Unit 4 17-20

사용자 모델 제작

UT-NodeJS / 04.28.2023

ut-nodejs.github.io

Contents / 내용

17. 데이터 모델의 개선

- 모델에 대한 유효성 평가 추가
- 모델을 위한 정적 인스턴스 메소드의 생성
- REPL에서의 모델 테스팅
- 다중 모델에서의 데이터 조합

18. 사용자 모델의 구현

- 사용자 모델과 연결 모델 생성
- 가상 속성 사용
- 사용자 모델에서 CRUD의 구현
- 데이터베이스 내 모든 사용자들의 열람을 위한 인덱스 페이지 만들기

19. 모델의 생성과 읽기

- 모델 생성 폼의 구축
- 브라우저에서 데이터베이스로 사용자 저장
- 뷰에서 연관 모델 출력

20. 모델의 업데이트와 삭제

- 모델 수정 폼의 구축
- 데이터베이스에서 사용자 레코드 업데이트
- 사용자 레코드 삭제

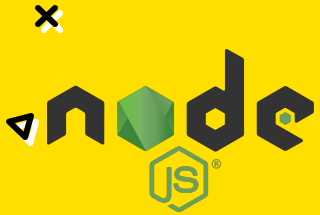
17

데이터 모델의 개선

모델에 대한 유효성 평가 추가
모델을 위한 정적 인스턴스 메소드의 생성
REPL에서의 모델 테스팅
다중 모델에서의 데이터 조합

p. 239-256

17. 데이터 모델의 개성



책과 조금 다른 순서

1. Add **validations** to the Subscriber model in models/Subscriber.js.
   models/Subscriber.js의 Subscriber 모델에 **유효성 평가자**를 추가합니다.

2. Add **instance methods** to the Subscriber model in models/Subscriber.js.
   models/Subscriber.js의 Subscriber 모델에 **인스턴스 메서드**를 추가합니다.

3. Create a **Course model** in models/Course.js.
   models/Course.js에 **Course 모델**을 생성합니다.

4. Set up **associations** between the Subscriber and Course models in models/Subscriber.js.
   models/Subscriber.js의 Subscriber와 Course 모델 **사이의 연관 관계**를 설정합니다.

5. **Seed** the database with courses in seedCourses.js.
   seedCourses.js에서 데이터베이스에 코스를 씁니다. **(REPL 대신)**

6. **Seed** the database with subscribers in seedSubscribers.js.
   seedSubscribers.js에서 데이터베이스에 구독자를 씁니다. **(REPL 대신)**

7. ~~Randomly associate subscribers with courses in and populate subscribers' courses.~~
   ~~구독자를 코스에 무작위로 연결하고 구독자의 코스를 채웁니다.~~ **(나중에)**

17.1 모델에 유효성 평가 추가

models/Subscriber.js의 **Subscriber** 모델에 유효성 평가자를 추가합니다.

Listing 17.1 subscriber.js에서의 구독자 스키마 정의

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

이름, 이메일, 우편번호를 포함한 "subscriberSchema" 정의

지금까지 Mongoose로 모델을 만들어왔다. 생성한 모델은 몽고DB 안에서 도큐먼트로 표현된 데이터의 추상화 결과다. 이 추상화 덕분에 Mongoose 스키마를 사용해 어떻게 데이터가 보이고 동작하게 돼야 하는지 청사진을 만들 수 있다.

- 하지만 속성들이 복제 가능한지 사이즈 제한(예를 들어 우편번호는 15자리까지 저장 가능) 이 있는지, 이런 특성들이 데이터베이스로 저장 시 요구되는지 등은 알려주지 않는다. 만일 구독지들이 빈칸으로 양식을 제출한다면 이 스키마로는 이를 막을 수 없다.

17.1 모델에 유효성 평가 추가

models/Subscriber.js의 **Subscriber** 모델에 유효성 평가자를 추가합니다.

스키마 타입

Mongoose는 스키마 내에서 정의할 수 있는 몇 가지 데이터 타입을 정의한다. 보통 이를 스키마 타입이라고 하며 JavaScript의 데이터 타입과 비슷하다. 다만 차이가 있다면 일반 JavaScript 데이터 타입은 갖고 있지 않은 Mongoose 라이브러리와의 특정 연관관계가 있다는 것이다. 알아둬야 할 스키마 타입은 다음과 같다.

- String: Boolean과 Number처럼 간단하다. String 속성은 (null이나 undefined가 아닌) JavaScript의 String 타입으로 제공된 데이터를 저장하겠다는 의미다.
- Date: 날짜는 데이터 도큐먼트에서 자주 사용되며, 언제 데이터가 저장되거나 수정됐는지, 또는 모델과 관련된 뭔가 발생할 때 이를 통해 알 수 있다. 이 타입은 JavaScript의 Date 객체를 받을 수 있다.
- Array: Array 타입을 사용하면 리스트 아이템을 속성에 저장할 수 있다. Array 타입 지정 시 이름 대신 대괄호([])로 묶은 배열 리터럴을 사용한다.
- Mixed: 이 타입은 JavaScript 객체와 매우 유사하며 키-값 쌍을 모델에 저장한다 .Mixed 타입을 사용하려면 mongoose.Schema.Types.Mixed를 정의해야 한다.
- ObjectId: 몽고DB 데이터베이스에서 각 도큐먼트의 ObjectId와 유사하게 이 타입은 이 객체를 참조한다. 이 타입은 모델 상호간 조합이 일어날 때 특히 중요하다. 이 타입을 사용하려면 mongoose.Schema.Types.ObjectId를 정의해야 한다.

기존 모델의 개선을 위해 Mongoose 의 유효성 평가자Validator를 추가한다. 유효성 평가자는 모델 속성에 적용되며 유효성 검증에 실패하면 데이터베이스 저장 등을 하지 못하게 한다.

스키마 타입:

- String
- Boolean
- Number
- Date
- Array
- Mixed (Object)
- ObjectId

17.1 모델에 유효성 평가 추가

models/Subscriber.js의 **Subscriber** 모델에 유효성 평가자를 추가합니다.

Listing 17.2 Subscriber.js에서 구독자 스키마에 유효성 평가자 추가

```
const mongoose = require("mongoose");

const subscriberSchema = new mongoose.Schema({
  name: {                                    // ← name 속성 요청
    type: String,
    required: true
  },
  email: {                                   // ← email 속성 요청과 lowercase 속성 추가
    type: String,
    required: true,
    lowercase: true,
    unique: true
  },
  zipCode: {                                 // ← 사용자 정의 에러 메시지로 우편번호 속성 설정
    type: Number,
    min: [10000, "Zip code too short"],
    max: 99999
  }
});
```

[노트] email 속성에서 사용된 unique 옵션은 유효성 평가자는 아니며 Mongoose 스키마의 헬퍼에 가깝다. 헬퍼는 경우에 따라 유효성 평가자와 같이 동작을 하는 메소드 같은 것이다.

17.1 모델에 유효성 평가 추가

models/Subscriber.js의 **Subscriber** 모델에 인스턴스 메서드를 추가합니다.

Listing 17.3 subscriber.js에서 스키마에 인스턴스 메소드 추가

```
구독자의 Full info을 구하기
위한 인스턴스 메소드 추가

subscriberSchema.methods.getInfo = function() {
  return `Name: ${this.name} Email: ${this.email} Zip Code:
  ${this.zipCode}`;
};

subscriberSchema.methods.findLocalSubscribers = function() {
  return this.model("Subscriber")
    .find({zipCode: this.zipCode})
    .exec();
};

                                  같은 우편번호를 가지는
                                  구독자를 구하기 위한
                                  인스턴스 메소드 추가

find 메소드를 사용하기 위한
구독자 모델에 액세스
```

instance 메소드가 Subscriber 모델의 인스턴스 (Mongoose 도큐멘트)에서 운용 되며 이는 subscriberSchema.methods 에 의해 정의된다. Static 메소드는 많은 Subscriber 인스턴스들에 연관된 쿼리에 사용되며 이는 subscriberSchema.statics 에 의해 정의된다.

exec 에서 비동기 콜백을 여기에 추가해야 하는 것 대신에 프라미스를 되돌려 받게 된다.

[주의] 앞서 기술했지만, Mongoose를 갖고 메소드를 사용하면 ES6의 회실표 함수를 완전히 사용할 수 없다. Mongoose는 this를 사용하며 여기서는 화실묘 힘수가 제거됐다. 하지만 함수 내부에서는 다시 ES6를 사용할 수 있다.

17.1 모델에 유효성 평가 추가

models/Subscriber.js의 **Subscriber** 모델에 인스턴스 메서드를 추가합니다.

Mongoose 쿼리에 대한 자세한 정보는 http://mongoosejs.com/ docs/ queries.html을 방문하기바란다.

표 17.1 Mongoose 쿼리

쿼리	설명
find	쿼리 매개변수에 맞는 레코드의 배열을 돌려준다. 이름이 "Jon"이라는 구독자를 검색하려면 Subscriber.find({name: "Jon"}으로 수행하면 된다.
findOne	배열 형태의 데이터 대신에 단일 레코드를 돌려준다. Subscriber.findOne({name: "Jon"}을 수행하면 결과들은 하나의 도큐먼트로 담겨 돌려받게 된다.
findById	ObjectId를 키로 데이터베이스에 쿼리를 할 수 있게 한다. 이 쿼리는 데이터베이스에 존재하는 데이터를 수정하는 데 유용하다. 구독자의 ObjectId를 알고 있다고 가정하면 이를 찾는 명령은 Subscriber.findById("598695b29ff27740c5715265")처럼 된다.
remove	모든 도큐먼트를 삭제하기 위해 Subscriber.remove({}) 구문을 사용해 데이터베이스 내 도큐먼트를 삭제하게 한다. 이 쿼리를 사용할 때는 주의를 요한다. subscriber.remove({})와 같이 특정 인스턴스의 삭제도 가능하다.

[노트] 이들 각 쿼리는 프라미스를 돌려준다 따라서 then과 catch를 사용해 결과 데이터나 에러 처리를 해야 한다.

17.2 하지 않을 내용 (REPL)

17.2 REPL에서의 테스팅 모델

Subscriber 모델을 사용해 데이터베이스와 연동을 시작하기 위해 새로운 터미널 창에서 node를 입력, REPL 모드로 들어가 Listing 17.4의 코드를 입력한다.

[탑] 이 절의 코드들은 재사용이 가능하다. REPL 코드들을 repl.js로 만들고 프로젝트 디렉토리에 추가한다. 다음번에 REPL 실행 시 해당 파일의 내용이 환경으로 로딩된다. Node.js는 비동기로 실행됨을 기억하라. 만일 하나의 명령으로 한 개의 레코드를 생성하고 이어서 이 레코드를 찾는 쿼리를 실행한다면 이 둘은 거의 동시에 실행될 가능성이 있다. 이런 경우 오류를 회피하려면 명령을 각기 시간을 둬 명령어를 따로 수행하거나 네스트 쿼리 (nest query) 형태로 then 블록을 만들어 수행한다.

