

PG6300-14 Webutvikling & API-design lecture notes 09: Testing

Martin Lehmann

13th April 2015

1 Intro

Core functionality is now complete (except security in WebSockets – not part of the curriculum).

Stability time! Testing is a great way of ensuring stability AND expressing user stories (thus removing the need for extensive documentation).

Tests developers know when they made a mistake that messed up the rest of the application. You’ve all done (part of) this before.

A well-written application uses both end-to-end and unit tests on both the client and the server.

Minor side note: **Dev dependencies**

```
$ npm install --save mongoose
```

```
$ npm install --save-dev gulp
```

2 End-to-end testing

- Test *everything* from UI to database
- Slow, but very good for catching errors (catch everything)

2.1 Protractor

- A tool for running end-to-end tests in Angular JS applications
- Not limited to Angular, but specifically designed for Angular (by the AngularJS team)
- Protractor is a Node.js app

- Uses WebDriver & Selenium to run an actual browser (thus, slow)
- Install as dev dependency via NPM

```
$ npm install --save-dev protractor
```

- Protractor has very large files for an NPM package – don't include it as a production dependency as deployment will be slowed down.

- Set up WebDriver with Selenium "automagically"

```
$ ./node_modules/.bin/webdriver-manager update
```

```
Updating selenium standalone
```

```
downloading https://selenium-release.storage.googleapis.com/2.45/selenium-server-standalone-2.45.0.jar
```

```
Updating chromedriver
```

```
downloading https://chromedriver.storage.googleapis.com/2.14/chromedriver_mac32.zip ...
```

```
chromedriver_2.14.zip downloaded to /Users/theneva/Dropbox/WACT/PG6300 Webutvikling og
```

```
selenium-server-standalone-2.45.0.jar downloaded to /Users/theneva/Dropbox/WACT/PG6300
```

```
$
```

- Protractor is just a test *runner*. Time to add a testing framework!
- Several options
 - QUnit (for JQuery): inflexible, verbose, weak when it comes to asynchronous and promise-based testing
 - Jasmine (for Behaviour-Driven Development): Based on testing in Ruby. Much more concise than QUnit, but weak when it comes to asynchronous and promise-based testing. The choice for most Angular apps (including the Angular team)
 - Mocha: The choice for most Node applications. Flexible, pick & choose tools that fit your application. Well supported with all Angular tools. Will be using this.

2.2 Mocha

- Test framework (you write tests with Mocha, just like with JUnit for Java, NUnit for .NET, XCTest for Objective-C and Swift, and so on)
- Requires a few configuration options, but nothing too bad

2.3 Basic Protractor test

- Convention to put all tests in *project root/test/*
- Three different test methods:

- End-to-end: *project/test/e2e/*
- Node server: *project/test/node*
- Angular: *project/test/angular*
- ex:

```
$ mkdir -p test/e2e test/angular test/node
```

```
project root
'- test
  '- angular
    '- e2e
      '- node
```

2.3.1 First test time!

- Protractor is great for describing user stories; design tests with users in mind
- E-commerce: Come to the site, find a product, add it to the shopping card, complete the order
- Tests hit many (all?) parts of the application and ensure that common flows are always stable
- Too many tests (or too fine-grained tests) become too slow, take too long to write, and thus changing design becomes bad. Feature tests are good, but use with care

First test: *test/e2e/making-a-post.spec.js*

```
describe('making a post', function() {
  it('logs in and creates a new post', function() {
    // go to homepage
    // click 'login'
    // fill out and submit login form
    // submit a new post on the posts page

    // The new post should be visible as the first post on the page
  });
});
```

- Tests MUST be named *.spec.js for Protractor to find them – this allows us to have normal .js utility files that are not treated as tests
- "describe" describes a test scenario to give context. Can be nested!
- "it" is an actual test

- Final assertion on its own line ('The new post should be visible as the first post on the page'): Think of the final assertion first. What should ultimately be tested? Could describe the flow backwards so that you only need to think of one prerequisite at a time.

