Unit 3 13-15

데이터베이스 연결

UT-NodeJS / 04.14.2023

ut-nodejs.github.io

Contents / 내용

설치와 사용하는 방법

13. MongoDB 셋업

- MongoDB 설치
- MongoDB 셸에서의 데이터 읽기와 쓰기
- Node.js 애플리케이션의 MongoDB 접속

14. Mongoose를 사용한 모델 제작

- Mongoose의 설치와 Node.js 애플리케이션으로 연결
- 스키마 생성
- Mongoose 데이터 모델의 생성과 초기화
- 사용자 정의 메소드로 데이터 읽기와 저장하기

15. 컨트롤러와 모델과의 연결

- 컨트롤러와 모델과의 연결
- 컨트롤러 액션을 통한 데이터 저장
- 프라미스로 데이터베이스 쿼리 실행
- 폼 데이터 포스팅의 처리

13

MongoDB 셋업

MongoDB 설치
MongoDB 셸에서의 데이터 읽기와 쓰기
Node.js 애플리케이션의 MongoDB 접속

p. 189-202

13.1 MongoDB 세팅

MongoDB를 이해

MVC 모델을 사용해 애플리케이션을 구조화하고 컨트롤러와 뷰를 제공함으로 요청을 처리할 수 있게 됐다. 이 모델에서 세 번째 핵심 조각은 영속적으로 저장하고 사용할 데이터다.

데이터 저장은 애플리케이션 개발에서 중요하다.

데이터베이스란 사용자가 데이터에 접근하거나 애플리케이션이 데이터를 변경할 때 이를 쉽게 해주는 기구다. 데이터베이스는 창고와 비슷하다. 저장해야 할 아이템이 많을수록 이를 저장하고 찾을 수 있는 시스템을 통해 수월함을 느낄 것이다. 웹 서버처럼 이 애플리케이션도 몽고DB 데이터베이스로 접속하고 데이터를 요청한다.

몽고DB는 오픈소스 데이터베이스로서 도큐먼트를 사용해 데이터를 정렬하는 것이 특징이다. 몽고DB 도큐먼트는 데이터를 JSON 형태의 구조로 저장하며 특성별 데이터 객체의 **키-값 조합**을 사용할 수 있게 한다.

이 저장 시스템은 JavaScript 문법과 비 슷하다. 사실 몽고DB는 도큐먼트를 **BSON**Binary JSON 형태로 저장한다.

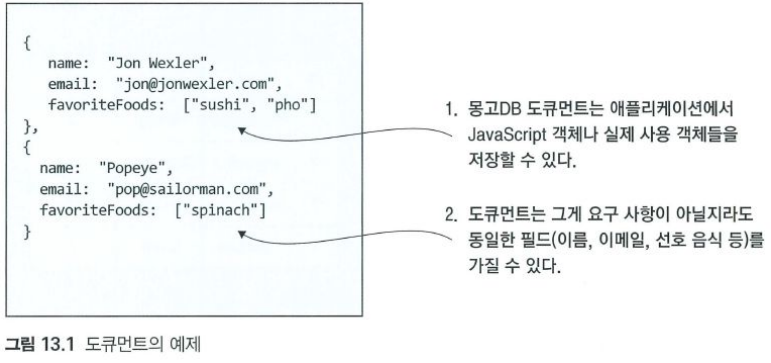


그림 13.1 도큐먼트의 예제

13.1 MongoDB 세팅

관계형 데이터베이스 살펴보기

People 테이블

id	first	last	enroll
1	Jon	Wexler	true
2	William	Wonka	true
3	Cookie	Monster	true
4	Alfredo	Linguini	true

Course 테이블

id	title	topic	max
1	Wheaties	Bread	3
2	Sweeties	Chocolate	15
3	Tortellinies	Pasta	10
4	Meaties	Steak	7

People-Course 연동 테이블

id	personId	courseId
1	4	3
2	2	2
3	1	1
4	3	1

이 예제에서 가운데 있는 테이블은 연동된 people과 course의 ID들을 나타낸다. 각 연동은 고유 ID를 갖는다.

관계형 데이터베이스 구조의 예

대부분의 데이터베이스는 관계형이며 이는 스프레드시트와 같이 **테이블 형태**로 데이터가 연관돼 있다는 뜻이다. 테이블에서 열은 저장될 데이터 유형을 정의하며 행에는 이 열에 해당되는 값이 저장된다.

테이블 조인은 일반적으로 아이템 간의 관계를 정의하기 위해 ID만을 취한다. 이 관계는 데이터베이스 시스템이 이름을 참조 ID를 통해 얻도록 설계돼 있다. 이런 구조를 사용하는 데이터베이스는 SQL 베이스 시스템으로 부르며, 그 반대 의미로 **몽고DB의 경우는 NoSQL 데이터베이스 시스템으로 부른다.**

몽고DB의 쿼리 언어는 JavaScript 백그라운드를 갖고 있는 사람이라면 이해하기 쉽다.

13.1 MongoDB 세팅

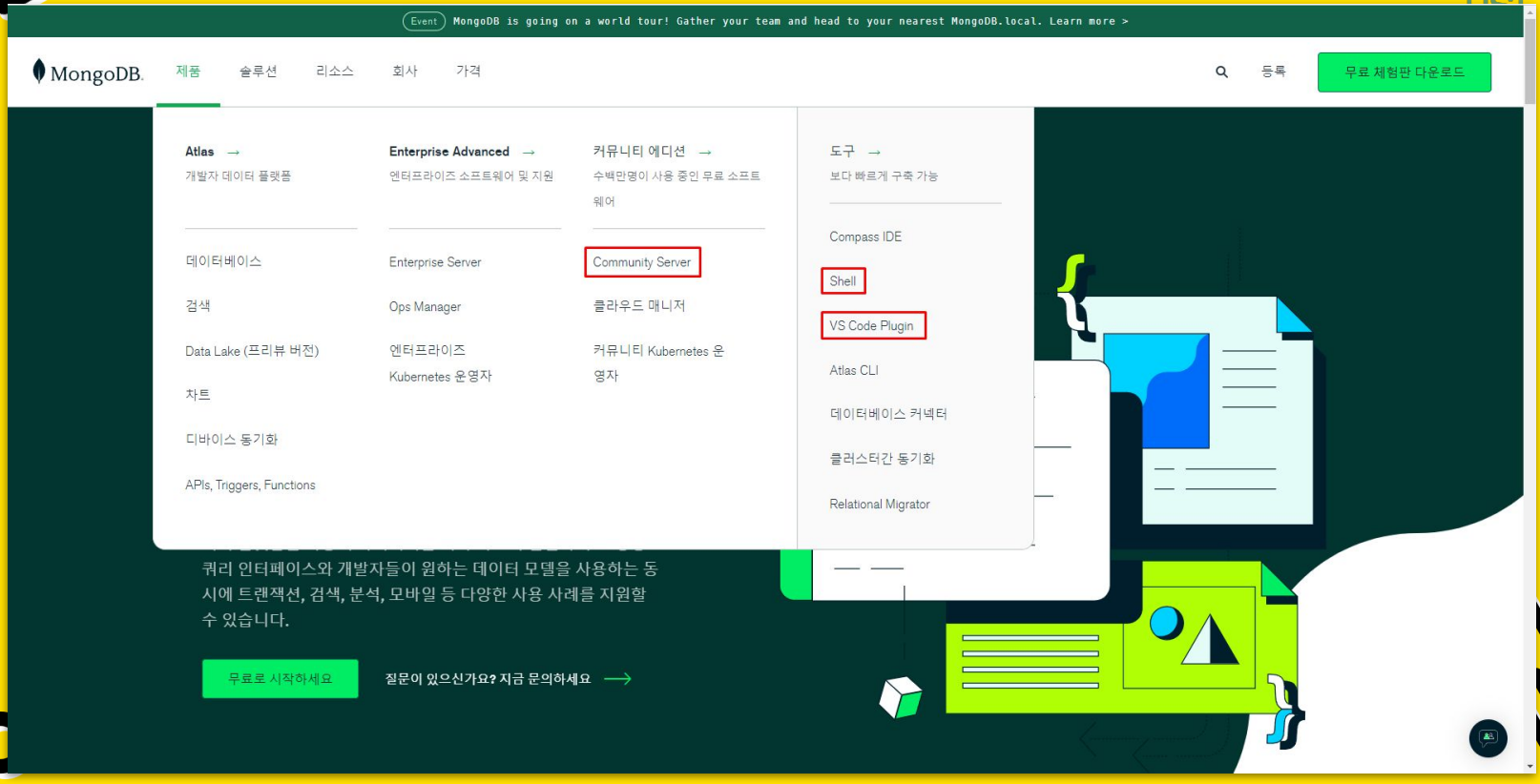
MongoDB 설치

윈도우에서는 다음의 단계를 따른다.

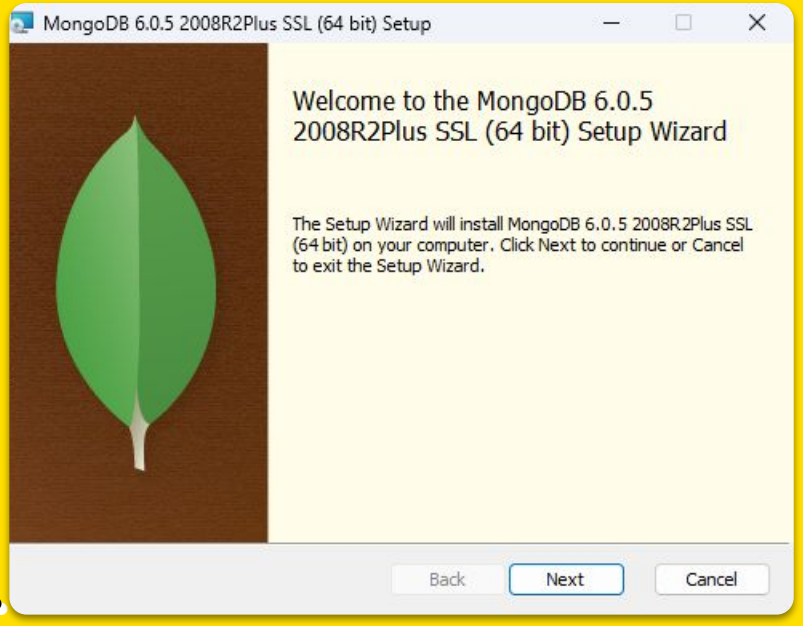
- https://www.mongodb.com/download-center/community로 접속한다.
- MongoDB for 윈도우(`.msi`)를 다운로드 한다.
- 설치 파일을 다운로드했다면 이를 실행하고 디폴트 설치 과정으로 설치한다.
- 설치가 완료되면 `C:\`로 가서 `data` 폴더를 생성하고 그 하위에 공유 폴더를 생성한다.

