Frontend testing standards and style guidelines

There are two types of test suites encountered while developing frontend code at GitLab. We use Jest for JavaScript unit and integration testing, and RSpec feature tests with Capybara for e2e (end-to-end) integration testing.

Unit and feature tests need to be written for all new features. Most of the time, you should use RSpec for your feature tests. For more information on how to get started with feature tests, see get started with feature tests

Regression tests should be written for bug fixes to prevent them from recurring in the future.

See the Testing Standards and Style Guidelines page for more information on general testing practices at GitLab.

Vue.js testing

If you are looking for a guide on Vue component testing, you can jump right away to this section.

Jest

We use Jest to write frontend unit and integration tests. Jest tests can be found in `/spec/frontend` and `/ee/spec/frontend` in EE.

Limitations of jsdom

Jest uses jsdom instead of a browser for running tests. This comes with a number of limitations, namely:

- No scrolling support
- No element sizes or positions
- No layout engine in general

See also the issue for support running Jest tests in browsers.

Debugging Jest tests

Running `yarn jest-debug` runs Jest in debug mode, allowing you to debug/inspect as described in the Jest docs.

Timeout error

The default timeout for Jest is set in `/spec/frontend/test_setup.js`.

If your test exceeds that time, it fails.

If you cannot improve the performance of the tests, you can increase the timeout for a specific test using `jest.setTimeout`

```
describe('Component', () => {
  it('does something amazing', () => {
    jest.setTimeout(500);
    // ...
  });
});
```

Remember that the performance of each test depends on the environment.

Test-specific stylesheets

To help facilitate RSpec integration tests we have two test-specific stylesheets. These can be used to do things like disable animations to improve test speed, or to make elements visible when they need to be targeted by Capybara click events:

- `app/assets/stylesheets/disable_animations.scss`
- `app/assets/stylesheets/test_environment.scss`

Because the test environment should match the production environment as much as possible, use these minimally and only add to them when necessary.

What and how to test

Before jumping into more gritty details about Jest-specific workflows like mocks and spies, we should briefly cover what to test with Jest.

Don't test the library

Libraries are an integral part of any JavaScript developer's life. The general advice would be to not test library internals, but expect that the library knows what it's supposed to do and has test coverage on its own. A general example could be something like this

```
import { convertToFahrenheit } from 'temperatureLibrary'

function getFahrenheit(celsius) {
  return convertToFahrenheit(celsius)
}
```

It does not make sense to test our `getFahrenheit` function because underneath it does nothing else but invoking the library function, and we can expect that one is working as intended.

Let's take a short look into Vue land. Vue is a critical part of the GitLab JavaScript codebase. When writing specs for Vue components, a common gotcha is to actually end up testing Vue provided functionality, because it appears to be the easiest thing to test. Here's an example taken from our codebase.

```
// Component script
{
  computed: {
    hasMetricTypes() {
      return this.metricTypes.length;
    },
}
```

```
<!-- Component template -->
<template>
  <gl-dropdown v-if="hasMetricTypes">
    <!-- Dropdown content -->
  </gl-dropdown>
</template>
```

Testing the `hasMetricTypes` computed prop would seem like a given here. But to test if the computed property is returning the length of `metricTypes`, is testing the Vue library itself. There is no value in this, besides it adding to the test suite. It's better to

test a component in the way the user interacts with it: checking the rendered template.

```
// Bad
describe('computed', () => {
  describe('hasMetricTypes', () => {
    it('returns true if metricTypes exist', () => {
      factory({ metricTypes });
      expect(wrapper.vm.hasMetricTypes).toBe(2);
    });

    it('returns true if no metricTypes exist', () => {
      factory();
      expect(wrapper.vm.hasMetricTypes).toBe(0);
    });
  });
});

// Good
it('displays a dropdown if metricTypes exist', () => {
  factory({ metricTypes });
  expect(wrapper.findComponent(GlDropdown).exists()).toBe(true);
});

it('does not display a dropdown if no metricTypes exist', () => {
  factory();
  expect(wrapper.findComponent(GlDropdown).exists()).toBe(false);
});
```

Keep an eye out for these kinds of tests, as they just make updating logic more fragile and tedious than it needs to be. This is also true for other libraries. A suggestion here is: if you are checking a `wrapper.vm` property, you should probably stop and rethink the test to check the rendered template instead.

Some more examples can be found in the Frontend unit tests section

Don't test your mock

Another common gotcha is that the specs end up verifying the mock is working. If you are using mocks, the mock should support the test, but not be the target of the test.

```
const spy = jest.spyOn(idGenerator, 'create')
spy.mockImplementation = () = '1234'

// Bad
expect(idGenerator.create()).toBe('1234')

// Good: actually focusing on the logic of your component and just leverage the controllable mocks output
expect(wrapper.find('div').html()).toBe('<div id="1234">...</div>')
```

Follow the user

The line between unit and integration tests can be quite blurry in a component heavy world. The most important guideline to give is the following:

- Write clean unit tests if there is actual value in testing a complex piece of logic in isolation to prevent it from breaking in the future
- Otherwise, try to write your specs as close to the user's flow as possible

For example, it's better to use the generated markup to trigger a button click and validate the markup changed accordingly than to call a method manually and verify data structures or computed properties. There's always the chance of accidentally breaking the user flow, while the tests pass and provide a false sense of security.

Common practices

These some general common practices included as part of our test suite. Should you stumble over something not following this guide, ideally fix it right away.

How to query DOM elements

When it comes to querying DOM elements in your tests, it is best to uniquely and semantically target the element.

Preferentially, this is done by targeting what the user actually sees using DOM Testing Library. When selecting by text it is best to use the `byRole` query as it helps enforce accessibility best practices. `findByRole` and the other DOM Testing Library queries are available when using `shallowMountExtended` or `mountExtended`.

When writing Vue component unit tests, it can be wise to query children by component, so that the unit test can focus on comprehensive value coverage rather than dealing with the complexity of a child component's behavior.

