**E2E Playwright**

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

Following the deployment of our application, we now shift our focus to a critical aspect of development: End-to-End (E2E) testing. This stage is particularly vital in the web3 space. Given the potentially irreversible and costly nature of blockchain transactions, ensuring your application functions correctly from the user's perspective *before* final deployment is paramount. Mistakes involving incorrect transactions on a live blockchain can have severe consequences.

This lesson aims to introduce the fundamental concepts of E2E testing and demonstrate a basic implementation using Playwright, supplemented by Synpress for web3-specific interactions. While we won't build an exhaustive test suite, you'll gain a practical understanding to build upon. Think of this as an essential primer before tackling the final deployment.

**What is End-to-End (E2E) Testing?**

End-to-End (E2E) testing is a methodology used to validate the complete workflow of an application from start to finish, mimicking real user scenarios. It tests the application from the user's viewpoint, ensuring all integrated components – frontend, backend, APIs, databases, and crucially for us, blockchain interactions – function together seamlessly.

**Why is E2E Testing Critical in web3?**

The immutable nature of many blockchain operations makes E2E testing indispensable. Smart contracts, once deployed, and transactions, once executed, often cannot be easily altered or reversed. E2E tests act as a crucial safeguard, helping to catch integration bugs and validate entire user journeys (like connecting a wallet, filling a form, signing, and sending a transaction) before they can lead to financial loss or protocol failure in a production environment. An incorrect transaction on the mainnet isn't just a bug; it's often a significant, irreversible problem.

**Contrast with Unit Testing**

It's important to distinguish E2E testing from unit testing. Previously, we used vitest to perform unit testing on specific functions, such as the calculateTotal utility found in src/utils/calculateTotal/calculateTotal.test.ts.

* Unit Tests: Focus on verifying small, isolated pieces of code, like a single function, in isolation. They ensure that the individual building blocks work correctly.
* E2E Tests: Focus on verifying the integrated system as a whole. They simulate user interactions through the user interface (UI) and across different parts of the system to ensure the entire flow works as expected.

Here's an illustrative example of the structure of our previous unit test:

// src/utils/calculateTotal/calculateTotal.test.ts (Illustrative structure)

import { describe, expect, it } from 'vitest';

import { calculateTotal } from './calculateTotal';

​

describe('calculateTotal', () => {

it('should work with newlines', () => {

expect(calculateTotal('100\n200')).toBe(300);

expect(calculateTotal('100\n200\n300')).toBe(600);

});

// ... other tests ensuring the function works in isolation

});

E2E tests, in contrast, wouldn't just test calculateTotal; they would test the entire process of a user entering numbers into the UI, clicking a button, and seeing the correct total displayed, potentially involving multiple components and even backend interactions.

**The Limitations of Manual E2E Testing**

Throughout this course, we've inadvertently performed manual E2E testing. Consider the steps involved in testing our airdrop functionality manually:

1. Start the application locally (localhost:3000).
2. Navigate to the airdrop page.
3. Connect a wallet (e.g., MetaMask).
4. Enter a token address into the form.
5. Input recipient addresses.
6. Input corresponding amounts.
7. Click the "Send tokens" button.
8. Interact with the MetaMask pop-up (review transaction details, confirm or reject).
9. Observe console logs for feedback.
10. Potentially verify the transaction results on a block explorer or check balances.

While effective for simple checks, this manual approach quickly becomes inefficient and prone to human error as the application grows. Adding more features, smart contracts, or UI elements makes manual E2E testing tedious and difficult to scale reliably.

**Automating E2E Tests: Playwright and Synpress**

To overcome the limitations of manual testing, we turn to automated E2E testing tools. For our purposes, we'll use:

* Playwright: (playwright.dev) A powerful NodeJS library developed by Microsoft for browser automation and E2E testing. It enables programmatic control over modern browsers like Chromium, Firefox, and WebKit, allowing us to script user interactions.
* Synpress: (synpress.io / GitHub: Synthetixio/synpress) An E2E testing framework built specifically for web3 decentralized applications (dapps). Developed by Synthetix, Synpress extends Playwright (and previously Cypress) with essential capabilities for interacting with browser wallet extensions, most notably MetaMask. While it can sometimes be described as "finicky," it provides indispensable functionality for testing web3 user flows that involve wallet interactions.

**Adding a Testable Feature: Conditional UI Rendering**

To demonstrate a practical E2E test, let's add a simple feature to our application: the main airdrop form should only be visible *after* the user has successfully connected their wallet. If no wallet is connected, a message prompting the user to connect should be displayed instead.

**Implementation Steps:**

1. Create a HomeContent Component: Encapsulate the existing AirdropForm within a new component. This helps organize the conditionally rendered content.

// src/components/ui/HomeContent.tsx (Simplified structure)

import AirdropForm from "./AirdropForm"; // Adjust path as needed

​

export default function HomeContent() {

return (

<div>

<AirdropForm />

</div>

);

}

1. **Modify the Main Page (page.tsx):** Introduce client-side logic using wagmi to check the wallet connection status and render content conditionally.

// src/app/page.tsx (Key additions/changes)

'use client'; // Required for client-side hooks like useAccount

​

import HomeContent from "@/components/ui/HomeContent"; // Import the new component

import { useAccount } from "wagmi"; // Import the wagmi hook

​

export default function Home() {

const { isConnected } = useAccount(); // Get wallet connection status

​

return (

<div>

{/\* Conditionally render based on connection status \*/}

{isConnected ? (

<div> {/\* Wrap conditional components \*/}

<HomeContent />

</div>

) : (

<div> {/\* Wrap conditional components \*/}

Please connect a wallet...

