

# Introduction

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## Before we begin

### NOTE

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

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

## What is SvelteKit?

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

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

## What is Svelte?

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

## SvelteKit vs Svelte



KIT | Docs

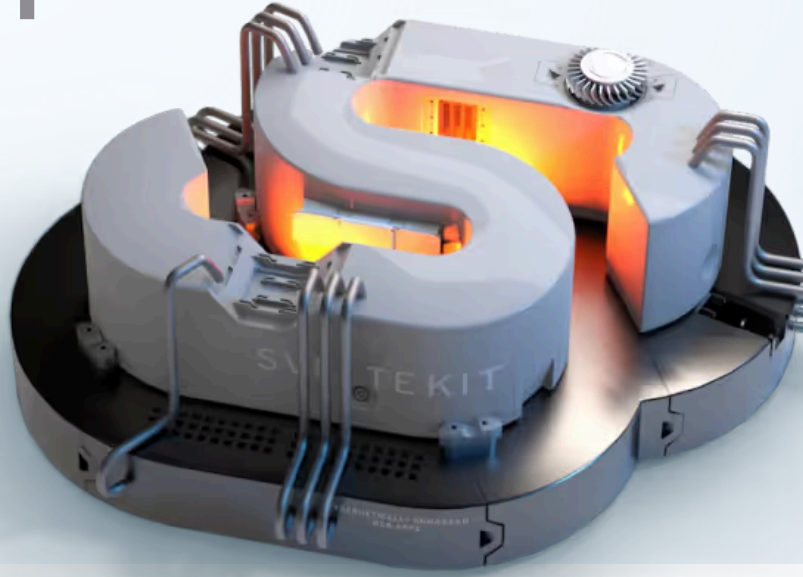


Svelte renders UI components. You can compose these components and render an entire page with just

# SVELTEKIT

web development, streamlined

[read the docs](#)



## fast

Powered by Svelte and Vite, speed is baked into every crevice: fast setup, fast dev, fast builds, fast page loads, fast navigation. Did we mention it's fast?

## fun

No more wasted days figuring out bundler configuration, routing, SSR, CSP, TypeScript, deployment settings and all the other boring stuff. Code with joy.

## flexible

SPA? MPA? SSR? SSG? Check. SvelteKit gives you the tools to succeed whatever it is you're building. And it runs wherever JavaScript does.

# Introduction

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## Before we begin

### NOTE

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

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

## What is SvelteKit?

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

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

## What is Svelte?

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

## SvelteKit vs Svelte



KIT | Docs



Svelte renders UI components. You can compose these components and render an entire page with just

# Before we begin

## NOTE

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

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

## What is SvelteKit?

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

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

## What is Svelte?

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

## SvelteKit vs Svelte

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

SvelteKit 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.



# What is SvelteKit?

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

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

## What is Svelte?

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

## SvelteKit vs Svelte

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

SvelteKit 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).

[PREVIOUS](#)

[NEXT](#)  
Creating a project

# What can I make with SvelteKit?

SvelteKit can be used to create most kinds of applications. Out of the box, SvelteKit supports many features including:

- Dynamic page content with load functions and API routes.
- SEO-friendly dynamic content with server-side rendering (SSR).
- User-friendly progressively-enhanced interactive pages with SSR and Form Actions.
- Static pages with prerendering.

SvelteKit can also be deployed to a wide spectrum of hosted architectures via adapters. In cases where SSR is used (or server-side logic is added without prerendering), those functions will be adapted to the target backend. Some examples include:

- Self-hosted dynamic web applications with a Node.js backend.
- Serverless web applications with backend loaders and APIs deployed as remote functions. See zero-config deployments for popular deployment options.
- Static pre-rendered sites such as a blog or multi-page site hosted on a CDN or static host. Statically-generated sites are shipped without a backend.
- Single-page Applications (SPAs) with client-side routing and rendering for API-driven dynamic content. SPAs are shipped without a backend and are not server-rendered. This option is commonly chosen when bundling SvelteKit with an app written in PHP, .Net, Java, C, Golang, Rust, etc.
- A mix of the above; some routes can be static, and some routes can use backend functions to fetch dynamic information. This can be configured with page options that includes the option to opt out of SSR.

