[스파르타코딩클럽] Node.js 심화반 - 2주차

매 주차 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

[수업 목표]

1. JWT(JSON Web Token)이 무엇인지 알고, 어떻게 사용해야 할지 알게 된다.
2. 로그인 기능을 직접 구현해본다.
3. 익명 쇼핑몰을 회원제 쇼핑몰로 바꿔본다.

[목차]

모든 토글을 열고 닫는 단축키
Windows : `Ctrl` + `alt` + `t`
Mac : `⌘` + `⌥` + `t`

---

00. 선행 지식

이 지식들은 기초 수업 또는 앞선 수업에서 배운 내용이기 때문에 본 수업에서는 자세한 설명을 생략해요!
그러나 기억이 나지 않을 여러분을 위해 아주 간단한 내용이라도 정리해두었어요 😇

▼ 1) 쿠키 & 세션

- **쿠키**: 브라우저가 서버로부터 응답으로 Set-Cookie 헤더를 받은 경우 해당 데이터를 저장한 뒤 모든 요청에 포함하여 보냅니다.
  - 데이터를 여러 사이트에 공유할 수 있기 때문에 보안에 취약할 수 있습니다.
- **세션:** 쿠키를 기반으로 구성된 기술입니다. 단, 클라이언트가 마음대로 데이터를 확인 할 수 있던 쿠키와는 다르게 세션은 데이터를 서버에만 저장하기 때문에 보안이 좋으나, 반대로 사용자가 많은 경우 서버에 저장해야 할 데이터가 많아져서 서버 컴퓨터가 감당하지 못하는 문제가 생기기 쉽습니다.

▼ 2) express - 미들웨어의 개념

express에서의 미들웨어는 어떠한 요청에 대해서 공통적으로 처리하는 로직을 모아둔 코드 덩어리입니다.

우리가 사용해본 `express.static`, `express.json`, `express.urlencoded` 같은 함수도 사실은 미들웨어를 만들어주는 함수입니다! 이것들에 비유해서 부가 설명을 해보자면,

- `express.static(path)`: path에 입력한 경로에 있는 파일을 그대로 서빙해주는 기능을 수행하는 미들웨어입니다.
  router의 기능을 일부 가지고 있는것이죠!
- `express.json`: HTTP Request에서 Body에 담긴 JSON 형식의 데이터를 express 서버에서 사용할 수 있게 해주는 미들웨어예요!
- `express.urlencoded`: 이것도 HTTP Request에서 Body에 담긴 Form(URL Encoded) 형식의 데이터를 express 서버에서 사용할 수 있게 해주는 미들웨어입니다 🙂 (참고)

▼ 3) ES6 - 구조 분해 할당(Destructuring assignment) 문법

이러한 문법을 배운적이 있을거예요!

```
const { email, password } = req.body;
```

자세한 내용은 MDN 문서를 참고하세요!

▼ 4) mongoose 사용법

모델 구현하는 방법, 데이터 저장하고 읽고 쓰는 방법은 이미 모두 배웠어요!
무언가 기억이 나지 않는다면 공식 문서를 찾아보시면 됩니다 😉

▼ 5) REST API

1주차 수업 시간에 배운 내용중 하나입니다!
아래와 같이 설명했었죠 😉

> REST 아키텍쳐를 따라 구현된 API를 REST API라고 부릅니다.
> 간단히 말하면 원래 있던 방법보다 더 쉽고 사람이 읽기 편한 방식으로 원칙을 세워놨고, 개발자들의 생산성과 상호작용을 증진시키는것에 목적이 있습니다.

01. 2주차에서 배울 것

기초 과정에서 배웠던 내용을 복습하는 1주차와 달리, 앞으로는 기초 과정에서 구현하던 쇼핑몰에 기능을 더 추가해볼건데요 오늘은 로그인 기능을 추가해볼 예정입니다!

