SVELTEKIT • CORE CONCEPTS

Loading data

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.ts

JS TS

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

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

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

JS TS

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

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

Legacy mode show all

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

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

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

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

```ts
import * as db from '$lib/server/database';
import type { PageServerLoad } from './$types';

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

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

Layout data

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

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

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

src/routes/blog/[slug]/+layout.svelte JS TS

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

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

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

<aside>
  <h2>More posts</h2>
  <ul>
    {#each data.posts as post}
      <li>
        <a href="/blog/{post.slug}">
          {post.title}
        </a>
      </li>
    {/each}
  </ul>
</aside>
```

Data returned from layout `load` functions is available to child `+layout.svelte` components and the `+page.svelte` component as well as the layout that it 'belongs' to.

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

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

```
  // we can access `data.posts` because it's returned from
  // the parent layout `load` function
  let index = $derived(data.posts.findIndex(post => post.slug === $page.params.slug));
  let next = $derived(data.posts[index + 1]);
</script>

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

{#if next}
  <p>Next post: <a href="/blog/{next.slug}">{next.title}</a></p>
{/if}
```

> If multiple `load` functions return data with the same key, the last one 'wins' — the result of a layout `load` returning `{ a: 1, b: 2 }` and a page `load` returning `{ b: 3, c: 4 }` would be `{ a: 1, b: 3, c: 4 }`.

$page.data

The `+page.svelte` component, and each `+layout.svelte` component above it, has access to its own data plus all the data from its parents.

In some cases, we might need the opposite — a parent layout might need to access page data or data from a child layout. For example, the root layout might want to access a `title` property returned from a `load` function in `+page.js` or `+page.server.js`. This can be done with `$page.data`:

src/routes/+layout.svelte

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

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

Universal vs server

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

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

`+page.server.js` and `+layout.server.js` files export *server* `load` functions that only run server-side

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

When does which load function run?

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

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

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

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

Input

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

`cookies`, `locals`, `platform` and `request` from `RequestEvent`.

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

Output

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

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

When to use which

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

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

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

src/routes/+page.server.ts JS TS

Docs

```
    serverMessage: 'hello from server load function'
  };
};
```

src/routes/+page.ts JS TS

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

Using URL data

Often the `load` function depends on the URL in one way or another. For this, the `load` function provides you with `url`, `route` and `params`.

url

An instance of `URL`, containing properties like the `origin`, `hostname`, `pathname` and `searchParams` (which contains the parsed query string as a `URLSearchParams` object). `url.hash` cannot be accessed during `load`, since it is unavailable on the server.

> In some environments this is derived from request headers during server-side rendering. If you're using adapter-node, for example, you may need to configure the adapter in order for the URL to be correct.

route

Contains the name of the current route directory, relative to `src/routes`:

```
export const load: PageLoad = ({ route }) => {
  console.log(route.id); // '/a/[b]/[...c]'
};
```

params

`params` is derived from `url.pathname` and `route.id`.

Given a `route.id` of `/a/[b]/[...c]` and a `url.pathname` of `/a/x/y/z`, the `params` object would look like this:

```
{
  "b": "x",
  "c": "y/z"
}
```

Making fetch requests

To get data from an external API or a `+server.js` handler, you can use the provided `fetch` function, which behaves identically to the native `fetch` web API with a few additional features:

- It can be used to make credentialed requests on the server, as it inherits the `cookie` and `authorization` headers for the page request.
- It can make relative requests on the server (ordinarily, `fetch` requires a URL with an origin when used in a server context).
- Internal requests (e.g. for `+server.js` routes) go directly to the handler function when running on the server, without the overhead of an HTTP call.
- During server-side rendering, the response will be captured and inlined into the rendered HTML by hooking into the `text`, `json` and `arrayBuffer` methods of the `Response` object. Note that headers will *not* be serialized, unless explicitly included via

and preventing an additional network request - if you received a warning in your browser console when using the browser `fetch` instead of the `load` `fetch`, this is why.

src/routes/items/[id]/+page.ts JS TS

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

export const load: PageLoad = async ({ fetch, params }) => {
  const res = await fetch(`/api/items/${params.id}`);
  const item = await res.json();

  return { item };
};
```

Cookies

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

src/routes/+layout.server.ts JS TS

```ts
import * as db from '$lib/server/database';
import type { LayoutServerLoad } from './$types';

export const load: LayoutServerLoad = async ({ cookies }) => {
  const sessionid = cookies.get('sessionid');

  return {
    user: await db.getUser(sessionid)
  };
};
```

Cookies will only be passed through the provided `fetch` function if the target host is the same as the SvelteKit application or a more specific subdomain of it.

For example, if SvelteKit is serving my.domain.com:

domain.com WILL NOT receive cookies

sub.my.domain.com WILL receive cookies

Other cookies will not be passed when `credentials: 'include'` is set, because SvelteKit does not know which domain which cookie belongs to (the browser does not pass this information along), so it's not safe to forward any of them. Use the handleFetch hook to work around it.

Headers

Both server and universal `load` functions have access to a `setHeaders` function that, when running on the server, can set headers for the response. (When running in the browser, `setHeaders` has no effect.) This is useful if you want the page to be cached, for example:

src/routes/products/+page.ts

JS TS

