

# De18: 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
- HCL Ambassador 2021 and 2020

*Email: [paul.harrison@focul.net](mailto:paul.harrison@focul.net)*

*Twitter: [@PaulHarrison](https://twitter.com/PaulHarrison)*

*Skype: [edt.paul](https://www.skype.com/people/edt.paul)*



# 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 😊”
- Ringing Master at Ripon Cathedral

*Email: [martin.davies@focul.net](mailto:martin.davies@focul.net)*

*Twitter: [@martin\\_davies](https://twitter.com/martin_davies)*



# Agenda

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

# Intro – Reasons Not To Test

- Resource expensive – not just the initial test plan generation, but its ongoing maintenance
- Lack of resources run the tests
- Testing environments can be complex to setup and manage
- Commercial testing tools were prohibitively expensive, particularly for smaller organisations
- Yet more things to learn
- Inconsistent and prone to human error

# Intro – Reasons To Test (1/2)

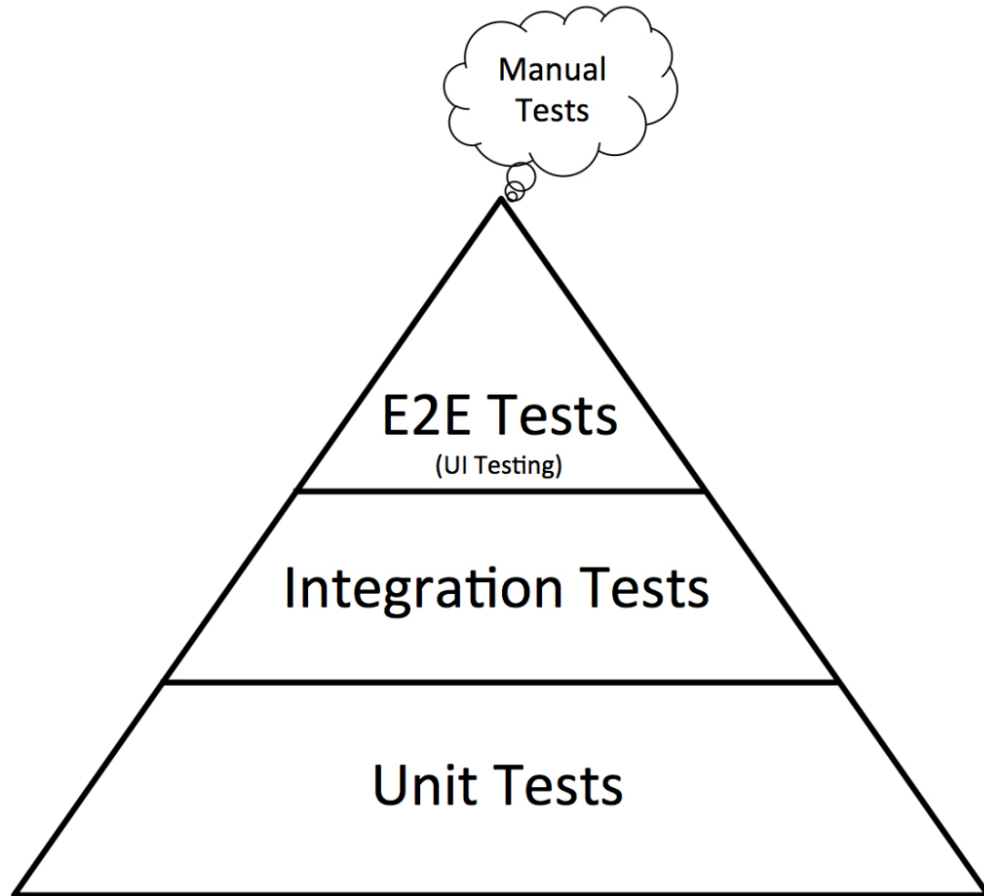
- Gives your organisation and your customers confidence in your shipped products
- Allows you to more frequently ship application updates
- Option to integrate into a code release pipeline (CI/CD\*)
- Testing can ultimately result in fewer shipped bugs
- Reduced support tickets