Sometimes, neither of the above are feasible. In these cases, adding test attributes to simplify the selectors might be the best option. A list of possible selectors include:

- A semantic attribute like `name` (also verifies that `name` was setup properly)
- A `data-testid` attribute (recommended by maintainers of `@vue/test-utils`) optionally combined with `shallowMountExtended` or `mountExtended`
- a Vue `ref` (if using `@vue/test-utils`)

```
import { shallowMountExtended } from 'helpers/vue_test_utils_helper'

const wrapper = shallowMountExtended(ExampleComponent);

it('exists', () => {
  // Best (especially for integration tests)
  wrapper.findByRole('link', { name: /Click Me/i })
  wrapper.findByRole('link', { name: 'Click Me' })
  wrapper.findByText('Click Me')
  wrapper.findByText(/Click Me/i)

  // Good (especially for unit tests)
  wrapper.findComponent(FooComponent);
  wrapper.find('input[name=foo]');
  wrapper.find('[data-testid="my-foo-id"]');
  wrapper.findByTestId('my-foo-id'); // with shallowMountExtended or mountExtended - check below
  wrapper.find({ ref: 'foo'});

  // Bad
  wrapper.find('.js-foo');
  wrapper.find('.btn-primary');
  wrapper.find('.qa-foo-component');
  wrapper.find('[data-qa-selector="foo"]');
});
```

It is recommended to use `kebab-case` for `data-testid` attribute.

It is not recommended that you add `.js-*` classes just for testing purposes. Only do this if there are no other feasible options available.

Do not use a `.qa-*` class or `data-qa-selector` attribute for any tests other than QA end-to-end testing.

Querying for child components

When testing Vue components with `@vue/test-utils` another possible approach is querying for child components instead of querying for DOM nodes. This assumes that implementation details of behavior under test should be covered by that component's individual unit test. There is no strong preference in writing DOM or component queries as long as your tests reliably cover expected behavior for the component under test.

Example:

```
it('exists', () => {
  wrapper.findComponent(FooComponent);
});
```

Naming unit tests

When writing describe test blocks to test specific functions/methods, use the method name as the describe block name.

Bad:

```
describe('#methodName', () => {
  it('passes', () => {
    expect(true).toEqual(true);
  });
});

describe('.methodName', () => {
  it('passes', () => {
    expect(true).toEqual(true);
  });
});
```

Good:

```
describe('methodName', () => {
  it('passes', () => {
    expect(true).toEqual(true);
  });
});
```

Testing promises

When testing Promises you should always make sure that the test is asynchronous and rejections are handled. It's now possible to use the `async/await` syntax in the test suite:

```
it('tests a promise', async () => {
  const users = await fetchUsers()
  expect(users.length).toBe(42)
});

it('tests a promise rejection', async () => {
  await expect(user.getUserName(1)).rejects.toThrow('User with 1 not found.');
});
```

You can also return a promise from the test function.

Using the `done` and `done.fail` callbacks is discouraged when working with promises. They should not be used.

Bad:

```
// missing return
it('tests a promise', () => {
  promise.then(data => {
    expect(data).toBe(asExpected);
  });
});

// uses done/done.fail
it('tests a promise', done => {
  promise
    .then(data => {
      expect(data).toBe(asExpected);
    })
    .then(done)
    .catch(done.fail);
});
```

Good:

```
// verifying a resolved promise
it('tests a promise', () => {
  return promise
    .then(data => {
      expect(data).toBe(asExpected);
    });
});

// verifying a resolved promise using Jest's `resolves` matcher
it('tests a promise', () => {
  return expect(promise).resolves.toBe(asExpected);
});

// verifying a rejected promise using Jest's `rejects` matcher
it('tests a promise rejection', () => {
  return expect(promise).rejects.toThrow(expectedError);
});
```

Manipulating Time

Sometimes we have to test time-sensitive code. For example, recurring events that run every X amount of seconds or similar. Here are some strategies to deal with that:

`...` in application

`setTimeout()` / `setInterval()` in application

If the application itself is waiting for some time, mock await the waiting. In Jest this is already done by default (see also Jest Timer Mocks).

```
const doSomethingLater = () => {
  setTimeout(() => {
    // do something
  }, 4000);
};
```

in Jest:

```
it('does something', () => {
  doSomethingLater();
  jest.runAllTimers();

  expect(something).toBe('done');
});
```

Mocking the current location in Jest

NOTE: The value of `window.location.href` is reset before every test to avoid earlier tests affecting later ones.

If your tests require `window.location.href` to take a particular value, use the `setWindowLocation` helper:

```
import setWindowLocation from 'helpers/set_window_location';

it('passes', () => {
  setWindowLocation('https://gitlab.test/foo?bar=true');

  expect(window.location).toMatchObject({
    hostname: 'gitlab.test',
    pathname: '/foo',
    search: '?bar=true',
  });
});
```

To modify only the hash, use either the `setWindowLocation` helper, or assign directly to `window.location.hash`, for example:

```
it('passes', () => {
  window.location.hash = '#foo';

  expect(window.location.href).toBe('http://test.host/#foo');
});
```

If your tests need to assert that certain `window.location` methods were called, use the `useMockLocationHelper` helper:

```
import { useMockLocationHelper } from 'helpers/mock_window_location_helper';

useMockLocationHelper();

it('passes', () => {
  window.location.reload();

  expect(window.location.reload).toHaveBeenCalled();
});
```

Waiting in tests

Sometimes a test needs to wait for something to happen in the application before it continues.

You should try to avoid:

- `setTimeout` because it makes the reason for waiting unclear. Additionally, it is faked in our tests so its usage is tricky.
- `setImmediate` because it is no longer supported in Jest 27 and later. See this epic for details.

