Testing Overview

- Why test?
- JavaScript testing
  - Setup and commands
  - Routing and infrastructure checks
    - Public tooling consumers
  - Folder structure
  - Importing tests
  - Describing tests
  - Setup and teardown methods
  - Expected console calls
  - Mocking dependencies
    - Dependency injection
    - Imported dependencies
  - Testing globals
  - User interactions
  - Integration testing for block UI
  - Snapshot testing
    - TL;DR Broken snapshots
    - What are snapshots?
    - Advantages
    - Disadvantages
    - Use cases
    - Working with snapshots
    - Pain points
    - Best practices
  - Debugging Vitest unit tests
  - Browser Mode failure artifacts
- End-to-end testing

    - Using wp-env
    - Scenario testing
    - Core block testing
    - Flaky tests
- PHP testing
    - Testing Prefixed Functions
        - Writing Tests for Prefixed Functions & Classes
- Performance testing

Gutenberg contains both PHP and JavaScript code, and encourages testing and code style linting for both.

Why test?

Aside from the joy testing will bring to your life, tests are important not only because they help to ensure that our application behaves as it should, but also because they provide concise examples of how to use a piece of code.

Tests are also part of our code base, which means we apply to them the same standards we apply to all our application code.

As with all code, tests have to be maintained. Writing tests for the sake of having a test isn't the goal – rather we should try to strike the right balance between covering expected and unexpected behaviours, speedy execution and code maintenance.

When writing tests consider the following:

- What behaviour(s) are we testing?
- What errors are likely to occur when we run this code?
- Does the test test what we think it is testing? Or are we introducing false positives/negatives?
- Is it readable? Will other contributors be able to understand how our code behaves by looking at its corresponding test?

JavaScript testing

JavaScript unit and integration tests use Vitest. Import `describe`, `test`, `expect`, hooks, and `vi` explicitly from `vitest`. Globals are disabled. Gutenberg's shared setup is internal; external projects

should follow the consumer migration guide.

Setup and commands

Start with the development prerequisites, then run these commands from the repository root:

```
npm ci
npm run build
npm exec --no --workspace @wordpress/unit-tests -- playwright install chromium
```

Chromium is required for the full suite because it includes Browser Mode tests. A local WordPress site is not required for JavaScript unit tests.

```
# Lint, then run the complete Vitest suite once.
npm test

# Run the complete suite once without linting.
npm run test:unit

# Focus on a file or directory, optionally filtering by test name.
npm run test:unit -- packages/escape-html/src/test/index.ts
npm run test:unit -- packages/escape-html/src/test/index.ts -t escapeAttribute

# Watch a focused set and rerun it after edits. Press q to quit.
npm run test:unit:watch -- packages/escape-html/src/test/index.ts

# Update snapshots for matching tests, then review the generated diff.
npm run test:unit:update -- path/to/tests

# Pause a Node or jsdom test worker until a debugger connects.
npm run test:unit:debug -- packages/escape-html/src/test/index.ts
```

Paths filter discovered files; `-t` filters test names. Use `--project=node`, `--project=jsdom`, or `--project=browser` to select an environment. These filters do not change the environment selected by the filename. For example, `npm run test:unit -- --project=browser` runs the Browser suite.

`npm run test:unit:vitest` and its watch/update variants remain compatible aliases. Gutenberg-owned tests run only through Vitest. External projects that keep Jest can use the public `wp-scripts test-unit-jest` adapter and follow the consumer migration guide.

Run `npm run lint` independently for code style checks. ESLint enforces JavaScript rules; `npm run typecheck` checks TypeScript and checked JavaScript, including tests and stories. `npm run build` emits declarations but does not typecheck. Run `npm run test:unit:routing` and `npm run test:unit:conventions` to check discovery, imports, and environment conventions. Configure an editor linting integration for feedback while editing.

Routing and infrastructure checks

`npm run test:unit:routing` compares live Vitest discovery with the repository's test-file patterns. Each discovered test must belong to exactly one project, selected by its filename. The check rejects missing tests, duplicate ownership, per-file environment overrides, and obsolete Jest runner infrastructure. It does not depend on a fixed test count or migration metadata. The required `All` CI check runs it.

