Unit Tests and Code Coverage

in Javascript with Help of Node.js

- Author: **Alexander Shvets**
- Role: **Developer**
- Year: **2013**

Goals of this presentation

- Discuss the two most uncovered area: **unit testing** and **code coverage** for Javascript.
- Prove that Node.js can be used for **maintaining any Web Project**, whether it's written in Java, Ruby or Python.
- Explore ideas about how to provide **code modularity** in javascript.
- We use Node.js as **complimentary technology** that doesn't replace your current one.

Why is it important?

- Having unit tests **increases quality** of code.
- Having code coverage provides **awareness and better control** of what is going on with project.
- Modularity in code **increases maintainability** of the project, letting us to to break code into **manageable parts**, **easy to read** and **easy to fix** forthcoming issues.

Node.js

What is it?

- Was created by **Ryan Dahl** starting in 2009, and its development and maintenance is sponsored by **Joyent**, his former employer.
- It's **scripting language** - no compilation required, convenient for writing automation scripts.
- It **utilizes JavaScript** as its scripting language. It uses Google Chrome v8 Javascript engine.
- It's javascript that is **freed from the browser's chains**.
- It can be used from **command line** and as part of **server-side** collection of technologies.
- Achieves high throughput via **non-blocking I/O model** and a **single-threaded event loop**.
- It has a lot of **packages** (libraries) for easy extending existing functionality.
- You use **node package manager** to deliver node packages to your computer.
- It's **technology-in-demand**. Some companies that use it: **github.com**, **linkedin.com**, **vonage.com**, **ebay.com**, **microsoft.com**, **trello.com**.

- It’s **new and promising technology**. See Tessel - internet-connected microcontroller progammable in Node.js.

- A lot of hosting services support Node.js: **Heroku**, **Joyent**, **CloudFoundry**, **OpenShift**, **Cloudnode**, **WindowsAzure**.

Installation

If you are on Apple computer, you can use **homebrew** tool to install it:

```
brew install node
```

It will install **node package manager** (npm) as well:

```
node -v
npm -v
```

On Windows you can download node and npm as one installation from Node.js Downloads Page.

Using Node.js: Webserver example

This simple web server responds with “Hello World” for every request.

```
// hello_world.js

var http = require('http');

var server = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
});

server.listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

To run this code, execute it from the command line:

```
Server running at http://127.0.0.1:1337/
```

Installing node packages

You can install node packages:

- **globally** (package will be available for all projects): in **/usr/local//lib/node_modules**;
- **locally** (package will be available for current project only): in **your_project_root/node_modules**;

This is the example of global installation:

```
npm install -g grunt-cli
```

and local installation:

```
npm install grunt-cli
```

Saving packages locally is good practice - this way you will have the ability to quickly reproduce your production environment - dev/prod parity.

In example above we are installing **grunt** build tool.

You can execute grunt command now:

```
cd your_project_root
```

```
node_modules/grunt-cli/bin/grunt -version
```

Keep track of node packages used by project

You can create **package.json** file used by npm to keep track of project dependencies. This is sample file:

```
{
  "name": "your_project",
  "version": "1.0.0",
  "description": "Npm package for your_project",
  "author": "John Smith <jsmith@site.com>",
  "engines": {
    "node": ">= 0.10.16"
  },
  "dependencies": {},
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-cli": "~0.1.9"
  }
}
```

With this file in place you can install all required dependencies (grunt, grunt-cli) in one command:

```
npm install
```

You can also install single package with automatic insert of package name/description into **package.json** as development dependency:

```
npm install grunt-cli --save-dev
```

Memo for Java/Ruby Developers

Command

- Interpr.
- Pack. Man.
- Vers. Man.
- Build

Node.js

- node
- npm
- nvm
- grunt

Ruby

- ruby
- gem
- rvm
- rake

Java

- java
- maven?
- ???
- ant/maven

Karma Framework

What is it?

- Created by **AngularJS** team.
- Previously named as **Testacular**.
- Universal **layer built on top** of other testing/coverage/etc libraries **with common configuration**.
- **Agnostic to testing framework**: you describe your tests with Jasmine, Mocha, QUnit, or write a simple adapter for any framework you like.
- Can test **same code in different browsers** simultaneously.
- Can **debug** with the help of WebStorm IDE or Google Chrome.
- Can run your tests in **headless mode** with the help of **PhantomJS** library.

Installation

Install karma:

