SVELTEKIT • CORE CONCEPTS

Routing

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`

> 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

on the server (SSR) for the initial request and in the browser (CSR) for subsequent navigation.

src/routes/+page.svelte

```
<h1>Hello and welcome to my site!</h1>
<a href="/about">About my site</a>
```

src/routes/about/+page.svelte

```
<h1>About this site</h1>
<p>TODO...</p>
<a href="/">Home</a>
```

Pages can receive data from `load` functions via the `data` prop.

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

JS TS

```
<script lang="ts">
  import type { PageData } from './$types';

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

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

Legacy mode

show all

Note that SvelteKit uses `<a>` elements to navigate between routes, rather than a framework-specific `<Link>` component.

+page.js

Often, a page will need to load some data before it can be rendered. For this, we add a `+page.js` module that exports a `load` function:

```
import type { PageLoad } from './$types';

export const load: PageLoad = ({ params }) => {
  if (params.slug === 'hello-world') {
    return {
      title: 'Hello world!',
      content: 'Welcome to our blog. Lorem ipsum dolor sit amet...'
    };
  }

  error(404, 'Not found');
};
```

This function runs alongside `+page.svelte`, which means it runs on the server during server-side rendering and in the browser during client-side navigation. See `load` for full details of the API.

As well as `load`, `+page.js` can export values that configure the page's behaviour:

- `export const prerender = true` or `false` or `'auto'`
- `export const ssr = true` or `false`
- `export const csr = true` or `false`

You can find more information about these in page options.

+page.server.js

If your `load` function can only run on the server — for example, if it needs to fetch data from a database or you need to access private environment variables like API keys — then you can rename `+page.js` to `+page.server.js` and change the `PageLoad` type to `PageServerLoad`.

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

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

```
    return post;
  }

  error(404, 'Not found');
};
```

During client-side navigation, SvelteKit will load this data from the server, which means that the returned value must be serializable using devalue. See `load` for full details of the API.

Like `+page.js`, `+page.server.js` can export page options — `prerender`, `ssr` and `csr`.

A `+page.server.js` file can also export *actions*. If `load` lets you read data from the server, `actions` let you write data *to* the server using the `<form>` element. To learn how to use them, see the form actions section.

+error

If an error occurs during `load`, SvelteKit will render a default error page. You can customise this error page on a per-route basis by adding an `+error.svelte` file:

src/routes/blog/[slug]/+error.svelte

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

<h1>{$page.status}: {$page.error.message}</h1>
```

SvelteKit will 'walk up the tree' looking for the closest error boundary — if the file above didn't exist it would try `src/routes/blog/+error.svelte` and then `src/routes/+error.svelte` before rendering the default error page. If *that* fails (or if the error was thrown from the `load` function of the root `+layout`, which sits 'above' the root `+error`), SvelteKit will bail out and render a static fallback error page, which you can boundary in the tree is an `+error.svelte` file *above* that layout (not next to it).

If no route can be found (404), `src/routes/+error.svelte` (or the default error page, if that file does not exist) will be used.

> `+error.svelte` is *not* used when an error occurs inside `handle` or a +server.js request handler.

You can read more about error handling here.

+layout

So far, we've treated pages as entirely standalone components — upon navigation, the existing `+page.svelte` component will be destroyed, and a new one will take its place.

But in many apps, there are elements that should be visible on *every* page, such as top-level navigation or a footer. Instead of repeating them in every `+page.svelte` , we can put them in *layouts*.

+layout.svelte

To create a layout that applies to every page, make a file called `src/routes/+layout.svelte` . The default layout (the one that SvelteKit uses if you don't bring your own) looks like this...

```
<script>
  let { children } = $props();
</script>

{@render children()}
```

...but we can add whatever markup, styles and behaviour we want. The only requirement is that the component includes a `@render` tag for the page content. For example, let's add a

```
<script>
  let { children } = $props();
</script>

<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/settings">Settings</a>
</nav>

{@render children()}
```

If we create pages for `/`, `/about` and `/settings` ...

src/routes/+page.svelte

```
<h1>Home</h1>
```

src/routes/about/+page.svelte

```
<h1>About</h1>
```

src/routes/settings/+page.svelte

```
<h1>Settings</h1>
```

...the nav will always be visible, and clicking between the three pages will only result in the `<h1>` being replaced.

Layouts can be *nested*. Suppose we don't just have a single `/settings` page, but instead have nested pages like `/settings/profile` and `/settings/notifications` with a shared submenu (for a real-life example, see github.com/settings).

We can create a layout that only applies to pages below `/settings` (while inheriting the root layout with the top-level nav):

src/routes/settings/+layout.svelte JS TS

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

<h1>Settings</h1>

<div class="submenu">
  {#each data.sections as section}
    <a href="/settings/{section.slug}">{section.title}</a>
  {/each}
</div>

{@render children()}
```

You can see how `data` is populated by looking at the `+layout.js` example in the next section just below.

By default, each layout inherits the layout above it. Sometimes that isn't what you want - in this case, advanced layouts can help you.

+layout.js

Just like `+page.svelte` loading data from `+page.js` , your `+layout.svelte` component can get data from a `load` function in `+layout.js` .

src/routes/settings/+layout.ts

JS TS

```
import type { LayoutLoad } from './$types';

export const load: LayoutLoad = () => {
  return {
    sections: [
      { slug: 'profile', title: 'Profile' },
      { slug: 'notifications', title: 'Notifications' }
    ]
  };
};
```

If a `+layout.js` exports page options — `prerender` , `ssr` and `csr` — they will be used as src/routes/settings/profile/+page.svelte

JS TS