,··

Listing 17.4 REPL 터미널에서의 구독자 모델 설정
Listing 17.5 REPL 터미널에서의 **Mongoose** 쿼리 모델 메소드의 테스트

[노트] 이메일은 고웃값이어야 하므로 동일한 값으로 중복돼 입력하면 키 중복 에러가 발생할 것이다. 이런 경우에는 Subscriber.remove({})를 수행해 데이터베이스 내 모든 데이터를 삭제할 수 있다.

17.3 모델 조합의 생성

models/Course.js에 **Course** 모델을 생성합니다.

Listing 17.6 course.js에서 새로운 스키마와 모델의 생성

```
const mongoose = require("mongoose");

const courseSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
    unique: true
  },
  description: {
    type: String,
    required: true
  },
  items: [],
  zipCode: {
    type: Number,
    min: [10000, "Zip code too short"],
    max: 99999
  }
});

module.exports = mongoose.model("Course", courseSchema);
```

강좌 스키마에 속성 추가

Course 모델에 subscribers 속성을 추가할 수 있는데 Course는 구독자의 ObjectId를 통해 구독자 레퍼런스를 저장하는 모델이며 이 ObjectId는 몽고DB 에서 가져온다.

기술적으로는 모델 간 참조는 필요 사항은 아니지만 하나 정도 참조하는 것은 괜찮다. 따라서 Subscriber 모델과 조합을 만든다.

다시 subscriber.js로 되돌아가 다음 속성을 subscriberSchema 에 추가한다.

```
courses: [{type: mongoose.Schema.Types.ObjectId, ref: "Course"}]
```

[노트] 구독자와 강좌 간에 많은 연관성을 가질 수 있는 가능성을 반영해 해당 속성의 이름이 복수형임을 알 수 있다.

17.3 모델 조합의 생성

models/Course.js에 **Course** 모델을 생성합니다.

여러분이 만일 관계형 DB에 백그라운드가 있다면 그림 17.2와 같은 방법에 더 친숙할지도모르겠다. 하지만 여기서는 도큐먼트 베이스의 데이터베이스로 작업을 하기 때문에 테이블이 존재하지 않는다.

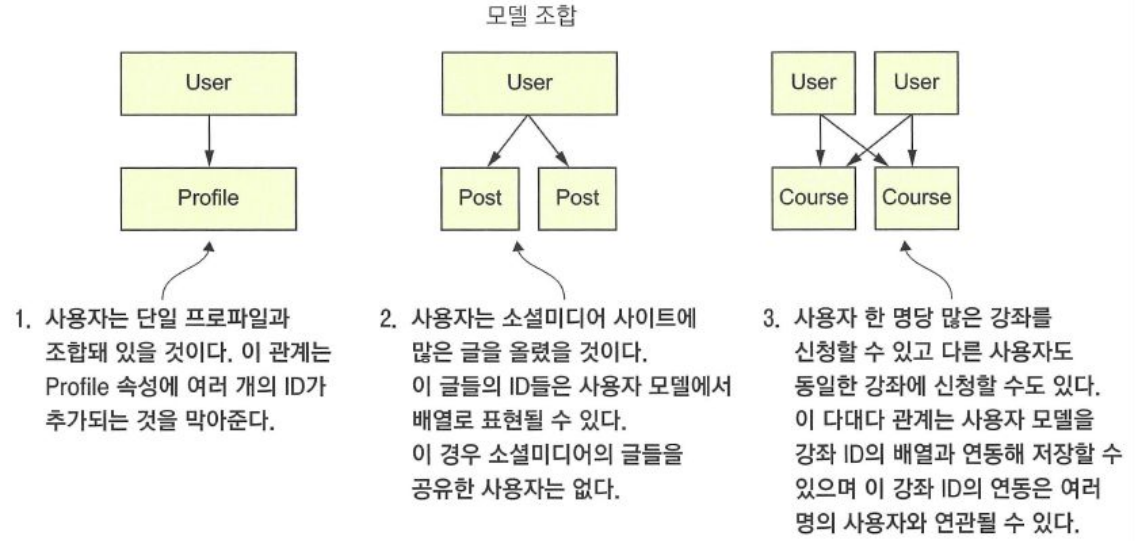


그림 17.2 관계형 데이터베이스 조합

17.3 모델 조합의 생성

models/Course.js에 **Course** 모델을 생성합니다.

표 17.2 데이터 관계 종류

관계	설명
일대일	하나의 모델이 다른 모델 하나와 관계를 맺을 수 있는 경우. 예를 들어 User와 하나의 Profile과의 관계와 같으며 하나의 사용자는 하나의 프로파일만 존재한다.
일대다	하나의 모델이 다른 여러 모델들과 관계를 가질 수 있는 경우. 하지만 상대 모델은 하나의 모델하고만 관계를 가질 수 있다. 이 관계는 Company와 Employee의 관계와 같다. 이 예제에서 직원(Employee)은 하나의 회사(Company)에 관계돼 있으며, 회사는 많은 직원들과 관계를 갖고 있다.
다대다	하나의 모델에서 다수의 인스턴스가 다른 다수의 모델과 관계를 가질 수 있으며 그 반대 방향도 가능하다. 다수의 Theatre 인스턴스는 같은 Movie 인스턴스를 상영할 수 있고 각 Movie는 다수의 Theatre 이어질 수 있다. 일반적으로 관계형 데이터베이스에서는 서로 레코드끼리 매핑하기 위해 table join이 사용된다.

하나의 주문 대 하나의 지불

한 명의 사용자 대 여러 장의 사진

다수의 클래스 대 다수의 등록 학생

만일 두 개의 모델이 어떤 방식에 의해 연결됐다면 연결된 모델 이름을 속성으로 추가한다. 여기에서 type은 Schema.Types.ObjectId가 되고 ref 속성인 연결된 모델 이름으로 설정되며 Schema는 mongoose.Schema가 된다. pictures: [{type: Schema.Types.ObjectId, ref: "Picture"}] 는 사용자와 사진의 다대다 관계를 나타내는 코드다.

17.3 모델 조합의 생성

models/Subscriber.js의 **Subscriber**와 **Course** 모델 사이의 연관 관계를 설정합니다.

개별 모델의 두 인스턴스를 실제로 연결시키려면 JavaScript 할당 연산자를 사용한다.

구독자를 변수 subscriber1 에 할당했고 강좌 인스턴스는 course1로 나타낸다고 가정하자.
이 두 인스턴스를 연결시키기 위해 구독자 모델은 다수의 강좌와 연결이 가능하다고 가정하고 **subscriber1.courses.push(course1)** 를 실행시켜야 한다. subscriber1.courses는 배열이기 때문에 push 메소드를 써서 새 강좌를 추가한다.

다른 방법으로 모든 강좌 객체를 사용하는 것 대신 ObjectId를 subscriber.courses로 푸시할 수도 있다. course10이 ObjectId "5c23mdsnn3k43k2kuu" 를 갖고 있다고 가정하면 코드는 **subscriber1.courses.push("5c23mdsnn3k43k2kuu")** 처럼 될 것이다.

강좌 데이터를 구독자로부터 검색하려면 강좌의 ObjectID와 Course 모델에 대한 쿼리를 사용하거나 연결된 강좌의 콘텐츠에 따른 구독자 쿼리를 위한 populate 메소드를 사용할 수 있다. subscriber1 몽고DB 도큐먼트 안에는 course1 도큐먼트가 내장될 수도 있다. 결과적으로 연결된 모델의 ObjectID들만 얻을 수 있게 된다.

17.4 연결 모델로부터의 데이터 포퓰레이팅

models/Subscriber.js의 **Subscriber**와 **Course** 모델 사이의 연관 관계를 설정합니다.

포플레이션 Population은 모델과 연결돼 있는 모든 도큐먼트를 갖고 쿼리 결과에 추가하게 해주는 메소드다. 쿼리 결과를 populate하면 연결된 도큐먼트의 ObjectId들을 도큐먼트의 콘텐츠로 바꿔준다. 이 작업은 보통 모델 쿼리 후에 이어서 해줘야 한다.

예를 들어 Subscriber.populate(subscriber, "course") 는 subscriber 객체와 연결된 모든 강좌를 가져와 이들의 ObjectId를 구독자의 courses 배열에 있는 모든 Course 도큐먼트로 바꿔준다.

[노트] http://mongoosejs.com/docs/populate.html에서 유용한 예제를 찾을 수 있다.

(나중에)

17.4 하지 않을 내용 (REPL)

17.4 연결 모델로부터의 데이터 포퓰레이팅

[노트] 이 예제를 위해 에러 처리를 위한 errors를 사용하지 않아 코드가 짧다. 하지만 테스트 시에는 이런 에러 처리를 하려 할 수도 있다. catch(error => console.log(error.message))와 같은 간단한 코드도 디버깅 시 많은 도움이 된다.

Listing 17.7 REPL 터미널을 사용해 연결 모델 테스팅하기
Listing 17.8 터미널에서 **REPL**로부터의 콘솔 로그 결과
Listing 17.9 REPL.js에서의 명령 수행

명령 순서들은 어떻게 REPL 코드가 코드를 테스팅하는지 보여주고 있다.

[팁] Mongoose와 몽고 DB로의 쿼리는 좀 복잡해질 수 있다. 동일한 쿼리를 Mongoose로 해보고 몽고 DB 쿼리 문법과 연계시키는 연습을 하는 것을 권한다.

seedCourses.js에서 데이터베이스에 코스를 씁니다. **(REPL 대신)**

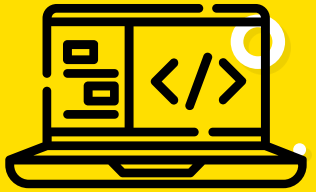
1. Delete all previous data.
   이전 데이터 모두 삭제
2. Set a timeout to allow the database to be cleared.
   데이터베이스가 지워지는 것을 기다리기 위해 타임아웃 설정
3. Create a promise for each **courses** object.
   코스 객체마다 프라미스 생성.
4. Use Promise.all() to wait for all promises to resolve.
   모든 프라미스가 해결될 때까지 기다리기 위해 Promise.all() 사용.
5. Close the connection to the database.
   데이터베이스 연결 닫기.

```
var courses = [
  {
    _id: "nodejs101",
    title: "Node.js 101",
    description: "웹 개발로 알아보는 백엔드 자바스크립트의 이해",
    price: 20000,
    courseImg:
      "",
  },
  {
    _id: "htmlcssjs101",
    title: "HTML, CSS, JS 101",
    description: "웹 개발의 시작은 여기서부터",
    price: 10000,
    courseImg:
      "",
  },
  {
    _id: "python101",
    title: "Python 101",
    description: "파이썬 기초 문법을 배워봅시다",
    price: 10000,
    courseImg:
      "",
  },
  {
    _id: "aiml101",
    title: "AI + ML 101",
    description: "인공지능과 머신러닝의 기초를 배워봅시다",
    price: 30000,
    courseImg:
      "",
  },
  {
    _id: "react101",
    title: "React 101",
    description: "리액트를 배워봅시다",
    price: 15000,
    courseImg:
      "",
  },
];
```

seedSubscribers.js에서 데이터베이스에 구독자를 씁니다. **(REPL 대신)**

1. Delete all previous data.
   이전 데이터 모두 삭제
2. Set a timeout to allow the database to be cleared.
   데이터베이스가 지워지는 것을 기다리기 위해 타임아웃 설정
3. Create a promise for each **subscriber** object.
   구독자 객체마다 프라미스 생성.
4. Use Promise.all() to wait for all promises to resolve.
   모든 프라미스가 해결될 때까지 기다리기 위해 Promise.all() 사용.
5. Close the connection to the database.
   데이터베이스 연결 닫기.

