

5. 보안서버 만들기

• HTTPS(HTTP over SSL)

- 데이터 암호화 통신

• SSL

- 넷스케이프 SSL → IETF의 TLS
- 대칭키 방식

• SSL 인증서

- 서비스를 제공하는 서버의 정보
- 신뢰성 있는 인증 기관(CA)의 서버 인증
- 서버 공개키 : 비대칭 암호화 용

HTTPS는 암호화 기능을 제공하는 SSL(Secure Socket Layer)을 사용하는 보안 HTTP 프로토콜이다.

SSL은 초기 넷스케이프 사에서 개발했고, 이후 국제 인터넷 표준화 위원회(IETF)에서 표준으로 채택하고 TLS라는 이름으로 계속 개발되고 있다.

SSL이라는 이름이 유명해서 아직도 SSL이라는 이름이 널리 사용되고 있다.

SSL은 대칭키를 이용해서 암호화한다.

HTTPS를 사용할 때는 인증서와 함께 사용한다.

인증서는 서비스를 제공하는 서버의 자격을 인증하며, 신뢰성 있는 인증 기관에서 인증서를 발급받아서 사용한다.

인증서에는 서버의 인증 정보와 비대칭키 암호화를 위한 공개키를 함께 저장하고 있다.

HTTPS는 비대칭 암호화와 대칭 암호화를 모두 사용하는데, 비대칭 암호화는 대칭 암호화에서 사용하는 키를 교환할 때 사용한다.

5. 보안서버 만들기

● 인증서 발급

▶ 공인된 인증 기관에서 인증서 발급

- 유료
- 인증 기관 : Verisign, Comodo

▶ 사설 인증서 발급

- 키와 인증서 생성 프로그램 : openssl
- <http://www.openssl.org>
- 경고 발생

HTTPS 통신을 하려면 서버에 인증서가 필요하다.

서비스를 할 인증서는 공인된 인증 기관에서 발급받는 것이 일반적이고 대부분 유료로 발급받는다.

사설 인증 기관으로는 verisign, comodo 등이 있다.

개발하는 과정에서는 구지 유료로 인증서를 유료로 발급받지 않고 자체 인증하는 인증서를 사용할 수 있다.

대신 공인된 기관에서 인증되지 않은 인증서를 사용하면, 웹 브라우저와 같은 클라이언트 프로그램에서는 경고 메시지를 보여준다.

사설 인증서를 발급하려면 <http://openssl.org> 에 방문해서 플랫폼에 맞는 openssl를 다운로드하고 설치한다.

5. 보안서버 만들기

● 사설 인증서로 보안 서버 만들기

▶ 보안 서버를 만들기 위해 필요한 것

- 키
- 인증서

▶ 인증서 발급 단계



사설 인증서를 발급받는 절차다.

첫 번째로 개인키를 생성하는 것으로 시작한다.

두 번째로 개인키를 이용해서 인증서 발급 요청(csr)을 생성한다.

세 번째로 인증서 발급 요청에서 인증서를 발급한다.

5. 보안서버 만들기

● 사설 인증서로 보안 서버 만들기

▶ 키 만들기

- `openssl genrsa -out key.pem 2048`

그림 속 명령은 openssl을 이용해서 개인키를 생성한다.
개인키는 key.pem 파일로 생성된다.

5. 보안서버 만들기

● 사설 인증서로 보안 서버 만들기

▶ 인증서 발급 요청(csr)

- `openssl req -new -key key.pem -out req.csr`

- 결과

Country Name (2 letter code) [AU]:kr

State or Province Name (full name) [Some-State]:seoul

Locality Name (eg, city) []:seoul

...

그림 속 명령은 개인키를 이용해서 인증서 발급 요청 파일(CSR, Certificate Signing Request)을 생성한다.

명령을 입력하면 서비스에 대한 정보를 요청한다.

정보를 입력을 마치면 CSR이 생성된다.

5. 보안서버 만들기

● 사설 인증서로 보안 서버 만들기

▶ 인증서 발급 승인

- `openssl x509 -req -in req.csr -signkey key.pem -out cert.pem -days 365`

- 결과

Signature ok

subject=/C=kr/ST=seoul/L=seoul/O=vanillastepdev/
emailAddress=wannabewize@gmail.com

Getting Private key

그림 속 명령을 이용해서 인증서를 발급받는다.

