SVELTEKIT • CORE CONCEPTS

Form actions

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.ts

JS TS

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

export const actions = {
  default: async (event) => {
    // TODO log the user in
  }
} satisfies Actions;
```

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>
  <button>Log in</button>
</form>
```

If someone were to click the button, the browser would send the form data via `POST` request to the server, running the default action.

> Actions always use `POST` requests, since `GET` requests should never have side-effects.

We can also invoke the action from other pages (for example if there's a login widget in the nav in the root layout) by adding the `action` attribute, pointing to the page:

src/routes/+layout.svelte

```
<form method="POST" action="/login">
  <!-- content -->
</form>
```

Named actions

Instead of one `default` action, a page can have as many named actions as it needs:

src/routes/login/+page.server.ts

JS TS

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

export const actions = {
  default: async (event) => {
  login: async (event) => {
    // TODO log the user in
  },
  register: async (event) => {
    // TODO register the user
  }
} satisfies Actions;
```

Docs

```
<form method="POST" action="?/register">
```

src/routes/+layout.svelte

```
<form method="POST" action="/login?/register">
```

As well as the `action` attribute, we can use the `formaction` attribute on a button to `POST` the same form data to a different action than the parent `<form>` :

src/routes/login/+page.svelte

```
<form method="POST" action="?/login">
  <label>
    Email
    <input name="email" type="email">
  </label>
  <label>
    Password
    <input name="password" type="password">
  </label>
  <button>Log in</button>
  <button formaction="?/register">Register</button>
</form>
```

> We can't have default actions next to named actions, because if you POST to a named action without a redirect, the query parameter is persisted in the URL, which means the next default POST would go through the named action from before.

Anatomy of an action

Each action receives a `RequestEvent` object, allowing you to read the data with `request.formData()` . After processing the request (for example, logging the user in by setting a cookie), the action can respond with data that will be available through the `form` property on the corresponding page and through `$page.form` app-wide until the next

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

export const load: PageServerLoad = async ({ cookies }) => {
  const user = await db.getUserFromSession(cookies.get('sessionid'));
  return { user };
};

export const actions = {
  login: async ({ cookies, request }) => {
    const data = await request.formData();
    const email = data.get('email');
    const password = data.get('password');

    const user = await db.getUser(email);
    cookies.set('sessionid', await db.createSession(user), { path: '/' });

    return { success: true };
  },
  register: async (event) => {
    // TODO register the user
  }
} satisfies Actions;
```

src/routes/login/+page.svelte

JS TS

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

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

{#if form?.success}
  <!-- this message is ephemeral; it exists because the page was rendered in
         response to a form submission. it will vanish if the user reloads -->
  <p>Successfully logged in! Welcome back, {data.user.name}</p>
{/if}
```

Legacy mode show all

Docs

— along with the previously submitted form values — back to the user so that they can try again. The `fail` function lets you return an HTTP status code (typically 400 or 422, in the case of validation errors) along with the data. The status code is available through `$page.status` and the data through `form`:

src/routes/login/+page.server.ts

JS TS

```ts
import { fail } from '@sveltejs/kit';
import * as db from '$lib/server/db';
import type { Actions } from './$types';

export const actions = {
  login: async ({ cookies, request }) => {
    const data = await request.formData();
    const email = data.get('email');
    const password = data.get('password');

    if (!email) {
      return fail(400, { email, missing: true });
    }

    const user = await db.getUser(email);

    if (!user || user.password !== db.hash(password)) {
      return fail(400, { email, incorrect: true });
    }

    cookies.set('sessionid', await db.createSession(user), { path: '/' });

    return { success: true };
  },
  register: async (event) => {
    // TODO register the user
  }
} satisfies Actions;
```

Note that as a precaution, we only return the email back to the page — not the password.

src/routes/login/+page.svelte