▼ 1) JWT: JSON Web Token

👉 오늘은 여러분에게 JWT가 무엇인지 알려드리고, 이것을 이용하는 방법을 알려드릴거예요.

▼ 2) 로그인 기능 구현

👉 JWT를 이용한 로그인 기능 구현은 쿠키/세션으로 구현하는것과 어떻게 다른지 알아보는 시간을 가지게 될거예요.

02. JWT가 무엇인가요?

▼ 1) 간략한 정리!

- JSON 형태의 데이터를 안전하게 교환하여 사용할 수 있게 해줍니다.
- 인터넷 표준으로서 자리잡은 규격입니다.
- 여러가지 암호화 알고리즘을 사용할 수 있습니다.
- `header.payload.signature` 의 형식으로 3가지의 데이터를 포함한다. (개미처럼 머리, 가슴, 배)
  때문에 JWT 형식으로 변환 된 데이터는 항상 2개의 `.` 이 포함된 데이터여야 합니다.

▼ 2) 어떻게 생긴건가요?

https://jwt.io/ 에서 간단히 확인할 수 있는데요, 위에서 말했듯이 개미처럼 머리, 가슴, 배와 같은 3가지를 가졌습니다.

- header(머리)는 signature(배)에서 어떤 암호화를 사용하여 생성된 데이터인지 표현합니다.
- payload(가슴)는 개발자가 원하는 데이터를 저장합니다.
- signature(배)는 이 토큰이 변조되지 않은 정상적인 토큰인지 확인할 수 있게 도와줍니다.

▼ 3) 더 알아두어야 할 특성

- JWT는 암호 키를 몰라도 Decode가 가능합니다.
  변조만 불가능 할 뿐, 누구나 복호화하여 보는것은 가능하다는 의미가 됩니다!
- 때문에 민감한 정보(개인정보, 비밀번호 등)는 담지 않도록 해야합니다.
- 특정 언어에서만 사용 가능한것은 아닙니다!
  단지 개념으로서 존재하고, 이 개념을 코드로 구현하여 공개된 코드를 우리가 사용하는게 일반적입니다.

▼ 4) 쿠키, 세션과 어떻게 다른가요?

데이터를 교환하고 관리하는 방식인 쿠키/세션과 달리, **JWT는** 단순히 데이터를 표현하는 형식입니다.

- JWT로 만든 데이터를 브라우저로 보내도 쿠키처럼 자동으로 저장되지는 않지만, 변조가 거의 불가능하고 서버에 데이터를 저장하지 않기 때문에 서버를 stateless(무상태)로 관리할 수 있기 때문에 최근 많이 쓰이는 기술중 하나입니다.
- stateless(무상태)와 stateful(상태 보존)의 차이를 간단히 설명하자면,
  Node.js 서버가 언제든 죽었다 살아나도 똑같은 동작을 하면 Stateless하다고 볼 수 있습니다.
  반대로 서버가 죽었다 살아났을때 조금이라도 동작이 다른 경우 Stateful하다고 볼 수 있겠죠.
- 서버가 스스로 어떤 기억을 갖고 다른 결정을 하냐 마냐의 차이라고 보면 더 쉽습니다 😉
- 로그인 정보를 서버에 저장하게 되면 무조건 Stateful(상태 보존)이라고 볼 수 있죠!

03. JWT는 어떻게 사용하면 되나요?

▼ 1) 오픈소스 라이브러리를 이용합니다.

우리는 제일 사용량이 많은 jsonwebtoken 라이브러리를 사용해볼게요!

프로젝트로 이용할 폴더를 생성하고, 모듈을 설치해줍니다!

```
npm i jsonwebtoken -S
```

프로젝트가 있는 경로에서 위 명령어로 우리가 필요한 모듈을 설치합니다!

▼ 2) 우리가 원하는 JSON 데이터를 암호화합니다.