인증서 발급 요청 파일(req.csr)과 개인 키(key.pem)을 입력하고 유효기간을 365로 한 x509 포맷의 인증서를 발급하는 명령이다.

5. 보안서버 만들기

● 사설 인증서로 보안 서버 만들기

- https 모듈 사용
- 보안 서버 생성
 - ✓ `https.createServer(option, [REQUEST LISTENER])`
- 옵션
 - ✓ `key`
 - ✓ `cert`
 - ✓ `passphrase` : 개인키 암호

개인키와 서버 인증서가 준비됐다.

HTTPS 모듈을 이용해서 HTTPS 서버를 생성할 수 있다.

HTTPS 모듈은 기본 모듈이므로 별도의 설치 과정이 필요없다.

`https.createServer()` 함수를 이용해서 생성하고, 파라미터에는 개인키와 인증서 파일의 위치를 입력한다.

5. 보안서버 만들기

https 서버

```
var options = {  
  key: fs.readFileSync('./key.pem'),  
  cert: fs.readFileSync('./cert.pem')  
};  
https.createServer(options, function(req, res) {}).listen(3001);  
http.createServer(function(req, res) {}).listen(3000);
```

그림 속 코드는 http와 https 서버를 생성하는 코드다.

https 서버를 생성하는 함수에는 key, cert 이름을 키와 인증서 파일을 입력했다.

http와 https 서버는 서로 다른 포트를 이용한다.

https를 이용하는 요청 URL은 다음과 같다.

https://myServer.com:3001/path/resource

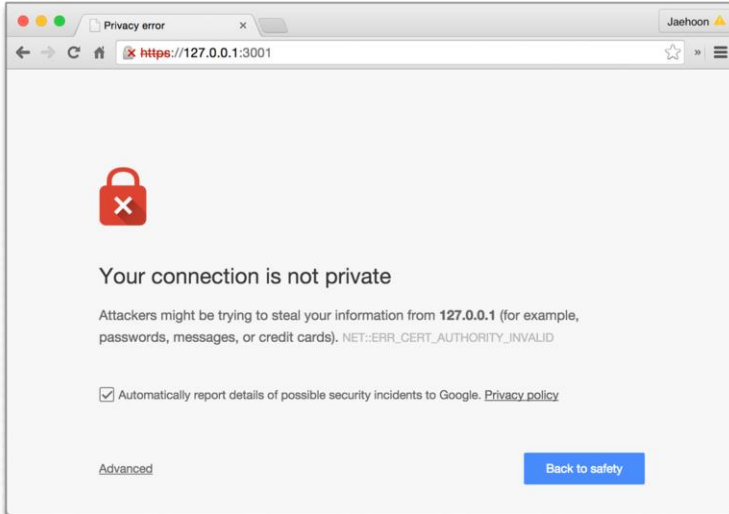
http를 이용하는 서비스 URL은 다음과 같다.

http://myServer.com:3000/path/resource

5. 보안서버 만들기

● 보안 서버 실행

▶ 브라우저 : <https://localhost:3001>



그림은 자체 발급받은 인증서를 사용한 서버에 접속한 모습이다.
공인된 기관에서 발급받은 인증서가 아니므로 그림과 같이 경고 메시지가 출력된다.

5. 보안서버 만들기

Express와 사용

```
var express = require('express');
var app = express();

// HTTP 요청
var server = http.createServer(app);
server.listen(3000);

// HTTPS 요청
var options = { key:..., cert: ...};
var secureServer = https.createServer(options, app);
secureServer.listen(3001);
```

Express를 이용해서 HTTPS 서비스를 작성하는 방법이다.

HTTP 서버와 HTTPS 서버를 작성하고 두 서버의 request 이벤트 리스너로 express 객체를 입력한다.

```
http.createServer(app);
```

```
https.createServer(option, app);
```

5. 보안서버 만들기

- 영리 목적으로 하는 개인 정보 수집 서비스
- 보안 서버 의무화
- 무료
 - ✓ <https://www.startssl.com>
 - ✓ <https://letsencrypt.org>
- 유료
 - ✓ verisign
 - ✓ symantec
 - ✓ comodo

현재 대한민국에서 사용자의 정보를 수집하는 서비스는 모두 보안서버 사용을 의무화하고 있다는 것을 참고하자.

