MUI Testing

Thanks for writing tests! Here's a quick run-down on our current setup.

Getting started

1. Add a unit test to `packages/*/src/TheUnitInQuestion/TheUnitInQuestion.test.js` or an integration test `packages/*/test/`.
2. Run `yarn t TheUnitInQuestion`.
3. Implement the tested behavior
4. Open a PR once the test passes or you want somebody to review your work

Tools we use

- @testing-library/react
- Chai
- Sinon
- Mocha
- Karma
- Playwright
- jsdom
- enzyme (old tests only)

Writing tests

For all unit tests, please use the return value from `test/utils/createRenderer`. It prepares the test suite and returns a function with the same interface as `render` from `@testing-library/react`.

```
describe('test suite', () => {
  const { render } = createRenderer();

  test('first', () => {
    render(<input />);
  });
});
```

For new tests please use `expect` from the BDD testing approach. Prefer to use as expressive matchers as possible. This keeps the tests readable, and, more importantly, the message if they fail as descriptive as possible.

In addition to the core matchers from `chai` we also use matchers from `chai-dom`.

Deciding where to put a test is (like naming things) a hard problem:

- When in doubt, put the new test case directly in the unit test file for that component e.g. `packages/mui-material/src/Button/Button.test.js`.

- If your test requires multiple components from the library create a new integration test.
- If you find yourself using a lot of `data-testid` attributes or you're accessing a lot of styles consider adding a component (that doesn't require any interaction) to `test/regressions/tests/` e.g. `test/regressions/tests/List/ListWithSomeStyleProp`
- If you have to dispatch and compose many different DOM events prefer end-to-end tests (Checkout the end-to-end testing readme for more information.)

Unexpected calls to `console.error` or `console.warn`

By default, our test suite fails if any test recorded `console.error` or `console.warn` calls that are unexpected.

The failure message includes the full test name (suite names + test name). This should help locating the test in case the top of the stack can't be read due to excessive error messages. The error includes the logged message as well as the stacktrace of that message.

You can explicitly expect no console calls for when you're adding a regression test. This makes the test more readable and properly fails the test in watchmode if the test had unexpected `console` calls.

Writing a test for `console.error` or `console.warn`

If you add a new warning via `console.error` or `console.warn` you should add tests that expect this message. For tests that expect a call you can use our custom `toWarnDev` or `toErrorDev` matchers. The expected messages must be a subset of the actual messages and match the casing. The order of these messages must match as well.

Example:

```
function SomeComponent({ variant }) {
  if (process.env.NODE_ENV !== 'production') {
    if (variant === 'unexpected') {
      console.error("That variant doesn't make sense.");
    }
    if (variant !== undefined) {
      console.error('`variant` is deprecated.');
    }
  }

  return <div />;
}
expect(() => {
  render(<SomeComponent variant="unexpected" />);
}).toErrorDev(["That variant doesn't make sense.", '`variant` is deprecated.']);
```

```
function SomeComponent({ variant }) {
  if (process.env.NODE_ENV !== 'production') {
    if (variant === 'unexpected') {
      console.error("That variant doesn't make sense.");
    }
    if (variant !== undefined) {
      console.error('`variant` is deprecated.');
    }
  }

  return <div />;
}
expect(() => {
  render(<SomeComponent />);
}).not.toErrorDev();
```

Commands

MUI uses a wide range of tests approach as each of them comes with a different trade-off, mainly completeness vs. speed.

React API level

Debugging tests If you want to debug tests with the e.g. Chrome inspector (chrome://inspect) you can run `yarn t <testFilePattern> --debug`. Note that the test will not get executed until you start code execution in the inspector.

We have a dedicated task to use VSCode's integrated debugger to debug the currently opened test file. Open the test you want to run and press F5 (launch "Test Current File").

Run the core mocha unit/integration test suite To run all of the unit and integration tests run `yarn test:unit`

If you want to `grep` for certain tests add `-g STRING_TO_GREP` though for development we recommend `yarn t <testFilePattern>`.

Watch the core mocha unit/integration test suite `yarn t <testFilePattern>`

First, we have the **unit test** suite. It uses mocha and a thin wrapper around `@testing-library/react`. Here is an example with the `Dialog` component.

Next, we have the **integration** tests. They are mostly used for components that act as composite widgets like `Select` or `Menu`. Here is an example with the `Menu` component.

Create HTML coverage reports `yarn test:coverage:html`

When running this command you should get under `coverage/index.html` a full coverage report in HTML format. This is created using Istanbul's HTML reporter and gives good data such as line, branch and function coverage.

