

Ruben Vermeersch (<https://savanne.be>)

About (<https://savanne.be/about/>)

Articles (<https://savanne.be/articles/>)

Blog (<https://savanne.be/blog/>)

---

# Testing with Angular.JS

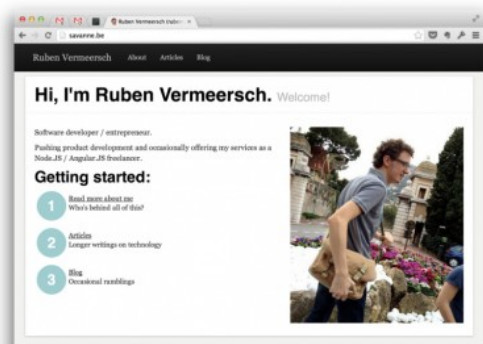
---

On Mar 5, 2014, I gave a talk for the Belgian Angular.JS Meetup group (<http://www.meetup.com/The-Belgian-AngularJS-Meetup-Group/>) about testing. On this page you'll find the annotated version of the slides I used that day.



(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.001.jpg>)

Ruben Vermeersch (@rubenv)  
[ruben@rocketeer.be](mailto:ruben@rocketeer.be)



(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.002.jpg>)

If you don't feel like reading and just want the slides: download them here (</wp-content/uploads/2014/03/Angular.JS-Testing.pdf>). There were a lot of code examples in this talk though, so you probably want to check the code examples below as well.

Let's go!

## The obligatory introduction slide!

The web has always been this thing that I have a hate-love relationship with. I started out with web technology a long time ago and did loads of different things in the years afterwards: desktop development in the GNOME project, a company that builds mobile applications, ... But I keep coming back to the web. In recent years it has become so much fun!

If you'd like to know more about me, just check the about page (</about>). My activities these days are Angular.JS and Node.JS, all day, every day. And that is why I was asked to give this talk.

Oh and while you're at it: I'm on Twitter too

<https://twitter.com/rubenv>).

## Two assumptions

- You should test more



<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.004.jpg>



<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.006.jpg>

Part one: Unit testing

Part two: E2E testing

<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.007.jpg>

## Two assumptions

I've made two assumptions while preparing this talk.

First one: you should test more. I will not try to convince anyone that you should test: the benefits of that should be obvious. I assumed that we agree on that one.

But we don't test enough, because it's just very hard to get started. This presentation should help with that. You'll see that it's not actually all that hard.

The second assumption is that you know what AngularJS is and does. If not, check the [tutorial](http://docs.angularjs.org/tutorial) (<http://docs.angularjs.org/tutorial>) first.

The most important fact about AngularJS is that we try hard to not manipulate the DOM ourselves. Instead, there's this thing called `$scope`, which is just a plain standard object, but AngularJS treats it in a special way. The scope gets injected in controllers, where you can manipulate it. For instance, in this example we set a small array on it.

We also annotate our HTML with special directives, e.g. `ng-repeat`. AngularJS will then automatically synchronize the DOM and the scope. Magic!

This talk is split in two parts: unit testing and E2E testing (we'll get to the difference between those in a minute). There will be drinks in between.

Each part is roughly: an introduction with a simple test scenario, how to get the tools up and running and then a bunch of tips & tricks for when it gets difficult.

You'll need to supply your own drinks if you're reading this at home.

# Unit vs E2E

## Unit vs E2E

Unit test	E2E test
APIs	UIs
Tiny	Large
White-box	Black-box
Isolated from the world	Make the pieces fit together

(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.014.jpg>)

Unit tests and E2E (which stands for End-to-End and is sometimes referred to as Integration) tests are two very different things which serve different needs. But you do need them both.

Unit tests are generally about testing APIs, the direct result of our labor as a developer. It's the automated equivalent of writing some code and checking the output of `console.log`. They're isolated from the world to make it easy to simulate weird error conditions.

E2E tests verify that all the pieces fit together. We're no longer talking about code here: this is automated interaction with the user interfaces, which for us web developers means HTML in a browser.

## Unit testing

`$scope` is a wonderful thing



(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.016.jpg>)

Let me start out by saying that `$scope` is a truly wonderful thing. Not only does the scope shield us from much of the pain that comes along with DOM manipulation, it is also the thing that makes Angular.JS inherently testable. This testability comes from a decoupling of the logic we want to test and the visualization.

