**E2E Synpress  
End-to-End Testing for web3 Dapps with Playwright and Synpress**

End-to-end (E2E) testing is crucial for ensuring the quality and reliability of any web application. For decentralized applications (Dapps), this takes on added complexity due to the need to interact with blockchain wallets like MetaMask. This lesson demonstrates how to use Playwright, a powerful E2E testing framework, extended with Synpress, a specialized tool designed to handle web3 wallet interactions, to test a Dapp's UI based on wallet connection status.

**Setting Up Your Environment**

Before diving into the tests, ensure you have NodeJS and a package manager (like pnpm, npm, or yarn) installed. You'll also need Playwright set up in your project. If you haven't already, you can typically add it using:

pnpm add -D @playwright/test

pnpm exec playwright install # Installs necessary browser binaries

Next, install Synpress:

pnpm add -D @synthetixio/synpress

For simpler local testing and debugging, especially when dealing with asynchronous wallet interactions, it's often helpful to configure Playwright to run tests sequentially. Modify your playwright.config.ts file:

import { defineConfig } from '@playwright/test';

​

export default defineConfig({

testDir: './test', // Adjust if your tests are elsewhere

/\* Run tests sequentially for simplicity \*/

fullyParallel: false,

workers: 1,

/\* Other configurations \*/

retries: process.env.CI ? 2 : 0,

forbidOnly: !!process.env.CI,

// ... rest of your config

});

Setting fullyParallel: false and workers: 1 prevents potential race conditions or interference between tests when dealing with a single MetaMask instance locally.

**Your First Playwright Test**

Let's start with a basic Playwright test to confirm our setup works and the application loads correctly. This test simply navigates to the Dapp's page and verifies the title. Create a test file (e.g., test/playwright/basic.spec.ts initially):

import { test, expect } from '@playwright/test';

​

test('has title', async ({ page }) => {

await page.goto('/'); // Assumes your app runs at the root path configured in playwright.config.ts

// Expect a title "to contain" a substring.

await expect(page).toHaveTitle(/TSender/); // Adjust "TSender" to match your Dapp's title

});

This confirms Playwright can launch the browser, navigate, and perform basic assertions, but it doesn't interact with any web3 components yet.

**The Challenge of Automating Wallet Interactions**

The primary goal often involves testing UI changes based on wallet connection. For example, displaying a specific form only *after* the user connects their wallet. Automating interactions with browser extensions like MetaMask directly within standard E2E frameworks like Playwright is notoriously difficult due to their isolated nature and complex UI flows (popups, approvals).

**Introducing Synpress for web3 Testing**

This is where Synpress comes in. Synpress is built on top of Playwright (and Cypress) and provides specialized commands and utilities to seamlessly automate interactions with wallets like MetaMask. It handles the complexities of finding and interacting with the wallet extension's UI elements, allowing you to focus on your Dapp's logic.

**Configuring Synpress**

To use Synpress, you need two key pieces of configuration:

1. **Wallet Setup File:** Synpress requires instructions on how to set up the MetaMask instance it will control during tests. This typically involves providing a seed phrase and password. Create a setup file, for example, test/wallet-setup/basic.setup.ts:

import { defineWalletSetup } from '@synthetixio/synpress';

import { MetaMask } from '@synthetixio/synpress/playwright'; // Use Playwright-specific import

​

// WARNING: Do NOT commit real seed phrases. Use environment variables for secrets.

const SEED\_PHRASE = 'test test test test test test test test test test test junk'; // Example phrase (e.g., from Anvil/Hardhat)

const PASSWORD = 'Tester@1234'; // Example password

​

export default defineWalletSetup(PASSWORD, async (context, walletPage) => {

// Synpress uses the context and walletPage from Playwright fixtures

const metamask = new MetaMask(context, walletPage, PASSWORD);

​

// Import the wallet using the defined seed phrase

await metamask.importWallet(SEED\_PHRASE);

​

// You can add more setup here, like adding networks or importing tokens

// await metamask.addNetwork({ name: 'Anvil', rpcUrl: 'http://127.0.0.1:8545', chainId: 31337, symbol: 'ETH' });

});

**Security Note:** Avoid hardcoding sensitive information like seed phrases and passwords directly in your code. Use environment variables or a secure secret management system in real-world scenarios.

1. **Synpress Cache:** Synpress needs to prepare a specific version of MetaMask for automation. Run the following command to download and cache it:

pnpm synpress

This creates a .cache-synpress/ directory. Add this directory to your .gitignore file to avoid committing it to version control:

# .gitignore entry

.cache-synpress/

**Implementing the Wallet Connection Test**

Now, let's modify our test file to use Synpress to test the wallet connection flow. We'll aim to verify that initially, a "Please connect" message is shown, and after connecting via Synpress, an "Airdrop Form" (identified perhaps by a "Token Address" label) becomes visible.