*\*Continuous Integration / Continuous Delivery or Deployment*

# Intro - Reasons To Test (2/2)

- Can save time and resource costs in the medium to longer term
- Allows for a larger number of tests to be run in a shorter period
- Consistent and highly repeatable
- Self-documenting
- Reusability of test suites
- Empowers Developers to test their own code as it is being developed

# Testing Approaches



Each layer of the pyramid has a different size, indicating the number of tests that should be written within each stage

## End-to-End (E2E)

- Tests the application UI independently of its actual code
- Runs at application-level
- Framework agnostic

## Integration

- Tests how libraries or packages of functions integrate and interact with each other
- Runs at code-level
- Framework specific

## Unit

- Tests basic functions
- Runs at code-level
- Framework specific

Credit: <https://www.ministryoftesting.com/dojo/lessons/the-mobile-test-pyramid>



# Testing Methodologies

## **Design Driven Development (DDD)**

- Develop tests *after* coding
- Easier to start with, as you likely already have some code to test
- Better suited to E2E testing, or when 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 (similar to flowcharting prior to coding)
- Can be a difficult concept to grasp - should perhaps be considered as a long-term goal, although it may not suit or be appropriate for all developers
- 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 actually tested
- Your goal should not necessarily be 100% test coverage of the entire app (Unit, Integration and E2E tests could have different percentages), but rather to ensure that you test a reasonable percentage of *your* code, and focus on those areas that are critical
- 60% - 80% is usually considered good
- For E2E, also include user workflows ('journeys' or 'stories')

## Consistent Test Data

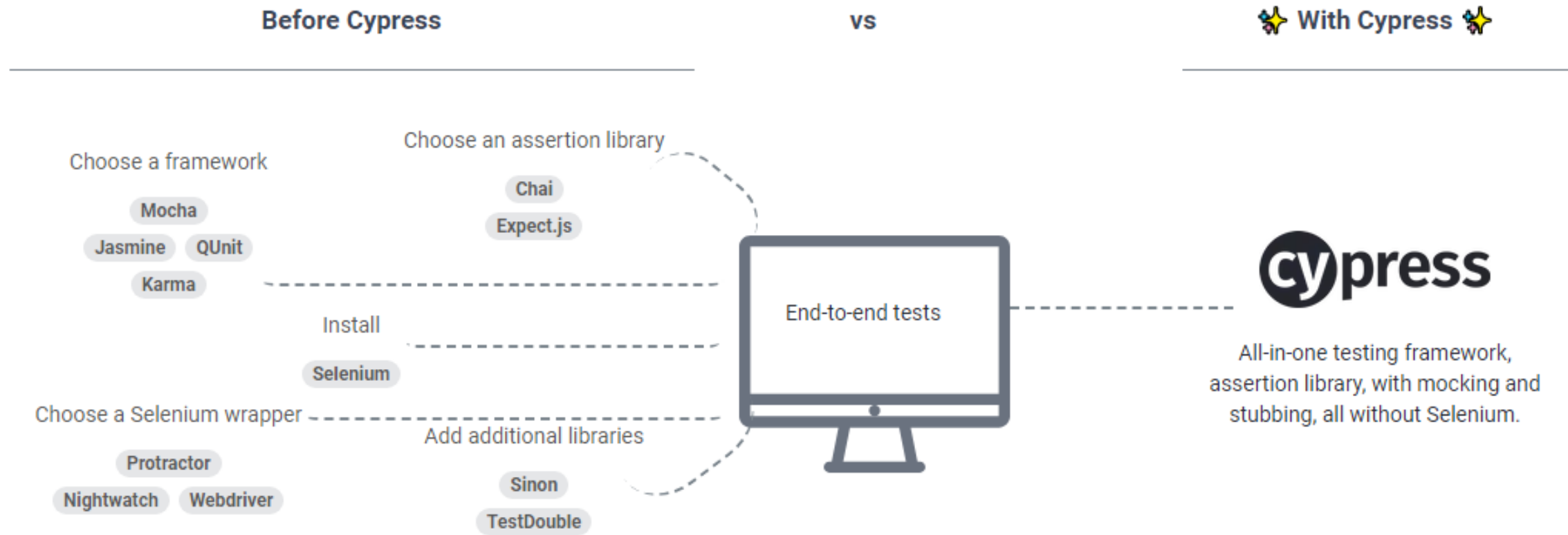
- Sample data (fixed or easily resettable)
- Mock data (artificially inserted into the response payload of an API request)
- Fake data (API that has working implementation, but it is not the same as the production one)