This is not a new idea in computer science: it was the idea behind MVC (but that never materialized) and it really came together in MVVM. The Angular.JS authors like to call their model MVW: Model-View-Whatever.

There are other modern frameworks that use this technique. It is a big shift in how we used to build web applications and in my eyes the reason why we can suddenly test frontend code, which was nearly impossible with older frameworks (a huge frustration I had with Backbone.JS).

This super-trivial application will be the main example that I use. It is ridiculously trivial, yet complex enough to be able to show all the

different things that matter. Real-life testing is a lot more complex, but the concepts are the same: once you know how to start, it is rinse and repeat.

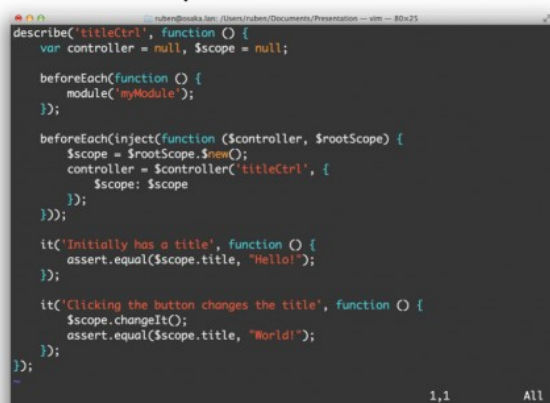
In essence, it's a page with a title (Hello!) and a button. When you click on the button, the title changes. That's what we'll test.

Note: There's not much on the slides, because most of the talk consisted of live demos. Look above the slides for tabs that allow you to switch between the slide and actual code.

Slide

Code

## Simple test case



```
describe('titleCtrl', function () {
  var controller = null, $scope = null;

  beforeEach(function () {
    module('myModule');
  });

  beforeEach(inject(function ($controller, $rootScope) {
    $scope = $rootScope.$new();
    controller = $controller('titleCtrl', {
      $scope: $scope
    });
  }));

  it('Initially has a title', function () {
    assert.equal($scope.title, 'Hello!');
  });

  it('Clicking the button changes the title', function () {
    $scope.changeIt();
    assert.equal($scope.title, 'World!');
  });
});
```

<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.021.jpg>

Here's a test case for this simple application. The first line contains a `describe` block. This is a way to group tests.

There are two `beforeEach` blocks:

1. Load the module that contains our application code.

Not everybody knows this, but an AngularJS application can be instantiated inside of a div, completely separate from the rest of the HTML page. You could even have the same application on a page twice.

The unit testing framework uses this property to instantiate a clean environment for each test. The `module` function is a helper method to do this. Very neat!

2. Inject the dependencies of the test using the `inject` helper method. We then create a new scope, to which we store a reference. Finally we manually invoke the `$controller` service, which constructs controllers. We pass it the scope we just gave.

You can pass any dependency of the controller in this second argument. Any dependencies not specified will be resolved through the normal dependency injection mechanism.

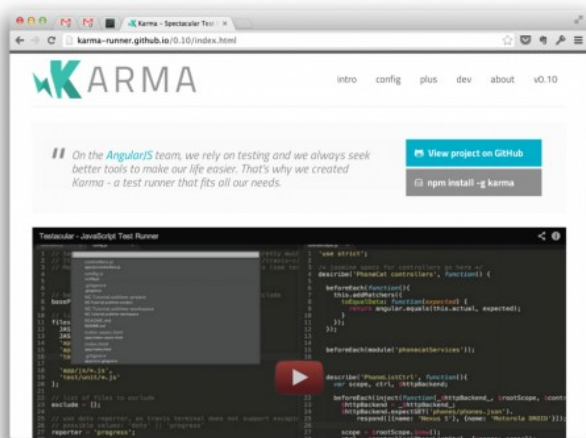
Unit testing in AngularJS is most of the time a matter of manipulating the scope and observing the behavior that follows. That's why we store a reference to it: we need it in the actual test cases.

Each test case is specified with an `it` function,

which has a descriptive title and a function that performs the actual test.

The first test simply verifies a property of the scope in its initial state (is the title "Hello?").

The second test calls the `changeIt` method, as Angular.JS would have done for us when the user clicks the button. We then check if the title has actually changed.