Update test/playwright/basic.spec.ts:

// Import Synpress essentials instead of plain Playwright

import basicSetup from '../wallet-setup/basic.setup'; // Import our wallet setup

import { testWithSynpress } from '@synthetixio/synpress';

import { MetaMask, metaMaskFixtures } from '@synthetixio/synpress/playwright';

​

// Create a Synpress test instance using Playwright fixtures and our setup

const test = testWithSynpress(metaMaskFixtures(basicSetup));

​

// Use expect from the Synpress test instance

const { expect } = test;

​

// Keep the basic title test (optional, but good practice)

test('has title', async ({ page }) => {

await page.goto('/');

await expect(page).toHaveTitle(/TSender/);

});

​

// The core test for wallet connection and UI change

test("should show the airdrop form when connected, otherwise not", async ({ page, context, metamaskPage, extensionId }) => {

// The test function now receives Synpress/MetaMask fixtures: context, metamaskPage, extensionId

​

await page.goto('/');

​

// 1. Verify Initial State (Disconnected)

// Use a specific locator like getByText or preferably getByTestId if available

await expect(page.getByText('Please connect')).toBeVisible(); // Check for a disconnected message

await expect(page.getByText('Token Address')).not.toBeVisible(); // Check that the form element is initially hidden

​

// 2. Initiate Connection Process

// Use getByTestId for robust element selection if your Dapp includes test IDs

await page.getByTestId('rk-connect-button').click(); // Click your Dapp's connect button

​

// Wait for the wallet selection modal (e.g., RainbowKit) and click MetaMask

// Adjust timeout and locator as needed

await page.getByTestId('rk-wallet-option-io.metamask').waitFor({

state: 'visible',

timeout: 30000 // Increase timeout if needed

});

await page.getByTestId('rk-wallet-option-io.metamask').click();

​

// 3. Automate MetaMask Connection using Synpress

// Instantiate MetaMask helper using provided fixtures

const metamask = new MetaMask(context, metamaskPage, basicSetup.walletPassword, extensionId);

​

// Synpress handles the popup interaction and approval

await metamask.connectToDapp();

​

// (Optional) Add a custom network if needed for testing (e.g., local Anvil/Hardhat)

// const customNetwork = {

// name: 'Anvil',

// rpcUrl: 'http://127.0.0.1:8545',

// chainId: 31337,

// symbol: 'ETH',

// };

// await metamask.addNetwork(customNetwork);

​

// 4. Verify Final State (Connected)

// Wait for potential asynchronous updates after connection

await expect(page.getByText('Token Address')).toBeVisible({ timeout: 10000 }); // Check if a form label is now visible

await expect(page.getByText('Please connect')).not.toBeVisible(); // Check the disconnected message is gone

});

This test now uses testWithSynpress which injects the necessary fixtures (context, metamaskPage, extensionId). It creates a MetaMask helper instance and uses metamask.connectToDapp() to handle the complex interaction with the MetaMask extension popup, approving the connection automatically. Finally, it asserts that the UI reflects the connected state.

**Running and Debugging Synpress Tests**

You can run your Playwright/Synpress tests using the standard Playwright command:

pnpm exec playwright test

For a better debugging experience, use the Playwright UI mode:

pnpm exec playwright test --ui

**Common Synpress Issue: Cache Mismatch**

Sometimes, especially after updates or on different machines, Synpress might fail with an error like:

Error: Cache for playwright-metamask-10.34.0 (expected path: /path/to/your/project/.cache-synpress/playwright-metamask-10.34.0) does not exist.

This indicates a mismatch between the cache Synpress expects and the one generated. A common workaround is:

1. Run pnpm synpress again to regenerate the cache.
2. Observe the exact folder name Synpress *expected* from the error message (e.g., playwright-metamask-10.34.0).
3. Observe the folder name that was actually *generated* inside .cache-synpress/ (it might differ slightly, e.g., playwright-metamask-10.34.0-beta.0).
4. Manually rename the generated folder inside .cache-synpress/ to match the exact name Synpress expected in the error message.
5. Re-run your tests.

**Summary and Key Takeaways**

E2E testing is essential for building robust Dapps. While Playwright provides a solid foundation, automating interactions with browser wallets like MetaMask presents unique challenges. Synpress extends Playwright, offering specialized tools and fixtures (testWithSynpress, metaMaskFixtures, MetaMask class, connectToDapp) that significantly simplify this process. By configuring a wallet setup file and leveraging Synpress functions, you can write E2E tests that accurately simulate user flows involving wallet connections, ensuring your Dapp's UI behaves correctly in different states. Remember to handle secrets securely and be aware of potential cache issues during setup.