# Focus on End-to-End Testing

- Easy to quickly produce usable tests
- Allows testing of user workflows ('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 actually works
- Does not necessarily require development skills - can delegate to more junior colleagues or QA Team (where applicable)
- Great place to start your testing journey

# Why Cypress (1/4)

- Incorporates some of the “best-in-breed” open source testing libraries



Credit: <https://www.cypress.io/how-it-works>

# Why Cypress (2/4)

- Runs inside the actual browser (tests written in JavaScript/Typescript)
- Very well documented with lots of good examples
- Open Source
- Docker compatible
- Supports “live-reload” and can run in foreground, or headless mode
- Responsiveness capability using pre-set or custom viewport sizes

# Why Cypress (3/4)

- Time Travel – DOM snapshots during test execution and review
- Use existing Dev Tools to examine & debug the application under test
- Automatically waits for commands and assertions before moving on
- Can relatively easily verify and control the behaviour of functions, network traffic, API responses and timers / clocks

# Why Cypress (4/4)

- Extensible - custom functions, and 3<sup>rd</sup> party custom plug-ins
- Screenshots – programmatically, or on failure (in headless mode)
- Video – records a video of the test suite execution (in headless mode)
- Cross browser testing – run tests locally within Firefox and Chromium based browsers (Chrome, Edge and Electron etc.), or in a CI/CD pipeline (headless mode)