In order to support SSR, a JS backend – such as Node.js or Deno-based server, serverless function, or edge function – is required.

It is also possible to write custom adapters or leverage community adapters to deploy SvelteKit to more platforms such as specialized server environments, browser extensions, or native applications. See integrations for more examples and integrations.

# How do I use HMR with SvelteKit?

SvelteKit has HMR enabled by default powered by `svelte-hmr`. If you saw Rich's presentation at the 2020 Svelte Summit, you may have seen a more powerful-looking version of HMR presented. This demo had `svelte-hmr`'s `preserveLocalState` flag on. This flag is now off by default because it may lead to unexpected behaviour and edge cases. But don't worry, you are still getting HMR with SvelteKit! If you'd like to preserve local state you can use the `@hmr:keep` or `@hmr:keep-all` directives as documented on the `svelte-hmr` page.

# What is SvelteKit?

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

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

# What is Svelte?

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

# SvelteKit vs Svelte

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

SvelteKit 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).

# What is SvelteKit?

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

To learn more about the kinds of applications you can build with SvelteKit, see the [FAQ](#).

# What is Svelte?

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

# SvelteKit vs Svelte

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

SvelteKit 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).



# Routing

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

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

## SPA

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

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

## SSG

Static Site Generation (SSG) is a term that refers to a site where every page is prerendered. SvelteKit was not built to do only static site generation like some tools and so may not scale as well to efficiently render a very large number of pages as tools built specifically for that purpose. However, in contrast to most purpose-built SSGs, SvelteKit does nicely allow for mixing and matching different rendering types on different pages. One benefit of fully prerendering a site is that you do not need to maintain or pay for servers to perform SSR. Once generated, the site can be served from CDNs, leading to great "time to first byte" performance. This delivery model is often referred to as JAMstack.

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

## SSR

Server-side rendering (SSR) is the generation of the page contents on the server. SSR is generally preferred for SEO. While some search engines can index content that is dynamically generated on the

# Service workers

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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';
```



## 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>
```



### NOTE

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

# Page options

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

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

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

## prerender

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

```
+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;
```



# SPA

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

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

# SSG

Static Site Generation (SSG) is a term that refers to a site where every page is prerendered. SvelteKit was not built to do only static site generation like some tools and so may not scale as well to efficiently render a very large number of pages as tools built specifically for that purpose. However, in contrast to most purpose-built SSGs, SvelteKit does nicely allow for mixing and matching different rendering types on different pages. One benefit of fully prerendering a site is that you do not need to maintain or pay for servers to perform SSR. Once generated, the site can be served from CDNs, leading to great "time to first byte" performance. This delivery model is often referred to as JAMstack.

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

# SSR

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

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

# CSR

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

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

## Hydration

Svelte components store some state and update the DOM when the state is updated. When fetching data during SSR, by default SvelteKit will store this data and transmit it to the client along with the server-rendered HTML. The components can then be initialized on the client with that data without having to call the same API endpoints again. Svelte will then check that the DOM is in the expected state and attach event listeners in a process called hydration. Once the components are fully hydrated, they can react to changes to their properties just like any newly created Svelte component.

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

## Prerendering

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

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

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

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

# Prerendering

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

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

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

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

## Routing

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

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

## SPA

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

# Images

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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



# Creating a project

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

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



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

There are two basic concepts:

Each page of your app is a Svelte component

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

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

## Editor setup

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

PREVIOUS

[Introduction](#)

NEXT

[Project structure](#)

# Introduction

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## Before we begin

### NOTE

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

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

## What is SvelteKit?

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

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

## What is Svelte?

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

## SvelteKit vs Svelte



KIT | Docs



Svelte renders UI components. You can compose these components and render an entire page with just

# Creating a project

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

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



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

There are two basic concepts:

Each page of your app is a Svelte component

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

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

## Editor setup

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

PREVIOUS

[Introduction](#)

NEXT

[Project structure](#)

## GETTING STARTED

# Project structure

[Edit this page on GitHub](#)

### ON THIS PAGE



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 `npm create svelte@latest`).

