AMI Cypress Tests

Cypress.io

Cypress Documentation

The Cypress website is here, and you can find their Guides here.

Installing Cypress

1. Make sure that you have an up to date version of AMI's source code, including the **AMI/endToEndTests** folder.

2. Make sure that you have Node.js installed. (We recommend you use the LTS version.)

3. In PowerShell, go to your **AMI/endToEndTests** folder and type:

```
npm install
```

4. Also in PowerShell, type:

```
    npm install -g junit-merge
```

   This utility is needed by the runCypress.ps1 script (see below)

5. In the same folder, create a file called **cypress.json** looking something like this:

```
{
"baseUrl": "https://localhost:3002",
"video": false,
"viewportWidth": 1300,
"viewportHeight": 860
}
```

   Change the baseUrl setting to be wherever you normally get AMI from. The two viewport settings affect the browser size when running Cypress tests. Change or remove them as you please.

Running Cypress

1. Launch the Cypress UI by running the script **AMI/scripts/openCypress.ps1**.

2. Run all the Cypress tests by running the script **AMI/scripts/runCypress.ps1**.

By default all the tests will run in **index.html**; however, you can override this by using the **-dev** switch:

```
 openCypress -dev
runCypress -dev
```

in which case they will run in **devindex.html**.

Also by default, the tests will run using your default Agility Server. However, the Cypress tests should expect to run using the WEBTST database, not NWC72, say. To override this, do the following:

1. Use SSAMI as your web server
2. Specify the alternate Agility Server in SSAMI's **config.json** file in the **alternativeServers** section
3. Create a file called **AMI/endToEndTests/cypress.env.json** with the following content:

```
{
        "agilityServer": "<name of server as used in config.json>"
}
```

Now when you run Cypress, it will pick up the value of **agilityServer** and attach it to AMI's URL. At the time of writing an Agility Server that is connected to the WEBTST database is at **http://172.24.5.159:8080**.

Writing Tests

1. As well as the tests we write, the ones in **AMI/endToEndTests/cypress/integration**, there are some example files from Cypress in **AMI/endToEndTests/cypress/examples**.
2. Be aware that Cypress contains **jQuery** and **lodash** (an alternative to the underscore library), which you access as **Cypress.$** and **Cypress._**. It also provides the functionality of the Mocha, Sinon and Chai unit testing frameworks.
3. If Cypress has the browser open with a test, then it will automatically re-run the test as you edit its source.
4. You can use the debugger inside Cypress's browser.
5. On the left hand side Cypress shows you all the commands it has run in the current test. If you mouse over these, Cypress will show you the state of the browser at that pointer.
6. There are four standard AMI users that can be used in tests: **cypress1**, **cypress2**, **cypress3**, and **cypress4**. To avoid clashing with these when you're developing your own tests or debugging existing ones, you can override the username that's specified in a test by editing your **cypress.env.json** file and adding an entry for **userName**; e.g.

```
{
"agilityServer": "<name of server as used in config.json>",
"userName": "canderson"
}
```

File Location and Naming Convention

Save individual cypress test files in to an appropriate folder. When related to a particular JIRA Wave, prefix the test filename with the JIRA number and make the filename descriptive.

- Eg. WAVE-4676 relates to the Relations gadgets
- File location: ../cypress/integration/relationGadgets
- Filename: WAVE-4676-Relations-gadgets-contain-select-suffix-attribute.js