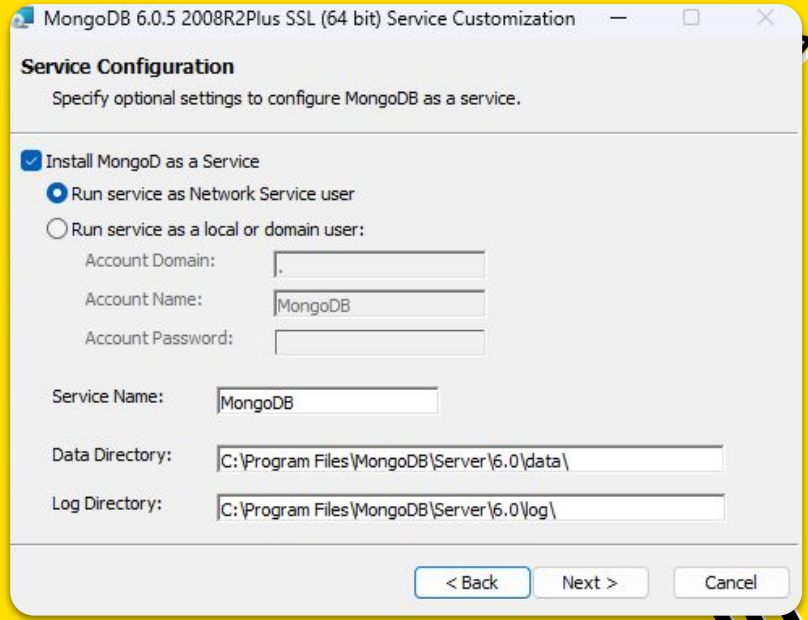
[노트] 윈도우의 경우에는 환경변수 중 `PATH` 에 몽고DB 폴더 경로를 추가해야 한다. 몽고DB 경로는 대부분 `C:\ProgramFiles\MongoDB\Server\3.6.2\bin\mongod.exe` 와 같을 것이다.

