SVELTE • MISC

Testing

ON THIS PAGE

Testing helps you write and maintain your code and guard against regressions. Testing frameworks help you with that, allowing you to describe assertions or expectations about how your code should behave. Svelte is unopinionated about which testing framework you use — you can write unit tests, integration tests, and end-to-end tests using solutions like Vitest, Jasmine, Cypress and Playwright.

Unit and integration testing using Vitest

Unit tests allow you to test small isolated parts of your code. Integration tests allow you to test parts of your application to see if they work together. If you're using Vite (including via SvelteKit), we recommend using Vitest.

To get started, install Vitest:

```
npm install -D vitest
```

Then adjust your `vite.config.js`:

vite.config.js

```
import { defineConfig } from 'vitest/config';

export default defineConfig({
  // ...
  // Tell Vitest to use the `browser` entry points in `package.json` files, even though i
  resolve: process.env.VITEST
```

```
});
```

> If loading the browser version of all your packages is undesirable, because (for example) you also test backend libraries, you may need to resort to an alias configuration

You can now write unit tests for code inside your `.js/.ts` files:

multiplier.svelte.test.js

```
import { flushSync } from 'svelte';
import { expect, test } from 'vitest';
import { multiplier } from './multiplier.js';

test('Multiplier', () => {
  let double = multiplier(0, 2);

  expect(double.value).toEqual(0);

  double.set(5);

  expect(double.value).toEqual(10);
});
```

Using runes inside your test files

It is possible to use runes inside your test files. First ensure your bundler knows to route the file through the Svelte compiler before running the test by adding `.svelte` to the filename (e.g `multiplier.svelte.test.js`). After that, you can use runes inside your tests.

multiplier.svelte.test.js

```
import { flushSync } from 'svelte';
import { expect, test } from 'vitest';
import { multiplier } from './multiplier.svelte.js';

test('Multiplier', () => {
  let count = $state(0);
```

```
  count = 5;

  expect(double.value).toEqual(10);
});
```

If the code being tested uses effects, you need to wrap the test inside `$effect.root`:

logger.svelte.test.js

```
import { flushSync } from 'svelte';
import { expect, test } from 'vitest';
import { logger } from './logger.svelte.js';

test('Effect', () => {
  const cleanup = $effect.root(() => {
    let count = $state(0);

    // logger uses an $effect to log updates of its input
    let log = logger(() => count);

    // effects normally run after a microtask,
    // use flushSync to execute all pending effects synchronously
    flushSync();
    expect(log.value).toEqual([0]);

    count = 1;
    flushSync();

    expect(log.value).toEqual([0, 1]);
  });

  cleanup();
});
```

Component testing

It is possible to test your components in isolation using Vitest.

> Before writing component tests, think about whether you actually need to test the

To get started, install jsdom (a library that shims DOM APIs):

```
npm install -D jsdom
```

Then adjust your `vite.config.js`:

vite.config.js

```
import { defineConfig } from 'vitest/config';

export default defineConfig({
  plugins: [
    /* ... */
  ],
  test: {
    // If you are testing components client-side, you need to setup a DOM environment.
    // If not all your files should have this environment, you can use a
    // `// @vitest-environment jsdom` comment at the top of the test files instead.
    environment: 'jsdom'
  },
  // Tell Vitest to use the `browser` entry points in `package.json` files, even though i
  resolve: process.env.VITEST
    ? {
        conditions: ['browser']
      }
    : undefined
});
```

After that, you can create a test file in which you import the component to test, interact with it programmatically and write expectations about the results:

component.test.js

```
import { flushSync, mount, unmount } from 'svelte';
import { expect, test } from 'vitest';
import Component from './Component.svelte';
```

```
    props: { initial: 0 }
  });

  expect(document.body.innerHTML).toBe('<button>0</button>');

  // Click the button, then flush the changes so you can synchronously write expectations
  document.body.querySelector('button').click();
  flushSync();

  expect(document.body.innerHTML).toBe('<button>1</button>');

  // Remove the component from the DOM
  unmount(component);
});
```

While the process is very straightforward, it is also low level and somewhat brittle, as the precise structure of your component may change frequently. Tools like @testing-library/svelte can help streamline your tests. The above test could be rewritten like this:

component.test.js

```
import { render, screen } from '@testing-library/svelte';
import userEvent from '@testing-library/user-event';
import { expect, test } from 'vitest';
import Component from './Component.svelte';

test('Component', async () => {
  const user = userEvent.setup();
  render(Component);

  const button = screen.getByRole('button');
  expect(button).toHaveTextContent(0);

  await user.click(button);
  expect(button).toHaveTextContent(1);
});
```

When writing component tests that involve two-way bindings, context or snippet props, it's best to create a wrapper component for your specific test and interact with that.

E2E (short for 'end to end') tests allow you to test your full application through the eyes of the user. This section uses Playwright as an example, but you can also use other solutions like Cypress or NightwatchJS.

To get start with Playwright, either let you guide by their VS Code extension, or install it from the command line using `npm init playwright`. It is also part of the setup CLI when you run `npx sv create`.

After you've done that, you should have a `tests` folder and a Playwright config. You may need to adjust that config to tell Playwright what to do before running the tests - mainly starting your application at a certain port:

playwright.config.js

```
const config = {
  webServer: {
    command: 'npm run build && npm run preview',
    port: 4173
  },
  testDir: 'tests',
  testMatch: /(.+\.)?(test|spec)\.[jt]s/
};

export default config;
```

You can now start writing tests. These are totally unaware of Svelte as a framework, so you mainly interact with the DOM and write assertions.

tests/hello-world.spec.js

```
import { expect, test } from '@playwright/test';

test('home page has expected h1', async ({ page }) => {
  await page.goto('/');
  await expect(page.locator('h1')).toBeVisible();
});
```

Imperative component API

TypeScript