Unit 4 21

CRUD 모델 추가

(최종 프로젝트를 시작하기)

UT-NodeJS / 05.19.2023

ut-nodejs.github.io

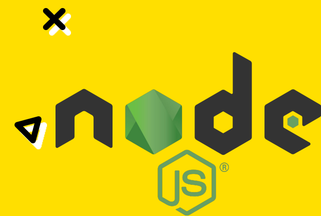
21

캡스톤 프로젝트 4

CRUD 모델 추가
(최종 프로젝트를 시작하기)

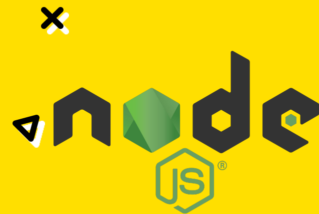
p. 301-320

사용자 구독 저장



이 캡스톤 프로젝트에서 사용자가 계정 생성과 편집, 수정 그리고 삭제를 할 수 있는 로그인 폼을 만들려고 한다. 강좌 및 Coffetti Cuisine 뉴스레터 구독자를 위해 프로세스 대부분을 반복할 것이다 (Lesson 17-20).

① 시작하기



추가 적으로 method-override 패키지도 현재 HTTP 링크와 폼으로 지원하지 않는 부분의 보완을 위해 설치해야 한다. 설치는 새로운 터미널 윈도우에서:

```
npm i method-override -S
```

로 가능하다.

그 후 `override = require("method-override");` 를 main.js 상단에 추가함으로써 method-override의 요청을 한다. 그리고 다른 메소드와 마찬가지로 GET과 POST를 위한 method-override를 사용하기 위해:

```
app.use(methodOverride("_method", {methods: ["POST", "GET"]}));
```

을 추가한다.

npm install **method-override**

② 모델 생성

다음으로 이 프로젝트의 디렉터리 구조를 어떻게 갖춰야 할지 생각해야 한다. 사실 이는 어느 정도 완료됐다. 이미 CRUD 기능을 3개 모델에 추가했고 3개의 새로운 컨트롤러 및 3개의 새로운 폴더를 views 폴더 내에 생성할 것이기 때문이다. 전체 구조는 그림 21.1과같다.

여기서 4개의 뷰(index, new, show, edit) 만을 생성한 것에 주목하자 delete도 삭제 확인 페이지라는 자체 뷰를 가질 수도 있으나, 여기서는 각 모델의 index 페이지 내 링크로 처리 할것이다.

timestamps는 Mongoose가 이 모델에서의 createdAt 및 updatedAt의 기록을 위해 제공되는 추가 속성이다.

