Testing Matters – Don’t Let Users Test Your Code!

An introduction to automated testing, focusing on End-to-End Testing of browser-based applications

Paul Harrison
Martin Davies
About Paul Harrison

• Developer at FoCul, focusing on Front-end development using Angular

• Over 20 years' experience with HCL Notes/Domino - everything from support and administration, to infrastructure, migrations and development

Email: paul.harrison@focul.net
Twitter: @PaulHarrison
About Martin Davies

• Delivery and Technical lead at FoCul
• Worked with Notes/Domino and associated technologies since 1993 - Admin, Dev, Infrastructure, Associated Technologies, Consultancy
• Consequently “Jack of All Trades, Master of None 😊”
• Bell Ringer and Runner

Email: martin.davies@focul.net
Twitter: @martin_davies
Agenda

- Introduction
- Approaches
- Methodologies
- Other Test Considerations
- End-to-End Testing using Cypress
- How-to and Demo
- XPages Hints and Tips
- Best Practices
- Summary
Introduction - Testing Challenges

• Testing is expensive
• Manual test plans can be inconsistent and prone to human error
• Testing environments can be complex to setup and manage
• Commercial testing tools were prohibitively expensive, particularly for smaller organisations
Introduction - Automated Testing

• Gives your organisation and your customers increased confidence in your shipped products
• Consistent, highly repeatable and reusable
• Self-documenting
• Allows you to more frequently ship application updates
• Fewer shipped bugs == reduced support tickets
• Free to use Open-Source or low-cost tools now available
Testing Approaches

End-to-End Tests (E2E)
- Tests the application User Interface independently of its actual code
- Runs at application-level
- Language & framework agnostic

Integration Tests
- Tests how libraries or packages of functions integrate and interact with each other
- Runs at code-level
- Language & framework specific

Unit Tests
- Tests basic functions
- Runs at code-level
- Language specific

Each layer of the pyramid has a different size, indicating the number of tests that could be written within each stage.
Testing Methodologies

**Code Driven Development**
- Develop tests *after* coding
- Easier to start with, as you likely already have some code to test
- Better suited to adding tests to existing code

**Test Driven Development (TDD)**
- Develop tests *before* any code (based on provided design specs)
- Can make for more efficient code (like flowcharting prior to coding)
- Can be a difficult concept to grasp
- Better suited to Unit and Integration testing or when starting to develop new code
Other Testing Considerations

**Test Coverage**
- What percent of your code is tested
- Your goal should not necessarily be 100% test coverage of the entire application (Unit, Integration and E2E tests could each have different percentages), but rather to ensure that you test a reasonable percentage of the code, and focus on those areas that are critical
- 60% - 80% is usually considered very good

**Consistent Test Data**
- Sample data (fixed or easily resettable)
- Mock or fake data (artificially inserted into the response payload of an API request)
Why Focus on End-to-End Testing?

• Easy to quickly produce usable tests
• Allows testing of user workflows (aka ‘journeys’ or ‘stories’), as well as the UI and DOM styling and layout
• Can focus on writing tests without having to worry about how the code itself works
• Does not necessarily require development skills - can delegate to more junior colleagues or QA Team
Introducing Cypress (1/3)

Cypress incorporates some of the “best-in-breed”, open-source testing libraries

Credit: https://www.cypress.io/how-it-works
Introducing Cypress (2/3)

• Tests run inside the actual browser (tests written in JavaScript/Typescript)
• Very well documented with lots of good examples
• Free to use, Open Source and under active development
• Supports “live-reload” and can run in foreground, or headless mode
• Responsiveness capability using pre-set or custom viewport sizes
Introducing Cypress (3/3)

• Time Travel – DOM snapshots during test execution and review
• Automatically waits for commands and assertions
• Extensible - custom functions, and 3rd party custom plug-ins
• Screenshots – programmatically, or on failure
• Video – records a video of the test suite execution
• Cross browser support:
  • Chromium: Chrome, Edge and Electron
  • Firefox
  • WebKit: Safari (experimental)
Cypress Cloud (formally Dashboard) is an optional web-based paid for service (an initial free tier is available) which provides additional features:

- Test result consolidation
- Test optimisation
- Parallelisation and load balancing to improve testing performance
- Improved historical reporting
How To and Demo Time!