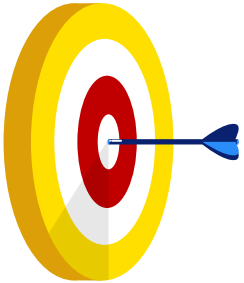
```
var subscribers = [
  {
    name: "Yoo Jae-suk",
    email: "yjs@running.com",
    phoneNumber: "",
    newsletter: true,
    courses: ['nodejs101', 'htmlcssjs101', 'python101', 'aiml101', 'react101'],
    profileImg:
      "https://newsimg-hams.hankookilbo.com/2022/01/21/f96adb47-e8b1-43fe-aa16-f2043012cbec.jpg",
  },
  {
    name: "Haha",
    email: "hhh@running.com",
    phoneNumber: "010-????-????",
    newsletter: true,
    courses: [],
    profileImg:
      "https://image.ytn.co.kr/general/jpg/2020/0719/202007191609185642_t.jpg",
  },
  {
    name: "Jee Seok-jin",
    email: "bignose@running.com",
    phoneNumber: "010-1234-5678",
    newsletter: true,
    courses: [],
    profileImg:
      "https://cdnimg.melon.co.kr/cm/artistcrop/images/000/11/114/11114_500.jpg/melon/optimize/90",
  },
  {
    name: "Kim Jong-kook",
    email: "gymjk@running.com",
    phoneNumber: "010-7777-7777",
    newsletter: true,
    courses: [],
    profileImg:
      "https://newsimg.hankookilbo.com/cms/articlerelease/2021/06/17/b8223d84-e72f-42a3-a984-0ae23ba70aa4.jpg",
  },
  {
    name: "Song Ji-hyo",
    email: "mungjh@running.com",
    phoneNumber: "",
    newsletter: true,
    courses: [],
    profileImg:
      "https://image.ajunews.com//content/image/2021/08/18/20210818194549250993.jpg",
  },
```



Coding!

Listing 17.2, 17.3 & 17.6

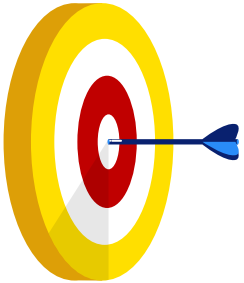
p. 242, 243, 249-250



퀵 체크 17.1

Mongoose로 프라미스를 사용할 때 쿼리는 무엇을 돌려주는가?

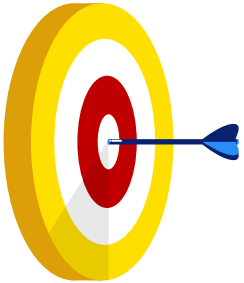
Mongoose로 프라미스를 사용하면 데이터베이스 쿼리 결과로 프라미스를 돌려받게 된다. 프라미스로 결과를 받는다는 것은 결과나 에러 처리에서 비동기 쿼리의 타이밍 이슈를 신경 쓰지 않아도 되는 것을 의미한다.



퀵 체크 17.2

왜 REPL에서 코드를 테스트하는 데 데이터베이스 커넥션과 Mongoose가 필요할까?

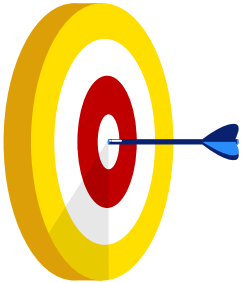
데이터베이스와 연동하기 위한 뷰가 완성되기 전까지, REPL은 모델에서 CRUD를 수행하기 위한 좋은 도구다. 하지만 수행 테스트 시 모듈들을 요청해야 하는데 이 요청을 통해 REPL 환경에서 어떤 데이터베이스에 저장해야 하는지와 어떤 Subscriber 모델이 테스트돼야 하는지 알 수 있기 때문이다.



퀵 체크 17.3

하나의 인스턴스에 연결된 모델과 다수의
인스턴스에 연결된 모델을 어떻게 구별하는가?

모델 스키마를 정의할 때, 연결된 모델을 대괄호로 둘러암으로 일대다 관계를 정의할 수 있다. 대괄호는 연결된 레코드의 배열을 의미한다. 대괄호가 없으면 관계는 일대일 관계가 된다.



퀵 체크 17.4

왜 모든 쿼리상의 관계 모델을
populate하지 않는 것일까?

populate 메소드는 레코드를 위해 모든 연결된 데이터를 모으는 작업에 유용하다. 하지만 이를 남용하면 처리에 오버헤드가 걸릴 수 있으며 레코드를 위한 저장 공간도 만만치 않다. 일반적으로 연결된 특정 레코드의 세부 사항까지 액세스할 필요가 없다면 populate를 쓸 필요는 없다.

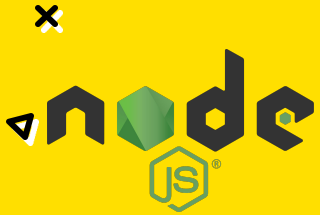
18

사용자 모델의 구현

사용자 모델과 연결 모델 생성
가상 속성 사용
사용자 모델에서 CRUD의 구현
데이터베이스 내 모든 사용자들의 열람을 위한
인덱스 페이지 만들기

p. 257-273

18. 사용자 모델의 구현



책과 조금 다른 순서

1. Build a **User model** in models/User.js and a add virtual attribute.
   models/User.js에 **User 모델**을 생성하고 가상 속성을 추가합니다.

2. **Seed** the database with users in seedUsers.js.
   seedUsers.js에서 데이터베이스에 사용자를 씁니다. **(REPL 대신)**

3. **Associate** subscribers with users.
   구독자를 사용자와 연결합니다. **(링크 나중에)**

4. Create a **userController.js** in the controllers folder.
   controllers 폴더에 **userController.js**를 생성합니다.

5. Create a **users folder** in the views folder and add **index.ejs** to it.
   views 폴더에 **users 폴더**를 생성하고 **index.ejs**를 추가합니다.

18.1 사용자 모델 작성

models/User.js에 **User** 모델을 생성하고 가상 속성을 추가합니다.

시용자 모델은 데이터베이스에서 CRUD 기능으로 대부분 현대 애플리케이션들이 갖고 있다.

원치 않는 데이터를 데이터베이스에 입력하는 것을 방지하는 모델을 갖게 됐고 이제부터 동일하게 주요 모델들에 이를 적용해야 한다.

구독자와 강좌에 대한 연동도 필요하다(예를 들어 이전 구독자가 사용자로 등록하기로 했다면 이 두 모델의 연동이 필요하다).

[주의] 여기에서는 비밀번호를 일반 텍스트로 저장할 것이다. 이런 방법은 보안상 권장되지 않으며, 5부에서 관련 내용을 다룬다.

사용자는 단일 구독자의 계정과 연결된다. 새로운 속성 세트인 createdAt과 updatedAt은 사용자
인스턴스의 생성 및 모델에서의 변경 발생 시간을 포퓰레이트한다. timestamp 속성은 Mongoose에게 createdAt과 updatedAt 이 포함됐다는 것을 알려주는데 이는 어떻게 그리고 언제 데이터가 변경됐는지 기록할 때 유용하다.

[노트] Mongoose Schema 객체에서 객체 소멸 (object destruct) 의 사용에 주목하자. {Schema}는 Mongoose 의 Schema 객체를 동일한 이름의 상수로 할당한다. 나중에 이 새로운 형식을 다른 모델에 적용할 것이다.

18.1 사용자 모델 작성

models/User.js에 **User** 모델을 생성하고 가상 속성을 추가합니다.

name 속성이 단일 String 속성인 것 대신에, 여기서는 first와 last 2개의 객체로 구성돼있다. 이렇게 분리하면 성 또는 이름으로만 사용자를 나타내야할 때 유용하다. 이 속성이 데이터베이스에 저장될 때 공백 없이 저장되도록 trim 속성이 true로 설정된 점에 주목하자.

이름first과 성 last 이 때로는 한 줄로 유용할 수 있다는 것을 감안할 때, Mongoose의 가상 속성Virtual attribute을 이용해 각 인스턴스의 해당 데이터 저장을 할 수 있다. 가상 속성(계산된 속성 Computed attribute 이라고도 한다)은 정규 스키마 속성과 유사하지만 데이터베이스에 저장되지는 않는다. 이 값은 user.fullName으로 검색할 수 있다.

Listing 18.2 user.js에서 사용자 모델에 가상 속성 추가

```
userSchema.virtual("fullName")
  .get(function() {
    return `${this.name.first} ${this.name.last}`;
  });

module.exports = mongoose.model("User", userSchema);
```

사용자의 풀 네임을 얻기 위한 가상 속성 추가

18.1 사용자 모델 작성

models/User.js에 **User** 모델을 생성하고 가상 속성을 추가합니다.

Listing 18.1 user.js에서 사용자 모델 생성

```
const mongoose = require("mongoose"),
  {Schema} = mongoose,

  userSchema = new Schema({
  name: {
    first: {
      type: String,
      trim: true
    },
    last: {
      type: String,
      trim: true
    }
  },
  email: {
    type: String,
    required: true,
    lowercase: true,
    unique: true
  },
```

사용자 스키마 생성

name 속성에 이름(first)과 성(last) 추가

```
  zipCode: {
    type: Number,
    min: [1000, "Zip code too short"],
    max: 99999
  },
  password: {
    type: String,
    required: true
  },
  courses: [{type: Schema.Types.ObjectId, ref: "Course"}],
  subscribedAccount: {type: Schema.Types.ObjectId, ref:
  "Subscriber"}
}, {
  timestamps: true
});
```

비밀번호 속성 추가

사용자와 강좌를 연결 시켜주기 위한 강좌 속성 추가

subscribedAccount를 사용자와 구독자를 연결하기 위해 추가

timestamps 속성을 추가해 createAt 및 updateAt 시간 기록

18.1 하지 않을 내용 (REPL)

Listing 18.3 REPL 터미널에서 새로운 사용자 생성
Listing 18.4 터미널에서 사용자 객체의 저장 결과
Listing 18.5 REPL 터미널에서 구독자와 사용자의 연결
Listing 18.6 REPL 터미널에서 사용자와 구독자의 연결

[노트] 메일 어드레스의 충돌 관련 에러 메시지가 출력 된다면 데이터베이스에 동일한 이메일을 갖고 있는 사용자가 있다는 의미다(이는 우리가 세운 스키마에 의하면 있어서는 안 될 일이다). 이런 상황을 우회하기 위해 다른 이메일을 사용하든지 아니면 find() 메소드를 생성 대선에 사용한다. 이는

```
User.findOne({email: "jon@jonwexler.com"})
        .then(u=> testUser = u)
        .catch(e => console. log(e.message));
```

처럼 사용하면 된다.