`npm run test:unit:conventions` checks explicit Vitest imports, workspace dependencies, the TypeScript test graph, environment conventions, and isolation defaults. Keep the existing Node 24/26 matrix, four Node/jsdom shards per runtime, one Chromium job on Node 24, timezone checks, and Storybook smoke coverage when changing test infrastructure.

`npm run test:unit:vitest:shuffled` shuffles files and tests inside each file. CI uses `GITHUB_RUN_NUMBER` as the seed across all Node/jsdom shards and the Chromium job. Each new workflow run uses a different seed; rerunning that workflow keeps the same seed. Each job logs its seed and a reproduction command. Use the same checkout and Node.js version when reproducing a failure.

Locally, Vitest chooses a seed from the current time unless you pass `--sequence.seed`. Supply the seed from a CI log to reproduce its ordering, retaining the project and shard arguments shown there. For example:

```
npm run test:unit:vitest:shuffled -- --sequence.seed=12345 --project=node --project=jsdom --shard=1/4
npm run test:unit:vitest:shuffled -- --sequence.seed=12345 --project=browser
```

A new seed can expose an existing test-order dependency on an unrelated pull request. When this happens, reproduce the failure with the logged seed and fix the shared state or missing setup and cleanup. Do not push an unrelated change just to get a different seed. If a fix cannot be made promptly, report the failure with its seed, commit, Node.js version, project, and shard, and agree on a temporary quarantine with the maintainers. Keep any quarantine limited to the affected test and link a tracking issue for the fix and removal of the quarantine. Rerun the original failing seed to verify the fix.

Public tooling consumers

Run `npm run test:unit:consumers` to pack the public tooling, install it outside the workspace, and exercise the documented configuration and commands. The check covers Vitest and the maintenance-only Jest adapter independently of Gutenberg's unit-test runner. Preserve the dependencies that these isolated consumers install, including the published WordPress Jest preset and Babel transformer.

- `--vite=<version>` selects a supported Vite version.
- `--browser` includes Chromium and generated CSS coverage.
- `--node=/absolute/path/to/node` selects the consumer runtime.
- `--lockfile=/path/to/package-lock.json` replays an earlier consumer resolution while repacking the changed packages. Also verify fresh installs so a lockfile does not hide peer conflicts.
- `--scripts=<published-version>` and `--eslint-plugin=<published-version>` verify registry releases. Packed-source checks do not replace this release verification.

The check reports the actual Node, Vite, Vitest, and build versions. Verify the supported Node/Vite combinations and record the released versions and results with the tooling change. The package release guide describes the protected publication process. Remaining migration release and npm deprecation gates are tracked in #80855.

Folder structure

Keep your tests in a `test` folder in your working directory. The test file should have the same name as the test subject file.

Use `*.jsdom.test.*` for DOM structure, semantics, events, state, and other deterministic behavior that does not depend on browser rendering. Use `*.browser.test.*` for generated and computed styles, cascade, responsive behavior, layout, geometry, rendered visibility, animation, scrolling, native focusability and Tab order, pointer behavior, caret geometry, `ResizeObserver`, and media queries.

In direct React Browser Mode tests, import and await `render` or `renderHook` from `vitest-browser-react`. Import `userEvent` from `vitest/browser` and prefer locators for asynchronous browser state. Testing Library query helpers can remain when Browser Mode has no equivalent, but use the Browser Mode React renderer. The shared `initializeEditor` integration helper is the existing renderer exception.

Supplied rectangles, observer notifications, and timers can remain in jsdom when they are deliberate inputs to algorithm or lifecycle tests. A browser API in setup alone does not establish that the assertions need Browser Mode. Keep exceptions specific and remove them when no longer needed.

Browser Mode loads only CSS imported by the test graph or its setup. Import a package's global Sass explicitly when the assertion depends on styles that WordPress normally enqueues

separately. Keep deterministic browser API mocks local to nonvisual tests and restore them afterwards.