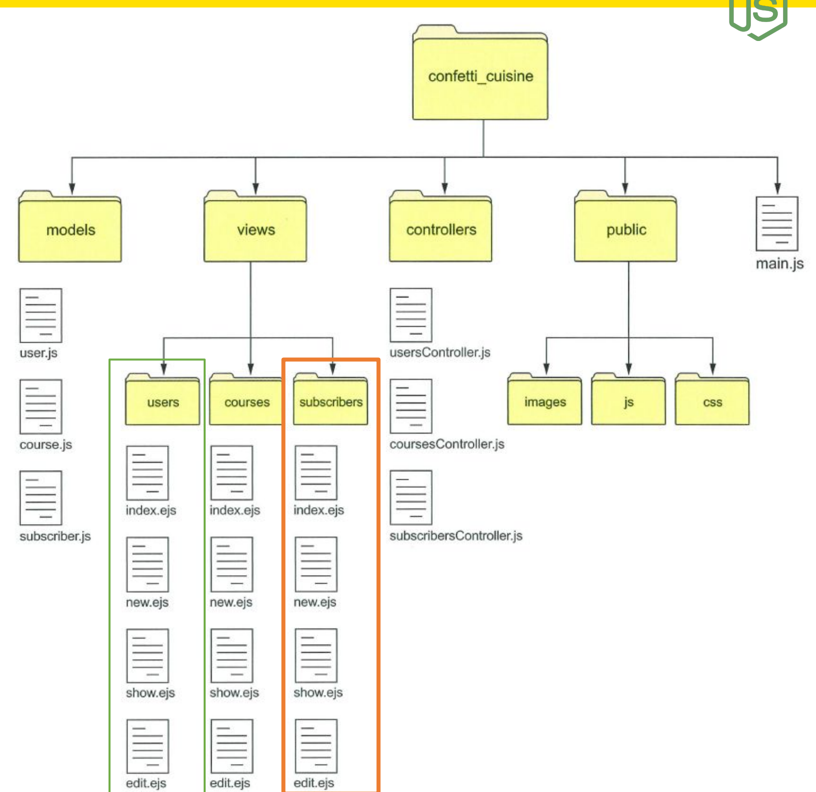


그림 21.1 캡스톤 프로젝트의 파일 구조

② 모델 생성

다음으로 이 프로젝트의 디렉터리 구조를 어떻게 갖춰야 할지 생각해야 한다. 사실 이는 어느 정도 완료됐다. 이미 CRUD 기능을 3개 모델에 추가했고 3개의 새로운 컨트롤러 및 3개의 새로운 폴더를 views 폴더 내에 생성할 것이기 때문이다. 전체 구조는 그림 21.1과같다.

여기서 4개의 뷰(index, new, show, edit) 만을 생성한 것에 주목하자 delete도 삭제 확인 페이지라는 자체 뷰를 가질 수도 있으나, 여기서는 각 모델의 index 페이지 내 링크로 처리 할것이다.

timestamps는 Mongoose가 이 모델에서의 createdAt 및 updatedAt의 기록을 위해 제공되는 추가 속성이다.

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

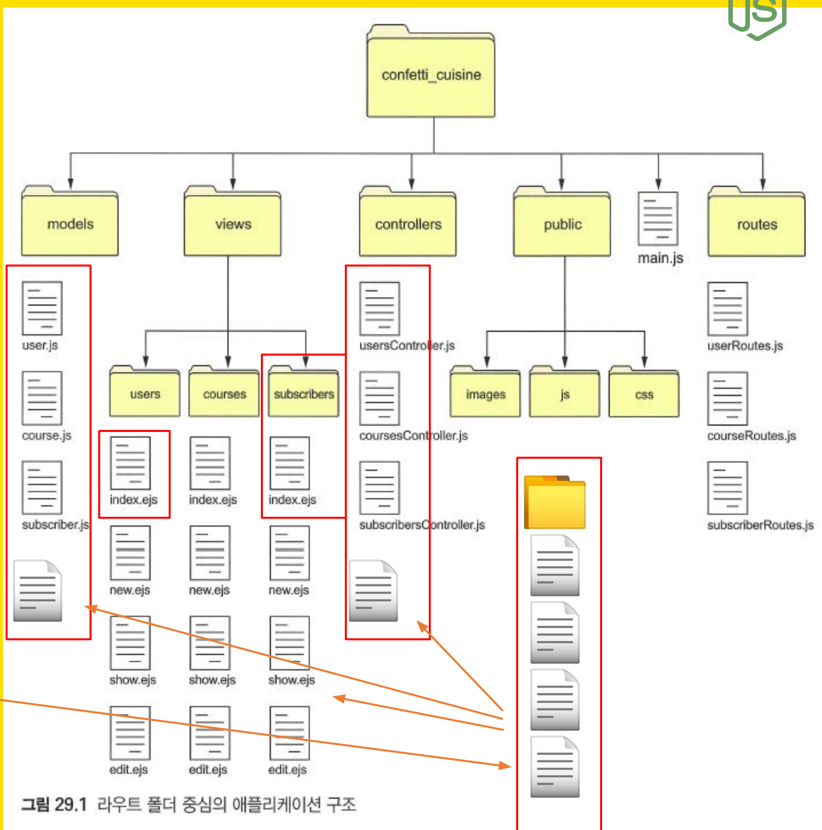


그림 29.1 라우트 폴더 중심의 애플리케이션 구조

②Subscriber.js에서의 Subscriber 수정

다음으로 이 프로젝트의 디렉터리 구조를 어떻게 갖춰야 할지 생각해야 한다. 사실 이는 어느 정도 완료됐다. 이미 CRUD 기능을 3개 모델에 추가했고 3개의 새로운 컨트롤러 및 3개의 새로운 폴더를 views 폴더 내에 생성할 것이기 때문이다. 전체 구조는 그림 21.1과같다.

