SVELTEKIT • ADVANCED

# Hooks

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.

> 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 example).

src/hooks.server.ts

JS TS

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

export const handle: Handle = async ({ event, resolve }) => {
  if (event.url.pathname.startsWith('/custom')) {
    return new Response('custom response');
  }

  const response = await resolve(event);
  return response;
};
```

> Requests for static assets — which includes pages that were already prerendered — are *not* handled by SvelteKit.

If unimplemented, defaults to `({ event, resolve }) => resolve(event)`.

## locals

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

src/hooks.server.ts

JS TS

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

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

  const response = await resolve(event);
  response.headers.set('x-custom-header', 'potato');

  return response;
};
```

function.

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

- `transformPageChunk(opts: { html: string, done: boolean }): MaybePromise<string | undefined>` — applies custom transforms to HTML. If `done` is true, it's the final chunk. Chunks are not guaranteed to be well-formed HTML (they could include an element's opening tag but not its closing tag, for example) but they will always be split at sensible boundaries such as `%sveltekit.head%` or layout/page components.
- `filterSerializedResponseHeaders(name: string, value: string): boolean` — determines which headers should be included in serialized responses when a `load` function loads a resource with `fetch`. By default, none will be included.
- `preload(input: { type: 'js' | 'css' | 'font' | 'asset', path: string }): boolean` — determines what files should be added to the `<head>` tag to preload it. The method is called with each file that was found at build time while constructing the code chunks — so if you for example have `import './styles.css` in your `+page.svelte`, `preload` will be called with the resolved path to that CSS file when visiting that page. Note that in dev mode `preload` is *not* called, since it depends on analysis that happens at build time. Preloading can improve performance by downloading assets sooner, but it can also hurt if too much is downloaded unnecessarily. By default, `js` and `css` files will be preloaded. `asset` files are not preloaded at all currently, but we may add this later after evaluating feedback.

src/hooks.server.ts

JS TS

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

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

Docs

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

## handleFetch

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

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

src/hooks.server.ts

JS TS

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

export const handleFetch: HandleFetch = async ({ request, fetch }) => {
  if (request.url.startsWith('https://api.yourapp.com/')) {
    // clone the original request, but change the URL
    request = new Request(
      request.url.replace('https://api.yourapp.com/', 'http://localhost:9999/'),
      request
    );
  }

  return fetch(request);
};
```

### Credentials

subdomain of the app — for example if your app is on `my-domain.com`, and your API is on `api.my-domain.com`, cookies will be included in the request.

If your app and your API are on sibling subdomains — `www.my-domain.com` and `api.my-domain.com` for example — then a cookie belonging to a common parent domain like `my-domain.com` will *not* be included, because SvelteKit has no way to know which domain the cookie belongs to. In these cases you will need to manually include the cookie using `handleFetch`:

src/hooks.server.ts

JS TS

```ts
import type { HandleFetch } from '@sveltejs/kit';
export const handleFetch: HandleFetch = async ({ event, request, fetch }) => {
  if (request.url.startsWith('https://api.my-domain.com/')) {
    request.headers.set('cookie', event.request.headers.get('cookie'));
  }

  return fetch(request);
};
```

## Shared hooks

The following can be added to `src/hooks.server.js` *and* `src/hooks.client.js`:

### handleError

If an unexpected error is thrown during loading or rendering, this function will be called with the `error`, `event`, `status` code and `message`. This allows for two things:

- you can log the error
- you can generate a custom representation of the error that is safe to show to users, omitting sensitive details like messages and stack traces. The returned value, which defaults to `{ message }`, becomes the value of `$page.error`.

average user).

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

src/app.d.ts

```
declare global {
  namespace App {
    interface Error {
      message: string;
      errorId: string;
    }
  }
}

export {};
```

src/hooks.server.ts

JS TS

```
import * as Sentry from '@sentry/sveltekit';
import type { HandleServerError } from '@sveltejs/kit';

Sentry.init({/*...*/})

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

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

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

```
import * as Sentry from '@sentry/sveltekit';
import type { HandleClientError } from '@sveltejs/kit';

Sentry.init({/*...*/})

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

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

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

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

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

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

> Make sure that `handleError` *never* throws an error

## Universal hooks

The following can be added to `src/hooks.js`. Universal hooks run on both server and client (not to be confused with shared hooks, which are environment-specific).

routes. The returned pathname (which defaults to `url.pathname`) is used to select the route and its parameters.

For example, you might have a `src/routes/[[lang]]/about/+page.svelte` page, which should be accessible as `/en/about` or `/de/ueber-uns` or `/fr/a-propos`. You could implement this with `reroute`:

src/hooks.ts

JS TS

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

const translated: Record<string, string> = {
  '/en/about': '/en/about',
  '/de/ueber-uns': '/de/about',
  '/fr/a-propos': '/fr/about',
};

export const reroute: Reroute = ({ url }) => {
  if (url.pathname in translated) {
    return translated[url.pathname];
  }
};
```

The `lang` parameter will be correctly derived from the returned pathname.

Using `reroute` will *not* change the contents of the browser's address bar, or the value of `event.url`.

## Further reading

- Tutorial: Hooks