(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.022.jpg>)

[Slide](#)

[Sample configuration](#)

#### Install:

```
npm -g install karma
```

```
npm install --save-dev karma-mocha karma-chai
```

#### Configure & run:

```
karma init
```

```
karma start karma.conf.js
```

(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.023.jpg>)

## Karma

The tool that executes the tests in browsers is Karma (<http://karma-runner.github.io/>), a test runner built by the Angular.JS team. While originally built specifically for Angular.JS, it is now usable with pretty much any framework.

Karma takes care of starting browsers, running the tests and reporting the results. Every browser we care about is supported and it integrates with most libraries and CI environments.

You can get karma through NPM. By default it comes with Jasmine, but I happen to prefer Mocha (<http://visionmedia.github.io/mocha/>), which you can add by installing karma-mocha. The second framework I install is karma-chai, which provides the Chai (<http://chaijs.com/>) assertion library.

Running `karma init` guides you through the creation of a configuration file. A comprehensive overview of supported options is available on the karma website (<http://karma-runner.github.io/0.10/config/configuration-file.html>).

In your configuration file there's a section for loading your source files. Besides any libraries you use, your source code and test cases, be sure to include the angular-mocks (<http://docs.angularjs.org/api/ngMock>) library. This library provides helper functions for stubbing out external dependencies.

Once that's done, just run

```
karma start karma.conf.js
```

 to run your tests.

## Debugging



# Debugging tests



(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.024.jpg>)

Sometimes you'll run into a nasty test failure that needs debugging. For those occasions, there's a "Debug" button in the browser that Karma opened for you. Click on it and you'll get a new tab page that looks blank. Open your developer console to see what really happened: this seemingly blank page executes your tests. You can set breakpoints here and do the usual step-through debugging (<https://developers.google.com/chrome-developer-tools/docs/javascript-debugging>).

Hit reload to run the tests again.

Important thing to remember: close the browser tab once you're done debugging. Karma seems to be in a habit of blocking or losing track of your browser when the debug mode is open. This is annoying. Closing the tab and triggering the tests again fixes things.

Slide

Controller

Test

## Faking HTTP

Unit tests run in complete isolation of the outside world. This makes them fast (we can replace slow external systems with fake fast ones) and powerful (we can simulate failures and other freak accidents).

This also means that there's no HTTP. The angular-mocks library provides an implementation of `$httpBackend` which disables all normal calls through `$http`. Instead, it allows creating mock responses. This gives us the exact control over network calls and allows you to verify that it does exactly what you want.

I modified the controller slightly to include a HTTP call: the changed title is now loaded over a fictive backend API (which doesn't exist, but it doesn't have to: we'll fake it).

In the test case, we inject a new dependency: `_$httpBackend_`. This is just normal `$httpBackend` (from angular-mocks). What you see is a clever trick that was built into `inject`: you can add two underscores, which it will ignore while resolving the requested dependencies. The benefit of this is that you get the dependency

## Faking HTTP

There's no HTTP in unit tests!  
Because we like to control the environment.

`$httpBackend` is your friend.

(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.026.jpg>)

under a different name which makes it easy to assign it in a higher variable scope under its real name (notice how we store `$httpBackend` in the `describe` scope so that we can use it in the test methods).

Also new: an `afterEach` block that calls the `verifyNoOutstandingExpectation` and `verifyNoOutstandingRequest` methods of `$httpBackend`. These respectively check that there are no missing HTTP calls and no unexpected HTTP calls. Any behavior that differs from what is specified in the test case is considered invalid: exactly what we want.

In the last two test cases you see the specification of HTTP calls. We can either simulate success and return the data we want or simulate a failure. Try that when talking to a real backend.

[Slide](#)

[Controller](#)

[Test](#)

## Faking time

There's no time in unit tests!  
Has to be fast!

`$timeout` is your friend.

### Faking time

There's no real time either. You need to manually control time in your unit tests. This allows you to precisely validate timed functionality and lets you test tricky async code in a synchronous way. But best of all: it eliminates the waiting from your tests, which means they're ridiculously fast.

<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.028.jpg>

## Learn when to `$digest`

Especially when testing promises and directives.