## Project files

# Web standards

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

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

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

## Fetch APIs

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

### NOTE

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

# Routing

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

`src/routes` is the root route

`src/routes/about` creates an `/about` route

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

### NOTE

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.

# Loading data

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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.js`



```
/** @type {import('./$types').PageLoad} */
export function load({ params }) {
  return {
    post: {
      title: `Title for ${params.slug} goes here`,
      content: `Content for ${params.slug} goes here`
    }
  };
}
```

`src/routes/blog/[slug]/+page.svelte`



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

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

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



# Form actions

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

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

## Default actions

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

`src/routes/login/+page.server.js`



```
/** @type {import('.$types').Actions} */
export const actions = {
  default: async (event) => {
    // TODO log the user in
  }
};
```

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

`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 type="submit">Log in</button>
</form>
```





# Page options

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

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

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

## prerender

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

```
+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
```



# State management

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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.js



```
let user;

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

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

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

# Building your app

[✎ Edit this page on GitHub](#)

### ON THIS PAGE



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

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

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

## During the build

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

```
+import { building } from '$app/environment';
import { setupMyDatabase } from '$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 –



# Adapters

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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
    })
  }
};
```

# Zero-config deployments

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

`@sveltejs/adapter-cloudflare` for Cloudflare Pages

`@sveltejs/adapter-netlify` for Netlify

`@sveltejs/adapter-vercel` for Vercel

`svelte-adapter-azure-swa` for Azure Static Web Apps

`svelte-kit-sst` for AWS via SST

`@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.

# Node servers

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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
```



# Static site generation

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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`



# Single-page apps

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

```
src/routes/+layout.js
```



```
export const ssr = false;
```

### NOTE

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
    })
  }
}
```



# Cloudflare Pages

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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 values specific to Cloudflare Workers are 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: 'wrangler.toml'
  }
}
```



# Cloudflare Workers

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

### NOTE

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({
      config: 'wrangler.toml',
      platformProxy: {
        configPath: 'wrangler.toml',
        environment: undefined,
        experimentalJsonConfig: false,
        persist: false
      }
    })
  }
};
```

# Netlify

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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:

# Vercel

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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...

# Writing adapters

[Edit this page on GitHub](#)

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:



# Advanced routing

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## Rest parameters

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

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



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

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



### NOTE

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

## 404 pages

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

```
src/routes/
├ marx-brothers/
│   ├── chico/
│   ├── harpo/
│   ├── kit/
│   └── docs
├ groucho/
└ ...
```



# Hooks

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



'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.

### NOTE

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`



```

KIT | Docs
/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
```



# Errors

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

## Error objects

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

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

## Expected errors

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

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



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

/** @type {import('.$types').PageServerLoad} */
export async function load({ params }) {
  const post = await db.getPost(params.slug);

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





# Link options

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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>
```



# Service workers

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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';
```



# Server-only modules

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



Like a good friend, SvelteKit keeps your secrets. When writing your backend and frontend in the same repository, it can be easy to accidentally import sensitive data into your front-end code (environment variables containing API keys, for example). SvelteKit provides a way to prevent this entirely: server-only modules.

## Private environment variables

The `$env/static/private` and `$env/dynamic/private` modules, which are covered in the modules section, can only be imported into modules that only run on the server, such as `hooks.server.js` or `+page.server.js`.

## 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`

# Snapshots

[✎ Edit this page on GitHub](#)

Ephemeral DOM state – like scroll positions on sidebars, the content of `<input>` elements and so on – is discarded when you navigate from one page to another.