```
npm install karma --save-dev
```

Create karma configuration file. You can keep it in javascript:

```
node_modules/karma/bin/karma init karma.conf.js
```

or coffeescript:

```
node_modules/karma/bin/karma init karma.conf.coffee
```

Script will ask few questions and at the end create **karma.conf.js** or **karma.conf.coffee** file.

Installation (continued)

Now, you need to install additional packages.

Install browser launchers:

```
npm install karma-chrome-launcher --save-dev
npm install karma-firefox-launcher --save-dev
npm install karma-safari-launcher --save-dev
npm install karma-phantomjs-launcher --save-dev
```

Install preprocessors:

```
npm install karma-coffee-preprocessor --save-dev
npm install karma-html2js-preprocessor --save-dev
```

Install support for jasmine:

```
npm install karma-jasmine --save-dev
```

Karma Configuration File

Review your configuration file:

```
# karma.conf.coffee

module.exports = (config) ->
  config.set
    basePath: '.'

    frameworks: ['jasmine']

    files: [
      # external libraries

      process.env.GEM_HOME '/gems/jquery-rails-3.0.4/vendor/assets/javascripts/jquery.js'

      # project libraries

      'app/assets/javascripts/*.js',
      'app/assets/javascripts/*.coffee',

      # specs

      {pattern: 'spec/javascripts/*_spec.js', included: true}
      {pattern: 'spec/javascripts/*_spec.coffee', included: true}
    ]

    # list of files to exclude
    exclude: []
    ...
```

Karma Configuration File (continued)

```
...
preprocessors:
  '**/*.coffee': ['coffee']

reporters: ['dots']

port: 9876

colors: true

logLevel: config.LOG_INFO

autoWatch: false

browsers: ["PhantomJS"]

captureTimeout: 60000

singleRun: true

reportSlowerThan: 500
```

Karma Configuration File (continued)

- **basePath** points to the root of your project
- **frameworks** describes used frameworks (we use jasmine only)
- **files** should include
- original javascript code to be tested
- dependent external libraries
- specs and fixtures (if you plan to use them)
- **preprocessors** describe different actions/filters. Some of them:
  - how to process coffescript files (coffee);
  - how to build fixtures (html2js);
  - what files to include into code coverage (coverage);
- **reporters** define usage of “dots” reporter
- **browsers** describe in which browsers code should be tested. We use **PhantomJS** for headless tests
- **singleRun**: true is useful for running in CI server

Using Karma

Start karma:

```
node_modules/karma/bin/karma start
```

This command will run all specs in **spec/javascripts** folder and output results to console.

```
INFO [karma]: Karma v0.10.2 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.1 (Mac OS X)]: Connected on socket _juBSCUpZSBdU1wA-URx
.................................................
PhantomJS 1.9.1 (Mac OS X): Executed 47 of 47 SUCCESS (0.167 secs / 0.027 secs)
```

Code Coverage with Karma

- Install **karma-coverage** package:

```
npm install karma-coverage --save-dev
```

- Register your javascript/coffeescript libraries with **coverage** preprocessor and add **coverage** reporter:

```
# karma.conf.coffee
module.exports = (config) ->
  config.set
    ...

    preprocessors:
      'app/assets/javascripts/**/*.js': ['coverage']
      'app/assets/javascripts/**/*.coffee': ['coverage']

    reporters: ['dots', 'coverage']

    coverageReporter:
      type: 'html'
      dir: 'coverage'
    ...
```

- **coverageReporter** describe configuration of “coverage” reporter: type of report and location of it.

Using fixture for your specs

- Create fixture file:

```
<!-- spec/javascripts/fixtures/template.html -->
<div>something</div>
```

- Add fixture location to **files** sections and specify **html2js** preprocessor for html files inside **fixtures** directory:

```
# karma.conf.coffee
module.exports = (config) ->
  config.set
    ...

    files: [
      ...
      'spec/javascripts/fixtures/**/*.html'
    ]

    preprocessors:
      '**/*.html': ['html2js']

    ...
```

Using fixture for the spec (continued)

- Access **fixture** from the test through global **window.__html__** variable:

```
# spec/javascripts/some_coffee_spec.coffee
describe 'some_coffee_code', ->
  fixture = undefined

  beforeEach ->
    window.__html__ = window.__html__ || {};
    fixture = window.__html__['spec/javascripts/fixtures/template.html']

  it 'access div element', ->
    el = $(fixture).find('#div')
    expect(el).toBeDefined()
```

Modularity in Node.js and Browser Javascript

Options

- Use **home-grown library** for implementing modularity
- Use **CommonJS** specification
- Use **RequireJS** framework as implementation of CommonJS specification
- Wait for upcoming **ECMA script 6** implementation for javascript
- Use **other language** that support modules, e.g. **Dart**

Using Anonymous Closure

- You can simulate modularity in Javascript with the help of anonymous closure:

```
(function () {
  // - all vars and functions are in this scope only
  // - still maintains access to all globals
}());
```

- It creates an anonymous function, and execute it immediately. All of the code inside the function lives in a closure.

- Notice the **()** around the anonymous function. Including () creates a function expression instead of function declaration. For example:

```
var MyModule = (function() {
  var exports = {};

  //Export foo to the outside world
  exports.foo = function() {
     return "foo";
  }

  //Keep bar private
  var bar = "bar";

  //Expose interface to outside world
  return exports;
})();

MyModule.foo(); // OK
MyModule.bar(); // error
```

Using JQuery.extend

You can use jquery's **extend** API in order to implement module:

```
function ModularityLibrary() {}

ModularityLibrary.prototype.createClass = function(definitions, extra_definitions) {
  var klass = function() {
    this.initialize.apply(this, arguments);
  };

  jQuery.extend(klass.prototype, definitions);

  if(extra_definitions) {
    jQuery.extend(klass.prototype, extra_definitions);
  }

  return klass;
};

ModularityLibrary.prototype.extendClass = function(baseClass, methods) {
  var klass = function() {
    this.initialize.apply(this, arguments);
  };

  jQuery.extend(klass.prototype, baseClass.prototype);
  jQuery.extend(klass.prototype, methods);

  return klass;
};

var Modularity = new ModularityLibrary();
```

Using JQuery.extend (continued)

Now, you can use it in your code:

```
// Create new class

var DisplayModule = Modularity.createClass({
  initialize: function () {},

  display: function(connector) {
    console.log("display");
  }

});

// Create instance of class

var displayObject = new DisplayModule();

// Call instance function

custoModule.display();
```

Working with CommonJS

- CommonJS is the **set of specifications** that define how to do modules in Javascript.
- Instead of running your Javascript code from a global scope, CommonJS starts out each of your Javascript files in their own **unique module context.**
- CommonJS adds two new variables which you can use to import and export other modules:
  - **module.exports** exposes variables to other libraries;
  - **require** function helps to **import** your module into another module.
- For example, javascript class and jasmine spec for it could look like:

```
// app/assets/javascripts/commonjs/example.js

module.exports.hello = function() {
  return 'Hello World'
};

// spec/javascripts/commonjs/example_spec.js
var example = require('../../../app/assets/javascripts/commonjs/example');

describe('example', function() {
  it("tests CommonJS", function() {
    example.hello();
```

```
    });
});
```

Working with CommonJS (continued)

- Configure **karma.conf.coffee**:

```
# karma.conf.coffee
 module.exports = (config) ->
   config.set
     ...

     frameworks: ['jasmine', 'commonjs]

     files: [
       'app/assets/javascripts/commonjs/*.js',
       {pattern: 'spec/javascripts/commonjs/*_spec.js', included: true}
       {pattern: 'spec/javascripts/commonjs/*_spec.coffee', included: true}
     ]

     preprocessors:
       'app/assets/javascripts/commonjs/*.js': ['commonjs'],
       'spec/javascripts/commonjs/*_spec.js': ['commonjs']
       'spec/javascripts/commonjs/*_spec.coffee': ['commonjs']
     ...
```

- You add **commonjs** as framework and mark files that use CommonJS with **commonjs** preprocessor.

CommonJS implementations

- Because CommonJS is just specification, you cannot use it directly in the browser. Node.js has it’s own implementation, but we cannot use it on clent side.
- Developers have different options to have it in browser. Some of them:
  - browserify, webmake - command line tools that wraps up your CommonJS-compatible code with simple implementation of **require** and **module.exports**.
  - NodeJS - asynchronous implementation of CommonJS specification.
  - List of other solutions

Working with RequireJS

- RequireJS uses another module format: Asynchronous Module Definition (AMD), originally created as part of the Dojo web framework.

- Compared to CommonJS, the main differences of AMD are:
  - Special syntax for specifying module imports - **define** (must be done at the top of each script)
  - No tooling required to use, works within browsers out of the box.
- Create RequireJS-compatible js code:

```
// app/assets/javascripts/requirejs/example.js

define('example', function() {
  var message = "Hello!";

  return {
    message: message
  };
});
```

Working with RequireJS (continued)

- Create jasmine spec for it:

```
// spec/javascripts/requirejs/example.js

require(['example'], function(example) {
  describe("Example", function() {
    it("should have a message equal to 'Hello!'", function() {
      console.log(example.message);
      expect(example.message).toBe('Hello!');
    });
  });
});
```

- Configure **karma.conf.coffee** to recognize RequireJS framework:

```
# karma.conf.coffee
 module.exports = (config) ->
   config.set
     ...
     frameworks: ['jasmine', 'requirejs]

     files: [
       'app/assets/javascripts/requirejs/*.js',
       {pattern: 'spec/javascripts/requirejs/*_spec.js', included: true}
       {pattern: 'spec/javascripts/requirejs/*_spec.coffee', included: true}
       'spec/javascripts/requirejs/spec-main.js'
     ]
     ...
```

You don’t have to preprocess requirejs files - it’s already part of karma framework.

Working with RequireJS (continued)

- Create main RequireJS file for tests only:

```
// spec/javascripts/requirejs/spec-main.js

// Grabs specs
var specs = [];

for (var file in window.__karma__.files) {
  if (window.__karma__.files.hasOwnProperty(file)) {
    if (/spec\.js$/.test(file)) {
      specs.push(file);
    }
  }
}

console.log(specs);

// Configures RequireJS for tests
requirejs.config({
  // Karma serves files from '/base'
  baseUrl: '/base/app/assets/javascripts/requirejs',

  paths: {
    'jquery': process.env.GEM_HOME + '//gems/jquery-rails-3.0.4/vendor/assets/javascripts/jquery'
  },

  // ask Require.js to load these files (all our tests)
  deps: specs,

  // start test run, once Require.js is done
  callback: window.__karma__.start
});
```

Using RequireJS in browser

- For using RequireJS in browser you have to download it and include into your html file.
- Your **html template file**:

```
\%html{:lang => "en"}
  \%head
    = javascript_include_tag "requirejs-2.1.8.min"
    = javascript_include_tag  "helper"
    = javascript_include_tag  "application"
```

- And your **application.js**:

```
require.config({
  baseUrl: 'assets/javascripts',

  paths: {
    app: '.'
  }
});

// Start the main app logic.
require(['jquery-1.10.2.min', 'helper'], function($, helper) {

  helper.do_something();
});
```

Links

- Node.js Home - http://nodejs.org
- Node Weekly newspaper archive - http://nodeweekly.com/archive
- Javascript Weekly newspaper archive - http://javascriptweekly.com/archive (http://nodeweekly.com/archive)
- Karma Runner Home - https://github.com/karma-runner/karma
- CommonJS Home - http://www.commonjs.org/
- CommonJS: Why and How - http://0fps.wordpress.com/2013/01/22/commonjs-why-and-how
- [JavaScript Module Pattern: In-Depth - http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html] (http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html)
- RequireJS Home - http://requirejs.org
- [Writing JavaScript Modules for Both Browser and Node.js - http://www.matteoagosti.com/blog/2013/02/24/writing-javascript-modules-for-both-browser-and-node] (http://www.matteoagosti.com/blog/2013/02/24/writing-javascript-modules-for-both-browser-and-node)
- [Tessel: internet-connected microcontroller progammable in Node.js - http://www.dragoninnovation.com/projects/22-tessel] (http://www.dragoninnovation.com/projects/22-tessel)
- [Design Patterns for JavaScript Applications - http://www.infoq.com/news/2013/09/javascript-design-patterns] (http://www.infoq.com/news/2013/09/javascript-design-patterns)

- [Getting Started with Node.js on Heroku - https://devcenter.heroku.com/articles/getting-started-with-nodejs] (https://devcenter.heroku.com/articles/getting-started-with-nodejs)
- [Five Helpful Tips When Using RequireJS - http://tech.pro/blog/1561/five-helpful-tips-when-using-requirejs] (http://tech.pro/blog/1561/five-helpful-tips-when-using-requirejs)

Thank You!

Hola! # Спасибо! # 谢谢! # Toda!

Questions?

Suggestions?