UI tests

Magnolia 5.5 and above

This page and its subpages are outdated partially.

For a more up-to-date documentation regarding UI tests for Magnolia 6.1 and higher, please also check the documentation on bitbucket.

Caution

You need the docker-engine installed on your local machine. (For macOS have a look here)

Location

All UI-Tests are located in the bundles under either `ce/magnolia-integration-tests/tests` or `ee/magnolia-ee-integration-tests/tests`

Running the tests

Automated run

We use *Selenium* for testing our ui and *HtmlUnit* for our integration tests. The UI-Tests will are part of `magnolia-integration-test/tests` and use its `test-webapp` & `test-public-webapp`. UI-Tests are only triggered if you specify the corresponding maven profile `ui-tests`.

In short, use the following command to automatically run all UI-Tests

launch uitests

```
..../magnolia-integration-tests/tests mvn clean install -P ui-tests
```

Run all tests in a single class

launch uitests for one class

```
..../magnolia-integration-tests/tests mvn clean install -P ui-tests -DfailIfNoTests=false -Dit.test=MyTestClass
```

Run a single test case

launch uitests for one class

```
..../magnolia-integration-tests/tests mvn clean install -P ui-tests -DfailIfNoTests=false -Dit.
test=MyTestClass#myTest
```

Manual run

If you want to run the ui tests manually to debug the ui-test code you could basically just run the test in our IDE. The drawback with this simple approach is that the whole environment with the web & selenium containers including magnolia installation is done with every start.

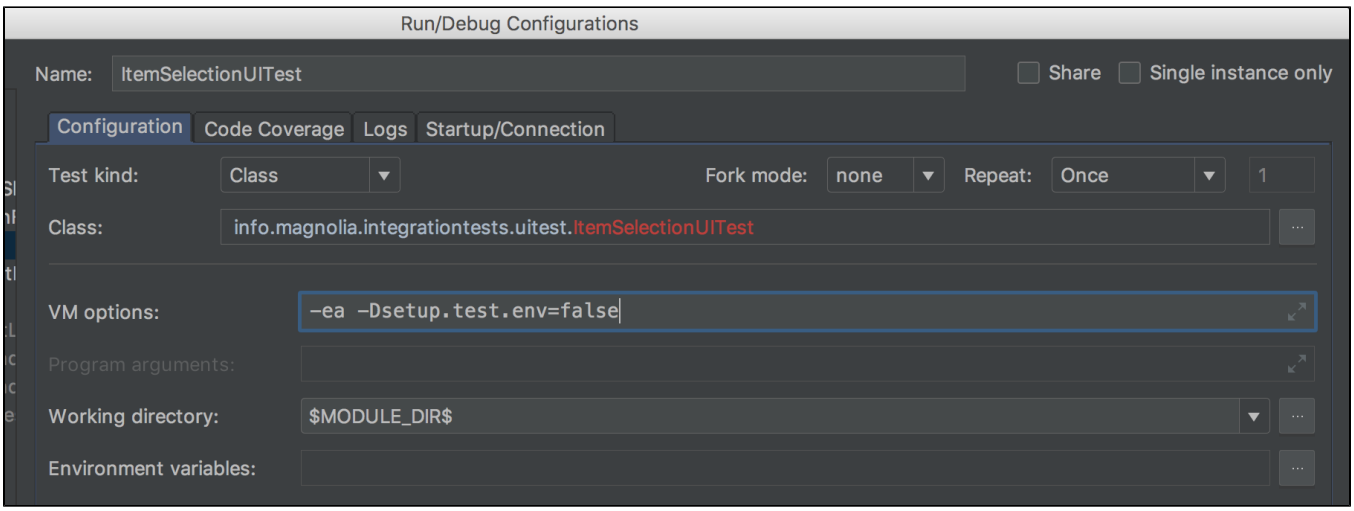
So to have a smaller footprint we first have to start just the web container with magnolia and the selenium host with the following script

launch tomcat and selenium container

```
$ mvn clean install
$ ./localtest.sh
```

(It's important to run install before because `localtest.sh` will take the produced webapp wars and create a docker image out of it)

Then you have to launch the test with `-Dsetup.test.env=false` so the test environment is not started and stopped with every test launch (thats what you have already done with `localtest.sh`)



To observe the browser with the test execution launch a VNC client and connect to `localhost:5901` with password "`secret`". (Under macOS that easily done with the app *Screen Sharing*)

Tip

Giving Docker more resources than the default seems to help making the local testing environment more stable and faster. E.g.

