


You are in: [DevOps Technical Workshop Wiki](#) > [V2 Lab III - Revenge of the Functional Testing](#) > V2 Lab 3 - Steps 2E & 2F

## V2 Lab 3 - Steps 2E & 2F

[Like](#) | Updated 7 December 2017 by [Roy Mitchley](#) | Tags: *None*



Anything in this box needs to be edited in Atom Text



This box contains lists of commands that should be executed in order on the Terminal

### (E) Enhance the pipeline with end-to-end tests

**The purpose of this lab is to add automated user testing / e2e testing into our pipeline and then to write e2e tests that test the functionality of our new feature.**

End-to-end (e2e) tests effectively simulate user behavior by driving a browser. They are called end to end because they often drive lots of functionality to simulate whole or part of a user journey through the application. Assertions can be made against both the model/data in the application, and the view by ensuring the ui looks a certain way for particular circumstances.

Currently our pipeline builds, tests client-side code and server-side code, and deploys the application. In this lab we will enhance our pipeline so that our e2e tests run against the deployed ci and si applications.


- Open the e2e spec in Atom and take a look at the layout of the spec



Open `e2e/todos/todos.spec.js`

Jasmine is used as the assertion language for these tests, so you will see that the syntax looks very similar to that of the client-side tests. However, a major difference to the client-side tests is that the actual execution of the tests do not happen on the browser, so we do not need a runner like Karma. Selenium is a very popular tool for driving the browser with code/configuration. Protractor is built on top of Selenium and provides some additional functionality for testing AngularJS applications.

- Protractor needs the chromedriver in order to drive chrome. Run the npm script to update the webdriver:



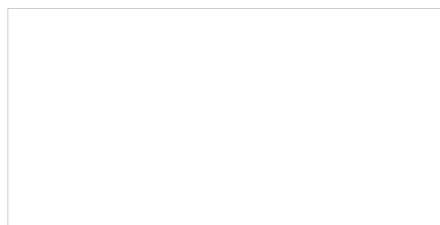
```
cd ~/todolist
npm run-script update-webdriver
```

- To see run the e2e tests against our local application, run the following in the terminal from `~/todolist` directory:




```
grunt test:e2e
```

Protractor will finish and report success in the terminal as follows:



When the tests were running, you would have seen chrome open. If you brought the window to the front, you would be able to see the browser being driven by the tests. In order to execute e2e tests as part of the pipeline, Jenkins would also need to open a browser so that the tests can drive that browser, however Jenkins is a user without access to a gui. One alternative to this is to use a headless browser, such as phantomjs. We installed phantomjs globally earlier in the lab for karma.

- Test that you can use phantomjs to run the e2e scripts locally by executing:



```
grunt test:e2e:phantom
```

```

9 specs, 0 failures
Finished in 28.372 seconds
Shutting down selenium standalone server.
[launcher] 0 instance(s) of WebDriver still running
[launcher] phantomjs #1 passed

Done, without errors.

Execution Time (2016-08-23 10:06:08 UTC)
concurrent:test      749ms  2%
express:dev          389ms  1%
protractor:phantom  30.6s  94%
Total 32.4s

```

**NOTE:** the first run you may get a spec failure. Run it again and it should pass.

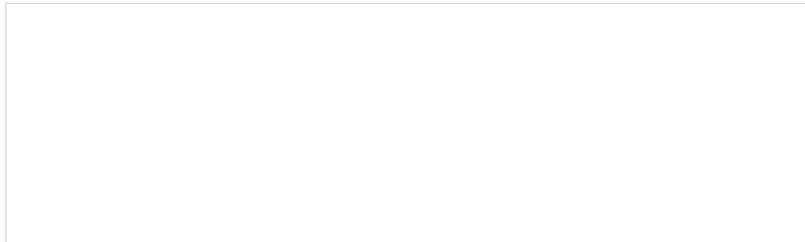
– Now configure the *todolist-deploy-ci* job to execute e2e tests against the ci server after a deploy. Go to the Jenkins homepage: <http://localhost:8080>. Open the *todolist-deploy-ci* job and click *Configure*. Navigate to the *Build* section change the *Execute shell* step to add in the update-webdriver step: `npm run-script update-webdriver` and the grunt task step. Your *Execute shell* section step will now look like this:

```

npm install
bower install
grunt build
grunt build-image:${BUILD_TAG}
grunt deploy:ci:${BUILD_TAG}
npm run-script update-webdriver
grunt test:e2e:ci

```

– This grunt task is configured to report the results in a junit format therefore in Jenkins will be able to report and trend test results. Navigate to *Post-build Actions* and click *Add post-build action* and then select *Publish JUnit test result report*. In the *Test report XMLs* enter `reports/e2e/**/*.xml`. The configuration should look as follows:




– Click *Save*, and run the job, passing in the latest *todolist-build* tag. Once it has run twice you will see that the *Test Result Trend* has been plotted in the job page.

