

강의 소개
TDD(Test Driven Development)를 이용한 개발

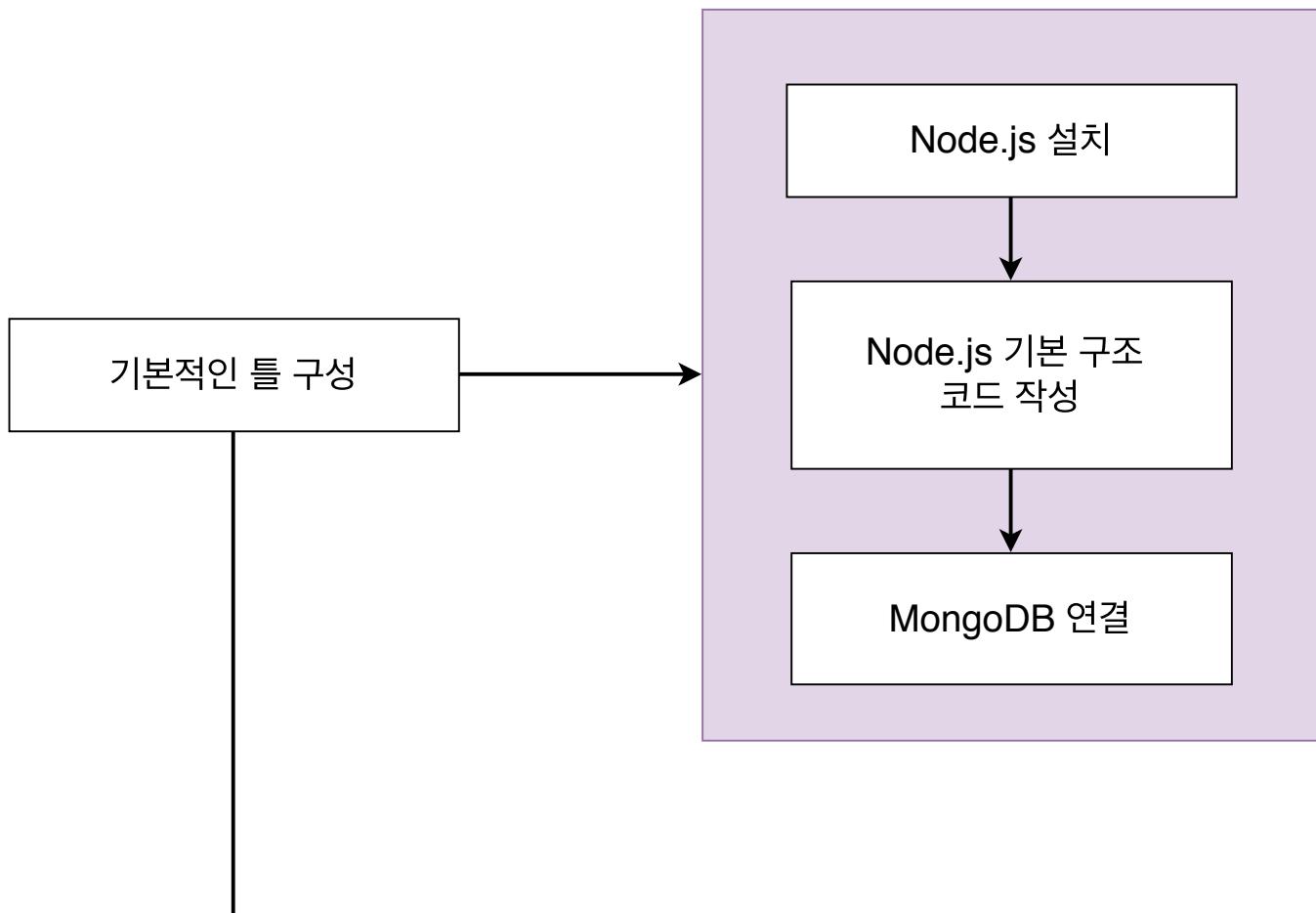
왜 어플리케이션을 TEST 해야 할까요?

간단하게 더 안정적인 어플리케이션을 위해서는 여러 방법으로 테스트를 해줘야 더 안정적인 어플리케이션이 될 수 있습니다.

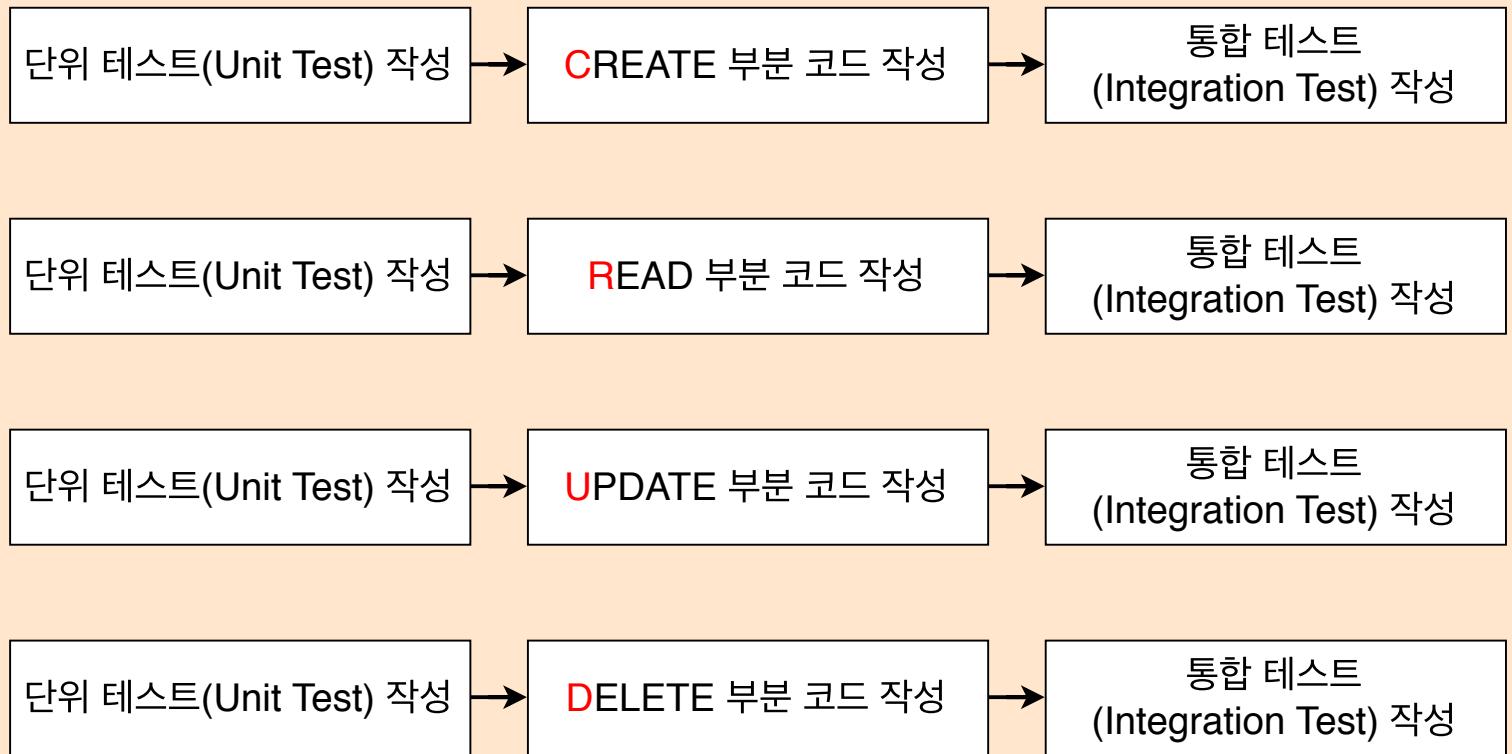
테스팅으로 얻는 이점은 무엇일까요?