13.1 MongoDB 설치

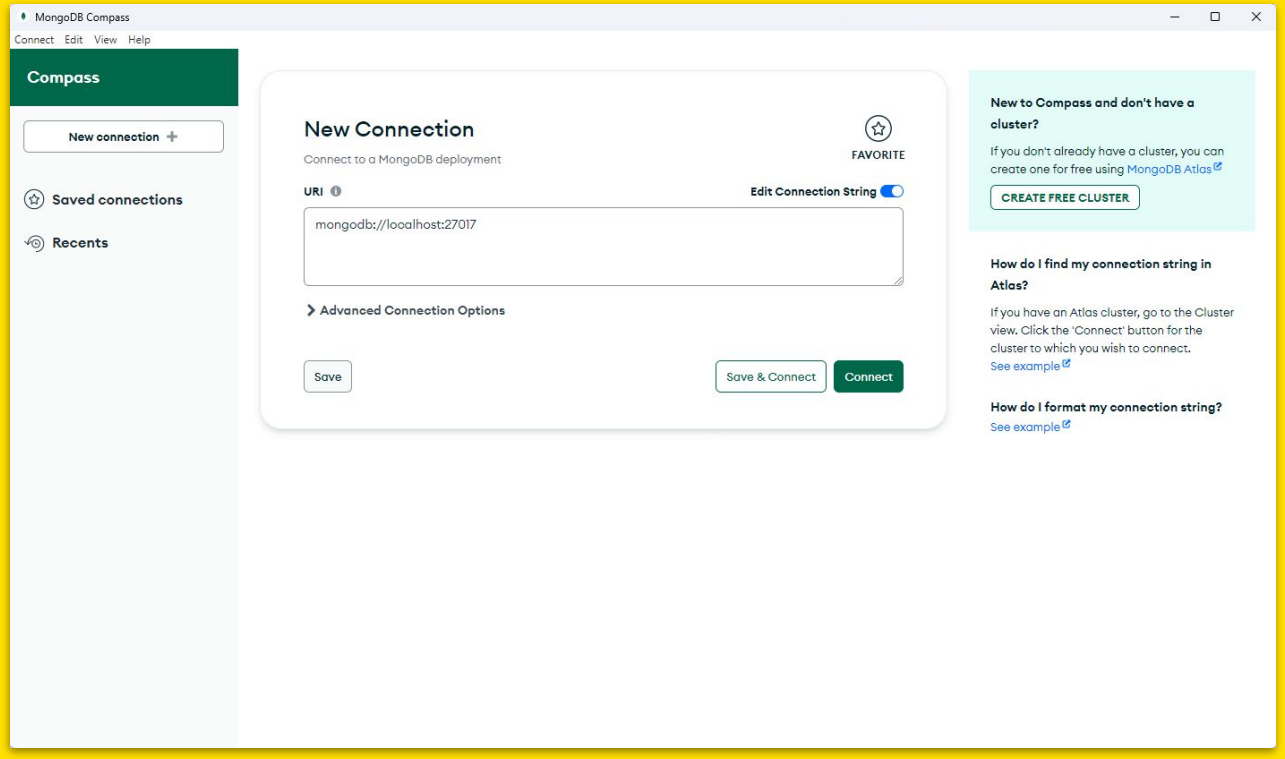


13.1 MongoDB 설치

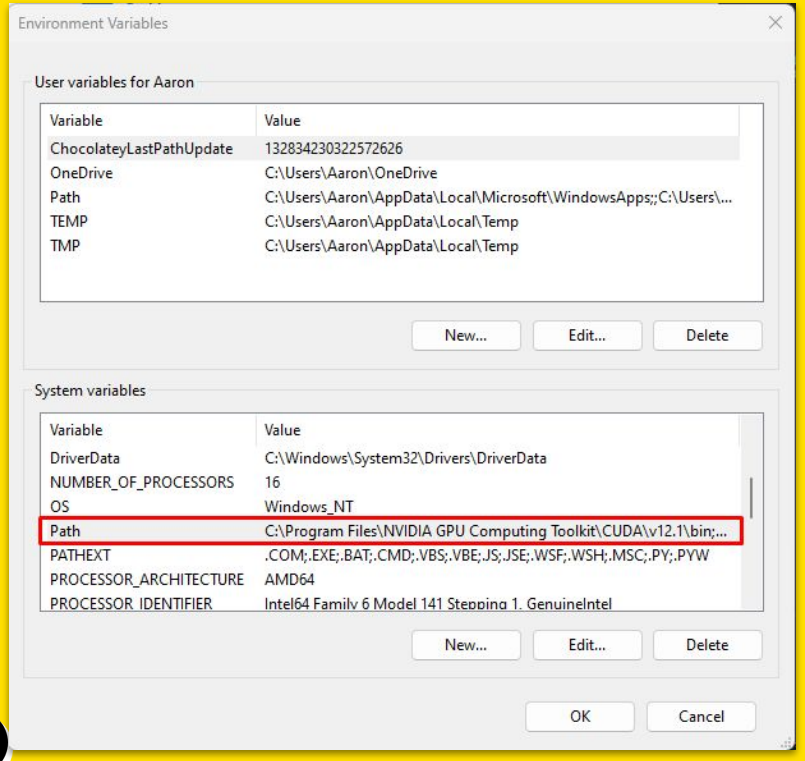


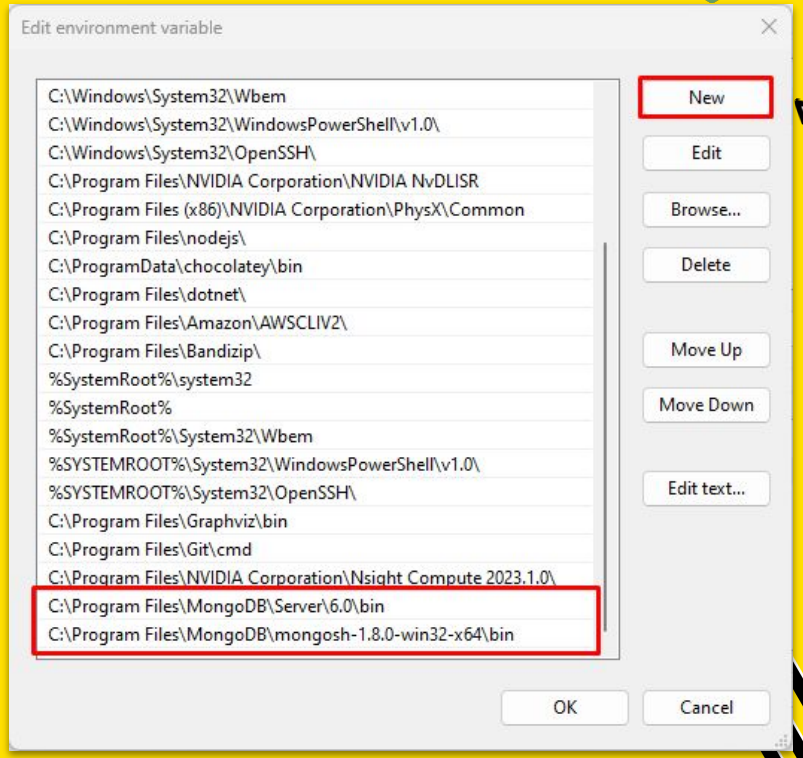


13.1 MongoDB Compass



13.1 MongoDB를 PATH에 추가하기

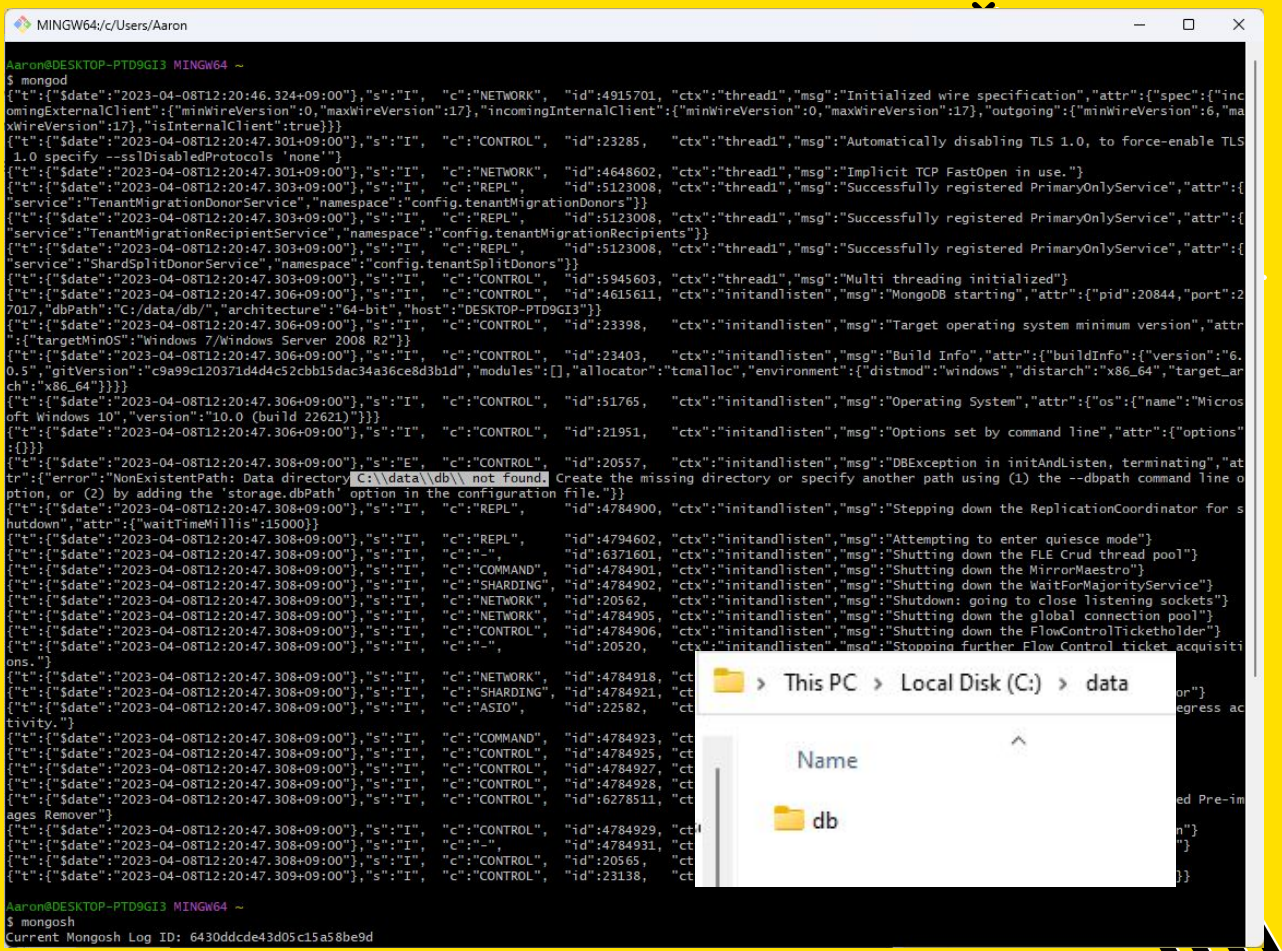




13.1 MongoDB REPL

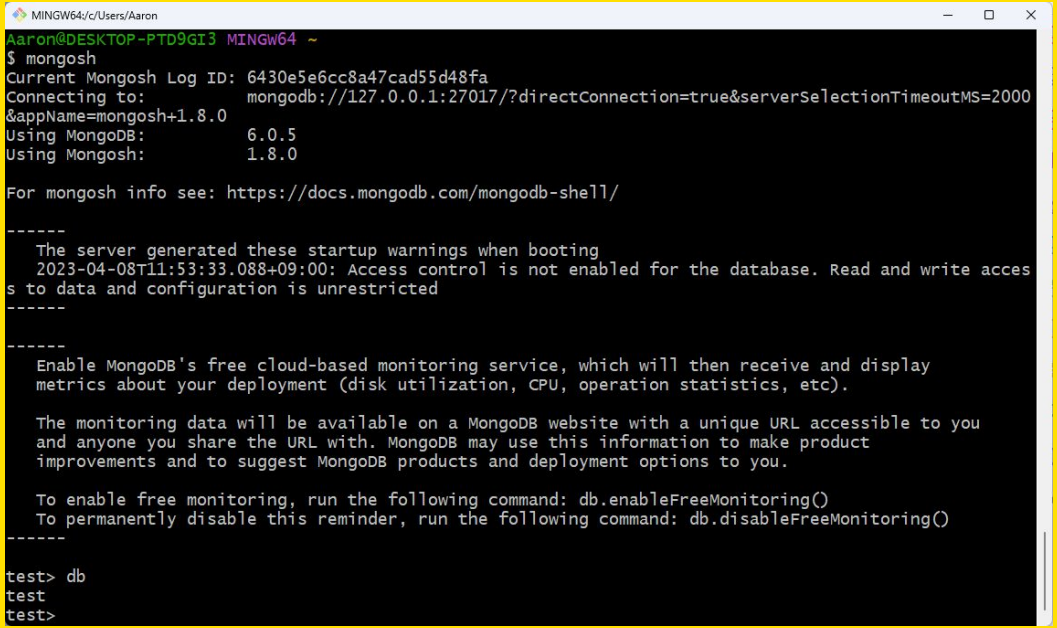
MongoDB Shell은 REPL 환경을 사용하기 위해 필요하다.
다운로드하고 바이너리가 PATH에 있는지 확인하라.
그런 다음 새 터미널 창에서 **mongosh**를 실행하여 REPL 환경을 시작하라.

또한 MongoDB는 **C:/data/db** 디렉토리를 사용하여 데이터를 저장합니다.
따라서 이 디렉토리를 만들 수 있습니다.



13.2 MongoDB셸에서 명령어 실행

애플리케이션에 몽고DB를 연결시키기 전에 몽고DB 셸에서 몇 가지 명령어를 테스트해보자.



[주의] 몽고DB 셸에서의 명령은 되돌릴 수 없다. 데이터 (또는 데이터베이스 전체) 삭제 명령을 내렸다면 이를 복구할 방법은 없다.

터미널창에서 **mongosh**를 입력해 실행한다. 시작을 위해 몽고DB에는 test 데이터베이스가 기본으로 들어있다. 현재 데이터베이스 목록을 보여주는 **db** 명령어를 사용해 test 데이터베이스를 볼 수있다. 모든 가용 데이터베이스를 보려면 **show dbs**를 사용한다.

```
test> show dbs
admin   40.00 KiB
config  72.00 KiB
local   40.00 KiB
test>
```

터미널에서 새로운 컬렉션으로 데이터 추가

새로운 데이터베이스를 만들 수 있고 즉시 이를 **use <새로운 데이터베이스 이름>**을 입력해 스위칭할 수 있다. **use recipe_db**를 입력해 한번 레시피 애플리케이션용 데이터베이스로 스위치해보라.

```
test> use recipe_db
switched to db recipe_db
recipe_db>
```

데이터베이스에 데이터를 추가하려면 데이터가 연관돼 있는 *컬렉션 네임*을 특정해야 한다. 몽고DB의 컬렉션은 데이터 모델을 표현하며, 동일한 그룹 내에서 그 모델과 관계된 모든 도큐먼트들을 저장한다. 예를 들어 레시피 애플리케이션을 위한 연락처 리스트를 생성하려고 한다면 새로운 컬렉션을 만들어 Listing 13.3과 같은 명령으로 데이터 아이템을 추가한다. **insert** 메소드는 몽고DB에서 JavaScript 객체 엘리먼트를 새로운 도큐먼트에 추가하기 위한 명령을 수행한다.

```
recipe_db> db.contacts.insert({
... name: "Aaron Snowberger",
... email: "aaron@ut.ac.kr",
... note: "From Jeonju."
... })
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("6430ec060bb7daf8d12441b0") }
}
recipe_db>
```

터미널에서 모든 데이터 보기

이때 컬렉션 구조는 매우 유연해 앞선 데이터 패턴에 따라야 할 필요 없이 **어떤 값이라도** 도큐먼트에 추가할 수 있다. MongoDB는 문제없이 데이터 추가를 허가할 것이다.

```
recipe_db> db.contacts.insertOne({
... name: "Tom Snowberger",
... home: "Wyoming",
... job: "UPS driver"
... })
{
  acknowledged: true,
  insertedId: ObjectId("6430ece10bb7daf8d12441b1")
}
recipe_db>
```

[노트] 몽고DB가 일관성이 없는 데이터 저장을 문제 삼지 않는다고 해서 꼭 이렇게 해야 된다는 것은 아니다. 14장에서 애플리케이션 모델을 둘러싼 데이터 분류를 논의한다.

