failure }
  | { type: 'redirect'; status: number; location: string }
  | { type: 'error'; status?: number; error: any };
```

Actions

Shape of the `export const actions = {…}` object in `+page.server.js`. See form actions for more information.

```
type Actions<
  Params extends Partial<Record<string, string>> = Partial<
    Record<string, string>
  >,
  OutputData extends Record<string, any> | void = Record<
    string,
    any
  > | void,
  RouteId extends string | null = string | null
> = Record<string, Action<Params, OutputData, RouteId>>;
```

Adapter

Adapters are responsible for taking the production build and turning it into something that can be deployed to a platform of your choosing.

```
interface Adapter {…}
```

```
name: string;
```

The name of the adapter, using for logging. Will typically correspond to the package name.

```
adapt: (builder: Builder) => MaybePromise<void>;
```

`builder` An object provided by SvelteKit that contains methods for adapting the app

This function is called after SvelteKit has built your app.

```
supports?: {…}
```

Checks called during dev and build to determine whether specific features will work in production with this adapter

```
read?: (details: { config: any; route: { id: string } }) => boolean;
```

`config` The merged route config

Test support for `read` from `$app/server`

```
emulate?: () => MaybePromise<Emulator>;
```

On this page

- @sveltejs/kit
- Server
- VERSION
- error
- fail
- isActionFailure
- isHttpError
- isRedirect
- json
- normalizeUrl
- redirect
- text
- Action
- ActionFailure
- ActionResult
- Actions
- Adapter
- AfterNavigate
- AwaitedActions
- BeforeNavigate
- Builder
- ClientInit
- Cookie

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

@sveltejs/kit/hooks

On this page
@sveltejs/kit/hooks
sequence

```
import { sequence } from '@sveltejs/kit/hooks';
```

sequence

A helper function for sequencing multiple `handle` calls in a middleware-like manner. The behavior for the `handle` options is as follows:

- `transformPageChunk` is applied in reverse order and merged
- `preload` is applied in forward order, the first option "wins" and no `preload` options after it are called
- `filterSerializedResponseHeaders` behaves the same as `preload`

src/hooks.server.ts JS TS

```
import { sequence } from '@sveltejs/kit/hooks';
import type { Handle } from '@sveltejs/kit';

const first: Handle = async ({ event, resolve }) => {
  console.log('first pre-processing');
  const result = await resolve(event, {
    transformPageChunk: ({ html }) => {
      // transforms are applied in reverse order
      console.log('first transform');
      return html;
    },
    preload: () => {
      // this one wins as it's the first defined in the chain
      console.log('first preload');
      return true;
    }
  });
  console.log('first post-processing');
  return result;
};
const second: Handle = async ({ event, resolve }) => {
  console.log('second pre-processing');
  const result = await resolve(event, {
    transformPageChunk: ({ html }) => {
      console.log('second transform');
      return html;
    },
    preload: () => {
      console.log('second preload');
      return true;
    },
    filterSerializedResponseHeaders: () => {
      // this one wins as it's the first defined in the chain
      console.log('second filterSerializedResponseHeaders');
      return true;
    }
  });
  console.log('second post-processing');
  return result;
};

export const handle = sequence(first, second);
```

The example above would print:

```
first pre-processing
first preload
second pre-processing
second filterSerializedResponseHeaders
second transform
first transform
second post-processing
first post-processing
```

```
function sequence(
  ...handlers: import('@sveltejs/kit').Handle[]
): import('@sveltejs/kit').Handle;
```

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

@sveltejs/kit/node

```
import {
  createReadableStream,
  getRequest,
  setResponse
} from '@sveltejs/kit/node';
```

On this page
@sveltejs/kit/node
createReadableStream
getRequest
setResponse

createReadableStream

AVAILABLE SINCE 2.4.0

Converts a file on disk to a readable stream

```
function createReadableStream(file: string): ReadableStream;
```

getRequest

```
function getRequest({
  request,
  base,
  bodySizeLimit
}: {
  request: import('http').IncomingMessage;
  base: string;
  bodySizeLimit?: number;
}): Promise<Request>;
```

setResponse

```
function setResponse(
  res: import('http').ServerResponse,
  response: Response
): Promise<void>;
```

Edit this page on GitHub

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

@sveltejs/kit/node/polyfills

```
import { installPolyfills } from '@sveltejs/kit/node/polyfills';
```

installPolyfills

Make various web APIs available as globals:

- `crypto`
- `File`

```
function installPolyfills(): void;
```

Edit this page on GitHub

PREVIOUS
@sveltejs/kit/hooks

NEXT
@sveltejs/kit/node

On this page

@sveltejs/kit/node/polyfills
installPolyfills

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

@sveltejs/kit/vite

```
import { sveltekit } from '@sveltejs/kit/vite';
```

sveltekit

Returns the SvelteKit Vite plugins.

```
function sveltekit(): Promise<import('vite').Plugin[]>;
```

Edit this page on GitHub

On this page
@sveltejs/kit/vite
sveltekit

Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node

SVELTEKIT • BEST PRACTICES

Accessibility

On this page
Accessibility
Route announcements
Focus management
The "lang" attribute
Further reading

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:

src/routes/+page.svelte

```
<svelte:head>
  <title>Todo List</title>
</svelte:head>
```

This will allow screen readers and other assistive technology to identify the new page after a navigation occurs. Providing a descriptive title is also important for SEO.

Focus management

In traditional server-rendered applications, every navigation will reset focus to the top of the page. This ensures that people browsing the web with a keyboard or screen reader will start interacting with the page from the beginning.

To simulate this behavior during client-side routing, SvelteKit focuses the `<body>` element after each navigation and enhanced form submission. There is one exception - if an element with the `autofocus` attribute is present, SvelteKit will focus that element instead. Make sure to consider the implications for assistive technology when using that attribute.

If you want to customize SvelteKit's focus management, you can use the `afterNavigate` hook:

```
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:

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:

src/app.html

```
<html lang="%lang%">
```

src/hooks.server.ts JS TS

```
import type { Handle } from '@sveltejs/kit';

export const handle: 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)

Edit this page on GitHub

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Adapters

On this page
Adapters
Using adapters
Platform-specific context

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

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

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

Additional community-provided adapters exist for other platforms.

Using adapters

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

svelte.config.js

```
import adapter from 'svelte-adapter-foo';

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

export default config;
```

Platform-specific context

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

Winning adopters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT · ADVANCED

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...

```
/files/[owner]/[repo]/[branch]/[...file]
```

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

```
{
  owner: 'sveltejs',
  repo: 'kit',
  branch: 'main',
  file: 'documentation/docs/04-advanced-routing.md'
}
```

💡 `src/routes/a/[...rest]/+page.svelte` will match `/a/z` (i.e. there's no parameter at all) as well as `/a/b/c` and `/a/b/c/d` 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/hurl`, because no route was matched. If you want to render the nested error page, you should create a route that matches any `/marx-brothers/*` request, and return a 404 from it:

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

```
src/routes/marx-brothers/[...path]/+page.ts
```

```
import { error } from '@sveltejs/kit';
import type { PageLoad } from './$types';

export const load: PageLoad = (event) => {
  error(404, 'Not found');
};
```

💡 If you don't handle 404 cases, they will appear in `$page.error`.

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 untaxed.

Matching

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

```
src/params/fruits
```

```
import type { ParamMatcher } from '@sveltejs/kit';

export const match = ({param: string}: param is {param: 'apple' | 'orange'}) => {
  return param === 'apple' || param === 'orange';
}) satisfies ParamMatcher;
```

...and augmenting your routes:

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

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

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

💡 Matchers run both on the server and in the browser.

Sorting

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

```
src/routes/[...catchall]/+page.svelte
src/routes/[[a=z]]/+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=z]]/+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+25]`
- `/` — `[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 `/weird:/:?` route, you would create a `src/routes/weird:[x+3a]:[x+3f]+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+d83a][u+dd1a]/+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 prefers `.svelte` with directories with a leading `_` character, you may find it useful to encode these characters when creating e.g. `$api/v1/user` routes: `src/routes/[v=v1]api[u+5f]user/...`

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 (`/about` 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 `(render children()).` 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
```

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
```

On this page

Advanced routing
Rest parameters
Optional parameters
Matching
Sorting
Encoding
Advanced layouts
Further reading

Server only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node

SVELTEKIT • BEST PRACTICES

Auth

Auth refers to authentication and authorization, which are common needs when building a web application. Authentication means verifying that the user is who they say they are based on their provided credentials. Authorization means determining which actions they are allowed to take.

On this page
Auth
Sessions vs tokens
Integration points
Guides

Sessions vs tokens

After the user has provided their credentials such as a username and password, we want to allow them to use the application without needing to provide their credentials again for future requests. Users are commonly authenticated on subsequent requests with either a session identifier or signed token such as a JSON Web Token (JWT).

Session IDs are most commonly stored in a database. They can be immediately revoked, but require a database query to be made on each request.

In contrast, JWT generally are not checked against a datastore, which means they cannot be immediately revoked. The advantage of this method is improved latency and reduced load on your datastore.

Integration points

Auth cookies can be checked inside server hooks. If a user is found matching the provided credentials, the user information can be stored in `locals`.

Guides

Lucia is a good reference for session-based web app auth. It contains example code snippets and projects for implementing session-based auth within SvelteKit and other JS projects. You can add code which follows the Lucia guide to your project with `npx sv create` when creating a new project or `npx sv add lucia` for an existing project.

An auth system is tightly coupled to a web framework because most of the code lies in validating user input, handling errors, and directing users to the appropriate next page. As a result, many of the generic JS auth libraries include one or more web frameworks within them. For this reason, many users will find it preferrable to follow a SvelteKit-specific guide such as the examples found in Lucia rather than having multiple web frameworks inside their project.

Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private

SVELTEKIT • APPENDIX

Breakpoint Debugging

On this page
Breakpoint Debugging
Visual Studio Code
Other Editors
Google Chrome and Microsoft Edge
Developer Tools
References

In addition to the `@debug` tag, you can also debug Svelte and SvelteKit projects using breakpoints within various tools and development environments. This includes both frontend and backend code.

The following guides assume your JavaScript runtime environment is Node.js.

Visual Studio Code

With the built-in debug terminal, you can set up breakpoints in source files within VSCode.

1. Open the command palette: `CMD/Ctrl` + `Shift` + `P`.
2. Find and launch "Debug: JavaScript Debug Terminal".
3. Start your project using the debug terminal. For example: `npm run dev`.
4. Set some breakpoints in your client or server-side source code.
5. Trigger the breakpoint.

Launch via debug pane

You may alternatively set up a `.vscode/launch.json` in your project. To set one up automatically:

1. Go to the "Run and Debug" pane.
2. In the "Run" select menu, choose "Node.js...".
3. Select the "run script" that corresponds to your project, such as "Run script: dev".
4. Press the "Start debugging" play button, or hit `F5` to begin breakpoint debugging.

Here's an example `launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "command": "npm run dev",
      "name": "Run development server",
      "request": "launch",
      "type": "node-terminal"
    }
  ]
}
```

Further reading: https://code.visualstudio.com/docs/editor/debugging.

Other Editors

If you use a different editor, these community guides might be useful for you:

- WebStorm Svelte: Debug Your Application
- Debugging JavaScript Frameworks in Neovim

Google Chrome and Microsoft Edge Developer Tools

It's possible to debug Node.js applications using a browser-based debugger.

> Note this only works with debugging client-side SvelteKit source maps.

1. Run the `--inspect` flag when starting the Vite server with Node.js. For instance: `NODE_OPTIONS="--inspect" npm run dev`
2. Open your site in a new tab. Typically at `localhost:5173`.
3. Open your browser's dev tools, and click on the "Open dedicated DevTools for Node.js" icon near the top-left. It should display the Node.js logo.
4. Set up breakpoints and debug your application.

You may alternatively open the debugger devtools by navigating to `chrome://inspect` in Google Chrome, or `edge://inspect` in Microsoft Edge.

References

- Debugging Node.js

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Building your app

On this page
Building your app
During the build
Preview your app

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

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

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

During the build

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

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

if (!building) {
  setupMyDatabase();
}

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

Preview your app

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

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Cloudflare Pages

On this page
Cloudflare Pages
Comparisons
Usage
Options
Deployment
Runtime APIs
Notes
Troubleshooting

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

This adapter will be installed by default when you use `adapter-auto`. If you plan on staying with Cloudflare Pages, you can switch from `adapter-auto` to using this adapter directly so that `event.platform` is emulated during local development, type declarations are automatically applied, and the ability to set Cloudflare-specific options is provided.

Comparisons

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

Usage

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

svelte.config.js

```
import adapter from '@sveltejs/adapter-cloudflare';

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

Options

routes

Allows you to customise the `_routes.json` file generated by `adapter-cloudflare`.

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

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

platformProxy

Preferences for the emulated `platform.env` local bindings. See the getPlatformProxy Wrangler API documentation for a full list of options.

Deployment

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

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

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

Runtime APIs

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

```
export async function POST({ request, platform }) {

  const x = platform.env.YOUR_DURABLE_OBJECT_NAMESPACE.idFromName('x');
}
```

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

To make these types available to your app, install `@cloudflare/workers-types` and reference them in your `src/app.d.ts`:

src/app.d.ts

```
import { KVNamespace, DurableObjectNamespace } from "@cloudflare/workers-types";

declare global {
  namespace App {
    interface Platform {
      env?: {
        YOUR_KV_NAMESPACE: KVNamespace;
        YOUR_DURABLE_OBJECT_NAMESPACE: DurableObjectNamespace;
      };
    }
  }
}

export {};
```

Testing Locally

Cloudflare Workers specific values in the `platform` property are emulated during dev and preview modes. Local bindings are created based on your Wrangler configuration file and are used to populate `platform.env` during development and preview. Use the adapter config `platformProxy` option to change your preferences for the bindings.

For testing the build, you should use Wrangler version 3. Once you have built your site, run `wrangler pages dev .svelte-kit/cloudflare`.

Notes

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

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

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

Troubleshooting

Further reading

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

Node.js compatibility

If you would like to enable Node.js compatibility, you can add the `nodejs_compat` compatibility flag to your Wrangler configuration file:

wrangler.jsonc

```
{
  "compatibility_flags": ["nodejs_compat"]
}
```

Worker size limits

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

Accessing the file system

You can't use `fs` in Cloudflare Workers — you must prerender the routes in question.