```
    Email
    <input name="email" type="email" value={form?.email ?? ''}>
  </label>
  <label>
    Password
    <input name="password" type="password">
  </label>
  <button>Log in</button>
  <button formaction="?/register">Register</button>
</form>
```

The returned data must be serializable as JSON. Beyond that, the structure is entirely up to you. For example, if you had multiple forms on the page, you could distinguish which `<form>` the returned `form` data referred to with an `id` property or similar.

Redirects

Redirects (and errors) work exactly the same as in `load`:

src/routes/login/+page.server.ts

JS TS

```
import { fail, redirect } from '@sveltejs/kit';
import * as db from '$lib/server/db';
import type { Actions } from './$types';

export const actions = {
  login: async ({ cookies, request, url }) => {
    const data = await request.formData();
    const email = data.get('email');
    const password = data.get('password');

    const user = await db.getUser(email);
    if (!user) {
      return fail(400, { email, missing: true });
    }

    if (user.password !== db.hash(password)) {
      return fail(400, { email, incorrect: true });
    }
```

```
      redirect(303, url.searchParams.get('redirectTo'));
    }

    return { success: true };
  },
  register: async (event) => {
    // TODO register the user
  }
} satisfies Actions;
```

Loading data

After an action runs, the page will be re-rendered (unless a redirect or an unexpected error occurs), with the action's return value available to the page as the `form` prop. This means that your page's `load` functions will run after the action completes.

Note that `handle` runs before the action is invoked, and does not rerun before the `load` functions. This means that if, for example, you use `handle` to populate `event.locals` based on a cookie, you must update `event.locals` when you set or delete the cookie in an action:

src/hooks.server.ts JS TS

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

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

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

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

export const load: PageServerLoad = (event) => {
  return {
    user: event.locals.user
  logout: async (event) => {
    event.cookies.delete('sessionid', { path: '/' });
    event.locals.user = null;
  }
} satisfies Actions;
```

Progressive enhancement

In the preceding sections we built a `/login` action that works without client-side JavaScript — not a `fetch` in sight. That's great, but when JavaScript *is* available we can progressively enhance our form interactions to provide a better user experience.

use:enhance

The easiest way to progressively enhance a form is to add the `use:enhance` action:

src/routes/login/+page.svelte

JS TS

```
<script lang="ts">
  import { enhance } from '$app/forms';
  import type { ActionData } from './$types';

  let { form }: { form: ActionData } = $props();
</script>

<form method="POST" use:enhance>
```

> `use:enhance` can only be used with forms that have `method="POST"`. It will not work with `method="GET"`, which is the default for forms without a specified method. Attempting to use `use:enhance` on forms without `method="POST"` will result in an error.

> Yes, it's a little confusing that the `enhance` action and `<form action>` are both called 'action'. These docs are action-packed. Sorry.

Docs

without the full-page reloads. It will:

- update the `form` property, `$page.form` and `$page.status` on a successful or invalid response, but only if the action is on the same page you're submitting from. For example, if your form looks like `<form action="/somewhere/else" ..>`, `form` and `$page` will *not* be updated. This is because in the native form submission case you would be redirected to the page the action is on. If you want to have them updated either way, use `applyAction`
- reset the `<form>` element
- invalidate all data using `invalidateAll` on a successful response
- call `goto` on a redirect response
- render the nearest `+error` boundary if an error occurs
- reset focus to the appropriate element

Customising use:enhance

To customise the behaviour, you can provide a `SubmitFunction` that runs immediately before the form is submitted, and (optionally) returns a callback that runs with the `ActionResult`. Note that if you return a callback, the default behavior mentioned above is not triggered. To get it back, call `update`.

```
<form
  method="POST"
  use:enhance={({ formElement, formData, action, cancel, submitter }) => {
    // `formElement` is this `<form>` element
    // `formData` is its `FormData` object that's about to be submitted
    // `action` is the URL to which the form is posted
    // calling `cancel()` will prevent the submission
    // `submitter` is the `HTMLElement` that caused the form to be submitted

    return async ({ result, update }) => {
      // `result` is an `ActionResult` object
      // `update` is a function which triggers the default logic that would be triggered
```

You can use these functions to show and hide loading UI, and so on.