Promises and Ajax calls

Register handler functions to wait for the `Promise` to be resolved.

```
const askTheServer = () => {
  return axios
    .get('/endpoint')
    .then(response => {
      // do something
    })
    .catch(error => {
      // do something else
    });
};
```

in Jest:

```
it('waits for an Ajax call', async () => {
  await askTheServer()
  expect(something).toBe('done');
});
```

If you cannot register handlers to the `Promise`, for example because it is executed in a synchronous Vue lifecycle hook, take a look at the `waitFor` helpers or flush all pending `Promise`s with:

in Jest:

```
it('waits for an Ajax call', async () => {
  synchronousFunction();

  await waitForPromises();

  expect(something).toBe('done');
});
```

Vue rendering

Use `nextTick()` to wait until a Vue component is re-rendered.

in Jest:

```
import { nextTick } from 'vue';

// ...

it('renders something', async () => {
  wrapper.setProps({ value: 'new value' });

  await nextTick();

  expect(wrapper.text()).toBe('new value');
});
```

Events

If the application triggers an event that you need to wait for in your test, register an event handler which contains the assertions:

```
it('waits for an event', () => {
  eventHub.$once('someEvent', eventHandler);

  someFunction();

  return new Promise((resolve) => {
    function expectEventHandler() {
      expect(something).toBe('done');
      resolve();
    }
  });
});
```

In Jest you can also use a `Promise` for this:

```
it('waits for an event', () => {
  const eventTriggered = new Promise(resolve => eventHub.$once('someEvent', resolve));

  someFunction();

  return eventTriggered.then(() => {
    expect(something).toBe('done');
  });
});
```

Manipulate `gon` object

`gon` (or `window.gon`) is a global object used to pass data from the backend. If your test depends on its value you can directly modify it:

```
describe('when logged in', () => {
  beforeEach(() => {
    gon.current_user_id = 1;
  });

  it('shows message', () => {
    expect(wrapper.text()).toBe('Logged in!');
  });
})
```

`gon` is reset in every test to ensure tests are isolated.

Ensuring that tests are isolated

Tests are normally architected in a pattern which requires a recurring setup of the component under test. This is often achieved by making use of the `beforeEach` hook.

Example

```
  let wrapper;

  beforeEach(() => {
    wrapper = mount(Component);
  });
```

With enableAutoDestroy, it is no longer necessary to manually call `wrapper.destroy()`. However, some mocks, spies, and fixtures do need to be torn down, and we can leverage the `afterEach` hook.

Example

```
  let wrapper;

  afterEach(() => {
    fakeApollo = null;
    store = null;
  });
```

Testing local-only Apollo queries and mutations

To add a new query or mutation before it is added to the backend, we can use the `@client` directive. For example:

```
mutation setActiveBoardItemEE($boardItem: LocalBoardItem, $isIssue: Boolean = true) {
  setActiveBoardItem(boardItem: $boardItem) @client {
    ...Issue @include(if: $isIssue)
    ...EpicDetailed @skip(if: $isIssue)
  }
}
```

When writing test cases for such calls, we can use resolvers to make sure they are called with the correct parameters.

For example, when creating the wrapper, we should make sure the resolver is mapped to the query or mutation. The mutation we are mocking here is `setActiveBoardItem`:

```
const mockSetActiveBoardItemResolver = jest.fn();
const mockApollo = createMockApollo([], {
    Mutation: {
      setActiveBoardItem: mockSetActiveBoardItemResolver,
    },
});
```

In the following code, we must pass four arguments. The second one must be the collection of input variables of the query or

In the following code, we must pass four arguments. The second one must be the collection of input variables of the query or mutation mocked. To test that the mutation is called with the correct parameters:

```
it('calls setActiveBoardItemMutation on close', async () => {
    wrapper.findComponent(GlDrawer).vm.$emit('close');

    await waitForPromises();

    expect(mockSetActiveBoardItemResolver).toHaveBeenCalledWith(
        {},
        {
            boardItem: null,
        },
        expect.anything(),
        expect.anything(),
    );
});
```

Jest best practices

> Introduced in GitLab 13.2.

Prefer `toBe` over `toEqual` when comparing primitive values

Jest has `toBe` and `toEqual` matchers. As `toBe` uses `Object.is` to compare values, it's faster (by default) than using `toEqual`. While the latter eventually falls back to leverage `Object.is`, for primitive values, it should only be used when complex objects need a comparison.

Examples:

```
const foo = 1;

// Bad
expect(foo).toEqual(1);

// Good
expect(foo).toBe(1);
```

Prefer more befitting matchers

Jest provides useful matchers like `toHaveLength` or `toBeUndefined` to make your tests more readable and to produce more understandable error messages. Check their docs for the full list of matchers.

Examples:

```
const arr = [1, 2];

// prints:
// Expected length: 1
// Received length: 2
expect(arr).toHaveLength(1);

// prints:
// Expected: 1
// Received: 2
expect(arr.length).toBe(1);

// prints:
// expect(received).toBe(expected) // Object.is equality
// Expected: undefined
// Received: "bar"
const foo = 'bar';
expect(foo).toBe(undefined);

// prints:
// expect(received).toBeUndefined()
// Received: "bar"
const foo = 'bar';
expect(foo).toBeUndefined();
```

Avoid using `toBeTruthy` or `toBeFalsy`

Jest also provides following matchers: `toBeTruthy` and `toBeFalsy`. We should not use them because they make tests weaker and produce false-positive results.

For example, `expect(someBoolean).toBeFalsy()` passes when `someBoolean === null`, and when `someBoolean === false`.

Tricky `toBeDefined` matcher

Jest has the tricky `toBeDefined` matcher that can produce false positive test. Because it validates the given value for `undefined` only.

```
// Bad: if finder returns null, the test will pass
expect(wrapper.find('foo')).toBeDefined();

// Good
expect(wrapper.find('foo').exists()).toBe(true);
```

Avoid using `setImmediate`

Try to avoid using `setImmediate`. `setImmediate` is an ad-hoc solution to run your callback after the I/O completes. And it's not part of the Web API, hence, we target NodeJS environments in our unit tests.