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Cloudflare Workers

On this page
Cloudflare Workers
Usage
Options
Basic Configuration
Runtime APIs
Troubleshooting

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

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

Usage

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

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

export default {
  kit: {
    adapter: adapter({
      // see below for options that can be set here
    })
  }
};
```

Options

config

Path to your Wrangler configuration file. If you would like to use a Wrangler configuration filename other than `wrangler.jsonc`, you can specify it using this option.

platformProxy

Preferences for the emulated `platform.env` local bindings. See the getPlatformProxy Wrangler API documentation for a full list of options.

Basic Configuration

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

wrangler.jsonc
```
{
  "name": "<your-service-name>",
  "account_id": "<your-account-id>",
  "main": "./.cloudflare/worker.js",
  "site": {
    "bucket": "./.cloudflare/public"
  },
  "build": {
    "command": "npm run build"
  },
  "compatibility_date": "2021-11-12"
}
```

`<your-service-name>` can be anything. `<your-account-id>` can be found by running `wrangler whoami` using the Wrangler CLI tool or by logging into your Cloudflare dashboard and grabbing it from the end of the URL:

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

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

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

```
npm i -D wrangler
wrangler login
```

Then, you can build your app and deploy it:

```
wrangler deploy
```

Runtime APIs

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

```
export async function POST({ request, platform }) {

  const x = platform.env.YOUR_DURABLE_OBJECT_NAMESPACE.idFromName('x');
}
```

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

To make these types available to your app, install `@cloudflare/workers-types` and reference them in your `src/app.d.ts`:

src/app.d.ts
```
import { KVNamespace, DurableObjectNamespace } from '@cloudflare/workers-types';

declare global {
  namespace App {
    interface Platform {
      env?: {
        YOUR_KV_NAMESPACE: KVNamespace;
        YOUR_DURABLE_OBJECT_NAMESPACE: DurableObjectNamespace;
      };
    }
  }
}

export {};
```

Testing Locally

Cloudflare Workers specific values in the `platform` property are emulated during dev and preview modes. Local bindings are created based on your Wrangler configuration file and are used to populate `platform.env` during development and preview. Use the adapter config `platformProxy` option to change your preferences for the bindings.

For testing the build, you should use Wrangler version 3. Once you have built your site, run `wrangler dev`.

Troubleshooting

Node.js compatibility

If you would like to enable Node.js compatibility, you can add the `nodejs_compat` compatibility flag to your Wrangler configuration file:

wrangler.jsonc
```
{
  "compatibility_flags": ["nodejs_compat"]
}
```

Worker size limits

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

Accessing the file system

You can't use `fs` in Cloudflare Workers — you must prerender the routes in question.

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT • REFERENCE

Command Line Interface

On this page
Command Line Interface
svelte-kit sync

SvelteKit projects use Vite, meaning you'll mostly use its CLI (albeit via `npm run dev/build/preview` scripts):

- `vite dev` — start a development server
- `vite build` — build a production version of your app
- `vite preview` — run the production version locally

However SvelteKit includes its own CLI for initialising your project:

svelte-kit sync

`svelte-kit sync` creates the `tsconfig.json` and all generated types (which you can import as `./$types` inside routing files) for your project. When you create a new project, it is listed as the `prepare` script and will be run automatically as part of the npm lifecycle, so you should not ordinarily have to run this command.

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT · REFERENCE

Configuration

Your project's configuration lives in a `svelte.config.js` file at the root of your project. As well as SvelteKit, this config object is used by other tooling that integrates with Svelte such as editor extensions.

```
svelte.config.js
import adapter from '@sveltejs/adapter-auto';

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

export default config;
```

Config

```
interface Config {…}
```

```
compilerOptions?: CompilerOptions;
```

DEFAULT `{}`

Options passed to `svelte.compile`.

```
extensions?: string[];
```

DEFAULT `["\.svelte"]`

List of file extensions that should be treated as Svelte files.

```
kit?: KitConfig;
```

SvelteKit options

```
preprocess?: any;
```

Preprocessor options, if any. Preprocessing can alternatively also be done through Vite's preprocessor capabilities.

```
vitePlugin?: PluginOptions;
```

`vite-plugin-svelte` plugin options.

```
[key: string]: any;
```

Any additional options required by tooling that integrates with Svelte.

KitConfig

The `kit` property configures SvelteKit, and can have the following properties:

adapter

DEFAULT `undefined`

Your adapter is run when executing `vite build`. It determines how the output is converted for different platforms.

alias

DEFAULT `{}`

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

```
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 `.svelte-kit/tsconfig.json` or `jsconfig.json`.

appDir

DEFAULT `"_app"`

The directory where SvelteKit keeps its stuff, including static assets (such as JS and CSS) and internally-used routes.

If `paths.assets` is specified, there will be two app directories — `${paths.assets}/${appDir}` and `${paths.base}/${appDir}`.

csp

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...

```
svelte.config.js
/** @type {import('@sveltejs/kit').Config} */
const config = {
	kit: {
	  csp: {
		directives: {
		  'script-src': ['self']
		},
		// must be specified with either the 'report-uri' or 'report-to' directives, or b
		reportOnly: {
		  'script-src': ['self']
		  'report-uri': ['/']
		}
	  }
	}
};

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 `src/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 prerendered 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`.

If this level of configuration is insufficient and you have more dynamic requirements, you can use the `handle` hook to roll your own CSP.

```
mode?: 'hash' | 'nonce' | 'auto';
```

Whether to use hashes or nonces to restrict `<script>` and `<style>` elements. `'auto'` will use hashes for prerendered pages, and nonces for dynamically rendered pages.

```
directives?: CspDirectives;
```

Directives that will be added to `Content-Security-Policy` headers.

```
reportOnly?: CspDirectives;
```

Directives that will be added to `Content-Security-Policy-Report-Only` headers.

csrf

Protection against cross-site request forgery (CSRF) attacks.

```
checkOrigin?: boolean;
```

DEFAULT `true`

Whether to check the incoming `origin` header for `POST`, `PUT`, `PATCH`, or `DELETE` form submissions and verify that it matches the server's origin.

To allow people to make `POST`, `PUT`, `PATCH`, or `DELETE` requests with a `Content-Type` of `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain` to your app from other origins, you will need to disable this option. Be careful!

embedded

DEFAULT `false`

Whether or not the app is embedded inside a larger app. If `true`, SvelteKit will add its event listeners related to navigation etc on the parent of `%sveltekit.body%` instead of `window`, and will pass `params` from the server rather than inferring them from `location.pathname`. Note that it is generally not supported to embed multiple SvelteKit apps on the same page and use client-side SvelteKit features within them (things such as pushing to the history state assume a single instance).

env

Environment variable configuration

```
dir?: string;
```

DEFAULT `"."`

The directory to search for `.env` files.

```
publicPrefix?: string;
```

DEFAULT `"PUBLIC_"`

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.

```
privatePrefix?: string;
```

DEFAULT `""`

AVAILABLE SINCE v1.21.0

A prefix that signals that an environment variable is unsafe to expose to client-side code. Environment variables matching neither the public nor the private prefix will be discarded completely. See `$env/static/private` and `$env/dynamic/private`.

files

Where to find various files within your project.

```
assets?: string;
```

DEFAULT `"static"`

a place to put static files that should have stable URLs and undergo no processing, such as `favicon.ico` or `manifest.json`.

```
hooks?: {…}
```

```
client?: string;
```

DEFAULT `"src/hooks.client"`

The location of your client hooks.

```
server?: string;
```

DEFAULT `"src/hooks.server"`

The location of your server hooks.

```
universal?: string;
```

DEFAULT `"src/hooks"`

AVAILABLE SINCE v2.3.0

The location of your universal hooks.

```
lib?: string;
```

DEFAULT `"src/lib"`

On this page

Configuration
Config
KitConfig
adapter
alias
appDir
csp
csrf
embedded
env
files
inlineStyleThreshold
moduleExtensions
outDir
output
paths
prerender
router
serviceWorker
typescript
version

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT • GETTING STARTED

Creating a project

On this page
Creating a project
Editor setup

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

```
npx sv create my-app
cd my-app
npm install
npm run dev
```

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

There are two basic concepts:

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

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

Editor setup

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

Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT · ADVANCED

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.

On this page
Errors
Error objects
Expected errors
Unexpected errors
Responses
Type safety
Further reading

Error objects

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

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

Expected errors

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

src/routes/blog/[slug]/+page.server.ts

```
import { error } from '@sveltejs/kit';
import * as db from '$lib/server/database';
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async ({ params }) => {
  const post = await db.getPost(params.slug);

  if (!post) {
    error(404, {
      message: 'Not found'
    });
  }

  return { post };
};
```

This throws an exception that SvelteKit catches, causing it to set the response status code to 404 and render an `+error.svelte` component, where `page.error` is the object provided as the second argument to `error(...)`.

src/routes/+error.svelte

```
<script lang="ts">
  import { page } from '$app/state';
</script>

<h1>{page.error.message}</h1>
```

Legacy mode hide all

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

You can add extra properties to the error object if needed...

```
error(404, {
  message: 'Not found',
  code: 'NOT_FOUND'
});
```

...otherwise, for convenience, you can pass a string as the second argument:

```
error(404, { message: 'Not found' });
error(404, 'Not found');
```

In SvelteKit 1.x you had to `throw` the `error` yourself

Unexpected errors

An *unexpected* error is any other exception that occurs while handling a request. Since these can contain sensitive information, unexpected error messages and stack traces are not exposed to users.

By default, unexpected errors are printed to the console (or, in production, your server logs), while the error that is exposed to the user has a generic shape:

```
{ "message": "Internal Error" }
```

Unexpected errors will go through the `handleError` hook, where you can add your own error handling — for example, sending errors to a reporting service, or returning a custom error object which becomes `$page.error`.

Responses

If an error occurs inside `handle` or inside a `+server.js` request handler, SvelteKit will respond with either a fallback error page or a JSON representation of the error object, depending on the request's `Accept` headers.

You can customise the fallback error page by adding a `src/error.html` file:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>%sveltekit.error.message%</title>
  </head>
  <body>
    <h1>My custom error page</h1>
    <p>Status: %sveltekit.status%</p>
    <p>Message: %sveltekit.error.message%</p>
  </body>
</html>
```

SvelteKit will replace `%sveltekit.status%` and `%sveltekit.error.message%` with their corresponding values.

If the error instead occurs inside a `load` function while rendering a page, SvelteKit will render the `+error.svelte` component nearest to where the error occurred. If the error occurs inside a `load` function in `+layout(.server).js`, the closest error boundary in the tree is an `+error.svelte` file *above* that layout (not next to it).

The exception is when the error occurs inside the root `+layout.js` or `+layout.server.js`, since the root layout would ordinarily *contain* the `+error.svelte` component. In this case, SvelteKit uses the fallback error page.

Type safety

If you're using TypeScript and need to customize the shape of errors, you can do so by declaring an `App.Error` interface in your app (by convention, in `src/app.d.ts`, though it can live anywhere that TypeScript can 'see'):

src/app.d.ts

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

export {};
```

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

Further reading

- Tutorial: Errors and redirects
- Tutorial: Hooks

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT / CORE CONCEPTS

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.

On this page
Form actions
Default actions
Named actions
Anatomy of an action
Loading data
Progressive enhancement
Alternatives
GET vs POST
Further reading

Default actions

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

src/routes/login/+page.server.ts
```
import type { Actions } from './$types';

export const actions = {
  default: async (event) => {
    // TODO log the user in
  }
} satisfies Actions;
```

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

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

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

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

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

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:

src/routes/login/+page.server.ts
```
import type { Actions } from './$types';

export const actions = {
  default: async (event) => {
  login: async (event) => {
    // TODO log the user in
  },
  register: async (event) => {
    // TODO register the user
  }
} satisfies Actions;
```

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

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

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>`:

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

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

Anatomy of an action

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

src/routes/login/+page.server.ts
```
import * as db from '$lib/server/db';
import type { PageServerLoad, Actions } from './$types';

export const load: PageServerLoad = async ({ cookies }) => {
  const user = await db.getUserFromSession(cookies.get('sessionid'));
  return { user };
};
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), { path: '/' });

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

src/routes/login/+page.svelte
```
<script lang="ts">
  import type { PageProps } from './$types';

  let { data, form }: PageProps = $props();
</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}
```

Legacy mode hide all

`PageProps` was added in 2.16.0. In earlier versions, you had to type the `data` and `form` properties individually:

+page.svelte
```
  import type { PageData, ActionData } from './$types';

  let { data, form }: { data: PageData, form: ActionData } = $props();
```

In Svelte 4, you'd use `export let data` and `export let form` instead to declare properties.

Validation errors

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

src/routes/login/+page.server.ts
```
import { fail } from '@sveltejs/kit';
import * as db from '$lib/server/db';
import type { Actions } from './$types';

export const actions = {
  login: async ({ cookies, request }) => {
    const data = await request.formData();
    const email = data.get('email');
    const password = data.get('password');

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

    const user = await db.getUser(email);

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

    cookies.set('sessionid', await db.createSession(user), { path: '/' });

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

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

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

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

Redirects

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

src/routes/login/+page.server.ts
```
import { fail, redirect } from '@sveltejs/kit';
import * as db from '$lib/server/db';
import type { Actions } from './$types';

export const actions = {
  login: async ({ cookies, request, url }) => {
    const data = await request.formData();
    const email = data.get('email');
    const password = data.get('password');

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

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

    cookies.set('sessionid', await db.createSession(user), { path: '/' });

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

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

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 return 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:

src/hooks.server.ts
```
import type { Handle } from '@sveltejs/kit';

export const handle: Handle = async ({ event, resolve }) => {
  event.locals.user = await getUser(event.cookies.get('sessionid'));
  return resolve(event);
};
```

src/routes/account/+page.server.ts
```
import type { PageServerLoad, Actions } from './$types';

export const load: PageServerLoad = (event) => {
  return {
    user: event.locals.user
  };
};
export const actions = {
  logout: async (event) => {
    event.cookies.delete('sessionid', { path: '/' });
    event.locals.user = null;
```

Frequently asked questions

Other resources

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

What can I make with SvelteKit?

See the documentation regarding project types for more details.

How do I include details from package.json in my application?

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:

```
svelte.config.js
import pkg from './package.json' with { type: 'json' };
```

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

Most issues related to including a library are due to incorrect packaging. You can check if a library's packaging is compatible with Node.js by entering it into the publint website.

Here are a few things to keep in mind when checking if a library is packaged correctly:

- `exports` takes precedence over the other entry point fields such as `main` and `module`. Adding an `exports` field may not be backwards-compatible as it prevents deep imports.
- ESM files should end with `.mjs` unless `"type": "module"` is set in which any case CommonJS files should end with `.cjs`.
- `main` should be defined if `exports` is not. It should be either a CommonJS or ESM file and adhere to the previous bullet. If a `module` field is defined, it should refer to an ESM file.
- Svelte components should be distributed as uncompiled `.svelte` files with any JS in the package written as ESM only. Custom script and style languages, like TypeScript and SCSS, should be preprocessed as vanilla JS and CSS respectively. We recommend using svelte-package for packaging Svelte libraries, which will do this for you.

Libraries work best in the browser with Vite when they distribute an ESM version, especially if they are dependencies of a Svelte component library. You may wish to suggest to library authors that they provide an ESM version. However, CommonJS (CJS) dependencies should work as well since, by default, vite-plugin-svelte will ask Vite to pre-bundle them using `esbuild` to convert them to ESM.

If you are still encountering issues we recommend searching both the Vite issue tracker and the issue tracker of the library in question. Sometimes issues can be worked around by fiddling with the `optimizeDeps` or `ssr` config values though we recommend this as only a short-term workaround in favor of fixing the library in question.

How do I use the view transitions API?

While SvelteKit does not have any specific integration with view transitions, you can call `document.startViewTransition` in `onNavigate` to trigger a view transition on every client-side navigation.

```
import { onNavigate } from '$app/navigation';

onNavigate(navigation) => {
  if (!document.startViewTransition) return;

  return new Promise(resolve) => {
    document.startViewTransition(async () => {
      resolve();
      await navigation.complete;
    });
  });
});
```


Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores

For more, see "Unlocking view transitions" on the Svelte blog.