여기서 4개의 뷰(index, new, show, edit) 만을 생성한 것에 주목하자 delete도 삭제 확인 페이지라는 자체 뷰를 가질 수도 있으나, 여기서는 각 모델의 index 페이지 내 링크로 처리
할것이다.

timestamps는 Mongoose가 이 모델에서의 createdAt 및 updatedAt의 기록을 위해 제공되는 추가 속성이다.

```
module.exports = mongoose.model("Subscriber",
➥ subscriberSchema);
```

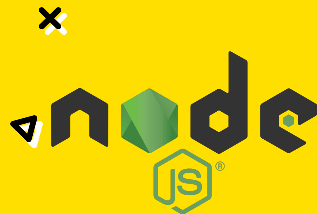
Subscriber 모델의 export

Listing 21.1 subscriber.js에서의 Subscriber 수정

```
const mongoose = require("mongoose"),
  { Schema } = mongoose,
  subscriberSchema = new Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    lowercase: true,
    unique: true
  },
  zipCode: {
    type: Number,
    min: [10000, "Zip code too short"],
    max: 99999
  },
  courses: [{type: Schema.Types.ObjectId, ref: "Course"}]
}, {
  timestamps: true
});

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

- Mongoose의 요청
- 스키마 속성의 추가
- 복수 강좌의 연계
- getInfo 인스턴스 메소드 추가

② Course.js에서의 Course 수정



모든 강좌는 제목과 입력 제한이 없는 설명을 필요로 한다 각 강조}는 maxStudents와 cost 속성이 있고 초기 값은 0 이며 음수는 입력될 수 없다. 만일 음수가 오변 에러 메시지가 뜰 것이다.

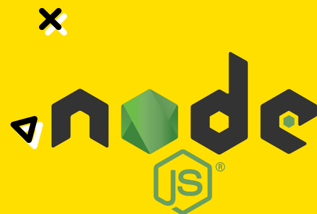
Listing 21.2 course.js에서 Course의 수정

```
const mongoose = require("mongoose"),
  { Schema } = require("mongoose"),
  courseSchema = new Schema(
    {
      title: {
        type: String,
        required: true,
        unique: true
      },
      description: {
        type: String,
        required: true
      },
      maxStudents: {

        type: Number,
        default: 0,
        min: [0, "Course cannot have a negative number of students"]
      },
      cost: {
        type: Number,
        default: 0,
        min: [0, "Course cannot have a negative cost"]
      }
    },
    {
      timestamps: true
    }
  );
module.exports = mongoose.model("Course", courseSchema);
```

제목(title)과 설명(description) 요청

maxStudents와 cost는 초깃값이 0이며 음수가 올 수 없다.

② User.js에서의 User 수정



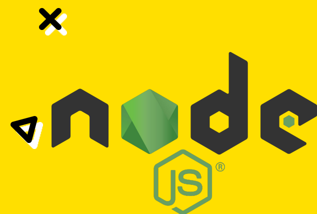
Listing 21.3 user.js에서 User 모델 생성

