mocha : node.js 테스트 프레임워크

2012/04/14 04:39 by Outsider in node.js

제가 쓴 책인 Node.js 프로그래밍의 "유닛 테스트"장에서 expresso와 vows를 테스트 프레임워로 설명했었습니다. 이 두 프레임워크는 node.js TDD 프레임워크 : expresso와 node.js를 위한 비동기 BDD 프레임워크 Vows에서도 포스팅을 올린적이 있습니다. 이 두 테스트 프레임워크는 작년에 제가 쓰던 프레임워크였습니다. 책을 쓰는 작업이 마무리에 들어 갈 때 쯤 새로운 테스트 프레임워크인 mocha에 대해서 알게 되었습니다. mocha에 대한 설명을 추가할까 하는 고민도 잠시 했었지만 이미 책이 마무리 단계였기 때문에 책에는 넣지 않고 블로그에서 설명합니다.

mocha

mocha는 사이트에 사이트에서 simple, flexible, fun javascript test framework for node.js & the browser라고 소개하고 있습니다. mocha는 expresso를 만든 LearnBoost의 TJ Holowaychuk가 만든 새로운 테스트 프레임워크로 expresso의 사이트를 보면 Be sure to check out Expresso's successor Mocha라고 나와있습니다. TJ Holowaychuk이 expresso를 만들고 개선해야 되겠다고 느낀 것들이 mocha로 발전해서 새로 개발한 것으로 보이고 더이상 expresso는 수정이나 기능추가를 하지 않는 것으로 보입니다.

테스트 프레임워크는 테스트의 접근방식에서 크게 TDD와 BDD로 나눌 수 있습니다. 위에서 얘기했던 expresso는 TDD 프레임워크이고 vows는 BDD 프레임워크입니다. mocha 사이트에도 얘기하듯이 mocha는 TDD와 BDD를 모두 지원하고 있기 때문에 테스트프레임워크를 바꾸지 않더라도 개발자가 원하는 방식으로 테스트를 할 수 있습니다. mocha는 많은 기능들을 내세우고 있지만 관심갈만한 기능들을 보면 다음과 같습니다.

- 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

mocha를 테스트 프레임워크라고 얘기했지만 실제로 사용해 보면 일반적인 테스트 프레임워크와는 약간 다른 형태를 띄고 있습니다. 실제로 사용해 보면 테스트 프레임워크라기 보다는 테스트 프레임워크를 담는 인터페이스 같은 형태를 띄고 있고 러너를 포함하고 있습니다. 그래서 다음과 같이 글로벌로 설치합니다.

```
1 $ npm install -g mocha                                    ?
```

앞에서 인터페이스 같은 형태를 띄고 있다고 설명했는데 이는 테스트를 검증하기 위한 Assertion을 포함하고 있지 않기 때문입니다. expresso와 vows를 보면 테스트 코드 형식을 포함해서 node.js의 Assert를 확장한 Assertion을 포함하고 있지만 mocha는 다릅니다. mocha에서는 어떤 Assertion 라이브러리라도 가져다가 사용할 수 있으며 자체 assertion은 지원하지 않습니다. mocha홈페이지에서는 TJ Holowaychuk가 만든 BDD 스타일의 should.js를 대표적으로 언급하고 있으며 expect() 스타일의 expect.js나 expect(), assert(), should 스타일을 모두 지원하는 chai를 소개하고 있지만 원하는 어떤 라이브러리든 사용할 수 있습니다.

그래서 mocha는 소스레벨에서 require하지 않으며 사용하는 assertion 라이브러리를 require()해서 사용하면 됩니다. 뒤에서도 설명하겠지만 mocha는 개발자에게 상당한 자유도는 주어 선호하는 assert스타일이나 TDD/BDD 스타일을 자유롭게 선택하도록 하고 있습니다. 항상 그렇듯이 자유도는 익숙해 지면 편리하고 강력한 장점이 있는 반면 "어떻게 쓰라는 거야?"라는 생각이 들만큰 다양한 방법을 제공하기 때문에 처음에 배우는 입장에서는 학습곡선이 큰 부담도 있습니다.(이 포스팅은 mocha에 대한 설명이므로 should.js같은 별도의 assertion 라이브러리에 대해서는 설명하지 않습니다.)

인터페이스

BDD

mocha는 기본적으로 BDD 스타일을 채택하고 있습니다. 테스트의 인터페이스는 다음과 같습니다.

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

describe()와 it()으로 테스트슈트와 유닛테스트를 작성하고 각 슈트와 테스트 전에 실행할 작업을 before(), after(), beforeEach(), afterEach()로 작성합니다.