### `$digest`

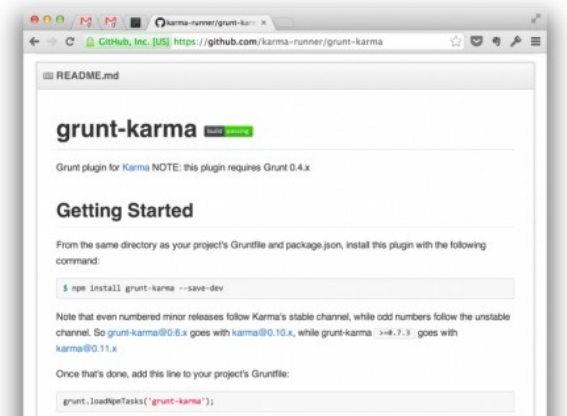
Even the event loop is stopped during unit testing. Callbacks to promises won't happen automatically. You can push things forward by calling `$rootScope.$digest()`. Give it a go when things that should happen don't seem to be going forward.

You'll learn way more than you ever wanted to know about the Angular.JS digest loop during unit testing. This is a good thing, it's also a pain in the ass. Just know when to digest.

<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.028.jpg>

[/2014/03/ngmeetup-testing.029.jpg](http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.029.jpg)

## Use grunt!



[/2014/03/ngmeetup-testing.030.jpg](http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.030.jpg)

[Slide](#)

[Controller](#)

[Test](#)

## Dependencies

AngularJS is like an onion.

[/2014/03/ngmeetup-testing.031.jpg](http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.031.jpg)

## E2E testing

### Unit vs E2E

Unit test	E2E test
APIs	UIs
Tiny	Large
White-box	Black-box
Isolated from the world	Make the pieces fit together

**Browser automation!**

### grunt

I don't work with karma directly in a real project. I strongly recommend using a tool like [grunt](http://gruntjs.com/) (<http://gruntjs.com/>) to integrate testing into your build. There's a [grunt-karma](https://github.com/karma-runner/grunt-karma) (<https://github.com/karma-runner/grunt-karma>) plugin that works great.

### Replacing deep dependencies

I have this trick I use to replace deep dependencies during testing. It uses the fact that the AngularJS module system has one global namespace. Just create a new module with the module to replace and load it after the original module.

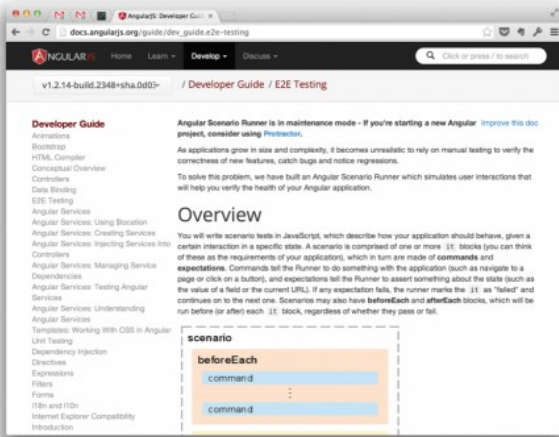
Somebody in the audience pointed out that you could use `$provide` for this, which is probably the way to go.

End-to-end tests are the complete opposite of the low-level unit tests. Instead of small tests that verify individual components, we'll now be testing the whole application. We'll test this in the same way that users use the application: by interacting with the user interface.

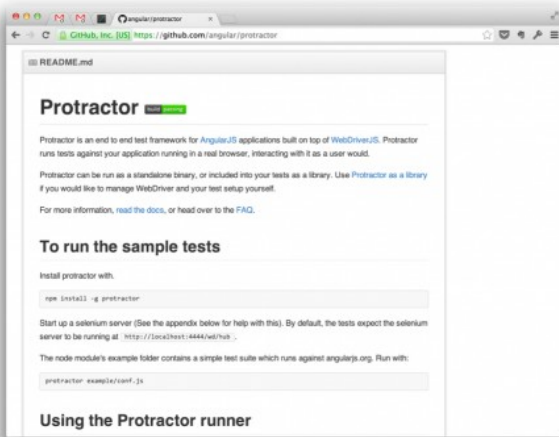
Shortly: E2E is about browser automation.



<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.036.jpg>



<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.037.jpg>



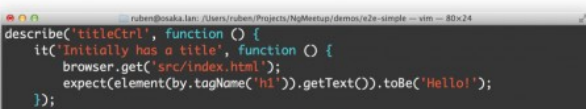
<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.038.jpg>



<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.039.jpg>

Slide

Code



This is the page on E2E testing in the Angular.JS developer guide (<http://docs.angularjs.org/guide>). It's a very long page that describes a framework named angular-scenario. Don't use it. It's no longer the recommended framework, instead you're advised to use Protractor. There's a tiny notice about this on top. It should be enormous and eye-catching.