Instead of `setImmediate`, use `jest.runAllTimers` or `jest.runOnlyPendingTimers` to run pending timers. The latter is useful when you have `setInterval` in the code. **Remember:** our Jest configuration uses fake timers.

Avoid non-deterministic specs

Non-determinism is the breeding ground for flaky and brittle specs. Such specs end up breaking the CI pipeline, interrupting the work flow of other contributors.

1. Make sure your test subject's collaborators (for example, Axios, Apollo, Lodash helpers) and test environment (for example, Date) behave consistently across systems and over time.
2. Make sure tests are focused and not doing "extra work" (for example, needlessly creating the test subject more than once in an individual test).

Faking `Date` for determinism

`Date` is faked by default in our Jest environment. This means every call to `Date()` or `Date.now()` returns a fixed deterministic

value.

If you really need to change the default fake date, you can call `useFakeDate` within any `describe` block, and the date will be replaced for that specs within that `describe` context only:

```
import { useFakeDate } from 'helpers/fake_date';

describe('cool/component', () => {
  // Default fake `Date`
  const TODAY = new Date();

  // NOTE: `useFakeDate` cannot be called during test execution (that is, inside `it`, `beforeEach`, `beforeAll`
  describe("on Ada Lovelace's Birthday", () => {
    useFakeDate(1815, 11, 10)

    it('Date is no longer default', () => {
      expect(new Date()).not.toEqual(TODAY);
    });
  });

  it('Date is still default in this scope', () => {
    expect(new Date()).toEqual(TODAY)
  });
})
```

Similarly, if you really need to use the real `Date` class, then you can import and call `useRealDate` within any `describe` block:

```
import { useRealDate } from 'helpers/fake_date';

// NOTE: `useRealDate` cannot be called during test execution (that is, inside `it`, `beforeEach`, `beforeAll`, 
describe('with real date', () => {
  useRealDate();
});
```

Faking `Math.random` for determinism

Consider replacing `Math.random` with a fake when the test subject depends on it.

```
beforeEach(() => {
  // https://xkcd.com/221/
  jest.spyOn(Math, 'random').mockReturnValue(0.4);
});
```

Factories

TBU

Mocking Strategies with Jest

Stubbing and Mocking

Stubs or spies are often used synonymously. In Jest it's quite easy thanks to the `.spyOn` method. Official docs The more challenging part are mocks, which can be used for functions or even dependencies.

Manual module mocks

Manual mocks are used to mock modules across the entire Jest environment. This is a very powerful testing tool that helps simplify unit testing by mocking out modules that cannot be easily consumed in our test environment.

> **WARNING:** Do not use manual mocks if a mock should not be consistently applied in every spec (that is, it's only needed by a few specs). Instead, consider using `jest.mock(..)` (or a similar mocking function) in the relevant spec file.

Where should you put manual mocks?

Jest supports manual module mocks by placing a mock in a `__mocks__/` directory next to the source module (for example, `app/assets/javascripts/ide/__mocks__`). **Don't do this.** We want to keep all of our test-related code in one place (the `spec/` folder).

If a manual mock is needed for a `node_modules` package, use the `spec/frontend/__mocks__` folder. Here's an example of a Jest mock for the package `monaco-editor`.

If a manual mock is needed for a CE module, place the implementation in `spec/frontend/__helpers__/mocks` and add a line to the `frontend/test_setup` (or the `frontend/shared_test_setup`) that looks something like:

```
// "~/lib/utils/axios_utils" is the path to the real module
// "helpers/mocks/axios_utils" is the path to the mocked implementation
jest.mock('~/lib/utils/axios_utils', () => jest.requireActual('helpers/mocks/axios_utils'));
```

Manual mock examples

- `__helpers__/mocks/axios_utils` - This mock is helpful because we don't want any unmocked requests to pass any tests. Also, we are able to inject some test helpers such as `axios.waitForAll`.
- `__mocks__/mousetrap/index.js` - This mock is helpful because the module itself uses AMD format which webpack understands, but is incompatible with the jest environment. This mock doesn't remove any behavior, only provides a nice es6 compatible wrapper.
- `__mocks__/monaco-editor/index.js` - This mock is helpful because the Monaco package is completely incompatible in a Jest environment. In fact, webpack requires a special loader to make it work. This mock makes this package consumable by Jest.

Keep mocks light

Global mocks introduce magic and technically can reduce test coverage. When mocking is deemed profitable:

- Keep the mock short and focused.
- Leave a top-level comment in the mock on why it is necessary.

Additional mocking techniques

Consult the official Jest docs for a full overview of the available mocking features.

Running Frontend Tests

Before generating fixtures, make sure you have a running GDK instance.

For running the frontend tests, you need the following commands:

- `rake frontend:fixtures` (re-)generates fixtures. Make sure that fixtures are up-to-date before running tests that require them.
- `yarn jest` runs Jest tests.

Live testing and focused testing -- Jest

While you work on a test suite, you may want to run these specs in watch mode, so they rerun automatically on every save.

```
# Watch and rerun all specs matching the name icon
yarn jest --watch icon

# Watch and rerun one specific file
yarn jest --watch path/to/spec/file.spec.js
```

You can also run some focused tests without the `--watch` flag

```
# Run specific jest file
yarn jest ./path/to/local_spec.js
# Run specific jest folder
yarn jest ./path/to/folder/
# Run all jest files which path contain term
yarn jest term
```

Frontend test fixtures

Frontend fixtures are files containing responses from backend controllers. These responses can be either HTML generated from HAML templates or JSON payloads. Frontend tests that rely on these responses are often using fixtures to validate correct integration with the backend code.

Use fixtures

To import a JSON or HTML fixture, `import` it using the `test_fixtures` alias.

```
import responseBody from 'test_fixtures/some/fixture.json' // loads tmp/tests/frontend/fixtures-ee/some/fixture.j

it('makes a request', () => {
  axiosMock.onGet(endpoint).reply(200, responseBody);

  myButton.click();

  // ...
});
```

Generate fixtures

You can find code to generate test fixtures in:

- `spec/frontend/fixtures/`, for running tests in CE.
- `ee/spec/frontend/fixtures/`, for running tests in EE.

