SVELTEKIT • CORE CONCEPTS

State management

ON THIS PAGE

If you're used to building client-only apps, state management in an app that spans server and client might seem intimidating. This section provides tips for avoiding some common gotchas.

Avoid shared state on the server

Browsers are *stateful* — state is stored in memory as the user interacts with the application. Servers, on the other hand, are *stateless* — the content of the response is determined entirely by the content of the request.

Conceptually, that is. In reality, servers are often long-lived and shared by multiple users. For that reason it's important not to store data in shared variables. For example, consider this code:

+page.server.ts

JS TS

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

export const load: PageServerLoad = () => {
  return { user };
};

export const actions = {
  default: async ({ request }) => {
    const data = await request.formData();
  }
} satisfies Actions
```

The `user` variable is shared by everyone who connects to this server. If Alice submitted an embarrassing secret, and Bob visited the page after her, Bob would know Alice's secret. In addition, when Alice returns to the site later in the day, the server may have restarted, losing her data.

Instead, you should *authenticate* the user using `cookies` and persist the data to a database.

No side-effects in load

For the same reason, your `load` functions should be *pure* — no side-effects (except maybe the occasional `console.log(...)`). For example, you might be tempted to write to a store inside a `load` function so that you can use the store value in your components:

+page.ts JS TS

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

export const load: PageLoad = async ({ fetch }) => {
  const response = await fetch('/api/user');

  // NEVER DO THIS!
  user.set(await response.json());
};
```

As with the previous example, this puts one user's information in a place that is shared by *all* users. Instead, just return the data...

+page.ts JS TS

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

```
    user: await response.json()
  };
};
```

...and pass it around to the components that need it, or use `$page.data` .

If you're not using SSR, then there's no risk of accidentally exposing one user's data to another. But you should still avoid side-effects in your `load` functions — your application will be much easier to reason about without them.

Using stores with context

You might wonder how we're able to use `$page.data` and other app stores if we can't use our own stores. The answer is that app stores on the server use Svelte's context API — the store is attached to the component tree with `setContext` , and when you subscribe you retrieve it with `getContext` . We can do the same thing with our own stores:

src/routes/+layout.svelte

JS TS

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

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

  // Create a store and update it when necessary...
  const user = writable(data.user);
  $effect.pre(() => {
    user.set(data.user);
  });

  // ...and add it to the context for child components to access
  setContext('user', user);
</script>
```

Docs

```
  import { getContext } from 'svelte';

  // Retrieve user store from context
  const user = getContext('user');
</script>

<p>Welcome {$user.name}</p>
```

Updating the value of a context-based store in deeper-level pages or components while the page is being rendered via SSR will not affect the value in the parent component because it has already been rendered by the time the store value is updated. In contrast, on the client (when CSR is enabled, which is the default) the value will be propagated and components, pages, and layouts higher in the hierarchy will react to the new value. Therefore, to avoid values 'flashing' during state updates during hydration, it is generally recommended to pass state down into components rather than up.

If you're not using SSR (and can guarantee that you won't need to use SSR in future) then you can safely keep state in a shared module, without using the context API.

Component and page state is preserved

When you navigate around your application, SvelteKit reuses existing layout and page components. For example, if you have a route like this…

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

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

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

  // THIS CODE IS BUGGY!
  const wordCount = data.content.split(' ').length;
  const estimatedReadingTime = wordCount / 250;
</script>
```

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

...then navigating from `/blog/my-short-post` to `/blog/my-long-post` won't cause the layout, page and any other components within to be destroyed and recreated. Instead the `data` prop (and by extension `data.title` and `data.content`) will update (as it would with any other Svelte component) and, because the code isn't rerunning, lifecycle methods like `onMount` and `onDestroy` won't rerun and `estimatedReadingTime` won't be recalculated.

Instead, we need to make the value *reactive*:

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

JS TS

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

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

  let wordCount = $state(data.content.split(' ').length);
  let estimatedReadingTime = $derived(wordCount / 250);
</script>
```

> If your code in `onMount` and `onDestroy` has to run again after navigation you can use afterNavigate and beforeNavigate respectively.

Reusing components like this means that things like sidebar scroll state are preserved, and you can easily animate between changing values. In the case that you do need to completely destroy and remount a component on navigation, you can use this pattern:

```
{#key $page.url.pathname}
  <BlogPost title={data.title} content={data.title} />
{/key}
```

rules on a table, URL search parameters (like `?sort=price&order=ascending`) are a good place to put them. You can put them in `<a href="...">` or `<form action="...">` attributes, or set them programmatically via `goto('?key=value')`. They can be accessed inside `load` functions via the `url` parameter, and inside components via `$page.url.searchParams`.

Storing ephemeral state in snapshots

Some UI state, such as 'is the accordion open?', is disposable — if the user navigates away or refreshes the page, it doesn't matter if the state is lost. In some cases, you *do* want the data to persist if the user navigates to a different page and comes back, but storing the state in the URL or in a database would be overkill. For this, SvelteKit provides snapshots, which let you associate component state with a history entry.

Edit this page on GitHub

PREVIOUS
Page options

NEXT
Building your app