Leave Node-compatible test names without an environment suffix. Every new test runs in Vitest automatically. The filename selects its environment. Do not use per-file environment overrides.

```
+-- test
|   +-- bar.test.js
+-- bar.js
```

Only test files (with at least one test case) should live directly under `/test`. If you need to add external mocks or fixtures, place them in a sub folder, for example:

- `test/mocks/[file-name].js`
- `test/fixtures/[file-name].js`

Importing tests

Given the previous folder structure, try to use relative paths when importing of the **code you're testing**, as opposed to using project paths.

Recommended:

`import { bar } from '../bar';`

Avoid this pattern:

`import { bar } from 'components/foo/bar';`

It will make your life easier should you decide to relocate your code to another position in the application directory.

Describing tests

Use a `describe` block to group test cases. Each test case should ideally describe one behaviour only.

In test cases, try to describe in plain words the expected behaviour. For UI components, this might entail describing expected behaviour from a user perspective rather than explaining code internals.

Recommended:

```
import { describe, test } from 'vitest';
```

```
describe( 'CheckboxWithLabel', () => {
    test( 'checking checkbox should disable the form submit button', () => {
        ...
    } );
} );
```

Avoid this pattern:

```
import { describe, test } from 'vitest';

describe( 'CheckboxWithLabel', () => {
    test( 'checking checkbox should set this.state.disableButton to `true`', () => {
        ...
    } );
} );
```

Setup and teardown methods

Vitest provides setup and teardown methods that allow you to perform tasks *before* and *after* each or all of your tests, or tests within a specific `describe` block.

These methods can handle asynchronous code to allow setup that you normally cannot do inline. As with asynchronous tests, return a Promise or use `async`/`await`. Vitest waits for completion:

```
import { afterAll, beforeAll } from 'vitest';

// one-time setup for *all* tests
beforeAll( () =>
    someAsyncAction().then( ( resp ) => {
        window.someGlobal = resp;
    } )
);

// one-time teardown for *all* tests
afterAll( () => {
    window.someGlobal = null;
} );
```

`afterEach` and `afterAll` provide a perfect (and preferred) way to ‘clean up’ after our tests, for example, by resetting state data.

Avoid placing clean up code after assertions since, if any of those tests fail, the clean up won't take place and may cause failures in unrelated tests.

Vitest resets mock implementations and call history, restores spies, resets stubbed globals and environment variables, and restores real timers between tests. Configure required mock implementations in each test's setup hooks. Imported module state is not reset automatically. Reset it explicitly or use `vi.resetModules()` when a fresh module instance is required. Do not disable module isolation or enable global Vitest APIs.

`wpVitest` is an explicit opt-in for jsdom suites that need hoist-safe helpers inside `vi.hoisted()`.

Expected console calls

Gutenberg's internal Vitest setup clears `@wordpress/deprecated`'s logged-message history before each test. A message still logs only once within a test. Suites do not need their own deprecation-history cleanup. To suppress a known deprecation in behavior tests, set its `logged` entry in `beforeEach` and delete that entry in the test that asserts the warning.

Gutenberg's internal Vitest setup fails a test when `console.error`, `console.warn`, `console.info`, or `console.log` has calls that the test did not explicitly expect. Use `toHaveErroredWith`, `toHaveWarnedWith`, `toHaveInformedWith`, or `toHaveLoggedWith` with specific arguments. Asymmetric matchers such as `expect.objectContaining` are supported.

```
expect( console ).toHaveWarnedWith( 'The setting is deprecated.' );
```

A successful positive assertion accounts for every matching call already in the mock history, including duplicates. It leaves the history intact, so repeated assertions and separate call-count checks still work. Other calls, including a later call with the same arguments, need their own assertion. Negative assertions and failed assertions do not account for any calls.

The argument-free matchers, such as `toHaveWarned()`, remain supported for compatibility and account for all calls to that method already in the history. A broad assertion can therefore still hide an unrelated call. Prefer specific arguments so only matching calls are accounted for. Standard spy assertions such as `toHaveBeenCalledWith` do not account for console calls in this helper.