TDD

TDD 스타일도 사용할 수 있는데 TDD 인터페이스는 다음과 같이 사용합니다.

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

TDD 스타일에서는 suite()와 test()로 작성하며 각 슈트와 테스트전/후에 실행할 작업은 suiteSetup(), suiteTeardown(), setup(), teardown()을 사용합니다. suiteSetup(), suiteTeardown()는 문서화도 안되어 있네요 ㅠㅠ BDD 스타일이 기본이므로 다른 인터페이스를 사용하려면 뒤에서 설명하는 테스트 실행시 옵션으로 인터페이스를 지정해 주어야 합니다.

exports

mocha에서는 expresso에서 사용하던 형태에 exports 인터페이스도 지원하고 있습니다.

```
module.exports = {
  before: function() {
    // excuted before test suite
  },
  after: function() {
    // excuted after test suite
  },
  beforeEach: function() {
    // excuted before every test
  },
  afterEach: function() {
    // excuted after every test
  },

  'exports style': {
    '#example': {
      'this is a test': function() {
        // write test logic
      }
    }
  }
};
```

위와 같이 테스트 슈트를 JSON 리터럴로 작성하며 before, after,, beforeEach, afterEach를 지원하고 있습니다.

QUnit

jQuery이 테스트 프레임워크인 QUnit 스타일의 테스트 인터페이스도 사용할 수 있습니다.

```
suite('Qunit');

test('#example', function(){
});

suite('Qunit part 2');

test('#example 2', function(){
});
```

이와 같은 인터페이스를 선택적으로 사용할 수 있습니다. 자신이 선호하는 스타일을 사용하면 됩니다.

테스트 작성

테스트는 동기와 비동기 테스트를 모두 지원하고 있습니다.

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

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

예제이므로 assert라이브러리를 따로 사용하지 않고 node.js에 내장되어 있는 assert모듈을 사용했습니다. BDD 인터페이스를 사용했으면 테스트부분인 it()에 테스트로직과 assertion을 작성하면 됩니다.

```
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, dat
        done();
      });
    });
  });
});
```

비동기 테스트도 간단하게 사용할 수 있습니다. expresso에서 vows로 바꾸었던 이유가 node.js에서 수없이 사용하게 되는 비동기로직 때문인데 상당히 쉽게 사용할 수 있습니다. 위처럼 it()에 사용하는 함수에 파라미터로 done을 전달하면 자동으로 비동기 테스트로 인식하고 비동기 로직이 완료된 시점에서 파라미터로 받은 done()을 실행해 주면 테스트가 완료됩니다. assertion은 done()을 실행하기 전에 작성하면 됩니다. done()을 실행하지 않으면 기본 타임아웃인 2000ms후에 타임아웃 실패로 간주합니다. 이 done()함수는 node.js의 관례를 따르는 함수이기 때문에 다음과 같이 비동기 함수에 바로 전달해 주어도 됩니다.

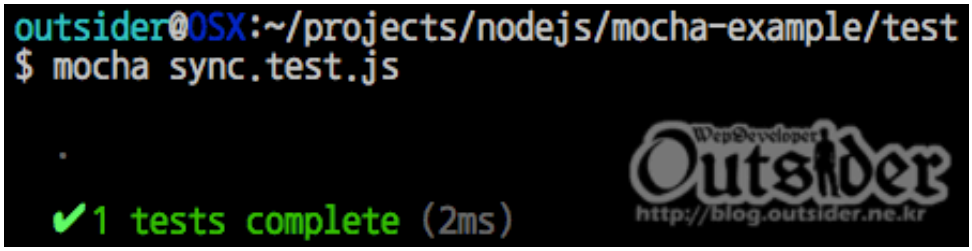
```
var assert  = require('assert')
    fs = require('fs');

```

```
4  describe('Example', function() {
5    describe('calculation', function() {
6      it('1+1 should be 2', function(done) {
7        fs.readFile('example.txt', done);
8      });
9    });
10 });
```

이렇게 콜백파라미터로 전달할 때는 첫 파라미터를 error 파라미터로 사용하는 node.js의 콜백함수 관례를 따라야 합니다.

테스트 실행

앞에서 mocha를 글로벌로 설치했으므로 mocha는 커맨드라인 명령어로 사용합니다. 위에서 작성한 코드는 require한 부분 외에는 별도의 의존성을 갖지 않으므로 mocha로 실행할 수 있습니다.



