- Integrations
- Supported languages

-
- Guides
- Components (experimental)

On this page

Components (experimental)

Introduction

Playwright Test can now test your components.

Example

Here is what a typical component test looks like:

```
test('event should work', async ({ mount }) => {
  let clicked = false;

  // Mount a component. Returns locator pointing to the component.
  const component = await mount(
    <Button title="Submit" onClick={() => { clicked = true }}></Button>
  );

  // As with any Playwright test, assert locator text.
  await expect(component).toContainText('Submit');

  // Perform locator click. This will trigger the event.
  await component.click();

  // Assert that respective events have been fired.
  expect(clicked).toBeTruthy();
});
```

How to get started

Adding Playwright Test to an existing project is easy. Below are the steps to enable Playwright Test for a React, Vue, Svelte or Solid project.

Step 1: Install Playwright Test for components for your respective framework

- npm
- yarn
- pnpm

```
npm init playwright@latest -- --ct
```

```
yarn create playwright --ct
pnpm create playwright --ct
```

This step creates several files in your workspace:

playwright/index.html

```
<html lang="en">
  <body>
    <div id="root"></div>
    <script type="module" src="./index.ts"></script>
  </body>
</html>
```

This file defines an html file that will be used to render components during testing. It must contain element with `id="root"`, that's where components are mounted. It must also link the script called `playwright/index.{js,ts,jsx,tsx}`.

You can include stylesheets, apply theme and inject code into the page where component is mounted using this script. It can be either a `.js`, `.ts`, `.jsx` or `.tsx` file.

playwright/index.ts

```
// Apply theme here, add anything your component needs at runtime here.
```

Step 2. Create a test file `src/App.spec.{ts,tsx}`

- React
- Solid
- Svelte
- Vue

```
import { test, expect } from '@playwright/experimental-ct-react';
import App from './App';

test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(<App />);
  await expect(component).toContainText('Learn React');
});
import { test, expect } from '@playwright/experimental-ct-vue';
import App from './App.vue';

test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(App);
  await expect(component).toContainText('Vite + Vue');
});
```

If using TypeScript and Vue make sure to add a `vue.d.ts` file to your project:

```
declare module '*.vue';
import { test, expect } from '@playwright/experimental-ct-svelte';
import App from './App.svelte';
```

```
test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(App);
  await expect(component).toContainText('Vite + Svelte');
});
import { test, expect } from '@playwright/experimental-ct-solid';
import App from './App';

test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(<App />);
  await expect(component).toContainText('Learn Solid');
});
```

Step 3. Run the tests

You can run tests using the VS Code extension or the command line.

```
npm run test-ct
```

Further reading: configure reporting, browsers, tracing

Refer to Playwright config for configuring your project.

Test stories

When Playwright Test is used to test web components, tests run in Node.js, while components run in the real browser. This brings together the best of both worlds: components run in the real browser environment, real clicks are triggered, real layout is executed, visual regression is possible. At the same time, test can use all the powers of Node.js as well as all the Playwright Test features. As a result, the same parallel, parametrized tests with the same post-mortem Tracing story are available during component testing.

This however, is introducing a number of limitations:

- You can't pass complex live objects to your component. Only plain JavaScript objects and built-in types like strings, numbers, dates etc. can be passed.

```
test('this will work', async ({ mount }) => {
  const component = await mount(<ProcessViewer process={{ name:
'playwright' }}/>);
});

test('this will not work', async ({ mount }) => {
  // `process` is a Node object, we can't pass it to the browser and expect
it to work.
  const component = await mount(<ProcessViewer process={process}/>);
});
```

- You can't pass data to your component synchronously in a callback:

```
test('this will not work', async ({ mount }) => {
  // () => 'red' callback lives in Node. If `ColorPicker` component in the
browser calls the parameter function
  // `colorGetter` it won't get result synchronously. It'll be able to get
it via await, but that is not how
  // components are typically built.
  const component = await mount(<ColorPicker colorGetter={() => 'red'}/>);
});
```

Working around these and other limitations is quick and elegant: for every use case of the tested component, create a wrapper of this component designed specifically for test. Not only it will mitigate the limitations, but it will also offer powerful abstractions for testing where you would be able to define environment, theme and other aspects of your component rendering.