The check runs after the test and its cleanup hooks, so assertions in `afterEach` are supported. Clearing, resetting, or restoring a mock does not excuse unaccounted calls. Assert expected calls before clearing their history. Accounting starts fresh for each test, including after a failed check.

Mocking dependencies

Dependency injection

Passing dependencies to a function as arguments can often make your code simpler to test. Where possible, avoid referencing dependencies in a higher scope.

Avoid this pattern:

```
import VALID_VALUES_LIST from './constants';


function isValueValid( value ) {
    return VALID_VALUES_LIST.includes( value );
}
```

Here we'd have to import and use a value from `VALID_VALUES_LIST` in order to pass:

`expect( isValueValid( VALID_VALUES_LIST[ 0 ] ) ).toBe( true );`

The above assertion is testing two behaviours: 1) that the function can detect an item in a list, and 2) that it can detect an item in `VALID_VALUES_LIST`.

But what if we don't care what's stored in `VALID_VALUES_LIST`, or if the list is fetched via an HTTP request, and we only want to test whether `isValueValid` can detect an item in a list?

Recommended:

```
function isValueValid( value, validValuesList = [] ) {
    return validValuesList.includes( value );
}
```

Because we're passing the list as an argument, we can pass mock `validValuesList` values in our tests and, as a bonus, test a few more scenarios:

`expect( isValueValid( 'hulk', [ 'batman', 'superman' ] ) ).toBe( false );`

`expect( isValueValid( 'hulk', null ) ).toBe( false );`

`expect( isValueValid( 'hulk', [] ) ).toBe( false );`

`expect( isValueValid( 'hulk', [ 'iron man', 'hulk' ] ) ).toBe( true );`

Imported dependencies

Often our code will use methods and properties from imported external and internal libraries in multiple places, which makes passing around arguments messy and impracticable. For these cases, use `vi.mock` to replace a module's exports.

This example uses a named export in `config.js` and a module that reads it:

```
// config.js
export const isEnabled = () => false;
```

```
// bilbo.js
import { isEnabled } from './config';

export const isBilboVisible = () => ! isEnabled( 'the-ring' );
```

```
// test/bilbo.test.js
import { beforeEach, describe, expect, test, vi } from 'vitest';
import { isEnabled } from '../config';
import { isBilboVisible } from '../bilbo';

vi.mock( import( '../config' ), () => ( {
    isEnabled: vi.fn(),
} ) );

beforeEach( () => {
    vi.mocked( isEnabled ).mockReturnValue( false );
} );

describe( 'The bilbo module', () => {
    test( 'shows Bilbo when the ring is disabled', () => {
        expect( isBilboVisible() ).toBe( true );
    } );

    test( 'hides Bilbo when the ring is enabled', () => {
        vi.mocked( isEnabled ).mockImplementationOnce(
            ( name ) => name === 'the-ring'
        );
        expect( isBilboVisible() ).toBe( false );
    } );
} );
```

`vi.mock` is hoisted. Return an object with the exports being replaced, including a `default` key for default exports. Use `vi.hoisted` if a factory needs shared mock values. Gutenberg resets mock implementations and restores spies between tests, so configure mocks in `beforeEach` or the test body. Imported module state is separate; reset it explicitly when needed.

Testing globals

Use `vi.spyOn` for an existing method and restore it after the test. This jsdom example replaces `window.open` locally:

```
// test/window.jsdom.test.js
import { afterEach, expect, test, vi } from 'vitest';

afterEach( () => {
    vi.restoreAllMocks();
} );

test( 'opens the requested page', () => {
    const open = vi.spyOn( window, 'open' ).mockReturnValue( null );
    window.open( '/example' );
    expect( open ).toHaveBeenCalledWith( '/example' );
} );
```

Use `vi.stubGlobal` or `vi.stubEnv` when replacing a global or environment value. Keep deterministic browser mocks local to jsdom tests and restore them. Do not mock layout or browser interaction to avoid Browser Mode.