You can generate fixtures by running:

- `bin/rake frontend:fixtures` to generate all fixtures
- `bin/rspec spec/frontend/fixtures/merge_requests.rb` to generate specific fixtures (in this case for `merge_request.rb`)

You can find generated fixtures are in `tmp/tests/frontend/fixtures-ee`.

Creating new fixtures

For each fixture, you can find the content of the `response` variable in the output file. For example, a test named `"merge_requests/diff_discussion.json"` in `spec/frontend/fixtures/merge_requests.rb` produces an output file `tmp/tests/frontend/fixtures-ee/merge_requests/diff_discussion.json`. The `response` variable gets automatically set if the test is marked as `type: :request` or `type: :controller`.

When creating a new fixture, it often makes sense to take a look at the corresponding tests for the endpoint in `(ee/)spec/controllers/` or `(ee/)spec/requests/`.

GraphQL query fixtures

You can create a fixture that represents the result of a GraphQL query using the `get_graphql_query_as_string` helper method. For example:

```
# spec/frontend/fixtures/releases.rb

describe GraphQL::Query, type: :request do
  include GraphqlHelpers

  all_releases_query_path = 'releases/graphql/queries/all_releases.query.graphql'

  it "graphql/#{all_releases_query_path}.json" do
    query = get_graphql_query_as_string(all_releases_query_path)

    post_graphql(query, current_user: admin, variables: { fullPath: project.full_path })

    expect_graphql_errors_to_be_empty
  end
end
```

This will create a new fixture located at `tmp/tests/frontend/fixtures-ee/graphql/releases/graphql/queries/all_releases.query.graphql.json`.

You can import the JSON fixture in a Jest test using the `test_fixtures` alias as described previously.

Data-driven tests

Similar to RSpec's parameterized tests, Jest supports data-driven tests for:

- Individual tests using `test.each` (aliased to `it.each`).
- Groups of tests using `describe.each`.

These can be useful for reducing repetition within tests. Each option can take an array of data values or a tagged template literal.

For example:

```
// function to test
const icon = status => status ? 'pipeline-passed' : 'pipeline-failed'
```

```
const message = status => status ? 'pipeline-passed' : 'pipeline-failed'

// test with array block
it.each([
    [false, 'pipeline-failed'],
    [true, 'pipeline-passed']
])('icon with %s will return %s',
 (status, icon) => {
    expect(renderPipeline(status)).toEqual(icon)
 }
);
```

NOTE: Only use template literal block if pretty print is not needed for spec output. For example, empty strings, nested objects etc.

For example, when testing the difference between an empty search string and a non-empty search string, the use of the array block syntax with the pretty print option would be preferred. That way the differences between an empty string (`''`) and a non-empty string (`'search string'`) would be visible in the spec output. Whereas with a template literal block, the empty string would be shown as a space, which could lead to a confusing developer experience.

```
// bad
it.each`
    searchTerm | expected
    ${''} | ${{ issue: { users: { nodes: [] } } }}
    ${'search term'} | ${{ issue: { other: { nested: [] } } }}
`('when search term is $searchTerm, it returns $expected', ({ searchTerm, expected }) => {
  expect(search(searchTerm)).toEqual(expected)
});

// good
it.each([
    ['', { issue: { users: { nodes: [] } } }],
    ['search term', { issue: { other: { nested: [] } } }],
])('when search term is %p, expect to return %p',
 (searchTerm, expected) => {
    expect(search(searchTerm)).toEqual(expected)
 }
);
```

```
// test suite with tagged template literal block
describe.each`
    status   | icon                 | message
    ${false} | ${'pipeline-failed'} | ${'Pipeline failed - boo-urns'}
    ${true}  | ${'pipeline-passed'} | ${'Pipeline succeeded - win!'}
`('pipeline component', ({ status, icon, message }) => {
    it(`returns icon ${icon} with status ${status}`, () => {
        expect(icon(status)).toEqual(message)
    })

    it(`returns message ${message} with status ${status}`, () => {
        expect(message(status)).toEqual(message)
    })
});
```

Gotchas

RSpec errors due to JavaScript

By default RSpec unit tests don't run JavaScript in the headless browser and rely on inspecting the HTML generated by rails.

If an integration test depends on JavaScript to run correctly, you need to make sure the spec is configured to enable JavaScript when the tests are run. If you don't do this, the spec runner displays vague error messages.

To enable a JavaScript driver in an `RSpec` test, add `:js` to the individual spec or the context block containing multiple specs that need JavaScript enabled:

```
# For one spec
it 'presents information about abuse report', :js do
  # assertions...
end

describe "Admin::AbuseReports", :js do
  it 'presents information about abuse report' do
    # assertions...
  end
  it 'shows buttons for adding to abuse report' do
    # assertions...
  end
end
```

Jest test timeout due to asynchronous imports

If a module asynchronously imports some other modules at runtime, these modules must be transpiled by the Jest loaders at runtime. It's possible that this can cause Jest to timeout.

If you run into this issue, consider eager importing the module so that Jest compiles and caches it at compile-time, fixing the runtime timeout.

Consider the following example:

```
// the_subject.js

export default {
  components: {
    // Async import Thing because it is large and isn't always needed.
    Thing: () => import(/* webpackChunkName: 'thing' */ './path/to/thing.vue'),
  }
};
```

Jest doesn't automatically transpile the `thing.vue` module, and depending on its size, could cause Jest to time out. We can force Jest to transpile and cache this module by eagerly importing it like so:

```
// the_subject_spec.js

import Subject from '~/feature/the_subject.vue';

// Force Jest to transpile and cache
// eslint-disable-next-line no-unused-vars
import _Thing from '~/feature/path/to/thing.vue';
```

NOTE: Do not disregard test timeouts. This could be a sign that there's actually a production problem. Use this opportunity to analyze the production webpack bundles and chunks and confirm that there is not a production issue with the asynchronous imports.