From pseudocode to a (barely) running test (test/e2e/making-a-post.spec.js):

```
describe('making a post', function() {
  it('logs in and creates a new post', function() {
    // go to homepage
    browser.get('http://localhost:3000');
    // click 'login'
    // fill out and submit login form
    // submit a new post on the posts page

    // The new post should be visible as the first post on the page
  });
});
```

Configure Protractor to use Mocha (and tell it where to find your tests)

project root/protractor.conf.js

```
exports.config = {
  framework: 'mocha',
  specs: [
    'test/e2e/**/*.spec.js'
  ]
};
```

Install Mocha as a dev dependency:

```
$ npm install --save-dev mocha
```

Run Protractor (the same command as previously, but without 'update'). **NB:** This requires the file retrieved at <http://localhost:3000> to be an actual Angular application. You cannot run this on an empty HTML file!

```
$ ./node_modules/.bin/protractor
```

This...

- Opens a browser window briefly (that's WebDriver/Selenium, required to actually perform the tests)
- Prints information in the console:

```
./node_modules/.bin/protractor
Starting selenium standalone server ...
[launcher] Running 1 instances of WebDriver
Selenium standalone server started at http://10.21.24.41:64200/wd/hub
```

```
. making a post logs in and creates a new post: 457ms
```

```
1 passing (460ms)
```

```
Shutting down selenium standalone server.
[launcher] 0 instance(s) of WebDriver still running
[launcher] chrome #1 passed
```

Start node inside Protractor to avoid having to keep the server running at all times.

project root/protractor.conf.js

```
exports.config = {
  framework: 'mocha',
  specs: [
    'test/e2e/**/*.spec.js'
  ],
  onPrepare: function() {
    require('./server-node/hello-tests-server.js');
  }
};
```

...but this makes you unable to already have a server running. Use a different port set by environment variables!

Edit server.js to pick its port from an environment variable:

```
var express = require('express');
var app = express();
...
var port = process.env.PORT || 3000;
...
app.listen(port, function() {
  console.log('Listening on port ' + port);
});
```

Then set the environment variable inside Protractor's onPrepare:

project root/protractor.conf.js

```

exports.config = {
  framework: 'mocha',
  specs: [
    'test/e2e/**/*.spec.js'
  ],
  onPrepare: function() {
    process.env.PORT = 3001;
    require('./server-node/hello-tests-server.js');
  }
};

```

...and finally point the test code to the port of the server started by Protractor (3001):

```

describe('making a post', function() {
  it('logs in and creates a new post', function() {
    // go to homepage
    browser.get('http://localhost:3001');
    // click 'login'
    // fill out and submit login form
    // submit a new post on the posts page

    // the new post should be visible as the first post on the page
  });
});

```

Protractor **Locators**

Interact with the DOM elements on the page! For more information, see <https://github.com/angular/protractor/blob/master/docs/locators.md>.

First, get a hold of the DOM element:

```

// find an element using css selector
by.css('someclass');

// find an element using id
by.id('someid');

// find an element by ng-model (e.g. username)
by.model('username');

// find an element by binding (e.g., {{currentUser}})
by.binding('currentUser');

```

```
// find an element by repeater (e.g., item in items  
by.repeater('item in items');
```

Pass the locator to `element(locator)`, and you can interact with it:

```
// click a button or link  
element(by.css('.mybutton')).click();  
  
// fill out a text input  
element(by.css('.username-input')).sendKeys('theneva');
```

NB: These are all asynchronous events!

Now to actually write a test, add the class 'login' to the Login link in the navbar:

```
<a class="navbar-text navbar-right login" href="/#/login" ng-if="!currentUser">Log in</a>
```

Then use Protractor locators to select and click the link:

```
// click 'login'  
var loginLink = element(by.css('nav .login'));  
loginLink.click();
```

NB: Entering a nonexistent ID results in a failed test.