이 사용자로부터 동일한 이메일을 갖고 있는 구독자와 연결할 수 있는 정보를 사용할 수 있게 됐다.

[노트] 데이터베이스 쿼리의 테스트를 위해 REPL 환경으로 옮겼다. 따라서 더 이상 필요치 않은 subscriber의 요청 부분은 삭제 가능하다.

seedUsers.js에서 데이터베이스에 사용자를 씁니다. **(REPL 대신)**

1. Delete all previous data.
   이전 데이터 모두 삭제
2. Set a timeout to allow the database to be cleared.
   데이터베이스가 지워지는 것을 기다리기 위해 타임아웃 설정
3. Create a promise for each **user** object.
   사용자 객체마다 프라미스 생성.
4. Use Promise.all() to wait for all promises to resolve.
   모든 프라미스가 해결될 때까지 기다리기 위해 Promise.all() 사용.
5. Close the connection to the database.
   데이터베이스 연결 닫기.

```
var users = [
  {
    name: {
      first: "Jae-suk",
      last: "Yoo",
    },
    email: "yjs@running.com",
    username: "grasshopper",
    phoneNumber: "",
    password: "1234",
    courses: [],
    profileImg:
      "https://newsimg-hams.hankookilbo.com/2022/01/21/f96adb47-e8b1-43fe-aa16-f2043012cbec.jpg",
  },
  {
    name: {
      first: "Haha",
      last: "",
    },
    email: "hhh@running.com",
    username: "haroro",
    phoneNumber: "010-????-????",
    password: "1234",
    courses: [],
    profileImg:
      "https://image.ytn.co.kr/general/jpg/2020/0719/202007191609185642_t.jpg",
  },
  {
    name: {
      first: "Myeong-su",
      last: "Park",
    },
    email: "pms@challenge.com",
    username: "mspark",
    phoneNumber: "010-1234-5678",
    password: "1234",
    courses: [],
    profileImg: "",
  },
```

18.2 모델에 CRUD 메소드 붙이기

구독자를 사용자와 연결합니다. (링크 나중에)

사용자, 구독자 그리고 그룹 모델에서 다뤄야 하는 다음 단계를 논의한다. 이 3가지 모델은 모두 스키마와 REPL상에서 연관성을 갖고 있으나, 이를 브라우저에서 보여줄 필요가 있을 것이다. 좀 더 구체적으로 말하면 각 모델에 관해 사이트 관리자로의 데이터 관리 및 사용자들에게 자신만의 계정을 만들기를 원할 수 있다.

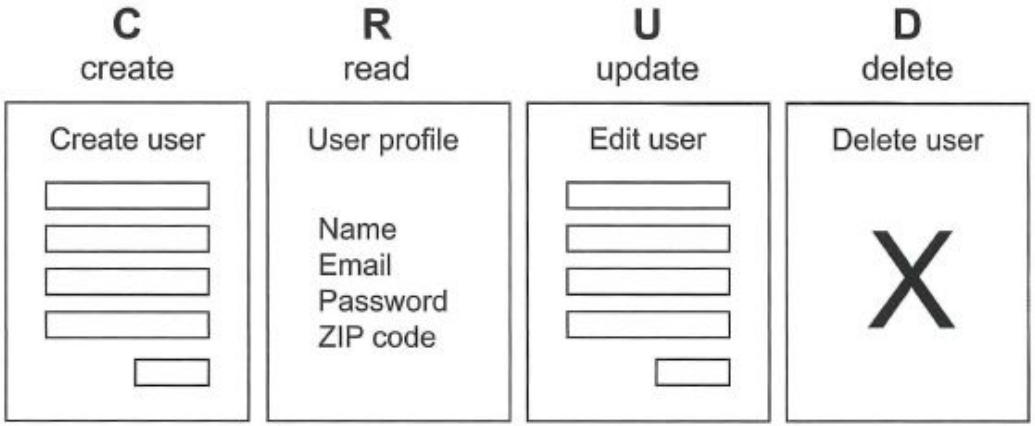


그림 18.1 CRUD 액션에 대한 뷰

18.2 모델에 CRUD 메소드 붙이기

controllers 폴더에 **userController.js**를 생성합니다.

웹 개발에서 CRUD는 대규모의 애플리케이션에서의 기초로서 각 모델은 표 18.1에서와같은 액션을 필요로 한다.

표 18.1 CRUD 액션

액션	설명
Create	create 함수는 크게 new와 create로 나뉜다. new는 모델의 새 인스턴스를 생성할 폼을 보여주기 위한 뷰와 라우트를 나타낸다. 예를 들어 새로운 사용자를 생성하려면 http://localhost:3000/users/new를 방문해 new.ejs에 있는 사용자 생성 폼을 보여주면 된다. create 라우트와 액션은 이 폼으로부터의 모든 POST 요청을 처리한다.
Read	Read 함수는 단 하나의 라우트, 액션, 뷰를 갖는다. 이 책에서는 show라는 이름으로 모델의 정보(대부분은 프로필 페이지)를 보여주고 있다. 아직 데이터베이스로부터 읽어들이고 있지만, 이 show 액션과 show.ejs 뷰는 Read 함수에서 흔히 쓰이는 이름이다.
Update	update 함수는 new와 비슷하게 크게 edit와 update.edit로 나뉘며, edit 라우트 및 edit.ejs로의 GET 요청을 처리한다. edit.ejs에서는 모델 속성 값 변경을 위한 폼이 존재한다. PUT 요청을 이용해 변경될 값을 폼을 통해 제출하면 update 라우트와 액션이 이 요청을 처리한다. 이들 함수는 데이터베이스에 모델의 인스턴스가 미리 존재해야 한다.
Delete	delete 함수가 가장 간단한 함수일 것이다. 레코드를 정말 삭제할 것인지를 묻는 뷰를 만들 수 있고 이 함수는 보통 DELETE 요청을 해당 사용자 ID를 키값으로 삭제하는 라우트로 보낸다. 그러면 delete 라우트와 액션은 데이터베이스에서 해당 레코드를 삭제한다.

18.2 모델에 CRUD 메소드 붙이기

구독자를 사용자와 연결합니다. (링크 나중에)

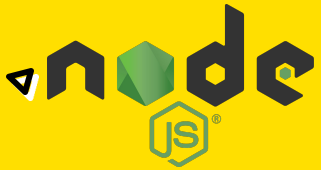
웹 개발에서 CRUD는 대규모의 애플리케이션에서의 기초로서 각 모델은 표 18.1에서와같은 액션을 필요로 한다.

표 18.1 CRUD 액션

액션	설명
Create	create 함수는 크게 new와 create로 나뉜다. new는 모델의 새 인스턴스를 생성할 폼을 보여주기 위한 뷰와 라우트를 나타낸다. 예를 들어 새로운 사용자를 생성하려면 http://localhost:3000/users/new를 방문해 new.ejs에 있는 사용자 생성 폼을 보여주면 된다. create 라우트와 액션은 이 폼으로부터의 모든 POST 요청을 처리한다.
Read	Read 함수는 단 하나의 라우트, 액션, 뷰를 갖는다. 이 책에서는 show라는 이름으로 모델의 정보(대부분은 프로필 페이지)를 보여주고 있다. 아직 데이터베이스로부터 읽어들이고 있지만, 이 show 액션과 show.ejs 뷰는 Read 함수에서 흔히 쓰이는 이름이다.
Update	update 함수는 new와 비슷하게 크게 edit와 update,edit로 나뉘며, edit 라우트 및 edit.ejs로의 GET 요청을 처리한다. edit.ejs에서는 모델 속성 값 변경을 위한 폼이 존재한다. PUT 요청을 이용해 변경될 값을 폼을 통해 제출하면 update 라우트와 액션이 이 요청을 처리한다. 이들 함수는 데이터베이스에 모델의 인스턴스가 미리 존재해야 한다.
Delete	delete 함수가 가장 간단한 함수일 것이다. 레코드를 정말 삭제할 것인지를 묻는 뷰를 만들 수 있고 이 함수는 보통 DELETE 요청을 해당 사용자 ID를 키값으로 삭제하는 라우트로 보낸다. 그러면 delete 라우트와 액션은 데이터베이스에서 해당 레코드를 삭제한다.

18.2 모델에 CRUD 메소드 붙이기

구독자를 사용자와 연결합니다. (링크 나중에)



앞에서 **GET**과 **POST** HTTP 메소드를 배웠다. 이 둘은 인터넷상에서 가장 빈번하게 사용되는 요청 방식이다. 많은 다른 HTTP 메소드들은 특별한 경우에만 사용되며, 여기 그 특별한 경우 중 2가지를 표 18.2에 소개한다.

표 18.2 PUT과 DELETE HTTP 메소드

HTTP 메소드	설명
PUT	기존 레코드의 업데이트나 수정을 위해 데이터를 애플리케이션 서버로 보내는 것을 가리키기 위해 사용되는 메소드다. PUT은 일부 변경되지 않더라도 보통 기존 레코드를 새로운 속성 세트로 변경한다. PUT이 업데이트를 위한 대표 메소드이기는 하지만 어떤 이는 PATCH 메소드를 더 선호하기도 하는데 PATCH는 변경되지 않는 속성만 골라 변경하는 것이 특징이다. 이 업데이트 라우트를 Express.js에서 처리하기 위해 app.put을 사용할 수 있다.
DELETE	데이터베이스에서 레코드를 삭제한다고 지정할 때 사용되는 메소드다. Express.js에서 이를 처리하기 위해 app.delete를 사용할 수 있다.

GET 및 POST 메소드를 이용해 레코드의 업데이트와 삭제가 가능하지만 HTTP 메소드 사용 시에는 이 베스트 프랙티스를 따르는 게 좋다. 일관성을 통해 애플리케이션은 오류가 거의 없어질 것이며 오류가 발생해도 투명성을 확보해 수정이 쉬워질 것이다.

18.2 모델에 CRUD 메소드 붙이기

구독자를 사용자와 연결합니다. (링크 나중에)

더 많은 액션을 만들수록 export를 그만큼 반복하게 되며 컨트롤러 모듈의 관점에서는 그다지 효율적이지 못하다. 이는 객체 리터럴에 module.exports를 모두 함께 익스포트시켜 컨트롤러 액션을 정리할 수 있다.

Listing 18.7 homeController.js에서의 액션 수정

모든 컨트롤러 액션과 함께 객체 리터럴 익스포트

```
module.exports = {
  showCourses: (req, res) => {
    res.render("courses", {
      offeredCourses: courses
    });
  }
};
```

이 구조를 다른 컨트롤러(errorController.js 및 subscribersController.js) 에도 적용하고 **모든 컨트롤러**들을 앞으로 옮긴다. 이런 수정은 CRUD 액션 및 라우트에서 미들웨어 구조를 생성할 때 중요하다.