No Docker Setup (Manual run)

If Docker performance on MacOS is questionable in your case then you would want this setup method, which is:

+ Start test webapps locally

+ Run Selenium server locally

+ Run UI tests from your IDE and monitoring test execution with local Chrome browser

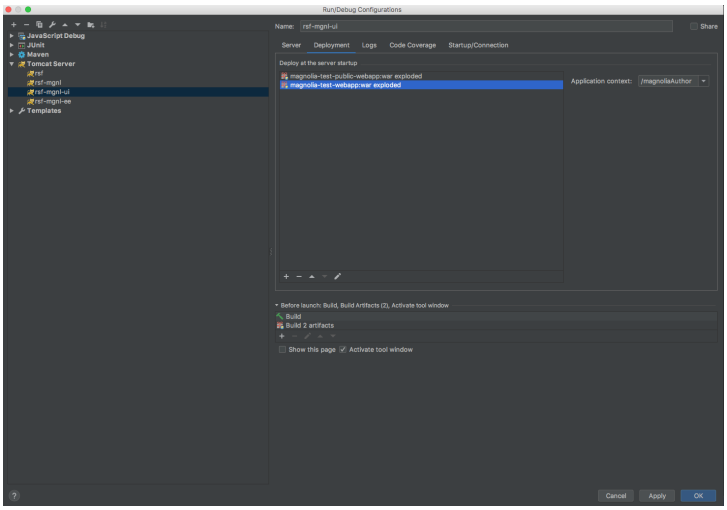
Step:

Patch CE integration test setup, so that it can by-pass completely docker related calls, can be done with this patch:

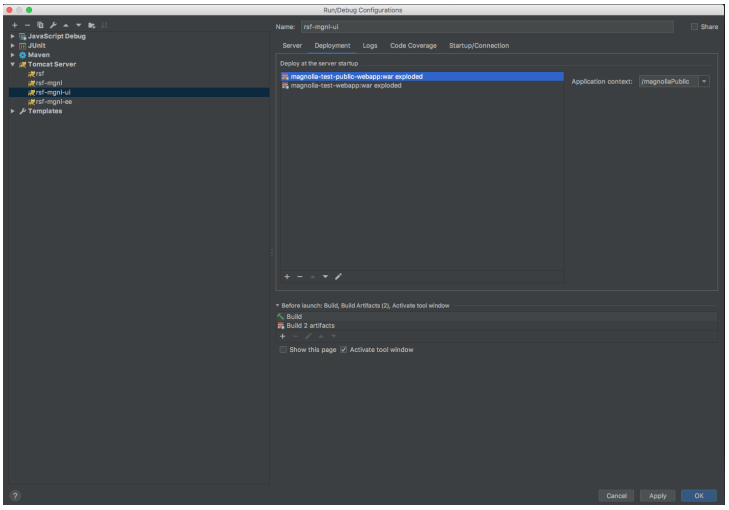
no_docker_patch.diff

Start test webapps (author and public) ensure that selenium server can communicate with them via:

+ http://localhost:8080/magnoliaAuthor



+ http://localhost:8080/magnoliaPublic



Start Selenium server (with chrome driver in the same folder)