Let's say you'd like to test following component:

input-media.tsx

```
import React from 'react';

type InputMediaProps = {
  // Media is a complex browser object we can't send to Node while testing.
  onChange(media: Media): void;
};

export function InputMedia(props: InputMediaProps) {
  return <></> as any;
}
```

Create a story file for your component:

input-media.story.tsx

```
import React from 'react';
import InputMedia from './import-media';

type InputMediaForTestProps = {
  onMediaChange(mediaName: string): void;
};

export function InputMediaForTest(props: InputMediaForTestProps) {
  // Instead of sending a complex `media` object to the test, send the
media name.
  return <InputMedia onChange={media => props.onMediaChange(media.name)}
/>;
}
// Export more stories here.
```

Then test the component via testing the story:

input-media.test.spec.tsx

```
test('changes the image', async ({ mount }) => {
  let mediaSelected: string | null = null;

  const component = await mount(
    <InputMediaForTest
      onMediaChange={mediaName => {
        mediaSelected = mediaName;
```

```
      }}
    />
  );
  await component
    .getByTestId('imageInput')
    .setInputFiles('src/assets/logo.png');

  await expect(component.getByAltText(/selected image/i)).toBeVisible();
  await expect.poll(() => mediaSelected).toBe('logo.png');
});
```

As a result, for every component you'll have a story file that exports all the stories that are actually tested. These stories live in the browser and "convert" complex object into the simple objects that can be accessed in the test.

Under the hood

Here is how component testing works:

- Once the tests are executed, Playwright creates a list of components that the tests need.
- It then compiles a bundle that includes these components and serves it using a local static web server.
- Upon the `mount` call within the test, Playwright navigates to the facade page `/playwright/index.html` of this bundle and tells it to render the component.
- Events are marshalled back to the Node.js environment to allow verification.

Playwright is using Vite to create the components bundle and serve it.

API reference

props

Provide props to a component when mounted.

- React
- Solid
- Svelte
- Vue

```
test('props', async ({ mount }) => {
  const component = await mount(<Component msg="greetings" />);
});
test('props', async ({ mount }) => {
  const component = await mount(<Component msg="greetings" />);
});
test('props', async ({ mount }) => {
  const component = await mount(Component, { props: { msg: 'greetings' }
});
});
test('props', async ({ mount }) => {
  const component = await mount(Component, { props: { msg: 'greetings' }
```

```
});
});
```

callbacks / events

Provide callbacks/events to a component when mounted.

- React
- Solid
- Svelte
- Vue

```
test('callback', async ({ mount }) => {
  const component = await mount(<Component callback={() => {}} />);
});
test('callback', async ({ mount }) => {
  const component = await mount(<Component callback={() => {}} />);
});
test('event', async ({ mount }) => {
  const component = await mount(Component, { on: { callback() {} } });
});
test('event', async ({ mount }) => {
  const component = await mount(Component, { on: { callback() {} } });
});
```

children / slots

Provide children/slots to a component when mounted.

- React
- Solid
- Svelte
- Vue

```
test('children', async ({ mount }) => {
  const component = await mount(<Component>Child</Component>);
});
test('children', async ({ mount }) => {
  const component = await mount(<Component>Child</Component>);
});
test('slot', async ({ mount }) => {
  const component = await mount(Component, { slots: { default: 'Slot' } });
});
test('slot', async ({ mount }) => {
  const component = await mount(Component, { slots: { default: 'Slot' } });
});
```

hooks

You can use `beforeMount` and `afterMount` hooks to configure your app. This lets you set up things like your app router, fake server etc. giving you the flexibility you need. You can also pass custom configuration from the `mount` call from a test, which is accessible from the `hooksConfig` fixture. This includes any config that needs to be run before or after mounting the component. An example of configuring a router is provided below:

- React
- Solid
- Vue3
- Vue2

playwright/index.tsx

```
import { beforeMount, afterMount } from '@playwright/experimental-ct-
react/hooks';
import { BrowserRouter } from 'react-router-dom';

export type HooksConfig = {
  enableRouting?: boolean;
}

beforeMount<HooksConfig>(async ({ App, hooksConfig }) => {
  if (hooksConfig?.enableRouting)
    return <BrowserRouter><App /></BrowserRouter>;
});
```

src/pages/ProductsPage.spec.tsx

```
import { test, expect } from '@playwright/experimental-ct-react';
import type { HooksConfig } from '../playwright';
import { ProductsPage } from './pages/ProductsPage';

test('configure routing through hooks config', async ({ page, mount }) => {
  const component = await mount<HooksConfig>(<ProductsPage />, {
    hooksConfig: { enableRouting: true },
  });
  await expect(component.getByRole('link')).toHaveAttribute('href',
'/products/42');
});
```

playwright/index.tsx

```
import { beforeMount, afterMount } from '@playwright/experimental-ct-
solid/hooks';
import { Router } from '@solidjs/router';

export type HooksConfig = {
  enableRouting?: boolean;
}

beforeMount<HooksConfig>(async ({ App, hooksConfig }) => {
  if (hooksConfig?.enableRouting)
    return <Router><App /></Router>;
});
```

src/pages/ProductsPage.spec.tsx

```
import { test, expect } from '@playwright/experimental-ct-solid';
import type { HooksConfig } from '../playwright';
import { ProductsPage } from './pages/ProductsPage';

test('configure routing through hooks config', async ({ page, mount }) => {
  const component = await mount<HooksConfig>(<ProductsPage />, {
    hooksConfig: { enableRouting: true },
  });
  await expect(component.getByRole('link')).toHaveAttribute('href',
'/products/42');
});
```

playwright/index.ts

```
import { beforeMount, afterMount } from '@playwright/experimental-ct-
vue/hooks';
import { router } from '../src/router';
```

```
export type HooksConfig = {
  enableRouting?: boolean;
}

beforeMount<HooksConfig>(async ({ app, hooksConfig }) => {
  if (hooksConfig?.enableRouting)
    app.use(router);
});
```

src/pages/ProductsPage.spec.ts

```
import { test, expect } from '@playwright/experimental-ct-vue';
import type { HooksConfig } from '../playwright';
import ProductsPage from './pages/ProductsPage.vue';

test('configure routing through hooks config', async ({ page, mount }) => {
  const component = await mount<HooksConfig>(ProductsPage, {
    hooksConfig: { enableRouting: true },
  });
  await expect(component.getByRole('link')).toHaveAttribute('href',
'/products/42');
});
```

playwright/index.ts

```
import { beforeMount, afterMount } from '@playwright/experimental-ct-
vue2/hooks';
import Router from 'vue-router';
import { router } from '../src/router';

export type HooksConfig = {
  enableRouting?: boolean;
}

beforeMount<HooksConfig>(async ({ app, hooksConfig }) => {
  if (hooksConfig?.enableRouting) {
    Vue.use(Router);
    return { router }
  }
});
```

src/pages/ProductsPage.spec.ts

```
import { test, expect } from '@playwright/experimental-ct-vue2';
import type { HooksConfig } from '../playwright';
import ProductsPage from './pages/ProductsPage.vue';

test('configure routing through hooks config', async ({ page, mount }) => {
  const component = await mount<HooksConfig>(ProductsPage, {
    hooksConfig: { enableRouting: true },
  });
  await expect(component.getByRole('link')).toHaveAttribute('href',
'/products/42');
});
```

unmount

Unmount the mounted component from the DOM. This is useful for testing the component's behavior upon unmounting. Use cases include testing an "Are you sure you want to leave?" modal or ensuring proper cleanup of event handlers to prevent memory leaks.