Overview of Frontend Testing Levels

Main information on frontend testing levels can be found in the Testing Levels page.

Tests relevant for frontend development can be found at the following places:

- `spec/frontend/`, for Jest tests
- `spec/features/`, for RSpec tests

RSpec runs complete feature tests, while the Jest directories contain frontend unit tests, frontend component tests, and frontend integration tests.

Before May 2018, `features/` also contained feature tests run by Spinach. These tests were removed from the codebase in May 2018 (#23036).

See also Notes on testing Vue components.

Test helpers

Test helpers can be found in `spec/frontend/__helpers__`. If you introduce new helpers, place them in that directory.

Vuex Helper: `testAction`

We have a helper available to make testing actions easier, as per official documentation:

```
// prefer using like this, a single object argument so parameters are obvious from reading the test
await testAction({
  action: actions.actionName,
  payload: { deleteListId: 1 },
  state: { lists: [1, 2, 3] },
  expectedMutations: [ { type: types.MUTATION} ],
  expectedActions: [],
});

// old way, don't do this for new tests
testAction(
  actions.actionName, // action
  { }, // params to be passed to action
  state, // state
  [
    { type: types.MUTATION},
    { type: types.MUTATION_1, payload: {}},
  ], // mutations committed
  [
    { type: 'actionName', payload: {}},
    { type: 'actionName1', payload: {}},
  ] // actions dispatched
  done,
);
```

Wait until Axios requests finish

The Axios Utils mock module located in `spec/frontend/__helpers__/mocks/axios_utils.js` contains two helper methods for Jest tests that spawn HTTP requests. These are very useful if you don't have a handle to the request's Promise, for example when a Vue component does a request as part of its life cycle.

- `waitFor(url, callback)`: Runs `callback` after a request to `url` finishes (either successfully or unsuccessfully).
- `waitForAll(callback)`: Runs `callback` once all pending requests have finished. If no requests are pending, runs `callback` on the next tick.

Both functions run `callback` on the next tick after the requests finish (using `setImmediate()`), to allow any `.then()` or `.catch()` handlers to run.

`shallowMountExtended` and `mountExtended`

The `shallowMountExtended` and `mountExtended` utilities provide you with the ability to perform any of the available DOM Testing Library queries by prefixing them with `find` or `findAll`.

```
import { shallowMountExtended } from 'helpers/vue_test_utils_helper';

describe('FooComponent', () => {
  const wrapper = shallowMountExtended({
    template: `
      <div data-testid="gitlab-frontend-stack">
        <p>GitLab frontend stack</p>
        <div role="tablist">
          <button role="tab" aria-selected="true">Vue.js</button>
          <button role="tab" aria-selected="false">GraphQL</button>
          <button role="tab" aria-selected="false">SCSS</button>
        </div>
      </div>
    `,
  });

  it('finds elements with `findByTestId`', () => {
    expect(wrapper.findByTestId('gitlab-frontend-stack').exists()).toBe(true);
  });

  it('finds elements with `findByText`', () => {
    expect(wrapper.findByText('GitLab frontend stack').exists()).toBe(true);
    expect(wrapper.findByText('TypeScript').exists()).toBe(false);
  });

  it('finds elements with `findAllByRole`', () => {
    expect(wrapper.findAllByRole('tab').length).toBe(3);
  });
});
```

Check an example in `spec/frontend/alert_management/components/alert_details_spec.js`.

Testing with older browsers

Some regressions only affect a specific browser version. We can install and test in particular browsers with either Firefox or BrowserStack using the following steps:

BrowserStack

BrowserStack allows you to test more than 1200 mobile devices and browsers. You can use it directly through the live app or you can install the chrome extension for easy access. Sign in to BrowserStack with the credentials saved in the **Engineering** vault of the GitLab shared 1Password account.

Firefox

macOS

You can download any older version of Firefox from the releases FTP server, https://ftp.mozilla.org/pub/firefox/releases/:

1. From the website, select a version, in this case `50.0.1`.
2. Go to the mac folder.
3. Select your preferred language. The DMG package is inside. Download it.
4. Drag and drop the application to any other folder but the `Applications` folder.
5. Rename the application to something like `Firefox_Old`.
6. Move the application to the `Applications` folder.
7. Open up a terminal and run `/Applications/Firefox_Old.app/Contents/MacOS/firefox-bin -profilemanager` to create a new profile specific to that Firefox version.
8. Once the profile has been created, quit the app, and run it again like normal. You now have a working older Firefox version.

Snapshots

By now you've probably heard of Jest snapshot tests and why they are useful for various reasons. To use them within GitLab, there are a few guidelines that should be highlighted:

- Treat snapshots as code
- Don't think of a snapshot file as a black box
- Care for the output of the snapshot, otherwise, it's not providing any real value. This will usually involve reading the generated snapshot file as you would read any other piece of code

Think of a snapshot test as a simple way to store a raw `String` representation of what you've put into the item being tested. This can be used to evaluate changes in a component, a store, a complex piece of generated output, etc. You can see more in the list below for some recommended `Do's and Don'ts`. While snapshot tests can be a very powerful tool. They are meant to supplement, not to replace unit tests.

Jest provides a great set of docs on best practices that we should keep in mind when creating snapshots.

How does a snapshot work?

A snapshot is purely a stringified version of what you ask to be tested on the left-hand side of the function call. This means any kind of changes you make to the formatting of the string has an impact on the outcome. This process is done by leveraging serializers for an automatic transform step. For Vue this is already taken care of by leveraging the `vue-jest` package, which offers the proper serializer.

Should the outcome of your spec be different from what is in the generated snapshot file, you'll be notified about it by a failing test in your test suite.