How do I set up 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.server.js` and import your database helpers into any endpoint that needs them.

You can use the Svelte CLI to automatically set up database integrations.

How do I use a client-side library accessing 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:

```
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:

```
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.

```
import { onMount } from 'svelte';
import { method } from 'some-browser-only-library';

onMount(() => {
  method('hello world');
});
```

Finally, you may also consider using an `{#await}` block:

```
index.svelte                                  JS TS
<script lang="ts">
  import { browser } from '$app/environment';

  const ComponentConstructor = browser ?
    import('some-browser-only-library').then((module) => module.Component) :
    new Promise(() => {});
</script>

{#await ComponentConstructor}
  <p>Loading...</p>
{:then component}
  <svelte:component this={component} />
{:catch error}
  <p>Something went wrong: {error.message}</p>
{/await}
```

How do I use a different backend API server?

You can use `event.fetch` to request data from an external API server, but be aware that you would need to deal with CORS, which will result in complications such as generally requiring requests to be preflighted resulting in higher latency. Requests to a separate subdomain may also increase latency due to an additional DNS lookup, TLS setup, etc. If you wish to use this method, you may find handleFetch helpful.

Another approach is to set up a proxy to bypass CORS headaches. In production, you would rewrite a path like `/api` to the API server; for local development, use Vite's `server.proxy` option.

How to setup rewrites in production will depend on your deployment platform. If rewrites aren't an option, you could alternatively add an API route:

```
src/routes/api/[...path]/+server.ts                    JS TS
import type { RequestHandler } from './$types';

export const GET: RequestHandler = ({ params, url }) => {
  return fetch(`https://my-api-server.com/${params.path + url.search}`);
};
```

(Note that you may also need to proxy `POST` / `PATCH` etc requests, and forward `request.headers`, depending on your needs.)

How do I use middleware?

`adapter-node` builds a middleware that you can use with your own server for production mode. In dev, you can add middleware to Vite by using a Vite plugin. For example:

```
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 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 `.yarrc.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
```

`.yarrc.yml`

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 `.yarrc.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.

Edit this page on GitHub

PREVIOUS
SEO

NEXT
Integrations

On this page
Frequently asked questions
Other resources
What can I make with SvelteKit?
How do I include details from package.json in my application?
How do I fix the error I'm getting trying to include a package?
How do I use the view transitions API?
How do I set up a database?
How do I use a client-side library accessing document or window?
How do I use a different backend API server?
How do I use middleware?
How do I use Yarn?

Glossary

The core of SvelteKit provides a highly configurable rendering engine. This section describes some of the terms used when discussing rendering. A reference for setting these options is provided in the documentation above.

CSR

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

In SvelteKit, client-side rendering will be used by default, but you can turn off JavaScript with the `csr = false` page option.

Edge

Rendering on the edge refers to rendering an application in a content delivery network (CDN) near the user. Edge rendering allows the request and response for a page to travel a shorter distance thus improving latency.

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

Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private

will then check that the DOM is in the expected state and attach event listeners in a process called hydration. Once the components are fully hydrated, they can react to changes to their properties just like any newly created Svelte component.

In SvelteKit, pages will be hydrated by default, but you can turn off JavaScript with the `csr = false` page option.

ISR

Incremental static regeneration (ISR) allows you to generate static pages on your site as visitors request those pages without redeploying. This may reduces build times compared to SSG sites with a large number of pages. You can do ISR with `adapter-vercel`.

MPA

Traditional applications that render each page view on the server — such as those written in languages other than JavaScript — are often referred to as multi-page apps (MPA).

Prerendering

Prerendering means computing the contents of a page at build time and saving the HTML for display. This approach has the same benefits as traditional server-rendered pages, but avoids recomputing the page for each visitor and so scales nearly for free as the number of visitors increases. The tradeoff is that the build process is more expensive and prerendered content can only be updated by building and deploying a new version of the application.

Not all pages can be prerendered. The basic rule is this: for content to be prerenderable, any two users hitting it directly must get the same content from the server, and the page must not contain actions. Note that you can still prerender content that is loaded based on the page's parameters as long as all users will be seeing the same prerendered content.

Pre-rendered pages are not limited to static content. You can build personalized pages if user-specific data is fetched and rendered client-side. This is subject to the caveat that you will experience the downsides of not doing SSR for that content as discussed above.

In SvelteKit, you can control prerendering with the `prerender` page option and `prerender` config in `svelte.config.js`.

PWA

A progressive web app (PWA) is an app that's built using web APIs and technologies, but functions like a mobile or desktop app. Sites served as PWAs can be installed, allowing you to add a shortcut to the application on your launcher, home screen, or start menu. Many PWAs will utilize service workers to build offline capabilities.

Routing

By default, when you navigate to a new page (by clicking on a link or using the browser's forward or back buttons), SvelteKit will intercept the attempted navigation and handle it instead of allowing the browser to send a request to the server for the destination page. SvelteKit will then update the displayed contents on the client by rendering the component for the new page, which in turn can make calls to the necessary API endpoints. This process of updating the page on the client in response to attempted navigation is called client-side routing.

In SvelteKit, client-side routing will be used by default, but you can skip it with `data-sveltekit-reload`.

SPA

A single-page app (SPA) is an application in which all requests to the server load a single HTML file which then does client-side rendering of the requested contents based on the requested URL. All navigation is handled on the client-side in a process called client-side routing with per-page contents being updated and common layout elements remaining largely unchanged. SPAs do not provide SSR and thus have worse performance and SEO characteristics. However, some applications are not greatly impacted by these shortcomings such as a complex business application behind a login where SEO would not be important and it is known that users will be accessing the application from a consistent computing environment.

In SvelteKit, you can build an SPA with `adapter-static`.

SSG

Static Site Generation (SSG) is a term that refers to a site where every page is prerendered. One benefit of fully prerendering a site is that you do not need to maintain or pay for servers to perform SSR. Once generated, the site can be served from CDNs, leading to great "time to first byte" performance. This delivery model is often referred to as JAMstack.

In SvelteKit, you can do static site generation by using `adapter-static` or by configuring every page to be prerendered using the `prerender` page option or `prerender` config in `svelte.config.js`.

SSR

Server-side rendering (SSR) is the generation of the page contents on the server. SSR is generally preferred for SEO. While some search engines can index content that is dynamically generated on the client-side it may take longer even in these cases. It also tends to improve perceived performance and makes your app accessible to users if JavaScript fails or is disabled (which happens more often than you probably think).

In SvelteKit, pages are server-side rendered by default. You can disable SSR with the `ssr` page option.

On this page
Glossary
CSR
Edge
Hydration
ISR
MPA
Prerendering
PWA
Routing
SPA
SSG
SSR

Winning adopters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT / ADVANCED

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 three hooks files, all optional:

- `src/hooks.server.js` — your app's server hooks
- `src/hooks.client.js` — your app's client hooks
- `src/hooks.js` — your app's hooks that run on both the client and server

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).

```
src/hooks.server.js
import type { Handle } from '@sveltejs/kit';

export const handle: Handle = async ({ 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)`.

During prerendering, SvelteKit crawls your pages for links and hrefs and renders each route it finds. Rendering the route invokes the `handle` function (and all other route dependencies, like `load`). If you need to exclude some code from running during this phase, check that the app is not `building` beforehand.

locals

To add custom data to the request, which is passed to handlers in `+server.js` and server `load` functions, populate the `event.locals` object, as shown below.

```
src/hooks.server.js
import type { Handle } from '@sveltejs/kit';

export const handle: Handle = async ({ event, resolve }) {
  event.locals.user = await getUserInformation(event.cookies.get('sessionid'));

  const response = await resolve(event);

  // Note that modifying response headers isn't always safe.
  // Response objects can have immutable headers
  // (e.g. Response.redirect() returned from an endpoint).
  // Modifying immutable headers throws a TypeError.
  // In that case, clone the response or avoid creating a
  // response object with immutable headers.
  response.headers.set('x-custom-header', 'potato');

  return response;
};
```

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

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

- `transformPageChunk(opts: { html: string, done: boolean }): MaybePromise<string | undefined>` — applies custom transforms to HTML. If `done` is true, it's the final chunk. Chunks are not guaranteed to be well-formed HTML (they could include an element's opening tag but not its closing tag, for example) but they will always be split at sensible boundaries such as `%sveltekit.head%` or layout/page components.
- `filterSerializedResponseHeaders(name: string, value: string): boolean` — determines which headers should be included in serialized responses when a `load` function loads a resource with `fetch`. By default, none will be included.
- `preload(input: { type: 'js' | 'css' | 'font' | 'asset' | 'data.+(server|universal).json', 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 '/style.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.

```
src/hooks.server.js
import type { Handle } from '@sveltejs/kit';

export const handle: Handle = async ({ event, resolve }) => {
  const response = await resolve(event, {
    transformPageChunk: ({ html }) => html.replace('old', 'new'),
    filterSerializedResponseHeaders: (name) => name.startsWith('x-'),
    preload: ({ type, path }) => type === 'js' || path.includes('/important/')
  });

  return response;
};
```

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

handleFetch

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

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

```
src/hooks.server.js
import type { HandleFetch } from '@sveltejs/kit';

export const handleFetch: HandleFetch = async ({ 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`:

```
src/hooks.server.js
import type { HandleFetch } from '@sveltejs/kit';
export const handleFetch: HandleFetch = async ({ 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`, `event`, `status` code and `message`. 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, which defaults to `{ message }`, becomes the value of `$page.error`.

For errors thrown from your code (or library code called by your code) the status will be 500 and the message will be "Internal Error". While `error.message` may contain sensitive information that should not be exposed to users, `message` is safe (albeit meaningless to the average user).

To add more information to the `$page.error` object in a type-safe way, you can customize the expected shape by declaring an `App.Error` interface (which must include `message: string`, to guarantee sensible fallback behavior). This allows you to — for example — append a tracking ID for users to quote in correspondence with your technical support staff:

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

export {};
```

```
src/hooks.server.js
import * as Sentry from '@sentry/sveltekit';

import type { HandleServerError } from '@sveltejs/kit';

Sentry.init({/*...*/})
export const handleError: HandleServerError = async ({ error, event, status, message })
  const errorId = crypto.randomUUID();

  // example integration with https://sentry.io/
  Sentry.captureException(error, {
    extra: { event, errorId, status }
  });

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

```
src/hooks.client.js
import * as Sentry from '@sentry/sveltekit';

import type { HandleClientError } from '@sveltejs/kit';

Sentry.init({/*...*/})
export const handleError: HandleClientError = async ({ error, event, status, message })
  const errorId = crypto.randomUUID();

  // example integration with https://sentry.io/
  Sentry.captureException(error, {
    extra: { event, errorId, status }
  });

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

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

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

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

> Make sure that `handleError` never throws an error

init

This function runs once, when the server is created or the app starts in the browser, and is a useful place to do asynchronous work such as initializing a database connection.

> If your environment supports top-level await, the `init` function is really no different from writing your initialisation logic at the top level of the module, but some environments — most notably, Safari — don't.

```
src/hooks.server.js
import * as db from '$lib/server/database';
import type { ServerInit } from '@sveltejs/kit';
```

Getting started
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node

SVELTEKIT · BEST PRACTICES

Images

Images can have a big impact on your app's performance. For best results, you should optimize them by doing the following:

- generate optimal formats like `.avif` and `.webp`
- create different sizes for different screens
- ensure that assets can be cached effectively

Doing this manually is tedious. There are a variety of techniques you can use, depending on your needs and preferences.

Vite's built-in handling

Vite will automatically process imported assets for improved performance. This includes assets referenced via the CSS `url()` function. Hashes will be added to the filenames so that they can be cached, and assets smaller than `assetsInlineLimit` will be inlined. Vite's asset handling is most often used for images, but is also useful for video, audio, etc.

```
<script>
  import logo from '$lib/assets/logo.png';
</script>

<img alt="The project logo" src={logo} />
```

@sveltejs/enhanced-img

`@sveltejs/enhanced-img` is a plugin offered on top of Vite's built-in asset handling. It provides plug and play image processing that serves smaller file formats like `.avif` or `.webp`, automatically sets the intrinsic `width` and `height` of the image to avoid layout shift, creates images of multiple sizes for various devices, and strips EXIF data for privacy. It will work in any Vite-based project including, but not limited to, SvelteKit projects.

> As a build plugin, `@sveltejs/enhanced-img` can only optimize files located on your machine during the build process. If you have an image located elsewhere (such as a path served from your database, CMS, or backend), please read about loading images dynamically from a CDN.
>
> WARNING: The `@sveltejs/enhanced-img` package is experimental. It uses pre-1.0 versioning and may introduce breaking changes with every minor version release.

Setup

Install:

```
npm install --save-dev @sveltejs/enhanced-img
```

Adjust `vite.config.js`:

```
import { sveltekit } from '@sveltejs/kit/vite';
import { enhancedImages } from '@sveltejs/enhanced-img';
import { defineConfig } from 'vite';

export default defineConfig({
  plugins: [
    enhancedImages(),
    sveltekit()
  ]
});
```

Building will take longer on the first build due to the computational expense of transforming images. However, the build output will be cached in `./node_modules/.cache/imagetools` so that subsequent builds will be fast.

Basic usage

Use in your `.svelte` components by using `enhanced:img` rather than `<img>` and referencing the image file with a Vite asset import path:

```
<enhanced:img src="./path/to/your/image.jpg" alt="An alt text" />
```

At build time, your `enhanced:img` tag will be replaced with an `<img>` wrapped by a `<picture>` providing multiple image types and sizes. It's only possible to downscale images without losing quality, which means that you should provide the highest resolution image that you need — smaller versions will be generated for the various device types that may request an image.

You should provide your image at 2x resolution for HiDPI displays (a.k.a. retina displays). `enhanced:img` will automatically take care of serving smaller versions to smaller devices.

If you wish to add styles to your `enhanced:img`, you should add a `class` and target that.

Dynamically choosing an image

You can also manually import an image asset and pass it to an `enhanced:img`. This is useful when you have a collection of static images and would like to dynamically choose one or iterate over them. In this case you will need to update both the `import` statement and `<img>` element as shown below to indicate you'd like process them.

```
<script>
  import MyImage from './path/to/your/image.jpg?enhanced';
</script>

<enhanced:img src={MyImage} alt="some alt text" />
```

You can also use Vite's, `import.meta.glob`. Note that you will have to specify `enhanced` via a custom query:

```
<script>
  const imageModules = import.meta.glob(
    '/path/to/assets/*.{avif,gif,heif,jpeg,jpg,png,tiff,webp,svg}',
    {
      eager: true,
      query: {
        enhanced: true
      }
    }
  )
</script>

{#each Object.entries(imageModules) as [_path, module]}
  <enhanced:img src={module.default} alt="some alt text" />
{/each}
```

Intrinsic Dimensions

`width` and `height` are optional as they can be inferred from the source image and will be automatically added when the `enhanced:img` tag is preprocessed. With these attributes, the browser can reserve the correct amount of space, preventing layout shift. If you'd like to use a different `width` and `height` you can style the image with CSS. Because the preprocessor adds a `width` and `height` for you, if you'd like one of the dimensions to be automatically calculated then you will need to specify that:

```
<style>
  .hero-image img {
    width: var(--size);
    height: auto;
  }
</style>
```

srcset and sizes

If you have a large image, such as a hero image taking the width of the design, you should specify `sizes` so that smaller versions are requested on smaller devices. E.g. if you have a 1280px image you may want to specify something like:

```
<enhanced:img src="./image.png" sizes="min(1280px, 100vw)"/>
```

If `sizes` is specified, `enhanced:img` will generate small images for smaller devices and populate the `srcset` attribute.