그림 속 중간에는 무료를 인증서를 발급받을 수 있는 서비스가 있다.

학습정리



- 지금까지 ‘보안’에 대해 살펴보았습니다.

보안

보안에 대한 기초 개념을 알아보았습니다. 데이터의 암호화/복호화는 상황에 따라서 선택해서 사용합니다.

해시

단방향 암호화인 해시에 대해서 알아보았습니다.

사용자 인증 정보 암호화

해시를 이용해서 사용자 비밀번호를 암호화해봤습니다. 사전 대입 공격을 막기위해서 솔트를 적용해봤습니다.

- 지금까지 ‘보안’에 대해 살펴보았습니다.

암호화

데이터 암호화와 복호화가 가능한 암호화에 대해서 알아보았습니다. 대칭키와 비대칭키 암호화를 알아보고 관련 기능을 제공하는 모듈을 사용해봤습니다.

보안서버 만들기

SSL 인증서를 이용해서 HTTPS 서버를 작성해봤습니다.

T아카데미 온라인 강의

Node.js 프로그래밍

24. 디버깅과 프로세스 관리



CONTENTS

1

디버깅

2

테스트

3

프로세스 관리

학습 목표

1. 디버그 모드에서 앱의 실행 상황을 추적할 수 있습니다.
2. 자동화된 테스트를 작성할 수 있습니다.
3. 프로세스 관리 모듈로 서비스를 운영할 수 있습니다.

1. 디버깅



1. 디버깅

● 로그 메시지 출력

▶ 콘솔의 로그 함수들

- `console.log` ('일반 로그 출력');
- `console.info` ('정보성 로그 출력');
- `console.warn` ('경고용 로그 출력');
- `console.error` ('에러 로그 출력');

서비스를 개발하는 과정이나, 혹은 서비스를 운영하는 과정에서 동작 상황을 아는 것은 중요하다.

동작 상황을 아는 가장 간단한 방법은 콘솔에 정보를 출력하는 것이다.

전역 객체인 `console`을 사용하면 간단하게 정보를 출력할 수 있다
정보의 종류에 따라서 `log`, `info`, `warn`, `error` 등의 함수로 출력한다.

1. 디버깅

● 로그 외 다른 디버깅 방법

- 코드 단계 별 동작 확인
- 실행 멈춤 - 브레이크 포인트
- 특정 시점에서의 스택 상황
- 스코프 내 변수의 값 확인

콘솔 출력 외에도 실행 중 정보를 얻는 방법으로 디버거를 이용할 수 있다.

디버거를 사용하는 방법으로는, 소스 코드에 실행을 멈추게 하는 브레이크 포인트를 걸고 그 실행 상황의 값을 확인한다.

1. 디버깅

● 디버깅 모드로 동작시키기

- `node --debug`
- `node --debug-brk`

● 디버깅 모드로 동작 중

- `$ node --debug server.js`
- Debugger listening on port 5858

디버거를 사용하려면 디버깅 모드로 Node.js 애플리케이션을 실행시켜야 한다. node 명령 중 `--debug`(- 기호가 2개), `--debug-brk` 를 사용한다. `--debug-brk`는 디버거가 시작되면서 소스코드 처음 부분에 자동으로 브레이크 포인트가 걸린다.

그림 하단은 `server.js` 코드를 디버깅 모드로 동작시키는 명령이다.

1. 디버깅

● Node-Inspector

📌 Webkit 기반으로 Node.JS App 동작 디버깅

- <https://github.com/node-inspector/node-inspector>
- [sudo] npm install -g node-inspector

📌 디버깅 시작하기

- Node -debug[-brk] app.js
- Node-inspector 실행, url 복사
- 웹 브라우저로 url 열기

디버깅 모드로 동작 중인 애플리케이션의 실행 상태는 인스펙터를 이용해서 볼 수 있다.