Now that we are confident that our pipeline is running e2e tests for our environments, we can write some e2e tests for user journeys that include our new high-priority functionality.

## (F) Implement e2e tests

The purpose of this lab is to write an e2e test that satisfies our user story: *As a doer I want to mark todos as high priority so that I can keep track of and complete important todos first.* The tests we will add will add a todo then mark it as high priority.

– Before making any changes, checkout a feature branch:

	<pre> git checkout develop # branch from develop git pull # update develop git checkout -b feature/D0421-high-priority-todos-e2e # create e2e branch git push --set-upstream origin feature/D0421-high-priority-todos-e2e # push branch to remote </pre>
---	--

– Add a new describe block to our e2e spec file.

	<p>Open the <code>e2e/todos/todos.spec.js</code> in Atom.</p> <p>Just below the <code>describe('On removing todo' block, add a new describe block and it block:</code></p> <pre> describe('on toggling todo priority ', function () { </pre>
---	--

```

        it('should change the priority to high and mark it with a high priority icon', function () {
        });
    });
});

```

```

describe('on toggling todo priority ', function () {
    it('should change the priority to high and mark it with a high priority icon', function () {
    });
});

```

- For this test, we want to have an existing todo element to toggle it's priority.



In the `e2e/todos/todos.spec.js`

Copy the `beforeEach` block from the `on removing todo` describe block, into the describe block you just created.

Modify the todos content if you like, it should now look something like this

```

describe('on toggling todo priority ', function () {
    beforeEach(function () {
        page.newToDoInputEl.sendKeys('Write an e2e that tests tofflePriority');
        page.newToDoInputEl.sendKeys(protractor.Key.ENTER);
    });
    it('should change the priority to high and mark it with a high priority icon', function () {
    });
});

```

- Now we need to get a handle on the newly created todo element. Protractor provides us some useful DOM manipulation features, similar to that of jQuery.



In the `e2e/todos/todos.spec.js` add this to our `it` block

```
var todoElToTogglePriority = page.todoEls.get(0);
```

- Now we need to drive the browser to perform the click on the high-priority icon. To make the high-priority icon appear, drive the browser by moving the mouse over the element by adding



In the `e2e/todos/todos.spec.js` add this to our `it` block

```
browser.actions().mouseMove(todoElToTogglePriority).perform();
```

- Now that the `.high-priority` element is visible, use protractor to get the element by adding using the element finder:



In the `e2e/todos/todos.spec.js` add this to our `it` block

```
var highPriorityToggleEl = todoElToTogglePriority.element(by.css('.high-priority'));
```

- Now click on the element by adding:



In the `e2e/todos/todos.spec.js` add this to our `it` code block

```
highPriorityToggleEl.click()
```

the test should now look like this:

```

describe('on toggling todo priority ', function () {
    beforeEach(function () {
        page.newToDoInputEl.sendKeys('Write an e2e that tests tofflePriority');
        page.newToDoInputEl.sendKeys(protractor.Key.ENTER);
    });
    it('should change the priority to high and mark it with a high priority icon', function () {
        var todoElToTogglePriority = page.todoEls.get(0);
        browser.actions().mouseMove(todoElToTogglePriority).perform();
        var highPriorityToggleEl = todoElToTogglePriority.element(by.css('.high-priority'));
        highPriorityToggleEl.click();
    });
});



```

To complete our test, we need to assert that the todo is now marked as high-priority. Driving the browser in this way is asynchronous. Due to JavaScript running on a single thread, it cannot wait for the reply, and instead, JavaScript provides a function to be called once this asynchronous operation has completed, called the callback function. The `click()` function returns an object, called a `promise`, that has a function called `then`. Then takes a callback function to execute once the `click()` function has finished executing. Our callback function is



where we will assert that the high-priority icon is now displayed.