- Installation
- Scripting
- Testing UI
- Demo
Cypress Installation and Configuration

• Installs as a regular node.js package (assume node and npm already installed)
• Cypress recommend non-global, dev-dependency installation e.g. npm install cypress --save-dev
• Can also be installed outside of an existing package (for ad-hoc testing, eg API, or other systems etc.)
• Slow to install and first-time run
• Run using:
  • npx cypress open
  • npx cypress run <options> (headless mode)
Cypress Components – Folder Structure

• cypress *(root folder)*
  • cypress.config.js *(tune cypress config)*

Default Folders
• fixtures *(store data objects used in testing)*
• e2e
  • Where we write our end to end tests
  • tests - can be grouped into additional sub-folders
  • samples *(lots of useful examples in here!)*
• plugins *(store any plugins required for testing)*
• support
  • command.js *(commonly used bespoke commands)*
  • Index.js *(runs before every test)*

Additional Output Folders
• downloads
• screenshots
• videos
Tests create in `<testname>.cy.js` files

Define Test Suite

- Define Test
  - Commands
  - Assertion

- Define Test
  - Commands
  - Assertion
Anatomy of a Simple Test (2/2)

Define Test Suite

describe('Test Suite', () => {

Define Test

  Define Test Commands

    Commands

  Define Test Assertion

    Assertion

Define Test Commands

  cy.visit('website')

Define Test Assertion

  cy.get('object')

    .should('have.length', 2)

})

})
Test Definitions

• Borrowed from Mocha

• Test Suite
  • describe() or context()
  • Contain test
  • Can contain child test suites

• Individual Tests
  • it() or specify()

Control Tests

• Switch off Tests
  – Prefix with x
    • xdescribe  xit
  – Append . Skip
    • describe.skip  it.skip

• Include Only Specific Tests
  – Append . only
    • describe.only  it.only
Commands

• All built-in cypress commands begin cy.

• cy.visit() – navigate directly to a page (relative to baseUrl or absolute)

• cy.contains – check that the page contains the required text

• cy.get – Get an element on the page and chain assertions (Chai and/or Mocha?) to it eg .should(assertion type(?), value )?

• See https://docs.cypress.io/api/table-of-contents
Assertions

- **Assertions**
  - Chai Assertion Library
  - Validations that confirm if a test has passed or failed
  - Assertions are automatically retried until they result or time out.

- **Default** - Many commands have a default, built-in assertion
  - `cy.visit()`  
    - expects page to send text/html and a 200 status code
  - `cy.get()`  
    - expects the element to exist in the DOM

- **Implicit** – preferred method. Add to the `cy` command chain
  - `should()`  
    - `cy.get(element).should('include', 'some text')`
  - `and ()`  
    - `cy.get(element).should('include', 'some text') and ('style','some style')`

- **Explicit** – asserts a specified subject. Good for Unit Tests. Not chainable
  - `expect()`  
    - eg `expect(actual).to.equal(expected)`
  - `assert()`  
    - eg `assert.equal(actual, expected, [message])`
Cypress Testing UI

• Test Runner
• Test Results
• Time Travel – Go Back In Time!
• Cypress Studio (Experimental... but Great!) – Record your own scripts
• Headless Mode

• Demo App
https://example.cypress.io/todo
Cypress Components – Understand Failures

1. Error name - This is the type of the error (e.g. AssertionError, CypressError).
2. Error message - This generally tells you what went wrong. It can vary in length. Some are short like in the example, while some are long, and may tell you exactly how to fix the error.
3. Learn more - Some error messages contain a Learn more link that will take you to relevant Cypress documentation.
4. Code frame file - This is usually the top line of the stack trace and it shows the file, line number, and column number that is highlighted in the code frame below. Clicking on this link will open the file in your preferred file opener and highlight the line and column in editors that support it.
5. Code frame - This shows a snippet of code where the failure occurred, with the relevant line and column highlighted.
6. View stack trace - Clicking this toggles the visibility of the stack trace. Stack traces vary in length. Clicking on a blue file path will open the file in your preferred file opener.
7. Print to console button - Click this to print the full error to your DevTools console. This will usually allow you to click on lines in the stack trace and open files in your DevTools.
Experimental Feature – visible in project settings

Add the following to cypress.config.js

```javascript
   e2e: {
      experimentalStudio: true
   },
```
Cypress Custom Commands

• Found in ./support/commands.js
• Build custom commands
• Used to group together a set of cy statements for example part of an it block
  • simply replace the lines in the it block using cy.customCommandName()
• Very useful for command function such as login/logout
• Good examples in sample file
Quickly run finished test scripts
Creates a mp4 of the test in /videos
Failed tests - screenshot in /screenshots
Run a single spec file or all tests
Uses Electron unless browser is specified

npx cypress run --browser <<browser>> --spec "specfile.js"
We have occasionally observed click failures when using the regular `click()` approach.

Alternative clicking method are available if required.

```javascript
// #1 XPages regular click approach, but seems to occasionally fail
cy.get('#view::id1::id2::id129::id200:buttonCancel').click();

// #2 XPages better alternative click approach if #1 above fails
  cy.get('#view::id1::id2::id129::id200:buttonCancel').trigger("click");

// #3 XPages best alternative click approach if #2 above fails
  cy.get('#view::id1::id2::id129::id200:buttonCancel').trigger("mouseover").click();
```
Due to their naming convention, selecting elements in XPages can often be problematic.