[노트] Controllers 폴더에 coursesController.js 및 usersController.js 파일을 생성해 19장에서 사용될 강좌와 사용자 모델을 위한 동일한 액션을 만들 수 있다.

18.2 모델에 CRUD 메소드 붙이기

구독자를 사용자와 연결합니다. (링크 나중에)

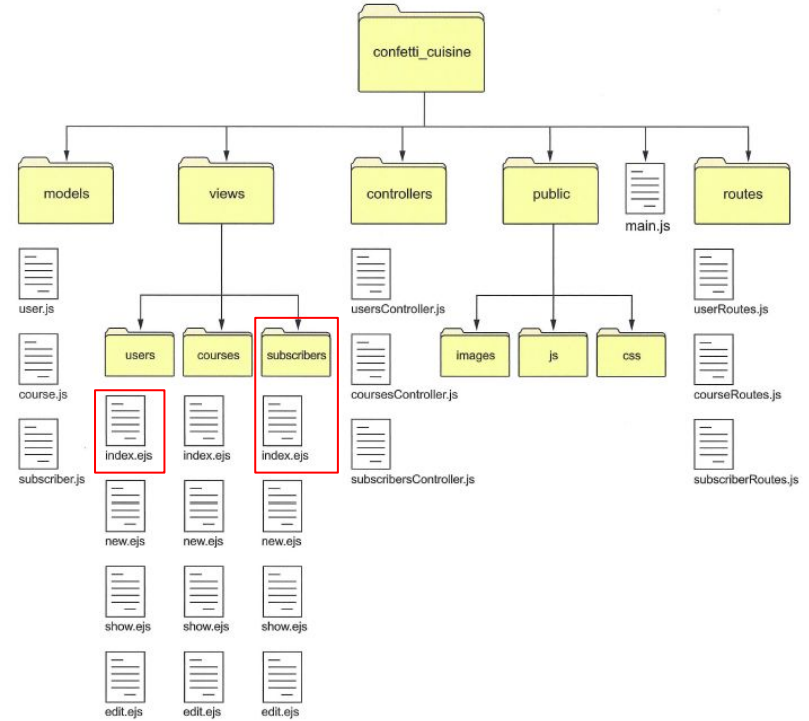


그림 29.1 라우트 폴더 중심의 애플리케이션 구조

먼저 애플리케이션에서 자주 간과되는 뷰인 index.ejs를 작성한다. 또한 각 애플리케이션 모델을 위한 인덱스 페이지를 작성한다. index 라우트, 액션, 뷰의 목적은 모든 레코드를 가져와 단일 페이지에 출력하는 것이다.

[노트] 여러분의 어플리케이션 내 다른 모델도 동일한 접근 방식을 적용해야 한다. 예를 들면 구독자 모델 뷰는 이제 views 폴더 내 subscribers 폴더로 이동해야 한다.

18.2 모델에 CRUD 메소드 붙이기

구독자를 사용자와 연결합니다. (링크 나중에)

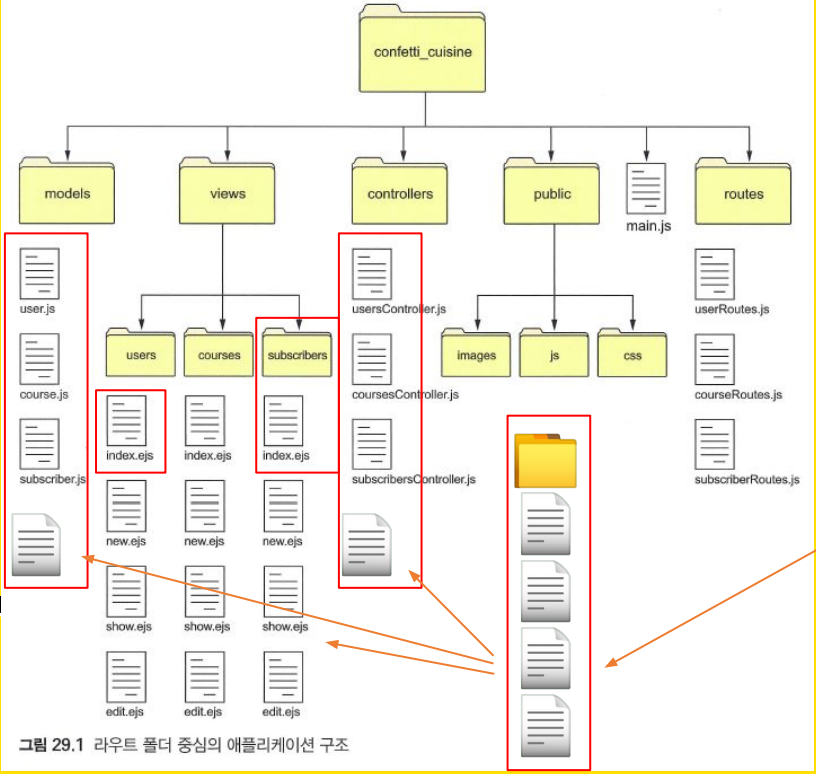


그림 29.1 라우트 폴더 중심의 애플리케이션 구조

먼저 애플리케이션에서 자주 간과되는 뷰인 index.ejs를 작성한다. 또한 각 애플리케이션 모델을 위한 인덱스 페이지를 작성한다. index 라우트, 액션, 뷰의 목적은 모든 레코드를 가져와 단일 페이지에 출력하는 것이다.

[노트] 여러분의 어플리케이션 내 다른 모델도 동일한 접근 방식을 적용해야 한다. 예를 들면 구독자 모델 뷰는 이제 views 폴더 내 subscribers 폴더로 이동해야 한다.

[최종 프로젝트] 새 스키마에 대한 새 모델, 보기, 컨트롤러를 추가합니다. **아이디어:** 교통 수단, 영화, 책, 게임, 스포츠, 관심사, 취미, 학교, 프로젝트 등.

18.3 인덱스 페이지 작성

controllers 폴더에 **userController.js**를 생성합니다.

Listing 18.9 userController.js에서 인덱스 액션 생성

```
const User = require("../models/user");          ← 사용자 모델 요청

module.exports = {
  index: (req, res) => {
    User.find({})
      .then(users => {
        res.render("users/index", {
          users: users
        })
      })
      .catch(error => {
        console.log(`Error fetching users: ${error.message}`)
        res.redirect("/");
      });
  }
};
```

사용자 배열로 index 페이지 렌더링

로그 메시지를 출력하고 홈페이지로 리디렉션

[노트] 구독자 컨트롤러에서 index 액션이 getAllSubscribers를 대체한다. main.js에서 액션 관련 라우트를 index를 가리키도록 수정하고 subscribers.ejs를 index.ejs로 변경된 점을 기억하자. 이 뷰는 views 폴더 아래 subscribers 폴더에 있어야 한다.

터미널에서 애플라케이션을 띄우고 http://localhost:3000/users를 접속해보자.

리스트 화면은 데이터베이스 내용 가운데 중요 데이터는 감춘 채 공개하는 화면이다.

18.3 인덱스 페이지 작성

controllers 폴더에 **userController.js**를 생성합니다.

Listing 18.11 userController.js에서 index 액션의 재방문

```
const User = require("../models/user");

module.exports = {
  index: (req, res, next) => {
    User.find()
      .then(users => {
        res.locals.users = users;
          next();
      })
      .catch(error => {
        console.log(`Error fetching users: ${error.message}`);
        next(error);
      });
  },
  indexView: (req, res) => {
    res.render("users/index");
  }
};
```

index 액션에서만 쿼리 실행

응답상에서 사용자 데이터를 저장하고 다음 미들웨어 함수 호출

에러를 캐치하고 다음 미들웨어로 전달

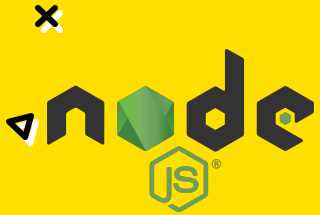
분리된 액션으로 뷰 렌더링

지금의 인덱스 액션은 데이터베이스로부터의 데이터를 보여주는 EJS 템플릿만을 위해 만들어졌다.

하지만 데이터를 뷰에서만 쓰게 하고 싶지는 않을 것이다. 이와 관련된 내용은 6부에서 다룬다. 이 액션을 좀 더 잘사용하려면 쿼리 단위로 액션들을 쪼개고 분리된 각 액션들은 뷰를 통해 결과를 출력한다.

18.1 사용자 모델 작성

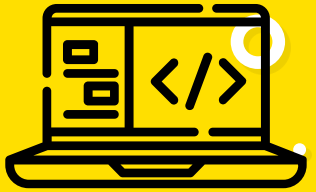
models/User.js에 **User** 모델을 생성하고 가상 속성을 추가합니다**.**



전과 같이 사용자의 인덱스 페이지에서 애플리케이션을 읽어들이려면 indexView를 라우트에서 index 액션을 따라가는 미들웨어 함수로 추가한다.

이렇게 함으로써 main.js 내의 /users 라우트를 app.get("/users", usersController.index, usersController.indexView)를 써서 변경한다 userController.index가 쿼리를 완료하고 응답 객체에 데이터를 보내면 userController.indexView가 뷰를 렌더링하기 위해 호출된다.

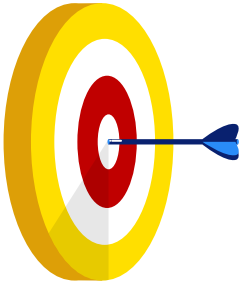
이 변경으로 나중에 다른 라우트에서의 인덱스 액션 이후 다른 미들웨어 함수 호출을 결정할 수 있게 된다. 이는 6부에서 수행할 것이다.



Coding!

Listing 18.1, 18.2 & 18.11

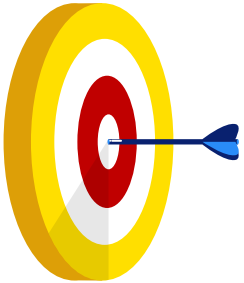
p. 259-260, 271



퀵 체크 18.1

가상 속성은 일반 모델 속성과 어떻게 다른가?

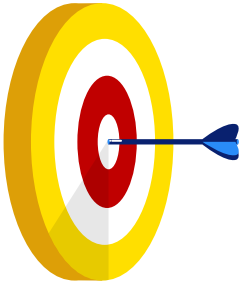
가상 속성은 데이터베이스에 저장되지 않는다. 이 속성은 다른 일반 속성과는 다르게 애플리케이션이 실행될 때만 존재한다. 몽고DB를 통해 검색되거나 데이터베이스로부터 추출될 수 없다.



퀵 체크 18.2

CRUD 중 자체 뷰를 필요로 하지 않는 것은 무엇인가?