We are about to make the test case asynchronous. Jasmine supports asynchronous testing by providing the `done` function to the spec as an argument. If you declare `done` as an argument in your `it` block, Jasmine will wait for the `done` function to be called in the spec before declaring that the spec has run. If you declare `done`, but do not call `done()`; then Jasmine will timeout waiting for `done()` to be called.

- Make your test asynchronous by passing in the `done` argument into your `it` block like so:

	<p>In the <code>e2e/todos/todos.spec.js</code> add this to our <code>it</code> function() argument</p>
	<pre>it('should change the priority to high and mark it with a high priority icon', function(done) {   var todoElToTogglePriority = page.todoEls.get(0);   browser.actions().mouseMove(todoElToTogglePriority).perform();   var highPriorityToggleEl = todoElToTogglePriority.element(by.css('.high-priority'));   highPriorityToggleEl.click(); });</pre>

- Provide a callback to `.click` that makes use of the `hasClass` helper method, located at the top of the file. Ensure that the element now has the class: `checked`, which will add the `display: block` style to the high-priority icon, causing it to be displayed on the todo. Be sure to call the `done` function, provided by Jasmine, so that Jasmine knows when your spec is finished. Your `describe` block will now look as follows:

	<p>In the <code>e2e/todos/todos.spec.js</code> add this to our <code>it</code> function() block</p> <pre>highPriorityToggleEl.click().then(function() {   expect(hasClass(highPriorityToggleEl, 'checked')).toBe(true);   done(); });</pre>
	<pre>describe('on toggling todo priority ', function () {   beforeEach(function () {     page.newToDoInputEl.sendKeys('Write an e2e that tests tofflePriority');     page.newToDoInputEl.sendKeys(protractor.Key.ENTER);   });   it('should change the priority to high and mark it with a high priority icon', function (done) {     var todoElToTogglePriority = page.todoEls.get(0);     browser.actions().mouseMove(todoElToTogglePriority).perform();     var highPriorityToggleEl = todoElToTogglePriority.element(by.css('.high-priority'));     highPriorityToggleEl.click().then(function(){       expect(hasClass(highPriorityToggleEl, 'checked')).toBe(true);       done();     });   }); });</pre>

- Rerun the e2e tests locally, if implemented correctly, these should be successful:


	<pre>grunt test:e2e</pre>
---	---------------------------

- At the time of writing these instructions, my e2e tests reported success first time. Remembering that it is best practice to ensure a new test fails before making it pass, copy and paste the `expect` statement to the line before we execute the `.click()`. Statement on the element. Rerun the test:


	<pre>grunt test:e2e</pre>
---	---------------------------

This should cause the tests to fail because, if we have implemented our UI correctly, the `checked` class should not be on the element before we click on it.


- Correct the expect statement to get your e2e tests to pass and rerun the e2e tests locally:

	<pre>grunt test:e2e</pre>
---	---------------------------

- Commit your code to the repository and push it to the remote with the usual workflow (you do not need to enter the comments, which includes the `#` and anything after it):

	<pre>git status #look through your changeset to ensure it contains everything you expect git add . #stage your changes by adding it to the index git commit -m 'testing priority toggling with an e2e test' #commit the staged changes git pull #get the latest changes from the remote git push #push your changes to the remote</pre>
---	---

- Merge your changes into develop:

	<pre>git checkout develop #switch to the develop branch git pull # sync your local with the remote git merge feature/D0421-high-priority-todos-e2e # merge the feature branch into develop git pull # sync your local with the remote git push # push the merged branch to the remote</pre>
---	---

## Extension tasks

### A) Write another e2e test

Write another end to end test. Your end to end test should follow the following:

**Given** I have a todo with high priority

**When** I click on the high priority icon

**Then** The high-priority icon should not be displayed

Once you have written the test, research into the *Given*, *When*, *Then* syntax and its role in BDD.

### B) Create smoke tests for production

In the root of the todomvc app directory, you will see a protractor configuration: *protractor-smoke.conf.js*, to run smoke tests. You can execute this task by running the following:

	<pre>grunt test:smoke:prod</pre>
---	----------------------------------

Write a simple test that will load the todomvc application in the browser, and check that it has been deployed correctly.

You will need to put the smoke tests in a place that protractor expects to find them

(**HINT** take a look at the *protractor-smoke.conf.js* config file)

Now add this step to the pipeline in Jenkins after the deployment to production.

### C) Create feature builder

- Implement a feature builder job on jenkins. This will watch any of the branches and run a build on every commit. It will run a basic build to give quick feedback on the most turbulent of code.

## Comments

There are no comments.