```
<script lang="ts">
  import type { PageData } from './$types';

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

  console.log(data.sections); // [{ slug: 'profile', title: 'Profile' }, ...]
</script>
```

> Often, layout data is unchanged when navigating between pages. SvelteKit will intelligently rerun `load` functions when necessary.

+layout.server.js

To run your layout's `load` function on the server, move it to `+layout.server.js`, and change the `LayoutLoad` type to `LayoutServerLoad`.

Like `+layout.js`, `+layout.server.js` can export page options — `prerender`, `ssr` and `csr`.

+server

As well as pages, you can define routes with a `+server.js` file (sometimes referred to as an 'API route' or an 'endpoint'), which gives you full control over the response. Your `+server.js` file exports functions corresponding to HTTP verbs like `GET`, `POST`, `PATCH`, `PUT`, `DELETE`, `OPTIONS`, and `HEAD` that take a `RequestEvent` argument and return a `Response` object.

For example we could create an `/api/random-number` route with a `GET` handler:

src/routes/api/random-number/+server.ts

JS TS

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

```
  const max = Number(url.searchParams.get('max') ?? '1');

  const d = max - min;

  if (isNaN(d) || d < 0) {
    error(400, 'min and max must be numbers, and min must be less than max');
  }

  const random = min + Math.random() * d;

  return new Response(String(random));
};
```

The first argument to `Response` can be a `ReadableStream`, making it possible to stream large amounts of data or create server-sent events (unless deploying to platforms that buffer responses, like AWS Lambda).

You can use the `error`, `redirect` and `json` methods from `@sveltejs/kit` for convenience (but you don't have to).

If an error is thrown (either `error(...)` or an unexpected error), the response will be a JSON representation of the error or a fallback error page — which can be customised via `src/error.html` — depending on the `Accept` header. The `+error.svelte` component will *not* be rendered in this case. You can read more about error handling here.

> When creating an `OPTIONS` handler, note that Vite will inject `Access-Control-Allow-Origin` and `Access-Control-Allow-Methods` headers — these will not be present in production unless you add them.

Receiving data

By exporting `POST` / `PUT` / `PATCH` / `DELETE` / `OPTIONS` / `HEAD` handlers, `+server.js` files can be used to create a complete API:

src/routes/add/+page.svelte

```
  async function add() {
    const response = await fetch('/api/add', {
      method: 'POST',
      body: JSON.stringify({ a, b }),
      headers: {
        'content-type': 'application/json'
      }
    });

    total = await response.json();
  }
</script>

<input type="number" bind:value={a}> +
<input type="number" bind:value={b}> =
{total}

<button on:click={add}>Calculate</button>
```

src/routes/api/add/+server.ts

JS TS

```
import { json } from '@sveltejs/kit';
import type { RequestHandler } from './$types';

export const POST: RequestHandler = async ({ request }) => {
  const { a, b } = await request.json();
  return json(a + b);
};
```

> In general, form actions are a better way to submit data from the browser to the server.

> If a `GET` handler is exported, a `HEAD` request will return the `content-length` of the `GET` handler's response body.

Fallback method handler

Exporting the `fallback` handler will match any unhandled request methods, including

```
import { json, text } from '@sveltejs/kit';
import type { RequestHandler } from './$types';

export async function POST({ request }) {
  const { a, b } = await request.json();
  return json(a + b);
}

// This handler will respond to PUT, PATCH, DELETE, etc.
export const fallback: RequestHandler = async ({ request }) => {
  return text(`I caught your ${request.method} request!`);
};
```

> For `HEAD` requests, the `GET` handler takes precedence over the `fallback` handler.

Content negotiation

`+server.js` files can be placed in the same directory as `+page` files, allowing the same route to be either a page or an API endpoint. To determine which, SvelteKit applies the following rules:

- `PUT` / `PATCH` / `DELETE` / `OPTIONS` requests are always handled by `+server.js` since they do not apply to pages
- `GET` / `POST` / `HEAD` requests are treated as page requests if the `accept` header prioritises `text/html` (in other words, it’s a browser page request), else they are handled by `+server.js` .
- Responses to `GET` requests will include a `Vary: Accept` header, so that proxies and browsers cache HTML and JSON responses separately.

$types

Throughout the examples above, we’ve been importing types from a `$types.d.ts` file. This is a file SvelteKit creates for you in a hidden directory if you’re using TypeScript (or

`+layout.svelte` file) tells TypeScript that the type of `data` is whatever was returned from `load`:

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

JS TS

```
<script lang="ts">
  import type { PageData } from './$types';

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

In turn, annotating the `load` function with `PageLoad`, `PageServerLoad`, `LayoutLoad` or `LayoutServerLoad` (for `+page.js`, `+page.server.js`, `+layout.js` and `+layout.server.js` respectively) ensures that `params` and the return value are correctly typed.

If you're using VS Code or any IDE that supports the language server protocol and TypeScript plugins then you can omit these types *entirely*! Svelte's IDE tooling will insert the correct types for you, so you'll get type checking without writing them yourself. It also works with our command line tool `svelte-check`.

You can read more about omitting `$types` in our blog post about it.

Other files

Any other files inside a route directory are ignored by SvelteKit. This means you can colocate components and utility modules with the routes that need them.

If components and modules are needed by multiple routes, it's a good idea to put them in `$lib`.

Further reading

- Tutorial: Routing

Docs

Edit this page on GitHub

PREVIOUS
Web standards

NEXT
Loading data