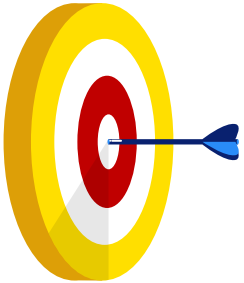
모든 CRUD 함수가 자체 뷰를 가질 수 있지만, 어떤 함수는 모달, 또는 기본 링크 요청으로 진행될 수 있다. 바로 delete 할수가 자체 뷰가 반드시 필요가 없는 함수인 데 레코드 삭제 영령을 보내기만 하면 되기 때문이다.



퀵 체크 18.3

인덱스 뷰의 용도는 무엇인가?

인덱스 뷰는 특정 모델의 모든 도큐먼트를 보여준다. 이는 회사에서 구독 중인 모든 사용자들의 이메일과 주소를 보여주는 데 사용될 수 있다. 또한 누가 등록을 했는지를 모든 이에게 보여줄 수도 있다.



퀵 체크 18.4

대부분 작업은 웹 브라우저에서 하는데 왜 에러 로그는 콘솔에서 출력하도록 할까?

더 많은 데이터와 기능을 뷰로 이관시커도 터미널은 여전히 애플리케이션의 중심이다. 콘솔윈도우는 여전히 애플리케이션 에러, 생성된 요정 그리고 사용자 정의 에러 메시지를 확인해 문제 발생 시 어디를 수정해야 할지 알게 해준다.

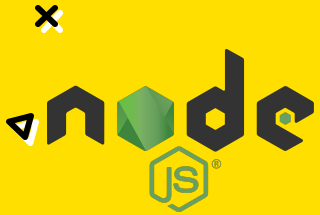
19

모델의 생성과 읽기

모델 생성 폼의 구축
브라우저에서 데이터베이스로 사용자 저장
뷰에서 연관 모델 출력

p. 275-287

19. 모델의 생성과 읽기



책과 조금 다른 순서

1. Create functions for **new** and **create** in controllers/userController.js.
   controllers/userController.js에 **new**와 **create** 함수를 생성합니다.

2. Add **new** and **create** routes for users in main.js.
   main.js에 사용자의 **new**와 **create** 라우트를 추가합니다.

3. Add a **pre('save') hook** to the User model in models/User.js.
   models/User.js의 User 모델에 **pre('save') 훅**을 추가합니다.

4. Add a **show** route for users in main.js using the **:id parameter.**
   main.js에 **:id 매개변수**를 사용하여 사용자의 **show** 라우트를 추가합니다.

5. Create a function for **show** in controllers/userController.js.
   controllers/userController.js에 **show** 함수를 생성합니다.

19.1 새로운 사용자 폼 제작

controllers/userController.js에 **new**와 **create** 함수를 생성합니다.

새로운 사용자가 본인의 계정을 생성할 수 있는 폼이 필요하다.CRUD 관점에서 이 폼은 new.ejs라는 파일에 위치한다.

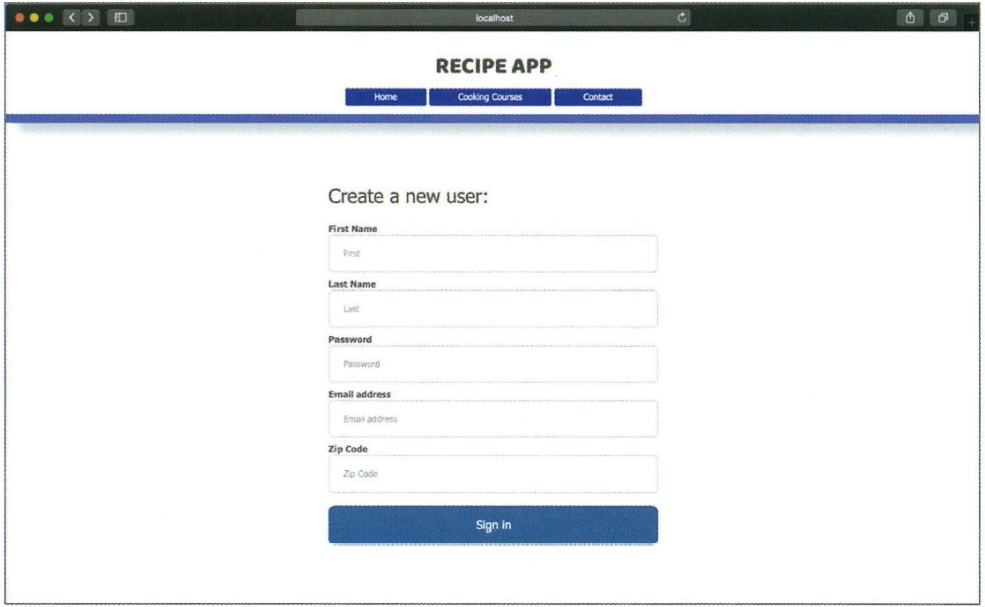


그림 19.1 브라우저에서의 사용자 생성 폼 예시

이 폼은 POST 요청을 /users/create 라우트로 폼 제출을 통해 발생시킨다. 이 라우트는 제출을 하기 전에 만들어져야 하며 이 순서가 지켜지지 않으면 애플리케이션은 크래시를 발생시킨다.

19.1 새로운 사용자 폼 제작

controllers/userController.js에 **new**와 **create** 함수를 생성합니다.

Listing 19.1 new.ejs에서의 사용자 생성 구축

```
<div class="data-form">
  <form action="/users/create" method="POST">
    <h2>Create a new user:</h2>
    <label for="inputFirstName">First Name</label>
    <input type="text" name="first" id="inputFirstName"
➥ placeholder="First" autofocus>
    <label for="inputLastName">First Name</label>
    <input type="text" name="last" id="inputLastName"
➥ placeholder="Last">
    <label for="inputPassword">Password</label>
    <input type="password" name="password" id="inputPassword"
➥ placeholder="Password" required>
    <label for="inputEmail">Email address</label>
    <input type="email" name="email" id="inputEmail"
➥ placeholder="Email address" required>
    <label for="inputZipCode">Zip Code</label>
    <input type="text" name="zipCode" id="inputZipCode" pattern="\d*"
➥ placeholder="Zip Code" required>
    <button type="submit">Sign in</button>
  </form>
</div>
```

사용자 계정을 생성하기 위한 폼을 생성한다.

폼의 input으로서 사용자 속성 추가

email과 password 필드의 보호를 위한 HTML 속성 적용

핵심은 각 사용자 속성이 폼 입력 형태로 나타난다는 것이다. 이 속성들의 이름은 input 값의 name 속성에 할당되며 이 경우에는 name="first" 가 된다.

나중에 컨트롤러에서 값들을 식별하기 위해 이 이름 속성을 사용할 것이다. password, email, zipCode 필드는 몇 가지 고유 속성을 가지고 있는 것에 주목하자.

이 HTML **유효성 평가**는 웹 페이지에서 애플라케이션으로 비보안적인 정보나 유효하지 않는 정보가 애플리케이션으로 유입되는 것을 막을 수 있다.

19.2 뷰로부터 새로운 사용자 생성

controllers/userController.js에 **new**와 **create** 함수를 생성합니다.

Listing 19.2 userController.js로의 액션 생성 추가

```
new: (req, res) => {
  res.render("users/new");
},

create: (req, res, next) => {
  let userParams = {
    name: {
      first: req.body.first,
      last: req.body.last
    },
    email: req.body.email,
    password: req.body.password,
    zipCode: req.body.zipCode
  };

  User.create(userParams)
      .then(user => {
        res.locals.redirect = "/users";
        res.locals.user = user;
        next();
      })
      .catch(error => {
        console.log(`Error saving user: ${error.message}`);
        next(error);
      });
},

redirectView: (req, res, next) => {
  let redirectPath = res.locals.redirect;
  if (redirectPath) res.redirect(redirectPath);
  else next();
}
```

폼의 렌더링을 위한 새로운 액션 추가

사용자를 데이터베이스에 저장하기 위한 create 액션 추가

폼 파라미터로 사용자 생성

분리된 redirectView 액션에서 뷰 렌더링

이제 새 로운 뷰를 만들었으며, 이를 위한 라우트와 컨트롤러 액션이 필요하다. 또한 create 라우트와 액션을 위해 추가한다.

새로운 사용자를 위한 폼은 사용자 스키마와 관련된 데이터를 수집하고, 그 다음으로 이 폼을 위한 액션의 생성이 필요하다.

new 액션은 새로운 사용자 생성의 요청을 받아 new.ejs에서 이를 렌더링한다.

create 액션은 new.ejs 폼으로부터 포스팅된 데이터를 받고 결과로서 생성된 사용자를 다음 미들웨어 함수로 응답 객체를 통해 전달한다.

다음 미들웨어 함수인 redirectView는 응답 객체의 일부로서 받은 리디렉션 경로를 기반으로 어떤 뷰를 보여줄지 결정한다.

19.2 뷰로부터 새로운 사용자 생성

main.js에 사용자의 **new**와 **create** 라우트를 추가합니다.

이 코드의 동작을 보면, new와 create 라우트를 main.js에 추가한다.

첫 번째 라우트는 /user/new로의 GET 요청을 new.ejs에서 받는다.
두 번째 라우트는 /user/create로의 POST 요청을 받고, 받은 요청 본문을 userController.js의 redirectView 액션을 통한 뷰 리디렉션으로 create 액션에 전달한다.

Listing 19.3 main.js에서 new와 create 라우트 추가

```
router.get("/users/new", usersController.new);
router.post("/users/create", usersController.create,
  usersController.redirectView);
```

생성 폼을 보기 위한 요청 처리

생성 폼으로부터의 데이터 제출과 뷰 출력을 위한 요청 처리

[노트] 구독자 컨트롤러에 new와 create 액션을 추가하는 것은 새로운 CRUD 액션에 맞춰 getAllSubscribers와 saveSubscriber 액션을 삭제할 수 있다는 의미다 게다가 롬 컨트롤러에서 할 것은 홈페이지인 index.ejs 제공밖에 없다.

19.2 뷰로부터 새로운 사용자 생성

main.js에 사용자의 **new**와 **create** 라우트를 추가합니다.

이제 main.js에서 사용하고 있는 라우터의 수가 증가하기 시작하고 있다. **const router = express.Router();** 코드를 main.js에 추가함으로써 Express.js에서 리우터 모듈을 사용할 수 있다. 이 코드 라인은 자체 미들웨어와 Express.js 앱 객체와 함께 라우팅을 제공하는 라우터 객체를 생성한다. 지금은 엡 대신에 사용할 라우터를 위해 라우트를 수정하자. 그 후 **app.use("/", router)** 를 main.js 내 라우트의 제일 위에 추가한다.

Listing 19.3 main.js에서 new와 create 라우트 추가

```
router.get("/users/new", usersController.new);
router.post("/users/create", usersController.create,
  usersController.redirectView);
```

생성 폼을 보기 위한 요청 처리

생성 폼으로부터의 데이터 제출과 뷰 출력을 위한 요청 처리

[노트] 구독자 컨트롤러에 new와 create 액션을 추가하는 것은 새로운 CRUD 액션에 맞춰 getAllSubscribers와 saveSubscriber 액션을 삭제할 수 있다는 의미다 게다가 롬 컨트롤러에서 할 것은 홈페이지인 index.ejs 제공밖에 없다.