위의 화면에서 처럼 mocha 명령어로 테스트파일을 실행하면 테스트가 수행됩니다. mocha 명령어로 파일을 지정하면 해당 파일의 테스트를 실행하고 파일을 지정하지 않으면 현재 위치에서 test/ 디렉토리에 있는 모든 자바스크립트 파일을 실행합니다. mocha 명령어에서는 다음과 같은 옵션을 사용할 수 있습니다.

- -h : mocha의 사용가능한 옵션들을 봅니다.
- -w, --watch : 이 옵션으로 붙히면 테스트 대상 파일을 모니터링하다가 수정이면 다시 테스트를 실행합니다.
- --compilers <ext>:<module> : 커피스크립트같은 transpiler를 사용할 경우 파일확장자와 트랜스파일러를 이 옵션으로 지정합니다. 커피스크립트같은 경우 --compilers coffee:coffee-script 와 같이 사용합니다.
- -b, --bail : 첫번째 예외에만 관심이 있다면 이 옵션을 사용합니다.

- -d, --debug : node.js의 디버거를 사용합니다.
- --globas <names> : 전역변수의 이름을 콤마로 분리해서 지정합니다. mocha는 전역변수의 메모리누출을 감지해서 경고하는데 의도한 전역변수인 경우 이 옵션으로 지정합니다.(여기서 전역변수는 var 없이 변수선언해서 사용하는 것을 말합니다. node.js는 실제로 전역범위를 갖지는 않습니다.)
- --ignore-leaks : 전역변수의 메모리누출이 감지될 경우 테스트가 실패하는데 이 옵션을 사용하면 이 기능을 사용하지 않습니다.
- -r, --require <name> : 이 옵션으로 노듈명을 지정하면 자동으로 테스트 수행시 해당 모듈을 require()해서 Object.prototype으로 포함시킵니다. assertion 라이브로리로 should.js를 사용하는 경우 각 테스트파일마다 var should = require('should');를 추가하는 대신에 --require should 옵션을 사용하면 파일내에서 should 변수를 사용할 수 있습니다.(테스트 결과 이렇게 사용할 경우 지정한 should가 글로벌로 설치가 되어 있어야 합니다.) 하지만 module.exports에 접근해야 한다면 소스에서 require()를 사용해야 합니다.
- -u, --ui <name> : 테스트 인터페이스를 지정합니다. 기본값은 bdd 이며 앞에서 설명한 다른 인터페이스를 사용할 경우 이 옵션으로 지정해야 합니다. tdd, exports, qunit 의 값을 사용할 수 있습니다.
- -R, --reporter <name> : 테스트 결과를 리포팅하는 형식을 지정합니다. 기본값은 dot 입니다. 사용할 수 있는 리포팅 형식에는 다음과 같은 형식들이 있습니다.(각 리포트형식의 스크린샷은 mocha 홈페이지에서 볼 수 있습니다.)
  - 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)

- -t, --timeout <ms> : 지정한 타임아웃이 지나면 테스트를 실패로 간주하고 기본값은 2초입니다. --timeout 2s나 --timeout 2000과 같이 지정합니다.
- -s, --slow <ms> : 시간이 오래걸리는 테스트를 하일라이트로 표시할 때의 기준시간으로 기본값은 75ms입니다.
- -g, --grep <pattern> : 정규표현식으로 지정한 패턴과 일치하는 테스트만 실행합니다.
- -G, --growl : 맥의 알림프로그램인 growl의 알림기능을 사용합니다.
- --interfaces : 사용할 수 있는 인터페이스 목록을 보여줍니다.
- --reporters : 사용할 수 있는 보고서 형식 목록을 보여줍니다.

앞에서 언급한대로 test/ 디렉토리 안에 테스트파일들을 모아두는 것이 관례인데 test/mocha.opts 파일로 위의 옵션들을 지정해 두면 mocha로 모든 테스트를 수행할 경우 자동으로 이 파일에 지정된 옵션을 사용합니다.(라인단위로 읽으므로 한 줄에 하나의 옵션을은 지정해서 여러 줄로 사용하면 됩니다.)

테스트 프레임워크를 이것 저것 사용해 보다가 mocha로 최근에 정착하고 있습니다. 아직 오랜 시간 사용하지는 않아서 차후에 어떨지는 모르겠지만 현재로써는 만족스럽게 사용하고 있습니다. 저로써는 일단 자유도가 높아서 mocha를 기반으로 다양한 필요에 따라 assertion 라이브러리르 사용할 수 있다는 점이 좋아보이더군요.