```ts
import type { PageLoad } from './$types';
export const load: PageLoad = async ({ fetch, setHeaders }) => {
  const url = `https://cms.example.com/products.json`;
  const response = await fetch(url);

  // Headers are only set during SSR, caching the page's HTML
  // for the same length of time as the underlying data.
  setHeaders({
    age: response.headers.get('age'),
    'cache-control': response.headers.get('cache-control')
  });

  return response.json();
};
```

Setting the same header multiple times (even in separate `load` functions) is an error. You can only set a given header once using the `setHeaders` function. You cannot add a `set-cookie` header with `setHeaders` — use `cookies.set(name, value, options)` instead.

which can be done with `await parent()`:

src/routes/+layout.ts

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

export const load: LayoutLoad = () => {
  return { a: 1 };
};
```

src/routes/abc/+layout.ts

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

export const load: LayoutLoad = async ({ parent }) => {
  const { a } = await parent();
  return { b: a + 1 };
};
```

src/routes/abc/+page.ts

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

export const load: PageLoad = async ({ parent }) => {
  const { a, b } = await parent();
  return { c: a + b };
};
```

src/routes/abc/+page.svelte

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

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

<!-- renders `1 + 2 = 3` -->
<p>{data.a} + {data.b} = {data.c}</p>
```

Notice that the `load` function in `+page.js` receives the merged data from both layout `load`

`+layout.server.js` files.

In `+page.js` or `+layout.js` it will return data from parent `+layout.js` files. However, a missing `+layout.js` is treated as a `({ data }) => data` function, meaning that it will also return data from parent `+layout.server.js` files that are not 'shadowed' by a `+layout.js` file

Take care not to introduce waterfalls when using `await parent()`. Here, for example, `getData(params)` does not depend on the result of calling `parent()`, so we should call it first to avoid a delayed render.

+page.ts

JS TS

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

export const load: PageLoad = async ({ params, parent }) => {
  const parentData = await parent();
  const data = await getData(params);
  const parentData = await parent();

  return {
    ...data,
    meta: { ...parentData.meta, ...data.meta }
  };
};
```

Errors

If an error is thrown during `load`, the nearest `+error.svelte` will be rendered. For *expected* errors, use the `error` helper from `@sveltejs/kit` to specify the HTTP status code and an optional message:

src/routes/admin/+layout.server.ts

JS TS

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

```
  if (!locals.user.isAdmin) {
    error(403, 'not an admin');
  }
};
```

Calling `error(...)` will throw an exception, making it easy to stop execution from inside helper functions.

If an *unexpected* error is thrown, SvelteKit will invoke `handleError` and treat it as a 500 Internal Error.

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

Redirects

To redirect users, use the `redirect` helper from `@sveltejs/kit` to specify the location to which they should be redirected alongside a `3xx` status code. Like `error(...)`, calling `redirect(...)` will throw an exception, making it easy to stop execution from inside helper functions.

src/routes/user/+layout.server.ts JS TS

```
import { redirect } from '@sveltejs/kit';
import type { LayoutServerLoad } from './$types';

export const load: LayoutServerLoad = ({ locals }) => {
  if (!locals.user) {
    redirect(307, '/login');
  }
};
```

> Don't use `redirect()` inside a `try {...}` block, as the redirect will immediately trigger the catch statement.

Docs

`goto` from `$app.navigation`.

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

Streaming with promises

When using a server `load`, promises will be streamed to the browser as they resolve. This is useful if you have slow, non-essential data, since you can start rendering the page before all the data is available:

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

JS TS

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

export const load: PageServerLoad = async ({ params }) => {
  return {
    // make sure the `await` happens at the end, otherwise we
    // can't start loading comments until we've loaded the post
    comments: loadComments(params.slug),
    post: await loadPost(params.slug)
  };
};
```

This is useful for creating skeleton loading states, for example:

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

JS TS

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

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

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

{#await data.comments}
```

```
  {/each}
{:catch error}
  <p>error loading comments: {error.message}</p>
{/await}
```

When streaming data, be careful to handle promise rejections correctly. More specifically, the server could crash with an "unhandled promise rejection" error if a lazy-loaded promise fails before rendering starts (at which point it's caught) and isn't handling the error in some way. When using SvelteKit's `fetch` directly in the `load` function, SvelteKit will handle this case for you. For other promises, it is enough to attach a noop- `catch` to the promise to mark it as handled.

src/routes/+page.server.ts JS TS

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

export const load: PageServerLoad = ({ fetch }) => {
  const ok_manual = Promise.reject();
  ok_manual.catch(() => {});

  return {
    ok_manual,
    ok_fetch: fetch('/fetch/that/could/fail'),
    dangerous_unhandled: Promise.reject()
  };
};
```

> On platforms that do not support streaming, such as AWS Lambda or Firebase, responses will be buffered. This means the page will only render once all promises resolve. If you are using a proxy (e.g. NGINX), make sure it does not buffer responses from the proxied server.
>
> Streaming data will only work when JavaScript is enabled. You should avoid returning promises from a universal `load` function if the page is server rendered, as these are *not* streamed — instead, the promise is recreated when the function reruns in the browser.

streamed.

Parallel loading

When rendering (or navigating to) a page, SvelteKit runs all `load` functions concurrently, avoiding a waterfall of requests. During client-side navigation, the result of calling multiple server `load` functions are grouped into a single response. Once all `load` functions have returned, the page is rendered.

Rerunning load functions

SvelteKit tracks the dependencies of each `load` function to avoid rerunning it unnecessarily during navigation.

For example, given a pair of `load` functions like these...

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

JS TS