1. 디버깅 시간을 단축! 만약 데이터가 잘못 나왔다면 그것이 UI의 문제인지 DB의 문제인지 등 전부 테스트를 해봐서 찾아야 하는데 테스팅 환경이 구축되어있다면 자동화 된 유닛 테스팅으로 특정 버그를 쉽게 찾아 낼 수 있습니다.
2. 더욱 안정적인 어플리케이션! 많은 테스트 코드와 함께 작성된 코드의 어플리케이션이 되기 때문에 훨씬 안정적인 어플리케이션이 됩니다.
3. 이밖에도 재설계 시간의 단축, 추가로 무언가를 더 구현해야 할 때 더 용이하게 할 수 있는 등의 이점들이 있습니다.

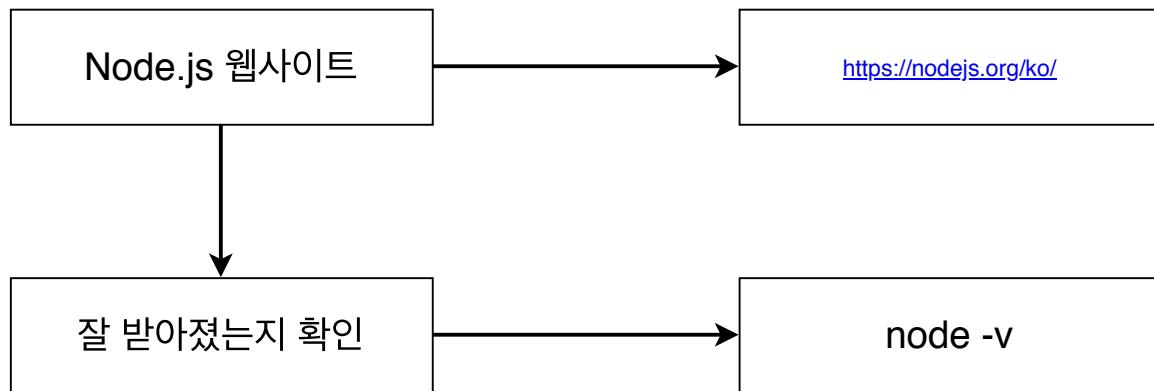
강의 순서



TDD 앱 개발



Node.js 설치



Node.js 기본 구조 코드 작성(package.json)

Package.json 파일

프로젝트의 정보와
프로젝트에서
사용 중인 패키지의 의존성을
관리하는 곳

npm init

사용할 모듈들 추가

Express

Node.js의 API를 단순화하고
유용한 기능들은 더 추가 시켜
Node.js를 더 편리하고 유용하
게 사용할 수 있게 해주는 모듈

Mongoose

MongoDB를 편리하게 사용하
게 해주는 노드의 확장 모듈
(뒤에서 더 자세히 설명)

Jest

단위 테스트를 위한 모듈

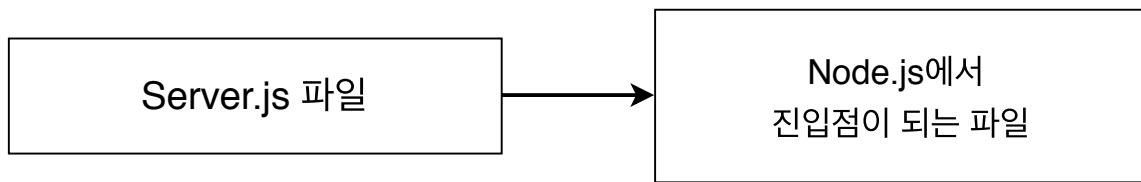
node-mocks-http

단위 테스트를 위한 모듈

supertest

통합 테스트를 위한 모듈

Node.js 기본 구조 코드 작성(server.js)



server.js(시작점) 만들기

```
const express = require('express');

// Constants
const PORT = 8080;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World');
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

→ Express 모듈 불러오기

→ Express 서버를 위한 포트 설정

→ 호스트 지정

→ 새로운 Express 어플 생성

→ "/" 이 경로로 요청이 오면
Hello World를 결과값으로 전달

→ 해당 포트와 호스트에서
HTTP 서버를 시작

This diagram illustrates the logic of the server.js code. It highlights specific lines of code with yellow boxes and maps them to their corresponding functions or configurations:

- `const express = require('express');` → Express 모듈 불러오기
- `const PORT = 8080;` → Express 서버를 위한 포트 설정
- `const HOST = '0.0.0.0';` → 호스트 지정
- `const app = express();` → 새로운 Express 어플 생성
- `app.get('/', (req, res) => {
 res.send('Hello World');});` → "/" 이 경로로 요청이 오면
Hello World를 결과값으로 전달
- `app.listen(PORT, HOST);` → 해당 포트와 호스트에서
HTTP 서버를 시작
- `console.log(`Running on http://${HOST}:${PORT}`);`

