SVELTE • RUNES

$effect

ON THIS PAGE

Effects are what make your application *do things*. When Svelte runs an effect function, it tracks which pieces of state (and derived state) are accessed (unless accessed inside `untrack`), and re-runs the function when that state later changes.

Most of the effects in a Svelte app are created by Svelte itself — they're the bits that update the text in `<h1>hello {name}!</h1>` when `name` changes, for example.

But you can also create your own effects with the `$effect` rune, which is useful when you need to synchronize an external system (whether that's a library, or a `<canvas>` element, or something across a network) with state inside your Svelte app.

> Avoid overusing `$effect`! When you do too much work in effects, code often becomes difficult to understand and maintain. See when not to use `$effect` to learn about alternative approaches.

Your effects run after the component has been mounted to the DOM, and in a microtask after state changes (demo):

```
<script>
  let size = $state(50);
  let color = $state('#ff3e00');

  let canvas;

  $effect(() => {
    const context = canvas.getContext('2d');
    context.clearRect(0, 0, canvas.width, canvas.height);
  });
</script>

<canvas bind:this={canvas} width="100" height="100" />
```

Re-runs are batched (i.e. changing `color` and `size` in the same moment won't cause two separate runs), and happen after any DOM updates have been applied.

You can place `$effect` anywhere, not just at the top level of a component, as long as it is called during component initialization (or while a parent effect is active). It is then tied to the lifecycle of the component (or parent effect) and will therefore destroy itself when the component unmounts (or the parent effect is destroyed).

You can return a function from `$effect`, which will run immediately before the effect re-runs, and before it is destroyed (demo).

```
<script>
  let count = $state(0);
  let milliseconds = $state(1000);

  $effect(() => {
    // This will be recreated whenever `milliseconds` changes
    const interval = setInterval(() => {
      count += 1;
    }, milliseconds);

    return () => {
      // if a callback is provided, it will run
      // a) immediately before the effect re-runs
      // b) when the component is destroyed
      clearInterval(interval);
    };
  });
</script>

<h1>{count}</h1>

<button onclick={() => (milliseconds *= 2)}>slower</button>
<button onclick={() => (milliseconds /= 2)}>faster</button>
```

`$effect` automatically picks up any reactive values (`$state` , `$derived` , `$props`) that are *synchronously* read inside its function body and registers them as dependencies. When those dependencies change, the `$effect` schedules a rerun.

Values that are read *asynchronously* — after an `await` or inside a `setTimeout` , for example — will not be tracked. Here, the canvas will be repainted when `color` changes, but not when `size` changes (demo):

```
$effect(() => {
  const context = canvas.getContext('2d');
  context.clearRect(0, 0, canvas.width, canvas.height);

  // this will re-run whenever `color` changes...
  context.fillStyle = color;

  setTimeout(() => {
    // ...but not when `size` changes
    context.fillRect(0, 0, size, size);
  }, 0);
});
```

An effect only reruns when the object it reads changes, not when a property inside it changes. (If you want to observe changes *inside* an object at dev time, you can use `$inspect` .)

```
<script>
  let state = $state({ value: 0 });
  let derived = $derived({ value: state.value * 2 });

  // this will run once, because `state` is never reassigned (only mutated)
  $effect(() => {
    state;
  });

  // this will run whenever `state.value` changes...
  $effect(() => {
    state.value;
    derived;
  });
</script>

<button onclick={() => (state.value += 1)}>
  {state.value}
</button>

<p>{state.value} doubled is {derived.value}</p>
```

An effect only depends on the values that it read the last time it ran. If `a` is true, changes to `b` will not cause this effect to rerun:

```
$effect(() => {
  console.log('running');

  if (a || b) {
    console.log('inside if block');
  }
});
```

$effect.pre

In rare cases, you may need to run code *before* the DOM updates. For this we can use the `$effect.pre` rune:

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

  let div = $state();
  let messages = $state([]);

  // ...