The smallest picture generated automatically will have a width of 540px. If you'd like smaller images or would otherwise like to specify custom widths, you can do that with the `w` query parameter:

```
<enhanced:img
  src="./image.png?w=1280;640;480"
  sizes="(min-width:1920px) 1280px, (min-width:1080px) 640px, (min-width:768px) 480px"
/>
```

If `sizes` is not provided, then a HiDPI/Retina image and a standard resolution image will be generated. The image you provide should be 2x the resolution you wish to display so that the browser can display that image on devices with a high device pixel ratio.

Per-image transforms

By default, enhanced images will be transformed to more efficient formats. However, you may wish to apply other transforms such as a blur, quality, flatten, or rotate operation. You can run per-image transforms by appending a query string:

```
<enhanced:img src="./path/to/your/image.jpg?blur=15" alt="An alt text" />
```

See the imagetools repo for the full list of directives.

Loading images dynamically from a CDN

In some cases, the images may not be accessible at build time — e.g. they may live inside a content management system or elsewhere.

Using a content delivery network (CDN) can allow you to optimize these images dynamically, and provides more flexibility with regards to sizes, but it may involve some setup overhead and usage costs. Depending on caching strategy, the browser may not be able to use a cached copy of the asset until a 304 response is received from the CDN. Building HTML to target CDNs allows using an `<img>` tag since the CDN can serve the appropriate format based on the `User-Agent` header, whereas build-time optimizations must produce `<picture>` tags with multiple sources. Finally, some CDNs may generate images lazily, which could have a negative performance impact for sites with low traffic and frequently changing images.

CDNs can generally be used without any need for a library. However, there are a number of libraries with Svelte support that make it easier. `@unpic/svelte` is a CDN-agnostic library with support for a large number of providers. You may also find that specific CDNs like Cloudinary have Svelte support. Finally, some content management systems (CMS) which support Svelte (such as Contentful, Storyblok, and Contentstack) have built-in support for image handling.

Icons

A great way to use icons is to define them purely in CSS. Iconify offers a huge set of icons available via CSS.

For icons defined in `.svelte` files, it is recommended to avoid libraries that provide a `.svelte` file per icon. These libraries can have thousands of `.svelte` files which really slow down Vite's dependency optimization. This can become especially pathological if the icons are imported both via an umbrella import and subpath import as described in the vite-plugin-svelte FAQ.

Best practices

- For each image type, use the appropriate solution from those discussed above. You can mix and match all three solutions in one project. For example, you may use Vite's built-in handling to provide images for `<meta>` tags, display images on your homepage with `@sveltejs/enhanced-img`, and display user-submitted content with a dynamic approach.
- Consider serving all images via CDN regardless of the image optimization types you use. CDNs reduce latency by distributing copies of static assets globally.
- Your original images should have a good quality/resolution and should have 2x the width it will be displayed at to serve HiDPI devices. Image processing can size images down to save bandwidth when serving smaller screens, but it would be a waste of bandwidth to invent pixels to size images up.
- For images which are much larger than the width of a mobile device (roughly 400px), such as a hero image taking the width of the page design, specify `sizes` so that smaller images can be served on smaller devices.
- For important images, such as the largest contentful paint (LCP) image, set `fetchpriority="high"` and avoid `loading="lazy"` to prioritize loading as early as possible.
- Give the image a container or styling so that it is constrained and does not jump around while the page is loading affecting your cumulative layout shift (CLS). `width` and `height` help the browser to reserve space while the image is still loading, so `@sveltejs/enhanced-img` will add a `width` and `height` for you.
- Always provide a good `alt` text. The Svelte compiler will warn you if you don't do this.
- Do not use `em` or `rem` in `sizes` and change the default size of these measures. When used in `sizes` or `@media` queries, `em` and `rem` are both defined to mean the user's default `font-size`. For a `sizes` declaration like `sizes="(min-width: 768px) 50vw, min(100vw, 1800px)"`, the actual `em` or `rem` that controls how the image is laid out on the page can be different if changed by CSS. For example, do not do something like `html { font-size: 62.5%; }` as the slot reserved by the browser preloader will now end up being larger than the actual slot of the CSS object model once it has been created.

Edit this page on GitHub

PREVIOUS
Performance

NEXT
Accessibility

On this page
Images
Vite's built-in handling
@sveltejs/enhanced-img
Loading images dynamically from a CDN
Icons
Best practices

Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private

SVELTEKIT • APPENDIX

Integrations

On this page
Integrations
vitePreprocess
Adders
Directory
Additional integrations
Vite plugins
Integration FAQs

vitePreprocess

Including `vitePreprocess` in your project will allow you to use the various flavors of CSS that Vite supports: PostCSS, SCSS, Less, Stylus, and SugarSS. If you set your project up with TypeScript it will be included by default:

```
// svelte.config.js
import { vitePreprocess } from '@sveltejs/vite-plugin-svelte';

export default {
  preprocess: [vitePreprocess()]
};
```

You will also need to use a preprocessor if you're using TypeScript with Svelte 4. TypeScript is supported natively in Svelte 5 if you're using only the type syntax. To use more complex TypeScript syntax in Svelte 5, you will need still need a preprocessor and can use `vitePreprocess({ script: true })`.

Adders

Run `npx sv add` to setup many different complex integrations with a single command including:

- prettier (formatting)
- eslint (linting)
- vitest (unit testing)
- playwright (e2e testing)
- lucia (auth)
- tailwind (CSS)
- drizzle (DB)
- paraglide (i18n)
- mdsvex (markdown)
- storybook (frontend workshop)

Directory

See sveltesociety.dev for a full listing of packages and templates available for use with Svelte and SvelteKit.

Additional integrations

svelte-preprocess

`svelte-preprocess` has some additional functionality not found in `vitePreprocess` such as support for Pug, Babel, and global styles. However, `vitePreprocess` may be faster and require less configuration, so it is used by default. Note that CoffeeScript is not supported by SvelteKit.

You will need to install `svelte-preprocess` with `npm install --save-dev svelte-preprocess` and add it to your `svelte.config.js`. After that, you will often need to install the corresponding library such as `npm install -D sass` or `npm install -D less`.

Vite plugins

Since SvelteKit projects are built with Vite, you can use Vite plugins to enhance your project. See a list of available plugins at `vitejs/awesome-vite`.

Integration FAQs

The SvelteKit FAQ answers many questions about how to do X with SvelteKit, which may be helpful if you still have questions.

Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT · ADVANCED

Link options

In SvelteKit, `<a>` elements (rather than framework-specific `<Link>` components) are used to navigate between the routes of your app. If the user clicks on a link whose `href` is 'owned' by the app (as opposed to, say, a link to an external site) then SvelteKit will navigate to the new page by importing its code and then calling any `load` functions it needs to fetch data.

You can customise the behaviour of links with `data-sveltekit-*` attributes. These can be applied to the `<a>` itself, or to a parent element.

These options also apply to `<form>` elements with `method="GET"`.

data-sveltekit-preload-data

Before the browser registers that the user has clicked on a link, we can detect that they've hovered the mouse over it (on desktop) or that a `touchstart` or `mousedown` event was triggered. In both cases, we can make an educated guess that a `click` event is coming.

SvelteKit can use this information to get a head start on importing the code and fetching the page's data, which can give us an extra couple of hundred milliseconds — the difference between a user interface that feels laggy and one that feels snappy.

We can control this behaviour with the `data-sveltekit-preload-data` attribute, which can have one of two values:

- `"hover"` means that preloading will start if the mouse comes to a rest over a link. On mobile, preloading begins on `touchstart`
- `"tap"` means that preloading will start as soon as a `touchstart` or `mousedown` event is registered

The default project template has a `data-sveltekit-preload-data="hover"` attribute applied to the `<body>` element in `src/app.html`, meaning that every link is preloaded on hover by default:

```
<body data-sveltekit-preload-data="hover">
  <div style="display: contents">%sveltekit.body%</div>
</body>
```

Sometimes, calling `load` when the user hovers over a link might be undesirable, either because it's likely to result in false positives (a click needn't follow a hover) or because data is updating very quickly and a delay could mean staleness.

In these cases, you can specify the `"tap"` value, which causes SvelteKit to call `load` only when the user taps or clicks on a link:

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

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

Data will never be preloaded if the user has chosen reduced data usage, meaning `navigator.connection.saveData` is `true`.

data-sveltekit-preload-code

Even in cases where you don't want to preload *data* for a link, it can be beneficial to preload the *code*. The `data-sveltekit-preload-code` attribute works similarly to `data-sveltekit-preload-data`, except that it can take one of four values, in decreasing 'eagerness':

- `"eager"` means that links will be preloaded straight away
- `"viewport"` means that links will be preloaded once they enter the viewport
- `"hover"` - as above, except that only code is preloaded
- `"tap"` - as above, except that only code is preloaded

Note that `viewport` and `eager` only apply to links that are present in the DOM immediately following navigation — if a link is added later (in an `{#if ...}` block, for example) it will not be preloaded until triggered by `hover` or `tap`. This is to avoid performance pitfalls resulting from aggressively observing the DOM for changes.

Since preloading code is a prerequisite for preloading data, this attribute will only have an effect if it specifies a more eager value than any `data-sveltekit-preload-data` attribute that is present.

As with `data-sveltekit-preload-data`, this attribute will be ignored if the user has chosen reduced data usage.

data-sveltekit-reload

Occasionally, we need to tell SvelteKit not to handle a link, but allow the browser to handle it. Adding a `data-sveltekit-reload` attribute to a link...

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

...will cause a full-page navigation when the link is clicked.

Links with a `rel="external"` attribute will receive the same treatment. In addition, they will be ignored during prerendering.

data-sveltekit-replacestate

Sometimes you don't want navigation to create a new entry in the browser's session history. Adding a `data-sveltekit-replacestate` attribute to a link...

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

...will replace the current `history` entry rather than creating a new one with `pushState` when the link is clicked.

data-sveltekit-keepfocus

Sometimes you don't want focus to be reset after navigation. For example, maybe you have a search form that submits as the user is typing, and you want to keep focus on the text input. Adding a `data-sveltekit-keepfocus` attribute to it...

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

...will cause the currently focused element to retain focus after navigation. In general, avoid using this attribute on links, since the focused element would be the `<a>` tag (and not a previously focused element) and screen reader and other assistive technology users often expect focus to be moved after a navigation. You should also only use this attribute on elements that still exist after navigation. If the element no longer exists, the user's focus will be lost, making for a confusing experience for assistive technology users.

data-sveltekit-noscroll

When navigating to internal links, SvelteKit mirrors the browser's default navigation behaviour: it will change the scroll position to 0,0 so that the user is at the very top left of the page (unless the link includes a `#hash`, in which case it will scroll to the element with a matching ID).

In certain cases, you may wish to disable this behaviour. Adding a `data-sveltekit-noscroll` attribute to a link...

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

...will prevent scrolling after the link is clicked.

Disabling options

To disable any of these options inside an element where they have been enabled, use the `"false"` value:

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

  <div data-sveltekit-preload-data="false">
    <!-- these links will NOT be preloaded -->
    <a href="/d">d</a>
    <a href="/e">e</a>
    <a href="/f">f</a>
  </div>
</div>
```

To apply an attribute to an element conditionally, do this:

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

Edit this page on GitHub

PREVIOUS
Errors

NEXT
Service workers

On this page
Link options
data-sveltekit-preload-data
data-sveltekit-preload-code
data-sveltekit-reload
data-sveltekit-replacestate
data-sveltekit-keepfocus
data-sveltekit-noscroll
Disabling options

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT / CORE CONCEPTS

Loading data

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

Page data

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

src/routes/blog/[slug]/+page.ts
```
import type { PageLoad } from './$types';

export const load: PageLoad = ({ params }) => {
  return {
    post: {
      title: `Title for ${params.slug} goes here`,
      content: `Content for ${params.slug} goes here`
    }
  };
};
```

src/routes/blog/[slug]/+page.svelte
```
<script lang="ts">
  import type { PageProps } from './$types';

  let { data }: PageProps = $props();
</script>

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

Legacy mode
hide all

Before version 2.16.0, the props of a page and layout had to be typed individually:

+page.svelte
```
import type { PageData } from './$types';

let { data }: { data: PageData } = $props();
```

In Svelte 4, you'd use `export let data` instead.

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

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

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

src/routes/blog/[slug]/+page.server.ts
```
import * as db from '$lib/server/database';
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async ({ params }) => {
  return {
    post: await db.getPost(params.slug)
  };
};
```

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

Layout data

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

src/routes/blog/[slug]/+layout.server.ts
```
import * as db from '$lib/server/database';
import type { LayoutServerLoad } from './$types';

export const load: LayoutServerLoad = async () => {
  return {
    posts: await db.getPostSummaries()
  };
};
```

src/routes/blog/[slug]/+layout.svelte
```
<script lang="ts">
  import type { LayoutProps } from '.$types';

  let { data, children }: LayoutProps = $props();
</script>

<main>
  <!-- +page.svelte is $render'ed here -->
  {@render children()}
</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>
```

Legacy mode
hide all

`LayoutProps` was added in 2.16.0. In earlier versions, properties had to be typed individually:

+layout.svelte
```
import type { LayoutData } from './$types';

let { data, children }: { data: LayoutData, children: Snippet } = $props();
```

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.

src/routes/blog/[slug]/+page.svelte
```
<script lang="ts">
  import { page } from $app/state';
  import type { PageProps } from './$types';

  let { data }: PageProps = $props();

  // we can access `data.posts` because it's returned from
  // the parent layout `load` function
  let index = $derived(data.posts.findIndex(post => post.slug === page.params.slug));
  let next = $derived(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`:

src/routes/+layout.svelte
```
<script lang="ts">
  import { page } from $app/state';
</script>

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

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

Legacy mode
hide all

`$app/state` was added in SvelteKit 2.12. If you're using an earlier version or are using Svelte 4, use `$app/stores` instead. It provides a `page` store with the same interface that you can subscribe to, e.g. `$page.data.title`.

Universal vs server

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

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

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

When does which load function run?

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

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

If a route contains both universal and server `load` functions, the server `load` runs first.

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

Input

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

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

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

Output

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

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

When to use which

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

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

In rare cases, you might need to use both together — for example, you might need to return an instance of a custom class that was initialised with data from your server. When using both, the server `load` return value is *not* passed directly to the page, but to the universal `load` function as the `data` property:

src/routes/+page.server.ts
```
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async () => {
  return {
    serverMessage: 'hello from server load function'
  };
};
```

src/routes/+page.ts
```
import type { PageLoad } from './$types';
export const load: PageLoad = async ({ data }) => {
  return {
    serverMessage: data.serverMessage,
    universalMessage: 'hello from universal load function'
  };
};
```

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.

