Mocha

expresso를 만든 LearnBoost의 TJ Holowaychuk가 만든 새로운 테스트 프레임워크

기능

Mocha는 TDD와 BDD를 모두 지원하는 것이 특징

- 브라우저 지원
- 간단한 비동기 테스트 지원
- 테스트 커버리지 리포팅
- 문자열 비교 지원
- 실행되는 테스트에 대한 자바스크립트 API
- CI에 대한 적절한 exit 상태 지원
- tty가 아닌 환경을 자동탐지하고 칼라 표시 무력화
- 제대로 된 테스트 케이스에 대해 uncaught exception 매핑
- 비동기 테스트 타임아웃 지원
- 테스트에 특화된 타임아웃
- growl 알림 지원
- 테스트 수행시간 보고
- 느린 테스트 하일라이팅
- 파일 워쳐 지원
- 전역 변수 메모리누출 탐지
- 선택적으로 정규표현직과 일치하는 테스트 수행
- 수행되는 루프에서 행이 걸리는 것을 막기 위해 자동으로 빠져나옴
- 테스트케이스와 테스트 슈트에 대한 쉬운 메다 생성
- mocha.opts 파일 지원
- node 디버거 지원
- done()의 다중 호출 탐지
- 원하는 assertion 라이브러리 사용가능
- 확장가능한 9개 이상의 보고서 지원
- 확장가능한 테스트 DSL이나 인터페이스
- before, after, before each, after each 훅
- 임의의 트랜스파일러(tranpiler) 지원(coffeed-script6 등)
- TextMate 번들

등등…

설치

```
$ sudo npm install -g mocha
```

Assertion

사용자가 원하는 어떤 assertion 라이브러리라도 가져다 쓸 수 있다. 공식 홈페이지에는 다음의 assertion 라이브러리를 소개 하고 있다.

- **should.js** : BDD style shown throughout these docs
- **expect.js** : expect() style assertions
- **chai** : expect(), assert() and should style assertions
- **better-assert** : c-style self-documenting assert()

인터페이스

BDD가 기본 스타일이다.

BDD

describe()와 it()으로 테스트슈트(유닛테스트를 하나로 묶은 것)와 유닛테스트를 작성하고 각 슈트와 테스트 전에 실행할 작업을 before(), after(), beforeEach(), afterEach()로 작성.

```
describe('BDD style', function() {
  before(function() {
    // excuted before test suite
  });

  after(function() {
    // excuted after test suite
  });

  beforeEach(function() {
    // excuted before every test
  });

  afterEach(function() {
    // excuted after every test
  });

  describe('#example', function() {
    it('this is a test.', function() {
      // write test logic
    });
  });
});
```

TDD

TDD 스타일에서는 suite()와 test()로 작성하며 각 슈트와 테스트전/후에 실행할 작업은 suiteSetup(), suiteTeardown(), setup(), teardown()을 사용. 기본 스타일이 아니기 때문에 실행 시 옵션으로 지정해야 하며 문서화도 잘 안되어 있다.

```
suite('TDD Style', function() {
  suiteSetup(function() {
    // excuted before test suite
  });

  suiteTeardown(function() {
    // excuted after test suite
  });

  setup(function() {
    // excuted before every test
  });

  teardown(function() {
    // excuted before every test
  });

  suite('#example', function() {
    test('this is a test', function() {
      // write test logic
    });
  });
});
```

기타

exports, QUnit 등의 스타일도 지원

테스트 작성

Synchronous code

테스트 부분인 it()에 테스트 로직과 assertion 작성. 예제에서는 node.js에 내장되어 있는 assert 함수를 사용했으나 편한 것을 사용하면 된다.

```
var assert  = require('assert');

describe('Example', function() {
  describe('calculation', function() {
    it('1+1 should be 2', function() {
      assert.equal(1+1, 2);
    });
  });
});
```

Asynchronous code

it()에 사용하는 함수에 파라미터로 done을 전달하면 자동으로 비동기 테스트로 인식하고 비동기 로직이 완료된 시점에서 파라미터로 받은 done()을 실행해 주면 테스트가 완료. assertion은 done()을 실행하기 전에 작성하면 된다. done()을 실행하지 않으면 기본 타임아웃인 2000ms후에 타임아웃 실패로 간주된다.(타임아웃은 변경 가능)

```
var assert  = require('assert')
    fs = require('fs');

describe('Example', function() {
  describe('calculation', function() {
    it('1+1 should be 2', function(done) {
      fs.readFile('example.txt', function(err, data) {
        done();
      });
    });
  });
});
```

done()함수는 node.js의 관례를 따르는 함수이기 때문에 다음과 같이 비동기 함수에 바로 전달해 주어도 된다.

```
var assert  = require('assert')
    fs = require('fs');

describe('Example', function() {
  describe('calculation', function() {
    it('1+1 should be 2', function(done) {
      fs.readFile('example.txt', done);
    });
  });
});
```