```
express.json()
```

Post 요청으로 들어오는 Body 부분 받기

```
//front-end
axios.post('/products', {
  name: 'phone', description: 'it is new'
})

//back-end
const express = require('express')
const app = express()
app.post('/products', (req, res) => {
  console.log('req.body : ' req.body)
})
```

undefined

undefined가 나오는 것에 해결 방안은 ...?

bodyParser모듈을 이용해서 해결해줄수있습니다.

하지만 express 버전 4.16.0 부터는
express에 들어 있는 내장 미들웨어 함수로
bodyParser 모듈을 대체해줄수 있습니다.

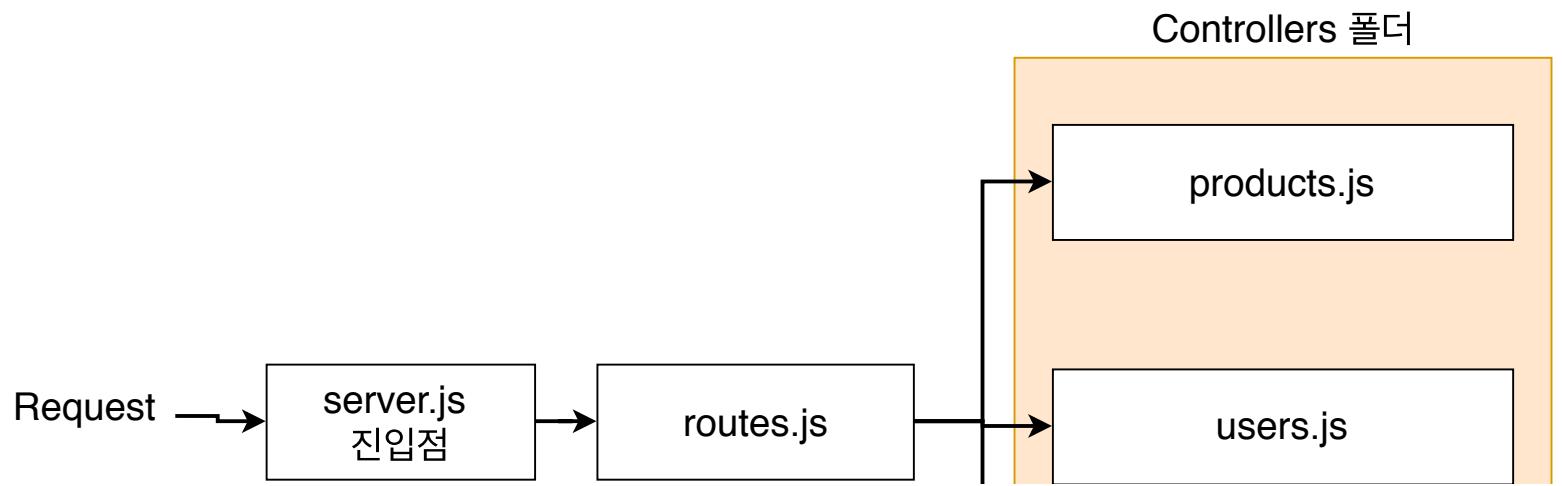
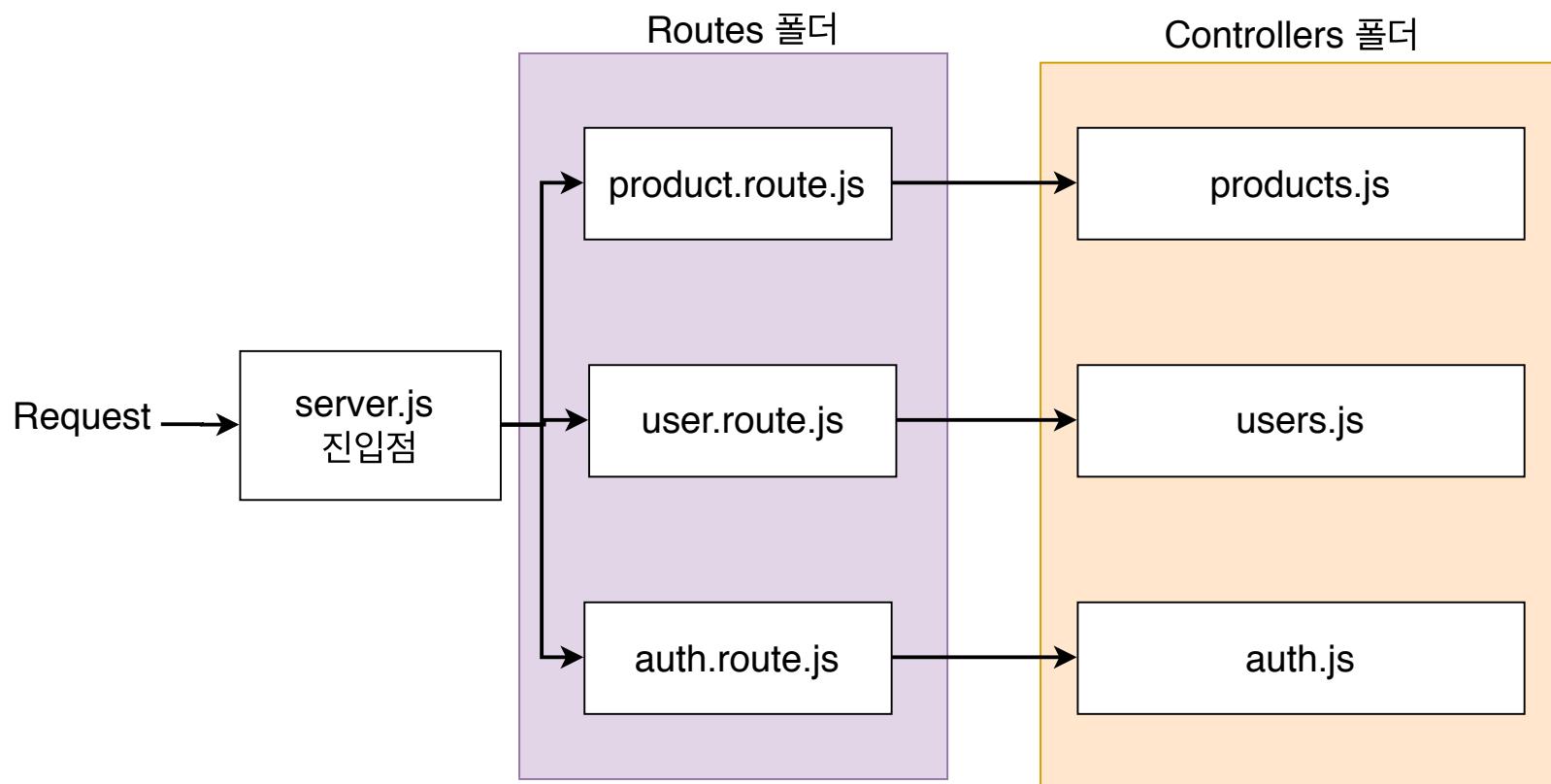
```
//front-end
axios.get('/products',
  { name: 'phone', description: 'it is new' })
```