# Cypress Dashboard (Paid-For Option)

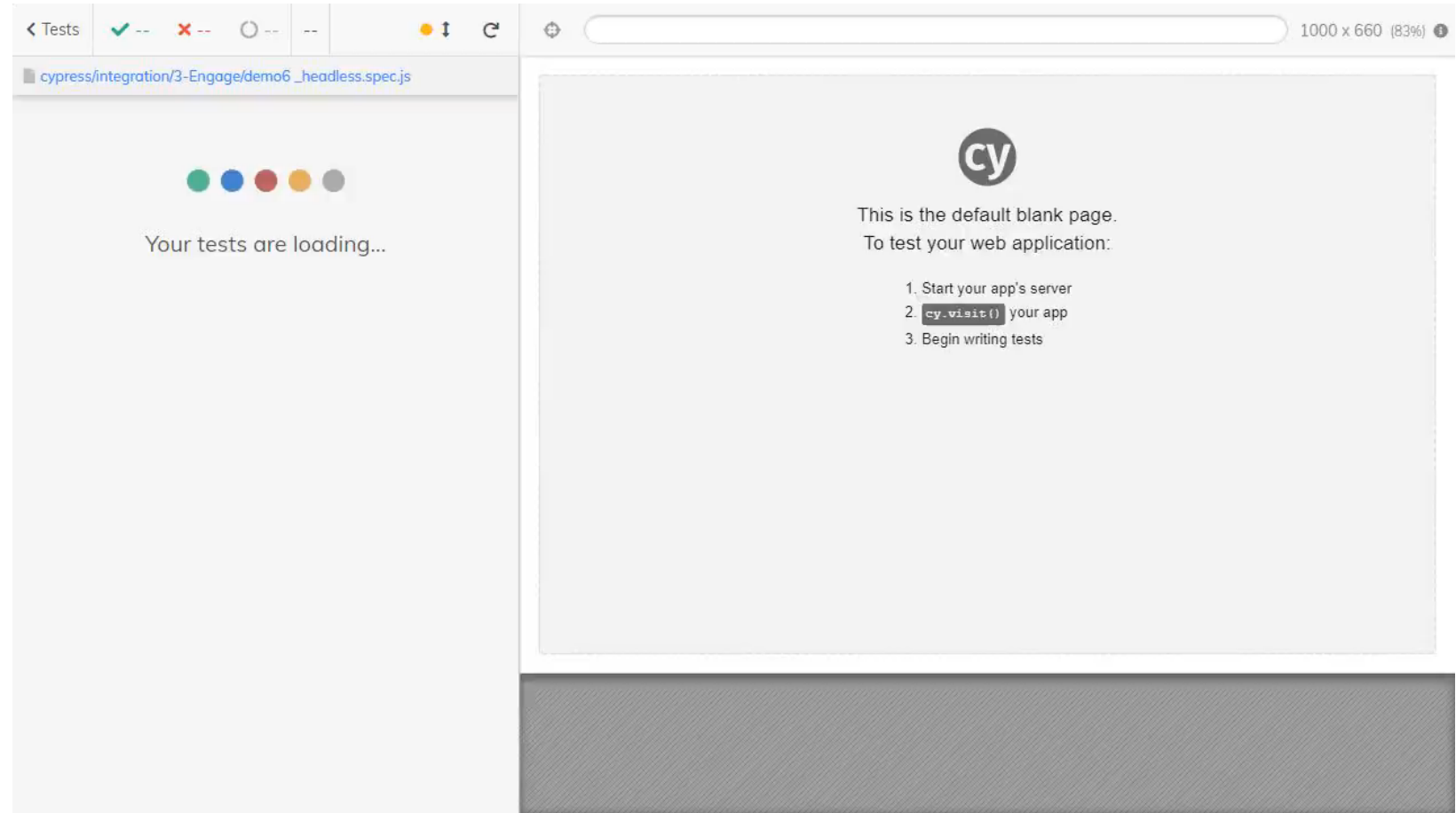
Cypress Dashboard is an optional web-based paid for service (an initial free tier is available) which provides additional features

- 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*)

## *Default Folders*

- fixtures (*store data objects used in testing*)
- integration
  - Where we write out 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*)

```
Directory of \Testing\project1\cypress
09/12/2021 18:17 <DIR> .
09/12/2021 18:17 <DIR> ..
10/05/2022 17:17 <DIR> downloads
09/12/2021 18:11 <DIR> fixtures
09/05/2022 22:30 <DIR> integration
09/12/2021 18:11 <DIR> plugins
09/12/2021 18:11 <DIR> support
```

## *Additional Output Folders*

- downloads
- screenshots
- videos

# Anatomy of a Simple Test

Define Test Suite

Define Test

Commands

Assertion

Define Test

Commands

Assertion

# Anatomy of a Simple Test

Define Test Suite

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

Define Test

```
  it('Test1', () => {
```

Commands

```
    cy.visit('website')
```

Assertion

```
    cy.get('object')
```

```
      .should('have.length', 2)
```

Define Test

```
  })
```

Commands

```
})
```

Assertion

# 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 Testss
  - 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])*