▼ 데이터를 암호화 해봅니다!

```
const jwt = require("jsonwebtoken");

const token = jwt.sign({ myPayloadData: 1234 }, "mysecretkey");
console.log(token);
```

- 개발자인 우리가 담을 데이터는 Payload에 담긴다고 했죠? jwt.io 에서도 아래처럼 데이터를 입력하면 인코딩 된 데이터를 볼 수 있습니다!

- 위처럼 코드를 작성하고 실행하면 어떤 화면이 보이나요?
  저와 같다면 `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJteVBheWxvYWREYXRhIjoxMjM0fQ.6XFgtNglH9hIzz5y8jAcI0g5kDnlAvnTTbxKIcL2CHY`
  이와 똑같은 데이터가 출력되어야 합니다!

▼ 3) 복호화를 해봅니다!

▼ 코드로 복호화해서 출력해봅시다!

```
const jwt = require("jsonwebtoken");

const token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJteVBheWxvYWREYXRhIjoxMjM0fQ.6XFgtNglH9hIzz5y8jAcI0g5kDnlAvnTTbxKIcL2CH
const decodedValue = jwt.decode(token);
```

앞서 말했듯이 JWT는 누구나 복호화가 가능해요!
그러나 검증을 통해 변조가 되지 않은 데이터인지 알 수 있죠!

▼ 복호화가 아닌, 변조되지 않은 데이터인지 검증해봅시다

```
const jwt = require("jsonwebtoken");

const token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJteVBheWxvYWREYXRhIjoxMjM0fQ.6XFgtNglH9hIzz5y8jAcI0g5kDnlAvnTTbxKIcL2CH
const decodedValue = jwt.verify(token, "myesecretkey");
```

만약 변조된 코드라면 위의 코드에서 에러가 발생할거예요!

- jwt.io 에서 복호화하여 확인해봅니다!

▼ 4) 이 암호화 된 데이터는 어떻게 쓸 수 있나요?

- 보통 암호화 된 데이터는 클라이언트(브라우저)가 전달받아 다양한 수단(쿠키, 로컬스토리지 등)을 통해 저장하여 API 서버에 요청을 할 때 서버가 요구하는 HTTP 인증 양식에 맞게 보내주어 인증을 시도합니다!
- 비유하자면, 놀이공원의 자유이용권과 비슷한거죠!
  - **회원가입**: 회원권 구매
  - **로그인**: 회원권으로 놀이공원 입장
  - **로그인 확인**: 놀이기구 탑승 전마다 유효한 회원권인지 확인
  - **내 정보 조회**: 내 회원권이 목에 잘 걸려 있는지 확인하고, 내 이름과 사진, 바코드 확인

04. 로그인 기능 구현하기

▼ 1) 로그인 기능이 추가된 쇼핑몰 프론트엔드 파일 적용하기

앞서 말했듯이 여러분이 Node.js 학습에 집중할 수 있게 하기 위해서, 프론트엔드는 이미 저희가 구현해둔 상태입니다!
아래 파일을 다운받아주세요!