19.2 뷰로부터 새로운 사용자 생성

models/User.js의 **User** 모델에 **pre('save')** 훅을 추가합니다.

Listing 19.4 user.js에 pre("save") 훅 추가

```
콜백에서 함수 키워드 사용
userSchema.pre("save", function (next) {        ← pre("save") 훅 설정
  let user = this;
  if (user.subscribedAccount === undefined) {
    Subscriber.findOne({                        ← 기존 Subscriber 연결을 위한 조건 체크 추가
      email: user.email
    })                                          ← Single Subscriber를 위한 쿼리
      .then(subscriber => {
        user.subscribedAccount = subscriber;    ← 사용자와 구독자 계정 연결
        next();
      })
      .catch(error => {
        console.log(`Error in connecting subscriber:
? ${error.message}`);
        next(error);                            ← 에러 발생 시 다음 미들웨어 함수로 전달
      });
  } else {
    next();                                     ← 이미 연결 존재 시 다음 미들웨어 함수 호출
  }
});
```

이상적으로는 사용자가 생성될 때마다 기존 구독자 중 같은 이메일 주소가 있는지를 찾고 이를 연결시키려 할 것이다. 이를 **Mongoose 의 pre("save") 훅**에서 해준다.

Mongoose에서는 훅hooks 이라고 하는 메소드를 제공한다. 이를 통해 저장과 같은 데이터베이스 변경이 실행되기 전에 특정한 오퍼레이션을 실행할 수 있다. 이 축은 사용자의 생성 또는 저장 직전에 실행된다.

여기에서는 화살표 함수를 사용할 수 없기 때문에, 사용자 정의 변수는 프라미스 체인 밖에서 정의를 해줘야 한다. 이 함수는 사용자가 관련된 구독자가 없을 경우에 사용하다.

또한 user.js에서 구독자 모델에 참조를 추가해야 하며 이는 **const Subscriber = require("./subscriber")** 명령 구문을 상단에 추가해 수행한다.

19.3 show를 통한 사용자 데이터 읽기

main.js에 **:id** 매개변수를 사용하여 사용자의 **show** 라우트를 추가합니다.

Listing 19.5 show.ejs에서의 사용자 Show 테이블

```
<h1>User Data for <%= user.fullName %></h1>

<table class="table">
  <tr>
    <th>Name</th>
    <td><%= user.fullName %></td>
  </tr>
  <tr>
    <th>Email</th>
    <td><%= user.email %></td>
  </tr>
  <tr>
    <th>Zip Code</th>
    <td><%= user.zipCode %></td>
  </tr>
  <tr>
    <th>Password</th>
    <td><%= user.password %></td>
  </tr>
</table>

<% if (user.subscribedAccount) { %>
    <h4 class="center"> This user has a
    <a href="<%=`/subscribers/${user.subscribedAccount}` %>">
    subscribed account</a>.
    </h4>
<% } %>
```

사용자 데이터를 보여주기 위한 테이블 추가

구독자 계정 확인

이제 사용자 데이터를 생성할 수 있게 됐으며, 사용자 정보를 사용자 프로필 페이지와 같은 연관 페이지에 출력하려고 한다. 이 경우 필요한 오퍼레이션은 단순한 데이터베이스 읽기가 될 것이며, 특정 ID를 통해 사용자를 찾고 관련 콘텐츠를 브라우저에 출력하는 형태다.

우선 새로운 뷰인 **show.ejs**를 생성하자. 마지막으로 사용자가 subscribedAccount를 가지고 있는지 체크한다.

[노트] 링크된 구독자 페이지가 제대로 동작하려면 이런 과정을 구독자를 위한 CRUD 함수 및 뷰의 생성에서도 동시에 따라야 할 것이다. Href 앵커 태그인 /subscribers/${user.subscribedAccount}는 구독자의 show 페이지를 나타낸다.

Listing 19.6 index.ejs에서의 name 데이터 업데이트

```
<td>
<a href="<%= `/users/${user._id}` %>">
    <%= user.fullName %>
  </a>
</td>
```

HTML에서 사용자 이름과 ID 삽입

19.3 show를 통한 사용자 데이터 읽기

controllers/userController.js에 show 함수를 생성합니다.

다음으로 show 액션을 userController.js로 추가한다. 먼저 사용자 ID를 URL 파라미터로부터 수집한다. 이 정보는 **req.params.id**로부터 얻을 수 있다. 이 코드는 라우트를 **:id**로 정의한 경우에 작동된다.

Listing19.7 userController.js에서 특정 사용자에 대한 Show 액션

```
show: (req, res, next) => {
  let userId = req.params.id;
  User.findById(userId)
      .then(user => {
        res.locals.user = user;
          next();
      })
      .catch(error => {
        console.log(`Error fetching user by ID: ${error.message}`);
        next(error);
      });
    },

showView: (req, res) => {
  res.render("users/show");
}
```

ID로 사용자 찾기

reqest params로부터 사용자 ID 수집

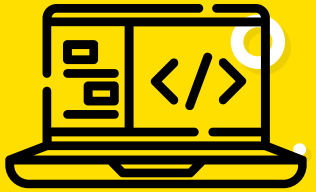
응답 객체를 통해 다음 미들웨어 함수로 사용자 전달

에러를 로깅하고 다음 함수로 전달

show 뷰의 렌더링

findById 쿼리를 사용해 사용자 ID를 전달한다. 각 ID는 고유하기 때문에 단일 사용자를 돌려받게 될 것이다.

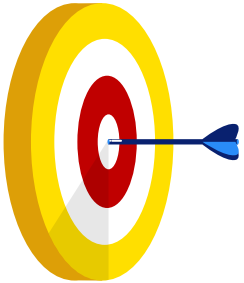
마지막으로 main.js 에 사용자를 위한 show 라우트를 **router.get("/users/:id", usersController.show, usersController.showView)** 로 추가한다.



Coding!

Listing 19.2-19.4 - 19.7

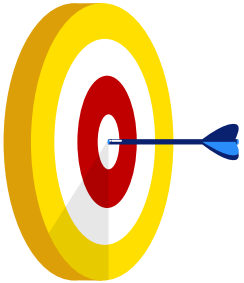
p. 278-281, 285



퀵 체크 19.1

폼 데이터를 확인하는 컨트롤러 액션을 위한 값이 꼭 필요한 input 속성은 무엇인가?

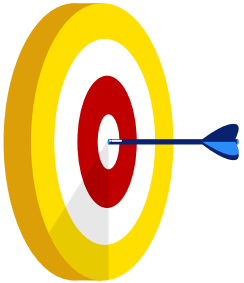
name 속성은 새로운 레코드 생성을 위해 반드시 채워져야 하는 속성이다. 이름 속성에 매핑되는 값은 컨트롤러가 모델 스키마와 비교하는 데 사용하는 값이다.



퀵 체크 19.2

왜 Mongoose의 pre(“save") 훅훅은
파라미터로서 next를 받을까?

pre(“save") 훅은 Mongoose의 미들웨어다. 따라서 다른 미들웨어와 같이 동작하기 위해서는 함수 수행이 종료됐을 때 다음 미들웨어 함수로 이동한다 next는 여기에서 미들웨어 체인 내의 다음 함수를 의미한다.



퀵 체크 19.3

참 또는 거짓! URL에서 사용자 ID를 나타내는 파라미터는 반드시 :id여야 한다.

거짓이다. :id 파라미터는 나타내려 하는 사용자의 ID를 받는데 필요한 요소나 이 파라미터는 독자가 선택한 그 어떤 것으로도 나타낼 수 있다. 만약 :userid를 사용하기로 했다면, 이 이름을 모든 코드에 일괄 적으로 적용하면 된다.

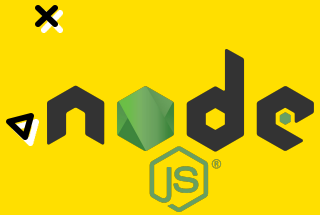
20

모델의 업데이트와 삭제

모델 수정 폼의 구축
데이터베이스에서 사용자 레코드 업데이트
사용자 레코드 삭제

p. 289-300

20. 모델의 업데이트와 삭제



책과 조금 다른 순서

1. Add **method-override** to your project and require it in main.js.
   method-override를 프로젝트에 추가하고 main.js에서 요구합니다.

2. Add **edit** and **update** routes for users and create functions for them in controllers/userController.js.
   controllers/userController.js에 사용자의 **edit**와 **update** 라우트를 추가하고 함수를 생성합니다.

3. Add a **delete** route for users and create a function for it in controllers/userController.js.
   controllers/userController.js에 사용자의 **delete** 라우트를 추가하고 함수를 생성합니다.

4. **Update the show route** for users in main.js to include a link to the edit and delete routes.
   main.js의 사용자의 **show 라우트를 업데이트**하여 edit와 delete 라우트에 대한 링크를 포함합니다.

20.1 사용자 정보 편집 폼 생성

method-override를 프로젝트에 추가하고 **main.js**에서 요구합니다.

사용자 정보를 업데이트하기 위해 사용자 정보 편집을 위한 폼을 만든다. 이 폼은 create.js와 비슷하나 폼의 액션은 users/create 대신에 **users/:id/update**로 바뀐다.

또한 기존 정보들을 입력된 값으로 대체도 해야 할 것이다.

Listing 20.1 edit.ejs에서의 사용자 입력 폼 예제

```
<input type="text" name="first" id="inputFirstName" value="<%=
➥ user.name.first %>" placeholder="First" autofocus>
```

편집 폼에 기존 사용자 속성 값을 적용

edit 링크를 위한 href 값은 GET 요청을 **/users+사용자 ID+/edit** 라우트로만든다.

Listing 20.2 index.ejs에서 사용자 편집을 위한 테이블 수정

```
<td>
  <a href="<%=`/users/${user._id}/edit` %>">
    Edit
  </a>
</td>
```

사용자 ID를 edit 태그 링크에 삽입

20.1 사용자 정보 편집 폼 생성

method-override를 프로젝트에 추가하고 **main.js**에서 요구합니다.

다음은 수정된 사용자 데이터를 PUT 요청으로 제출하기 위해 edit.ejs를 수정해야 한다. 하지만 HTML 폼은 GET 및 POST만 지원한다.

명확히 해야 할 문제 중 하나는 어떻게 Express.js가 이 요청을 받는다. Express.js는 POST 요청 형태로 HTML 폼을 전달받기 때문에 의도대로 HTTP 메소드로 요청을 해석하기 위한방법이 필요하다. **method-override** 패키지를 사용할 것이다.

method-override는 HTTP 메소드에서 특정 쿼리 파라미터에 따라 요청을 해석하는 미들웨어다. method=PUT을 쿼리에 추가함으로써 POST 요청을 PUT 요청으로 해석할 수 있다.

npm i method-override 명령으로 프로젝트 터미널 윈도우에서 설치하며 main.js에 추가한다.