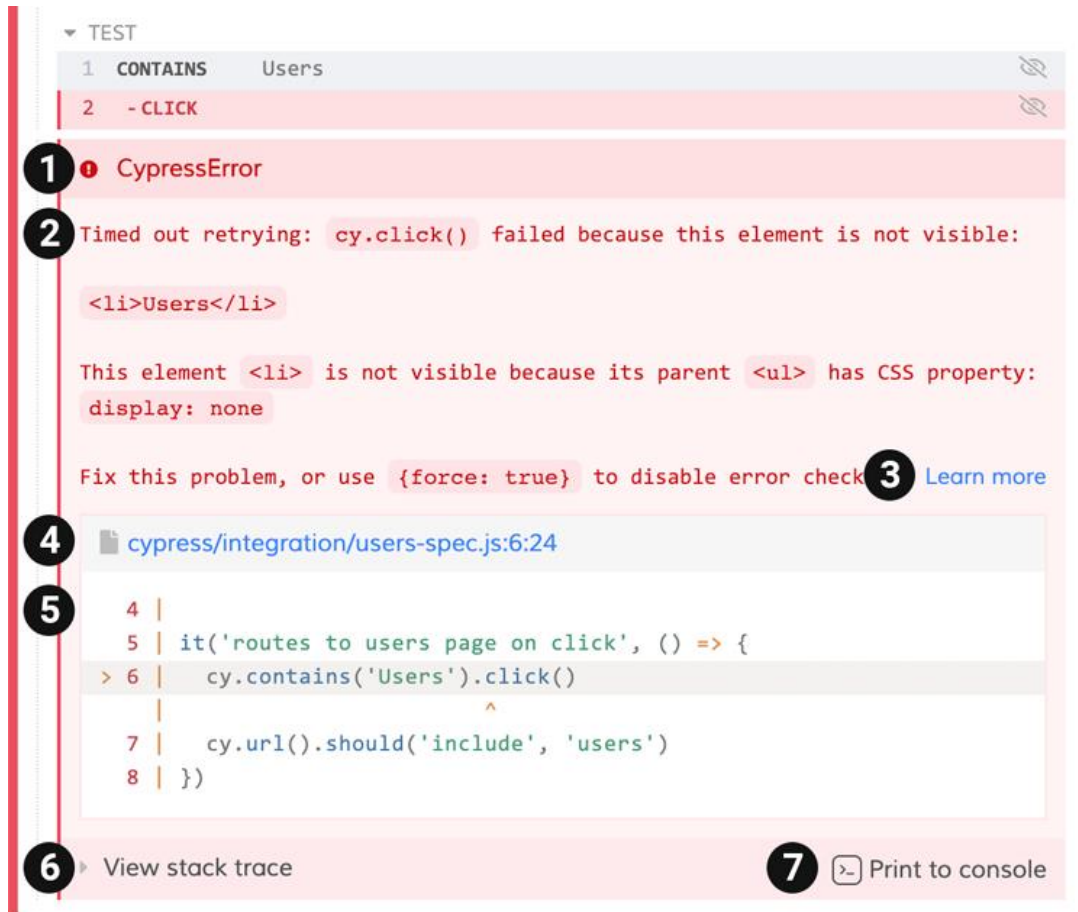
# Cypress Testing UI

- 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



The screenshot shows a Cypress test failure in the DevTools console. The error is a **CypressError** with the message: "Timed out retrying: cy.click() failed because this element is not visible: <li>Users</li>". It explains that the parent **<ul>** has **display: none** and suggests using **{force: true}** to bypass the error check. A "Learn more" link is provided. The stack trace shows the file **cypress/integration/users-spec.js** at line 6, column 24. The code snippet shows a test suite for "Users" where **cy.contains('Users').click()** is called. At the bottom, there are buttons to "View stack trace" and "Print to console".

1. **CypressError**

2. Timed out retrying: `cy.click()` failed because this element is not visible: `<li>Users</li>`

This element `<li>` is not visible because its parent `<ul>` has CSS property: `display: none`

Fix this problem, or use `{force: true}` to disable error check [3. Learn more](#)

4. `cypress/integration/users-spec.js:6:24`

5. 

```
4 |  
5 | it('routes to users page on click', () => {  
> 6 |   cy.contains('Users').click()  
   |                               ^  
7 |   cy.url().should('include', 'users')  
8 | })
```

6. View stack trace

7. Print to console

**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.

# 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

# Headless Test

- 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"`

# Best Practices (1/2)

- Be careful not to 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!
- Keep initial tests simple and generic (basic navigation etc.), then add new tests as your development progresses (new features, fixes etc.)
- Use a dedicated E2E tag (such as *data-cy* for Cypress) to uniquely identify and therefore directly target elements for testing

# Best Practices (2/2)

- Be mindful when testing date specific functions which might result in flaky tests (before, on or after a specific date)
- Only write test for your own code to avoid possible duplication by other developer