- React
- Solid

- Svelte
- Vue

```
test('unmount', async ({ mount }) => {
  const component = await mount(<Component/>);
  await component.unmount();
});
test('unmount', async ({ mount }) => {
  const component = await mount(<Component/>);
  await component.unmount();
});
test('unmount', async ({ mount }) => {
  const component = await mount(Component);
  await component.unmount();
});
test('unmount', async ({ mount }) => {
  const component = await mount(Component);
  await component.unmount();
});
```

update

Update props, slots/children, and/or events/callbacks of a mounted component. These component inputs can change at any time and are typically provided by the parent component, but sometimes it is necessary to ensure that your components behave appropriately to new inputs.

- React
- Solid
- Svelte
- Vue

```
test('update', async ({ mount }) => {
  const component = await mount(<Component/>);
  await component.update(
      <Component msg="greetings" callback={() => {}}>Child</Component>
  );
});
test('update', async ({ mount }) => {
  const component = await mount(<Component/>);
  await component.update(
      <Component msg="greetings" callback={() => {}}>Child</Component>
  );
});
test('update', async ({ mount }) => {
  const component = await mount(Component);
  await component.update({
    props: { msg: 'greetings' },
    on: { callback: () => {} },
    slots: { default: 'Child' }
  });
});
test('update', async ({ mount }) => {
  const component = await mount(Component);
  await component.update({
    props: { msg: 'greetings' },
    on: { callback: () => {} },
```

```
    slots: { default: 'Child' }
  });
});
```

Handling network requests

Playwright provides an **experimental** `router` fixture to intercept and handle network requests. There are two ways to use the `router` fixture:

- Call `router.route(url, handler)` that behaves similarly to page.route(). See the network mocking guide for more details.
- Call `router.use(handlers)` and pass MSW library request handlers to it.

Here is an example of reusing your existing MSW handlers in the test.

```
import { handlers } from '@src/mocks/handlers';

test.beforeEach(async ({ router }) => {
  // install common handlers before each test
  await router.use(...handlers);
});

test('example test', async ({ mount }) => {
  // test as usual, your handlers are active
  // ...
});
```

You can also introduce a one-off handler for a specific test.

```
import { http, HttpResponse } from 'msw';

test('example test', async ({ mount, router }) => {
  await router.use(http.get('/data', async ({ request }) => {
    return HttpResponse.json({ value: 'mocked' });
  }));

  // test as usual, your handler is active
  // ...
});
```

Frequently asked questions

What's the difference between `@playwright/test` and `@playwright/experimental-ct-{react,svelte,vue,solid}`?

```
test('…', async ({ mount, page, context }) => {
  // …
});
```

`@playwright/experimental-ct-{react,svelte,vue,solid}` wrap `@playwright/test` to provide an additional built-in component-testing specific fixture called `mount`:

- React
- Solid

- Svelte
- Vue

```
import { test, expect } from '@playwright/experimental-ct-react';
import HelloWorld from './HelloWorld';

test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(<HelloWorld msg="greetings" />);
  await expect(component).toContainText('Greetings');
});
import { test, expect } from '@playwright/experimental-ct-vue';
import HelloWorld from './HelloWorld.vue';

test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(HelloWorld, {
    props: {
      msg: 'Greetings',
    },
  });
  await expect(component).toContainText('Greetings');
});
import { test, expect } from '@playwright/experimental-ct-svelte';
import HelloWorld from './HelloWorld.svelte';

test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(HelloWorld, {
    props: {
      msg: 'Greetings',
    },
  });
  await expect(component).toContainText('Greetings');
});
import { test, expect } from '@playwright/experimental-ct-solid';
import HelloWorld from './HelloWorld';

test.use({ viewport: { width: 500, height: 500 } });

test('should work', async ({ mount }) => {
  const component = await mount(<HelloWorld msg="greetings" />);
  await expect(component).toContainText('Greetings');
});
```

Additionally, it adds some config options you can use in your `playwright-ct.config.{ts,js}`.