Now fill in the login form, click the login button, and save a new item:

```
// fill out and submit login form  
var usernameInput = element(by.model('username'));  
usernameInput.sendKeys('theneva');  
  
var passwordInput = element(by.model('password'));  
passwordInput.sendKeys('1234');  
  
// Click the login button  
var loginButton = element(by.css('.button-login'));  
loginButton.click();  
  
// save a new item on the home page  
var nameInput = element(by.model('newItem.name'));  
nameInput.sendKeys('Some random item');  
  
var saveButton = element(by.css('.button-save'));  
saveButton.click();
```

Now get a hold of the the item on top of the list so we are ready to make our assertion that the new item was added to the top:

```
// the new item should be visible as the first item on the page
element.all(by.repeater('item in items')).then(function(items) {
    var firstItem = items[0];
    firstItem.element(by.css('.item-name')).getText().then(function(firstItemName) {
        console.log(firstItemName);
    });

    firstItem.element(by.css('.item-author')).getText().then(function(firstItemAuthor) {
        console.log(firstItemAuthor);
    });
});
```

Finally, to make our assertions, we need an assertion library: `chai`. Install as a dev-dependency with NPM:

```
$ npm install --save-dev chai
```

Require the appropriate function and make assertions in `test/make-an-item.spec.js`:

```
var expect = require('chai').expect;
...
firstItem.element(by.css('.item-name')).getText().then(function(firstItemName) {
    expect(firstItemName).to.equal(itemContent);
});

firstItem.element(by.css('.item-author')).getText().then(function(firstItemAuthor) {
    expect(firstItemAuthor).to.equal(username);
});
```

If the returned content was wrong (e.g. the logged in user's username was 'hello', but the author of the post still showed as 'theneva', you would see message in 2.3.1.


```

1) making a post logs in and creates a new post
0 passing (1s)
1 failing

1) making a post logs in and creates a new post:
  AssertionError: expected 'theneva' to equal 'hello'
    + expected - actual
    + "hello"
    - "theneva"

    at test/e2e/make-an-item.spec.js:43:32
    at Array.forEach (native)
    From: Task: making a post logs in and creates a new post

```

Figure 1: Wrong assertion

`browser.pause()` in your tests is a nice tool that lets you pause the test execution and step through so you can see exactly what happens.

The current test stack is as follows: **Protractor** sets up **Selenium** and **WebDriver** and spins up a separate instance of the server on port 3001, before running all tests in `project root/test/e2e/` using the testing framework **Mocha**, which in turn uses **Chai** to make assertions.

A final trick for end-to-end testing can help avoid that the tests get messy. This can be avoided with the NPM package `chai-as-promised`, but we will not be looking into this in detail.

One key thing to note is that we are working directly with the application's database. Clearing out the test data is probably a good idea, but figuring out a good way to do it requires some thought. A very simple way to deal with this is simply dropping the database with a Protractor `afterEach` filter – simply require the database file (which exports the connected `Mongoose` instance) and run `the-module.connection.db.dropDatabase()`;

If you only have time and resources to test one aspect of your application, then end-to-end tests are probably the absolutely most important kind of tests to write. They use the application like a user is expected to use it, and will be a major player in preventing bugs that directly affect the end users.

3 Unit testing

- Test isolated components (single units, also known as functions and classes)
- Used in both Node and Angular, but completely separately
- Every test is ignorant of the other tests

- These are the tests you write before coding
- Great for test-driven development (TDD)

Unit testing in Node is not quite unit testing: True unit tests would not actually hit external dependencies like the database (but mock them). True unit tests in Node turns out to be hard work that results in inflexible code.

The Node server will be tested on the controller level, while the Angular code will be truly unit tested.