```
const mongoose = require("mongoose"),
  { Schema } = require("mongoose"),
  Subscriber = require("./subscriber"),
  userSchema = new Schema(
   {
     name: {                         ← first 및 last 속성 추가
       first: {
         type: String,
         trim: true
       },
       last: {
         type: String,
         trim: true
       }
     },
     email: {
       type: String,
       required: true,
       unique: true
     },
     zipCode: {
       type: Number,
       min: [10000, "Zip code too short"],
       max: 99999
     },
     password: {
       type: String,
       required: true
     },                              ← 패스워드 요청
     courses: [
       {
         type: Schema.Types.ObjectId,
         ref: "Course"
       }                             ← 복수의 강좌와 사용자의 연결
     ],
     subscribedAccount: {
       type: Schema.Types.ObjectId,
       ref: "Subscriber"
     }                               ← 사용자와 구독자의 연결
   },
   {
     timestamps: true                ← timestamps 속성 추가
   }
  );
module.exports = mongoose.model("User", userSchema);
```

User 모델의 대부분 처음부터 사용자 입력 시 비유효값을 차단하기 위해 필드들은 유효성 체크를 거치고 있다. 이 모델은 Course와 Subscriber 모델과 둘 다 연결돼 있다.

구독자들은 갑자기 사용자 계정을 만들기 때문에 이 두 개의 계정을 연결해야 한다. 또한 timestamps 속성을 두어 데이터베이스에서의 사용자 레코드 변경 이력을 남기도록 한다.

② User.js에 가상 속성과 pre("save") 훅



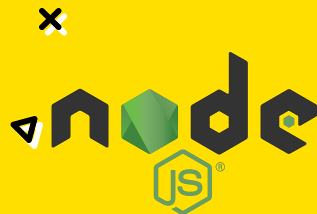
Listing 21.4 user.js에 가상 속성과 pre("save") 훅 추가

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

userSchema.pre("save", function (next) {     // 구독자와의 링크를 위해 pre("save") 훅을 추가
  let user = this;
  if (user.subscribedAccount === undefined) {     // 링크된 subscribedAccount 정의 여부 확인
    Subscriber.findOne({
      email: user.email
    })     // 사용자 email을 포함하는 구독자 도큐먼트 검색
      .then(subscriber => {
        user.subscribedAccount = subscriber;
        next();     // 다음 미들웨어 함수 호출
      })
      .catch(error => {
        console.log(`Error in connecting subscriber:
➥ ${error.message}`);
        next(error);
      });
  } else {
    next();
  }
});
```

사용자 모델에 더 추가한 것은 사용자의 풀 네임을 돌려주기 위한 가상 속성과 구독자와 사용자를 동일한 이메일 주소로 연결하기 위한 Mongoose 의 pre("save") 훅이다.

pre("save") 훅은 사용자가 데이터베이스에 저장되기 직전에 수행된다. 그리고 next 파라미터를 전달해 이 함수가 완료되면 다음 단계의 미들웨어 체인을 호출할 수 있도록 하고 있다.

③ 뷰의 생성 (views/subscribers/index.ejs)



Listing 21.5 index.ejs에서 구독자 목록 출력

```
<h2 class="center">Subscribers Table</h2>
<table class="table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th>Edit</th>
      <th>Delete</th>
    </tr>
  </thead>
  <tbody>
    <% subscribers.forEach(subscriber => { %>
    <tr>
     <td>
     <a href="<%= `/subscribers/${subscriber._id}` %>">
       <%= subscriber.name %>
     </a>
     </td>
      <td><%= subscriber.email %></td>
      <td>
      <a href="<%=`subscribers/${subscriber._id}/edit` %>">
        Edit
      </a>
      </td>
      <td>
        <a href="<%= `subscribers/${subscriber._id}/delete?_method=DELETE` %>"
  onclick="return confirm('Are you sure you want to delete this
  record?')">Delete</a>
      </td>
    </tr>
    <% }); %>
  </tbody>
</table>
```

- 인덱스 페이지에 테이블 추가
- 구독자 각각에 대한 열 출력
- 구독자명을 앵커 태그 처리
- DELETE 링크 추가

모델 설정을 갖고 이제 CRUD 기능을 만들어야 한다. 우선 **index.ejs, new.ejs, show.ejs, edit.ejs**로 뷰를 각각 만든다.

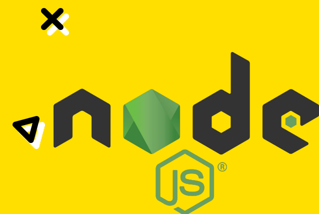
[노트] 이 3개의 뷰는 다른 모델 간에 같은 이름을 갖고 있어 모델 이름별 폴 더로 분리, 관리해야 한다. 예를 들어 **views/subscribers** 폴더는 자체 index.ejs를 갖고 있다.

