


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

V2 Lab 3 - Steps 2A & 2B & 2C & 2D

[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

Developing a new feature

The purpose of this lab is to develop and validate a new feature; and to promote the assured feature to production.

The user story for our new feature is as follows

StoryID: DO421	<i>As a doer I want to mark todos as high priority so that I can keep track of and complete important todos first</i>
Acceptance Criteria:	<p><i>On changing a todos priority:</i></p> <ul style="list-style-type: none"> • should be doable with a single click • should add an exclamation mark icon against the todo when marked high-priority • should remove the exclamation mark icon when high-priority removed • should not affect existing todos <p><i>On page load:</i></p> <ul style="list-style-type: none"> • should display existing todos that are not marked high-priority • should display existing todos that are marked high-priority with an exclamation mark

- When collaborating with other developers, it is not usually safe to all be committing to our live development branch. To solve this, we will make use of a git workflow called git flow, that allows you as a developer to separate yourself from external code turbulence whilst you develop a feature. Git flow is probably the most common git branching and release management strategy.

- Firstly, ensure you are on the development branch by executing the following from the ~/todolist directory:




```
cd ~/todolist
git checkout develop
```

- Ensure that you are up to date with the remote. You'll probably see some output saying New Tag on the terminal. This is Jenkins tagging our code levels. These can also be viewed in gitkracken :




```
git pull origin develop
```

- Create a feature branch for developing the high priority todos, the normal naming convention is `feature/` followed by the story id and an abridged name for the feature:



```
git checkout -b feature/DO421-high-priority-todos-client
```

- Push the feature branch to the remote:




```
git push -u origin feature/DO421-high-priority-todos-client
```

Now we are ready to make some code changes.

- Open the project in Atom by running the following command from a terminal:

	<code>atom ~/todolist</code>
---	------------------------------

- Open the angular html template for the todos: `client/app/todos/todos.html`. Add the high-priority button to the DOM write before the destroy button:

	<pre>in client/app/todos/todos.html <button class="high-priority icon"></button></pre>
---	--

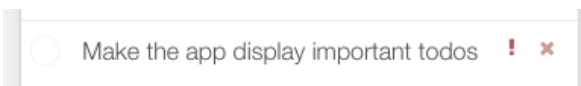
Your todos.html should look like the following:

```
<div class="view" class="todo">
  <button ng-class="{ 'checked': todo.completed}" class="toggle" ng-click="toggleCompleted(
  todo)"></button>
  <label ng-dblclick="editTodo(todo)">{{todo.title}}</label>
  <button class="high-priority icon"></button>
  <button class="destroy icon" ng-click="removeTodo(todo)"></button>
</div>
```

- Now check that the ui code fragment looks the way you expect by serving the application. From the `~/todolist` directory run the following:


	<code>grunt serve</code>
---	--------------------------

The resulting ui should look like this when hovering over the list element, next to the original destroy button:



Try clicking the high-priority button. This has not been wired into our client-side code yet, so we wouldn't expect anything to happen. The high-priority button now needs to mark that todo as high-priority and persist that information. This is going to require changes to both the client and server code.

- 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 'adding styling and ui changes for high-priority' #commit the staged changes git pull #get the latest changes from the remote git push #push your changes to the remote</pre>
---	--

NOTE: Jenkins will not build on this commit because the `todolist-build` is polling for changes on `*/develop`

(A) Enhance the pipeline with UI unit tests

When developing our code, we want to make sure we get feedback from the pipeline if our feature is working and to ensure that we have not broken anything existing in the code base. Currently our pipeline just builds and deploys the application. In this lab we will enhance our pipeline so that it unit tests the client code and performs API tests on the server side code.

Karma is a test runner for client-side, javascript code. Karma effectively creates a runtime in a browser in the form of an iframe where your tests are executed. It also provides some cool debugging and reporting features.

We are using PhantomJS as our browser. PhantomJS is a headless browser, therefore you will not see a browser open when the tests run and instead they run in the background. Usually Jenkins runs on a server without a ui, therefore a headless browser is desirable.

