CAN AI TEST YOUR WEBAPP?

RAPHAEL THEILER

WHO AM I?

- **RAPHAEL THEILER**

- 41 Years old
- 21 Years coding with DataFlex
- IT security as a hobby

- rthp@consultra.ch

NOBODY HAS TIME TWO WRITE TEST CASES

- Small Teams
- Customers with limited budget
- Do you want another feature or test cases?

- **Solution:**
  Let the AI write test cases for us, so we can focus on new features

PLAYWRIGHT

- **TEST FRAMEWORK**

- Framework to test WebApps
- Made by Microsoft
- APIs for TypeScript, Python and dotnet
- Has AI agents to generate test cases

PLAYWRIGHT IN 2 MINUTES
ONE COMMAND GIVES YOU A REAL BROWSER, A TEST RUNNER, AND A REPORT

- Drives a real browser - it tests what your users actually see
- Auto-waits for elements, so tests are far less flaky than Selenium-style waits
- Built-in test runner, traces, and an HTML report - debug failures by replaying them.
- Tests are plain TypeScript. They run locally and in CI exactly the same way.

```
# 0. create an empty folder for your test project
# 1. add Playwright to your project
npx create-playwright@latest

# 2. install the browsers
npx playwright install

# 3. run the example tests
npx playwright test  # --ui for a nice user interface

# 4. open the HTML report
npx playwright show-report
```

QUICK DEMO

AI AGENTS

- **AI AGENTS?!**

- Basically a text file (markdown) with instruction for the AI on how to behave, and what tools are available

```
---

You are a Playwright Test Generator, an expert in browser automation and end-to-end
Your specialty is creating robust, reliable Playwright tests that accurately simula
application behavior.

# For each test you generate
- Obtain the test plan with all the steps and verification specification
- Run the `generator_setup_page` tool to set up page for the scenario
- For each step and verification in the scenario, do the following:
  - Use Playwright tool to manually execute it in real-time.
```

PLAYWRIGHT AGENTS
THREE AGENTS DO THE WORK — YOU PICK THE AI RUNTIME THAT DRIVES THEM

1 Planner

Explores your live app in a real browser and writes a Markdown test plan in specs/.

2 Generator

Turns the plan into executable .spec.ts files, verifying locators against the page as it goes.

3 Healer

Runs the suite and repairs broken locators automatically — or skips a test if the app is truly broken.

One command — pick the AI runtime that drives the agents:

```
npx playwright init-agents --loop=claude
npx playwright init-agents --loop=vscode
npx playwright init-agents --loop=opencode
```

HOW AN AGENT RUNS
YOU PICK THE AGENT, GIVE IT A SEED AND A PROMPT — IT DRIVES A REAL BROWSER

1 Select the agent

In Claude Code, pick Planner, Generator, or Healer.

2 Seed + prompt

Hand it the seed test and a clear, specific instruction.

3 Browser via MCP

The agent launches a real browser and observes live DOM.