Listing 20.3 main.js에서 애플리케이션에 method-override 추가

```
const methodOverride = require("method-override");
router.use(methodOverride("_method", {
  methods: ["POST", "GET"];
}));
```

← 오버라이드 모듈의 요청

methodOverride를 미들웨어로 사용하기 위한 애플리케이션 라우터 설정

20.1 사용자 정보 편집 폼 생성

method-override를 프로젝트에 추가하고 **main.js**에서 요구합니다.

Listing 20.5 edit.ejs에서의 최종 사용자 편집 폼

```
<div class="data-form">          ←— 사용자 편집 폼 출력
  <form method="POST" action="<%=`/users/${user._id}/update
➥ ?_method=PUT`%>">
    <h2>Edit user:</h2>
    <label for="inputFirstName">First Name</label>
    <input type="text" name="first" id="inputFirstName" value="<%=
➥ user.name.first %>" placeholder="First" autofocus>
    <label for="inputLastName">Last Name</label>
    <input type="text" name="last" id="inputLastName" value="<%=
➥ user.name.last %>" placeholder="Last">
    <label for="inputPassword">Password</label>
    <input type="password" name="password" id="inputPassword"
➥ value="<%= user.password %>" placeholder="Password" required>
    <label for="inputEmail">Email address</label>
    <input type="email" name="email" id="inputEmail" value="<%=
➥ user.email %>" placeholder="Email address" required>
    <label for="inputZipCode">Zip Code</label>
    <input type="text" name="zipCode" id="inputZipCode"
➥ pattern="\d*" value="<%= user.zipCode %>" placeholder="Zip
➥ Code" required>
    <button type="submit">Update</button>
  </form>
</div>
```

POST 메소드로 **/users/:id/update?_method=PUT**로 제출하기 위해 edit.ejs에 있는 폼을 수정해야 한다. 폼 태그는 Listing 20.5와 같을 것이다.

20.2 뷰에서 사용자 수정 폼

controllers/userController.js에 사용자의 **edit**와 **update** 라우트를 추가하고 함수를 생성합니다.

이제 사용자 편집 폼은 자체 뷰를 갖게 됐다 여기에 폼의 보강을 위해 컨트롤러 액션과 라우트를 추가한다.

edit 라우트와 액션은 edit.ejs를 보여주기 위해 사용된다.
update 라우트와 액션은 데이터베이스 내 사용자 정보를 내부적으로 변경하기 위해 사용된다.

그런 다음 redirectView 액션은 update 후 작업으로 설정해 지정된 뷰로 리디렉션을 한다.

edit 액션은 보기 액션과 마찬가지로 사용자를 사용해 데이터베이스로부터 사용자 정보를 가져와 편집을 하기 위해 뷰로 읽어들인다.

update 액션은 create 액션과 마찬가지로 편집 폼이 제출될 때 호출되며 사용자 ID 및 userParams 값을 확정해 Mongoose 의 **findByIdAndUpdate** 메소드로 전달한다. 이 메소드는 **$set** 명령을 사용해 도큐먼트로 대체하려 하는 파라미터 다음에 오는 ID를 받는다. 사용자 업데이트가 성공하면 다음 미들웨어 내의 사용자 보기 경로로 리디렉션하다.

20.2 뷰에서 사용자 수정 폼

controllers/userController.js에 사용자의 **edit**와 **update** 라우트를 추가하고 함수를 생성합니다.

Listing 20.6 userController.js로의 edit와 update 액션 추가

```
edit: (req, res, next) => {                    ← edit 액션 추가
  let userId = req.params.id;
  User.findById(userId)                        ← ID로 데이터베이스에서 사용자를 찾기 위한 findById 사용
      .then(user => {
        res.render("users/edit", {
          user: user
        });                                    ← 데이터베이스 내 특정 사용자를 위한 편집 페이지 렌더링
      })
      .catch(error => {
        console.log(`Error fetching user by ID: ${error.message}`);
        next(error);
      });
},
```

edit 액션은 사용자를 사용해 데이터베이스로부터 사용자 정보를 가져와 편집을 하기 위해 뷰로 읽어들인다.

update 액션은 create 액션과 마찬가지로 편집 폼이 제출될 때 호출되다.

```
update: (req, res, next) => {                  ← update 액션 추가
  let userId = req.params.id,
    userParams = {
      name: {
        first: req.body.first,
        last: req.body.last
      },
      email: req.body.email,
      password: req.body.password,
      zipCode: req.body.zipCode
    };                                         ← 요청으로부터 사용자 파라미터 취득
  User.findByIdAndUpdate(userId, {
    $set: userParams
  })                                           ← ID로 사용자를 찾아 단일 명령으로 레코드를 수정하기 위한 findByIdAndUpdate의 사용
      .then(user => {
        res.locals.redirect = `/users/${userId}`;
        res.locals.user = user;
        next();                                ← 지역 변수로서 응답하기 위해 사용자를 추가하고 다음 미들웨어 함수 호출
      })
      .catch(error => {
        console.log(`Error updating user by ID: ${error.message}`);
        next(error);
      });
}
```

20.2 뷰에서 사용자 수정 폼

main.js의 사용자의 **show** 라우트를 업데이트하여 **edit**와 **delete** 라우트에 대한 링크를 포함합니다.

마지막으로 main.js에 Listing 20 .7의 코드를 추가해야 한다.
또한 응답의 **locals 객체**에서 특정된 뷰를 보여주기 위한 redirectView 액션을 재사용할 것이다.

Listing 20.7 main.js로의 edit 및 update 라우트 추가

```
router.get("/users/:id/edit", usersController.edit);
router.put("/users/:id/update", usersController.update,
  usersController.redirectView);
```

viewing을 처리하기 위한 라우트 추가

편집 폼에서 받아온 데이터의 처리와 결과를 사용자 보기 페이지에 보여주기

20.3 delete 액션에서 사용자 삭제

controllers/userController.js에 사용자의 **delete** 라우트를 추가하고 함수를 생성합니다.

편집 칼럼에서 그랬듯이, 각 사용자에 대한 삭제 링크를 **users/:id/delete**로 만들어준다. 애플리케이션이 GET 요청을 DELETE 요청으로 인식하기 위해 **_method=DELETE**를 사용해야 한다는 것을 상기하자.

[노트] onclick=“ return confirm(“Are you sure you want 10 delete this record?")" 의 추가로 좀 더 안전한 장치를 추가할 수 있다.

Listing 20.8 index.ejs에서의 사용자 삭제 링크

```
<td>
  <a href="<%= `users/${user._id}/delete?_method=DELETE `%>"
  onclick="return confirm("Are you sure you want to delete
  this record?")">Delete</a>
</td>
```

인덱스 페이지에서 delete 액션으로의 링크 추가

20.3 delete 액션에서 사용자 삭제

controllers/userController.js에 사용자의 **delete** 라우트를 추가하고 함수를 생성합니다.

다음으로 ID로 사용자 레코드 삭제하는 컨트롤러 액션을 추가한다.

클릭해 선택한 레코드를 데이터베이스에서 삭제시키기 위한 Mongoose의 **findByIdAndRemove** 메소드를 사용하고 있다. 도큐먼트 삭제에 성공했다면 사용자 삭제 로그를 콘솔에 남기고 다음 미들웨어 함수로 사용자 인덱스 페이지를 전달할 것이다.

Listing 20.9 userController.js에서 delete 액션의 추가

```
delete: (req, res, next) => {
  let userId = req.params.id;
  User.findByIdAndRemove(userId)
      .then(() => {
        res.locals.redirect = "/users";
        next();
      })
      .catch(error => {
        console.log(`Error deleting user by ID: ${error.message}`);
        next();
      });
}
```

findByIdAndRemove 메소드를 이용한 사용자 삭제

20.3 delete 액션에서 사용자 삭제

controllers/userController.js에 사용자의 **delete** 라우트를 추가하고 함수를 생성합니다.

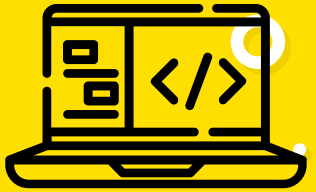
여기에서 빠진 부분은 main.js에 추가한 라우트 **router.delete("/users/:id/delete", usersController.delete, usersController.redirectView)**이다.

마지막으로 사용자 프로필 페이지에서 CRUD 액션을 쉽게 사용하기 위해 show.ejs의 하단에 링크를 추가한다.

Listing 20.10 show.ejs에 CRUD 액션을 위한 링크 추가

```
<div>
  <a href="/users">View all users</a>
</div>
<div>
  <a href="<%=`/users/${user._id}/edit`%>">
    Edit User Details
  </a>
</div>
<div>
  <a href="<%= `/users/${user._id}/delete?_method=DELETE` %>"
  ➥ onclick="return confirm("Are you sure you want to delete
  ➥ this record?")">Delete</a>
</div>
```

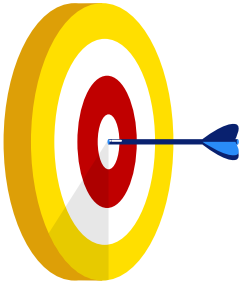
프로필 페이지에서 사용자 계정의 편집 및 삭제를 위한 링크 추가



Coding!

Listing 20.3, 20.6-20.7, 20.9

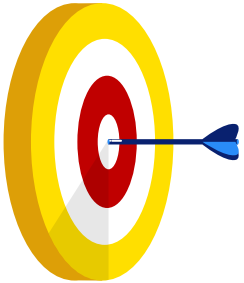
p. 292, 294-296, 298



퀵 체크 20.1

왜 편집 폼에는 PUT 메소드를,
신규 폼에는 POST 메소드를 사용할까?

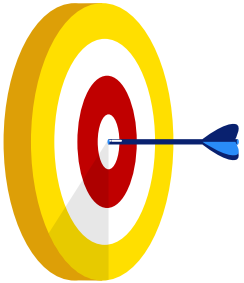
편집 폼은 기존 레코드 수정을 위한 데이터의 업데이트다.
일반적으로 서버로의 데이터 제출은 HTTP PUT 메소드를 써 왔다.
새로운 레코드 생성은 POST를 사용한다.



퀵 체크 20.2

참 또는 거짓! findByIdAndUpdate는 Mongoose 메소드다.

참이다. findByIdAndUpdate는 서버노드에서 여러분의 쿼리를 간결하고 가독성 있게 만들어주는 Mongoose 메소드다. 따라서 Mongoose 패키지가 설치돼 있지 않으면 사용할 수 없다.



퀵 체크 20.3

왜 `?_method=DELETE`는 링크 경로 뒤에 붙여야 할까?

method-override는 쿼리 파라미터중 method를 찾아 메소드를 매핑시킨다, 유입되는 GET 및 POST 요청을 다른 대체 메소드로 필터링시키기 위해 이 파라미터 및 값을 뒤에 붙여야 한다.

과제 타임!

한번 해 보자!~