SVELTE • RUNTIME

Context

ON THIS PAGE

Most state is component-level state that lives as long as its component lives. There's also section-wide or app-wide state however, which also needs to be handled somehow.

The easiest way to do that is to create global state and just import that.

state.svelte.js

```
export const myGlobalState = $state({
  user: {
    /* ... */
  }
  /* ... */
});
```

App.svelte

```
<script>
  import { myGlobalState } from './state.svelte';
  // ...
</script>
```

This has a few drawbacks though:

- it only safely works when your global state is only used client-side - for example, when you're building a single page application that does not render any of your components on the server. If your state ends up being managed and updated on the server, it could end up being shared between sessions and/or users, causing bugs
- it may give the false impression that certain state is global when in reality it should only

problems.

Setting and getting context

To associate an arbitrary object with the current component, use `setContext`.

```
<script>
  import { setContext } from 'svelte';

  setContext('key', value);
</script>
```

The context is then available to children of the component (including slotted content) with `getContext`.

```
<script>
  import { getContext } from 'svelte';

  const value = getContext('key');
</script>
```

`setContext` and `getContext` solve the above problems:

> the state is not global, it's scoped to the component. That way it's safe to render your components on the server and not leak state
>
> it's clear that the state is not global but rather scoped to a specific component tree and therefore can't be used in other parts of your app

> `setContext` / `getContext` must be called during component initialisation.

Context is not inherently reactive. If you need reactive values in context then you can pass a `$state` object into context, whose properties *will* be reactive.

Docs

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

  let value = $state({ count: 0 });
  setContext('counter', value);
</script>

<button onclick={() => value.count++}>increment</button>
```

Child.svelte

```
<script>
  import { getContext } from 'svelte';

  const value = getContext('counter');
</script>

<p>Count is {value.count}</p>
```

To check whether a given `key` has been set in the context of a parent component, use `hasContext`.

```
<script>
  import { hasContext } from 'svelte';

  if (hasContext('key')) {
    // do something
  }
</script>
```

You can also retrieve the whole context map that belongs to the closest parent component using `getAllContexts`. This is useful, for example, if you programmatically create a component and want to pass the existing context to it.

```
<script>
  import { getAllContexts } from 'svelte';

  const contexts = getAllContexts();
</script>
```

The above methods are very unopinionated about how to use them. When your app grows in scale, it's worthwhile to encapsulate setting and getting the context into functions and properly type them.

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

let userKey = Symbol('user');

export function setUserContext(user: User) {
  setContext(userKey, user);
}

export function getUserContext(): User {
  return getContext(userKey) as User;
}
```

Edit this page on GitHub

PREVIOUS
Stores

NEXT
Lifecycle hooks