컬렉션 내 콘텐츠 목록을 보려면 **`db.contacts.find()`** 명령을 입력하면 된다. 위에서 추가한 2개의 아이템이 몽고DB가 제공하는 추가 속성으로 모두 보인다. `id` 특성은 고윳값을 저장하며 데이터베이스에서 특정 아이템을 구분하고 저장할 때 사용한다.

```
recipe_db> db.contacts.find()
[
  {
    _id: ObjectId("6430ec060bb7daf8d12441b0"),
    name: 'Aaron Snowberger',
    email: 'aaron@ut.ac.kr',
    note: 'From Jeonju.'
  },
  {
    _id: ObjectId("6430ece10bb7daf8d12441b1"),
    name: 'Tom Snowberger',
    home: 'Wyoming',
    job: 'UPS driver'
  }
]
recipe_db>
```

ObjectId

contacts 컬렉션에서 **db.contacts.find(_id: ObjectId("5941fce5cda203f026856a5d"))** 를 입력해 특정 아이템을 한번 찾아보라. **[노트]** 위 명령에서 ObjectId를 데이터베이스 출력 결과 중 다른 것으로 바꿔 실행해보자.

```
recipe_db> db.contacts.find({_id: ObjectId("6430ece10bb7daf8d12441b1")})
[
  {
    _id: ObjectId("6430ece10bb7daf8d12441b1"),
    name: 'Tom Snowberger',
    home: 'Wyoming',
    job: 'UPS driver'
  }
]
recipe_db>
```

ObjectId

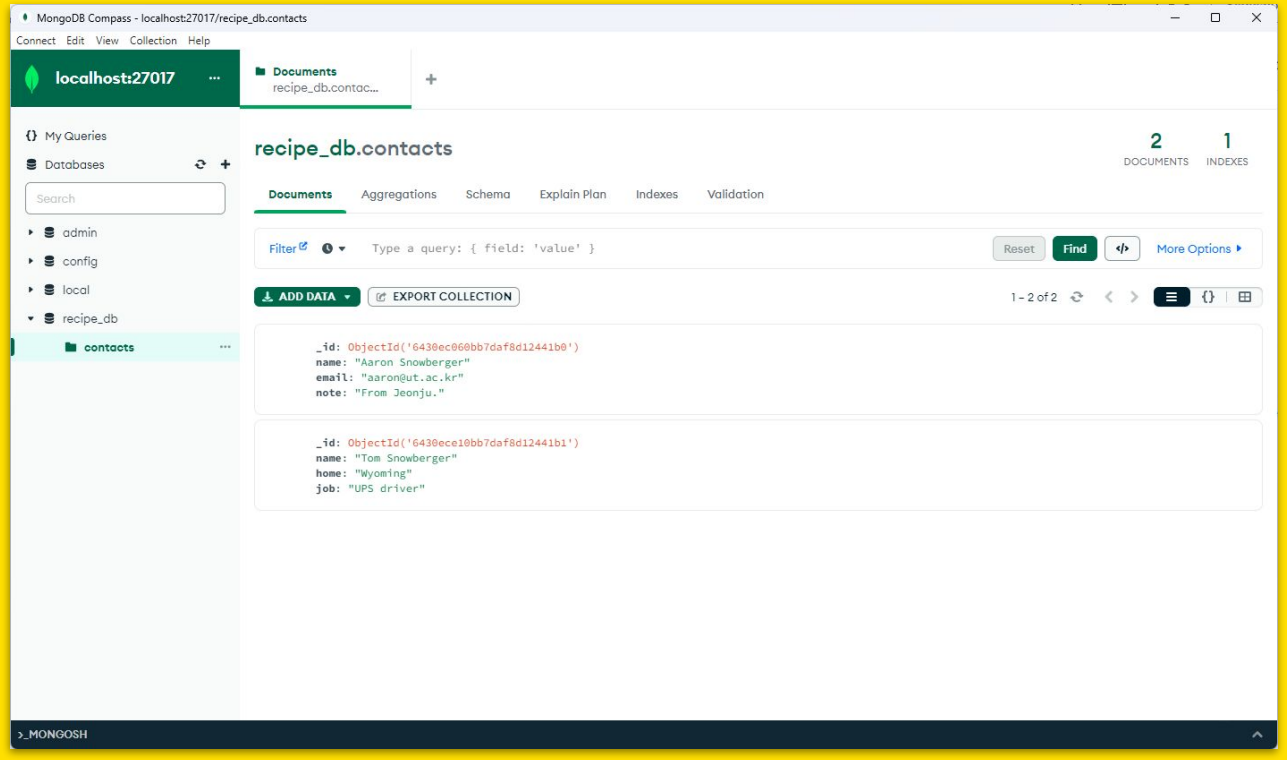
데이터를 유니크하게 분류하기 위해 몽고DB는 ObjectId 클래스를 사용한다. 이는 데이터베이스 도큐먼트의 중요 정보를 기록한다.

예를 들어 ObjectId("5941fe7acda203f026856a5e")는 데이터베이스의 도큐먼트를 나타내는 새로운 ObjectId를 만든다. 16진수 값이 도큐먼트 컨스트럭터, 기록 타임스탬프, 데이터베이스 시스템에 대한 정보를 참조하는 ObjectId로 할당되는 것이다.

ObjectId 인스턴스의 결과는 데이터베이스에서 많은 유용한 메소드를 제공하며 이를 통해 데이터 정렬이나 구조화를 할 수 있다. 결과적으로 _id 특성은 몽고DB에서 도큐먼트 ID 스트링보다 더 유용한 기능이 된다.

13.1 MongoDB Compass

가급적이면 몽고DB Compass는 애플리케이션에서 몽고DB 관련 작업 시 보조적 도구로 사용하기를 권한다.



MongoDB의 셸 명령

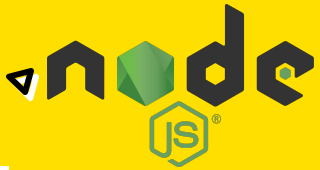


표 13.1 몽고DB 셸 커맨드

커맨드	설명
`show collections`	데이터베이스 내 모든 컬렉션을 출력한다. 나중에 이 컬렉션들은 독자의 모델에 매치해야 한다.
`db.contacts.findOne`	하나의 아이템을 임의로 돌려주거나, 파라미터로 받은 값에 맞는 값을 돌려준다(예: `findOne(name:"Jon")`).
`db.contacts.update({name: "Jon"}, {name: "Jon Wexler"})`	파라미터의 첫 번째 항목을 두 번째 항목으로 업데이트한다.
`db.contacts.delete({name: "Jon Wexler"})`	매칭되는 도큐먼트를 컬렉션에서 삭제한다.
`db.contacts.deleteMany({})`	컬렉션 내에 있는 모든 도큐먼트를 삭제한다. 이 명령은 원상복구되지 않는다.

더 많은 명령어를 보려면 치트 시트인
https://www.mongodb.com/developer/products/mongodb/cheat-sheet/을
참조하라. 새로운 mongosh와 연결하려면 mongo를 **`mongosh`**로 바꾸면 됩니다.

13.3 애플리케이션에 MongoDB 연결

표 13.1 몽고DB 셸 커맨드

커맨드	설명
`show collections`	데이터베이스 내 모든 컬렉션을 출력한다. 나중에 이 컬렉션들은 독자의 모델에 매치해야 한다.
`db.contacts.findOne`	하나의 아이템을 임의로 돌려주거나, 파라미터로 받은 값에 맞는 값을 돌려준다(예: `findOne(name:"Jon")`).
`db.contacts.update({name: "Jon"}, {name: "Jon Wexler"})`	파라미터의 첫 번째 항목을 두 번째 항목으로 업데이트한다.
`db.contacts.delete({name: "Jon Wexler"})`	매칭되는 도큐먼트를 컬렉션에서 삭제한다.
`db.contacts.deleteMany({})`	컬렉션 내에 있는 모든 도큐먼트를 삭제한다. 이 명령은 원상복구되지 않는다.

더 많은 명령어를 보려면 치트 시트인 https://www.mongodb.com/developer/products/mongodb/cheat-sheet/을 참조하라. 새로운 mongosh와 연결하려면 mongo를 **mongosh**로 바꾸면 됩니다.

13.3 애플리케이션에 MongoDB 연결

```
// main.js
"use strict";

const port = 3000,
  express = require("express"),
  layouts = require("express-ejs-layouts"),
  homeController = require("./controllers/homeController"),
  errorController = require("./controllers/errorController"),
  app = express(),
  MongoDB = require("mongodb").MongoClient, // 몽고DB 모듈의 요청
  dbURL = "mongodb://localhost:27017/",
  dbName = "recipe_db";

// 로컬 데이터베이스 서버 연결 설정
MongoDB.connect(dbURL, (error, client) => {
  if (error) throw error;
  let db = client.db(dbName); // 몽고DB 서버로의 recipe_db 데이터베이스 연결 취득
  db.collection("contacts")
    .find()
    .toArray((error, data) => { // contacts 컬렉션의 모든 레코드 찾기
      if (error) throw error;
      console.log(data); // 콘솔에 결과 출력
    });
});
```

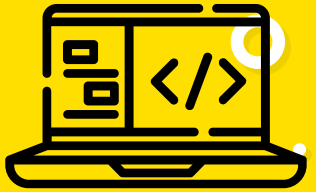
몽고DB 에 Node.js 애플리케이션을 연결하기 위해 터미널에서 **npm install mongodb** 명령을 실행하다.

그런 다음 main.js에 옆에 있는 코드를 추가합니다.