Node-Inspector는 웹킷을 이용해서 브라우저에서 애플리케이션의 코드와 실행 상황을 볼 수 있게 하는 모듈이다.

npm을 이용해서 설치하자. 전역 모드로 설치하는 것을 권장한다.

디버깅 모드로 앱을 실행하고, 다른 콘솔을 열고 node-inspector 를 실행한다. node-inspector를 실행하면 url이 출력되고, 이 주소를 이용해서 웹 브라우저로 접속한다.

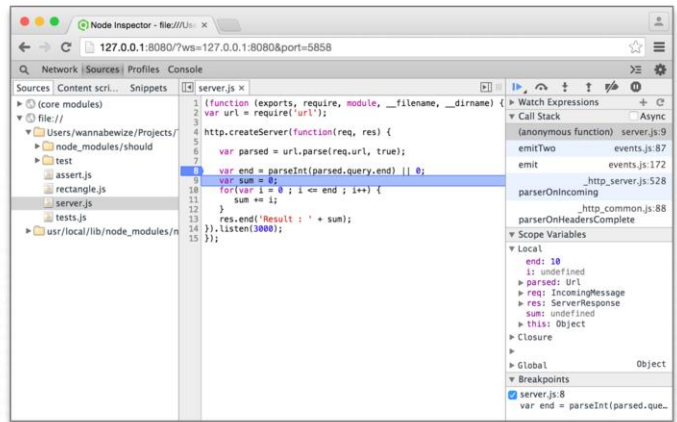
1. 디버깅

● 사용방법

- 디버그 모드로 Node.js 동작
- node-debug 명령
- 웹 브라우저로 동작 환경 확인

● 실행정보

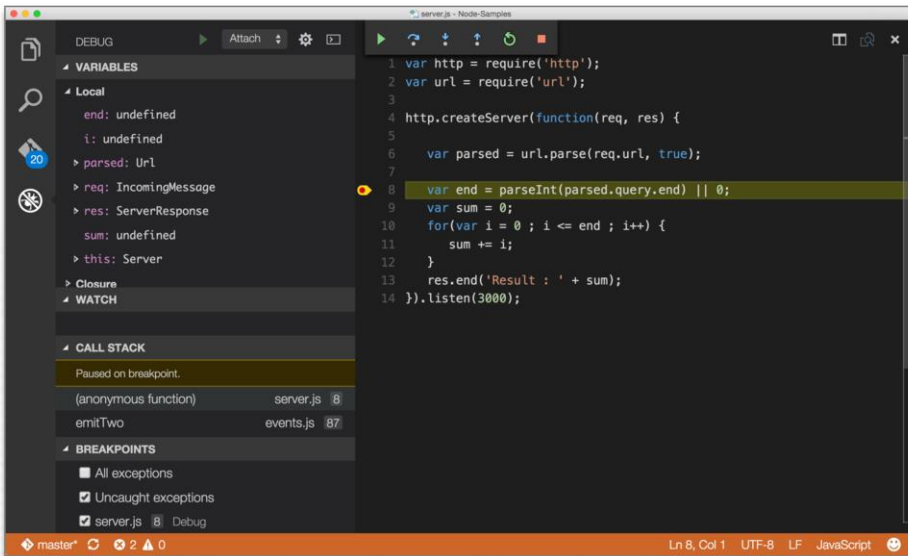
- CallStack
- Variables
- Breakpoints



그림은 웹 브라우저에서 디버깅 상황을 보는 모습이다.
 중앙에는 코드와 현재 동작중인 브레이크 포인트가 표시된다.
 오른쪽에는 콜 스택과 스코프 내 변수의 값들을 볼 수 있다.

1. 디버깅

개발툴의 디버깅 기능



개발툴에서도 디버거를 사용할 수 있다.

그림 속 코드는 Visual Studio Code에서 브레이크 포인트를 걸고 실행 상태를 확인하는 모습이다.