User interactions

For deterministic jsdom behavior, use React Testing Library and `@testing-library/user-event`. Gutenberg supplies DOM matcher setup and React cleanup. This example checks the value reported by a DOM change event, without asserting layout or native focus behavior:

```
// test/input.jsdom.test.jsx
import { expect, test, vi } from 'vitest';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

test( 'reports the typed value', async () => {
    const user = userEvent.setup();
    const onChange = vi.fn();
    render(
        <input
            aria-label="Title"
            onChange={ ( event ) => onChange( event.target.value ) }
```

```
        />
    );

    await user.type(
        screen.getByRole( 'textbox', { name: 'Title' } ),
        'Hello'
    );
    expect( onChange ).toHaveBeenLastCalledWith( 'Hello' );
} );
```

Use Browser Mode for native interaction or rendered behavior. Import and await `render` or `renderHook` from `vitest-browser-react`. Import `userEvent` from `vitest/browser`, and await its actions. Prefer locators and `await expect.element( locator )` for asynchronous browser state. Testing Library query helpers can remain where Browser Mode has no equivalent; its React renderer is reserved for jsdom and the shared integration helper below.

```
// test/button.browser.test.jsx
import { expect, test, vi } from 'vitest';
import { userEvent } from 'vitest/browser';
import { render } from 'vitest-browser-react';

test( 'calls the handler when Save is clicked', async () => {
    const onClick = vi.fn();
    const screen = await render( <button onClick={ onClick }>Save</button> );

    await userEvent.click( screen.getByRole( 'button', { name: 'Save' } ) );
    expect( onClick ).toHaveBeenCalledTimes( 1 );
} );
```

Use synthetic `fireEvent` only when constructing a particular event is the behavior under test. Do not substitute it for browser interaction.

Integration testing for block UI

Integration tests render the components needed for a block or editor behavior as a group. They run with the unit-test commands. Their filename selects jsdom or Browser Mode. The tests render blocks in a `special instance of the block editor`.

This approach tests most block editor behavior without the full end-to-end framework. Use Browser Mode when an integration test depends on browser rendering or native input. Keep end-to-end tests for behavior that requires a complete WordPress site or navigation between pages.

`The Cover block` is an example of a block that uses this level of testing to provide coverage for a large percentage of the editor interactions.

To set up a Browser Mode integration test:

```
import { initializeEditor } from 'test/integration/helpers/integration-test-editor';

async function setup( attributes ) {
    const testBlock = { name: 'core/cover', attributes };
    return initializeEditor( testBlock );
}
```

The `initializeEditor` function returns the output of the `@testing-library/react` `render` method. This shared helper is the remaining exception to the standard Browser Mode renderer. New direct React Browser Mode tests must import and await `render` from `vitest-browser-react`. The helper also accepts an array of block metadata objects, allowing you to set up the editor with multiple blocks.

The integration test editor module also exports a `selectBlock` which can be used to select the block to be tested by the aria-label on the block wrapper, eg. “Block: Cover”.

Snapshot testing

This is an overview of snapshot testing and how to best leverage snapshot tests.

TL;DR Broken snapshots

When a snapshot test fails, it just means that a component’s rendering has changed. If that was unintended, then the snapshot test just prevented a bug □□

However, if the change was intentional, follow these steps to update the snapshot. Run the following to update the snapshots:

```
# Update snapshots for matching unit or integration tests.
npm run test:unit:update -- path/to/tests

# Update snapshot for e2e tests
npm run test:e2e -- --update-snapshots path/to/spec
```

1. Review the diff and ensure the changes are expected and intentional.
2. Commit.

What are snapshots?

Snapshots are just a representation of some data structure generated by tests. Snapshots are stored in files and committed alongside the tests. When the tests are run, the data structure generated is compared with the snapshot on file.