13.3 애플리케이션에서 데이터 삽입

```
14  // 로컬 데이터베이스 서버 연결 설정
15  MongoDB.connect(dbURL, (error, client) => {
16    if (error) throw error;
17    let db = client.db(dbName); // 몽고DB 서버로의 recipe_db 데이터베이스 연결 취득
18    db.collection("contacts")
19      .find()
20      .toArray((error, data) => { // contacts 컬렉션의 모든 레코드 찾기
21        if (error) throw error;
22        console.log(data); // 콘솔에 결과 출력
23      });
24
25    db.collection("contacts")
26      .insertOne({
27        name: "Megan Snowberger",
28        relationship: "sister",
29        location: "Jeonju"
30      }, (error, result) => { // 데이터베이스에 새 정보 삽입
31        if (error) throw error;
32        console.log(result); // 삽입 결과 출력
33      });
34  });
```

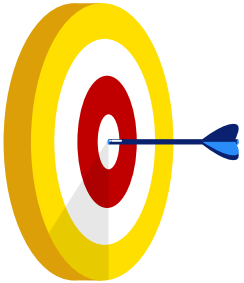
Node.js 애플리케이션 안에서 몽고DB 셸에서 내렸던 명령을 동일하게 사용할 수 있다.



Coding!

Listing 13.5 & 13.6

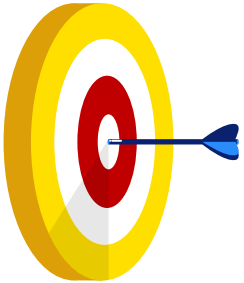
p. 200-201



퀵 체크 13.1

몽고DB에서 데이터 저장을 위해 사용되는
데이터 구조는 무엇인가?

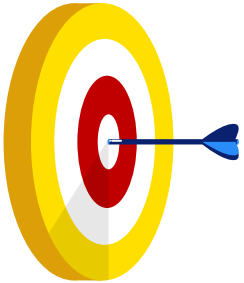
몽고DB는 데이터 저장에 '도큐먼트'를 사용한다.



퀵 체크 13.2

데이터베이스 내의 컬렉션을 보는
몽고DB 명령어는 무엇인가?

몽고DB 셸에서 show collections 명령을 치면
모든 컬렉션들이 리스트된다.



퀵 체크 13.3

참 또는 거짓! 존재하지 않은 데이터베이스를 접속하려고 하면 몽고DB는 에러를 발생시킨다.

거짓이다. 몽고DB는 에러를 발생시키는 대신 제공된 이름으로 새로운 데이터베이스를 생성한다.

14

Mongoose를 사용한 모델 제작

Mongoose의 설치와 Node.js 애플리케이션으로 연결
스키마 생성
Mongoose 데이터 모델의 생성과 초기화
사용자 정의 메소드로 데이터 읽기와 저장하기

p. 203-211

애플리케이션으로 **Mongoose** 설정

Mongoose는 애플리케이션의 객체지향 구조를 보장하는 방법으로 몽고DB 명령어를 실행할 수 있는 객체-도큐먼트 매퍼(ODM[object-document mapper])다. 예를 들어 몽고DB 단독으로는 하나의 도큐먼트가 저장되고 다음 도큐먼트가 저장될 때 일관성을 유지하기가 힘들다.
Mongoose는 어떤 형태의 데이터가 저장돼야 하는지에 대한 스키마로 모델을 만드는 도구이고 이런 상황을 개선시킬 수 있다.

```
npm install mongoose
```

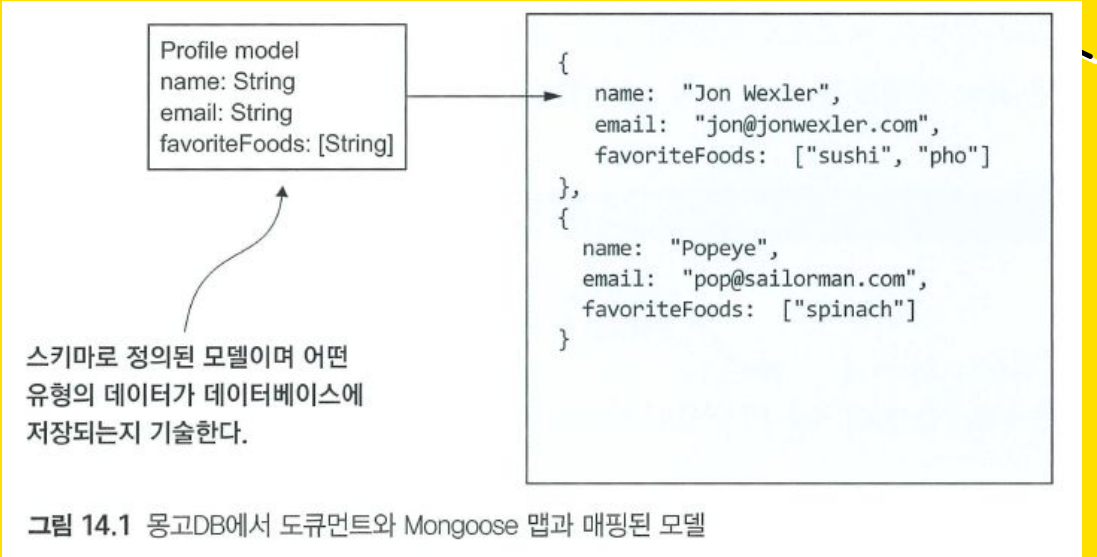


그림 14.1 몽고DB에서 도큐먼트와 Mongoose 맵과 매핑된 모델

- 모델-뷰-컨트롤러 아키텍처에 대해 이미 2부에서 논의했다. 어떻게 컨트롤러가 뷰와 모델 간에 정확한 데이터가 애플리케이션에서 전달되기 위한 커뮤니케이션을 하는지도 기술했다. 모델은 JavaScript 객체와 비슷하며 Mongoose는 이를 데이터베이스 쿼리를 분류하는 데 사용한다.

애플리케이션을 통한 **Mongoose** 설정

Mongoose를 사용하면 더 이상 mongodb를 main.js에서 요청을 하지 않아도 되며 13장에서의 몽고DB도 사용하지 않는다.

Listing 14.1 main.js에서 Node.js 애플리케이션을 통한 Mongoose 설정

```
const mongoose = require("mongoose");          ← Mongoose 요청
mongoose.connect(
  "mongodb://localhost:27017/recipe_db",       ← 데이터베이스 커넥션 설정
  {useNewUrlParser: true}
);
const db = mongoose.connection;                ← db 변수에 데이터베이스 할당
```

Listing 14.2 main.js에서 데이터베이스 접속 시 메시지 로깅

```
                                  애플리케이션이 데이터베이스에
                                  연결됐을 때 메시지 로깅
db.once("open", () => {
  console.log("Successfully connected to MongoDB using Mongoose!");
});
```

[노트] 이때도 몽고DB 서버가 백그라운드에 실행 중이어야 함을 기억하라. 몽고DB의 실행은 터미널 윈도우에서 **mongod** 명령으로 실행한다.

14.2 스키마의 생성

스키마는 프로그래밍 언어에서 **클래스 정의** 또는 더 나아가 애플리케이션에서 특정 객체를 위해 어떻게 데이터를 분류해야 하는지에 관한 **청사진**과 같다.

데이터가 일관성을 잃어버리는 경우, 이를테면 어떤 도큐멘트는 email 필드를 갖고 있고 어떤 것은 없다면 모든 contact 객체는 데이터베이스에 저장되기 위해서는 email 필드가 필요하다고 기술하는 스키마를 생성할 수 있다. 레시피 애플리케이션에 뉴스레터 구독 양식을 추가하기 위해 구독자(subscriber) 스키마를 생성하자.

Listing 14.3 main.js에서의 구독자 스키마

```
const subscriberSchema = mongoose.Schema({
  name: String,
  email: String,
  zipCode: Number
});
```

mongoose.Schema로 새로운 스키마 만들기

스키마 속성 추가

[노트] 몽고 DB가 스키마를 수행하지 않는다. Mongoose가 실행한다.

Mongoose 스키마 데이터에 대한 자세한 정보는 Mongoose 사이트에 참조하라.

main.js에서 생성과 저장 구문

이제 스키마가 정의됐다. 아래 있는 명령을 통해 이를 모델이 적용해야 한다.
`var Subscriber = mongoose.model( "Subscriber", subscriberSchema)`
Subscriber 의 참조를 통해 이 모델의 새로운 객체를 초기화할 수 있다. Listing 14
4에서와 같이 새로운 객체를 얻는 방법은 **두 가지**가 있다.

1

```
var subscriber1 = new Subscriber({
  name: "Jon Wexler",
  email: "jon@jonwexler.com"
});
        .save().then(...
subscriber1.save((error, savedDocument) => {
  if (error) console.log(error);
  console.log(savedDocument);
});
```

새로운 subscriber의 초기화
subscriber를 데이터베이스에 저장
에러 발생 시 다음 미들웨어 함수로 에러를 전달
저장된 데이터 도큐먼트의 로그

2

```
  .create({...}).then(...
Subscriber.create(
  {
    name: "Jon Wexler",
    email: "jon@jonwexler.com"
  },
  function (error, savedDocument) {
    if (error) console.log(error);
    console.log(savedDocument);
  }
);
```

create는 new와 save를 한 번에 하는 역할을 한다. 생성과 저장을 한 번에 하려면 이 Mongoose 메소드를 사용하라.

[에러] MongooseError: Model.save() no longer accepts a callback
[에러] MongooseError: Model.create() no longer accepts a callback

Mongoose 7의 콜백은 더 이상 사용되지 않으므로 `.then()`과 함께 JavaScript 프라미스를 사용하십시오.

14.3 모델의 분류

이제 Mongoose 모델의 형태로 데이터를 저장하는 방법을 알게 됐다. 하지만 **데이터를 잘 분류해 main.js 파일을 복잡하게 하지 않기**를 원할 수 있을 것이다. 뷰와 컨트롤러에 했던 것처럼 애플리케이션의 루트 레벨에서 **model 폴더**를 만든다. Model 폴더 안에 subscriber.js라는 파일을 만든다. 이 파일에 모델 코드를 옮겨 기입한다. 스키마 및 모델 정의 관련 코드들을 모두 옮기고 모델을 파일의 exports 객체로 옮긴다.