1. 디버깅

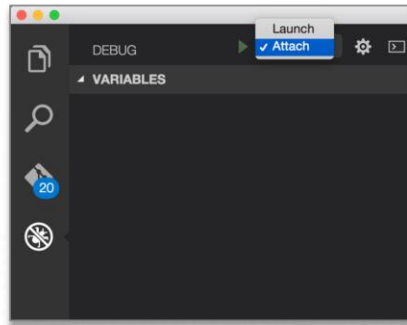
● 개발툴의 디버그 모드

▶ Launch

- 디버그 용 스크립트를 실행
- 디버그 모드로 앱 동작 불필요

▶ Attach

- 디버그 모드로 동작 중인 프로세스에 연결
- 디버그로 앱 시작 후 시작 버튼



Visual Studio Code에서는 왼쪽 수직 탭에서 4번째 버튼을 눌러서 디버그 모드로 툴을 변환할 수 있다.

json파일을 이용해서 동작시키는 Launch 모드와 디버깅 상태로 동작 중인 애플리케이션에 연결하는 Attach 모드가 있다.

툴 마다 디버거 사용하는 방법은 다르므로, 툴의 사용 설명서를 확인해야 한다.

1. 디버깅

- Launch 모드로 디버깅

- ▶ 디버깅 정보 : launch.json

- ▶ program 항목에 소스 코드 이름 작성

```
{
  "configurations": [
    {
      "name": "Launch",
      "program": "server.js",
      "stopOnEntry": false,
      "args": [],
      "env": {
        "NODE_ENV": "development"
      }
    }
  ]
}
```

그림은 Launch 모드에서 사용하는 json 파일이다.

program 프로퍼티에 디버깅 모드로 동작시킬 소스 파일의 이름을 입력한다.

2. 테스트



2. 테스트

● 테스트 코드 작성

- 코드로 테스트하기
- 테스트 자동화

● 테스트 모듈

- assert
- should
- mocha

애플리케이션의 품질을 높이기 위한 방법으로 테스트 코드를 작성하는 방법이 있다. 테스트에 관해서는 별도의 문서를 찾아보는 것을 추천한다. 테스트 코드를 작성하면 테스트 자동화가 가능하기 때문에 직접 손으로 테스트하는 것보다 빠르고 간단하게 테스트할 수 있다. 테스트용 모듈에는 assert, should, mocha 등이 있다.

2. 테스트

Assert 모듈 : assert

- 기본 모듈. 별도 설치 불필요
- `var assert = require('assert');`

테스트하기

- **참 테스트**
`assert.ok(value[, message])`
- **동등 테스트**
`assert.equal(actual, expected[, message])`
- **에러 발생**
`assert.throws(block[, error][, message])`

assert 모듈은 기본 모듈이다.

assert 모듈은 다양하게 값을 테스트할 수 있는 assert 함수를 제공한다.

assert 함수는 조건에 맞지 않으면 메시지를 출력하고 실행을 멈춘다.

2. 테스트

- Assert 로 테스트 작성하기
 - `var trueValue = true;`
 - `assert(trueValue);`
 - `assert.ok(trueValue, '참 판단');`

그림 속 코드는 assert를 이용해서 값을 테스트하는 코드다.

`assert()`나 `assert.ok()` 함수는 파라미터의 결과가 true이어야 한다.

값이 0인 경우에는 false로 취급되니, 값이 0이어야 하는 경우는 `assert(value == 0)`로 코드를 작성해야 한다.

2. 테스트

Assert 에러 → 테스트 통과 실패

- AssertionError 발생
- 실행 멈추고 실패 메시지 출력

```
var falseValue = false
assert(falseValue, 'False Value');
```

- 출력되는 메시지

```
throw new assert.AssertionError({
  ^
AssertionError: False Value
```

그림은 assert에서 조건이 거짓이라서 테스트 통과가 안되는 경우다.
그림 하단과 같이 에러가 발생하면서, 메시지가 출력된다.
에러가 발생한 위치에서 바로 실행이 멈추므로 디버깅하기가 쉬워진다.

2. 테스트

Equality Assertion

- `assert.equal(actual, expected[, message])` // 같은 객체 비교
- `assert.deepEqual(actual, expected[, message])` // 내용 비교
- `assert.strictEqual(actual, expected[, message])` // `===` 비교

테스트 작성

- `var intVal = 9;`
- `assert.equal(intVal, 9, 'equal 9');`
- `assert.equal(intVal, '9', 'equal "9"');`
- `assert.deepEqual(intVal, '9', 'deepEqual "9"');`
- `assert.strictEqual(intVal, '9', 'strictEqual "9");` // `AssertionFail`

`assert` 모듈에는 다양한 형태의 `assertion` 함수를 제공한다.