- First, ensure that phantomjs is installed as a global node module by looking in the `node-js.pp` manifest puppet script. Open the lab-material project in Atom:

	<code>atom /share/lab-material</code>
---	---------------------------------------

- Open the puppet script at `puppet/node-js.pp`. You should see that the `phantomjs` package is installed and that the `phantom` environment variable has been set for every user (and jenkins). It should look as follows (note that this step requires no action, and is there to show you the relevance of using Puppet to do this):



```
package { 'phantomjs':
  ensure => '2.1.1',
  provider => 'npm',
  require => Package['nodejs']
} ->
file { ['/etc/profile.d/phantomjs.sh', '/var/lib/jenkins/.profile']:
  content => inline_template('export
PHANTOMJS_BIN="/usr/bin/phantomjs"'),
  ensure => present
}
```

- If you made any changes to the manifest, rerun the puppet script and restart jenkins:



```
sudo puppet apply build-vm.pp
jenkins_restart
```

- Verify that phantomjs is installed by swapping to jenkins and running the following:




```
sudo su - jenkins
phantomjs --version
```

- Verify that `PHANTOMJS_BIN` env was installed:



```
echo ${PHANTOMJS_BIN}
exit
```

- Install the `karma-phantomjs-launcher` by executing the following from `~/todolist` and commit the changes made to the `package.json`



```
cd ~/todolist
npm install --save-dev karma-phantomjs-launcher@1.0.1
git add .
git commit -m 'installing phantom karma launcher for ui tests'
git push
```

- Lets see how the Karma tests run locally. We use the `grunt karma runner` to configure and run karma for us. from the `~/todolist` directory, run:



```
grunt test:client
```

The runner will finish and report success in the terminal as follows:

```
Running "karma:unit" (karma) task
INFO [karma]: Karma v0.12.37 server started at http://localhost:8082/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on socket _Ai_6iaZris0xP-IB-Ya with id 28373887
PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 12 of 12 SUCCESS (0.084 secs / 0.082 secs)

Done, without errors.

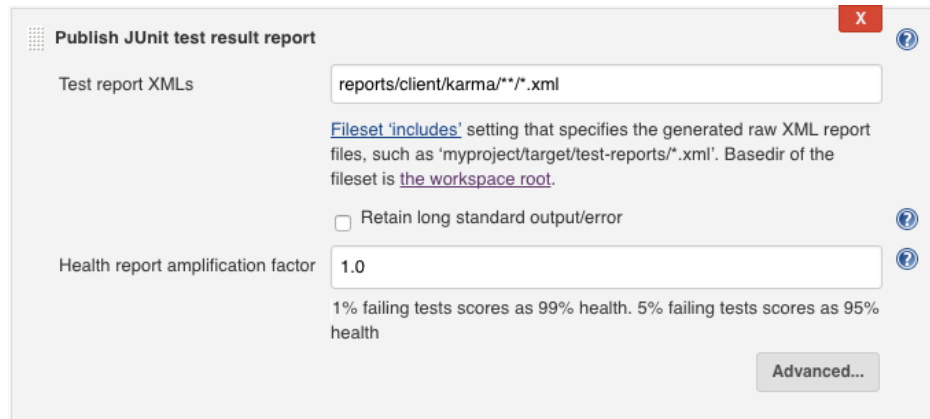
Execution Time (2016-08-17 15:04:05 UTC)
loading tasks      66ms    2%
concurrent:test    593ms   22%
injector:scripts   27ms    1%
injector:css       138ms   5%
autoprefixer:dist  476ms  18%
karma:unit         1.3s   49%
Total 2.7s
```

- To configure karma testing to execute as part of the pipeline, go to the Jenkins homepage: <http://localhost:8080>. Open the `todolist-build` job

and click *Configure*. Navigate to the *Build* section change the *Execute shell* step to add in the karma test. Your *Execute shell* section step will now look like this:

```
npm install
bower install
grunt test:client
grunt build
```

- Providing that the test runner outputs reports in a JUnit format, Jenkins will be able to report and trend test results. To setup test reporting in Jenkins, navigate to the *Post-build Actions* section and click *Add post-build action* then select *Publish JUnit test result report*. In the *Test report XMLs* enter `reports/client/karma/**/*.xml`. The configuration should look as follows:



Publish JUnit test result report

Test report XMLs:

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is the workspace root.

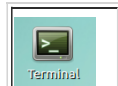
☐ Retain long standard output/error

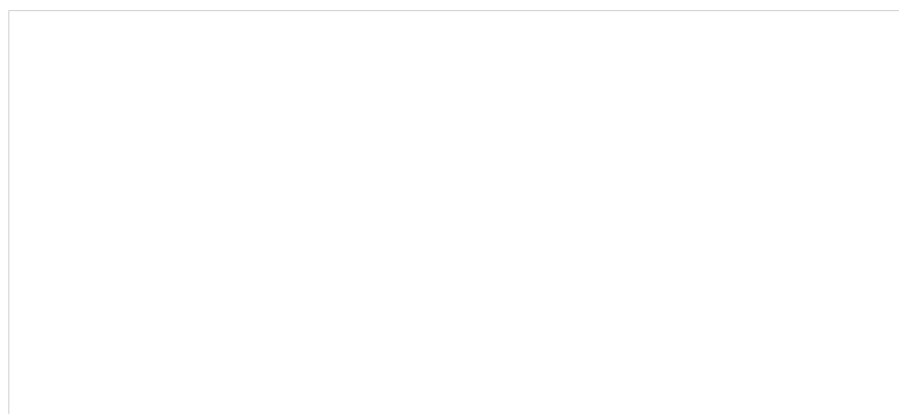
Health report amplification factor:

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

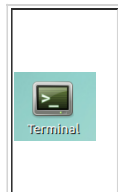
Advanced...