Find all the details in Jests official documentation https://jestjs.io/docs/snapshot-testing

How to take a snapshot

```
it('makes the name look pretty', () => {
  expect(prettifyName('Homer Simpson')).toMatchSnapshot()
})
```

When this test runs the first time a fresh `.snap` file will be created. It will look something like this:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`makes the name look pretty`] = `
Sir Homer Simpson the Third
`
```

Now, every time you call this test, the new snapshot will be evaluated against the previously created version. This should highlight the fact that it's important to understand the content of your snapshot file and treat it with care. Snapshots will lose their value if the output of the snapshot is too big or complex to read, this means keeping snapshots isolated to human-readable items that can be either evaluated in a merge request review or are guaranteed to never change. The same can be done for `wrappers` or `elements`

```
it('renders the component correctly', () => {
  expect(wrapper).toMatchSnapshot()
  expect(wrapper.element).toMatchSnapshot();
})
```

The above test will create two snapshots. It's important to decide which of the snapshots provide more value for codebase safety. That is, if one of these snapshots changes, does that highlight a possible break in the codebase? This can help catch unexpected changes if something in an underlying dependency changes without our knowledge.

Pros and Cons

Pros

- Speed up the creation of unit tests
- Easy to maintain
- Provides a good safety net to protect against accidental breakage of important HTML structures

Cons

- Is not a catch-all solution that replaces the work of integration or unit tests
- No meaningful assertions or expectations within snapshots
- When carelessly used with GitLab UI it can create fragility in tests when the underlying library changes the HTML of a component we are testing

A good guideline to follow: the more complex the component you may want to steer away from just snapshot testing. But that's not to say you can't still snapshot test and test your component as normal.

When to use

Use snapshots when

- to capture a components rendered output
- to fully or partially match templates
- to match readable data structures
- to verify correctly composed native HTML elements
- as a safety net for critical structures so others don't break it by accident

- Template heavy component
- Not a lot of logic in the component
- Composed of native HTML elements

When not to use

Don't use snapshots when

- To capture large data structures just to have something
- To just have some kind of test written
- To capture highly volatile UI elements without stubbing them (Think of GitLab UI version updates)

Get started with feature tests

What is a feature test

A feature test, also known as `white-box testing`, is a test that spawns a browser and has Capybara helpers. This means the test can:

- Locate an element in the browser.
- Click that element.
- Call the API.

Feature tests are expensive to run. You should make sure that you **really want** this type of test before running one.

All of our feature tests are written in `Ruby` but often end up being written by `JavaScript` engineers, as they implement the user-facing feature. So, the following section assumes no prior knowledge of `Ruby` or `Capybara`, and provide a clear guideline on when and how to use these tests.

When to use feature tests

You should use a feature test when the test:

- Is across multiple components.
- Requires that a user navigate across pages.
- Is submitting a form and observing results elsewhere.
- Would result in a huge number of mocking and stubbing with fake data and components if done as a unit test.

Feature tests are especially useful when you want to test:

- That multiple components are working together successfully.
- Complex API interactions. Feature tests interact with the API, so they are slower but do not need any level of mocking or fixtures.

When not to use feature tests

You should use `jest` and `vue-test-utils` unit tests instead of a feature test if you can get the same test results from these methods. Feature tests are quite expensive to run.

You should use a unit test if:

- The behavior you are implementing is all in one component.
- You can simulate other components' behavior to trigger the desired effect.
- You can already select UI elements in the virtual DOM to trigger the desired effects.

Also, if a behavior in your new code needs multiple components to work together, you should consider testing your behavior higher in the component tree. For example, let's say that we have a component called `ParentComponent` with the code:

```
<script>
export default{
  name: ParentComponent,
  data(){
    return {
      internalData: 'oldValue'
    }
  },
   methods:{
    changeSomeInternalData(newVal){
      this.internalData = newVal
    }
   }
}
</script>
<template>
 <div>
  <child-component-1 @child-event="changeSomeInternalData" />
  <child-component-2 :parent-data="internalData" />
 </div>
</template>
```

In this example:

- `ChildComponent1` emits an event.
- `ParentComponent` changes its `internalData` value.
- `ParentComponent` passes the props down to `ChildComponent2`.

You can use a unit test instead by:

- From inside the `ParentComponent` unit test file, emitting the expected event from `childComponent1`
- Making sure the prop is passed down to `childComponent2`.

Then each child component unit tests what happens when the event is emitted and when the prop changes.

This example also applies at larger scale and with deeper component trees. It is definitely worth using unit tests and avoiding the extra cost of feature tests if you can:

- Confidently mount child components.
- Emit events or select elements in the virtual DOM.
- Get the test behavior that you want.

Where to create your test

Feature tests live in `spec/features` folder. You should look for existing files that can test the page you are adding a feature to. Within that folder, you can locate your section. For example, if you wanted to add a new feature test for the pipeline page, you would look in `spec/features/projects/pipelines` and see if the test you want to write exists here.

How to run a feature test

1. Make sure that you have a working GDK environment.
2. Start your `gdk` environment with `gdk start` command.
3. In your terminal, run:

3. In your terminal, run:

```
bundle exec rspec path/to/file:line_of_my_test
```

You can also prefix this command with `WEBDRIVER_HEADLESS=0` which will run the test by opening an actual browser on your computer that you can see, which is very useful for debugging.

How to write a test

Basic file structure

1. Make all string literals unchangeable

In all feature tests, the very first line should be:

```
# frozen_string_literal: true
```

This is in every `Ruby` file and makes all string literals unchangeable. There are also some performance benefits, but this is beyond the scope of this section.

1. Import dependencies.

You should import the modules you need. You will most likely always need to require `spec_helper`:

```
require 'spec_helper'
```

Import any other relevant module.

1. Create a global scope for RSpec to define our tests, just like what we do in jest with the initial describe block.

Then, you need to create the very first `RSpec` scope.

```
RSpec.describe 'Pipeline', :js do
  ...
end
```

What is different though, is that just like everything in Ruby, this is actually a `class`. Which means that right at the top, you can `include` modules that you'd need for your test. For example, you could include the `RoutesHelpers` to navigate more easily.