Even though they are technically valid from a browser perspective, selectors containing colons (:) need to be “double-escaped” (prefixed with `\`) otherwise the element will not be found when the test executes.

```
cy.get('#view:_id1:_id2:_id343:button1').click()
```

```
cy.get('#view\:_id1\:_id2\:_id343\:button1').click() // now works because all : have been double-escaped
```

This impacts elements added either manually (based on Dev Tools inspection), or via the Selector Playground.

Strangely, Cypress Studio seems to work correctly.
XPages Hints and Tips #2 - Selectors (2/2)

• Dev Tools => 😞

```html
<div class="lotusform28buttons">
  <button id="view: id1: id2: id343:button1" name="view: id1: id2: id343:button1"
type="button" class="btn btn-default">Edit</button>
  <button id="view: id1: id2: id343:button3" name="view: id1: id2: id343:button3"
type="button" class="btn btn-default">Delete</button>
</div>
```

```javascript
cy.get('#view: id1: id2: id343:button1').click()
```

AssertionError
Timed out retrying after 4000ms: Expected to find element: #view: id1: id2: id343:button1, but never found it.

• Cypress Selector Playground => 😞

```javascript
cy.get('#view:id1:id2:id343:button1').click()
```

AssertionError
Timed out retrying after 4000ms: Expected to find element: #view:id1:id2:id343:button1, but never found it.

• Double-escape : with \\ (or use Cypress Studio) => 😊

```javascript
cy.get('#view\\:id1\\:id2\\:id343\\:button1').click() // now works because all : have been double-escaped
```
• Because selectors in XPages automatically generated, they can unexpectedly change, resulting in a failed test

• Mitigate by adding dedicated (and unique) data-cy tags, or aliases, to your markup, which then allows direct selector targeting

```javascript
cy.get('#view\:_id1\:_id2\:_id343\:button1').click()
```

```javascript
cy.get('[data-cy="EditKeywordButton"]').click() // can now use once data-cy attr added to element
```

• `data-cy` tags can easily be added using Domino Designer via either the Design or the Source tab on the relevant page

• Selector Playground and Cypress Studio will then pick the `data-cy` tag in preference to any other possible option
XPages Hints and Tips #3 - data-cy Tag (2/2)

- Dev Tools before adding data-cy tag (Double-escaped 😊)

- Add data-cy tag attribute in Domino Designer

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessibility</td>
<td></td>
</tr>
<tr>
<td>basics</td>
<td></td>
</tr>
<tr>
<td>attr [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data-cy</td>
</tr>
<tr>
<td>rendered</td>
<td></td>
</tr>
<tr>
<td>uri</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>EditKeywordButton</td>
</tr>
</tbody>
</table>

- Dev Tools now shows data-cy tag

```xml
<button data-cy="EditKeywordButton" id="view\_id1\_id2\_id343\_button1" name="view\_id1\_id2\_id343\_button1" type="button" class="btn btn-default">Edit</button>
<button id="view\_id1\_id2\_id343\_button3" name="view\_id1\_id2\_id343\_button3" type="button" class="btn btn-default">Delete</button>
```

```javascript
// can now use once data-cy attr added to element
cy.get('[data-cy="EditKeywordButton"]').click();
```
Best Practices (1/2)

• Take care to not inadvertently leave authentication credentials (or other personal data) in test scripts or related files.

• When creating a new test, always start with an initial failing test - the test tests the code, and the code tests the test!

• Begin with simple, generic tests (basic page navigation etc.) to build experience, then add new more complex tests as your test script skills develop (new features, bug fixes etc.)
Use a dedicated element selector tag (such as “data-cy” for Cypress) as an alias, to uniquely identify and therefore directly target elements for E2E testing.

Be mindful when testing date specific functions which might result in flaky tests (for example, before, on or after a specific date).

As a developer, only write tests for your own code.
Cypress Weaknesses

• Multi-tab (browser) based applications/external links
• iFrames (selecting or accessing elements within it)*

*Was flagged as ‘Planned’ on the Cypress Roadmap
Summary

• Automated testing can be beneficial to shipping quality applications
• Awareness of general testing types and testing terminology
• Demonstrated how to begin implementing simple E2E tests using Cypress
• Automated testing can be a steep learning curve, but the technical and non-technical business benefits of improved productivity and business reputation etc., should far outweigh the implementation cost and effort many times over
• You No Longer Need To Let Users Test Your Code!
Thank You for Listening!

Testing Matters - Don’t Let Users Test Your Code!

Do you have any questions for us?
Resources

• Cypress
  • https://www.cypress.io/

• Cypress API and Commands
  • https://docs.cypress.io/api/table-of-contents

• Cypress Plugins
  • https://docs.cypress.io/plugins/directory

• Cypress Cloud (paid for option)
  • https://www.cypress.io/cloud/

• Useful Cypress Blog articles:
  • When Can The Test Click? => https://www.cypress.io/blog/2019/01/22/when-can-the-test-click/
  • When Can The Test Start? => https://www.cypress.io/blog/2018/02/05/when-can-the-test-start/