- Click Save the configuration and now we have to merge our feature code onto the develop. High functioning teams will often peer review code before merging onto the develop branch. This practice leads to higher quality of code and increases the cross team knowledge of each feature being implemented. Jenkins is set to track changes off the `develop` branch so merging our feature onto this is necessary. Have your neighbour review your code before running your commands to merge, run to produce a diff of the changes between the branches.

	<pre>git diff feature/DO421-high-priority-todos-client..develop</pre>
--	---

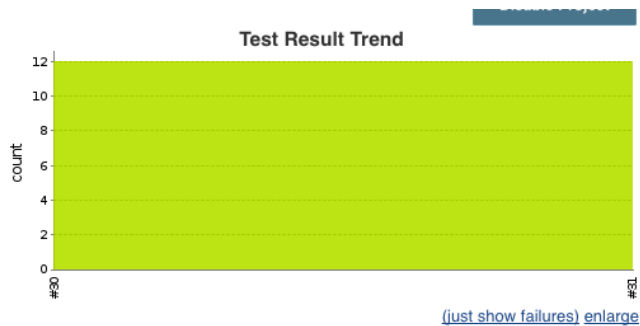


- Once your partner is happy with the changes you're made, hit 'q' to quit the diff and commit them using the following

	<pre>git checkout develop #switch to the develop branch git pull # sync your local with the remote git merge feature/DO421-high-priority-todos-client # merge the feature branch into develop git pull # sync your local with the remote git push # push the merged branch to the remote</pre>
--	--

TODO - opportunity to peer review code in Gitkraken instead of just on cmd line

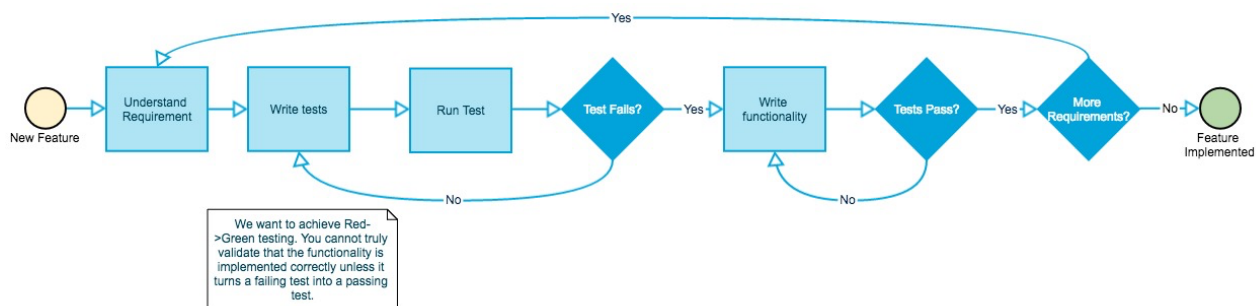
- This should trigger a build, ensure it is successful and then run it again (manually). After two successful runs, you will be provided with a test results trend in the job page:



Now that our pipeline assures our client-side code changes by running Karma tests on every build, let's implement some functionality.

(B) Test driven development - client-side using Karma and Jasmine

Kent Beck, the creator of Extreme Programming, rediscovered and advocated the use of Test Driven Development (TDD) as part of Extreme Programming. Test Driven Development has gone on to become a common principle embraced by developers in both small and large projects and is not only applicable for agile development. TDD means writing test first before developing functionality. The workflow is as follows:



Developing software in this manner ensures you end up with the leanest code that meets the actual requirements. It also ensures that your feature is well tested and therefore has high code coverage.

In BDD you are still writing tests, and you still write them before you start to write your code. There are a lot of opinions as to the differences between TDD and BDD. Ultimately, BDD tests are written in such a way that they are meaningful to a business user, whereas TDD is usually meaningful for the developer to understand and write their code. Many tools are labelled BDD tools because they provide syntactical sugar to make the tests read well to a business user. For the purpose of this course, you can think of them as one and the same thing.