For example, if the user fills out a form but clicks a link before submitting, then hits the browser's back button, the values they filled in will be lost. In cases where it's valuable to preserve that input, you can take a *snapshot* of DOM state, which can then be restored if the user navigates back.

To do this, export a `snapshot` object with `capture` and `restore` methods from a `+page.svelte` or `+layout.svelte`:

`+page.svelte`



```
<script>
  let comment = '';

  /** @type {import('./$types').Snapshot<string>} */
  export const snapshot = {
    capture: () => comment,
    restore: (value) => comment = value
  };
</script>

<form method="POST">
  <label for="comment">Comment</label>
  <textarea id="comment" bind:value={comment} />
  <button>Post comment</button>
</form>
```

When you navigate away from this page, the `capture` function is called immediately before the page updates, and the returned value is associated with the current entry in the browser's history stack. If you navigate back, the `restore` function is called with the stored value as soon as the page is updated.

The data must be serializable as JSON so that it can be persisted to `sessionStorage`. This allows the state to be restored when the page is reloaded, or when the user navigates back from a different site.

## NOTE

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`.

# Shallow routing

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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



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

# Packaging

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



You can use SvelteKit to build apps as well as component libraries, using the `@sveltejs/package` package ( `npm create svelte` has an option to set this up for you).

When you're creating an app, the contents of `src/routes` is the public-facing stuff; `src/lib` contains your app's internal library.

A component library has the exact same structure as a SvelteKit app, except that `src/lib` is the public-facing bit, and your root `package.json` is used to publish the package. `src/routes` might be a documentation or demo site that accompanies the library, or it might just be a sandbox you use during development.

Running the `svelte-package` command from `@sveltejs/package` will take the contents of `src/lib` and generate a `dist` directory (which can be configured) containing the following:

All the files in `src/lib`. Svelte components will be preprocessed, TypeScript files will be transpiled to JavaScript.

Type definitions ( `d.ts` files) which are generated for Svelte, JavaScript and TypeScript files. You need to install `typescript >= 4.0.0` for this. Type definitions are placed next to their implementation, hand-written `d.ts` files are copied over as is. You can disable generation, but we strongly recommend against it – people using your library might use TypeScript, for which they require these type definition files.

### NOTE

`@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.



# Performance

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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



Kit - Lighthouse, Network, and Performance devtools

Firefox - Network and Performance devtools



# Images

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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



# Accessibility

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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.

# SEO

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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.

#### NOTE

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.

# Configuration

[✎ Edit this page on GitHub](#)

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;
```

```
interface Config {...}
```

```
compilerOptions?: CompileOptions;
```

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;
```



# Command Line Interface

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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.

PREVIOUS

[Configuration](#)

NEXT

[Modules](#)

# Modules

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



SvelteKit makes a number of modules available to your application.

## `$app/environment`

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



### `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;
```

# Types

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## Public types

The following types can be imported from `@sveltejs/kit`:

### 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;
```

# Frequently asked questions

 [Edit this page on GitHub](#)

## ON THIS PAGE



## Other resources

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

## What can I make with SvelteKit?

SvelteKit can be used to create most kinds of applications. Out of the box, SvelteKit supports many features including:

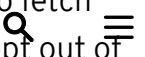
- Dynamic page content with load functions and API routes.
- SEO-friendly dynamic content with server-side rendering (SSR).
- User-friendly progressively-enhanced interactive pages with SSR and Form Actions.
- Static pages with prerendering.

SvelteKit can also be deployed to a wide spectrum of hosted architectures via adapters. In cases where SSR is used (or server-side logic is added without prerendering), those functions will be adapted to the target backend. Some examples include:

- Self-hosted dynamic web applications with a Node.js backend.
- Serverless web applications with backend loaders and APIs deployed as remote functions. See [zero-config deployments](#) for popular deployment options.
- Static pre-rendered sites such as a blog or multi-page site hosted on a CDN or static host. Statically-generated sites are shipped without a backend.
- Single-page Applications (SPAs) with client-side routing and rendering for API-driven dynamic content. SPAs are shipped without a backend and are not server-rendered. This option is commonly chosen when bundling SvelteKit with an app written in PHP, .Net, Java, C, Golang, Rust, etc.