DOM API level

Run the mocha test suite using the karma runner `yarn test:karma`

Testing the components at the React level isn't enough; we need to make sure they will behave as expected with a **real DOM**. To solve that problem we use karma, which is almost a drop-in replacement of jsdom. Our tests run on different browsers to increase the coverage:

- Headless Chrome
- Chrome, Firefox, Safari, and Edge thanks to BrowserStack

BrowserStack We only use BrowserStack for non-PR commits to save ressources. BrowserStack rarely reports actual issues so we only use it as a stop-gap for releases not merges.

To force a run of BrowserStack on a PR you have to run the pipeline with `browserstack-force` set to `true`. For example, you've opened a PR with the number 64209 and now after everything is green you want to make sure the change passes all browsers:

```
curl --request POST \
  --url https://circleci.com/api/v2/project/gh/mui/material-ui/pipeline \
  --header 'content-type: application/json' \
  --header 'Circle-Token: $CIRCLE_TOKEN' \
  --data-raw '{"branch":"pull/64209/head","parameters":{"browserstack-force":true}}'
```

Browser API level

In the end, components are going to be used in a real browser. The DOM is just one dimension of that environment, so we also need to take into account the rendering engine.

Visual regression tests Check out the visual regression testing readme for more information.

end-to-end tests Checkout the end-to-end testing readme for more information.

Development When working on the visual regression tests you can run `yarn test:regressions:dev` in the background to constantly rebuild the views used for visual regression testing. To actually take the screenshots you can then run `yarn test:regressions:run`. You can pass the same arguments as you could to `mocha`. For example, `yarn test:regressions:run --watch --grep "docs-system-basic"` to take new screenshots of every demo in `docs/src/pages/system/basic`. You can view the screenshots in `test/regressions/screenshots/chrome`.

Alternatively, you might want to open `http://localhost:3000` (while `yarn test:regressions:dev` is running) to view individual views separately.

Caveats

Accessibility tree exclusion Our tests also explicitly document which parts of the queried element are included in the accessibility (a11y) tree and which are excluded. This check is fairly expensive which is why it is disabled when tests are run locally by default. The rationale being that in almost all cases including or excluding elements from a query-set depending on their a11y-tree membership makes no difference.

The queries where this does make a difference explicitly include checking for a11y tree inclusion e.g. `getByRole('button', { hidden: false })` (see byRole documentation for more information). To see if your test (`test:karma` or `test:unit`) behaves the same between CI and local environment, set the environment variable `CI` to `'true'`.

Not considering a11y tree exclusion is a common cause of "Unable to find an accessible element with the role" or "Found multiple elements with the role".

Performance monitoring

We have a dedicated CI task that profiles our core test suite. Since this task is fairly expensive and not relevant to most day-to-day work it has to be started manually. The CircleCI docs explain how to start a pipeline manually in detail. Example: With an environment variable `$CIRCLE_TOKEN` containing a CircleCI personal access token.

The following command triggers the `profile` workflow for the pull request #24289.

```
curl --request POST \
  --url https://circleci.com/api/v2/project/gh/mui/material-ui/pipeline \
  --header 'content-type: application/json' \
  --header 'Circle-Token: $CIRCLE_TOKEN' \
  --data-raw '{"branch":"pull/24289/head","parameters":{"workflow":"profile"}}'
```

To analyze this profile run you can use https://mui-dashboard.netlify.app/test-profile/:job-number.

To find out the job number you can start with the response of the previous CircleCI API request which includes the created pipeline id. You then have to search in the CircleCI UI for the job number of `test_profile` that is part of the started pipeline. The job number can be extracted from the URL of a particular CircleCI job.

For example, in https://app.circleci.com/pipelines/github/mui/material-ui/32796/workflows/23f946de-328e-49b7-9c94-bfe0a0248a12/jobs/211258 `jobs/211258` points to the job number which is in this case `211258` which means you want to visit https://mui-dashboard.netlify.app/test-profile/211258 to analyze the profile.

Testing multiple versions of React

You can check integration of different versions of React (e.g. different release channels or PRs to React) by running `node scripts/use-react-dist-tag <dist-tag>`.

Possible values for `dist-tag`:

- default: `stable` (minimum supported React version)
- a tag on npm e.g. `next`, `experimental` or `latest`

CI You can pass the same `dist-tag` to our CircleCI pipeline as well:

With the following API request we're triggering a run of the default workflow in PR #24289 for `react@next`

```
curl --request POST \
  --url https://circleci.com/api/v2/project/gh/mui/material-ui/pipeline \
  --header 'content-type: application/json' \
  --header 'Circle-Token: $CIRCLE_TOKEN' \
  --data-raw '{"branch":"pull/24289/head","parameters":{"react-dist-tag":"next"}}'
```