As discussed before, Karma is just the test runner that provides the environment for the tests to execute in. The test framework we will use to write our tests in is Jasmine. You can find out more about Jasmine here: <http://jasmine.github.io/>. Jasmine is a BDD test framework. In Jasmine, test files are suffixed with .spec.js.

- Make sure we are on the feature branch in our git repository. From a terminal:

```

Terminal
cd ~/todolist
git status

```

If we are still on the develop branch, run:

```

Terminal
git checkout feature/D0421-high-priority-todos-client

```

- Open the project in Atom by running the following command from a terminal:

```

Terminal
atom ~/todolist

```

- Open the todos controller spec file: client/app/todos/todos.controller.spec.js


```

Atom
Open client/app/todos/todo.controller.spec.js

```

In AngularJS, the controller contains the logic for controlling the view. In this example it will contain the logic that decides whether or not to display the exclamation mark glyphicon for a todo.

- Let's make use of another test that is similar to the functionality we need for the `togglePriority` function. When our `togglePriority` function is called, it should change the priority of the todo to high and call the todos PUT api with the new priority.

	<p>Copy and paste the <code>toggleCompleted</code> describe block in <code>todo.controller.spec.js</code> (HINT: the describe block spans 16 lines)</p> <p>Change the describe title from <code>#toggleCompleted</code> to <code>#togglePriority</code></p> <p>Remove the second <code>it</code> block (HINT: the <code>it</code> block spans 7 lines)</p> <p>Change the title of the <code>it</code> block to 'should change the priority of the todo to high and call the todos PUT api with the new priority' so it looks as follows:</p>
---	--

```
describe('#togglePriority', function () {
  it('should change the priority of the todo to high and call the todos PUT api with the new priority', function() {
    mockTodos[1].completed = true // describes the environment
    scope.toggleCompleted(mockTodos[1]) // does the actual execution
    $httpBackend.expectPUT('/api/todos/' + 1, mockTodos[1]).respond({}) // mocks the response from the API and uses
    $httpBackend.flush() // required to make the mock PUT response above actually return
    expect(scope.todos[1].completed).toBe(false) // asserts the result (angular scope is the data/model provided to
  });
});
```

The `it('should` statement, describes the result and any behavior exhibited when calling the unit that we are testing.

- We will look at the first test first. The first part of the test should setup the test. The todo starts with low priority, so set the `highPriority` property to false.

- The second part of the test should actually execute the unit. On the second line, call the function on the scope, `scope.togglePriority(mockTodos[1])`.

- The final part of the test is to write what we expect to happen.

- Angular has given us a handy mock called `$httpBackend`. This allows us to mock endpoints and in turn assert the calls made against the mocked endpoint.
- We expect that the todos PUT api is called with the updated mockTodo. The third line should still read `$httpBackend.expectPUT('/api/todos/' + 1, mockTodos[1]).respond({})`. When using `$httpBackend`, you need to flush the backend to have the backend actually respond when a mocked endpoint is called. The fourth line should still read `httpBackend.flush()`.
- Finally, assert the result of the function, which is that the `highPriority` property will have now be `true`. We will use Jasmine's `expect` for this, `expect(scope.todos[1].highPriority).toBe(true)`. Your test case should look like the following:


```
describe('#togglePriority', function () {
  it('should change the priority of the todo to high and call the todos PUT api with the new priority', function() {
    mockTodos[1].highPriority = false // describes the environment
    scope.togglePriority(mockTodos[1]) // does the actual execution
    $httpBackend.expectPUT('/api/todos/' + 1, mockTodos[1]).respond({}) // mocks the response from the API and uses
    $httpBackend.flush() // required to make the mock PUT response above actually return
    expect(scope.todos[1].highPriority).toBe(true) // asserts the result (angular scope is the data/model provided to
  });
});
```

- Now execute the client tests:

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

We should have one test failure, complaining that `undefined` is not a constructor. This is because there is no function on scope called `togglePriority`.