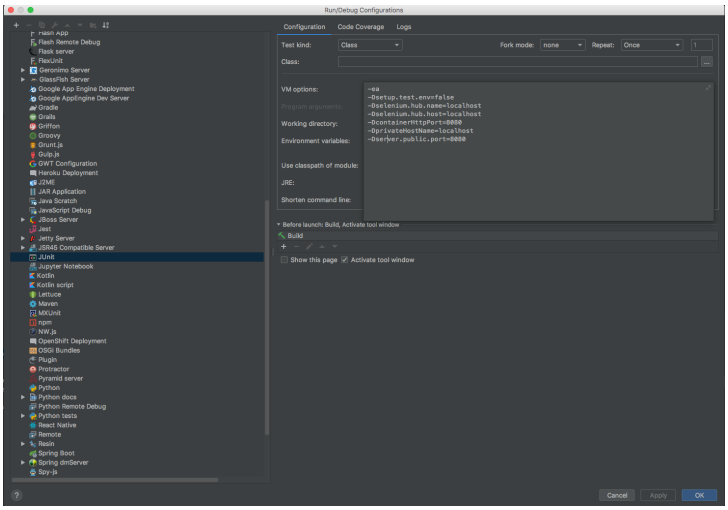
selenium

```
macos:selenium user$ ls -l

total 69216

-rwxr-xr-x  1 user  staff  12012868 Mar  1  2018 chromedriver

-rw-r--r--@ 1 user  staff  23422300 Mar 16  2018 selenium-server-standalone-3.11.0.jar


macos:selenium user$ java -jar selenium-server-standalone-3.11.0.jar
```

Launch tests with additional params

> **params**
>
> -ea -Dsetup.test.env=false -Dselenium.hub.name=localhost -Dselenium.hub.host=localhost -DcontainerHttpPort=8080 -DprivateHostName=localhost -Dserver.public.port=8080

Here is Junit test template that applied to all new test profile.



Test will be executed in a new Chrome browser window.

Alternative No Docker Setup (Manual run)

Another alternative method to run UI Tests without relying too much on Docker. This method has some advantages compared to the previous No docker setup:

+ Don't have to manually setup selenium every times

+ UI clients moved to docker so we can avoid test browser to be opened randomly on active monitors.

Step 1: Prepare Docker containers

docker commands

```
docker network create uitest
docker run -d --network uitest -p 4444:4444 --name hub selenium/hub
docker run -d --network uitest -e HUB_PORT_4444_TCP_ADDR=hub -e HUB_PORT_4444_TCP_PORT=4444 -p 5901:5900 --name
client1 selenium/node-chrome-debug
docker run -d --network uitest -e HUB_PORT_4444_TCP_ADDR=hub -e HUB_PORT_4444_TCP_PORT=4444 -p 5902:5900 --name
client2 selenium/node-chrome-debug
```

+ Create two docker virtual UI clients (client1, client2) which connect to one docker selenium hub (in the same shared network called "uitest". The reason for two virtual clients is that with only one client, there would be higher chances of unclear slow down and un-responsive. With two clients, selenium hub will automatically switch back/forth and tests run smoother.

Step 2: Apply patch to by-pass docker environment setup in ce

Step 3: Manually start Magnolia webapps as described in above No Docker Setup

Step 4: Run ui-tests with additional VM options

Additional VM options

```
-ea -Dsetup.test.env=false -Dselenium.hub.name=localhost -Dselenium.hub.host=localhost -DcontainerHttpPort=8080
-DprivateContainerHttpPort=8080 -DprivateHostName=<one-of-host-real-ip>
```

***** <one-of-host-real-ip>: can be either IP of wireless or wired adapter, so if the host is configured with DHCP, that address must be updated every run. We can create a virtual network adapter and assign that adapter with some fixed IP to avoid edit that param many times.**

Magnolia 5.4

Location

Our ui tests are placed in the ce-bundle/magnolia-integration-tests/tests because all the required setup (install and start an author and public instance) is already there.

Running the tests

Caution

Some of those tests are very sensitive - when running them on your local machine, make sure to not do anything else while they're running. e.g. don't switch to any application or type anything. This can be enough to make some of the UI tests fail.

You need docker installed as well as both the tomcat container as well as the browser for the ui-tests are started as docker containers. (For macOS have a look here)

Automated run

We use Selenium for testing our ui. The UITest will be part of magnolia-integration-test/tests and use its test-webapp & test-public-webapp. UITests are only triggered if you specify the corresponding profile (ui-tests).