`assert.equal()` 을 이용한 비교는 같은 인스턴스인지 비교하는 용도다.

`deepEqual`의 경우에는 내용을 비교하는 방법으로 `==`에 해당한다.

`strictEqual`의 경우에는 객체의 타입까지 비교하는 방법으로 `===`에 해당한다.

그림 하단에는 숫자 9와 문자 '9'의 비교 코드가 있다.

`strictEqual`의 경우 `===` 비교이므로 테스트가 실패한다.

2. 테스트

- Assert 모듈 : should
 - <https://github.com/tj/should.js/>
 - npm install should
 - BDD 방식의 assert 작성

값을 판단하는 assert 모듈에는 should 라는 모듈도 있다.
shoule는 확장 모듈이므로 npm을 이용해서 설치해야 한다.
npm install should

should는 BDD(Bahavior-Driven Development)에 어울리는 테스트 모듈이다.

2. 테스트

● 테스트 모듈 : should

▶ should

```
var intVal = 5;
intVal.should.ASSERT
```

▶ assert 함수

```
.eq(otherValue) // ==
.equal(otherValue) // ===
.startsWith(str), .endsWith(str)
```

▶ 체인 방식

```
.be.ok
.be.type(str)
.have.properties(propName1, propName2, ...)
```

그림은 should 모듈을 사용하는 코드다.

should 모듈의 특징은 should를 이용해서 영어 문장을 만들듯이 테스트 코드를 작성할 수 있다.

should와 assert 함수는 다음과 같이 사용한다.

```
intValue.should.ASSERT
```

assert 종류의 함수에는 eq, equal, startWith 등이 있다.

should는 체인 방식으로 다양한 비교 함수를 연달아 작성할 수 있다.

2. 테스트

● 테스트 모듈 : should

▶ 값 비교

```
var intVal = 5;
intVal.should.equal(5);
var strVal = 'Hello';
strVal.should.equal('Hello');
```

▶ 값 비교 실패

```
intVal.should.equal(4);
```

▶ 에러 메시지

```
AssertionError: expected 5 to be 4
```

그림은 should를 사용해서 테스트 코드를 작성한 것이다.

다음은 intVal의 값이 5이어야 테스트를 통과하고, 그렇지 않으면 테스트에 실패하고 실행을 멈추면서 테스트 실패 메시지가 출력된다.

```
intVal.should.euqal(5)
```

그림 하단에는 테스트를 실패해서 발생하는 에러 메시지가 있다.

2. 테스트

● 테스트 모듈 : should

▶ 객체 동등 비교

.eq() : 콘텐츠 비교

.equal() : strict equal, === 비교

▶ 예제 코드

```
var obj = {  
  value : 10  
};  
obj.should.eq({value:10});  
obj.should.equal({value:10}); // Assert Fail, ===
```

should의 동등 비교는 eq(), equal() 함수를 사용한다.

2. 테스트

● 테스트 모듈 : should

▶ 체인 방식의 Assertion

```
var strVal = 'Hello';  
strVal.should.startWith('H').and.endWith('o');
```

그림 속 코드는 체인 방식으로 `startWith`와 `endWith`를 연결해서 동작하도록 작성했다.

2. 테스트

● 테스트용 프레임워크 : mocha

- <https://mochajs.org>

📦 설치

- [sudo] npm install -g mocha

📦 테스트

- 테스트 자동화와 리포팅
- TDD, BDD
- 다른 Assert 라이브러리와 결합 사용

모카(mocha)는 테스트 코드를 자동화해서 동작시키고 리포팅하는 모듈이다. 사이트 주소는 mochajs.org이고 npm을 이용해서 설치해야 한다. 전역 설치를 권장한다.

2. 테스트

● BDD 테스트 작성

▶ Behavior Specification Code

```
describe('Calculator', function() {  
  it('should add',function() {  
    // assertion  
  });  
  it('should minus', function() {  
    // assertion  
  });  
});
```

BDD 방식으로 작성한 테스트 코드다.