SVELTEKIT / APPENDIX

Migrating to SvelteKit v2

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

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

redirect and error are no longer thrown by you

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

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

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

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

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

Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
$sveltejs/kit
$sveltejs/kit/hooks
$sveltejs/kit/node/polyfills
$sveltejs/kit/node
$sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/stores
$app/types/private
$app/types/public
$env/static/private
$env/static/public
$lib
$service-worker
$types
cli
configuration
errors
hooks
modules
packaging
prerendering
routing
runtime
server-side rendering
service-worker
state management
types
web standards

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

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

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

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

Top level promises are no longer awaited

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

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

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

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

goto(...) changes

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

paths are now relative by default

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

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

Server fetches are not trackable anymore

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

preloadCode arguments must be prefixed with base

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

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

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

resolvePath has been removed

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

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

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

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

`svelte-migrate` will do the method replacement for you, though if you later prepend the result with `base`, you need to remove that yourself.

Improved error handling

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

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

Dynamic environment variables cannot be used during prerendering

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

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

Because of this, dynamic environment variables can no longer be read during prerendering in SvelteKit 2 — you should use the `$static` modules instead. If the user lands on a prerendered page, SvelteKit will request up-to-date values for `$env/dynamic/public` from the server (by default from a module called `_app/env.js`) instead of reading them from the server-rendered HTML.

form and data have been removed form use:enhance callbacks

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

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

Forms containing file inputs must use multipart/form-data

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

Generated tsconfig.json is more strict

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

getRequest no longer throws errors

The `$sveltejs/kit/node` module exports helper functions for use in Node environments, including `getRequest` which turns a Node `IncomingRequest` into a standard `Request` object.

In SvelteKit 1, `getRequest` could throw if the `Content-Length` header exceeded the specified size limit. In SvelteKit 2, the error will not be thrown until later, when the request body (if any) is being read. This enables better diagnostics and simpler code.

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

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

Updated dependency requirements

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

- `svelte@4`
- `vite@5`
- `typescript@5`
- `@sveltejs/vite-plugin-svelte@3` (this is now required as a `peerDependency` of SvelteKit — previously it was directly depended upon)
- `@sveltejs/adapter-cloudflare@1` (if you're using these adapters)
- `@sveltejs/adapter-cloudflare-workers@2`
- `@sveltejs/adapter-netlify@3`
- `@sveltejs/adapter-node@2`
- `@sveltejs/adapter-static@3`
- `@sveltejs/adapter-vercel@4`

`svelte-migrate` will update your `package.json` for you.

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

SvelteKit 2.12: $app stores deprecated

On this page

Migrating to SvelteKit v2
redirect and error are no longer thrown by you
path is required when setting cookies
Top-level promises are no longer awaited
goto(...) changes
paths are now relative by default
Server fetches are not trackable anymore
preloadCode arguments must be prefixed with base
resolvePath has been removed
Improved error handling
Dynamic environment variables cannot be used during prerendering
form and data have been removed from use:enhance callbacks
Forms containing file inputs must use multipart/form-data
Generated tsconfig.json is more strict
getRequest no longer throws errors
vitePreprocess is no longer exported from $sveltejs/kit/vite
Updated dependency requirements
SvelteKit 2.12: $app stores deprecated

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Netlify

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

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

Usage

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

```
svelte.config.js
import adapter from '@sveltejs/adapter-netlify';

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

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

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

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

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

Node version

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

Netlify Edge Functions

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

```
svelte.config.js
import adapter from '@sveltejs/adapter-netlify';

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

Netlify alternatives to SvelteKit functionality

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

_headers and _redirects

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

Redirect rules

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

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

Netlify Forms

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

Netlify Functions

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

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

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

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

[functions]
  directory = "functions"
```

Troubleshooting

Accessing the file system

You can't use `fs` in edge deployments.

You *can* use it in serverless deployments, but it won't work as expected, since files are not copied from your project into your deployment. Instead, use the `read` function from `$app/server` to access your files. `read` does not work inside edge deployments (this may change in future).

Alternatively, you can prerender the routes in question.

On this page
Netlify
Usage
Netlify Edge Functions
Netlify alternatives to SvelteKit functionality
Troubleshooting

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT › BUILD AND DEPLOY

Node servers

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

Usage

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

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

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

Deploying

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

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

```
node build
```

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

Compressing responses

You will typically want to compress responses coming from the server. If you are already deploying your server behind a reverse proxy for SSL or load balancing, it typically results in better performance to also handle compression at that layer since Node.js is single-threaded.

However, if you're building a custom server and do want to add a compression middleware there, note that we would recommend using @polka/compression since SvelteKit streams responses and the more popular `compression` package does not support streaming and may cause errors when used.

Environment variables

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

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

```
npm install dotenv
```

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

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

If you use Node.js v20.6+, you can use the `env-file` flag instead:

```
node --env-file=.env build
```

PORT, HOST and SOCKET_PATH

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

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

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

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

ORIGIN, PROTOCOL_HEADER, HOST_HEADER, and PORT_HEADER

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

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

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

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

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

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

If you're hosting your proxy on a non-standard port and your reverse proxy supports `X-Forwarded-Port`, you can also set `PORT_HEADER=x-forwarded-port`.

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

Cross-site POST form submissions are forbidden

ADDRESS_HEADER and XFF_DEPTH

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

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

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

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

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

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

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

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

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

BODY_SIZE_LIMIT

The maximum request body size to accept in bytes including while streaming. The body size can also be specified with a unit suffix for kilobytes (`K`), megabytes (`M`), or gigabytes (`G`). For example, `512K` or `1M`. Defaults to 512kb. You can disable this option with a value of `Infinity` (0 in older versions of the adapter) and implement a custom check in `handle` if you need something more advanced.

SHUTDOWN_TIMEOUT

The number of seconds to wait before forcefully closing any remaining connections after receiving a `SIGTERM` or `SIGINT` signal. Defaults to `30`. Internally the adapter calls `closeAllConnections`. See Graceful shutdown for more details.

IDLE_TIMEOUT

When using systemd socket activation, `IDLE_TIMEOUT` specifies the number of seconds after which the app is automatically put to sleep when receiving no requests. If not set, the app runs continuously. See Socket activation for more details.

Options

The adapter can be configured with various options:

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

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

out

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

precompress

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

envPrefix

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

```
envPrefix: 'MY_CUSTOM_'
```

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

Graceful shutdown

By default `adapter-node` gracefully shuts down the HTTP server when a `SIGTERM` or `SIGINT` signal is received. It will:

1. reject new requests (`server.close()`)
2. wait for requests that have already been made but not received a response yet to finish and close connections once they become idle (`server.closeIdleConnections`)
3. and finally, close any remaining connections that are still active after `SHUTDOWN_TIMEOUT` seconds. (`server.closeAllConnections`)

If you want to customise this behaviour you can use a custom server.

You can listen to the `sveltekit:shutdown` event which is emitted after the HTTP server has closed all connections. Unlike Node's `exit` event, the `sveltekit:shutdown` event supports asynchronous operations and is always emitted when all connections are closed even if the server has dangling work such as open database connections.

```
process.on('sveltekit:shutdown', async (reason) => {
  await jobs.stop();
  await db.close();
});
```

The parameter `reason` has one of the following values:

- `SIGINT` - shutdown was triggered by a `SIGINT` signal
- `SIGTERM` - shutdown was triggered by a `SIGTERM` signal
- `IDLE` - shutdown was triggered by `IDLE_TIMEOUT`

Socket activation

Most Linux operating systems today use a modern process manager called systemd to start the server and run and manage services. You can configure your server to allocate a socket and start and scale your app on demand. This is called socket activation. In this case, the OS will pass two environment variables to your app — `LISTEN_PID` and `LISTEN_FDS`. The adapter will then listen on file descriptor 3 which refers to a systemd socket unit that you will have to create.

You can still use `envPrefix` with systemd socket activation. `LISTEN_PID` and `LISTEN_FDS` are always read without a prefix.

To take advantage of socket activation follow these steps.

1. Run your app as a systemd service. It can either run directly on the host system or

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT · ADVANCED

Packaging

You can use SvelteKit to build apps as well as component libraries, using the `@sveltejs/package` package (`npx sv create` has an option to set this up for you).

When you're creating an app, the contents of `src/routes` is the public-facing stuff; `src/lib` contains your app's internal library.

A component library has the exact same structure as a SvelteKit app, except that `src/lib` is the public-facing bit, and your root `package.json` is used to publish the package. `src/routes` might be a documentation or demo site that accompanies the library, or it might just be a sandbox you use during development.

Running the `svelte-package` command from `@sveltejs/package` will take the contents of `src/lib` and generate a `dist` directory (which can be configured) containing the following:

- All of the files in `src/lib`. Svelte components will be preprocessed, TypeScript files will be transpiled to JavaScript.
- Type definitions (`d.ts` files) which are generated for Svelte, JavaScript and TypeScript files. You need to install `typescript >= 4.0.0` for this. Type definitions are placed next to their implementation, hand-written `d.ts` files are copied over as is. You can disable generation, but we strongly recommend against it — people using your library might use TypeScript, for which they require these type definition files.

> `@sveltejs/package` version 1 generated a `package.json`. This is no longer the case and it will now use the `package.json` from your project and validate that it is correct instead. If you're still on version 1, see this PR for migration instructions.

Anatomy of a package.json

Since you're now building a library for public use, the contents of your `package.json` will become more important. Through it, you configure the entry points of your package, which files are published to npm, and which dependencies your library has. Let's go through the most important fields one by one.

name

This is the name of your package. It will be available for others to install using that name, and visible on `https://npmjs.com/package/<name>`.

```
{
  "name": "your-library"
}
```

Read more about it here.

license

Every package should have a license field so people know how they are allowed to use it. A very popular license which is also very permissive in terms of distribution and reuse without warranty is `MIT`.

```
{
  "license": "MIT"
}
```

Read more about it here. Note that you should also include a `LICENSE` file in your package.

files

This tells npm which files it will pack up and upload to npm. It should contain your output folder (`dist` by default). Your `package.json` and `README` and `LICENSE` will always be included, so you don't need to specify them.

```
{
  "files": ["dist"]
}
```

To exclude unnecessary files (such as unit tests, or modules that are only imported from `src/routes` etc) you can add them to an `.npmignore` file. This will result in smaller packages that are faster to install.

Read more about it here.

exports

The `"exports"` field contains the package's entry points. If you set up a new library project through `npx sv create`, it's set to a single export, the package root:

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

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

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

The `types` and `svelte` keys are export conditions. They tell tooling what file to import when they look up the `your-library` import:

- TypeScript sees the `types` condition and looks up the type definition file. If you don't publish type definitions, omit this condition.
- Svelte-aware tooling sees the `svelte` condition and knows this is a Svelte component library. If you publish a library that does not export any Svelte components and that could also work in non-Svelte projects (for example a Svelte store library), you can replace this condition with `default`.

> Previous versions of `@sveltejs/package` also added a `package.json` export. This is no longer part of the template because all tooling can now deal with a `package.json` not being explicitly exported.

You can adjust `exports` to your liking and provide more entry points. For example, if instead of a `src/lib/index.ts` file that re-exported components you wanted to expose a `src/lib/Foo.svelte` component directly, you could create the following export map:

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

...and a consumer of your library could import the component like so:

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

> Beware that doing this will need additional care if you provide type definitions. Read more about the caveat here.

In general, each key of the exports map is the path the user will have to use to import something from your package, and the value is the path to the file that will be imported or a map of export conditions which in turn contains these file paths.

Read more about `exports` here.

svelte

This is a legacy field that enabled tooling to recognise Svelte component libraries. It's no longer necessary when using the `svelte` export condition, but for backwards compatibility with outdated tooling that doesn't yet know about export conditions it's good to keep it around. It should point towards your root entry point.

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

sideEffects

The `sideEffects` field in `package.json` is used by bundlers to determine if a module may contain code that has side effects. A module is considered to have side effects if it makes changes that are observable from other scripts outside the module when it's imported. For example, side effects include modifying global variables or the prototype of built-in JavaScript objects. Because a side effect could potentially affect the behavior of other parts of the application, these files/modules will be included in the final bundle regardless of whether their exports are used in the application. It is a best practice to avoid side effects in your code.

Setting the `sideEffects` field in `package.json` can help the bundler to be more aggressive in eliminating unused exports from the final bundle, a process known as tree-shaking. This results in smaller and more efficient bundles. Different bundlers handle `sideEffects` in various manners. While not necessary for Vite, we recommend that libraries state that all CSS files have side effects so that your library will be compatible with webpack. This is the configuration that comes with newly created projects:

package.json

```
{
  "sideEffects": ["*.*/*..*.*.*
```

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT · CORE CONCEPTS

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 initialise 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.

On this page
Page options
prerender
entries
ssr
csr
trailingSlash
config
Further reading

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*.

```
+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:

```
+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'`:

```
+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 `id` elements that point to other pages that are candidates for prerendering — because of this, you generally don't need to specify which pages should be accessed. If you *do* need to specify which pages should be accessed by the prerenderer, you can do so with `config.kit.prerender.entries`, or by exporting an `entries` function from your dynamic route.