It’s very easy to make a snapshot:

```
import { expect, test } from 'vitest';

test( 'foobar test', () => {
    const foobar = { foo: 'bar' };

    expect( foobar ).toMatchSnapshot();
} );
```

This is the produced snapshot:

```
exports[ `foobar test 1` ] = `
{
  "foo": "bar",
}
`;
```

You should never create or modify a snapshot directly, they are generated and updated by tests.

Advantages

- Trivial and concise to add tests.
- Protect against unintentional changes.
- Simple to work with.
- Reveal internal structures without running the application.

Disadvantages

- Not expressive.
- Only catch issues when changes are introduced.
- Are problematic for anything non-deterministic.

Use cases

Snapshots can record changes to a component’s structure during a refactor. This jsdom example uses a local component and explicit Vitest imports:

```
import { describe, expect, test } from 'vitest';
import { render, screen } from '@testing-library/react';

function SolarSystem( { planets = false } ) {
    return <div>{ planets && <span>Mars</span> }</div>;
}

describe( 'SolarSystem', () => {
    test( 'should render', () => {
        const { container } = render( <SolarSystem /> );

        expect( container ).toMatchSnapshot();
    } );

    test( 'should contain mars if planets is true', () => {
        const { container } = render( <SolarSystem planets /> );

        expect( container ).toMatchSnapshot();
        expect( screen.getByText( /mars/i ) ).toBeInTheDocument();
    } );
} );
```

Reducer tests are also a great fit for snapshots. They are often large, complex data structures that shouldn't change unexpectedly, exactly what snapshots excel at!

Working with snapshots

You might be blindsided by CI tests failing when snapshots don't match. You'll need to update snapshots if the changes are expected:

```
npm run test:unit:update -- path/to/tests
```

The path is optional, but specifying one runs only matching tests and avoids updating unrelated snapshots. Review the diff before committing it.

Keep the focused watch command running as you work:

```
npm run test:unit:watch -- path/to/tests
```

When a snapshot test fails in watch mode, press `u` to update it.

Pain points

Non-deterministic tests may not make consistent snapshots, so beware. When working with anything random, time-based, or otherwise non-deterministic, snapshots will be problematic.

Connected components are tricky to work with. To snapshot a connected component you'll probably want to export the unconnected component:

```
// my-component.js
export { MyComponent };
export default connect( mapStateToProps )( MyComponent );

// test/my-component.js
import { MyComponent } from '..';
// run those MyComponent tests…
```

The connected props will need to be manually provided. This is a good opportunity to audit the connected state.

Best practices

If you're starting a refactor, snapshots are quite nice, you can add them as the first commit on a branch and watch as they evolve.

Snapshots themselves don't express anything about what we expect. Snapshots are best used in conjunction with other tests that describe our expectations, like in the example above:

```
import { expect, test } from 'vitest';
import { render, screen } from '@testing-library/react';

function SolarSystem( { planets = false } ) {
    return <div>{ planets && <span>Mars</span> }</div>;
}

test( 'should contain mars if planets is true', () => {
    const { container } = render( <SolarSystem planets /> );

    // Snapshot will catch unintended changes
    expect( container ).toMatchSnapshot();

    // This is what we actually expect to find in our test
    expect( screen.getByText( /mars/i ) ).toBeInTheDocument();
```

```
} );
```

Prefer direct assertions for the behavior that changes between two states. `@testing-library/jest-dom` remains supported; its name does not make it a Jest runner dependency.

Test rendered styles in Browser Mode. Node and jsdom use a CSS-module proxy, so their snapshots cannot establish computed style behavior. Browser Mode loads styles imported by the test graph. Import a package's global stylesheet explicitly when WordPress normally enqueues it separately. For a test in `packages/components/src/button/test/`, for example:

```
import { expect, test } from 'vitest';
import { render } from 'vitest-browser-react';
import Button from '../';
import '../style.scss';

test( 'loads the button stylesheet', async () => {
    const screen = await render( <Button __next40pxDefaultSize>Save</Button> );
    await expect
        .element( screen.getByRole( 'button', { name: 'Save' } ) )
        .toHaveStyle( { height: '40px' } );
} );
```

Debugging Vitest unit tests

For a Node or jsdom test, run the focused `npm run test:unit:debug -- path/to/test` command. It uses `--inspect-brk --no-file-parallelism --watch=false`, pauses the worker before test execution, and prints its inspector address. Open `chrome://inspect` or attach a Node inspector client. Resume execution, then use breakpoints or `debugger;` in the test. See the scripts debugger instructions.