A mix of the above; some routes can be static, and some routes can use backend functions to fetch dynamic information. This can be configured with page options that includes the option to opt out of SSR.



# Integrations

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## vitePreprocess

Including vitePreprocess in your project will allow you to use the various flavors of JS and CSS that Vite supports: TypeScript, 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()]
};
```



## Adders

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

CSS - Tailwind, Bootstrap, Bulma

database - Drizzle ORM

markdown - mdsvex

Storybook

## Directory

See [sveltesociety.dev](https://sveltesociety.dev) for a full listing of packages and templates available for use with Svelte and SvelteKit.



# Breakpoint Debugging

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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": [  
    {
```



# Migrating to SvelteKit v2

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



Upgrading from SvelteKit version 1 to version 2 should be mostly seamless. There are a few breaking changes to note, which are listed here. You can use `npx svelte-migrate@latest 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`.

## path is required when setting cookies

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



# Migrating from Sapper

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



SvelteKit is the successor to Sapper and shares many elements of its design.

If you have an existing Sapper app that you plan to migrate to SvelteKit, there are a number of changes you will need to make. You may find it helpful to view some examples while migrating.

## package.json

### type: "module"

Add `"type": "module"` to your `package.json`. You can do this step separately from the rest as part of an incremental migration if you are using Sapper 0.29.3 or newer.

## dependencies


Remove `polka` or `express`, if you're using one of those, and any middleware such as `sirv` or `compression`.

## devDependencies

Remove `sapper` from your `devDependencies` and replace it with `@sveltejs/kit` and whichever adapter you plan to use (see next section).

## scripts

Any scripts that reference `sapper` should be updated:

 `sapper build` should become `vite build` using the Node adapter  
`sapper export` should become `vite build` using the static adapter



# Additional resources

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## FAQs

Please see the SvelteKit FAQ for solutions to common issues and helpful tips and tricks.

The Svelte FAQ and vite-plugin-svelte FAQ may also be helpful for questions deriving from those libraries.

## Examples

We've written and published a few different SvelteKit sites as examples:

[sveltejs/realworld](#) contains an example blog site

[A HackerNews clone](#)

[kit.svelte.dev](#)

[svelte.dev](#)

SvelteKit users have also published plenty of examples on GitHub, under the [#sveltekit](#) and [#sveltekit-template](#) topics, as well as on the Svelte Society site. Note that these have not been vetted by the maintainers and may not be up to date.

## Support

You can ask for help on Discord and StackOverflow. Please first search for information related to your issue in the FAQ, Google or another search engine, issue tracker, and Discord chat history in order to be respectful of others' time. There are many more people asking questions than answering them, so this will help in allowing the community to grow in a scalable fashion.

# Glossary

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



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

## CSR

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

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




## Hydration

Svelte components store some state and update the DOM when the state is updated. When fetching data during SSR, by default SvelteKit will store this data and transmit it to the client along with the server-rendered HTML. The components can then be initialized on the client with that data without having to call the same API endpoints again. Svelte will then check that the DOM is in the expected state and attach event listeners in a process called hydration. Once the components are fully hydrated, they can react to changes to their properties just like any newly created Svelte component.

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

## Prerendering

Prerendering means computing the contents of a page at build time and saving the HTML for display.

 This approach has the same benefits as traditional server-rendered pages, but avoids recomputing the page for each visitor and so scales nearly for free as the number of visitors increases. The tradeoff is that  

# Introduction

[✎ Edit this page on GitHub](#)

## ON THIS PAGE



## Before we begin

### NOTE

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

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

## What is SvelteKit?

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

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

## What is Svelte?

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

## SvelteKit vs Svelte



KIT | Docs



Svelte renders UI components. You can compose these components and render an entire page with just