- Integrations
- Supported languages

-
- Guides
- Accessibility testing

On this page

Accessibility testing

Introduction

Playwright can be used to test your application for many types of accessibility issues.

A few examples of problems this can catch include:

- Text that would be hard to read for users with vision impairments due to poor color contrast with the background behind it
- UI controls and form elements without labels that a screen reader could identify
- Interactive elements with duplicate IDs which can confuse assistive technologies

The following examples rely on the `@axe-core/playwright` package which adds support for running the axe accessibility testing engine as part of your Playwright tests.

DISCLAIMER

Automated accessibility tests can detect some common accessibility problems such as missing or invalid properties. But many accessibility problems can only be discovered through manual testing. We recommend using a combination of automated testing, manual accessibility assessments, and inclusive user testing. For manual assessments, we recommend Accessibility Insights for Web, a free and open source dev tool that walks you through assessing a website for WCAG 2.1 AA coverage.

Example accessibility tests

Accessibility tests work just like any other Playwright test. You can either create separate test cases for them, or integrate accessibility scans and assertions into your existing test cases.

The following examples demonstrate a few basic accessibility testing scenarios.

Scanning an entire page

This example demonstrates how to test an entire page for automatically detectable accessibility violations. The test:

1. Imports the `@axe-core/playwright` package
2. Uses normal Playwright Test syntax to define a test case

3. Uses normal Playwright syntax to navigate to the page under test
4. Awaits `AxeBuilder.analyze()` to run the accessibility scan against the page
5. Uses normal Playwright Test assertions to verify that there are no violations in the returned scan results

- TypeScript
- JavaScript

```
import { test, expect } from '@playwright/test';
import AxeBuilder from '@axe-core/playwright'; // 1

test.describe('homepage', () => { // 2
  test('should not have any automatically detectable accessibility issues',
async ({ page }) => {
    await page.goto('https://your-site.com/'); // 3

    const accessibilityScanResults = await new AxeBuilder({ page
}).analyze(); // 4

    expect(accessibilityScanResults.violations).toEqual([]); // 5
  });
});
const { test, expect } = require('@playwright/test');
const AxeBuilder = require('@axe-core/playwright').default; // 1

test.describe('homepage', () => { // 2
  test('should not have any automatically detectable accessibility issues',
async ({ page }) => {
    await page.goto('https://your-site.com/'); // 3

    const accessibilityScanResults = await new AxeBuilder({ page
}).analyze(); // 4

    expect(accessibilityScanResults.violations).toEqual([]); // 5
  });
});
```

Configuring axe to scan a specific part of a page

`@axe-core/playwright` supports many configuration options for axe. You can specify these options by using a Builder pattern with the `AxeBuilder` class.

For example, you can use `AxeBuilder.include()` to constrain an accessibility scan to only run against one specific part of a page.

`AxeBuilder.analyze()` will scan the page *in its current state* when you call it. To scan parts of a page that are revealed based on UI interactions, use Locators to interact with the page before invoking `analyze()`:

```
test('navigation menu should not have automatically detectable
accessibility violations', async ({
  page,
}) => {
  await page.goto('https://your-site.com/');

  await page.getByRole('button', { name: 'Navigation Menu' }).click();
```

```
  // It is important to waitFor() the page to be in the desired
  // state *before* running analyze(). Otherwise, axe might not
  // find all the elements your test expects it to scan.
  await page.locator('#navigation-menu-flyout').waitFor();

  const accessibilityScanResults = await new AxeBuilder({ page })
      .include('#navigation-menu-flyout')
      .analyze();

  expect(accessibilityScanResults.violations).toEqual([]);
});
```

Scanning for WCAG violations

By default, axe checks against a wide variety of accessibility rules. Some of these rules correspond to specific success criteria from the Web Content Accessibility Guidelines (WCAG), and others are "best practice" rules that are not specifically required by any WCAG criterion.

