SVELTE • RUNTIME

Stores

ON THIS PAGE

A *store* is an object that allows reactive access to a value via a simple *store contract*. The `svelte/store` module contains minimal store implementations which fulfil this contract.

Any time you have a reference to a store, you can access its value inside a component by prefixing it with the `$` character. This causes Svelte to declare the prefixed variable, subscribe to the store at component initialisation and unsubscribe when appropriate.

Assignments to `$`-prefixed variables require that the variable be a writable store, and will result in a call to the store's `.set` method.

Note that the store must be declared at the top level of the component — not inside an `if` block or a function, for example.

Local variables (that do not represent store values) must *not* have a `$` prefix.

```
<script>
  import { writable } from 'svelte/store';

  const count = writable(0);
  console.log($count); // logs 0

  count.set(1);
  console.log($count); // logs 1

  $count = 2;
  console.log($count); // logs 2
</script>
```

or extracting logic. With runes, these use cases have greatly diminished.

> when extracting logic, it's better to take advantage of runes' universal reactivity: You can use runes outside the top level of components and even place them into JavaScript or TypeScript files (using a `.svelte.js` or `.svelte.ts` file ending)
>
> when creating shared state, you can create a `$state` object containing the values you need and then manipulate said state

Stores are still a good solution when you have complex asynchronous data streams or it's important to have more manual control over updating values or listening to changes. If you're familiar with RxJs and want to reuse that knowledge, the `$` also comes in handy for you.

svelte/store

The `svelte/store` module contains a minimal store implementation which fulfil the store contract. It provides methods for creating stores that you can update from the outside, stores you can only update from the inside, and for combining and deriving stores.

writable

Function that creates a store which has values that can be set from 'outside' components. It gets created as an object with additional `set` and `update` methods.

`set` is a method that takes one argument which is the value to be set. The store value gets set to the value of the argument if the store value is not already equal to it.

`update` is a method that takes one argument which is a callback. The callback takes the existing store value as its argument and returns the new value to be set to the store.

store.js

```
import { writable } from 'svelte/store';
```

```
  console.log(value);
}); // logs '0'

count.set(1); // logs '1'

count.update((n) => n + 1); // logs '2'
```

If a function is passed as the second argument, it will be called when the number of subscribers goes from zero to one (but not from one to two, etc). That function will be passed a `set` function which changes the value of the store, and an `update` function which works like the `update` method on the store, taking a callback to calculate the store's new value from its old value. It must return a `stop` function that is called when the subscriber count goes from one to zero.

store.js

```
import { writable } from 'svelte/store';

const count = writable(0, () => {
  console.log('got a subscriber');
  return () => console.log('no more subscribers');
});

count.set(1); // does nothing

const unsubscribe = count.subscribe((value) => {
  console.log(value);
}); // logs 'got a subscriber', then '1'

unsubscribe(); // logs 'no more subscribers'
```

Note that the value of a `writable` is lost when it is destroyed, for example when the page is refreshed. However, you can write your own logic to sync the value to for example the `localStorage`.

readable

Docs

initial value, and the second argument to `readable` is the same as the second argument to `writable`.

```
import { readable } from 'svelte/store';

const time = readable(new Date(), (set) => {
  set(new Date());

  const interval = setInterval(() => {
    set(new Date());
  }, 1000);

  return () => clearInterval(interval);
});

const ticktock = readable('tick', (set, update) => {
  const interval = setInterval(() => {
    update((sound) => (sound === 'tick' ? 'tock' : 'tick'));
  }, 1000);

  return () => clearInterval(interval);
});
```

derived

Derives a store from one or more other stores. The callback runs initially when the first subscriber subscribes and then whenever the store dependencies change.

In the simplest version, `derived` takes a single store, and the callback returns a derived value.

```
import { derived } from 'svelte/store';

const doubled = derived(a, ($a) => $a * 2);
```

The callback can set a value asynchronously by accepting a second argument, `set`, and an optional third argument, `update`, calling either or both of them when appropriate.

derived store before `set` or `update` is first called. If no initial value is specified, the store's initial value will be `undefined`.

```
import { derived } from 'svelte/store';

const delayed = derived(
  a,
  ($a, set) => {
    setTimeout(() => set($a), 1000);
  },
  2000
);

const delayedIncrement = derived(a, ($a, set, update) => {
  set($a);
  setTimeout(() => update((x) => x + 1), 1000);
  // every time $a produces a value, this produces two
  // values, $a immediately and then $a + 1 a second later
});
```

If you return a function from the callback, it will be called when a) the callback runs again, or b) the last subscriber unsubscribes.

```
import { derived } from 'svelte/store';

const tick = derived(
  frequency,
  ($frequency, set) => {
    const interval = setInterval(() => {
      set(Date.now());
    }, 1000 / $frequency);

    return () => {
      clearInterval(interval);
    };
  },
  2000
);
```

store.

```
import { derived } from 'svelte/store';

const summed = derived([a, b], ([$a, $b]) => $a + $b);

const delayed = derived([a, b], ([$a, $b], set) => {
  setTimeout(() => set($a + $b), 1000);
});
```

readonly

This simple helper function makes a store readonly. You can still subscribe to the changes from the original one using this new readable store.

```
import { readonly, writable } from 'svelte/store';

const writableStore = writable(1);
const readableStore = readonly(writableStore);

readableStore.subscribe(console.log);

writableStore.set(2); // console: 2
readableStore.set(2); // ERROR
```

get

Generally, you should read the value of a store by subscribing to it and using the value as it changes over time. Occasionally, you may need to retrieve the value of a store to which you're not subscribed. `get` allows you to do so.

> This works by creating a subscription, reading the value, then unsubscribing. It's therefore not recommended in hot code paths.

```
import { get } from 'svelte/store';
```

```
store = { subscribe: (subscription: (value: any) => void) => (() => void), set?: (val
```

You can create your own stores without relying on `svelte/store`, by implementing the *store contract*:

1. A store must contain a `.subscribe` method, which must accept as its argument a subscription function. This subscription function must be immediately and synchronously called with the store's current value upon calling `.subscribe`. All of a store's active subscription functions must later be synchronously called whenever the store's value changes.
2. The `.subscribe` method must return an unsubscribe function. Calling an unsubscribe function must stop its subscription, and its corresponding subscription function must not be called again by the store.
3. A store may *optionally* contain a `.set` method, which must accept as its argument a new value for the store, and which synchronously calls all of the store's active subscription functions. Such a store is called a *writable store*.

For interoperability with RxJS Observables, the `.subscribe` method is also allowed to return an object with an `.unsubscribe` method, rather than return the unsubscription function directly. Note however that unless `.subscribe` synchronously calls the subscription (which is not required by the Observable spec), Svelte will see the value of the store as `undefined` until it does.

Edit this page on GitHub

PREVIOUS
<svelte:options>

NEXT
Context