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

Prerendering server routes

Unlike the other page options, `prerender` also applies to `+server.js` files. These files are *not* affected by 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...

```
+page.js                                                    TS
import type { PageLoad } from './$types';
export const prerender = true;
export const load: PageLoad = async ({ 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 prerendered, 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`.

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 reached by the prerendering crawler and thus wasn't prerendered.

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

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

entries

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

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

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

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

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

```
src/routes/blog/[slug]/+page.server.js                      TS
import type {EntryGenerator} from './$types';

export const entries: EntryGenerator = () => {
  return [
    { slug: 'hello-world' },
    { slug: 'another-blog-post' }
  ];
};

export const prerender = true;
```

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

ssr

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

```
+page.js
export const ssr = false;
// If both `ssr` and `csr` are `false`, nothing will be rendered!
```

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.

Even with `ssr` set to `false`, code that relies on browser APIs should be imported in your `+page.svelte` or `+layout.svelte` file instead. This is because page options can be overridden and need to be evaluated by importing your `+page.js` or `+layout.js` file on the server (if you have a runtime) or at build time (in case of prerendering).

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:

```
+page.js
export const csr = false;
// If both `ssr` and `csr` are `false`, nothing will be rendered!
```

Disabling CSR, does not ship any JavaScript to the client. This means:

- The webpage should work with HTML and CSS only.
- `<script>` tags inside all Svelte components are removed.
- `<form>` elements cannot be progressively enhanced.
- Links are handled by the browser with a full page navigation.
- Hot Module Replacement (HMR) will be disabled.

You can enable `csr` during development (for example to take advantage of HMR) like so:

```
+page.js
import { dev } from '$app/environment';

export const csr = dev;
```

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.

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

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

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

config

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

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

```
src/routes/+page.js                                         TS
import type { Config } from 'some-adapter';

export const config: Config = {
```

Further reading

Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node

SVELTEKIT • BEST PRACTICES

Performance

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

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

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

Diagnosing issues

Google's PageSpeed Insights and (for more advanced analysis) WebPageTest are excellent ways to understand the performance characteristics of a site that is already deployed to the internet.

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

- Chrome - Lighthouse, Network, and Performance devtools
- Edge - Lighthouse, Network, and Performance devtools
- Firefox - Network and Performance devtools
- Safari - enhancing the performance of your webpage

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

Instrumenting

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

Optimizing assets

Images

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

Videos

Video files can be very large, so extra care should be taken to ensure that they're optimized:

- Compress videos with tools such as Handbrake. Consider converting the videos to web-friendly formats such as `.webm` or `.mp4`.
- You can lazy-load videos located below the fold with `preload="none"` (though note that this will slow down playback when the user *does* initiate it).
- Strip the audio track out of muted videos using a tool like FFmpeg.

Fonts

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

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

Reducing code size

Svelte version

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

Packages

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

External scripts

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

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

Selective loading

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

Navigation

Preloading

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

Non-essential data

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

Preventing waterfalls

One of the biggest performance killers is what is referred to as a *waterfall*, which is a series of requests that is made sequentially. This can happen on the server or in the browser.

- Asset waterfalls can occur in the browser when your HTML requests JS which requests CSS which requests a background image and web font. SvelteKit will largely solve this class of problems for you by adding `modulepreload` tags or headers, but you should view the network tab in your devtools to check whether additional resources need to be preloaded. Pay special attention to this if you use web fonts since they need to be handled manually.
- If a universal `load` function makes an API call to fetch the current user, then uses the details from that response to fetch a list of saved items, and then uses *that* response to fetch the details for each item, the browser will end up making multiple sequential requests. This is deadly for performance, especially for users that are physically located far from your backend. Avoid this issue by using server `load` functions where possible.
- Server `load` functions are also not immune to waterfalls (though they are much less costly since they rarely involve roundtrips with high latency). For example if you query a database to get the current user and then use that data to make a second query for a list of saved items, it will typically be more performant to issue a single query with a database join.

Hosting

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

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

Further reading

For the most part, building a performant SvelteKit app is the same as building any performant web app. You should be able to apply information from general performance resources such as Core Web Vitals to any web experience you build.

Edit this page on GitHub

PREVIOUS
Auth

NEXT
Images

On this page
Performance
Diagnosing issues
Optimizing assets
Reducing code size
Navigation
Hosting
Further reading

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT • GETTING STARTED

Project structure

On this page
Project structure
Project files
Other files

A typical SvelteKit project looks like this:

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

You'll also find common files like `.gitignore` and `.npmrc` (and `.prettierrc` and `eslint.config.js` and so on, if you chose those options when running `npx sv create`).

Project files

src

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

- `lib` contains your library code (utilities and components), which can be imported via the `$lib` alias, or packaged up for distribution using `svelte-package`
  - `server` contains your server-only library code. It can be imported by using the `$lib/server` alias. SvelteKit will prevent you from importing these in client code.
- `params` contains any param matchers your app needs
- `routes` contains the routes of your application. You can also colocate other components that are only used within a single route here
- `app.html` is your page template — an HTML document containing the following placeholders:
  - `%sveltekit.head%` — `<link>` and `<script>` elements needed by the app, plus any `<svelte:head>` content
  - `%sveltekit.body%` — the markup for a rendered page. This should live inside a `<div>` or other element, rather than directly inside `<body>`, to prevent bugs caused by browser extensions injecting elements that are then destroyed by the hydration process. SvelteKit will warn you in development if this is not the case
  - `%sveltekit.assets%` — either `paths.assets`, if specified, or a relative path to `paths.base`
  - `%sveltekit.nonce%` — a CSP nonce for manually included links and scripts, if used
  - `%sveltekit.env.[NAME]%` - this will be replaced at render time with the `[NAME]` environment variable, which must begin with the `publicPrefix` (usually `PUBLIC_`). It will fallback to `''` if not matched.
- `error.html` is the page that is rendered when everything else fails. It can contain the following placeholders:
  - `%sveltekit.status%` — the HTTP status
  - `%sveltekit.error.message%` — the error message
- `hooks.client.js` contains your client hooks
- `hooks.server.js` contains your server hooks
- `service-worker.js` contains your service worker

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

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

static

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

tests

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

package.json

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

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

svelte.config.js

This file contains your Svelte and SvelteKit configuration.

tsconfig.json

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

vite.config.js

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

Other files

.svelte-kit

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

Edit this page on GitHub

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT · GETTING STARTED

Project types

On this page
Project types
Default rendering
Static site generation
Single-page app
Multi-page app
Separate backend
Serverless app
Your own server
Container
Library
Offline app
Mobile app
Desktop app
Browser extension
Embedded device

SvelteKit offers configurable rendering, which allows you to build and deploy your project in several different ways. You can build all of the below types of applications and more with SvelteKit. Rendering settings are not mutually exclusive and you may choose the optimal manner with which to render different parts of your application.

If you don't have a particular way you'd like to build your application in mind, don't worry! The way your application is built, deployed, and rendered is controlled by which adapter you've chosen and a small amount of configuration and these can always be changed later. The project structure and routing will be the same regardless of the project type that you choose.

Default rendering

By default, when a user visits a site, SvelteKit will render the first page with server-side rendering (SSR) and subsequent pages with client-side rendering (CSR). Using SSR for the initial render improves SEO and perceived performance of the initial page load. Client-side rendering then takes over and updates the page without having to rerender common components, which is typically faster and eliminates a flash when navigating between pages. Apps built with this hybrid rendering approach have also been called transitional apps.

Static site generation

You can use SvelteKit as a static site generator (SSG) that fully prerenders your site with static rendering using `adapter-static`. You may also use the prerender option to prerender only some pages and then choose a different adapter with which to dynamically server-render other pages.

Tools built solely to do static site generation may scale the prerendering process more efficiently during build when rendering a very large number of pages. When working with very large statically generated sites, you can avoid long build times with Incremental Static Regeneration (ISR) if using `adapter-vercel`. And in contrast to purpose-built SSGs, SvelteKit allows for nicely mixing and matching different rendering types on different pages.

Single-page app

Single-page apps (SPAs) exclusively use client-side rendering (CSR). You can build single-page apps (SPAs) with SvelteKit. As with all types of SvelteKit applications, you can write your backend in SvelteKit or another language or framework. If you are building an application with no backend or a separate backend, you can simply skip over and ignore the parts of the docs talking about `server` files.

Multi-page app

SvelteKit isn't typically used to build traditional multi-page apps. However, in SvelteKit you can remove all JavaScript on a page with `csr = false`, which will render subsequent links on the server, or you can use `data-sveltekit-reload` to render specific links on the server.

Separate backend

If your backend is written in another language such as Go, Java, PHP, Ruby, Rust, or C#, there are a couple of ways that you can deploy your application. The most recommended way would be to deploy your SvelteKit frontend separately from your backend utilizing `adapter-node` or a serverless adapter. Some users prefer not to have a separate process to manage and decide to deploy their application as a single-page app (SPA) served by their backend server, but note that single-page apps have worse SEO and performance characteristics.

If you are using an external backend, you can simply skip over and ignore the parts of the docs talking about `server` files. You may also want to reference the FAQ about how to make calls to a separate backend.

Serverless app

SvelteKit apps are simple to run on serverless platforms. The default zero config adapter will automatically run your app on a number of supported platforms or you can use `adapter-vercel`, `adapter-netlify`, or `adapter-cloudflare` to provide platform-specific configuration. And community adapters allow you to deploy your application to almost any serverless environment. Some of these adapters such as `adapter-vercel` and `adapter-netlify` offer an `edge` option, to support edge rendering for improved latency.

Your own server

You can deploy to your own server or VPS using `adapter-node`.

Container

You can use `adapter-node` to run a SvelteKit app within a container such as Docker or LXC.

Library

You can create a library to be used by other Svelte apps with the `@sveltejs/package` add-on to SvelteKit by choosing the library option when running `sv create`.

Offline app

SvelteKit has full suppport for service workers allowing you to build many types of applications such as offline apps and progressive web apps.

Mobile app

You can turn a SvelteKit SPA into a mobile app with Tauri or Capacitor. Mobile features like the camera, geolocation, and push notifications are available via plugins for both platforms.

These mobile development platforms work by starting a local web server and then serving your application like a static host on your phone. You may find `bundleStrategy: 'single'` to be a helpful option to limit the number of requests made. E.g. at the time of writing, the Capacitor local server uses HTTP/1, which limits the number of concurrent connections.

Desktop app

You can turn a SvelteKit SPA into a desktop app with Tauri, Wails, or Electron.

Browser extension

You can build browser extensions using either `adapter-static` or community adapters specifically tailored towards browser extensions.

Embedded device

Because of its efficient rendering, Svelte can be run on low power devices. Embedded devices like microcontrollers and TVs may limit the number of concurrent connections. In order to reduce the number of concurrent requests, you may find `bundleStrategy: 'single'` to be a helpful option in this deployment configuration.

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT · CORE CONCEPTS

Routing

At the heart of SvelteKit is a *file system-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.

We'll introduce these files in a moment in more detail, but here are a few simple rules to help you remember how SvelteKit's routing works:

- All files can run on the server
- All files run on the client except `+server` files
- `+layout` and `+error` files apply to subdirectories as well as the directory they live in

+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.

src/routes/+page.svelte

```
<h1>Hello and welcome to my site!</h1>
<a href="/about">About my site</a>
```

src/routes/about/+page.svelte

```
<h1>About this site</h1>
<p>TODO...</p>
<a href="/">Home</a>
```

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

Pages can receive data from `load` functions via the `data` prop.

src/routes/blog/[slug]/+page.svelte TS

```
<script lang="ts">
  import type { PageProps } from './$types';

  let { data }: PageProps = $props();
</script>

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

Legacy mode: hide all

`PageProps` was added in 2.16.0. In earlier versions, you had to type the `data` property manually with `$update` instead, see $types.

In Svelte 4, you'd use `export let data` instead.

+page.js

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

src/routes/blog/[slug]/+page.ts TS

```
import { error } from '@sveltejs/kit';
import type { PageLoad } from './$types';

export const load: PageLoad = ({ params }) => {
  if (params.slug === 'hello-world') {
    return {
      title: 'Hello world!',
      content: 'Welcome to our blog. Lorem ipsum dolor sit amet...'
    };
  }

  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`.

src/routes/blog/[slug]/+page.server.ts TS

```
import { error } from '@sveltejs/kit';
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async ({ params }) => {
  const post = await getPostFromDatabase(params.slug);

  if (post) {
    return post;
  }

  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:

src/routes/blog/[slug]/+error.svelte TS

```
<script lang="ts">
  import { page } from '$app/state';
</script>

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

Legacy mode: hide all

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

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

If the error occurs inside a `load` function in `+layout(.server).js`, the closest error boundary in the tree is an `+error.svelte` file above that layout (or next to it).

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

`+error.svelte` is *not* used when an error occurs inside `hooks`, 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...

```
<script>
  let { children } = $props();
</script>

{@render children()}
```

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

src/routes/+layout.svelte TS

```
<script lang="ts">
  let { children } = $props();
</script>

<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/settings">Settings</a>
</nav>

{@render children()}
```

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

src/routes/+page.svelte

```
<h1>Home</h1>
```

src/routes/about/+page.svelte

```
<h1>About</h1>
```

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 inherithing the root layout with the top-level nav):

src/routes/settings/+layout.svelte TS

```
<script lang="ts">
  import type { LayoutProps } from './$types';

  let { data, children }: LayoutProps = $props();
</script>

<h1>Settings</h1>

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

{@render children()}
```

Legacy mode: hide all

`LayoutProps` was added in 2.16.0. In earlier versions, you had to type the properties manually instead.

You can see how `data` is populated by looking at the `+layout.js` example in the next section just below.

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

+layout.js

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

src/routes/settings/+layout.ts TS

```
import type { LayoutLoad } from './$types';

export const load: LayoutLoad = () => {
  return {
    sections: [
      { slug: 'profile', title: 'Profile' },
      { slug: 'notifications', title: 'Notifications' }
    ]
  };
};
```

If a `+layout.js` exports page options — `prerender`, `ssr` and `csr` — they will be used as default values for all pages that belong to this layout.

You can access data from `+layout.js` in `+layout.svelte` and other layouts and pages using the `page.data` object and `$app/state`.

src/routes/+layout.svelte

```
<script lang="ts">
  import { page } from '$app/state';
</script>

<h1>{page.data.title}</h1>
```

+layout.server.js

To invoke a load function only on the server, you can create a `+layout.server.js` file. You can also add server-only page options to this file.

If a `+layout.server.js` file exists, data returned from a server `load` function will be passed to the corresponding layout as `data`.

src/routes/+layout.server.ts TS

```
import type { LayoutServerLoad } from './$types';

export const load: LayoutServerLoad = () => {
  return {
    sections: [
      { slug: 'profile', title: 'Profile' },
      { slug: 'notifications', title: 'Notifications' }
    ]
  };
};
```

+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:

src/routes/blog/[slug]/+error.svelte TS

```
<script lang="ts">
  import { page } from '$app/state';
</script>

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

Legacy mode: hide all

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

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

If the error occurs inside a `load` function in `+layout(.server).js`, the closest error boundary in the tree is an `+error.svelte` file above that layout (or next to it).

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

`+error.svelte` is *not* used when an error occurs inside `hooks`, 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...

```
<script>
  let { children } = $props();
</script>

{@render children()}
```

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

src/routes/+layout.svelte TS

```
<script lang="ts">
  let { children } = $props();
</script>

<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/settings">Settings</a>
</nav>

{@render children()}
```

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

src/routes/+page.svelte

```
<h1>Home</h1>
```

src/routes/about/+page.svelte

```
<h1>About</h1>
```

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 inherithing the root layout with the top-level nav):

src/routes/settings/+layout.svelte TS

```
<script lang="ts">
  import type { LayoutProps } from './$types';

  let { data, children }: LayoutProps = $props();
</script>

<h1>Settings</h1>

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

{@render children()}
```

Legacy mode: hide all

`LayoutProps` was added in 2.16.0. In earlier versions, you had to type the properties manually instead.

You can see how `data` is populated by looking at the `+layout.js` example in the next section just below.

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

+layout.js

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

src/routes/settings/+layout.ts TS

```
import type { LayoutLoad } from './$types';

export const load: LayoutLoad = () => {
  return {
    sections: [
      { slug: 'profile', title: 'Profile' },
      { slug: 'notifications', title: 'Notifications' }
    ]
  };
};
```

If a `+layout.js` exports page options — `prerender`, `ssr` and `csr` — they will be used as default values for all pages that belong to this layout.

You can access data from `+layout.js` in `+layout.svelte` and other layouts and pages using the `page.data` object and `$app/state`.

src/routes/+layout.svelte

```
<script lang="ts">
  import { page } from '$app/state';
</script>

<h1>{page.data.title}</h1>
```

+layout.server.js

To invoke a load function only on the server, you can create a `+layout.server.js` file. You can also add server-only page options to this file.

If a `+layout.server.js` file exists, data returned from a server `load` function will be passed to the corresponding layout as `data`.

src/routes/+layout.server.ts TS

```
import type { LayoutServerLoad } from './$types';

export const load: LayoutServerLoad = () => {
  return {
    sections: [
      { slug: 'profile', title: 'Profile' },
      { slug: 'notifications', title: 'Notifications' }
    ]
  };
};
```

On this page
Routing
+page
+error
+layout
+server
$types
Other files
Further reading

Server only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper
Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node

SVELTEKIT • BEST PRACTICES

SEO

On this page
SEO
Out of the box
Manual setup

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. Read the performance page for more details.