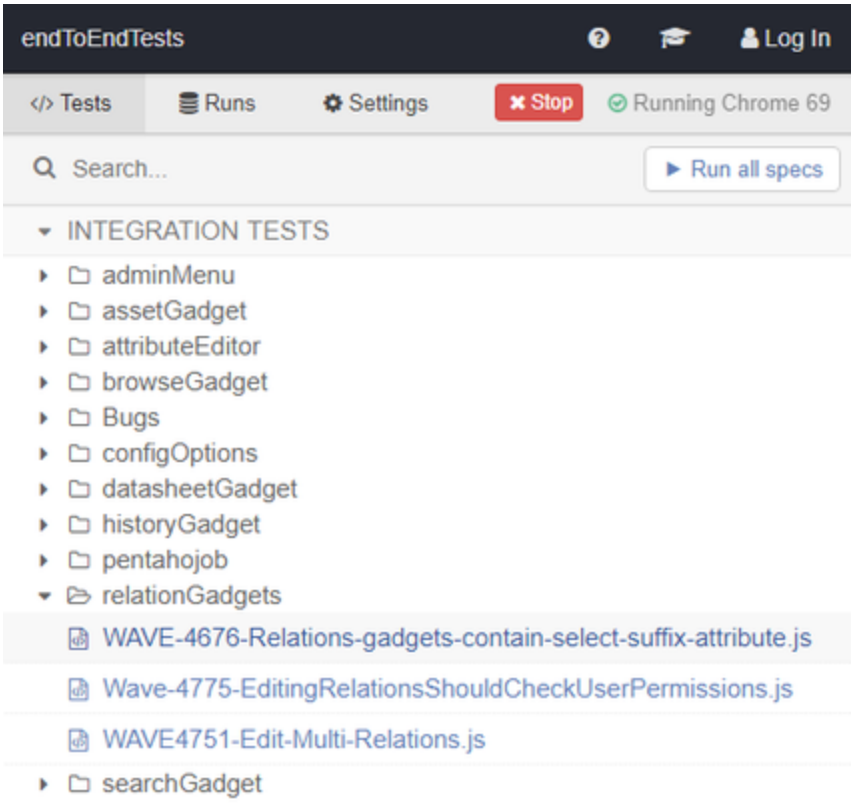
Example Test

```
describe('WAVE-4676 The Product and Brand Relations gadgets have an extra menu item called "Select Suffix
Attribute..."', function() {
it('Test to make sure options appear and they function correctly', function() {
// put cypress commands here
// the following line is to test that your test is working, remove from actual tests
expect(true).to.equal(true);
})
})
```

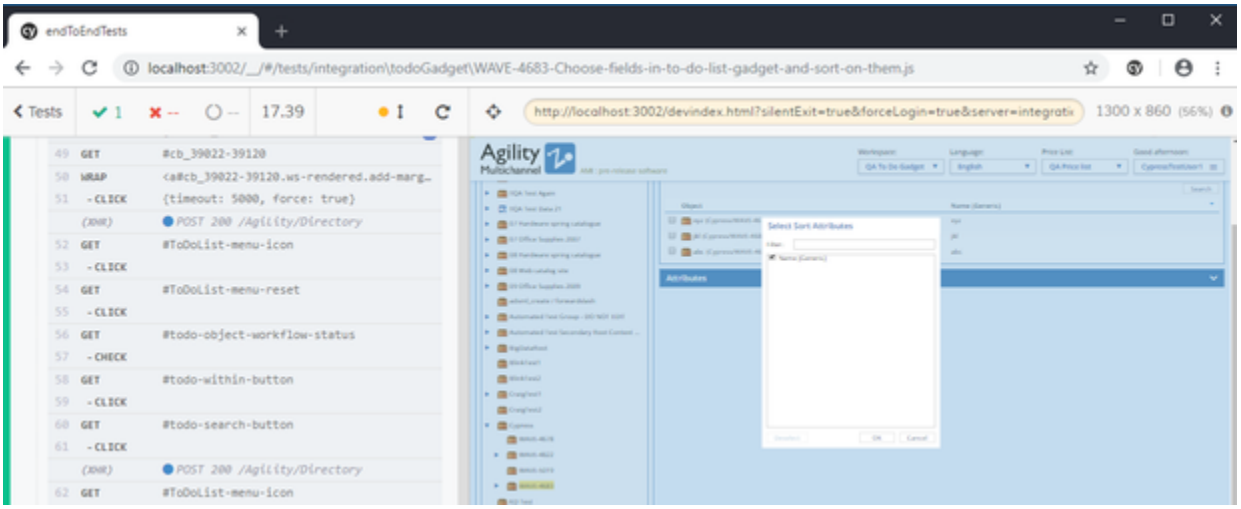
replace the describe and it strings with descriptions relating to the actual test

Launching Tests

Cypress will show the folder hierarchy of your tests. Click the test name to run a test.