```
//back-end
const express = require("express");
const app = express();
app.use(express.json())

app.post('/products', (req, res) => {
  console.log('req.body :', req.body)
})
```

Route, Controller

백단에서 처리하는 부분을 **server.js** 파일에서 모두 처리해줄 수 있지만
그렇게 되면 너무 정리가 안 되고 지저분해지기 때문에
Route와 Controller까지 나눠서 더 깔끔하게 해주겠습니다.





Route 구현 방법

1. express.Router 클래스를 이용해서 Router를 모듈로 작성하기

```
var router = express.Router();
```

2. Router에서 미들웨어 함수를 Load 하기.

```
router.get('/', function(req, res) {
  res.send('Birds home page');
});
```

3. 몇몇 Route를 정의하기

4. 기본앱의 한 경로에 라우터 모듈을 마운트하기

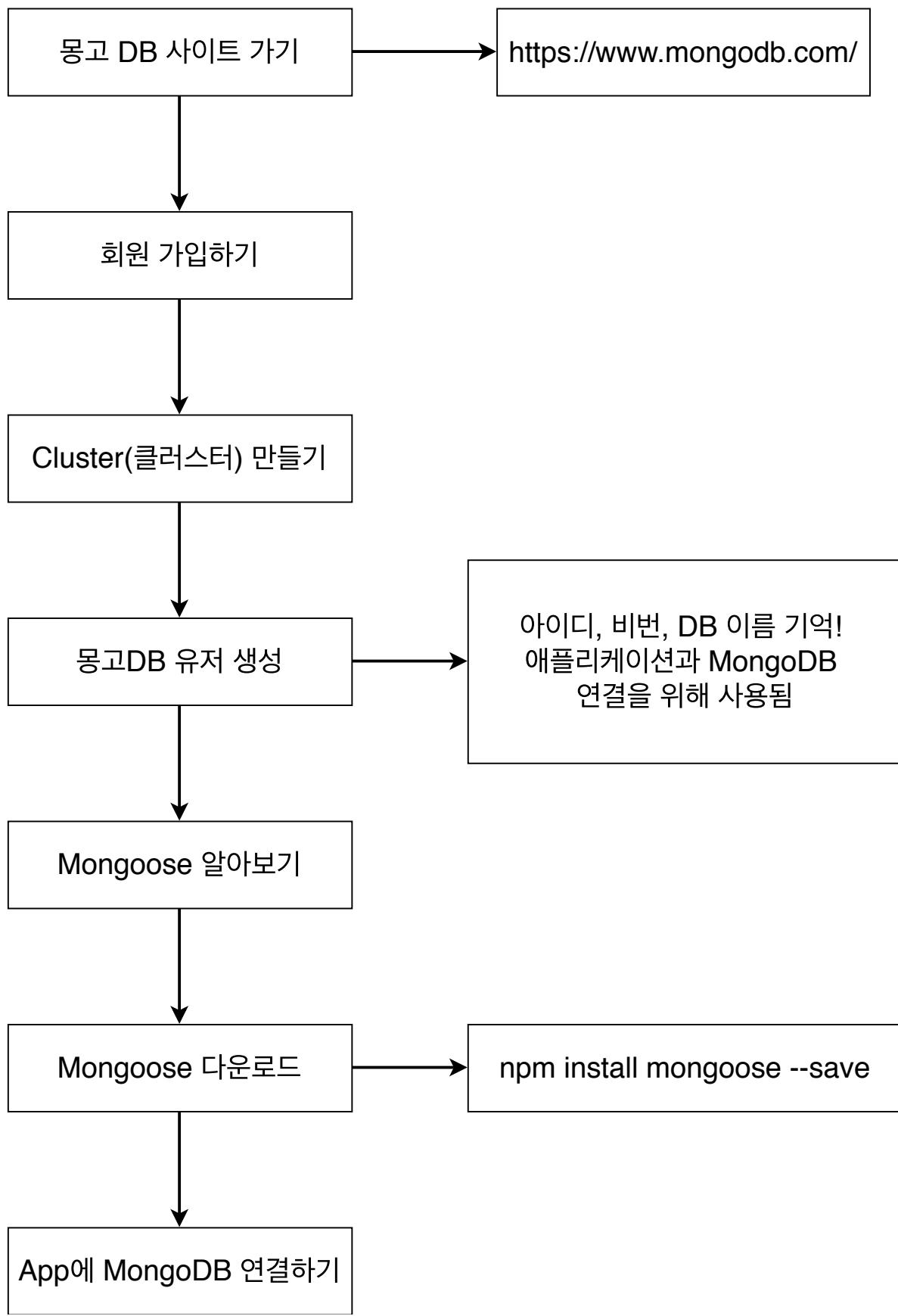
```
var birds = require('./birds');