각 구독자에 매치되는 열을 만들기 위해 구독자 객체의 배열로 돼 있는 subscribers 변수를 반복해 루프를 돈다.

삭제 링크는 경로 뒤에 쿼리 파라미터 **?_method=DELETE**를 붙여 method-override 미들헤어가 DELETE 요청으로 이를 처리할 수 있도록 한다. EJS의 코드 블록에서 이를 클로징해주는 것도 잊지 말길 바란다.

이와 동일한 구조를 **courses**와 **users** 인텍스 페이지에도 적용해 변수명과 속성만 변경하면 관련 모델과 매치되도록 할 것이다.

③뷰의 생성 (views/subscribers/new.ejs)



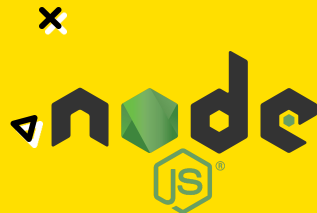
Listing 21.6 new.ejs에서의 새로운 구독자 생성 폼

```
<div class="data-form">
  <form action="/subscribers/create" method="POST">   ← 새로운 구독자 생성을 위한 폼 추가
    <h2>Create a new subscriber:</h2>
    <label for="inputName">Name</label>
    <input type="text" name="name" id="inputName" placeholder="Name"
➥ autofocus>
    <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="[0-9]{5}" placeholder="Zip Code" required>
    <button type="submit">Create</button>
  </form>
</div>
```

각 폼의 **name** 속성값은 매우 중요하며 이를 갖고 새로운 레코드 저장 시 필요한 데이터를 수집할 것이다.

users와 **courses**를 위해 이 폼을 다시 만들며, 이를 통해 폼의 액션과 입력 값들이 생성되는모델의 속성에 반영토록한다.

③ 뷰의 생성 (views/subscribers/edit.ejs)



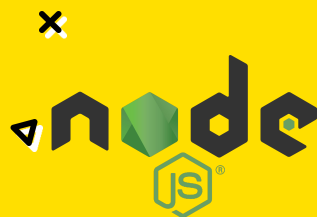
Listing 21.7 edit.ejs에서 구독자 편집 페이지

```
<form action="<%=`/subscribers/${subscriber._id}/update
➥ ?_method=PUT` %>" method="POST">        ← 구독자 편집 폼 표시
  <h2>Create a new subscriber:</h2>
  <label for="inputName">Name</label>
  <input type="text" name="name" id="inputName" value="<%=
➥ subscriber.name %>" placeholder="Name" autofocus>
  <label for="inputEmail">Email address</label>
  <input type="email" name="email" id="inputEmail" value="<%=
➥ subscriber.email %>" placeholder="Email address" required>
  <label for="inputZipCode">Zip Code</label>
  <input type="text" name="zipCode" id="inputZipCode"
➥ pattern="[0-9]{5}" value="<%= subscriber.zipCode %>"
➥ placeholder="Zip Code" required>
  <button type="submit">Save</button>
</form>
```

폼으로 작업하면서 new.ejs와 비슷한 폼을 가진 edit.ejs 뷰를 만든다. 달라진 점은 다음과 같다.

- **편집 폼:** 이 폼은 편집하고 있는 레코드로 액세스해야 한다. 이 경우에는 구독자는 구독자 컨트롤러에서 가져온다.
- **폼 액션:** 이 액션은 create 액션 대신 `/subscribers/${subscriber._id}/update?_method=PUT`으로 동작한다.
- **속성:** 각 입력 값 속성은 구독자 변수의 속성으로 `<input type="text" name="name" value="<%=subscriber.name %>">` 형태로 설정된다.