For Browser Mode, run a focused test with a visible browser and open its developer tools:

```
npm run test:unit:watch -- path/to/example.browser.test.jsx --browser.headless=false
```

Use the browser's debugger for browser code; the Node inspector command targets Node workers.

Browser Mode failure artifacts

The **JavaScript Browser (Chromium, Node.js 24)** job in the **Unit Tests** workflow uploads failure screenshots. Download the `vitest-browser-failures` artifact from the workflow run's summary page within three days of the run. The artifact is only uploaded when the job fails and files exist; a failure before browser tests start might have no artifact.

Tracing is off by default. For a focused diagnostic run, select **Run workflow**, choose the failing branch, and enter one repository-relative `*.browser.test.*` file in **browser-trace-file**. The Browser job runs that file with Playwright tracing and retains its trace if the file fails or raises an unhandled browser error. Other jobs keep their usual scope. This is a separate diagnostic run, so its result does not establish that the full Browser suite passes.

The artifact preserves these directories relative to the local `test-results/` directory:

- `vitest-attachments/failure-screenshots/`: automatic failure screenshots.
- `vitest-browser-screenshots/`: screenshots taken with Browser Mode's screenshot API, if present.
- `vitest-browser-traces/`: `.trace.zip` archives for failed files, preserving their repository-relative paths.

Test output identifies the files for each failure. Open PNG screenshots with an image viewer. To inspect a trace, use an absolute path to the extracted ZIP:

```
npm exec --no --workspace @wordpress/unit-tests -- playwright show-trace
/absolute/path/to/example.trace.zip
```

The Playwright Trace Viewer shows recorded Playwright actions, DOM snapshots, screenshots, and network activity. Each archive covers one test file, including its setup and teardown. Use the test failure output to identify the relevant actions; JavaScript assertions are not recorded as Playwright actions.

To record the same diagnostics locally:

```
WP_VITEST_BROWSER_TRACE=1 npm run test:unit -- --project=browser
path/to/example.browser.test.jsx
```

Omit `WP_VITEST_BROWSER_TRACE` to measure the same run without tracing. Tracing adds runtime and temporary disk usage even for passing files. Recordings from files that pass without unhandled errors are discarded without exporting a ZIP. Each traced run clears the previous trace output; screenshots from earlier local failures can remain. Tracing uses the worker's existing browser context and preserves Vitest's per-file iframe isolation. If an earlier file's recording is still active when the context is reused, it is retained before the next trace starts. Tracing does not add retries. Do not combine it with Vitest's `--browser.trace` option, which controls a separate tracing implementation.

End-to-end testing

End-to-end tests use Playwright as the testing framework. See the dedicated End-to-End Testing guide for best practices and detailed instructions.

Using wp-env

If you're using the built-in local environment, you can run the e2e tests locally using this command:

```
npm run test:e2e
```

or interactively

```
npm run test:e2e -- --ui
```

Scenario testing

If you find that end-to-end tests pass when run locally, but fail in GitHub Actions, you may be able to isolate a CPU- or network-bound race condition by simulating a slow CPU or network:

```
THROTTLE_CPU=4 npm run test:e2e
```

`THROTTLE_CPU` is a slowdown factor (in this example, a 4x slowdown multiplier)

See Chrome docs: setCPUThrottlingRate

```
SLOW_NETWORK=true npm run test:e2e
```

`SLOW_NETWORK` emulates a network speed equivalent to "Fast 3G" in the Chrome devtools.

See Chrome docs: emulateNetworkConditions and NetworkManager.js

```
OFFLINE=true npm run test:e2e
```

`OFFLINE` emulates network disconnection.