Calculator의 add와 minus 행위(Behavior)를 테스트하는 코드를 작성했다.

2. 테스트

● BDD 개별 테스트

▶ assert 코드

```
it('task spec1',function() {  
  assert.equal(value, expected);  
});
```

▶ should와 사용

```
it('task spec2',function() {  
  value.should.equal(expected);  
});
```

개별 테스트는 assert나 should 등의 테스트 모듈을 이용해서 작성한다.

```
it('spec', function() {  
  // assert, should를 이용한 테스트 코드 작성  
}).
```


2. 테스트

● BDD 개별 테스트

▶ 비동기 행위 테스트 코드

```
it('async task spec3',function(done) {  
  asyncApi(value, function() {  
    value.should.equal(expected);  
    done();  
  });  
});
```

비동기 함수를 사용하는 테스트 코드를 작성하려면 콜백 함수를 호출해서 테스트가 끝난다.

```
it('spec', function(done) {  
  done() // 테스트 종료  
});
```

2. 테스트

● 테스트 프레임워크 mocha

▶ 후크(hook)

- 모든 테스트 시작 전, 테스트 종료 후
`before(function() {});`
`after(function() {});`
- 개별 테스트 시작 전, 개별 테스트 종료 후
`beforeEach(function() {});`
`afterEach(function() {});`

테스트가 시작하기 전이나 혹은 테스트가 끝난 후에 자동으로 동작해야 하는 코드를 작성할 수 있다.

`before()`는 `describe` 내부의 테스트가 동작하기 전에 1번 동작하고 `after`는 모든 테스트가 끝나고 동작한다.

`beforeEach()`와 `afterEach()`는 개별 테스트 마다 동작한다.

2. 테스트

● BDD 인터페이스

```
describe('Calculator', function () {  
  var calculator;  
  
  before(function () {  
    calculator = new Calculator();  
  });  
  
  after(function() {});  
  beforeEach(function() {});  
  afterEach(function() {});  
  
  // tests  
  it('should add two value', function () {}  
});
```

그림은 Calculator의 행위를 테스트하는 코드다.

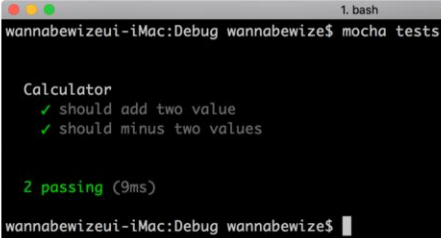
before 에서 테스트에서 사용할 Calculator 객체를 생성했다.

개별 테스트에서는 it함수를 이용해서 테스트 코드를 작성했다.

2. 테스트

● 테스트 동작

▪ \$ mocha test.js

A terminal window titled '1. bash' showing the execution of 'mocha tests'. The output displays 'Calculator' with two green checkmarks for 'should add two value' and 'should minus two values', followed by '2 passing (9ms)' and a new command prompt.

```
wannabewizeui-iMac:Debug wannabewize$ mocha tests

Calculator
  ✓ should add two value
  ✓ should minus two values

2 passing (9ms)

wannabewizeui-iMac:Debug wannabewize$
```

테스트 코드는 mocha를 이용해서 동작시킨다.
다음은 test.js에 작성한 테스트를 동작시키는 명령이다.

mocha test.js

테스트가 끝나면 그림 속 하단처럼 테스트 성공 여부가 출력된다.

2. 테스트

● 테스트 동작

- 테스트 코드 폴더 → /test
- test 폴더 내 모든 테스트 동작시키기 → \$mocha

서비스 작성 테스트 코드는 여러개의 파일로 작성하게 된다.

서비스를 위한 제품용 코드와 테스트용 코드를 서로 분리하는 것이 좋다.

테스트용 코드는 /test 폴더에 작성한다.

콘솔에서 mocha 명령만 입력하면, 자동으로 test 폴더에 있는 모든 테스트 코드를 동작시킨다.