First Node "unit" test with Mocha to test the environment: *project root*/test/node/controllers/items.spec.js

```
var expect = require('chai').expect;

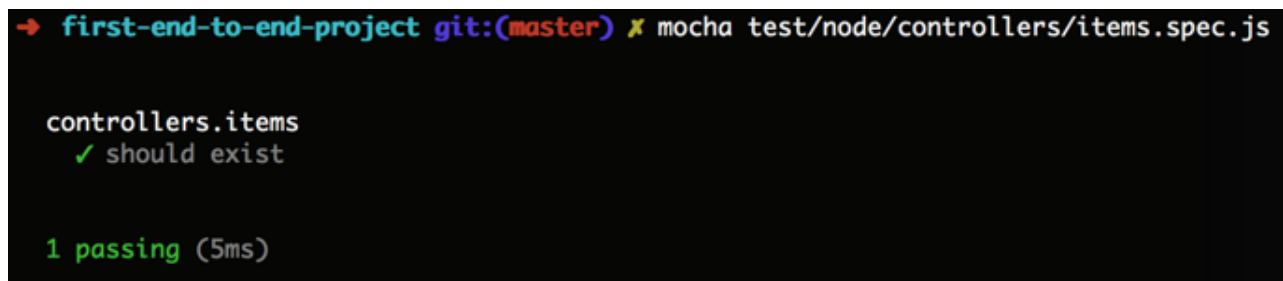
var controller = require('../../server-node/controllers/items');

describe('controllers.items', function() {
  it('should exist', function() {
    expect(controller).to.exist;
  });
});
```

This test can be run directly with Mocha (as long as it is in the path (for example installed globally with *npm install -g mocha*):

```
$ mocha test/node/controllers/items.spec.js
```

This should result in the output described in 3



```
→ first-end-to-end-project git:(master) ✖ mocha test/node/controllers/items.spec.js

controllers.items
  ✓ should exist

1 passing (5ms)
```

Figure 2: First Node Mocha test passing

Set up a single command to run all Node tests in *project root*/test/mocha.opts:

```
test/node
--recursive
```

Now all tests (found recursively) in `/test/server` will be executed by running `$ mocha`.

There are several different test reporters for Mocha. The default one is good for seeing which parts of the code are being tested. You can try others, though, such as

```
$ mocha -R nyan
```

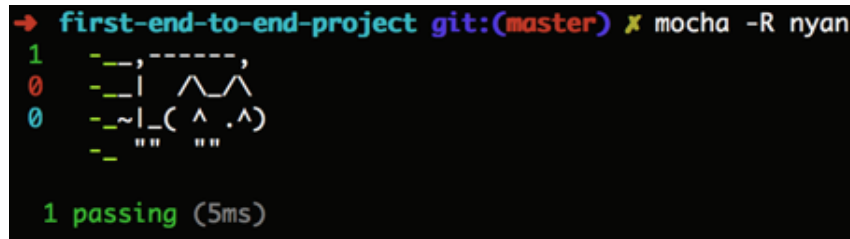


Figure 3: Nyan test reporter

3.1 Code coverage

Blanket!

```
$ npm install --save-dev blanket
```

Used by writing the following file `test/server/support/coverage.js`

```
var path = require('path');
var blanket = require('blanket');

blanket({
  pattern: [
    path.resolve(__dirname, '../..../controllers')
  ]
});
```

...and requiring it from `test/mocha.opts`:

```
test/server
--recursive
--require test/server/support/coverage
```

Create a report with the `html-cov` reporter in Mocha and writing the output to a file `coverage.html`, before opening the file in a browser:

```
$ mocha -R html-cov > coverage.html
$ open coverage.html
```

4 npm test

Let us finally use package.json for something other than dependencies!

```
{
  "name": "itemsApp",
  "dependencies": "...",
  "devDependencies": "...",
  "scripts": {
    "test": "./node_modules/.bin/mocha && ./node_modules/.bin/protractor"
  }
}
```

This *script* can be run by simply typing:

```
$ npm test
```