Tests run in a new window with the results of the test alongside the page(s) you are testing. Moving your mouse up/down through each step in the left hand section will step back/forward in time in your page view.



Subsequent saves of the file you are testing will automatically re-start the test.

Utilities

- Purpose: These hold re-usable functions useful for your testing
- Location: ../integration/utilities folder
- Example: loginUtils.js - this has a function to login to AMI and a function to log out
- Exporting for use: At the bottom of this file note the module exports and the format: functionName: functionImportName - these are generally the same
- Importing for use - require: At the top of your test file, add: const utilitiyCategory = require('../../utils/utilityFilename');
- Importing in your test file - function call: utilityCategory.functionImportName(parameters);

Debugging Tests

The best place to debug tests is in the cypress console. This has many useful features, such as being able to click on the command history on the left hand side to pin them and see what the browser was showing at that point.

Another useful feature is the cy.pause() function. Putting this into your test causes execution to stop at that point. The Cypress console now displays two icons to let you step or resume execution.

The .debug() function can be chained to most Cypress calls. It will add a line marked 'Debug' to the command history. If you click on this and look in the Chrome console log, you can see details of the function that debug was chained to. For example,

```
cy.get('.some-class').debug().click()
```

will let you find out what DOM elements matched '.some-class'.

Example Utility implementation

At the start of each test you are likely to log in to AMI and at the end of each test logout. The following does both using the two utilities in the loginUtils.js file

```
const loginUtils = require('../../utils/loginUtils');

describe('WAVE-4676 The Product and Brand Relations gadgets have an extra menu item called "Select Suffix
Attribute..."', function() {
it('Test to make sure options appear and they function correctly', function() {
loginUtils.loginToAMI('cypress1');
// the following line is to test that your test is working, remove from actual tests
loginUtils.logoutFromAMI();
})
})
```

Common Cypress commands and examples

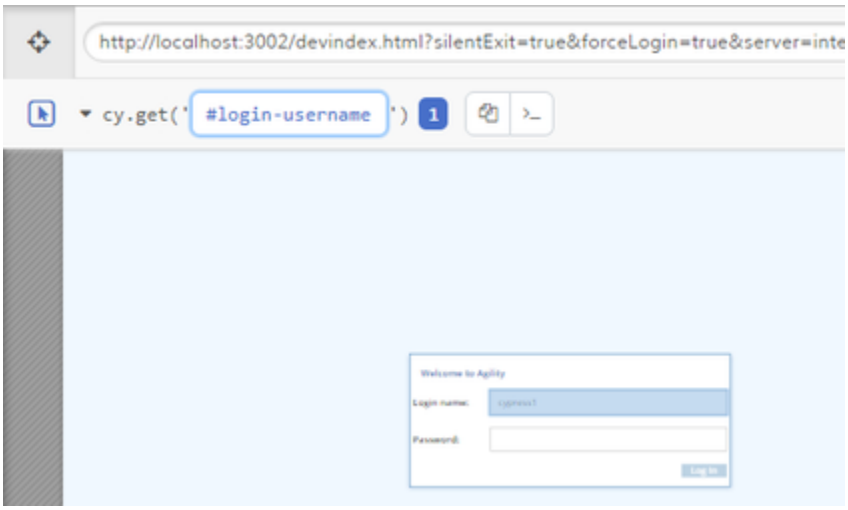
Item	Example description	Example cypress command
get	Locate an item in the DOM with id "itemId"	cy.get('#itemId')
get	Get first td element in #todo-table	cy.get('#todo-table td:first')
click	Locate item with id itemId and click it	cy.get('#itemId').click();
check	Locate the checkbox within the 10th li element within the item with id it emId and check it	cy.get('#itemId li:nth-child(10) input').check();
uncheck	Uncheck first input checkbox in #checkList	cy.get('#checkList input:checkbox').first().uncheck();
find	Find the span which has title of "Create Price List..." within #conf-toolbar and click it	cy.get('#conf-toolbar').find('span[title="Create Price List..."]').click();
type	Type "Hello, World" into #itemId	cy.get('#itemId').type('Hello, World');
should	Test if value in #itemId has value "Hello"	cy.get(#itemId).should('have.value', 'Hello');
should be empty	Get first td element in #todo-table and test if it's empty	cy.get('#todo-table td:first').should('be.empty');
type and should	(type number in to an input box and test it's value - useful for testing inputs with character limits) - type "ABCDE" into #itemId and check if the text is "ABC"	cy.get(#itemId).type("ABCDE").should('have.value', "ABC");
type and should not	Type "ABCDE" into #itemId and check if the text has value "ABC" and not the value "ABCDE"	cy.get(#inputId).type("ABCDE").should('have.value', "ABC").and('not.have.value', "ABCDE");
timeout	Set a time out of up to 5 seconds before clicking the item #itemId (Note , not the same as wait, wait definitely waits that amount of time, timeout allows up to that amount of time for that item to load so is in most instances quicker)	cy.get('#conf-priceLists', {timeout: 5000}).click();
contains	Click the p tag which contains "Hello"	cy.get('p:contains("Hello")').click();
should have prop	See if myCheckbox has property indeterminate	cy.get('#myCheckbox').should('have.prop', 'indeterminate');