In short, use the following command to automatically run the ui tests

launch uitests

```
..../magnolia-integration-tests/tests mvn clean install -P ui-tests -DseleniumBrowser=chrome
```

Run only the tests in one class

launch uitests for one class

```
..../magnolia-integration-tests/tests mvn clean install -P ui-tests -DseleniumBrowser=chrome -
DfailIfNoTests=false -Dit.test=MyTestClass
```

(Note that this will also trigger the crawling after the test, it would be nice to figure out a command that runs one class, but does not trigger the crawling.)

Run only one test

launch uitests for one test

```
..../magnolia-integration-tests/tests mvn clean install -P ui-tests -DseleniumBrowser=chrome -
DfailIfNoTests=false -Dit.test=MyTestClass#myTest
```

Manual run

If you want to run the ui tests manually from within your IDE you can start the author and public tests instance with

launch uitests

```
..../magnolia-integration-tests/tests mvn clean install -P manual-tests
```

Once this is running, then you can simply run or debug with JUnit in your IDE as you would a normal unit test.

Outlook

- Execute tests with different browsers (Firefox, Chrome, Safari, Ie, ...) an different OS's (OSX, Unix, Windows, iOs, ...) simulating different devices (e.g. ipad as well).

Implementation hints

Our UI tests are implemented with selenium. Despite the fact that this tool is really mature, those tests aren't as reproducible as ordinary unit-tests. Here's a list of hints that should help to write stable magnolia ui tests:

Issue	Potential fix	Remark
Element cannot be found although it's there	~~add a delay~~ Fix your xpath, or use one of the `waitUntil` methods.	querying for an element (`AbstractMagnoliaUITest#getElementByPath(By)`) will explicitly try for 5 seconds - if the test triggers a long running action (e.g. activation) this can take even longer so we might have to add an additional, explicit delay
Element is found although it should be gone	~~add a delay~~ Fix your xpath, or use one of the `waitUntil` methods.	unlike in the case where we check for existence of an element we don't have any implicit or explicit delay here - if the element needs some time to go away (e.g. Overlay fadeout) we have to add an explicit delay. use `waitUntil(elementIsGone()`
Input field value cannot be queried with xpath	dont use xpath	input[@class = 'classname' and @value = 'form input...'] could be changed to input[@class = 'classname] and use a condition like `waitUntil(attributeToBe(locator, "value", "form input..."))` to query the input value.
Form validation fails, even if fields are properly entered	ensure blur / change	after filling an input with sendKeys, one should explicitly blur the field - i.e. click anywhere else - and allow some time for the change event to occur. Only then, pressing 'save changes' will be properly aware of the modifications.
Getting another element instead of the expected one	scope XPath queries	making dead simple queries like //input[@class='v-textfield'] should be carefully considered, there may be more elements of same kind (inputs, buttons) currently loaded in the UI. Try to scope selectors at least to subApp level.
Querying for descendent elements	use .// XPath prefix	invoking `parent.findElement(By.xpath("//something"))` doesn't query only for sub-elements of parent. To evaluate an XPath expression relative to parent WebElement, one needs to use the .// prefix instead.

If you find it hard to create XPath queries, ~~you might find the following helpful:~~

1. ~~Firefox console can evaluate XPath expressions, through the `$x()` function~~
2. ~~Firebug + **FirePath** (depending on Firebug), which makes it easy to test xpath statements~~
3. ~~Firefox XPath Checker extension~~

If writing an XPath for the element you're looking is hard, XPath should probably not be used for this case.

Screenshots of each stage of the test

Our framework is rigged to generate a screenshot and save it during the test - this can help debug what went wrong with the test. These images are stored in tests/target/surefire-reports/

Even on hudson you can access these screenshots:

http://hudson.magnolia-cms.com/view/Product_Team/job/magnolia-bundle_trunk-with-selenium_profile/ws/magnolia-integration-tests/tests/target/surefire-reports/