③ 뷰의 생성 (views/subscribers/show.ejs)



Listing 21.8 show.ejs에서의 구독자 보기 뷰

```
<h1>Subscriber Data for <%= subscriber.name %></h1>     ← 구독자 속성의 출력

<table>
  <tr>
    <th>Name</th>
    <td><%= subscriber.name %></td>
  </tr>
  <tr>
    <th>Email</th>
    <td><%= subscriber.email %></td>
  </tr>
  <tr>
    <th>Zip Code</th>
    <td><%= subscriber.zipCode %></td>
  </tr>
</table>
```

Listing 21.9 show.ejs에서 구독 강좌 수 보기

```
<p>This subscriber has <%= subscriber.courses.length %> associated
➥course(s)</p>     ← 구독 강좌 수 출력
```

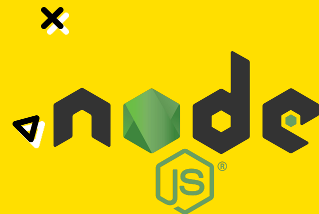
마지막으로 각 모델에 대한 뷰 페이지를 만든다.

[노트] 이러한 뷰 중 일부에 관해서는 해당 모델의 다른 관련 페이지로 이동하기 위한 링크를 추가할 것이다.

여기에 추가하려고 한 또 다른 것은 레코드가 데이터베이스상에서 다른 레코드와 연계돼 있는지를 보여주는 코드다. 구독자 영역에서는 구독하는 강좌 수를 보여주는 열을 추가하다.

이 열은 사이트에게 사람들이 구독 중인 강좌 수의 인사이트를 준다. 이 라인을 좀 더 발전시켜 Mongoose의 **populate** 메소드를 사용해 관련 강좌의 세부 정보로 연결시킬 수도 있다. **(나중에)**

④라우트 구조화



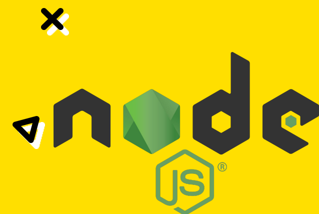
마지막단계는모델과뷰를같이 컨트롤러 액션과라우트로연결하는것이다.

main.js에서 필요한 CRUD 라우트를 추가하고 모든 것이 작동되기 위해 필요한 컨트롤러를 요청할 것이다.

다른 컨트롤러와 함께 상단에 subscribersController를 요청하기 위해 **`const subscribersController = require("./controllers/subscribersController");`** 를 추가한다.

또한 프로젝트에 Express.js Router를 적용해, main.js에서의 다른 설정으로부터의 애플리케이션 라우터와 구별을 하도록 하는데 이는 **`const router = express.Router();`** 를 추가해 적용한다. 이 라우터 객체의 작성으로, 엠 객체에 의해 처리되는 모든 라우트와 미들웨어를 이 라우트 객체를 사용하기 위해 변경한다. 그 후 애플리케이션에 main.js에 **`app.use("/", router);`** 를 추가해 이 라우터의 사용을 알려준다.

④ 라우트 구조화



Listing 21.10 main.js로의 구독자 CRUD 라우트 추가

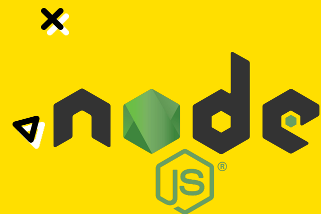
```
router.get("/subscribers", subscribersController.index,
➥ subscribersController.indexView);
router.get("/subscribers/new", subscribersController.new);
router.post("/subscribers/create", subscribersController.create,
➥ subscribersController.redirectView);
router.get("/subscribers/:id", subscribersController.show,
➥ subscribersController.showView);
router.get("/subscribers/:id/edit", subscribersController.edit);
router.put("/subscribers/:id/update", subscribersController.update,
➥ subscribersController.redirectView);
router.delete("/subscribers/:id/delete",
➥ subscribersController.delete,
➥ subscribersController.redirectView);
```

- 보기 뷰를 위한 GET 라우트 추가
- 생성을 위한 첫 번째 POST 라우트 추가
- ObjectId에 기초한 구독자 보기 라우트 추가
- 구독자 업데이트를 위한 라우트 추가
- 구독자 삭제를 위한 라우트 추가

동일한 7개의 라우트들이 **users**와 **courses**에도 적용돼야 한다. 또한 내비게이션 링크를 업데이트하는데 이때 연락처 링크는 구독자의 새로운 뷰로 강좌 목록 링크는 강좌의 인텍
스뷰로걸리게된다.