should have class	Check if itemId has class myClass	cy.get('itemId').should('have.class', 'myClass');
wait	Wait two seconds (use sparingly as adds to test time, use timeout instead)	cy.wait(2000);

All commands are listed on the left hand side of: https://docs.cypress.io/api/commands/and.html#Syntax

Locating elements in the DOM

- Most commands above need to act on an element, so these elements need to be located. Standard rules for DOM traversing apply and the above gives examples of some of these.
- Use developer tools in your browser to locate code relating to the element you want to locate.
- Use element ids wherever possible as these should be unique.
- Cypress has it's own useful element selector. To activate it click the target element to the left of the URL you are testing. You can then click the arrow icon and then locate the element on the page you are trying to access. The below example targets the login name input box and generates the cy.get command. For further detail see: https://docs.cypress.io/guides/core-concepts/test-runner.html#Running-Experiments



Notes

1. When running Cypress from the command line, it includes VT100 escape sequences to colourise its output. This will not work on PowerShell before v5.1—the sequences will appear similar to this: [90m. However, the **runCypress.ps1** and **openCypress.ps1** scripts filter out these sequences if the PowerShell version doesn't support them.
2. Sometimes eslint will complain that you have an unused expression in your test code. This is likely due to code like this:

```
expect(errorsFound).to.be.false;
```

The linter doesn't know that this Chai expression is actually doing something, so it looks like an unused expression. The solution is to put this line at the top of your test source file:

```
/* eslint-disable no-unused-expressions */
```

Skipping Tests

Although it is sensible to write tests for open AMI bugs, it makes less sense to run them as part of the full test suite, as they are guaranteed to fail. To let Cypress skip such tests, edit the test as follows:

```
describe("WAVE-5071: User Groups tool doesn't refresh when you add or delete user groups", function () {
```

becomes:

```
describe.skip("WAVE-5071: User Groups tool doesn't refresh when you add or delete user groups", function ()
{
```

The same trick can be used for the it() method to skip a single test instead of a whole test spec.

Continuous Integration

There is a Jenkins job that runs all the Cypress tests every weekday morning before we get in. It reports failure if any tests fail, which makes it particularly important to explicitly skip tests that we expect will fail (see above).

To Do

1. Cypress's Chrome browser puts up popup saying it was not shut down properly. It seems to be harmless, but it would be nice to get rid of. Only some Cypress users seem to see it.
2. The Cypress output includes checkmark symbols (ticks) for completed tests, but they are being rendered as ÔêÜ. I have not managed to work out what they really are; otherwise we could replace them in runCypress.ps1 and openCypress.ps1.