Finally, under the hood, each test re-uses the `context` and `page` fixture as a speed optimization for Component Testing. It resets them in between each test so it should be functionally equivalent to `@playwright/test`'s guarantee that you get a new, isolated `context` and `page` fixture per-test.

I have a project that already uses Vite. Can I reuse the config?

At this point, Playwright is bundler-agnostic, so it is not reusing your existing Vite config. Your config might have a lot of things we won't be able to reuse. So for now, you would copy your path mappings and other high level settings into the `ctViteConfig` property of Playwright config.

```
import { defineConfig } from '@playwright/experimental-ct-react';

export default defineConfig({
  use: {
    ctViteConfig: {
      // ...
    },
  },
});
```

You can specify plugins via Vite config for testing settings. Note that once you start specifying plugins, you are responsible for specifying the framework plugin as well, `vue()` in this case:

```
import { defineConfig, devices } from '@playwright/experimental-ct-vue';

import { resolve } from 'path';
import vue from '@vitejs/plugin-vue';
import AutoImport from 'unplugin-auto-import/vite';
import Components from 'unplugin-vue-components/vite';

export default defineConfig({
  testDir: './tests/component',
  use: {
    trace: 'on-first-retry',
    ctViteConfig: {
      plugins: [
        vue(),
        AutoImport({
          imports: [
            'vue',
            'vue-router',
            '@vueuse/head',
            'pinia',
            {
              '@/store': ['useStore'],
            },
          ],
          dts: 'src/auto-imports.d.ts',
          eslintrc: {
            enabled: true,
          },
        }),
        Components({
          dirs: ['src/components'],
          extensions: ['vue'],
        }),
      ],
      resolve: {
        alias: {
          '@': resolve(__dirname, './src'),
        },
      },
    },
```

```
  },
});
```

How can I test components that uses Pinia?

Pinia needs to be initialized in `playwright/index.{js,ts,jsx,tsx}`. If you do this inside a `beforeMount` hook, the `initialState` can be overwritten on a per-test basis:

playwright/index.ts

```
import { beforeMount, afterMount } from '@playwright/experimental-ct-vue/hooks';
import { createTestingPinia } from '@pinia/testing';
import type { StoreState } from 'pinia';
import type { useStore } from '../src/store';

export type HooksConfig = {
  store?: StoreState<ReturnType<typeof useStore>>;
}

beforeMount<HooksConfig>(async ({ hooksConfig }) => {
  createTestingPinia({
    initialState: hooksConfig?.store,
    /**
     * Use http intercepting to mock api calls instead:
     * https://playwright.dev/docs/mock#mock-api-requests
     */
    stubActions: false,
    createSpy(args) {
      console.log('spy', args)
      return () => console.log('spy-returns')
    },
  });
});
```

src/pinia.spec.ts

```
import { test, expect } from '@playwright/experimental-ct-vue';
import type { HooksConfig } from '../playwright';
import Store from './Store.vue';

test('override initialState ', async ({ mount }) => {
  const component = await mount<HooksConfig>(Store, {
    hooksConfig: {
      store: { name: 'override initialState' }
    }
  });
  await expect(component).toContainText('override initialState');
});
```

How do I access the component's methods or its instance?

Accessing a component's internal methods or its instance within test code is neither recommended nor supported. Instead, focus on observing and interacting with the component from a user's perspective, typically by clicking or verifying if something is visible on the page. Tests become less fragile and more valuable when they avoid interacting with internal implementation details, such as the component instance or its methods. Keep in mind that if a test fails when run from a user’s perspective, it likely means the automated test has uncovered a genuine bug in your code.

Previous
Clock

Next
Debugging Tests

- Blog
- Ambassadors