You can constrain an accessibility scan to only run those rules which are "tagged" as corresponding to specific WCAG success criteria by using `AxeBuilder.withTags()`. For example, Accessibility Insights for Web's Automated Checks only include axe rules that test for violations of WCAG A and AA success criteria; to match that behavior, you would use the tags `wcag2a`, `wcag2aa`, `wcag21a`, and `wcag21aa`.

Note that automated testing cannot detect all types of WCAG violations.

```
test('should not have any automatically detectable WCAG A or AA
violations', async ({ page }) => {
  await page.goto('https://your-site.com/');

  const accessibilityScanResults = await new AxeBuilder({ page })
      .withTags(['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa'])
      .analyze();

  expect(accessibilityScanResults.violations).toEqual([]);
});
```

You can find a complete listing of the rule tags axe-core supports in the "Axe-core Tags" section of the axe API documentation.

Handling known issues

A common question when adding accessibility tests to an application is "how do I suppress known violations?" The following examples demonstrate a few techniques you can use.

Excluding individual elements from a scan

If your application contains a few specific elements with known issues, you can use `AxeBuilder.exclude()` to exclude them from being scanned until you're able to fix the issues.

This is usually the simplest option, but it has some important downsides:

- `exclude()` will exclude the specified elements *and all of their descendants*. Avoid using it with components that contain many children.
- `exclude()` will prevent *all* rules from running against the specified elements, not just the rules corresponding to known issues.

Here is an example of excluding one element from being scanned in one specific test:

```
test('should not have any accessibility violations outside of elements with
known issues', async ({
  page,
}) => {
  await page.goto('https://your-site.com/page-with-known-issues');

  const accessibilityScanResults = await new AxeBuilder({ page })
      .exclude('#element-with-known-issue')
      .analyze();

  expect(accessibilityScanResults.violations).toEqual([]);
});
```

If the element in question is used repeatedly in many pages, consider using a test fixture to reuse the same `AxeBuilder` configuration across multiple tests.

Disabling individual scan rules

If your application contains many different preexisting violations of a specific rule, you can use `AxeBuilder.disableRules()` to temporarily disable individual rules until you're able to fix the issues.

You can find the rule IDs to pass to `disableRules()` in the `id` property of the violations you want to suppress. A complete list of axe's rules can be found in `axe-core`'s documentation.

```
test('should not have any accessibility violations outside of rules with
known issues', async ({
  page,
}) => {
  await page.goto('https://your-site.com/page-with-known-issues');

  const accessibilityScanResults = await new AxeBuilder({ page })
      .disableRules(['duplicate-id'])
      .analyze();

  expect(accessibilityScanResults.violations).toEqual([]);
});
```

Using snapshots to allow specific known issues

If you would like to allow for a more granular set of known issues, you can use Snapshots to verify that a set of preexisting violations has not changed. This approach avoids the downsides of using `AxeBuilder.exclude()` at the cost of slightly more complexity and fragility.

Do not use a snapshot of the entire `accessibilityScanResults.violations` array. It contains implementation details of the elements in question, such as a snippet of their rendered HTML; if you include these in your snapshots, it will make your tests prone to breaking every time one of the components in question changes for an unrelated reason:

```
// Don't do this! This is fragile.
expect(accessibilityScanResults.violations).toMatchSnapshot();
```

Instead, create a *fingerprint* of the violation(s) in question that contains only enough information to uniquely identify the issue, and use a snapshot of the fingerprint:

```
// This is less fragile than snapshotting the entire violations array.
expect(violationFingerprints(accessibilityScanResults)).toMatchSnapshot();

// my-test-utils.js
function violationFingerprints(accessibilityScanResults) {
  const violationFingerprints =
accessibilityScanResults.violations.map(violation => ({
    rule: violation.id,
    // These are CSS selectors which uniquely identify each element with
    // a violation of the rule in question.
    targets: violation.nodes.map(node => node.target),
  }));

  return JSON.stringify(violationFingerprints, null, 2);
}
```

Exporting scan results as a test attachment

Most accessibility tests are primarily concerned with the `violations` property of the axe scan results. However, the scan results contain more than just `violations`. For example, the results also contain information about rules which passed and about elements which axe found to have inconclusive results for some rules. This information can be useful for debugging tests that aren't detecting all the violations you expect them to.