...
app.use('/birds', birds);
```

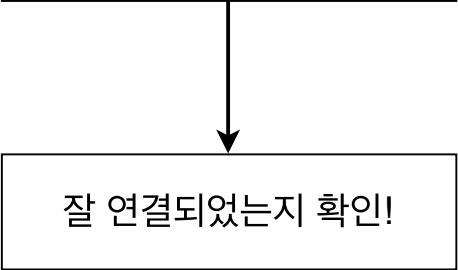
Controller 구현 방법

1. Router에 구현한 미들웨어 함수 부분을 떼어서 Controller 파일에 옮겨 줍니다.

2. Controller에 넘어간 미들웨어 함수를 Export하고 Route에서 import 해 줍니다.

MongoDB 클러스터 생성





잘 연결되었는지 확인!

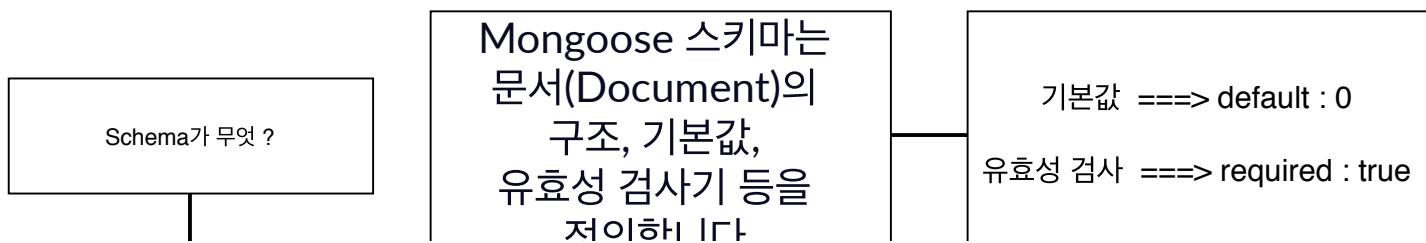
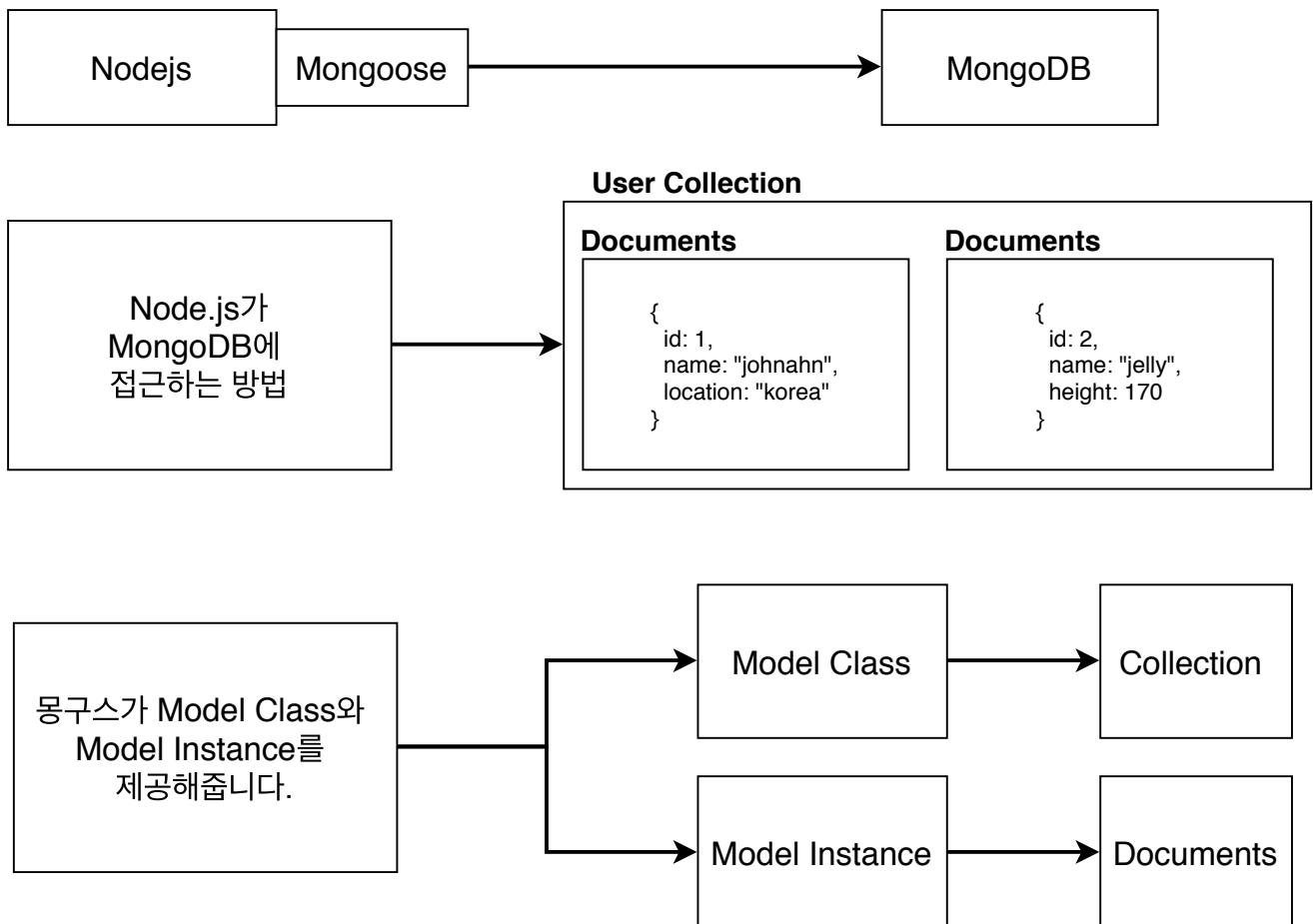
MongoDB Model & Schema

몽구스(mongoose)는 무엇인가요 ?

몽고 DB 사용을 위한 다양한 기능을 추가하고 몽고 DB를 더 편리하게 이용하기 위해서 사용하는 모듈 !

몽구스를 이용해서 데이터를 만들고 관리하기 위해서 먼저 Schema를 만들고 그 스키마로 모델을 만듭니다.

몽구스는 몽고DB를 쓸 때 사용해도 되고 안 써도 되는 선택사항입니다.



스키마를 만들고
그 스키마로 모델을 만듭니다

Model 이 무엇 ?

Mongoose 모델은 레코드
생성, 쿼리, 업데이트, 삭제
등을 위한 데이터베이스 인
터페이스를 제공합니다.