[노트] 이 시점에서 이제 사용하지 않는 라우드들을 정리할 수 있다. 구독자 컨트롤러에서 **getAllSubscribers, getSubscriptionPage, saveSubscriber** 그리고 홈 컨트롤러 에서의 **showCourses**로 연결되는 라우트 같은 것들이다. 또한 **홈페이지 라우트**를 홈 컨트롤러의 인덱스 액션으로 이동시킬 수 있다. 마지막으로 내비게이션 링크를 /contact 대신 /subscribers/new로 수정하는 것을 잊지 말자.

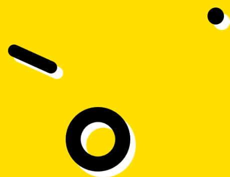
⑤컨트롤러 제작



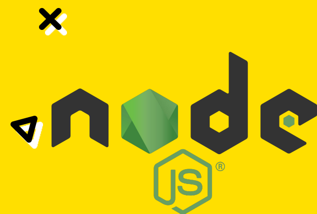
이제 남은 것은 연결될 컨트롤러를 만드는 일이다.

main.js에서 만든 라우트들은 **subscribersController, coursesController** 그리고 **usersController**가 필요하다. 이들을 controllers 폴더에 만들 것이다.

[노트] 앞의 애플리케이션 예제에서와 같이 **http-status-codes 및 error.ejs**를 사용하기 위해 에러 컨트롤러를 정리했다.



⑤컨트롤러 제작 (index + indexView)



Listing 21.11 subscribersController.js에서 구독자 컨트롤러 액션 추가

```
const Subscriber = require("../models/subscriber"),
  getSubscriberParams = (body) => {
    return {
      name: body.name,
      email: body.email,
      zipCode: parseInt(body.zipCode)
    };
  };

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

- 요청으로부터 구독자 데이터를 추출하기 위한 사용자 정의 함수 제작
- 모든 구독자 도큐먼트를 찾기 위한 index action 생성

이 파일로 Subscriber 모델을 요청한 뒤 데이터베이스에 있는 모든 구독자 도큐먼트를 찾기 위한 인텍스 액션을 만들고 이들을 subscribers 변수를 통해 indexView 액션으로 index.ejs로 전달한다. new와 edit 액션은 제출 및 구독자 데이터 수정을 위해 뷰를 렌더링한다.

⑤컨트롤러 제작 (new, create, redirectView)

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

create: (req, res, next) => {
  let subscriberParams = getSubscriberParams(req.body);
  Subscriber.create(subscriberParams)
    .then(subscriber => {
      res.locals.redirect = "/subscribers";
      res.locals.subscriber = subscriber;
      next();
    })
    .catch(error => {
      console.log(`Error saving subscriber:${error.message}`);
      next(error);
    });
},

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

새로운 구독자 생성을 위한 create action 생성

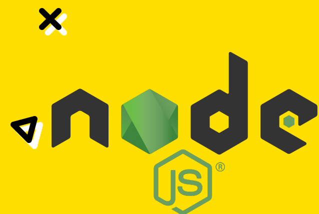
create 액션은 사사용자 정의 getSubscriberParams 함수 내에서 요청 본문(body) 파라미터를 수집하다다.

그 후 redirectView 액션에서 인덱스 페이지로 리디렉션이 되도록 지정할 것이다.