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.

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:

src/routes/sitemap.xml/+server.js

```
export async function GET() {
  return new Response(
    `
    <?xml version="1.0" encoding="UTF-8" ?>
    <urlset
      xmlns="https://www.sitemaps.org/schemas/sitemap/0.9"
      xmlns:xhtml="https://www.w3.org/1999/xhtml"
      xmlns:mobile="https://www.google.com/schemas/sitemap-mobile/1.0"
      xmlns:news="https://www.google.com/schemas/sitemap-news/0.9"
      xmlns:image="https://www.google.com/schemas/sitemap-image/1.1"
      xmlns:video="https://www.google.com/schemas/sitemap-video/1.1"
    >
      <!-- <url> elements go here -->
    </urlset>`.trim(),
    {
      headers: {
        'Content-Type': 'application/xml'
      }
    }
  );
}
```

AMP

An unfortunate reality of modern web development is that it is sometimes necessary to create an Accelerated Mobile Pages (AMP) version of your site. In SvelteKit this can be done by setting the `inlineStyleThreshold` option...

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`...

src/routes/+layout.server.js

```
export const csr = false;
```

...adding `amp` to your `app.html`

```
<html amp>
...
```

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

src/hooks.server.ts JS TS

```
import * as amp from '@sveltejs/amp';
import type { Handle } from '@sveltejs/kit';

export const handle: Handle = async ({ event, resolve }) => {
  let buffer = '';
  return await resolve(event, {
    transformPageChunk: ({ html, done }) => {
      buffer += html;
      if (done) return amp.transform(buffer);
    }
  });
};
```

To prevent shipping any unused CSS as a result of transforming the page to amp, we can use `dropcss`:

src/hooks.server.ts JS TS

```
import * as amp from '@sveltejs/amp';
import dropcss from 'dropcss';
import type { Handle } from '@sveltejs/kit';

export const handle: Handle = async ({ event, resolve }) => {
  let buffer = '';

  return await resolve(event, {
    transformPageChunk: ({ html, done }) => {
      buffer += html;

      if (done) {
        let css = '';
        const markup = amp
          .transform(buffer)
          .replace('⚡', 'amp') // dropcss can't handle this character
          .replace(/<style amp-custom([^>]+?)>([^]+?)<\/style>/, (match, attributes, con
            css = contents;
            return `<style amp-custom${attributes}></style>`;
          });

        css = dropcss({ css, html: markup }).css;
        return markup.replace('</style>', `${css}</style>`);
      }
    }
  });
};
```

> It's a good idea to use the `handle` hook to validate the transformed HTML using `amphtml-validator`, but only if you're prerendering pages since it's very slow.

Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT • ADVANCED

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.

On this page
Server-only modules
Private environment variables
Server-only utilities
Your modules
How it works
Further reading

Private environment variables

The `$env/static/private` and `$env/dynamic/private` modules can only be imported into modules that only run on the server, such as `hooks.server.js` or `+page.server.js`.

Server-only utilities

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

Your modules

You can make your own modules server-only in two ways:

- adding `.server` to the filename, e.g. `secrets.server.js`
- placing them in `$lib/server`, e.g. `$lib/server/secrets.js`

How it works

Any time you have public-facing code that imports server-only code (whether directly or indirectly)...

$lib/server/secrets.js

```
export const atlantisCoordinates = [/* redacted */];
```

src/routes/utils.js

```
export { atlantisCoordinates } from '$lib/server/secrets.js';

export const add = (a, b) => a + b;
```

src/routes/+page.svelte

```
<script>
  import { add } from './utils.js';
</script>
```

...SvelteKit will error:

```
Cannot import $lib/server/secrets.js into public-facing code:
- src/routes/+page.svelte
  - src/routes/utils.js
    - $lib/server/secrets.js
```

Even though the public-facing code — `src/routes/+page.svelte` — only uses the `add` export and not the secret `atlantisCoordinates` export, the secret code could end up in JavaScript that the browser downloads, and so the import chain is considered unsafe.

This feature also works with dynamic imports, even interpolated ones like `await import(`./${foo}.js`)`, with one small caveat: during development, if there are two or more dynamic imports between the public-facing code and the server-only module, the illegal import will not be detected the first time the code is loaded.

> Unit testing frameworks like Vitest do not distinguish between server-only and public-facing code. For this reason, illegal import detection is disabled when running tests, as determined by `process.env.TEST === 'true'`.

Further reading

- Tutorial: Environment variables

Edit this page on GitHub

Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT • ADVANCED

Service workers

On this page
Service workers
Inside the service worker
During development
Type safety
Other solutions
References

Service workers act as proxy servers that handle network requests inside your app. This makes it possible to make your app work offline, but even if you don't need offline support (or can't realistically implement it because of the type of app you're building), it's often worth using service workers to speed up navigation by precaching your built JS and CSS.

In SvelteKit, if you have a `src/service-worker.js` file (or `src/service-worker/index.js`) it will be bundled and automatically registered. You can change the location of your service worker if you need to.

You can disable automatic registration if you need to register the service worker with your own logic or use another solution. The default registration looks something like this:

```
if ('serviceWorker' in navigator) {
  addEventListener('load', function () {
    navigator.serviceWorker.register('./path/to/service-worker.js');
  });
}
```

Inside the service worker

Inside the service worker you have access to the `$service-worker` module, which provides you with the paths to all static assets, build files and prerendered pages. You're also provided with an app version string, which you can use for creating a unique cache name, and the deployment's `base` path. If your Vite config specifies `define` (used for global variable replacements), this will be applied to service workers as well as your server/client builds.

The following example caches the built app and any files in `static` eagerly, and caches all other requests as they happen. This would make each page work offline once visited.

```
/// <reference types="@sveltejs/kit" />
import { build, files, version } from '$service-worker';

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

const ASSETS = [
  ...build, // the app itself
  ...files  // everything in `static`
];

self.addEventListener('install', (event) => {
  // Create a new cache and add all files to it
  async function addFilesToCache() {
    const cache = await caches.open(CACHE);
    await cache.addAll(ASSETS);
  }

  event.waitUntil(addFilesToCache());
});

self.addEventListener('activate', (event) => {
  // Remove previous cached data from disk
  async function deleteOldCaches() {
    for (const key of await caches.keys()) {
      if (key !== CACHE) await caches.delete(key);
    }
  }

  event.waitUntil(deleteOldCaches());
});

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

  async function respond() {
    const url = new URL(event.request.url);
    const cache = await caches.open(CACHE);

    // `build`/`files` can always be served from the cache
    if (ASSETS.includes(url.pathname)) {
      const response = await cache.match(url.pathname);

      if (response) {
        return response;
      }
    }

    // for everything else, try the network first, but
    // fall back to the cache if we're offline
    try {
      const response = await fetch(event.request);

      // if we're offline, fetch can return a value that is not a Response
      // instead of throwing - and we can't pass this non-Response to respondWith
      if (!(response instanceof Response)) {
        throw new Error('invalid response from fetch');
      }

      if (response.status === 200) {
        cache.put(event.request, response.clone());
      }

      return response;
    } catch (err) {
      const response = await cache.match(event.request);

      if (response) {
        return response;
      }

      // if there's no cache, then just error out
      // as there is nothing we can do to respond to this request
      throw err;
    }
  }

  event.respondWith(respond());
});
```

Be careful when caching! In some cases, stale data might be worse than data that's unavailable while offline. Since browsers will empty caches if they get too full, you should also be careful about caching large assets like video files.

During development

The service worker is bundled for production, but not during development. For that reason, only browsers that support modules in service workers will be able to use them at dev time. If you are manually registering your service worker, you will need to pass the `{ type: 'module' }` option in development:

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

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

`build` and `prerendered` are empty arrays during development

Type safety

Setting up proper types for service workers requires some manual setup. Inside your `service-worker.js`, add the following to the top of your file:

```
/// <reference types="@sveltejs/kit" />
/// <reference no-default-lib="true"/>
/// <reference lib="esnext" />
/// <reference lib="webworker" />

const sw = /** @type {ServiceWorkerGlobalScope} */ (/** @type {unknown} */ (self));
```

```
/// <reference types="@sveltejs/kit" />
/// <reference no-default-lib="true"/>
/// <reference lib="esnext" />
/// <reference lib="webworker" />

const sw = self as unknown as ServiceWorkerGlobalScope;
```

This disables access to DOM typings like `HTMLElement` which are not available inside a service worker and instantiates the correct globals. The reassignment of `self` to `sw` allows you to type cast it in the process (there are a couple of ways to do this, but this is the easiest that requires no additional files). Use `sw` instead of `self` in the rest of the file. The reference to the SvelteKit types ensures that the `$service-worker` import has proper type definitions. If you import `$env/static/public` you either have to `// @ts-ignore` the import or add `/// <reference types="../.svelte-kit/ambient.d.ts" />` to the reference types.

Other solutions

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

References

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

Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT · ADVANCED

Shallow routing

On this page
Shallow routing
API
Loading data for a route
Caveats

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

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

SvelteKit makes this possible with the `pushState` and `replaceState` functions, which allow you to associate state with a history entry without navigating. For example, to implement a history-driven modal:

+page.svelte JS TS

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

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

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

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

API

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

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

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

Legacy mode hide all

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

Loading data for a route

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

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

src/routes/photos/+page.svelte JS TS

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

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

{#each data.thumbnails as thumbnail}
  <a
    href="/photos/{thumbnail.id}"
    onclick={async (e) => {
      if (innerWidth < 640        // bail if the screen is too small
        || e.shiftKey             // or the link is opened in a new window
        || e.metaKey || e.ctrlKey // or a new tab (mac: metaKey, win/linux: ctrlKey)
        // should also consider clicking with a mouse scroll wheel
      ) return;

      // prevent navigation
      e.preventDefault();

      const { href } = e.currentTarget;

      // run `load` functions (or rather, get the result of the `load` functions
      // that are already running because of `data-sveltekit-preload-data`)
      const result = await preloadData(href);

      if (result.type === 'loaded' && result.status === 200) {
        pushState(href, { selected: result.data });
      } else {
        // something bad happened! try navigating
        goto(href);
      }
    }}
  >
    <img alt={thumbnail.alt} src={thumbnail.src} />
  </a>
{/each}

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

Caveats

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

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

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Single-page apps

On this page
Single-page apps
Usage
Apache
Prerendering individual pages

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

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

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

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

Usage

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

```
svelte.config.js
import adapter from '@sveltejs/adapter-static';

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

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

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

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

Apache

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

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

Prerendering individual pages

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

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

Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Snapshots
Shallow routing
Packaging

Best practices
Auth
Performance
Images
Accessibility
SEO

Appendix
Frequently asked questions
Integrations
Breakpoint Debugging
Migrating to SvelteKit v2
Migrating from Sapper

SVELTEKIT • ADVANCED

Snapshots

On this page
Snapshots

Ephemeral DOM state — like scroll positions on sidebars, the content of `<input>` elements and so on — is discarded when you navigate from one page to another.

For example, if the user fills out a form but navigates away and then back before submitting, or if the user refreshes the page, the values they filled in will be lost. In cases where it's valuable to preserve that input, you can take a *snapshot* of DOM state, which can then be restored if the user navigates back.

To do this, export a `snapshot` object with `capture` and `restore` methods from a `+page.svelte` or `+layout.svelte`:

+page.svelte JS TS

```
<script lang="ts">
  import type { Snapshot } from './$types';

  let comment = $state('');
  export const snapshot: Snapshot<string> = {
    capture: () => comment,
    restore: (value) => comment = value
  };
</script>

<form method="POST">
  <label for="comment">Comment</label>
  <textarea id="comment" bind:value={comment} />
  <button>Post comment</button>
</form>
```

When you navigate away from this page, the `capture` function is called immediately before the page updates, and the returned value is associated with the current entry in the browser's history stack. If you navigate back, the `restore` function is called with the stored value as soon as the page is updated.

The data must be serializable as JSON so that it can be persisted to `sessionStorage`. This allows the state to be restored when the page is reloaded, or when the user navigates back from a different site.

> Avoid returning very large objects from `capture` — once captured, objects will be retained in memory for the duration of the session, and in extreme cases may be too large to persist to `sessionStorage`.

Edit this page on GitHub

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT • CORE CONCEPTS

State management

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

Avoid shared state on the server

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

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

```
+page.server.ts
import type { PageServerLoad, Actions } from './$types';
let user;
export const load: PageServerLoad = () => {
  return { user };
};
export const actions = {
  default: async ({ request }) => {
    const data = await request.formData();

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

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

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

No side-effects in load

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

```
+page.ts
import { user } from '$lib/user';
import type { PageLoad } from './$types';

export const load: PageLoad = async ({ fetch }) => {
  const response = await fetch('/api/user');

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

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

```
+page.ts
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async ({ fetch }) => {
  const response = await fetch('/api/user');

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

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

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

Using state and stores with context

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

```
src/routes/+layout.svelte
<script lang="ts">
  import { setContext } from 'svelte';
  import type { LayoutProps } from './$types';

  let { data }: LayoutProps = $props();

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

```
src/routes/user/+page.svelte
<script lang="ts">
  import { getContext } from 'svelte';

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

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

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

Legacy mode hide all

You also use stores from `svelte/store` for this, but when using Svelte 5 it is recommended to make use of universal reactivity instead.

Updating the value of context-based state in deeper-level pages or components while the page is being rendered via SSR will not affect the value in the parent component because it has already been rendered by the time the state value is updated. In contrast, on the client (when CSR is enabled, which is the default) the value will be propagated and components, pages, and layouts higher in the hierarchy will react to the new value. Therefore, to avoid values "flashing" during state updates during hydration, it is generally recommended to pass state down into components rather than up.

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

Component and page state is preserved

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

```
src/routes/blog/[slug]/+page.svelte
<script lang="ts">
  import type { PageProps } from './$types';

  let { data }: PageProps = $props();

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

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

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

...then navigating from `/blog/my-short-post` to `/blog/my-long-post` won't cause the layout, page and any other components within to be destroyed and recreated. Instead the `data` prop (and by extension `data.title` and `data.content`) will update (as it would with any other Svelte component) and, because the code isn't rerunning, lifecycle methods like `onMount` and `onDestroy` won't rerun and `estimatedReadingTime` won't be recalculated.

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

```
src/routes/blog/[slug]/+page.svelte
<script lang="ts">
  import type { PageProps } from './$types';

  let { data }: PageProps = $props();

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

If your code in `onMount` and `onDestroy` has to run again after navigation you can use afterNavigate and beforeNavigate respectively.

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

```
<script>
  import { page } from '$app/state';
</script>

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

Storing state in the URL

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

Storing ephemeral state in snapshots

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

On this page
State management
Avoid shared state on the server
No side-effects in load
Using state and stores with context
Component and page state is preserved
Storing state in the URL
Storing ephemeral state in snapshots

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules

SVELTEKIT › BUILD AND DEPLOY

Static site generation

On this page
Static site generation
Usage
Zero-config support
Options
GitHub Pages

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

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

Usage

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

```
svelte.config.js
import adapter from '@sveltejs/adapter-static';

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

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

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

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

Zero-config support

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

- Vercel

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

```
svelte.config.js
export default {
  kit: {
    adapter: adapter({...})
  }
};
```

Options

pages

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

assets

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

fallback

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

precompress

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

strict

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

GitHub Pages

When building for GitHub Pages, if your repo name is not equivalent to `your-username.github.io`, make sure to update `config.kit.paths.base` to match your repo name. This is because the site will be served from `https://your-username.github.io/your-repo-name` rather than from the root.