There are also issues with angular-scenario and Internet Explorer and it is a pain in the ass to test things that aren't pure Angular.JS (e.g. external libraries).

Let's quickly move on.

The new framework for Angular.JS integration testing is Protractor (<https://github.com/angular/protractor>). It's built specifically for Angular.JS and as of Angular.JS 1.2, the recommended way to do browser testing.

Protractor is a wrapper around Webdriver.JS with some specific additions to make your life a lot easier.

We'll use the same example application: a title and a button. Click on the button and the title changes.

Like unit tests, E2E tests are grouped into test suites and test cases, with `describe` and `it` respectively.

The assertion syntax is slightly different though.

```
it('Clicking the button changes the title', function () {
  browser.get('src/index.html');
  element(by.tagName('button')).click();
  expect(element(by.tagName('h1')).getText()).toBe('World!');
});
```

(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.040.jpg>)

Slide

Protractor

Grunt

package.json

#### Install:

```
npm install --save-dev grunt-contrib-connect
grunt-protractor-runner
```

#### Configure & run:

```
Gruntfile.js, protractor.conf.js

grunt
```

(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.041.jpg>)

Your main workhorses will be:

- `expect` : This method is used to construct expectations.
- `element` : Used to construct element matchers.

Both of these work fully asynchronous based on promises. Protractor uses all of these statements to build a command queue for the browser and will execute one after the other, while waiting for Angular.JS to do its thing between commands.

This clever design makes it ridiculously easy to write tests: they look like a sequential list of statements, while in reality they hide a complex asynchronous sequence of interactions.

Truly a beautiful API, once you get the hang of it.

Protractor also comes through NPM. E2E testing is slightly more complex in the sense that you need to have a server somewhere that browsers can access. Therefore I skip the part about using Protractor stand-alone, you'll most likely never do, instead I'll directly show the integration with grunt.

I'm using two grunt plugins in this example:

`grunt-protractor-runner`, which spawns Protractor and `grunt-contrib-connect`, which will provide the temporary HTTP server for our application.

The exact syntax / functionality of the Protractor configuration might change soon: it might become more aligned with the way karma works.

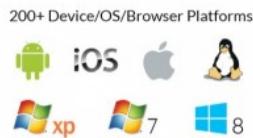
With this in place, running the tests simply becomes running `grunt`.

You'll need to have the Chrome Webdriver (<https://code.google.com/p/chromedriver/>)

before you can run tests. Protractor provides a handy tool for that. For projects (not the stuff that goes into NPM), I use a little trick in `package.json` to make sure this command is always run when you execute `npm install`. This hack will probably come back to bite me, but for now it works nicely and ensures that all developers have the right webdriver at all times.

# Execution

- chromeOnly
- Local Selenium
- Sauce Labs



<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.042.jpg>

## Locators

```
: Protractor.By.id function()
: Protractor.By.css function()
: Protractor.By.xpath function()
: Protractor.By.name function()
: Protractor.By.tagName function()
: Protractor.By.className function()
: Protractor.By.linkText function()
: Protractor.By.partialLinkText function()
: Protractor.By.js function()

P: Protractor.By.addLocator function( string function(string) )
P: Protractor.By.binding function()
P: Protractor.By.select function()
P: Protractor.By.selectedOption function()
P: Protractor.By.input function()
P: Protractor.By.model function()
P: Protractor.By.textarea function()
P: Protractor.By.repeater function()
: Protractor.By.buttonText function()
: Protractor.By.partialButtonText function()
```

<https://github.com/angular/protractor/blob/master/docs/api.md>

<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.043.jpg>

There are three execution modes in Protractor:

1. `chromeOnly`: The preferred development option, this bypasses Selenium and pokes at Google Chrome directly, which makes for very fast test execution.
2. A local Selenium server (which Protractor can start for you if needed). Good for running browsers yourself during continuous integration.
3. Sauce Labs (<https://saucelabs.com/>) an online service that provides you with browsers-as-a-service, for pretty much any platform in existence. Ask for your free account if you are testing an open-source project.

E2E tests simulate the end-user interaction, which ultimately is about manipulating HTML through a browser. To do that we need to specify our tests in terms of that UI: we'll speak about the different DOM elements and what we expect from them.