```
show: (req, res, next) => {
  var subscriberId = req.params.id;
  Subscriber.findById(subscriberId)
    .then(subscriber => {
      res.locals.subscriber = subscriber;
      next();
```

구독자 데이터를 출력하기 위한 show 액션 생성

⑤컨트롤러 제작 (edit, update)

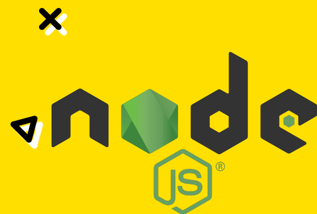


```
edit: (req, res, next) => {
  var subscriberId = req.params.id;
  Subscriber.findById(subscriberId)
    .then(subscriber => {
      res.render("subscribers/edit", {
        subscriber: subscriber
      });
    })
    .catch(error => {
      console.log(`Error fetching subscriber by ID:
➥ ${error.message}`);
      next(error);
    });
},

update: (req, res, next) => {
  let subscriberId = req.params.id,
    subscriberParams = getSubscriberParams(req.body);

  Subscriber.findByIdAndUpdate(subscriberId, {
    $set: subscriberParams
  })
    .then(subscriber => {
      res.locals.redirect = `/subscribers/${subscriberId}`;
      res.locals.subscriber = subscriber;
      next();
    })
    .catch(error => {
      console.log(`Error updating subscriber by ID:
➥ ${error.message}`);
      next(error);
    });
},
```

기존 구독자 도큐먼트의 새로운 값으로 설정하기 위한 update 액션 생성

update 액션은 create와 비슷하게 작동하며 findByIdAndUpdate라는 Mongoose 메소드를 사용해 기존 구독자 도큐먼트를 위한 새 값들을 설정한다. 여기서 정된 사용자 객체는 응답객체를 통해 전달하며 redirectView 액션에서 리디렉션될 뷰를 특정한다.

⑤컨트롤러 제작 (delete)



구독자 도큐먼트를 삭제하기 위한 delete 액션 생성

```
delete: (req, res, next) => {
  let subscriberId = req.params.id;
  Subscriber.findByIdAndRemove(subscriberId)
    .then(() => {
      res.locals.redirect = "/subscribers";
      next();
    })
    .catch(error => {
      console.log(`Error deleting subscriber by ID:
➥ ${error.message}`);
      next();
    });
  }
};
```

delete 액션은 요청 파라미터 중 구독자 ID를 사용해 데이터베이스로부터 매칭되는 도큐먼트를 findByIdAndRemove를 한다.

getSubscriberParams 함수는 코드의 반복을 줄이기 위해 만든 것이다.

create와 update 액션은 폼 파라미터를 사용하기 때문에 이 함수를 대신에 호출할 수 있다. redirectView 액션 역시 delete 액션을 포함해 코드 반복을 줄이기 위한 것이며 메인 함수가 완료되면 어떤 뷰를 렌더링할 것인지를 특정한다.

⑥ 실행 index.ejs

각 모델을 위한 이들 컨트롤러 액션의 설정으로, 애플리케이션은 레코드를 관리할 수 있는 기반을 마련했다. 각 모델에 대한 보기 뷰를 로딩하고, 새로운 **subscriber, course, user**를 생성한다.

localhost

CONFETTI CUISINE

Home Courses Contact Users

Subscribers Table

Name	Email	Edit	Delete
Jon Wexler	jon@jonwexler.com	Edit	Delete
Chef Eggplant	eggplant@recipeapp.com	Edit	Delete
Professor Souffle	souffle@recipeapp.com	Edit	Delete

그림 21.2 구독자 인덱스 페이지

⑥ 실행

index.ejs

각 모델을 위한 이들 컨트롤러 액션의 설정으로, 애플리케이션은 레코드를 관리할 수 있는 기반을 마련했다. 각 모델에 대한 보기 뷰를 로딩하고, 새로운 **subscriber, course, user**를 생성한다.

localhost

CONFETTI CUISINE

Home | Courses | Contact | Users

Edit subscriber:

Name

Jon Wexler

Email address

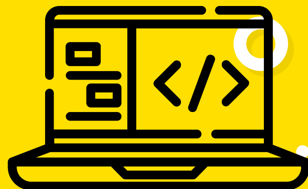
jon@jonwexler.com

Zip Code

10016

Save

그림 21.3 구독자 편집 페이지



Coding 과제!

캡스톤 프로젝트

(최종 프로젝트를 시작하기)

p. 301-320

과제 타임!

[한번 해 보자!~]