4 Writes the artifact

A plan, a .spec.ts file, or a fixed locator
Finally: You review the plan/code

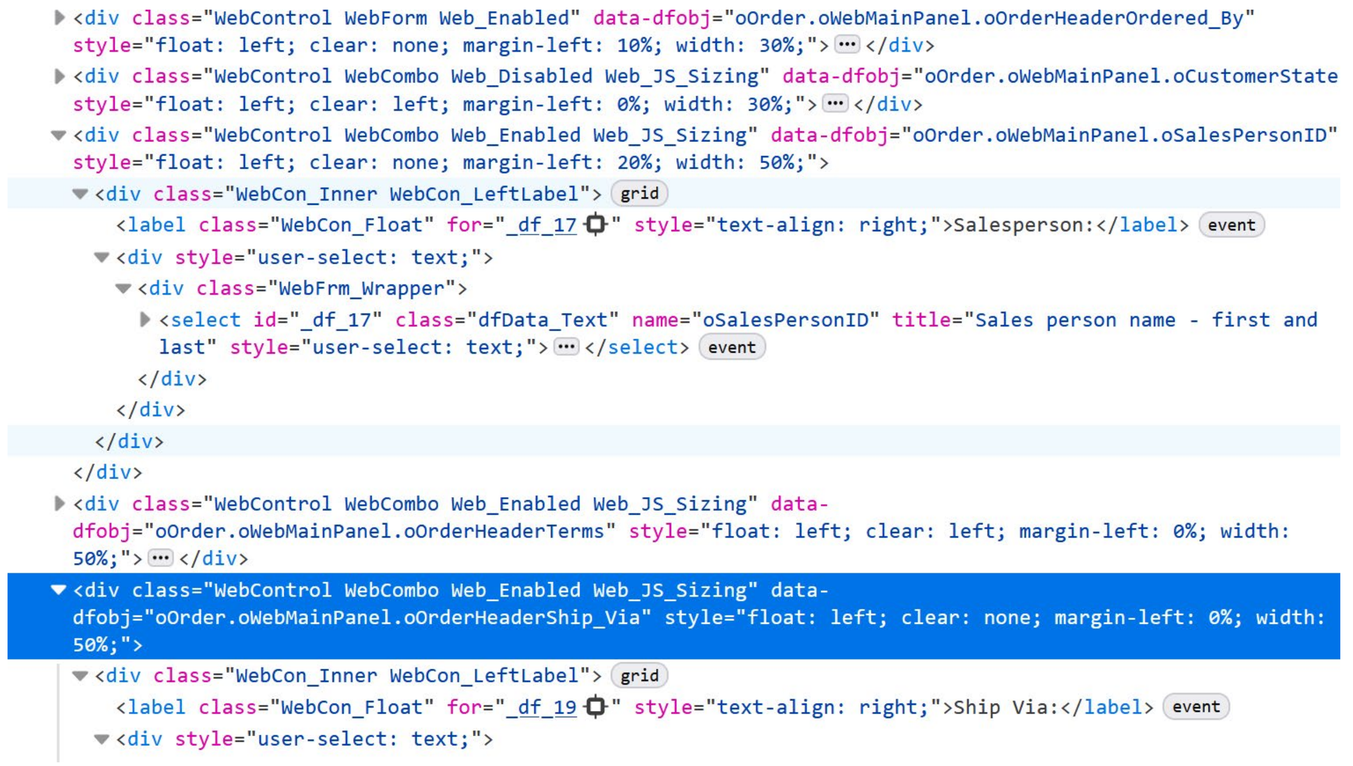
↻ *Healer re-runs the suite and feeds fixes back in — the loop repeats until the test is green or skipped.*

LOCATOR / DOM

- How does Playwright know where to click?

Either: **#_df_19 > option:nth-child(3)**

Or: **div.WebControl:nth-child(12)**



THE DATAFLEX ANGLE
USE DATA-DFOBJ AS YOUR STABLE TEST-ID

- **DataFlex WebApps generate the DOM. The id and name attributes can shift between builds - never lock tests to them.**
- Every WebApp object renders a data-dfobj attribute tied to the object's name in your source. That's the stable hook.
- Locate by TestId, not by CSS position or visible text - it survives restyling, relabelling, and reordering.
- Tip: register data-dfobj as Playwright's testIdAttribute so getByTestId() targets it automatically.

```
// What DataFlex renders
<input data-dfobj="oCustomer.oName" />

// Register it once (config)
use: { testIdAttribute: 'data-dfobj' }

// Now your locator is rock-solid
page.getByTestId('oCustomer.oName')
```

STEP 1 — THE SEED
GIVE THE AGENT A STARTING POSITION

- The seed test is deliberately boring: log in, navigate, assert the app is ready.
- Its only job is to bootstrap a known state so the Planner has somewhere to start exploring.
- It gets copied into every generated test, so put your shared setup here once.

seed.spec.ts

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

test('seed: logged-in order app', async ({ page }) => {
  // 1. open the WebApp
  await page.goto('/WebOrder_27_0/Index.html');
  // 2. log in via stable data-dfobj ids
  await page.getByTestId('oLoginDialog....oLoginName').fill(John');
  await page.getByTestId('oLog...oPassword').fill(John');
  await page.getByTestId('oLog...oLoginButton').click();
  // 3. assert we're ready to explore
  await expect(page.getByTestId('oOr...oDetailGrid')).toBeVisible();
});
```

STEP 2 — THE PLAN
THE PLANNER WRITES SPECS/*.MD

- Human-readable Markdown: scenarios, steps, expected results.
- You review and edit this BEFORE any code is generated - cheapest place to fix things.
- A vague plan makes vague tests. A specific plan is your real leverage.

specs/create-order.md