테스트 실행

프로젝트에 test 폴더를 생성하고 해당 폴더에 test할 코드들을 작성 후 프로젝트 폴더에서 다음을 실행한다.

```
$ mocha
```

다음과 같이 파라메터로 파일명을 주면 특정 파일만 테스트 가능하다.

```
$ mocha test.js
```

옵션

mocha 명령어의 옵션은 다음과 같은 것들이 있다.

-h	mocha의 사용가능한 옵션들을 본다.
-w, –watch	테스트 대상 파일을 모니터링하다가 수정이면 다시 테스트를 실행한다.
–compilers <ext>:<module>	커피스크립트같은 transpiler를 사용할 경우 파일확장자와 트랜스파일러를 이 옵션으로 지정한다. 커피스크립트같은 경우 –compilers coffee:coffee-script 와 같이 사용한다.
-b, –bail	첫번째 예외에만 관심이 있다면 이 옵션을 사용한다.
-d, –debug	node.js의 디버거를 사용한다.
–globas <names>	전역변수의 이름을 콤마로 분리해서 지정한다. mocha는 전역변수의 메모리누출을 감지해서 경고하는데 의도한 전역변수인 경우 이 옵션으로 지정한다.
–ignore-leaks	전역변수의 메모리누출이 감지될 경우 테스트가 실패하는데 이 옵션을 사용하면 이 기능을 사용하지 않는다.
-r, –require <name>	이 옵션으로 노듈명을 지정하면 자동으로 테스트 수행시 해당 모듈을 require()해서 Object.prototype으로 포함시킨다. assertion 라이브러리로 should.js를 사용하는 경우 각 테스트파일마다 var should = require('should');를 추가하는 대신에 –require should 옵션을 사용하면 파일내에서 should 변수를 사용할 수 있다.(테스트 결과 이렇게 사용할 경우 지정한 should가 글로벌로 설치가 되어 있어야한다.) 하지만 module.exports에 접근해야 한다면 소스에서 require()를 사용해야 한다.
-u, –ui <name>	테스트 인터페이스를 지정한다. 기본값은 bdd 이며 앞에서 설명한 다른 인터페이스를 사용할 경우 이 옵션으로 지정해야 한다. tdd, exports, qunit 의 값을 사용할 수 있다.
-t, –timeout <ms>	지정한 타임아웃이 지나면 테스트를 실패로 간주한다. 기본값은 2초. –timeout 2s나 –timeout 2000과 같이 지정한다.
-s, –slow <ms>	시간이 오래걸리는 테스트를 하일라이트로 표시할 때의 기준시간으로 기본값은 75ms이다.
-g, –grep <pattern>	정규표현식으로 지정한 패턴과 일치하는 테스트만 실행한다.
-G, –growl	맥의 알림프로그램인 growl의 알림기능을 사용한다.
–interfaces	사용할 수 있는 인터페이스 목록을 보여준다.
–reporters	사용할 수 있는 보고서 형식 목록을 보여준다.

추가로 **-R, –reporter <name>** 옵션이 있다. 이것은 테스트 결과를 리포팅하는 형식을 지정한다. 기본값은 dot 이다. 사용할 수 있는 리포팅 형식에는 다음과 같은 형식들이 있다.(각 리포트형식의 스크린샷은 mocha 홈페이지에서 볼 수 있다.)

- dot - dot matrix
- doc - html documentation
- spec - hierarchical spec list
- json - single json object
- progress - progress bar
- list - spec-style listing
- tap - test-anything-protocol
- landing - unicode landing strip
- xunit - xunit reportert
- teamcity - teamcity ci support
- html-cov - HTML test coverage
- json-cov - JSON test coverage
- min - minimal reporter (great with –watch)
- json-stream - newline delimited json events

- markdown - markdown documentation (github flavour)

mocha.opts

./test/mocha.opts에 위의 옵션들을 추가하여 mocha를 실행할 때마다 자동으로 옵션을 추가할 수 있다.

```
--require should
--reporter dot
--ui bdd
```

위와 같이 mocha.opts 파일을 작성한 후 mocha를 실행하면 mocha –require should –reporter dot –ui bdd를 실행한 것과 동일한 결과가 된다.

```
$ mocha --reporter list --growl
```

mocha.opts 파일을 작성한 상태에서 옵션을 추가하면 mocha.opts 파일에 포함되어 있는 옵션과 함께 실행된다.