- Let's implement the client feature code:

	<p>Open <code>client/app/todos/todos.controller.js</code></p> <p>Just after the <code>toggleCompleted</code> function, add a new function on the <code>\$scope</code> object, called <code>togglePriority</code> so it looks as follows:</p>
---	--


```
$scope.toggleCompleted = function (todo) {
  // the completed state has been toggled by a
  if (!todo) {
    return;
  }
  todo.completed = !todo.completed;
  $http.put('/api/todos/' + todo._id, todo);
};

$scope.togglePriority = function (todo) {};
```

- Rerun the tests:

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

- Now the tests should be failing with an error message `Unsatisfied requests: PUT /api/todos/1`. This is expected, because we have not actually made the PUT request. Let's implement that piece. Add the following code to your function:


	<p>In <code>client/app/todos/todos.controller.js</code> add <code>\$http.put('/api/todos/' + todo._id, todo);</code> into the <code>togglePriority</code> function</p>
---	--

```
$scope.togglePriority = function (todo) {
  $http.put('/api/todos/' + todo._id, todo);
};
```

- Rerun the tests:

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


- Now the tests should be failing with an error message `Expected false to be true`. Again, this is expected, because we have not changed the `highPriority` property to `true`. Add the following code:

	<p>In <code>client/app/todos/todos.controller.js</code> add <code>todo.highPriority = true;</code> into the <code>togglePriority</code> function</p>
---	--

```
$scope.togglePriority = function (todo) {
  todo.highPriority = true;
  $http.put('/api/todos/' + todo._id, todo);
};
```

Now the tests should be passing. We have not yet satisfying the acceptance criterion: *should remove the exclamation mark icon when high-priority removed*. Let's implement another test that: given the priority of a todo is already high, when we `togglePriority` then it should change the priority of the todo to low and call the todos PUT api with the new priority.

- Copy and paste the spec you wrote previously, and modify the title (it). The difference between this spec and the last one is that the `highPriority` property starts as `true` and that it should be `false` after `togglePriority` is executed. It should look as follows:


	<p>In <code>client/app/todos/todo.controller.spec.js</code> cp your previous test update the <code>it('')</code> with an appropriate test name setup the test with a todo's <code>highPriority</code> property set to <code>true</code> expect that, after executing the function, the todo's <code>highPriority</code> property set to <code>false</code></p>
---	--


```
describe('#togglePriority', function () {
  it('should change the priority of the todo to high and call the todos PUT api with the new priority', function() {
    mockTodos[1].highPriority = false // describes the environment
    scope.togglePriority(mockTodos[1]) // does the actual execution
    $httpBackend.expectPUT('/api/todos/' + 1, mockTodos[1]).respond({}) // mocks the response from the API and uses
    $httpBackend.flush() // required to make the mock PUT response above actually return
    expect(scope.todos[1].highPriority).toBe(true) // asserts the result (angular scope is the data/model provided to
  });
  it('should change the priority of the todo to low and call the todos PUT api with the new priority', function(){
    mockTodos[1].highPriority = true // describes the environment
    scope.togglePriority(mockTodos[1]) // does the actual execution
    $httpBackend.expectPUT('/api/todos/' + 1, mockTodos[1]).respond({}) // mocks the response from the API and uses
    $httpBackend.flush() // required to make the mock PUT response above actually return
    expect(scope.todos[1].highPriority).toBe(false) // asserts the result (angular scope is the data/model provided to
  });
});
```

- Rerun the tests:

	grunt test:client
---	-------------------

- This time the failure is that, for the new test, the `highPriority` property for the todo Expected true to be false after `togglePriority` was executed. This is fixed as follows:


	In <code>client/app/todos/todos.controller.js</code> add <code>todo.highPriority = !todo.highPriority;</code> into the <code>togglePriority</code> function
---	--

```
$scope.togglePriority = function (todo) {
  todo.highPriority = !todo.highPriority;
  $http.put('/api/todos/' + todo._id, todo)
};
```

- Rerun the tests:


	grunt test:client
---	-------------------

- The tests should now all pass. Commit and push your code to the remote:

	git status git add . git commit -m 'implement ui functionality for toggling priority' git pull git push
---	---


As you can see, we have written the bare minimum amount of code to make the tests pass. This may have seemed an obvious unit of code to write, however for more complex pieces of code, you can see how it is a great tool for ensuring developers write minimal clean functional code that meet the requirements.

- Now we have a function on the scope that toggles priority, we can call that when the high-priority button is clicked. Let's revisit the UI, open the angular html template for the todos: `client/app/todos/todos.html`.