```ts
import * as db from '$lib/server/database';
import type { PageServerLoad } from './$types';

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

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

JS TS

```ts
import * as db from '$lib/server/database';
import type { LayoutServerLoad } from './$types';

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

`diet` to `/blog/i-regret-my-choices` because `params.slug` has changed. The one in `+layout.server.js` will not, because the data is still valid. In other words, we won't call `db.getPostSummaries()` a second time.

A `load` function that calls `await parent()` will also rerun if a parent `load` function is rerun.

Dependency tracking does not apply *after* the `load` function has returned — for example, accessing `params.x` inside a nested promise will not cause the function to rerun when `params.x` changes. (Don't worry, you'll get a warning in development if you accidentally do this.) Instead, access the parameter in the main body of your `load` function.

Search parameters are tracked independently from the rest of the url. For example, accessing `event.url.searchParams.get("x")` inside a `load` function will make that `load` function re-run when navigating from `?x=1` to `?x=2`, but not when navigating from `?x=1&y=1` to `?x=1&y=2`.

Untracking dependencies

In rare cases, you may wish to exclude something from the dependency tracking mechanism. You can do this with the provided `untrack` function:

src/routes/+page.ts

JS TS

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

export const load: PageLoad = async ({ untrack, url }) => {
  // Untrack url.pathname so that path changes don't trigger a rerun
  if (untrack(() => url.pathname === '/')) {
    return { message: 'Welcome!' };
  }
};
```

Manual invalidation

`url` to avoid leaking secrets to the client.

A `load` function depends on `url` if it calls `fetch(url)` or `depends(url)`. Note that `url` can be a custom identifier that starts with `[a-z]:`:

src/routes/random-number/+page.ts

JS TS

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

export const load: PageLoad = async ({ fetch, depends }) => {
  // load reruns when `invalidate('https://api.example.com/random-number')` is called...
  const response = await fetch('https://api.example.com/random-number');

  // ...or when `invalidate('app:random')` is called
  depends('app:random');

  return {
    number: await response.json()
  };
};
```

src/routes/random-number/+page.svelte

JS TS

```svelte
<script lang="ts">
  import { invalidate, invalidateAll } from '$app/navigation';
  import type { PageData } from './$types';

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

  function rerunLoadFunction() {
    // any of these will cause the `load` function to rerun
    invalidate('app:random');
    invalidate('https://api.example.com/random-number');
    invalidate(url => url.href.includes('random-number'));
    invalidateAll();
  }
</script>

<p>random number: {data.number}</p>
<button on:click={rerunLoadFunction}>Update random number</button>
```

- It references a property of `params` whose value has changed
- It references a property of `url` (such as `url.pathname` or `url.search`) whose value has changed. Properties in `request.url` are *not* tracked
- It calls `url.searchParams.get(...)`, `url.searchParams.getAll(...)` or `url.searchParams.has(...)` and the parameter in question changes. Accessing other properties of `url.searchParams` will have the same effect as accessing `url.search`.
- It calls `await parent()` and a parent `load` function reran
- A child `load` function calls `await parent()` and is rerunning, and the parent is a server load function
- It declared a dependency on a specific URL via `fetch` (universal load only) or `depends`, and that URL was marked invalid with `invalidate(url)`
- All active `load` functions were forcibly rerun with `invalidateAll()`

`params` and `url` can change in response to a `<a href="..">` link click, a `<form>` interaction, a `goto` invocation, or a `redirect`.

Note that rerunning a `load` function will update the `data` prop inside the corresponding `+layout.svelte` or `+page.svelte`; it does *not* cause the component to be recreated. As a result, internal state is preserved. If this isn't what you want, you can reset whatever you need to reset inside an `afterNavigate` callback, and/or wrap your component in a `{#key ...}` block.

Implications for authentication

A couple features of loading data have important implications for auth checks:

- Layout `load` functions do not run on every request, such as during client side navigation between child routes. (When do load functions rerun?)
- Layout and page `load` functions run concurrently unless `await parent()` is called. If a

There are a few possible strategies to ensure an auth check occurs before protected code.

To prevent data waterfalls and preserve layout `load` caches:

- Use hooks to protect multiple routes before any `load` functions run
- Use auth guards directly in `+page.server.js` `load` functions for route specific protection

Putting an auth guard in `+layout.server.js` requires all child pages to call `await parent()` before protected code. Unless every child page depends on returned data from `await parent()`, the other options will be more performant.

Further reading

- Tutorial: Loading data
- Tutorial: Errors and redirects
- Tutorial: Advanced loading