See Chrome docs: emulateNetworkConditions

Core block testing

Every core block is required to have at least one set of fixture files for its main save function and one for each deprecation. These fixtures test the parsing and serialization of the block. See the integration tests fixtures readme for more information and instructions.

Flaky tests

A test is considered to be **flaky** when it can pass and fail across multiple retry attempts without any code changes. We auto retry failed tests at most **twice** on CI to detect them, and report them, together with their errors, as a single comment on the pull request via the `report-flaky-tests` GitHub action. Note that a test that failed three times in a row is not counted as a flaky test and will not be reported, and that flaky tests are only reported on pull requests.

PHP testing

Tests for PHP use PHPUnit as the testing framework. If you're using the built-in local environment, you can run the PHP tests locally using this command:

```
npm run test:php
```

To re-run tests automatically when files change (similar to Vitest), run:

```
npm run test:php:watch
```

Note: The phpunit commands require `wp-env` to be running and composer dependencies to be installed. The package script will start wp-env for you if it is not already running.

In other environments, run `composer run test` and `composer run test:watch`.

Code style in PHP is enforced using PHP_CodeSniffer. It is recommended that you install PHP_CodeSniffer and the WordPress Coding Standards for PHP_CodeSniffer ruleset using Composer. With Composer installed, run `composer install` from the project directory to install dependencies. The above `npm run test:php` will execute both unit tests and code linting. Code linting can be verified independently by running `npm run lint:php`.

To run unit tests only, without the linter, use `npm run test:unit:php` instead.

Testing Prefixed Functions

Gutenberg's build system automatically prefixes PHP functions with `gutenberg_` to avoid conflicts with WordPress Core. When writing tests for block functions, you must test the **built (prefixed) versions** of functions, not the source versions.

If the tests are backported to WordPress Core, then they must be updated to test the non-prefixed functions.

Writing Tests for Prefixed Functions & Classes

Always test the built (prefixed) function names and class names in your PHPUnit tests:

```
// phpunit/blocks/my-block-test.php
class My_Block_Test extends WP_UnitTestCase {
    public function test_my_function() {
        // Test the built function (with gutenberg_ prefix)
        $result = gutenberg_block_core_my_block_render_function( $args );
        $this->assertEquals( $expected, $result );
    }

    public function test_my_class() {
        // Test the built class (with _Gutenberg suffix)
        $handler = new WP_Example_Block_Handler_Gutenberg();
        $result = $handler->process( $input );
        $this->assertEquals( $expected, $result );
    }
}
```

For more detailed information about the build system and function prefixing, see the Build System: Function Prefixing and Block Loading documentation.

Performance testing

To ensure that the editor stays performant as we add features, we monitor the impact pull requests and releases can have on some key metrics including:

- The time it takes to load the editor.
- The time it takes for the browser to respond when typing.
- The time it takes to select a block.

Performance tests are end-to-end tests running the editor and capturing these measures. Make sure you have an e2e testing environment ready.

To set up the e2e testing environment, checkout the Gutenberg repository and switch to the branch that you would like to test. Run the following command to prepare the environment.

```
nvm use && npm install
npm run build
```

To run the tests run the following command:

```
npm run test:performance
```

This gives you the result for the current branch/code on the running environment.

In addition to that, you can also compare the metrics across branches (or tags or commits) by running the following command `npm exec --no release-cli -- perf [branches]`, example:

```
npm exec --no release-cli -- perf trunk v8.1.0 v8.0.0
```

Finally, you can pass an additional `--tests-branch` argument to specify which branch's performance test files you'd like to run. This is particularly useful when modifying/extending the perf tests:

```
npm exec --no release-cli -- perf trunk v8.1.0 v8.0.0 --tests-branch add/perf-tests-coverage
```

Note This command needs may take some time to perform the benchmark. While running make sure to avoid using your computer or have a lot of background process to minimize external factors that can impact the results across branches.

□□□ #2
□ □□□ □□□ 2026-10-31 18:25:24 UTC
□ □□□ □□□ 2026-10-31 18:25:27 UTC