▼ **[코드스니펫] 스파르타 쇼핑몰 프론트엔드 코드**

```
https://s3.ap-northeast-2.amazonaws.com/materials.spartacodingclub.kr/nodejs_advanced/week03/week2-shopping-front-end.zip
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/390e29cb-8643-45db-8dc6-9d38d1a61e1c/week2-shopping-front-end.zip

▼ 2) 새로 적용한 프론트엔드가 어떻게 동작하도록 바뀌었는지 설명

▼ 로그인 체크 기능이 추가됐습니다.

1. 로그인 화면, 회원가입 화면 외에 다른 모든 페이지에 접속하는 경우, API를 `GET /users/me` 경로로 호출합니다.
2. API의 HTTP Status Code가 200이 아니라면 로그인이 안되어 있는것으로 판단하여 로그인 화면으로 이동시킵니다.
   이와 함께 브라우저에 토큰이 저장되어 있던 경우 해당 토큰을 브라우저에서 제거합니다.
3. API의 응답이 정상적으로 온 경우, 응답 값에 있는 사용자의 정보를 사용하여 화면을 표시합니다.

▼ 로그인 화면이 추가됐습니다.

1. 브라우저에 토큰이 저장되어 있는 상태라면 이미 로그인을 했던것으로 간주하여 상품 목록 페이지로 이동시킵니다.
2. 로그인 버튼을 누르는 경우 `POST /auth` 경로로 API를 호출합니다.
   email, password 필드로 데이터를 보냅니다.
3. 로그인이 성공하는 경우 API 응답 값으로 받은 토큰을 브라우저에 저장하고, 상품 목록 페이지로 이동시킵니다.
4. 로그인이 실패하는 경우 모달을 띄워 로그인이 실패했다고 안내합니다.

▼ 회원가입 화면이 추가됐습니다.

1. 회원가입 버튼을 누르는 경우 `POST /users` 경로로 API를 호출합니다.
   email, nickname, password, confirmPassword 필드로 데이터를 보냅니다.
2. 회원가입에 성공한 경우 로그인 페이지로 이동시킵니다.
3. 회원가입에 실패한 경우 API 응답 값 중 "errorMessage" 필드에 있는 메세지를 보여줍니다.

▼ 3) 기능을 구현하기 전에 무엇이 필요한지 생각해보기!

- 일단 JWT 토큰을 만들기 위한 라이브러리가 하나 필요하겠죠?
- 위에서 비유했듯이 회원권 구매와, 놀이공원 입장, 내 정보 조회를 위한 기능을 만들어야겠죠?
  - 로그인 API
  - 회원가입 API
  - 내 정보 조회 API
- **로그인 확인**에 해당하는 기능은 미들웨어로 구현해서 여러 라우터에서 공용으로 사용하도록 하겠습니다!
- 혹시라도 코드가 엉켜서 동작하지 않는 상황을 방지하기 위해 기초 수업에서 구현한 부분을 제외하고 프로젝트를 새로 구성해 로그인 기능만 먼저 구현해보도록 하겠습니다!

▼ 4) User 모델 구현

mongoose를 이용해 데이터를 저장하려면 우선 모델부터 구현해야겠죠?
우선 아래 **시작 코드**를 복사해서 `app.js` 파일에 붙여넣어 준비하고, `models` 폴더를 만들고 그 안에 `user.js` 파일을 생성한 뒤 구현해 보세요!

▼ **[코드스니펫] 시작 코드**

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

mongoose.connect("mongodb://localhost/shopping-demo", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
const db = mongoose.connection;
db.on("error", console.error.bind(console, "connection error:"));

const app = express();
const router = express.Router();

app.use("/api", express.urlencoded({ extended: false }), router);
app.use(express.static("assets"));

app.listen(8080, () => {
  console.log("서버가 요청을 받을 준비가 됐어요");
});
```