	In <code>client/app/todos/todos.html</code> Add <code>ng-click="togglePriority(todo)"</code> to the high-priority button:
---	--

```
<button class="high-priority icon" ng-click="togglePriority(todo)"></button>
```

- The `togglePriority` function now sets the `highPriority` attribute against the todo in the model, which we can use in the view to display which todos are highPriority. `ng-class` can be used to conditionally apply styles. Add the following to the high-priority button:

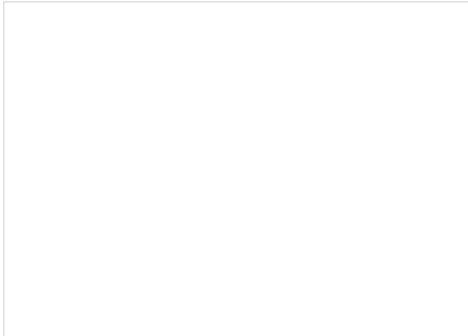
	In <code>client/app/todos/todos.html</code> Add <code>ng-class="{ 'checked': todo.highPriority}"</code> to the high-priority button:
---	---

```
<button class="high-priority icon" ng-class="{ 'checked': todo.highPriority}" ng-click="togglePriority(todo)"></button>
```



- Now serve the application to see your changes:

	<pre>grunt serve</pre>
---	------------------------


- In the todolist app, click on the priority exclamation mark, and you should now see that the high-priority flag stays against the todo:



- Refresh the page and see that all of your todos and their importance have been reset
- Commit the code and push it to the remote:

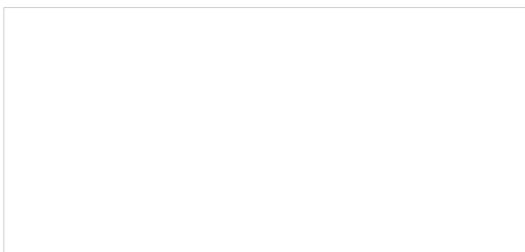
	<pre>git status git add . git commit -m 'implement high-priority ui' git pull git push</pre>
---	--

- Merge the code onto develop (the comments denoted with # don't need to be included):

	<pre>git checkout develop #switch to the develop branch git pull # sync your local with the remote git merge feature/DO421-high-priority-todos-client # merge the feature branch into develop git pull # sync your local with the remote git push # push the merged branch to the remote</pre>
---	--

NOTE: When developing a feature, it would usually be more appropriate to make changes to the backend and frontend before merging into develop, however we have merged at this stage to kick off a pipeline build.

- Go to the jenkins homepage and click on the *todolist-build* job. Once it has finished building, you will see the test trend increase because of the newly added tests:



- Once *todolist-deploy-si* completes, navigate to the deployed application on <http://localhost:9002/> and verify your changes work. Add a new todo, and mark it as high-priority then refresh the page. you should see that, although the todo has been persisted, the priority has not.
- There are some backend changes we need to make to ensure our data is persisted by the backend.

(C) Enhance the pipeline with API tests

As with the Karma tests for our client-side code, API tests ensure our server side code is working. Currently our pipeline builds, tests client-side code and deploys the application. In this lab we will enhance our pipeline so that it tests our APIs.

The API tests in this project are not concerned with the behavior of the APIs, only the result. This is known as black-box testing.

- Open the *server/api/todo/todo.spec.js* in Atom and take a look at the layout of the spec

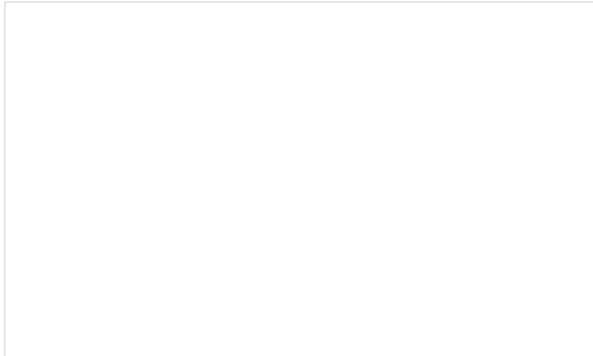
Mocha is a framework similar to Jasmine. Mocha is the test runner for API tests, and it executes within the NodeJS environment. `should` is a node module used for making assertion language more readable, and provides something similar to Jasmine's `expect` library.

- To see the output of mocha in the terminal, we can run it locally with a handy grunt task. From the ~/todolist directory, run:



NOTE: If you see the following error: *MongoDB connection error: MongoError: connect ECONNREFUSED* Then mongo is not running. Execute `docker ps -a` to get the container id and then execute `docker start <CONTAINER ID>`

The runner will finish and report success in the terminal as follows:

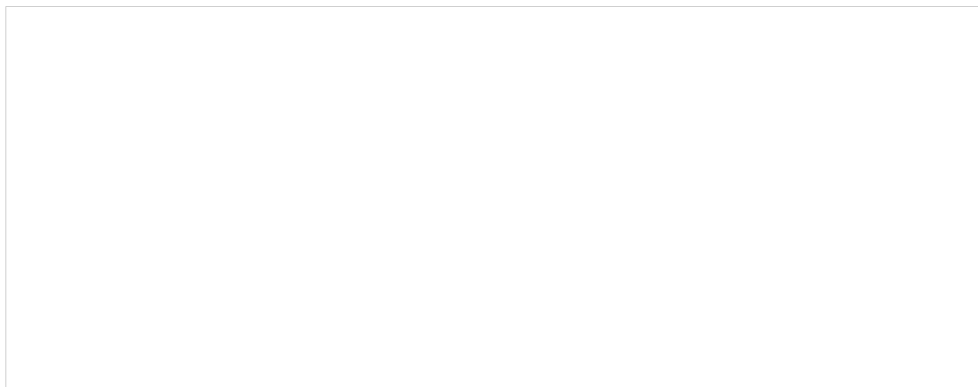


- To configure mocha tests execute as part of the pipeline, go to the Jenkins homepage: <http://localhost:8080>. Open the *todolist-build* job and click *Configure*. Navigate to the *Build* section change the *Execute shell* step to add in the grunt task that executes the mocha test: `grunt test:server-jenkins`. Your *Execute shell* section step will now look like this:

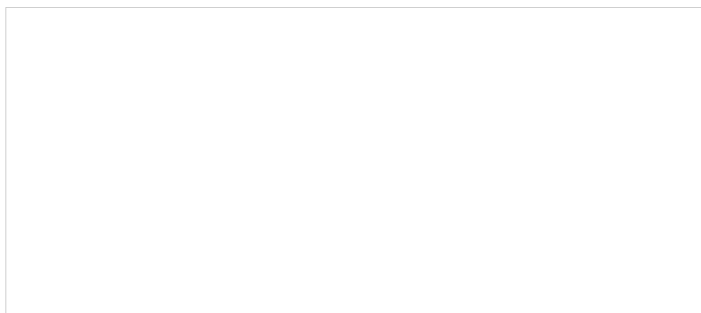
```
npm install
bower install
grunt test:client
grunt test:server-jenkins
grunt build
```

NOTE: server-jenkins means that mocha will use a junit reporter, which we need in order to publish the results on Jenkins.

- Now that mocha reports in JUnit format, Jenkins will be able to report and trend test results. To combine the rest results for the server and the client, navigate to the existing Junit report configuration in *Post-build Actions > Publish JUnit test result report*. In the *Test report XMLs* enter `reports/client/karma/**/*.xml,reports/server/mocha/**/*.xml` The configuration should look as follows:



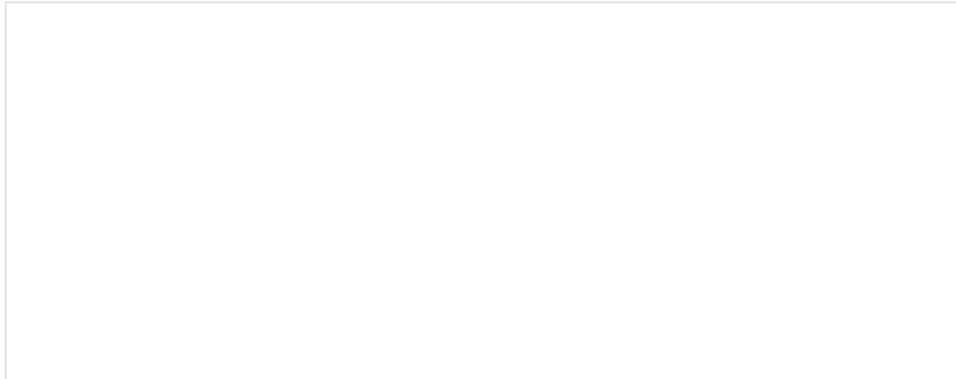
- Click *Save*, and run the build. Once it has completed, You will see that the *Test Result Trend* has increased to include all the existing server tests:



- API tests can also be used to prove your integration environment is working. In Jenkins, navigate to the *todolist-deploy-si* Jenkins job page. Click on *Configure* and navigate to the *Build* section. Change the *Execute shell* step to add in API tests. Your *Execute shell* section step will now look like this:

```
npm install
grunt deploy:si:${BUILD_TAG}
grunt test:server-jenkins:si
```

- Publish the test results by navigating to *Post-build Actions*, clicking on *Add post-build action* and selecting *Publish JUnit test result report*. In the *Test report XMLs* enter `reports/server/mocha/**/*.*.xml`. The configuration should look as follows:



The screenshot shows the Jenkins configuration page for the 'Publish JUnit test result report' step. The 'Test report XMLs' field is populated with the path `reports/server/mocha/**/*.*.xml`. The 'Publish JUnit test result report' step is selected, and the 'Test report XMLs' field is highlighted.