  $effect.pre(() => {
    if (!div) return; // not yet mounted
    // autoscroll when new messages are added
    if (div.offsetHeight + div.scrollTop > div.scrollHeight - 20) {
      tick().then(() => {
        div.scrollTo(0, div.scrollHeight);
      });
    }
  });
</script>

<div bind:this={div}>
  {#each messages as message}
    <p>{message}</p>
  {/each}
</div>
```

Apart from the timing, `$effect.pre` works exactly like `$effect`.

$effect.tracking

The `$effect.tracking` rune is an advanced feature that tells you whether or not the code is running inside a tracking context, such as an effect or inside your template (demo):

```
<script>
  console.log('in component setup:', $effect.tracking()); // false

  $effect(() => {
    console.log('in effect:', $effect.tracking()); // true
  });
</script>

<p>in template: {$effect.tracking()}</p> <!-- true -->
```

This allows you to (for example) add things like subscriptions without causing memory leaks, by putting them in child effects. Here's a `readable` function that listens to changes from a callback function as long as it's inside a tracking context:

```
export default function readable<T>(
  initial_value: T,
  start: (callback: (update: (v: T) => T) => T) => () => void
) {
  let value = $state(initial_value);

  let subscribers = 0;
  let stop: null | (() => void) = null;

  return {
    get value() {
      // If in a tracking context ...
      if ($effect.tracking()) {
        $effect(() => {
          // ...and there's no subscribers yet...
          if (subscribers === 0) {
            // ...invoke the function and listen to changes to update state
            stop = start((fn) => (value = fn(value)));
          }

          subscribers++;

          // The return callback is called once a listener unlistens
          return () => {
            tick().then(() => {
              subscribers--;
              // If it was the last subscriber...
              if (subscribers === 0) {
                // ...stop listening to changes
                stop?.();
                stop = null;
              }
            });
          };
        });
      }

      return value;
    }
  };
}
```

Docs

The `$effect.root` rune is an advanced feature that creates a non-tracked scope that doesn't auto-cleanup. This is useful for nested effects that you want to manually control. This rune also allows for the creation of effects outside of the component initialisation phase.

```
<script>
  let count = $state(0);

  const cleanup = $effect.root(() => {
    $effect(() => {
      console.log(count);
    });

    return () => {
      console.log('effect root cleanup');
    };
  });
</script>
```

When not to use $effect

In general, `$effect` is best considered something of an escape hatch — useful for things like analytics and direct DOM manipulation — rather than a tool you should use frequently. In particular, avoid using it to synchronise state. Instead of this...

```
<script>
  let count = $state(0);
  let doubled = $state();

  // don't do this!
  $effect(() => {
    doubled = count * 2;
  });
</script>
```

...do this:

```
  let count = $state(0);
  let doubled = $derived(count * 2);
</script>
```

> For things that are more complicated than a simple expression like `count * 2`, you can also use `$derived.by`.

You might be tempted to do something convoluted with effects to link one value to another. The following example shows two inputs for "money spent" and "money left" that are connected to each other. If you update one, the other should update accordingly. Don't use effects for this (demo):

```
<script>
  let total = 100;
  let spent = $state(0);
  let left = $state(total);

  $effect(() => {
    left = total - spent;
  });

  $effect(() => {
    spent = total - left;
  });
</script>

<label>
  <input type="range" bind:value={spent} max={total} />
  {spent}/{total} spent
</label>

<label>
  <input type="range" bind:value={left} max={total} />
  {left}/{total} left
</label>
```

Instead, use callbacks where possible (demo):

Docs

```
  let total = 100;
  let spent = $state(0);
  let left = $state(total);

  function updateSpent(e) {
    spent = +e.target.value;
    left = total - spent;
  }

  function updateLeft(e) {
    left = +e.target.value;
    spent = total - left;
  }
</script>

<label>
  <input type="range" value={spent} oninput={updateSpent} max={total} />
  {spent}/{total} spent
</label>

<label>
  <input type="range" value={left} oninput={updateLeft} max={total} />
  {left}/{total} left
</label>
```

If you need to use bindings, for whatever reason (for example when you want some kind of "writable `$derived`"), consider using getters and setters to synchronise state (demo):

```
<script>
  let total = 100;
  let spent = $state(0);

  let left = {
    get value() {
      return total - spent;
    },
    set value(v) {
      spent = total - v;
    }
  };
</script>
```

```
</label>

<label>
  <input type="range" bind:value={left.value} max={total} />
  {left.value}/{total} left
</label>
```

If you absolutely have to update `$state` within an effect and run into an infinite loop because you read and write to the same `$state`, use untrack.

Edit this page on GitHub