Listing 14.5 subscriber.js에서 모듈 분리를 위한 모델과 스키마의 이동

```
const mongoose = require("mongoose"),
  subscriberSchema = mongoose.Schema({
    name: String,
    email: String,
    zipCode: Number
  });
module.exports = mongoose.model("Subscriber", subscriberSchema);
```

module export 시에만 Subscriber를 익스포트

[노트] 이때 mongoose 모듈을 요청해야 한다. Mongoose 메소드를 스키마와 모델이 작업에 사용하기 때문이다. Node.js는 프로젝트에서 모듈을 한 번만 읽어들이기 때문에 **이 부분이 애플리케이션을 느리게 만들지는 않는다.** 이미 로딩된 모듈을 쓰는 것이기 때문이다.

다음으로 `const Subscriber = require("./models/subscriber")` 를 main.js에 추가해 모델을 요청한다.

main.js에서 쿼리 수행 예제

main.js에서 Mongoose의 `findOne`으로 데이터베이스의 도큐먼트를 찾고 `where` 로 메소드를 쿼리한다.

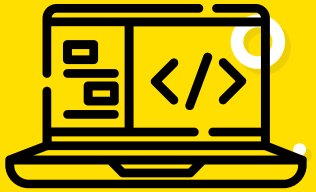
예를 들어 이메일에 wexler라는 스트링이 들어가 도큐먼트 하나를 찾으려고 하면 `Subscriber.findOne({name: "Jon Wexler"}). where("email", /wexler/)` 를 수행시키면된다. 이 간단한 사용자 정의 쿼리는 필요한 데이터를 얻기 위해 얼마나 유연하게 사용될 수 있는지 보여준다. Mongoose는 부분 부분의 쿼리를 연결하도록 해주며 변수에 쿼리를 저장하게도 한다. `var findWexlers` 라는 변수를 만들 수 있으며 wexler라는 단어가 들어 있는 이메일을 찾는 쿼리를 할당할 수도 있다. 그러면 나중에 `findWexlers.exec()` 를 사용해 쿼리를 수행할 수 있다 (exec 에 대한 자세한 사항은 15장을 참조하라).

exec 메서드 없이 쿼리를 실행하려 한다면 2개의 인수를 가지는 콜백 함수가 필요하다. 첫 번째 인수는 발생한 에러를 의미하며, 두 번째 인수는 데이터베이스가 돌려주는 데이터를 의미한다(Listing 14.6). https://mongoosejs.com/docs/queries.html에 있는 예제 쿼리들을 따라 해보고 쿼리를 만들어보라.

Listing 14.6 main.js에서 쿼리 수행 예제

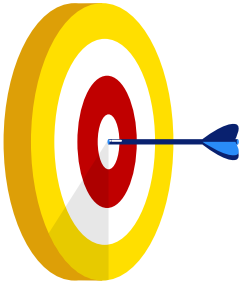
```
var myQuery = Subscriber.findOne({
    name: "Jon Wexler"
  })
  .where("email", /wexler/);
myQuery.exec((error, data) => {
  if (data) console.log(data.name);
});
```

데이터와 에러 처리를 위한 콜백 함수로 쿼리 실행



Coding!

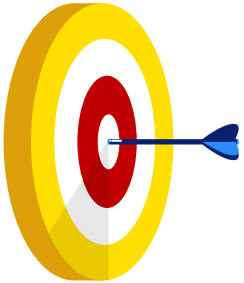
main1.js & main2.js
(Listing 14.1 - 14.4) & (Listing 14.5 - 14.6)
p. 203-211



퀵 체크 14.1

ODM이란 무엇인가?

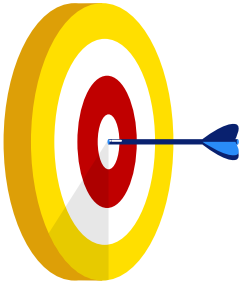
객체 도큐먼트 매퍼다. 바로 애플리케이션 내에서 Mongoose가 하는 역할이다. ODM(object-document mapper)(객체 관계 매퍼와 비슷하다)은 애플리케이션에서 객체 관점으로 고려하기 쉽게 만들며 어떻게 데이터베이스에 저장돼야 하는지 고민할 필요가 없게 한다.



퀵 체크 14.2

참 또는 거짓! new Subscriber({name: "Jon", email: “jon@jonwexler.com”})은 데이터베이스에 새로운 레코드를 기록한다.

거짓이다. 이 코드는 가상의 객체를 생성할 것이다. 이 행의 값을 변수에 저장하고 save를 그 변수에 대해 실행시키면 새로운 subscriber가 데이터베이스에 저장된다.



퀵 체크 14.3

Mongoose 스키마에서 각 특정된 필드를 위해 요구되는 두 개의 컴포넌트는 무엇인가?

스키마는 속성 이름(name)과 데이터(data)가 필요하다.

15

컨트롤러와 모델과의 연결

컨트롤러와 모델과의 연결
컨트롤러 액션을 통한 데이터 저장
프라미스로 데이터베이스 쿼리 실행
폼 데이터 포스팅의 처리

p. 213-226

서버 코드의 수정

15장에서는 **라우트를 이 모델과 컨트롤러에 연결해** 사용자의 URL 요청에 기반한 의미 있는 데이터를 저장할 수 있다. 먼저 새로운 구독자 라우트를 위한 컨트롤러를 생성한다. 그리고 이 라우트를 **JavaScript ES6을 지원하는 프라미스**로 전환할 것이다.

콜백은 쿼리가 복잡해지면 자칫 같이 복잡해질 수 있다. 다행히 좀 더 간결한 방법으로 콜백과 데이터 및 에러 처리를 위한 다른 여러 종류의 문법들을 사용할 수 있다. **프라미스**가 바로 그것이며 Mongoose는 애플리케이션 내에서 프라미스 구문 사용을 지원한다.

15.1 구독자를 위한 컨트롤러 만들기

컨트롤러C는 모델M(데이터)과 뷰V(웹 페이지)를 연결하는 역할을 한다는 점을 떠올려보자. 이제 모델을 설정했고 이 모델과 연관된 데이터를 특별히 찾는 외부 요청을 처리하는 컨트롤러가 필요하다. 만일 누군가가 홈 경로인 /를 요청한다면 홈 컨트롤러에서 로직을 따르는 뷰를 돌려줄 수 있다. 이제 다른 누군가가 구독자로서 등록 요청을 보내면 **구독자 컨트롤러**를 수행해야 한다. Controllers 폴더에 `subscribersController.js` 를 생성한다.

Listing 15.1 subscribersController.js에서의 구독자 컨트롤러의 제작

```
const Subscriber = require("../models/subscriber");

exports.getAllSubscribers = (req, res, next) => {
  Subscriber.find({}, (error, subscribers) => {
    if (error) next(error);
    req.data = subscribers;
    next();
  });
};
```

구독자 모듈의 요청

데이터베이스로부터의 데이터를 다음 미들웨어 함수로 전달하기 위한 getAllSubscribers의 exports

구독자 모델에서 검색 쿼리

에러를 미들웨어 함수로 전달

요청 객체에 대해 몽고DB로부터 돌아온 데이터의 세팅

다음 미들웨어 함수의 진행

노트 모델이 다른 폴더에 들어 있기 때문에 model 폴더에 접속해 요청하려면 앞에 ..를 붙여야 한다.

[노트] 일반적으로 컨트롤러는 모델의 여러 버전에서 이름이 지정된다. 엄격한 규칙은 없고, 이미 `homeController.js` 가 있지만 이 컨트롤러는 애플리케이션에서 모델을 나타내지 않는다.

[노트] `find` 메소드를 인수 없이 사용하는 것은 빈 객체 (`{}`)를 인수로 넣는 것과 동일하다. 여기서 **빈 객체의 의미는 모든 구독자**를 어떤 조건 없이 읽어오는 것을 원한다는 것을 의미한다.

main.js에서 구독자 컨트롤러 사용

다음 `const subscribersController = require("./models/subscribersController")` 를 main.js에 추가해 모델을 요청한다.

Listing 15.2 main.js에서 구독자 컨트롤러 사용

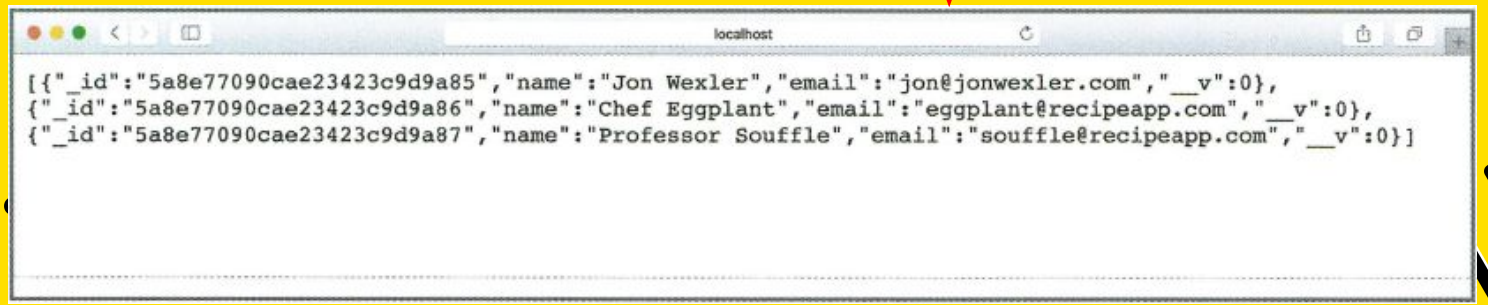
```
app.get("/subscribers", subscribersController.getAllSubscribers,
  (req, res, next) => {
    console.log(req.data);
    res.send(req.data);
});
```

getAllSubscribers 함수로 요청 전달

요청 객체로부터의 데이터 로깅

브라우저 창에 데이터 렌더링

계획대로 동작했다면 **http://localhost:3000/subscribers**로 방문해 이름과 이메일로 데이터베이스에 있는 구독자 목록을 볼 수 있다.



subscribers.ejs에서 구독자 정보의 출력

결과 데이터를 확인하는 것 대신 뷰 페이지에서 데이터로 반응하는 것으로 즉시 액션의 개선이 가능하다. 액션의 리턴 구문을 수정해 이를 Express.js 의 res.render로 바꾼다. 뷰를 렌더링하기 위한 라인은 subscribers.ejs를 호출하며 형태는 **`res.render("subscribers", {subscribers: subscribers})`**이다.
이 응답은 subscribers.ejs라 부르는 뷰를 렌더링하기 위해 호출하며 subscribers라는 변수를 통해 구독자 데이터를 데이터베이스로부터 이 뷰로 전달한다.

