SVELTEKIT • CORE CONCEPTS

Page options

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

`+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 `<a>` elements that point to other pages that are candidates for prerendering — because of this, you generally don't need to specify which pages should be accessed. If you *do* need to specify which pages should be accessed by the prerenderer, you can do so with `config.kit.prerender.entries`, or by exporting an `entries` function from your dynamic route.

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

Prerendering server routes

Unlike the other page options, `prerender` also applies to `+server.js` files. These files are *not* affected by layouts, but will inherit default values from the pages that fetch data from

```
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 prerenderable, any two users hitting it directly must get the same content from the server.

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

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

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

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

Route conflicts

Because prerendering writes to the filesystem, it isn't possible to have two endpoints that would cause a directory and a file to have the same name. For example

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 route 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

Docs

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.ts

JS TS

```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

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.

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 `csr` and `ssr` 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.

```
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

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.ts JS TS

```ts
import type { Config } from 'some-adapter';

export const config: Config = {
  runtime: 'edge'
};
```

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

src/routes/+layout.js

```js
export const config = {
  runtime: 'edge',
  regions: 'all',
  foo: {
    bar: true
  }
}
```

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

src/routes/+page.js

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

...which results in the config value `{ runtime: 'edge', regions: ['us1', 'us2'], foo: {`

Tutorial: Page options

Edit this page on GitHub

PREVIOUS
Form actions

NEXT
State management