# XPages Hints and Tips – Clicking

- We have occasionally observed click failures when using the regular `.click()` approach
- Alternative clicking methods are available if required

```
// #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();
```

# XPages Hints and Tips – Selectors

- 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” (prefix with \\) otherwise the element will not be found when the test actually 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
- Note that Cypress Studio seems to work correctly



- Using Dev Tools => FAIL



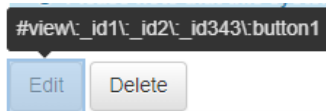
```
<div class="lotusForm2Buttons">
  <button id="view:_id1:_id2:_id343:button1" name="view:_id1:_id2:_id343:button1"
    type="button" class="btn btn-default">Edit</button>
  <button id="view:_id1:_id2:_id343:button3" name="view:_id1:_id2:_id343:button3"
    type="button" class="btn btn-default">Delete</button>
</div>
```

```
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.

- Using Selector Playground => FAIL



```
cy.get('#view\:_id1\:_id2\:_id343\button1') // 1
```

```
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 all : with \\ => SUCCESS

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

# XPages Hints and Tips – data-cy Tag

- Because selectors in XPages automatically generated, they can unexpectedly change, resulting in a failed test
- Best Practice is to add a dedicated (and unique) *data-cy* tag to your markup, which then allows direct selector targeting
- *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

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



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

- Using Dev Tools before adding data-cy tag



```
<div class="lotusForm2Buttons">
  <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>
```

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

- Add data-cy tag attribute in Domino Designer

Property	Value
> accessibility	
v basics	
v attrs	
v attr [0]	
loaded	
minimized	
name	data-cy
rendered	
uri	
value	EditKeywordButton
binding	

```
<xp:button value="Edit" id="button1">
  <xp:this.rendered><![CDATA[#{javascript:if (document1.getItemValueString("DBDesignOnly_Tx") == "Yes") {
    return sessionBean.currentUser.isDBDesign() && !document1.isEditable() ;
  } else {
    return !document1.isEditable() ;
  }}]></xp:this.rendered>
  <xp:this.attrs>
    <xp:attr name="data-cy" value="EditKeywordButton"></xp:attr>
  </xp:this.attrs>
  <xp:eventHandler event="onClick" submit="true"
    refreshMode="complete">
    <xp:this.action><![CDATA[#{javascript:var doc:NotesDocument = document1.getDocument();
    var url=view.getPageName()+"?action=editDocument&documentId="+doc.getUniversalID()
    context.redirectToPage(url, false)}}]></xp:this.action>
  </xp:eventHandler>
</xp:button>
```

- data-cy tag now shows using Dev Tools

```
<div class="lotusForm2Buttons">
  <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>
</div>
```

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

# Cypress Weaknesses

- Multi-tab (browser) based applications / external links
- Due to the asynchronous nature of Cypress, we are still finding that waits are often required (may just be our unfamiliarity) – see references for related blog articles
- iFrames (selecting or accessing elements within it)\*
- Does not currently support WebKit based browsers, such as Apple Safari\*

*\*Currently flagged as 'Work in progress' on Cypress Roadmap, but in the meantime, other E2E tools are available which may better fit your current requirements:*

- *Nightwatch*
- *Playwright*
- *WebdriverIO and Appium (Mobile native testing)*

# Summary

- Automated testing is beneficial to shipping quality applications
- It can be a steep learning curve, but the non-technical business benefits of improved productivity and business reputation etc., should far outweigh the implementation cost and effort many times over
- Enables integration with automated code release CI/CD pipelines
- Awareness of general testing terminology
- Understanding of how to begin implementing a simple E2E test using Cypress
- You No Longer Need To Let Users Test Your Code!

# Thank You!

Do you have any questions for us?

*... and please be so kind as to complete the evaluation for this  
Session (De18) via the Engage 2022 mobile agenda app*

# References

- 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 Dashboard (paid for option)
  - <https://www.cypress.io/dashboard/>
- 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/>