Protractor offers a very wide selection of locators (<https://github.com/angular/protractor/blob/master/docs/api.md>). Some of them (like `tagName`) are very generic, others are specific to Angular.JS (like `model`, which matches an input field with `ng-model`).

What you should use is up to you and depends on what you want to test. You can make very specific tests (e.g. the second div after h1), but those will break as soon as you change something to the page layout (could be a good thing too). Or you can write extremely generic tests by talking purely about Angular.JS bindings, but those might also miss layout issues.

### Page objects

E2E tests can get unmanageable fast: on a sufficiently complex page, you'll end up with rather complex selectors. Your tests will usually need updating when you change the page structure, which gets boring quite fast. This problem multiplies when you have lots of test cases.

Slide

Page object

Test

## Page objects

Reusing page structure among tests

<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.044.jpg>

There's a programming pattern that can reduce the pain somewhat: page objects. This pattern allows you to (somewhat) separate the structure mapping of your markup from the testing logic.

On the selenium website there's a documentation page for this pattern: [PageObjects](https://code.google.com/p/selenium/wiki/PageObjects) (<https://code.google.com/p/selenium/wiki/PageObjects>). Unfortunately it's written in Java, which makes it hard to apply this to Javascript.

The concept is quite simple though: put all selectors in an object that's reused between tests. Test cases are loaded by Protractor in Node.JS, so you can use standard `require` to reference files.

Notice how `header` and `button` are just plain properties. The `element` matcher is created right away, but Protractor will only match them against your page when needed. This is again an example of the clever things going on inside Protractor: you end up with very simple code and the test framework will resolve the headaches.

The test cases also turn into something that's much more readable. You can actually see what's going on now that the ugly details about page structure are gone.

[Slide](#)

[Page object](#)

[Test](#)

[App](#)

## Counting things

$1 + 1 \neq 3$

<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.045.jpg>

### Counting things

All the previous examples were quite trivial and I don't want to give you the impression that it's always trivial. Some things are really hard. Counting things is one of them. I'll give you an example of how I generally do it. This might not be the best way (it works), so if you know a better way: I'd love to hear it.

I've slightly changed the sample application for this example: it's now a list of items. When you click the button, an extra item is added.

Pop quiz: what happens if you click the button twice? Answer: it fails. Angular.JS does not allow duplicate objects in `ng-repeat`. This is related to the way Angular.JS tries to reuse DOM elements in `ng-repeat` (to make it fast). You can work around this with `track by`. [Here are](#)

the gory details (<http://www.bennadel.com/blog/2556-Using-Track-By-With-ngRepeat-In-AngularJS-1-2.htm>). Anyway, back to testing.

In the page object there's a new `items` function. This will return the result of `findElements`, which is a promise.

In the test case I first capture the row count and store it in a variable. After clicking the button, I'll do it again, this time expecting it to be greater than the last known value. The last bit is an alternative syntax which copes with Firefox failing sometimes.

All of this works, but probably not how you'd expect it to. What happens here is another nugget of Protractor magic. Each of the commands in the test case is executed immediately upon test instantiation. All of this happens before the browser is even started.

Internally Protractor creates a command queue that'll be executed step by step via `webdriver.js`. This queue isn't static: in the example new commands are created in the callbacks that happen after the `MainPage.items()` calls. These commands are added in the right place on the queue, so that it does what you'd expect.

But the beauty of it: all the asynchronous waiting is hidden from you. Unless Firefox tries to make your life worse, that is.

[Slide](#)

[Configuration](#)

## Authentication & Session setup

(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.046.jpg>)

### Authentication & Session setup

Most applications are business applications and pretty much all of those require authentication. You can start each test case with jumping through the login page and navigating to the desired page, but that gets boring soon (also, it's very slow).

There's another way though: inject some data into your page before you actually do things.

Injecting code from the test runner into the browser happens through

`browser.executeAsyncScript`. There's a few caveats: you can't pass arguments directly. The function you pass will be serialized using `Function.toString`, which only serializes the



body. You can use the `arguments` array to get them back though.

From there on, use `angular.element` to grab hold of an element that contains your application. You can then get the dependency injector using the `injector` method. And then you have the keys to the kingdom: request any application service you want through the injector.