```
const mongoose = ...;

const productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  description: {
    type: String,
    required: true
  },
  price: {
    type: Number
  }
});

const Product = mongoose.model("Product", productSchema);

module.exports = Product;
```

constructor 함수를
몽구스에서 제공해서
새로운 스키마를
정의할수있게 해줍니다.

단위 테스트(Unit Test)에 대하여

단위(Unit) 테스트란 ?

단위 테스트는 개발자가 수행하고 자신이 개발 한 코드 단위 (일명 모듈, 구성 요소)를 테스트합니다. 소스 코드의 **개별 단위**를 테스트하여 사용할 준비가 되었는지 확인하는 테스트 방법입니다. 개발 라이프 사이클의 초기 단계에서 버그가 식별되므로 버그 수정 비용을 줄이는 데 도움이됩니다. 간단하게 생각하면 메소드를 테스트하는 또 다른 메소드라고 생각하시면 됩니다.

단위(Unit) 테스트의 조건

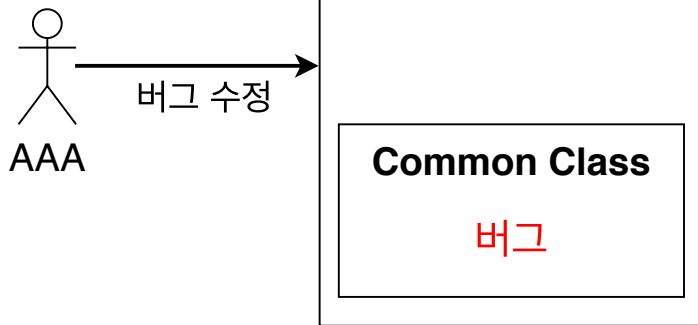
1. 독립적이어야 하며, 어떤 테스트도 다른 테스트에 의존하지 않아야 합니다.
2. 격리 되어야 합니다. Ajax, Axios, LocalStorage등 테스트 대상이 의존하는 것을 다른 것으로 대체해야 합니다.

왜 단위(Unit) 테스트를 하나요 ?

1. 첫번째는 프로그램이 크고, 메모리가 많이 들고, 다른 리소스(데이터베이스 등)이 필요한 경우 로컬 환경에서 쉽게 코드를 실행시켜보기 어렵기 때문이다. 이런 프로그램을 개발하는 개발자들은 유닛테스트를 만들어서 빠르게 자신의 코드가 정상적으로 작동 하는지 확인 할 수 있다.
2. 종속성이 있는 다른 클래스들에서 버그가 나는것을 방지하기 위해서 입니다.

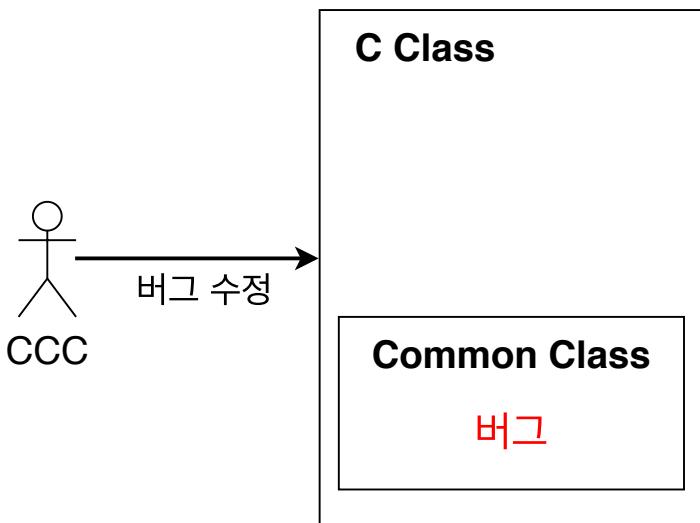
A 에서 버그가 나는데 알고 보니 그것이 Common Class 에서 나오는 에러 였음

A Class



그래서 Common Class 수정, A Class 버그 사라짐

하지만 Common Class는 C Class 에서도 사용중



위에서 Common Class를 수정 해서 이제는 C Class에서 버그가 생겨버림...

그래서 CCC가 C Class에서 생기는 버그를 잡기 위해서
Common Class를 다시 수정

이렇게 되면 A Class 에서 다시 에러가 생길 가능성 높다...

이러한 상황에서 Unit 테스트를 한다면 ...
Common Class에 의존하는 다른 클래스들도 확인 가능하기에
이 에러를 잡기 더욱 쉬워진다...

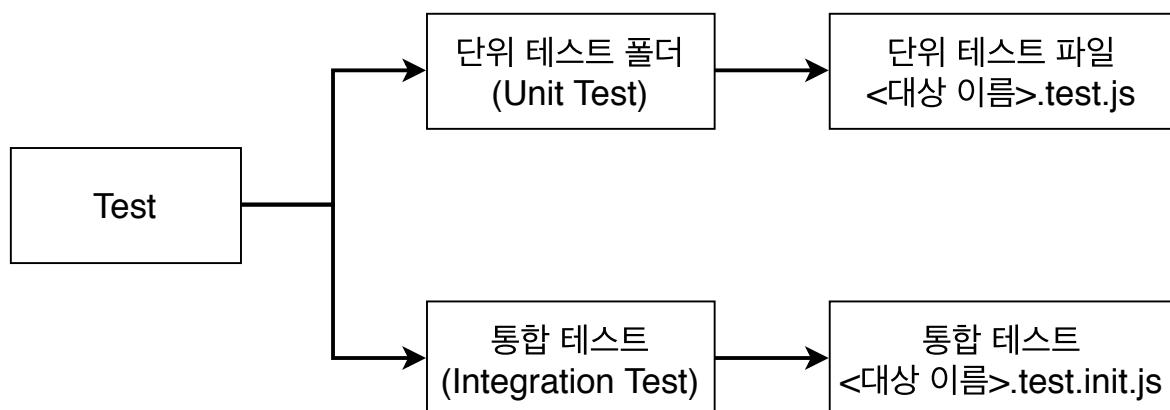
Jest에 대하여

Jest란 무엇인가요?

FaceBook에 의해서 만들어진 테스팅 프레임 워크입니다.
최소한의 설정으로 동작하며 Test Case 를 만들어서 어플리케이션 코드가
잘 돌아가는지 확인해줍니다.
단위 (Unit) 테스트를 위해서 이용합니다.

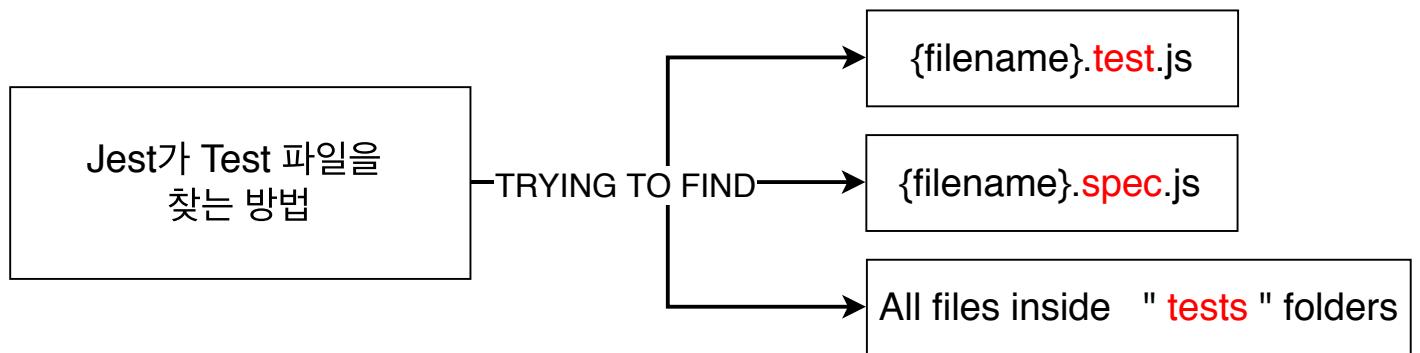
Jest 시작하기

1. Jest 라이브러리 설치 `npm install jest --save-dev`
2. Test 스크립트 변경 `"test" : "jest"` or `"jest --watchAll"`
3. 테스트 작성할 폴더 및 파일 기본 구조 생성