If you return a callback, you may need to reproduce part of the default `use:enhance` behaviour, but without invalidating all data on a successful response. You can do so with `applyAction`:

src/routes/login/+page.svelte

JS TS

```svelte
<script lang="ts">
  import { enhance, applyAction } from '$app/forms';
  import type { ActionData } from './$types';

  let { form }: { form: ActionData } = $props();
</script>

<form
  method="POST"
  use:enhance={({ formElement, formData, action, cancel }) => {
    return async ({ result }) => {
      // `result` is an `ActionResult` object
      if (result.type === 'redirect') {
        goto(result.location);
      } else {
        await applyAction(result);
      }
    };
  }}
>
```

The behaviour of `applyAction(result)` depends on `result.type`:

- `success`, `failure` — sets `$page.status` to `result.status` and updates `form` and `$page.form` to `result.data` (regardless of where you are submitting from, in contrast to `update` from `enhance`)
- `redirect` — calls `goto(result.location, { invalidateAll: true })`

Docs

Custom event listener

We can also implement progressive enhancement ourselves, without `use:enhance`, with a normal event listener on the `<form>`:

src/routes/login/+page.svelte

JS TS

```
<script lang="ts">
  import { invalidateAll, goto } from '$app/navigation';
  import { applyAction, deserialize } from '$app/forms';
  import type { ActionData } from './$types';
  import type { ActionResult } from '@sveltejs/kit';

  let { form }: { form: ActionData } = $props();

  async function handleSubmit(event: { currentTarget: EventTarget & HTMLFormElement}) {
    const data = new FormData(event.currentTarget);

    const response = await fetch(event.currentTarget.action, {
      method: 'POST',
      body: data
    });

    const result: ActionResult = deserialize(await response.text());

    if (result.type === 'success') {
      // rerun all `load` functions, following the successful update
      await invalidateAll();
    }

    applyAction(result);
  }
</script>

<form method="POST" on:submit|preventDefault={handleSubmit}>
  <!-- content -->
</form>
```

Note that you need to `deserialize` the response before processing it further using the

If you have a `+server.js` alongside your `+page.server.js`, `fetch` requests will be routed there by default. To `POST` to an action in `+page.server.js` instead, use the custom `x-sveltekit-action` header:

```
const response = await fetch(this.action, {
  method: 'POST',
  body: data,
  headers: {
    'x-sveltekit-action': 'true'
  }
});
```

Alternatives

Form actions are the preferred way to send data to the server, since they can be progressively enhanced, but you can also use `+server.js` files to expose (for example) a JSON API. Here's how such an interaction could look like:

src/routes/send-message/+page.svelte

```
<script>
  function rerun() {
    fetch('/api/ci', {
      method: 'POST'
    });
  }
</script>

<button on:click={rerun}>Rerun CI</button>
```

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

JS TS

```
import type { RequestHandler } from './$types';
export const POST: RequestHandler = () => {
  // do something
};
```

Docs

As we've seen, to invoke a form action you must use `method="POST"` .

Some forms don't need to `POST` data to the server — search inputs, for example. For these you can use `method="GET"` (or, equivalently, no `method` at all), and SvelteKit will treat them like `<a>` elements, using the client-side router instead of a full page navigation:

```
<form action="/search">
  <label>
    Search
    <input name="q">
  </label>
</form>
```

Submitting this form will navigate to `/search?q=...` and invoke your load function but will not invoke an action. As with `<a>` elements, you can set the `data-sveltekit-reload` , `data-sveltekit-replacestate` , `data-sveltekit-keepfocus` and `data-sveltekit-noscroll` attributes on the `<form>` to control the router's behaviour.

Further reading

- Tutorial: Forms

Edit this page on GitHub

PREVIOUS
Loading data

NEXT
Page options