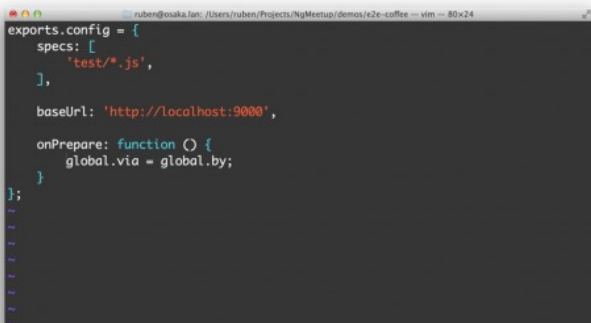
There's also a way to evaluate scope expressions.

Have a look at the Protractor FAQ

(<https://github.com/angular/protractor/blob/master/docs/faq.md#how-can-i-interact-directly-with-the-javascript-running-in-my-app>).

## Coffeescript

ERROR ON LINE 15: UNEXPECTED BY

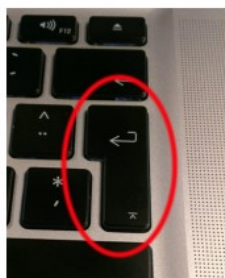


(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.049.jpg>)

`sendKeys(protractor.Key.RETURN)`

vs

`sendKeys(protractor.Key.ENTER)`



(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.050.jpg>)

# Testing with Angular.JS

Ruben Vermeersch  
@rubenv

## Coffeescript

The above causes one big problem if you try to use Coffeescript: `by` is a reserved keyword.

You can get around this by creating an alias for `by` in your `onPrepare` function. The global namespace in the browser is available as `global`.

## Keyboards

One last thing. Inputting text happens through `sendKeys`. Don't try to simulate form submission by hitting the enter key with `protractor.Key.ENTER`. That one works in Google Chrome, but will fail you in Firefox. There's also `protractor.Key.RETURN`. It does exactly the same and it works in both browsers.

Time to wrap up. This was a rather quick (blame the good weather) write-up of almost two hours of interactive presentation.

I'd love to hear what you think, send me an e-mail (/about) or send me a tweet (https://twitter.com/rubenv).



(<http://savanne.be/wp-content/uploads/2014/03/ngmeetup-testing.001.jpg>)

5 Comments

Rocketeer.be

Login ▾

Sort by Best ▾

Share  Favorite ★



Join the discussion...



**Chase Brammer** · 3 months ago

Thanks for this, this really helped me wrap my head around testing in ng. A very well written article. I really enjoyed the additional code and examples.

Thank you.

1 ^ | ▾ · Reply · Share ›



**Marcus Almeida** · 2 months ago

Great article! Nice explanation about angularjs testing. simple and precise.

^ | ▾ · Reply · Share ›



**saulo** · 3 months ago

awesome overview on angularjs testing .great work. tks

^ | ▾ · Reply · Share ›



**Peter Vandenabeele** · 8 months ago

Thanks !

One minor typo:

"we can suddenly testing frontend code"

^ | ▾ · Reply · Share ›



**Ruben Vermeersch** Author → **Peter Vandenabeele** · 8 months ago

Fixed, thanks!

^ | ▾ · Reply · Share ›

 Subscribe

 Add Disqus to your site

 Privacy

## RECENT POSTS

- Release notes: May 2014  
(<https://savanne.be/1052->

## THE AUTHOR

**Ruben Vermeersch**  
Product and company builder with a

## MORE

- Articles (</articles>)
- Flickr (<http://flickr.com/rubenv>)

- release-notes-may-2014/)
- Release Notes: Apr 2014 (<https://savanne.be/1044-release-notes-apr-2014/>)
  - Benchmarking on OSX: HTTP timeouts! (<https://savanne.be/1035-benchmarking-on-osx-http-timeouts/>)
  - Release Notes: Mar 2014 (<https://savanne.be/1031-release-notes-mar-2014/>)
  - Testing with Angular.JS (<https://savanne.be/1022-testing-with-angular-js/>)

passion for excellent engineering.

[More about me \(/about/\)](#)

Search for:

- [RSS Feeds \(/feeds\)](#)
- [Twitter \(http://twitter.com/rubenv/\)](http://twitter.com/rubenv/)

© 2003 - 2014 Ruben Vermeersch (RubenV) (<http://www.savanne.be/>)

This is a personal web page. Things said here do not represent the position of my employer.