```
✓ test
  ✓ integration
    JS products.int.test.js
  ✓ unit
```

JS products.test.js



Jest 파일 구조 & 사용법

Jest 파일 구조

```
describe("Product Controller create", () => {
  beforeEach(() => {
    req.body = newProduct;
  });
  it("should have a createProduct function", () => {
    expect(typeof productController.createProduct).toBe("function");
  });
  it("should call Product.create", async () => {
    await productController.createProduct(req, res, next);
    expect(Product.create).toBeCalledWith(newProduct);
  });
  it("should return 201 response code", async () => {
    await productController.createProduct(req, res, next);
    expect(res.statusCode).toBe(201);
    expect(res._isEndCalled()).toBeTruthy();
  });
  it("should return json body in response", async () => {
    Product.create.mockReturnValue(newProduct);
    await productController.createProduct(req, res, next);
    expect(res._getJSONData()).toStrictEqual(newProduct);
  });
  it("should handle errors", async () => {
    const errorMessage = { message: "Done property missing" };
    const rejectedPromise = Promise.reject(errorMessage);
    Product.create.mockReturnValue(rejectedPromise);
    await productController.createProduct(req, res, next);
    expect(next).toBeCalledWith(errorMessage);
  });
});
```

describe

test (it)

test (it)

test (it)

Explain

"describe"
argument (name, fn)

여러 관련 테스트를
그룹화하는 블록을 만듭니다.

"it" same as **test**
argument (name, fn, timeout)

개별 테스트를 수행하는 곳.
각 테스트를 작은 문장처럼
설명합니다.

Describe (과일)

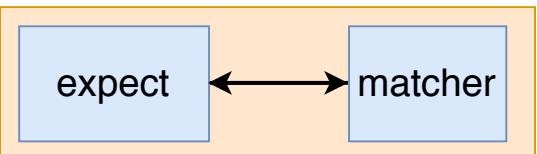
it 사과

it 바나나

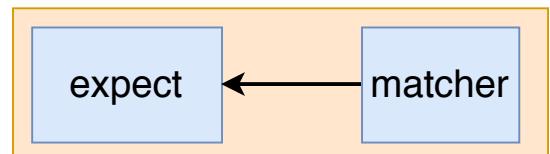
```
describe("Product Controller create", () => {
  beforeEach(() => {
    req.body = newProduct;
  });
  it("should have a createProduct function", () => {
    expect(typeof productController.createProduct).toBe("function");
  });
  it("should call Product.create", async () => {
    await productController.createProduct(req, res, next);
    expect(Product.create).toBeCalledWith(newProduct);
  });
  it("should return 201 response code", async () => {
    await productController.createProduct(req, res, next);
    expect(res.statusCode).toBe(201);
    expect(res._isEndCalled()).toBeTruthy();
  });
  it("should return json body in response", async () => {
    Product.create.mockReturnValue(newProduct);
    await productController.createProduct(req, res, next);
    expect(res._getJSONData()).toStrictEqual(newProduct);
  });
  it("should handle errors", async () => {
    const errorMessage = { message: "Done property missing" };
    const rejectedPromise = Promise.reject(errorMessage);
    Product.create.mockReturnValue(rejectedPromise);
    await productController.createProduct(req, res, next);
    expect(next).toBeCalledWith(errorMessage);
  });
});
```

describe

test (it)



test (it)



Explain

"expect"

expect 함수는 값을 테스트할 때마다 사용됩니다.
그리고 expect 함수 혼자서는 거의 사용 되지 않으며 matcher와 함께 사용됩니다.

"matcher"

다른 방법으로 값을 테스트 하도록 "매처"를 사용합니다.

```
test('two plus two is four', () => {
  expect(2 + 2).toBe(4);
});
```

matcher

PASS test/unit/**products.test.js**
✓ two plus two is four (2 ms)

Test Suites: 1 **passed**, 1 total
Tests: 1 **passed**, 1 total
Snapshots: 0 total
Time: 0.279 s, estimated 1 s
Ran all test suites.

Watch Usage: Press w to show more. █