- Click save and then run the phase again, passing in the most recent build tag.

Run this twice with the same build tag, and navigate to the job page to view the *Test Results Trend*.


(D) Test driven development - server-side using Mocha

As discussed before, Mocha is our test framework. You can find out more about Mocha here: <https://mochajs.org/>.

- Navigate to the todo endpoint directory in Atom: *server/api/todo*. This folder contains all of the logic needed for the endpoint. The *index.js* file routes the TODO requests to the controller methods. The *todo.controller.js* contains the logic for the endpoint, usually interacting with the mongodb, and the *todo.model.js* contains the schema for the todo object stored in the mongodb.

	<p>In <code>server/api/todo</code> Examine the folder and file structure</p>
---	--

- Open the todo endpoint spec. Remember that the `togglePriority` method we wrote in the client side code sends a *PUT* request to the `/api/todos/:id` API? This API needs to accept a new attribute in the request body, `highPriority`.

	<p>Open <code>server/api/todo/todo.spec.js</code></p>
---	---


- Before making any code changes, ensure you are on the development branch by executing the following from the `~/todolist` directory:

	<pre>cd ~/todolist git checkout develop</pre>
---	---

- Ensure that you are up to date with the remote:

	<pre>git pull origin develop</pre>
---	------------------------------------

- Create a feature branch for developing the high priority todos, the normal naming convention is `feature/` followed by the story id and an abridged name for the feature:

	<pre>git checkout -b feature/DO421-high-priority-todos-server</pre>
---	---

- Push the feature branch to the remote:

--	--



```
git push -u origin feature/D0421-high-priority-todos-server
```

- Let's make use of another test that is similar to the functionality we need for the togglePriority function.



Open `server/api/todo/todo.spec.js`

Navigate to the PUT `/api/todos/:id` describe block and copy and paste the `it('should update the todo' block (HINT: the it block spans 14 lines).`

Change the `it` title to `should update the todo with highPriority status`.

Change the request body to include `highPriority: true` so that it looks as follows:

```
it('should update the todo with highPriority status', function(done) {
  request(app)
    .put('/api/todos/' + todoId)
    .send({title: 'LOVE endpoint/server side testing!', completed: true, highPriority: true})
    .expect(200)
    .expect('Content-Type', /json/)
    .end(function(err, res) {
      if (err) return done(err);
      res.body.should.have.property('_id');
      res.body.title.should.equal('LOVE endpoint/server side testing!');
      res.body.completed.should.equal(true);
      done();
    });
});
```

- A PUT request responds with the data that it just updated, so provided that MongoDB accepted the change, it will respond with an object that has the `highPriority` property on it.



In `server/api/todo/todo.spec.js`

Add the assertion that `res.body.should.have.property('highPriority')`

Add another assertion to ensure the value: `res.body.highPriority.should.equal(true)` so that it looks as follows:

```
it('should update the todo with highPriority status', function(done) {
  request(app)
    .put('/api/todos/' + todoId)
    .send({title: 'LOVE endpoint/server side testing!', completed: true, highPriority: true})
    .expect(200)
    .expect('Content-Type', /json/)
    .end(function(err, res) {
      if (err) return done(err);
      res.body.should.have.property('highPriority');
      res.body.highPriority.should.equal(true);
      done();
    });
});
```

- Re-run the server tests:



```
grunt test:server
```

- You will see that there is an assertion error. MongoDB has not responded with the `todo` property: `highPriority` because we need to update the `todo` model's schema:



Open `server/api/todo/todo.model.js`

Add `highPriority` to the model

ensure that it is not mandatory so that we do not break the `addTodo` requests


```
var TodoSchema = new Schema({
  title: String,
  completed: Boolean,
  highPriority: {
    type: Boolean,
    required: false
  }
});
```

- Re-run the server tests:




```
grunt test:server
```

- The tests should now all pass. Commit and push your code to the remote:

 Terminal	<pre>git status git add . git commit -m 'implement server functionality for toggling priority' git pull git push</pre>
---	--

- Merge the code onto develop:

 Terminal	<pre>git checkout develop #switch to the develop branch git pull # sync your local with the remote git merge feature/DO421-high-priority-todos-server # merge the feature branch into develop git pull # sync your local with the remote git push # push the merged branch to the remote</pre>
---	--

- This change has kicked off a pipeline build. Once the pipeline completes the *todolist-deploy-si* job, navigate to the deployed application on <http://localhost:9002/> and verify your changes work. Add a new todo, and mark it as high-priority then refresh the page. you should see that your changes are now persisted.

Comments

There are no comments.