```
# Create a new order

## Scenario: happy path
1. Open the order screen
2. Pick a customer from the prompt
3. Add 2 order lines
4. Save the order

**Expected**
- Order number is shown
- Total equals sum of the lines
```

STEP 3 — THE TEST
YOUR FIRST GENERATED .SPEC.TS

Arrange

Open the screen, reach a known state.

Act

Drive the UI through stable data-dfobj ids.

Assert

Check the user-visible outcome, not internals.

tests/create-order.spec.ts

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

test('creates an order', async ({ page }) => {
  // Arrange
  await page.goto('/WebOrder_27_0/Index.html');
  // Act
  await page.getByTestId('oCustomer').selectOption('ACME');
  await page.getByTestId('oSaveOrder').click();
  // Assert
  await expect(page.getByTestId('oOrderNo'))
    .not.toBeEmpty();
});
```

WHAT I LEARNED

- Write a good first test case either manually or with the AI.
  - The AI is stupid and copies your mistakes. It also copies «the good stuff»

- If the same issue/bug comes up all the time
  - Add it to the Agent.md or Claude.md (or the equivalent of your AI)
    Tell the AI how to avoid it

- «Drill-down» style WebApps are easier to test than «Desktop style»
  - The menu to browse, save and delete records is not intuitiv
    (neither for humans nor for the AI)

PROMPT INJECTION
SECURITY

- Be aware of prompt injection
- Don’t trust the AI with any real credentials
  - Use local credentials or test credentials
- It’s not necessary the «AI» that steals your credentials but an attacker with promp injection
- Don’t add random libraries, skills or agents to your projects
- If the AI parses your code and the libraries it uses it may follow instructions embedded in it

BONUS: PROBE FOR SECURITY HOLES

SECURITY — XSS
DOES INPUT COME BACK AS CODE?

- Type a script payload into a field, save, reopen the record.
- If it runs, the app rendered your input as HTML - that's stored XSS.
- The test asserts the payload shows up as TEXT, and no dialog ever fires.
- Great agent task: 'try this payload on every free-text field'.

tests/security/xss.spec.ts

```
const payload = '<script>alert(1)</script>';

// fail loudly if a dialog ever pops
page.on('dialog', () => {
  throw new Error('XSS fired!');
});

await page.getByTestId('oComment').fill(payload);
await page.getByTestId('oSave').click();

// stored as text, not executed
await expect(page.getByTestId('oComment'))
  .toHaveText(payload);
```

SECURITY — SQL INJECTION
DOES THE APP SANITISE THE QUERY?

- Feed a classic payload into a search or login field.
- Two safe outcomes: it returns nothing, or it errors only.
- Red flag: extra rows, a stack trace, or a raw SQL error on screen.
- **The agent never touches the database — it only reads the response.**

tests/sqli-probe.spec.ts

```ts
// A string that breaks naive SQL
const payload = "' OR '1'='1";

await page.getByTestId('oSearch').fill(payload);
await page.getByTestId('oGo').click();

// No DB leakage, no raw SQL error
await expect(page.locator('body'))
  .not.toContainText(/SQL|syntax/i);
```

WRAP-UP
WHAT TO TAKE HOME

1. **Setup is two commands.** npx playwright install, then init-agents — minutes, not an afternoon.
2. **Agents do the grunt work, you stay in control.** Planner explores, Generator writes, Healer fixes drift — you review the plan.
3. **Stable locators make or break it.** Wire data-dfobj as testIdAttribute so getByTestId survives UI changes.
4. **A good plan equals a good test.** Specs in specs/*.md drive the arrange / act / assert structure.
5. **Then let it probe for security.** XSS and SQLi checks are just more assertions the agent can write.

QUESTIONS?

THANK YOU