2. 테스트

● TDD 기반으로 작성하기

▶ suite, test

```
suite('SUITE NAME', function(){  
  test('UNIT TEST', function() {  
    assert.equal(...)  
  });  
})
```

▶ hook

- suiteSetup, setup
- suiteTeardown, teardown

▶ TDD 기반 테스트 동작

- \$mocha -u tdd TESTCODE.js

mocha는 TDD 기반으로 테스트를 작성할 수 있다.

TDD 기반의 테스트 코드는 테스트 수트(suite)내부에 test() 함수로 작성한다.

suite내부에도 모든 테스트 전후로 동작하는 suiteSetup과 suiteTearDown 이 있고, 개별 테스트 전후에 동작하는 setup()과 teardown()을 작성할 수 있다.

TDD 기반으로 작성한 테스트를 동작시키려면 -u TDD 옵션을 주고 실행시킨다.

```
mocha -u tdd test.js
```

2. 테스트

● TDD 인터페이스

```
suite('Calaulator', function() {  
  var calculator;  
  
  suiteSetup(function() {  
    calculator = new Calculator();  
  });  
  
  suiteTeardown(function() {});  
  setup(function() {});  
  teardown(function() {});  
  
  test('Add', function() {  
    var value = calculator.add(1, 2);  
    assert.equal(value, 3, '1 + 2 = 3');  
  });  
  
  test('Minus', function() {  
  });  
});
```

그림 속 코드는 Calculator를 TDD 기반으로 작성한 코드다.
suite() 함수 내에 suiteSetup() 함수에서 Calculator 객체를 생성했다.
Suite 내부의 개별 테스트는 test() 함수로 작성했다.

3. 프로세스 관리



3. 프로세스 관리

● 콘솔로 서비스 실행

- 콘솔 종료하면 애플리케이션 종료

📦 forever

- 콘솔 종료와 관계 없이 애플리케이션 계속 실행
- 크래쉬 → 자동 재실행

📦 forever 설치(-g)

- `sudo npm install forever -g`

Node.js 애플리케이션을 동작시키는 방법으로 콘솔 명령 `node`를 사용했다.
그런데 콘솔을 종료하면 Node.js 애플리케이션도 함께 종료되는 문제가 있다.

`forever`라는 모듈은 콘솔 종료와 관계없이 애플리케이션을 실행시키는 모듈로, 앱이 크래쉬돼도 자동으로 재시작시켜주는 기능도 제공한다.
`npm`을 이용해서 설치하고, 전역 설치를 권장한다.

3. 프로세스 관리

● forever : Node.js 애플리케이션 계속 실행하기

▶ 명령

- `forever start server.js`
- `forever stop [UID]`
- `forever stopall`
- `forever list`

forever 모듈의 명령은 forever다.
forever의 옵션은 다음과 같다.

start : 애플리케이션 시작

stop : 개별 애플리케이션 종료. UID가 필요하다.

stopall : forever로 동작중인 모든 애플리케이션 종료

list : forever로 동작중인 애플리케이션 목록 보기

3. 프로세스 관리

● 동작중인 프로세스 목록

▶ forever list

```
$ forever list
```

```
info: Forever processes running
```

uid	command	script	rever pid	id	logfile	uptime
yISi	/node	server.js	13533	13534	/yISi.log	0:0:0:31.183
wP6v	/node	fileUplaod.js	13540	13541	/wP6v.log	0:0:0:25.371
OLwA	/node	tcpEchoServer.js	13548	13549	/OLwA.log	0:0:0:2.409

그림은 list 옵션을 이용해서 forever로 동작중인 애플리케이션 목록이다.
uid 항목의 값을 이용해서 개별 애플리케이션을 종료할 수 있다.

학습정리



- 지금까지 ‘디버깅과 프로세스 관리’에 대해 살펴보았습니다.

디버깅

앱의 실행 환경을 분석할 수 있는 디버그 모드와 인스펙터를 사용해봤습니다. 개발툴에서도 바로 디버그 모드로 전환해서 실행 환경을 확인할 수 있습니다.

테스트

자동화된 테스트를 작성하기 위해서 Assert와 테스트 모듈을 사용해봤습니다. 행위 기반의 테스트(BDD) 방식으로 테스트 코드를 작성해봤습니다.

프로세스 관리

콘솔의 세션 종료와 관계없이 서비스를 동작시키기 위한 방법을 알아봤습니다.