```
test('two plus two is not five', () => {
  expect(2 + 2).not.toBe(5);
});
```

matcher

PASS test/unit/**products.test.js**

✓ two plus two is four (2 ms)
✓ two plus two is not five (1 ms)

Test Suites: 1 **passed**, 1 total
Tests: 2 **passed**, 2 total
Snapshots: 0 total
Time: 0.281 s, estimated 1 s
Ran all test suites.

Watch Usage: Press w to show more. █

jest.fn()

jest.fn()이란 ...?

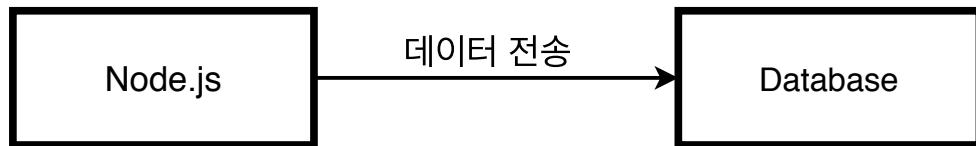
Mock 함수를 생성하는 함수입니다. mock을 한글 번역을 보면 모의 고사 할 때 그 모의 라는 뜻이 있으며 가짜, 흉내내는 이러한 뜻이 있습니다. 그래서 이 Mock 함수가 하는 일은 단위 테스트를 작성할 때, 해당 코드가 의존하는 부분을 가짜로 대체하는 일을 해줍니다.

단위 테스트는 왜 독립적이어야 하나요 ...?

의존적인 부분을 구현하기가 까다로울 경우가 있으며, 의존적인 부분의 상태에 따라서 테스트하고자 하는 부분의 테스트 결과가 영향을 받을 수 있기 때문입니다.

예를 들어서...

데이터베이스에 데이터를 저장하는 부분을 테스트를 한다면 ...



데이터 전송하는 가운데....

Network, I/O 작업, 트랙잭션 생성, 쿼리 전송 등등.....

그리고 테스트 종료 후,

데이터베이스에서 변경 데이터를 직접 원복하거나 트랙잭션 rollback 해줘야 하는 ...

그저 데이터베이스에 데이터를 저장하는 부분 테스트를 위해서

너무나 많은 작업이 들어가기 때문에 비효율적일 수 있습니다.

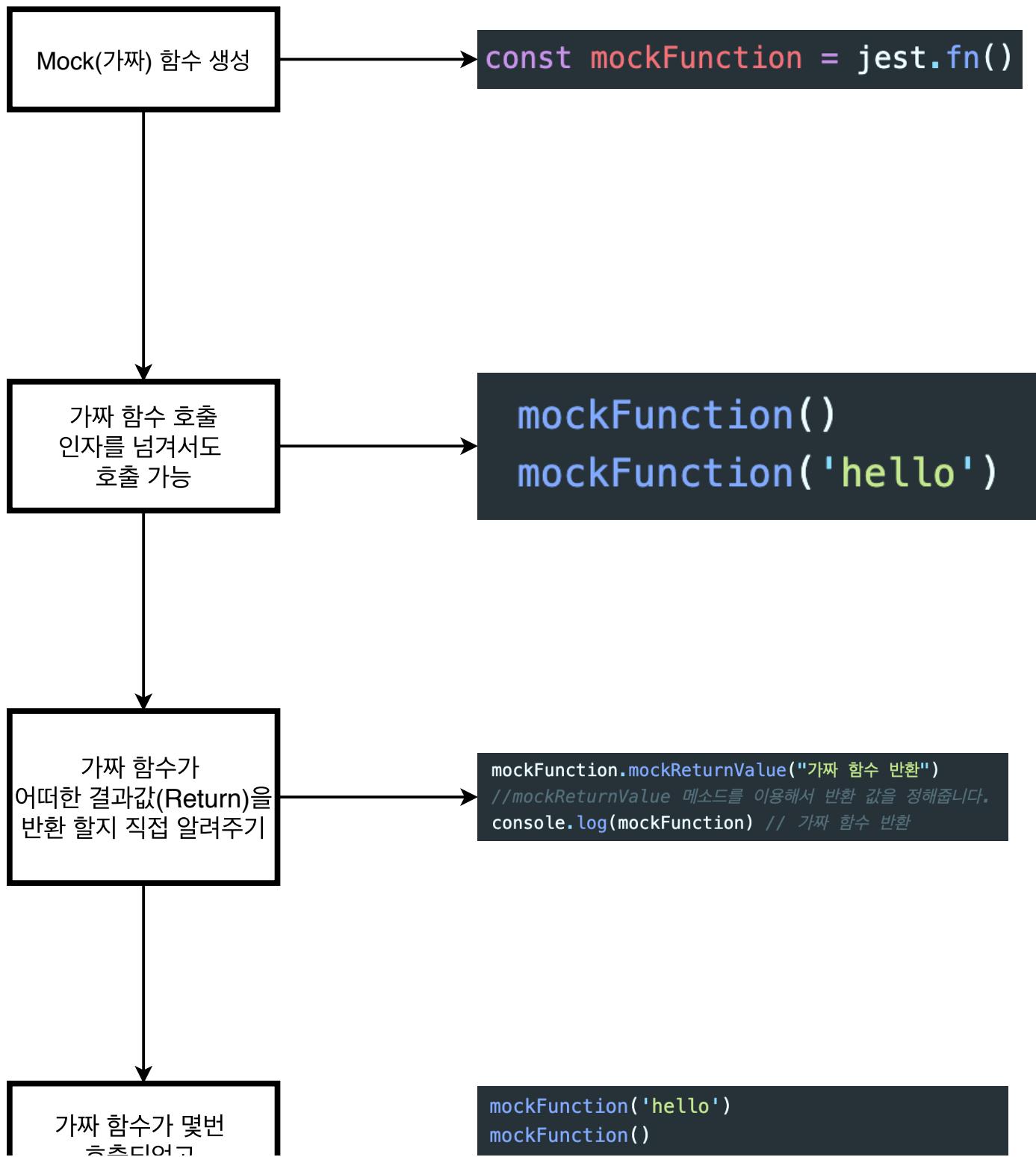
그리고 테스트 하는 가운데 데이터베이스가 잠시 죽어 있다면
테스트 결과에도 영향을 미치게 됩니다..

그네시 근위 테스트는 즉ち 기능은 준비에서 즉급으로 시동..

`jest.fn()` 를 이용해서 가짜 함수를 생성 하므로 인해서 의존적인 부분으로 인해 영향을 받는 테스트 상황을 해결할 수 있습니다.

이 `jest.fn()`이 생성한 가짜 함수는 이 함수에 어떤 일들이 발생했는지, 다른 코드들에 의해서 어떻게 호출되는지를 기억하기 때문에 이 함수가 내부적으로 어떻게 사용되는지 검증할 수도 있습니다. (Spy 역할)

`jest.fn()` 기본 사용법



호출되었고
어떤 인자가
넘어왔는지 검증

```
expect(mockFunction).toBeCalledWith('hello')
expect(mockFunction).toBeCalledTimes(2)
```