You'll also want to generate a fallback `404.html` page to replace the default 404 page shown by GitHub Pages.

A config for GitHub Pages might look like the following:

```
svelte.config.js
import adapter from '@sveltejs/adapter-static';

/** @type {import('@sveltejs/kit').Config} */
const config = {
  kit: {
    adapter: adapter({
      fallback: '404.html'
    }),
    paths: {
      base: process.argv.includes('dev') ? '' : process.env.BASE_PATH
    }
  }
};

export default config;
```

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

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

on:
  push:
    branches: 'main'

jobs:
  build_site:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      # If you're using pnpm, add this step then change the commands and cache key below
      # - name: Install pnpm
      #   uses: pnpm/action-setup@v3
      #   with:
      #     version: 8

      - name: Install Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: npm

      - name: Install dependencies
        run: npm install

      - name: build
        env:
          BASE_PATH: '/${{ github.event.repository.name }}'
        run: |
          npm run build

      - name: Upload Artifacts
        uses: actions/upload-pages-artifact@v3
        with:
          # this should match the `pages` option in your adapter-static options
          path: 'build/'

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

    permissions:
      pages: write
      id-token: write

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

    steps:
      - name: Deploy
        id: deployment
        uses: actions/deploy-pages@v4
```

If you're not using GitHub actions to deploy your site (for example, you're pushing the built site to its own repo), add an empty `.nojekyll` file in your `static` directory to prevent Jekyll from interfering.

Additional resources
Glossary

Reference
@sveltejs/kit
@sveltejs/kit/hooks
@sveltejs/kit/node/polyfills
@sveltejs/kit/node
@sveltejs/kit/vite
$app/environment
$app/forms
$app/navigation
$app/paths
$app/server
$app/state
$app/stores
$env/dynamic/private
$env/dynamic/public
$env/static/private
$env/static/public
$lib
$service-worker
Configuration
Command Line Interface
Types

SVELTEKIT › REFERENCE

Types

On this page
Types
Generated types
$lib
app.d.ts
Error
Locals
PageData
PageState
Platform

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:

src/routes/[foo]/[bar]/[baz]/+server.ts TS

```
import type { RequestHandler} from '@sveltejs/kit';
export const GET: RequestHandler={
  foo: string;
  bar: string;
  baz: string
  }> = async ({ 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:

.svelte-kit/types/src/routes/[foo]/[bar]/[baz]/$types.d.ts

```
import type * as Kit from '@sveltejs/kit';

type RouteParams = {
  foo: string;
  bar: string;
  baz: string;
};

export type RequestHandler = Kit.RequestHandler<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:

src/routes/[foo]/[bar]/[baz]/+server.ts TS

```
import type { RequestHandler } from './$types';

export const GET: RequestHandler = async ({ params }) => {
  // ...
};
```

src/routes/[foo]/[bar]/[baz]/+page.ts TS

```
import type { PageLoad } from './$types';

export const load: PageLoad = async ({ params, fetch }) => {
  // ...
};
```

The return types of the `load` functions are then available through the `$types` module as `PageData` and `LayoutData` respectively, while the union of the return values of all `Actions` is available as `ActionData`.

Starting with version 2.16.0, two additional helper types are provided: `PageProps` defines `data: PageData`, as well as `form: ActionData`, when there are actions defined, while `LayoutProps` defines `data: LayoutData`, as well as `children: Snippet`.

src/routes/+page.svelte TS

```
<script lang="ts">
  import type { PageProps } from './$types';

  let { data, form }: PageProps = $props();
</script>
```

Legacy mode hide all

Before 2.16.0:

src/routes/+page.svelte TS

```
<script lang="ts">
  import type { PageData, ActionData } from './$types';

  let { data, form }: { data: PageData, form: ActionData } = $props();
</script>
```

Using Svelte 4:

src/routes/+page.svelte TS

```
<script lang="ts">
  import type { PageData, ActionData } from './$types';


  export let data: PageData;

  export let form: ActionData;
</script>
```

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:

.svelte-kit/tsconfig.json

```
{
  "compilerOptions": {
    "paths": {
      "$lib": ["../src/lib"],
      "$lib/*": ["../src/lib/*"]
    },
    "rootDirs": ["..", "./types"]
  },
  "include": [
    "ambient.d.ts",
    "non-ambient.d.ts",
    "./types/**/$types.d.ts",
    "../vite.config.js",
    "../vite.config.ts",
    "../src/**/*.js",
    "../src/**/*.ts",
    "../src/**/*.svelte",
    "../tests/**/*.js",
    "../tests/**/*.ts",
    "../tests/**/*.svelte"
  ],
  "exclude": [
    "../node_modules/**",
    "../src/service-worker.js",
    "../src/service-worker/**/*.js",
    "../src/service-worker.ts",
    "../src/service-worker/**/*.ts",
    "../src/service-worker.d.ts",
    "../src/service-worker/**/*.d.ts"
  ]
}
```

Others are required for SvelteKit to work properly, and should also be left untouched unless you know what you're doing:

.svelte-kit/tsconfig.json

```
{
  "compilerOptions": {
    // this ensures that types are explicitly
    // imported with `import type`, which is
    // necessary as Svelte/Vite cannot
    // otherwise compile components correctly
    "verbatimModuleSyntax": true,

    // Vite compiles one TypeScript module
    // at a time, rather than compiling
    // the entire module graph
    "isolatedModules": true,

    // Tell TS it's used only for type-checking
    "noEmit": true,

    // This ensures both `vite build`
    // and `svelte-package` work correctly
    "lib": ["esnext", "DOM", "DOM.Iterable"],
    "moduleResolution": "bundler",
    "module": "esnext",
    "target": "esnext"
  }
}
```

$lib

This is a simple alias to `src/lib`, or whatever directory is specified as `config.kit.files.lib`. It allows you to access common components and utility modules without `../../../`-ish nonsense.

$lib/server

A subdirectory of `$lib`, SvelteKit will prevent you from importing any modules in `$lib/server` into client-side code. See server-only modules.

app.d.ts

The `app.d.ts` file is home to the ambient types of your apps, i.e. types that are available without explicitly importing them.

Always part of this file is the `App` namespace. This namespace contains several types that influence the shape of certain SvelteKit features you interact with.

Error

Defines the common shape of expected and unexpected errors. Expected errors are thrown using the `error` function. Unexpected errors are handled by the `handleError` hooks which should return this shape.

```
interface Error {...}
```

```
message: string;
```

Locals

The interface that defines `event.locals`, which can be accessed in server hooks (`handle`, and `handleError`), server-only `load` functions, and `+server.js` files.

```
interface Locals {}
```

PageData

Defines the common shape of the page.data state and $page.data store - that is, the data that is shared between all pages. The `load` and `ServerLoad` functions in `./$types` will be narrowed accordingly. Use optional properties for data that is only present on specific pages. Do not add an index signature (`[key: string]: any`).

```
interface PageData {}
```

PageState

The shape of the `page.state` object, which can be manipulated using the pushState and replaceState functions from `$app/navigation`.

```
interface PageState {}
```

Platform

If your adapter provides platform-specific context via `event.platform`, you can specify it here.

```
interface Platform {}
```

Edit this page on GitHub

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT › BUILD AND DEPLOY

Vercel

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

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

Usage

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

svelte.config.js

```
import adapter from '@sveltejs/adapter-vercel';

export default {
  kit: {
    adapter: adapter({
      // see below for options that can be set here
    })
  }
};
```

Deployment configuration

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

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

about/+page.ts

```
import type { Config } from '@sveltejs/adapter-vercel';

export const config: Config = {
  runtime: 'edge'
};
```

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

admin/+layout.ts

```
import type { Config } from '@sveltejs/adapter-vercel';

export const config: Config = {
  runtime: 'nodejs22.x'
};
```

The following options apply to all functions:

- `runtime`: `'edge'`, `'nodejs18.x'`, `'nodejs20.x'` or `'nodejs22.x'`. By default, the adapter will select the `'nodejs{version}.x'` corresponding to the Node version your project is configured to use on the Vercel dashboard
- `regions`: an array of edge network regions (defaulting to `['iad1']` for serverless functions) or `'all'` if `runtime` is `edge` (its default). Note that multiple regions for serverless functions are only supported on Enterprise plans
- `split`: if `true`, causes a route to be deployed as an individual function. If `split` is set to `true` at the adapter level, all routes will be deployed as individual functions

Additionally, the following option applies to edge functions:

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

And the following option apply to serverless functions:

- `memory`: the amount of memory available to the function. Defaults to `1024` Mb, and can be decreased to `128` Mb or increased in 64Mb increments up to `3008` Mb on Pro or Enterprise accounts
- `maxDuration`: maximum execution duration of the function. Defaults to `10` seconds for Hobby accounts, `15` for Pro and `900` for Enterprise
- `isr`: configuration Incremental Static Regeneration, described below

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

Image Optimization

You may set the `images` config to control how Vercel builds your images. See the image configuration reference for full details. As an example, you may set:

svelte.config.js

```
import adapter from '@sveltejs/adapter-vercel';

export default {
  kit: {
    adapter: adapter({
      images: {
        sizes: [640, 828, 1200, 1920, 3840],
        formats: ['image/avif', 'image/webp'],
        minimumCacheTTL: 300,
        domains: ['example-app.vercel.app'],
      }
    })
  }
};
```

Incremental Static Regeneration

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

> Use ISR only on routes where every visitor should see the same content (much like when you prerender). If there's anything user-specific happening (like session cookies), they should happen on the client via JavaScript only to not leak sensitive information across visits

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

```
import { BYPASS_TOKEN } from '$env/static/private';

export const config = {
  isr: {
    expiration: 60,
    bypassToken: BYPASS_TOKEN,
    allowQuery: ['search']
  }
};
```

> Using ISR on a route with `export const prerender = true` will have no effect, since the route is prerendered at build time

The `expiration` property is required; all others are optional. The properties are discussed in more detail below.

expiration

The expiration time (in seconds) before the cached asset will be re-generated by invoking the Serverless Function. Setting the value to `false` means it will never expire. In that case, you likely want to define a bypass token to re-generate on demand.

bypassToken

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

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

Note that the `BYPASS_TOKEN` string must be at least 32 characters long. You could generate one using the JavaScript console like so:

```
crypto.randomUUID();
```

Set this string as an environment variable on Vercel by logging in and going to your project then Settings > Environment Variables. For "Key" type `BYPASS_TOKEN` and for "value" use the string generated above, then hit "Save".

To get this key known about for local development, you can use the Vercel CLI by running the `vercel env pull` command locally like so:

```
vercel env pull .env.development.local
```

allowQuery

A list of valid query parameters that contribute to the cache key. Other parameters (such as utm tracking codes) will be ignored, ensuring that they do not result in content being re-generated unnecessarily. By default, query parameters are ignored.

> Pages that are prerendered will ignore ISR configuration.

Environment variables

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

+layout.server.ts

```
import { VERCEL_COMMIT_REF } from '$env/static/private';
import type { LayoutServerload } from './$types';

export const load: LayoutServerload = () => {
  return {
    deploymentGitBranch: VERCEL_COMMIT_REF
  };
};
```

+layout.svelte

```
<script lang="ts">
  import type { LayoutProps } from './$types';

  let { data }: LayoutProps = $props();
</script>

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

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

Skew protection

When a new version of your app is deployed, assets belonging to the previous version may no longer be accessible. If a user is actively using your app when this happens, it can cause errors when they navigate — this is known as *version skew*. SvelteKit mitigates this by detecting errors resulting from version skew and causing a hard reload to get the latest version of the app, but this will cause any client-side state to be lost. (You can also proactively mitigate it by observing the `updated` store value, which tells clients when a new version has been deployed.)

Skew protection is a Vercel feature that routes client requests to their original deployment. When a user visits your app, a cookie is set with the deployment ID, and any subsequent requests will be routed to that deployment for as long as skew protection is active. When they reload the page, they will get the newest deployment. (The `updated` store is exempted from this behaviour, and so will continue to report new deployments.) To enable it, visit the Advanced section of your project settings on Vercel.

Cookie-based skew protection comes with one caveat: if a user has multiple versions of your app open in multiple tabs, requests from older versions will be routed to the newer one, meaning they will fall back to SvelteKit's built-in skew protection.

Notes

Vercel functions

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

Node version

Projects created before a certain date may default to using an older Node version than what SvelteKit currently requires. You can change the Node version in your project settings.

Troubleshooting

Accessing the file system

You can't use `fs` in edge functions.

You *can* use it in serverless functions, but it won't work as expected, since files are not copied from your project into your deployment. Instead, use the `read` function from `$app/server` to access your files. `read` does not work inside routes deployed as edge functions (this may change in future).

Alternatively, you can prerender the routes in question.

Getting started
Introduction
Creating a project
Project types
Project structure
Web standards

Core concepts
Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers

SVELTEKIT • GETTING STARTED

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 Functions. During development, and in adapters for Node-based environments (including AWS Lambda), they're made available via polyfills where necessary (for now, that is — Node is rapidly adding support for more web standards).

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

Fetch APIs

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

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

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

Request

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

Response

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

Headers

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

src/routes/what-is-my-user-agent/+server.ts JS TS

```
import { json } from '@sveltejs/kit';
import type { RequestHandler } from './$types';

export const GET: RequestHandler = ({ request }) => {
  // log all headers
  console.log(...request.headers);

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

FormData

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

src/routes/hello/+server.ts JS TS

```
import { json } from '@sveltejs/kit';
import type { RequestHandler } from './$types';

export const POST: RequestHandler = async (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`:

```
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();
```

On this page
Web standards
Fetch APIs
FormData
Stream APIs
URL APIs
Web Crypto

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Writing adapters

On this page
Writing adapters

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

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

```
/** @param {AdapterSpecificOptions} options */
export default function (options) {
  /** @type {import('@sveltejs/kit').Adapter} */
  const adapter = {
    name: 'adapter-package-name',
    async adapt(builder) {
      // adapter implementation
    },
    async emulate() {
      return {
        async platform({ config, prerender }) {
          // the returned object becomes `event.platform` during dev, build and
          // preview. Its shape is that of `App.Platform`
        }
      }
    },
    supports: {
      read: ({ config, route }) => {
        // Return `true` if the route with the given `config` can use `read`
        // from `$app/server` in production, return `false` if it can't.
        // Or throw a descriptive error describing how to configure the deployment
      }
    }
  };

  return adapter;
}
```

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

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

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

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

Routing
Loading data
Form actions
Page options
State management

Build and deploy
Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

Advanced
Advanced routing
Hooks
Errors
Link options
Service workers
Server only modules

SVELTEKIT • BUILD AND DEPLOY

Zero-config deployments

On this page
Zero-config deployments
Environment-specific configuration
Adding community adapters

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

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

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

Environment-specific configuration

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

Adding community adapters

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

Edit this page on GitHub