예제

아래는 Mocha + Chai(Assertion 모듈)를 이용한 테스트 예제이다.

준비

testPrj 폴더를 만든 후 폴더로 이동하여 `$ npm install chai` 명령으로 chai를 설치한다. 그 후 lib 폴더와 test 폴더를 생성한다. lib 폴더에는 테스트할 모듈이 들어가며 test 폴더에 유닛 테스트할 코드들이 들어간다.

테스트할 모듈 작성

lib 폴더에 아래의 두 모듈을 생성한다.

tags.js

```
exports = module.exports = {};

exports.parse = function(args, defaults, replacements) {
    var options = {};
    if (typeof defaults === "object" && !(defaults instanceof Array)) {
        options = defaults
    }

    if (typeof replacements === "object" && !(defaults instanceof
Array)) {
        for (var i in args) {
            var arg = args[i];
            if (arg.charAt(0) === "-" && arg.charAt(1) != "-") {
                arg = arg.substr(1);
                if (arg.indexOf("=") !== -1) {
                    arg = arg.split("=");
                    var keys = arg.shift();
                    var value = arg.join("=");

                    arg = keys.split("");
                    var key = arg.pop();
                    if (replacements.hasOwnProperty(key)) {
                        key = replacements[key];
                    }

                    args.push("--" + key + "=" + value);
                } else {
                    arg = arg.split("");
                }

                arg.forEach(function(key){
                    if (replacements.hasOwnProperty(key)) {
                        key = replacements[key];
                    }
                    args.push("--" + key);
                });
            }
        }
    }

    for (var i in args) { //Cycle through args
        var arg = args[i];
        //Check if Long formed tag
        if (arg.substr(0, 2) === "--") {
            arg = arg.substr(2);
            //Check for equals sign
            if (arg.indexOf("=") !== -1) {
                arg = arg.split("=");
                var key = arg.shift();
                var value = arg.join("=");

                if (/^[0-9]+$/.test(value)) {
                    value = parseInt(value, 10);
                }
                options[key] = value;
            } else {
                options[arg] = true;
            }
        }
    }
    return options;
}
```

search.js

```
var fs = require("fs");

exports = module.exports = {};

//A Modified Snippet from Christopher Jeffrey
http://stackoverflow.com/questions/5827612/node-js-fs-readdir-recursive
-directory-search
exports.scan = function(dir, depth, done) {
    depth--;
    var results = [];
    fs.readdir(dir, function(err, list) {
        if (err) return done(err);
        var i = 0;
        (function next() {
            var file = list[i++];
            if (!file) return done(null, results);
            file = dir + '/' + file;
            fs.stat(file, function(err, stat) {
                if (stat && stat.isDirectory()) {
                    if (depth !== 0) {
                        var ndepth = (depth > 1) ? depth-1 : 1;
                        exports.scan(file, ndepth, function(err, res) {
                            results = results.concat(res);
                            next();
                        });
                    } else {
                        next();
                    }
                } else {
                    results.push(file);
                    next();
                }
            });
        })();
    });
};

exports.match = function(query, files){
    var matches = [];
    files.forEach(function(name) {
        if (name.indexOf(query) !== -1) {
            matches.push(name);
        }
    });
    return matches;
}
```

테스트 코드 작성

test 폴더 아래에 아래의 테스트할 코드들을 작성한다.

tagsSpec.js

```
var expect = require("chai").expect;
var tags = require("../lib/tags.js");

describe("Tags", function(){
    describe("#parse()", function(){
        it("should parse long formed tags and convert numbers",
function(){
            var args = ["--depth=4", "--hello=world"];
            var results = tags.parse(args);

            expect(results).to.have.a.property("depth", 4);
            expect(results).to.have.a.property("hello", "world");
        });
        it("should fallback to defaults", function(){
            var args = ["--depth=4", "--hello=world"];
            var defaults = { depth: 2, foo: "bar" };
            var results = tags.parse(args, defaults);

            var expected = {
                depth: 4,
                foo: "bar",
                hello: "world"
            };

            expect(results).to.deep.equal(expected);
        });
        it("should accept tags without values as a bool", function(){
            var args = ["--searchContents"];
            var results = tags.parse(args);

            expect(results).to.have.a.property("searchContents", true);
        });
        it("should accept short formed tags", function(){
            var args = ["-sd=4", "-h"];
            var replacements = {
                s: "searchContents",
                d: "depth",
                h: "hello"
            };

            var results = tags.parse(args, {}, replacements);

            var expected = {
                searchContents: true,
                depth: 4,
                hello: true
            };

            expect(results).to.deep.equal(expected);
        });
    });
});
```