▼ **[코드스니펫] User 모델 구현**

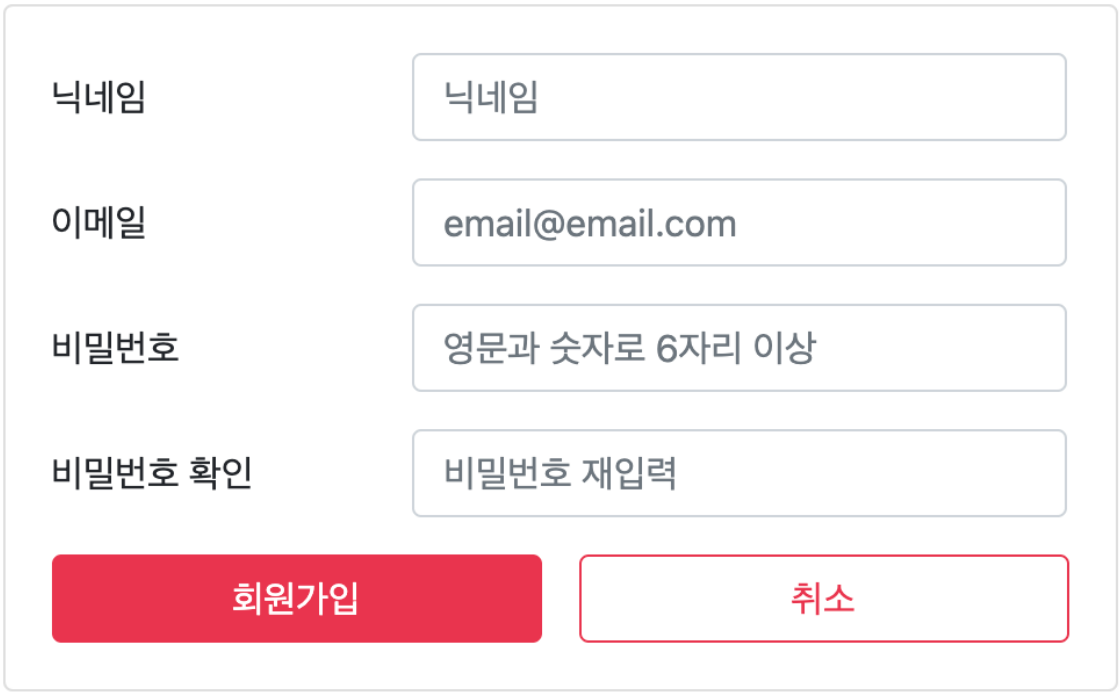
```
const mongoose = require("mongoose");

const UserSchema = new mongoose.Schema({
  email: String,
  nickname: String,
  password: String,
});
UserSchema.virtual("userId").get(function () {
  return this._id.toHexString();
});
UserSchema.set("toJSON", {
  virtuals: true,
});
module.exports = mongoose.model("User", UserSchema);
```

05. 회원가입 API 구현하기

📌 POST 메서드, `/users` 경로로 요청 받을수 있도록 구현해보세요!

▼ 1) 구현하려면 어떻게 해야 할까요?

이제 회원가입 API를 구현해볼건데요, 회원가입 할 때 프론트엔드로부터 입력받을 데이터는 4가지예요.



- `nickname`: 사용자가 사용하려는 닉네임 값
- `email`: 사용자가 사용하려는 이메일 값
- `password`: 사용자가 사용하려는 패스워드 값
- `confirmPassword`: 패스워드 확인 값 (잘못 입력한 채 가입되는것을 방지하기 위해 입력받아요!)

이 값을 입력받아 데이터베이스에 사용자의 정보를 저장하도록 하면 되겠죠?

비밀번호와 비밀번호 확인 값이 같은지 검사 → 닉네임,이메일 중복 데이터 검사 → 중복 데이터 없는 경우 사용자

단, 비밀번호 확인 데이터는 Validation을 위해서만 입력받고 데이터베이스에 저장할 필요는 없습니다!
그리고 닉네임이나 이메일이 다른 사용자와 중복되지 않도록 검사를 해주고, 만약 겹친다면 아래 **에러 메세지 응답 예시**를 참고해서 에러 메세지를 반환하도록 해보세요!

▼ 2) 에러 메세지 응답 예시

```
res.status(400).send({
  errorMessage: "상황에 맞는 에러 메세지를 입력해보세요!",
});
```

▼ 3) 회원가입 기능 구현 예시

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

// 회원가입 API
router.post("/users", async (req, res) => {
  const { email, nickname, password, confirmPassword } = req.body;

  if (password !== confirmPassword) {
    res.status(400).send({
      errorMessage: "패스워드가 패스워드 확인란과 다릅니다.",
    });
```