Listing 15.3 subscribers.ejs에서 구독자 정보의 출력

```
<% subscribers.forEach(s => { %>
  <p><%= s.name %></p>
  <p><%= s.email %></p>
<% }); %>
```

구독자 정보들을 탐색

뷰로 구독자 데이터 삽입

[노트] 결국 이 페이지는 애플리케이션 관리자가 누가 레시피 애플리케이션에 등록했는지 보기 위해 사용될 것이다.

- 하지만 당장은 이 페이지는 이 라우트로 접근하는 사람들은 누구나 볼 수 있다.

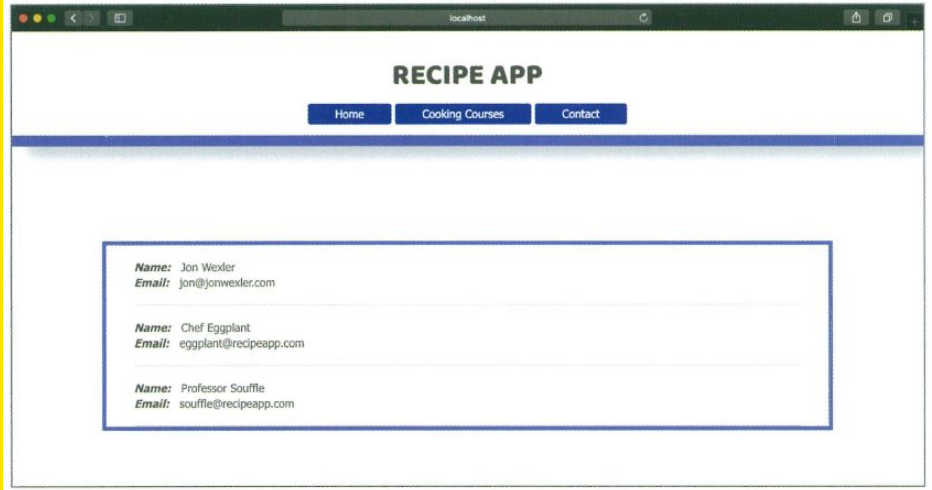


그림 15.2 구독자 데이터 목록 뷰를 사용한 브라우저 예시

15.2 포스팅 데이터를 모델로 저장

지금까지는요청이 애폴리케이션의 웹 서버로 전달될 때 단방향으로 흐르는 **데이터가 있어야 했다. 다음 단계는 사용자의 제출 데이터를 구독자 객체의 형태로 저장하는 것이다.** 그림 15.3은 폼으로부터 데이터베이스로의 정보의 흐름을 보여준다.

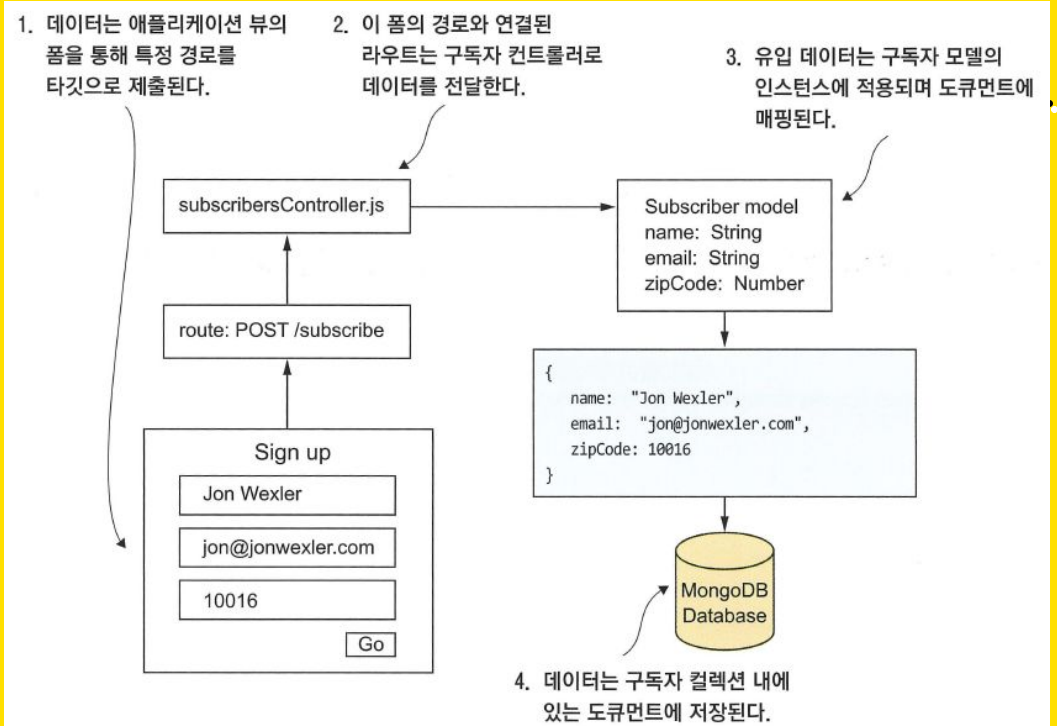


그림 15.3 웹 페이지 폼으로부터 데이터베이스까지의 흐름

구독자 데이터의 포스팅

폼은 HTTP POST 요청을 통해 데이터를 /subscribe로 제출할 것이다. 폼의 입력 부분은 구독자 모델의 필드와 매치된다.

contact.ejs가 렌더링될 때 이 폼이 나타나기 때문에 구독자 컨트롤러로부터 /contact 경로로 요청이 발생 시 이 뷰를 렌더링하기 위한 라우트를 생성한다. 그리고 /subscribe 경로를 위한 GET 라우트가 필요하며 기존의 /contact 경로를 위한 POST 라우트의 수정도 필요하다.

Listing 15.4 subscribe.ejs에서 구독자 데이터의 포스팅 폼

```
<form action="/subscribe" method="post">      ← 구독 폼 추가
  <input type="text" name="name" placeholder="Name">
  <input type="text" name="email" placeholder="Email">
  <input type="text" name="zipCode" placeholder="Zip Code" pattern="[0-9]{5}">
  <input type="submit" name="submit">
</form>
```

Listing 15.5 main.js에서 구독을 위한 라우트

```
app.get("/contact", subscribersController.getSubscriptionPage);
app.post("/subscribe", subscribersController.saveSubscriber);
```

구독 페이지를 위한 GET 라우트 추가

구독 데이터 처리를 위한 POST 라우트 추가

[노트] 이렇게 바꾸고 나면 homeController.js나 main.js의 라우트에 있는 연락처 몸이 더 이상 필요 없게된다.

구독자 데이터의 포스팅

작업을 완료하기 위해 **getSubscriptionPage**와 **saveSubscriber** 함수를 생성한다. subscribersController.js 내에 Listing 15 .6의 코드를 추가한다.

첫 번째 액션은 views 폴더로부터 EJS 페이지를 렌더링한다.

saveSubscriber는 요청으로부터 데이터를 수집하며 body-parser 패키지로 하여금 요청 내 본문 콘텐츠를 읽어들이게 한다. 새로운 모델의 인스턴스가 생성됐으면 구독자 저장을 한다. 만일 저장에 실패한다면 에러로 응답할 것이다. 성공한다면 thanks.ejs로 응답할 것이다.

http://localhost/contact 페이지에서의 폼을 채워 이 코드를 수행해볼 수 있다.

Listing 15.6 subscribersController.js에서 구독 라우트를 위한 컨트롤러 액션

```
exports.getSubscriptionPage = (req, res) => {
  res.render("contact");
};

exports.saveSubscriber = (req, res) => {
  let newSubscriber = new Subscriber({
    name: req.body.name,
    email: req.body.email,
    zipCode: req.body.zipCode
  });

  newSubscriber.save((error, result) => {
    if (error) res.send(error);
    res.render("thanks");
  });
};
```

구독 페이지를 렌더링하기 위한 액션 추가

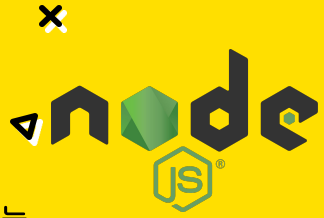
구독자들을 저장하기 위한 액션 추가

새로운 구독자 추가

새로운 구독자 저장

[노트] 몽고 DB는 새로운 구독자를 추가할 때 id를 반환한다. 이 예제에서 result 변수는 이를 포함한다.

15.3 Mongoose로 프라미스 사용



ES6은 비동기 쿼리에서 일반적으로 콜백 함수인 함수 체인을 사용하기 위해 프라미스를 사용하는 아이디어를 채택했다. *프라미스*란 JavaScript 객체로서, 함수 호출 상태에 대한 정보와 다음 체인에 따라 어떤 호출이 일어날까에 대한 정보를 갖고 있다. 미들웨어와 유사하게 프라미스는 함수를 시작시키고 다음 콜백 함수로 넘어가지 않고 완료될 때까지 대기하게 할 수 있다. 결국 프라미스는 중첩 콜백을 표현하는 데 간결한 방법을 제공하지만 지금 애플리케이션에 사용될 데이터베이스 쿼리로는 콜백 함수가 상당히 길어질 수 있다.