Synchronous code를 테스트하는 코드이다. it 함수에 테스트할 코드를 넣어두었다. 위 코드는 Tags 클래스(또는 네임스페이스 등…) 안의 parse() 함수를 테스트하겠다는 의미로 describe함수를 사용했다. 첫 번째 it 함수 블럭에서는 parse함수가 태그와 파라메터를 제대로 parsing하는지와 파라메터가 숫자인 경우 이를 제대로 변환하는지를 테스트한다. 이를 확인하기 위해 테스트할 argument를 정의하고 리턴값을 받아 chai 모듈의 expect 함수로 리턴값을 확인한다. 만약 parse함수에서 파라메터가 숫자인 경우 이를 변환하지 않았다면 (parseInt 부분이 없었다면) 관련 오류가 발생할 것이다. 첫 번째 테스트 블럭을 이해했다면 나머지 세 개의 테스트 블럭도 이해하기 어렵지 않을 것이다.

searchSpec.js

```
var expect = require("chai").expect;
var search = require("../lib/search.js");
var fs = require("fs");

describe("Search", function(){
    describe("#scan()", function(){
        before(function() {
            if (!fs.existsSync(".test_files")) {
                fs.mkdirSync(".test_files");
                fs.writeFileSync(".test_files/a", "");
                fs.writeFileSync(".test_files/b", "");
                fs.mkdirSync(".test_files/dir");
                fs.writeFileSync(".test_files/dir/c", "");
                fs.mkdirSync(".test_files/dir2");
                fs.writeFileSync(".test_files/dir2/d", "");
            }
        });

        after(function() {
            fs.unlinkSync(".test_files/dir/c");
            fs.rmdirSync(".test_files/dir");
            fs.unlinkSync(".test_files/dir2/d");
            fs.rmdirSync(".test_files/dir2");
            fs.unlinkSync(".test_files/a");
            fs.unlinkSync(".test_files/b");
            fs.rmdirSync(".test_files");
        });
        it("should retrieve the files from a directory", function(done)
{
            search.scan(".test_files", 0, function(err, flist){
                expect(flist).to.deep.equal([
                    ".test_files/a",
                    ".test_files/b",
                    ".test_files/dir/c",
                    ".test_files/dir2/d"
                ]);
                done();
            });
        });
        it("should stop at a specified depth", function(done) {
            search.scan(".test_files", 1, function(err, flist) {
                expect(flist).to.deep.equal([
                    ".test_files/a",
                    ".test_files/b",
                ]);
                done();
            });
        });
    });
    describe("#match()", function(){
        it("should find and return matches based on a query",
function(){
            var files = ["hello.txt", "world.js", "another.js"];
            var results = search.match(".js", files);
            expect(results).to.deep.equal(["world.js", "another.js"]);

            results = search.match("hello", files);
            expect(results).to.deep.equal(["hello.txt"]);
        });
    });
});
```

Asynchronous code를 테스트하는 코드이다. scan() 함수를 테스트하기 전 작업과 후 작업을 정의하기 위해 before() 함수와 after() 함수를 사용하였다. 이것은 scan() 함수의 디렉토리, 파일 스캔 기능을 위해 fs 모듈을 이용하여 생성하고 삭제하는 작업을 수행한다. 실제 존재하는 디렉토리, 파일을 이용할 수도 있으나 테스트하기 위한 용도로는 이 정도가 충분할 것이다. 첫 번째 테스트 블럭은 .test_files 폴더를 탐색 후 여기에 before() 함수에서 만들어둔 디렉토리, 파일이 스캔되는지 확인한다. scan() 함수는 비동기로 실행되기 때문에 done 파라메터를 이용하여 테스트가 끝난 후 done()함수가 호출되도록 했다. 이에 반해 마지막 테스트 블럭은 before(), after() 작업도 없으며 synchronous code를 테스트하도록 정의하였다. 자세한 내용은 아래 '참고' - '예제'의 Testing in Node.js 페이지를 참고한다.

위에서처럼 모듈을 테스트하려면 해당 모듈의 input, output이 확실히 정의되어야 한다. 그렇지 않으면 테스트 케이스를 작성하는데 어려움이 있을 것이다.

Webstorm을 이용한 테스트

다음 사이트를 참고한다.

- Debugging mocha unit tests with WebStorm step by step