To include *all* of the scan results as part of your test results for debugging purposes, you can add the scan results as a test attachment with `testInfo.attach()`. Reporters can then embed or link the full results as part of your test output.

The following example demonstrates attaching scan results to a test:

```
test('example with attachment', async ({ page }, testInfo) => {
  await page.goto('https://your-site.com/');

  const accessibilityScanResults = await new AxeBuilder({ page
}).analyze();

  await testInfo.attach('accessibility-scan-results', {
    body: JSON.stringify(accessibilityScanResults, null, 2),
    contentType: 'application/json'
  });

  expect(accessibilityScanResults.violations).toEqual([]);
});
```

Using a test fixture for common axe configuration

Test fixtures are a good way to share common `AxeBuilder` configuration across many tests. Some scenarios where this might be useful include:

- Using a common set of rules among all of your tests
- Suppressing a known violation in a common element which appears in many different pages
- Attaching standalone accessibility reports consistently for many scans

The following example demonstrates creating and using a test fixture that covers each of those scenarios.

Creating a fixture

This example fixture creates an `AxeBuilder` object which is pre-configured with shared `withTags()` and `exclude()` configuration.

- TypeScript
- JavaScript

axe-test.ts

```
import { test as base } from '@playwright/test';
import AxeBuilder from '@axe-core/playwright';

type AxeFixture = {
  makeAxeBuilder: () => AxeBuilder;
};

// Extend base test by providing "makeAxeBuilder"
//
// This new "test" can be used in multiple test files, and each of them
will get
// a consistently configured AxeBuilder instance.
export const test = base.extend<AxeFixture>({
  makeAxeBuilder: async ({ page }, use, testInfo) => {
    const makeAxeBuilder = () => new AxeBuilder({ page })
        .withTags(['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa'])
        .exclude('#commonly-reused-element-with-known-issue');

    await use(makeAxeBuilder);
  }
});
export { expect } from '@playwright/test';
// axe-test.js
const base = require('@playwright/test');
const AxeBuilder = require('@axe-core/playwright').default;

// Extend base test by providing "makeAxeBuilder"
//
// This new "test" can be used in multiple test files, and each of them
will get
// a consistently configured AxeBuilder instance.
exports.test = base.test.extend({
  makeAxeBuilder: async ({ page }, use, testInfo) => {
```

```
    const makeAxeBuilder = () => new AxeBuilder({ page })
        .withTags(['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa'])
        .exclude('#commonly-reused-element-with-known-issue');

    await use(makeAxeBuilder);
  }
});
exports.expect = base.expect;
```

Using a fixture

To use the fixture, replace the earlier examples' `new AxeBuilder({ page })` with the newly defined `makeAxeBuilder` fixture:

```
const { test, expect } = require('./axe-test');

test('example using custom fixture', async ({ page, makeAxeBuilder }) => {
  await page.goto('https://your-site.com/');

  const accessibilityScanResults = await makeAxeBuilder()
      // Automatically uses the shared AxeBuilder configuration,
      // but supports additional test-specific configuration too
      .include('#specific-element-under-test')
      .analyze();

  expect(accessibilityScanResults.violations).toEqual([]);
});
```

Previous
Library

Next
Actions

Learn

- Getting started
- Playwright Training
- Learn Videos

- Feature Videos

Community

- Stack Overflow
- Discord
- Twitter
- LinkedIn

More

- GitHub
- YouTube
- Blog
- Ambassadors