필요한 작업은 main.js의 상단부에 **`mongoose.Promise = global.Promise;`**를 추가하다.

이제 만들어진 각 쿼리를 이용해 일반적인 데이터베이스 응답을 돌려줄지, 또는 이런 응답 결과를 포함하는 프라미스를 돌려줄지를 선택할 수 있다.

이렇게 프라미스로 쿼리를 다시 작업해도 데이터베이스에 있는 모든 구독자 정보를 요청하는 것에는 변함이 없다. 쿼리 내에서 뷰를 바로 렌더링하는 것 대신에 뷰를 렌더링할지 또는 에러 로깅과 함께 리젝트할지에 대한 데이터를 포함한 프라미스를 돌려준다. find 에서 exec 호출을 통해 프라미스를 돌려주기 위한 쿼리를 수행하고 있는 것이다.

[노트] `exec` 사용 없이도 이후의 명령어를 처리하기 위해 `then`이나 `catch` 명령을 사용할 수 있다. 하지만 `exec`가 없다면 인증된 프라미스(프라미스 쿼리의 Mongoose 버전)가 아니다. 그러나 `save`와 같은 Mongoose 메소드는 `exec` 없이도 동작한다. 자세한 내용은 http://mongoosejs.com/docs/promises.html을 참조하기 바란다.

15.3 Mongoose로 프라미스 사용

처리 과정에서 에러가 발생하면 에러는 프라미스 체인에서 `catch` 블록으로 전파된다. 에러가 없다면 결과 데이터는 쿼리로부터 다음 `then` 블록으로 전파된다. 이 프라미스-체인 프로시저는 다음에 어떤 코드가 실행돼야 하는지 결정하기 위한 프라미스 블록에서의 결과 코드나 통상의 리젝트 프라미스를 따르게 된다.

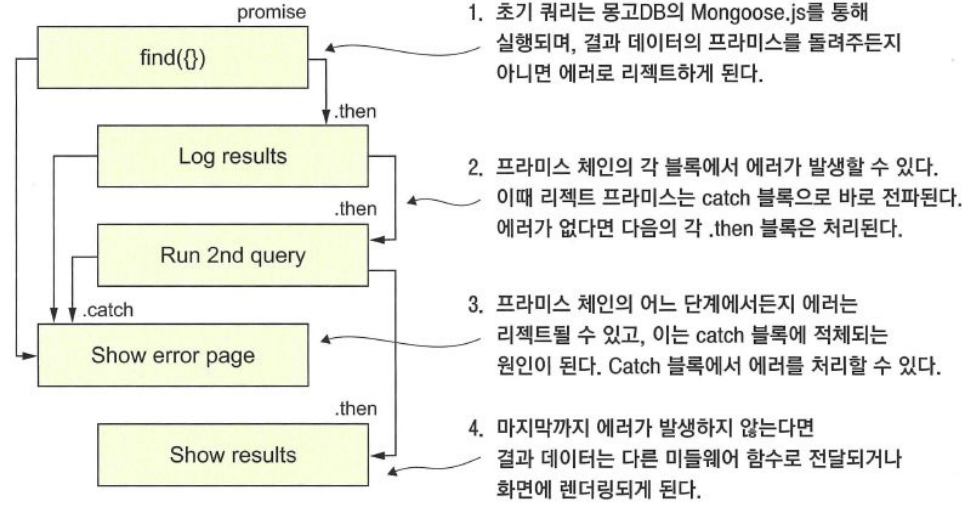


그림 15.4 Mongoose.js에서의 프라미스 체인

프라미스가 완료되면 Express.js에 있는 후속 미들웨어의 사용을 위해 `next`를 호출한다. 그리고 `then` 메소드에 연결해 프라미스에게 데이터베이스의 응답 후 바로 이 작업을 수행함을 알린다. 이 `then` 블록은 뷰를 렌더링하는 곳이다.

15.3 Mongoose로 프라미스 사용

Listing 15.7 subscribersController.js에서 모든 구독자를 가져오기 위한 프라미스 사용

```
exports.getAllSubscribers = (req, res) => {
  Subscriber.find({})
    .exec()
    .then((subscribers) => {
      res.render("subscribers", {
        subscribers: subscribers
      });
    })
    .catch((error) => {
      console.log(error.message);
      return [];
    })
    .then(() => {
      console.log("promise complete");
    });
};
```

getAllSubscribers 액션의 재작성

find 쿼리로부터의 프라미스 리턴

저장된 데이터를 다음 then 코드 블록에 전달

데이터베이스로부터 결과 제공

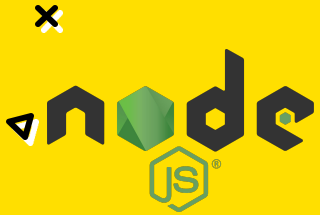
프라미스에서 리젝트된 에러들을 캐치

프라미스 체인의 종료와 메시지 로깅

[노트] then은 프라미스 환경에서만 사용된다, 그러고 next는 미들웨어 함수에서 사용된다. 만약 Listing 15 .7과 같이 둘 다 사용된다면 then 의 처리를 위해 프라미스를 기다리며 그 후에 next를 호출해 다른 미들웨어 함수로 넘어간다.

then 체인은 원하는 만큼 추가할 수 있다다.

15.3 Mongoose로 프라미스 사용



또한 Listing 15.8에서 보는 바와 같이 프라미스를 사용하기 위한 saveSubscriber에서의 save를 수정할 수 있다.

Lisiting 15.8 subscribersController.js에서 프라미스를 사용하기 위한 saveSubscriber의 수정

```
newSubscriber.save()
  .then(result => {
    res.render("thanks");
  })
  .catch(error => {
    if (error) res.send(error);
  });
```

프라미스로 새 구독자 저장

15.3 Mongoose로 프라미스 사용

Listing 15.9 seed.js에서의 새로운 데이터 생성

```
const mongoose = require("mongoose"),
  Subscriber = require("./models/subscriber");

mongoose.connect(
  "mongodb:// localhost:27017/recipe_db",
  { useNewUrlParser: true }
);

mongoose.connection;

var contacts = [
  {
    name: "Jon Wexler",
    email: "jon@jonwexler.com",
    zipCode: 10016
  },
  {
    name: "Chef Eggplant",
    email: "eggplant@recipeapp.com",
    zipCode: 20331
  },
  {
    name: "Professor Souffle",
    email: "souffle@recipeapp.com",
    zipCode: 19103
  }
];
```

데이터베이스 연결 설정

마지막으로 구독자 정보를 연락처 폼을 통해 하나하나 올리는 것 대신 한꺼번에 올리려면 이를 위한 모듈을 따로 만들 수 있다. 이 파일은 로컬 데이터베이스에 접속해 생성을 위한 구독자 정보 배열을 루프를 돌면서 처리한다.

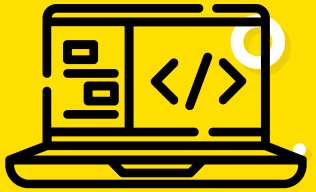
```
Subscriber.deleteMany()
  .exec()
  .then(() => {
    console.log("Subscriber data is empty!");
  });

var commands = [];

contacts.forEach((c) => {
    commands.push(Subscriber.create({
      name: c.name,
      email: c.email
    }));
});

Promise.all(commands)
  .then(r => {
    console.log(JSON.stringify(r));
    mongoose.connection.close();
  })
  .catch(error => {
    console.log(`ERROR: ${error}`);
  });
```

기존 데이터 제거

프라미스 생성을 위한 구독자 객체 루프

프라미스 생성 후 로깅 작업

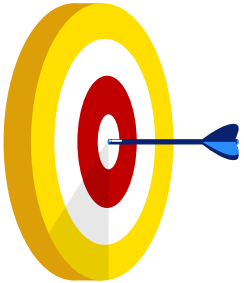
이 파일을 이어지는 각 장에서 실행해 비어 있거나 일관성이 깨진 데이터베이스를 피할 수 있다.



Coding!

Listing 15.1 - 15.9

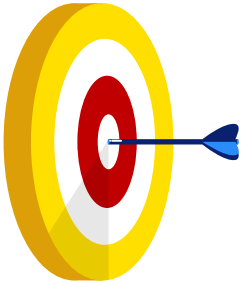
p. 213-226



퀵 체크 15.1

데이터를 뷰로 전달하는 모듈은 무엇인가?

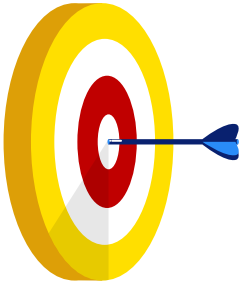
컨트롤러로부터 뷰로 데이터를 전달할 수 있다.
subscribersController.js 파일에서 구독자 배열을
렌더링된 subscribers.ejs를 통해 전달한다.



퀵 체크 15.2

폼에서 데이터를 처리하기 위한
Express.js의 추가 패키지는 무엇인가?

요청의 본문(body) 부분을 쉽게 파싱하려면 body-parser 패키지의 도움이 필요하다. 이 모듈은 유입된 요청과 Express.js 사이에서 미들웨어 역할을 한다.



퀵 체크 15.3

참 또는 거짓! Mongoose 쿼리에서 exec를 사용하는 것은 새로운 프라미스를 돌려주는 쿼리를 사용하는 것과 같다.

참이다. exec는 만약 프라미스가 Mongoose 설정 시 같이 실정됐다면 쿼리를 수행하고 프라미스를 돌려주도록 설계돼 있다.

과제 타임!

한번 해 보자!~