```
RSpec.describe 'Pipeline', :js do
  include RoutesHelpers
  ...
end
```

After all of this implementation, we have a file that looks something like this:

```
# frozen_string_literal: true

require 'spec_helper'

RSpec.describe 'Pipeline', :js do
  include RoutesHelpers
end
```

Seeding data

Each test is in its own environment and so you must use a factory to seed the required data. To continue on with the pipeline example, let's say that you want a test that takes you to the main pipeline page, which is at the route `/namespace/project/-/pipelines/:id/`.

Most feature tests at least require you to create a user, because you want to be signed in. You can skip this step if you don't have to be signed in, but as a general rule, you should **always create a user unless you are specifically testing a feature looked at by an anonymous user**. This makes sure that you explicitly set a level of permission that you can edit in the test as needed to change or test a new level of permission as the section changes. To create a user:

```
  let(:user) { create(:user) }
```

This creates a variable that holds the newly created user and we can use `create` because we imported the `spec_helper`.

However, we have not done anything with this user yet because it's just a variable. So, in the `before do` block of the spec, we could sign in with the user so that every spec starts with an authenticated user.

```
  let(:user) { create(:user) }

  before do
    sign_in(user)
  end
```

Now that we have a user, we should look at what else we'd need before asserting anything on a pipeline page. If you look at the route `/namespace/project/-/pipelines/:id/` we can determine we need a project and a pipeline.

So we'd create a project and pipeline, and link them together. Usually in factories, the child element requires its parent as an argument. In this case, a pipeline is a child of a project. So we can create the project first, and then when we create the pipeline, we are pass the project as an argument which "binds" the pipeline to the project. A pipeline is also owned by a user, so we need the user as well. For example, this creates a project and a pipeline:

```
  let(:user) { create(:user) }
  let(:project) { create(:project, :repository) }
  let(:pipeline) { create(:ci_pipeline, project: project, ref: 'master', sha: project.commit.id, user: user) }
```

In the same spirit, you could then create a job (build) by using the build factory and passing the parent pipeline:

```
  create(:ci_build, pipeline: pipeline, stage_idx: 10, stage: 'publish', name: 'CentOS')
```

There are many factories that already exists, so make sure to look at other existing files to see if what you need is available.

Navigation

You can navigate to a page by using the `visit` method and passing the path as an argument. Rails automatically generates helper paths, so make sure to use these instead of a hardcoded string. They are generated using the route model, so if we want to go to a pipeline, we'd use:

```
  visit project_pipeline_path(project, pipeline)
```

Before executing any page interaction when navigating or making asynchronous call through the UI, make sure to use `wait_for_requests` before proceeding with further instructions.

Elements interaction

There are a lot of different ways to find and interact with elements. For best practises, refer to the UI testing section.

To click a button, use `click_button` with the string of text found in the button:

```
click_button 'Text inside the button element'
```

If you want to follow a link, then there is `click_link`:

```
click_link 'Text inside the link tag'
```

You can use `fill_in` to fill input / form elements. The first argument is the selector, the second is `with:` which is the value to pass in.

```
fill_in 'current_password', with: '123devops'
```

Alternatively, you can use the `find` selector paired with `send_keys` to add keys in a field without removing previous text, or `set` which completely replaces the value of the input element.

You can find a more comprehensive list of actions in the feature tests actions documentation.

Assertions

To assert anything in a page, you can always access `page` variable, which is automatically defines and actually means the page document. This means you can expect the `page` to have certain components like selectors or content. Here are a few examples:

```
# Finding a button
expect(page).to have_button('Submit review')
```

```
# Finding by text
expect(page).to have_text('build')
```

```
# Finding by `href` value
expect(page).to have_link(pipeline.ref)
```

```
# Find by data-testid
# Like CSS selector, this is acceptable when there isn't a specific matcher available.
expect(page).to have_css('[data-testid="pipeline-multi-actions-dropdown"]')
```

```
# Finding by CSS selector. This is a last resort.
# For example, when you cannot add attributes on the desired element.
expect(page).to have_css('.js-icon-retry')
```

```
# You can combine any of these selectors with `not_to` instead
expect(page).not_to have_button('Submit review')
```

```
# When a test case has back to back expectations,
# it is recommended to group them using `:aggregate_failures`
it 'shows the issue description and design references', :aggregate_failures do
  expect(page).to have_text('The designs I mentioned')
  expect(page).to have_link(design_tab_ref)
  expect(page).to have_link(design_ref_a)
  expect(page).to have_link(design_ref_b)
end
```

You can also create a sub-block to look into, to:

- Scope down where you are making your assertions and reduce the risk of finding another element that was not intended.
- Make sure an element is found within the right boundaries.

```
page.within('[data-testid="pipeline-multi-actions-dropdown"]') do
  ...
end
```

You can find a more comprehensive list of matchers in the feature tests matchers documentation.

Feature flags

By default, every feature flag is enabled **regardless of the YAML definition or the flags you've set manually in your GDK**. To test when a feature flag is disabled, you must manually stub the flag, ideally in a `before do` block.

```
stub_feature_flags(my_feature_flag: false)
```

If you are stubbing an `ee` feature flag, then use:

```
stub_licensed_features(my_feature_flag: false)
```

Debugging

You can run your spec with the prefix `WEBDRIVER_HEADLESS=0` to open an actual browser. However, the specs goes though the commands quickly and leaves you no time to look around.

To avoid this problem, you can write `binding.pry` on the line where you want Capybara to stop execution. You are then inside the browser with normal usage. To understand why you cannot find certain elements, you can:

- Select elements.
- Use the console and network tab.
- Execute selectors inside the browser console.

Inside the terminal, where capybara is running, you can also execute `next` which goes line by line through the test. This way you can check every single interaction one by one to see what might be causing an issue.

Updating ChromeDriver

On MacOS, if you get a ChromeDriver error, make sure to update it by running

```
brew upgrade chromedriver
```

Return to Testing documentation