</div>

)}

</div>

);

}

1. **Manual Verification:** Before automating, manually test this change. Load the application, observe the "Please connect..." message. Connect your wallet, and verify that the message disappears and the AirdropForm (within HomeContent) becomes visible. Disconnect the wallet to ensure the message reappears.

**Setting Up Playwright and Synpress**

Now, let's install and configure the necessary testing tools.

1. **Install Dependencies:** Add Synpress as a development dependency. Since Synpress includes Playwright, we only need to install Synpress directly.

pnpm add -D @synthetixio/synpress

*(Using -D flags it as a development dependency, as it's not needed for the production build.)*

1. **Initialize Playwright:** Run the Playwright initialization command.

pnpm create playwright@latest

This will prompt you for configuration choices:

* + Tests Location: Change the default tests directory to test. Synpress typically expects tests to reside in a test directory by default.
  + GitHub Actions: Select false for now. We won't configure CI/CD pipelines in this lesson.
  + Install Browsers: Select true. Playwright needs the browser binaries (Chromium, Firefox, WebKit) to execute tests.

1. **Configure Playwright (playwright.config.ts):** Modify the generated playwright.config.ts file for our project needs.
   * baseURL: Uncomment this option and set it to your local development server address (http://127.0.0.1:3000). This allows tests to use relative paths like page.goto('/'), making them more maintainable.
   * projects: For faster test execution during development and demonstration, comment out the firefox and webkit projects. We'll focus on Chromium.
   * webServer: Uncomment and configure this section. This crucial setting tells Playwright to automatically start your development server (pnpm run dev) before running tests and ensures it waits until the server is ready at the specified URL. Set reuseExistingServer: true to avoid restarting the server if one is already running.

Here are the key parts of the modified configuration:

// playwright.config.ts (Key configurations highlighted)

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

​

export default defineConfig({

testDir: './test', // Set during initialization to 'test'

​

/\* Shared settings for all the projects below. See https://playwright.dev/docs/api/class-testoptions. \*/

use: {

/\* Base URL to use in actions like `await page.goto('/')`. \*/

baseURL: 'http://127.0.0.1:3000', // Uncommented and set

​

/\* Collect trace when retrying the failed test. See https://playwright.dev/docs/trace-viewer \*/

trace: 'on-first-retry',

},

​

/\* Configure projects for major browsers \*/

projects: [

{

name: 'chromium',

use: { ...devices['Desktop Chrome'] },

},

​

// { // Commented out for faster demo runs

// name: 'firefox',

// use: { ...devices['Desktop Firefox'] },

// },

​

// { // Commented out for faster demo runs

// name: 'webkit',

// use: { ...devices['Desktop Safari'] },

// },

],

​

/\* Run your local dev server before starting the tests \*/

webServer: { // Uncommented and configured

command: 'pnpm run dev',

url: 'http://127.0.0.1:3000',

reuseExistingServer: true, // Useful during development

},

});

**Writing and Running Your First E2E Test**

Playwright created an example test file (test/example.spec.ts). Let's modify it to test our application's title.

1. **Modify Example Test (test/example.spec.ts):**
   * Remove the second default test case (get started link).
   * Change the navigation target from https://playwright.dev/ to / (our baseURL).
   * Update the title assertion to match our application's title ("TSender").

// test/example.spec.ts (Modified for TSender)

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

​

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

// Navigate to the app's root page (uses baseURL from config)

await page.goto('/');

​

// Expect the page title to be exactly "TSender".

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

});

1. **Running Playwright Tests:** You can run tests using several commands:
   * pnpm exec playwright test: Executes all tests headlessly (no browser UI shown). This is typically used in CI environments.
   * pnpm exec playwright test --ui: Runs tests using the interactive Playwright Test UI. This mode is excellent for development and debugging. It visually shows the browser, steps executed, allows time-travel debugging through snapshots, and displays console logs and errors.
   * pnpm exec playwright show-report: Opens a detailed HTML report of the last test run in your browser.
2. **Demonstration:**
   * Run pnpm exec playwright test --ui. Observe the Playwright UI launching. It should start your dev server (if not already running) and then execute the 'has title' test against the Chromium browser.
   * You should see the test pass. Explore the UI: click on the page.goto('/') step to see a snapshot of your application as loaded by the test. Click on the expect(page).toHaveTitle() step.
   * (Try temporarily changing the expected title in the test to something incorrect, like "Wrong Title", and re-run via the UI. Observe how Playwright clearly indicates the failure, showing the expected vs. actual title.)
   * Finally, stop the UI and run pnpm exec playwright test in your terminal. Observe the headless execution output. Then, run pnpm exec playwright show-report to view the generated HTML report.

**Summary and Next Steps**

We've established the critical importance of E2E testing, especially for web3 applications where errors can be costly. We contrasted E2E testing with unit testing and saw the limitations of manual approaches. By installing and configuring Playwright and Synpress, and running a basic automated test against our application, you've taken the first step towards building a more robust and reliable dApp.

Playwright provides powerful browser automation capabilities, and its Test UI significantly aids in debugging. While we haven't yet leveraged Synpress's specific MetaMask interaction features, the foundation is now laid. The next logical steps involve writing more complex E2E tests that simulate core user flows, such as connecting a wallet (using Synpress commands), filling out the airdrop form, interacting with the subsequent MetaMask transaction prompts, and asserting the expected outcomes.