```
    return;
  }

  // email or nickname이 동일한게 이미 있는지 확인하기 위해 가져온다.
  const existsUsers = await User.findOne({
    $or: [{ email }, { nickname }],
  });
  if (existsUsers) {
    // NOTE: 보안을 위해 인증 메세지는 자세히 설명하지 않는것을 원칙으로 한다: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Che
    res.status(400).send({
      errorMessage: "이메일 또는 닉네임이 이미 사용중입니다.",
    });
    return;
  }

  const user = new User({ email, nickname, password });
  await user.save();

  res.status(201).send({});
});
```

▼ 4) ✅ `res.status(201).send({});` 코드에 대한 설명

이 코드는 Response의 Status Code를 201로 지정하고, `{}` 라는 데이터를 응답 값으로 보내는 코드예요!

Status code가 201인 이유는 REST API의 원칙에 따라 **Created라는 의미**로 지정했고, 응답 값으로 `{}` 를 보낸 이유는... 특별히 보내줄 데이터가 없었기 때문입니다 😅
그래서 여러분 마음대로 보내도 상관 없어요!

▼ 5) ✅ 패스워드를 그대로 저장하는것은 보안에 좋지 않아요!

이번 수업에서는 "로그인 기능"을 구현하는것이 목적이기 때문에 특별히 암호화에 대한 설명을 생략했지만, 실제로 운영할 서비스에서 이렇게 구현하는것은 보안에 아주 위협적인 요소가 됩니다!

행정안전부에서도 이와 관련된 내용에 대한 가이드라인을 제공하고 있으니 꼭 유의하길 바랍니다!
관련 글 참고

06. 로그인 API 구현하기

📌 POST 메서드, `/auth` 경로로 요청 받을수 있도록 구현해보세요!

▼ 1) 구현하려면 어떻게 해야 할까요?

이메일, 패스워드 값을 입력받아 데이터베이스에 있는 정보중 이메일 및 패스워드가 일치하면 로그인이 성공했다고 판단할 수 있습니다.

로그인에 성공하면 JWT 토큰 구성 예시와 같은 방법으로 데이터를 반환하면 됩니다.
그리고 응답 값 예시와 같은 양식으로 반환하면 해당 토큰을 프론트엔드가 받아서 사용하도록 만들어놨어요!

▼ 2) JWT 토큰 구성 예시

```
const token = jwt.sign({ userId: user.userId }, "customized-secret-key");
```

▼ 3) 응답 값 예시

```
res.send({
  token: "JWT로 만들어진 토큰을 반환하게 해보세요!"
});
```

▼ 4) 로그인 API 구현 예시

```
const jwt = require("jsonwebtoken");

router.post("/auth", async (req, res) => {
  const { email, password } = req.body;

  const user = await User.findOne({ email });
```

```
  // NOTE: 인증 메세지는 자세히 설명하지 않는것을 원칙으로 한다: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.ht
  if (!user || password !== user.password) {
    res.status(400).send({
      errorMessage: "이메일 또는 패스워드가 틀렸습니다.",
    });
    return;
  }

  res.send({
    token: jwt.sign({ userId: user.userId }, "customized-secret-key"),
  });
});
```

▼ 5) ✅ 왜 경로가 `/auth` 인가요?

Authenticate를 줄인 단어인데, 로그인 한다는 행위 자체를 "사용자가 자신의 정보를 인증한다" 라고 보기 때문에 일반적으로 로그인에 자주 사용되는 경로 중 하나입니다!

그리고 이번 수업 때 쓰이는 프론트엔드에서 이 주소로 데이터를 보내도록 구현해놨기 때문에 다른 경로로 구현하면 동작하지 않을거예요 😢

▼ 6) ✅ 왜 POST 메서드로 구현하나요?

1. 보안 관점: GET 메서드는 데이터를 URL에 표현해야 하기 때문에 보안에 취약합니다.
2. REST API 관점: 인증 정보를 "생성"해서 받아온다고 보면 POST 메서드가 적합합니다.

07. 사용자 인증 미들웨어 구현하기

▼ 1) 구현하려면 어떻게 해야 할까요?

미들웨어를 구현하는건 기초 수업에서 이미 해본적이 있을거예요!

지금 구현된 프론트엔드는 로그인이 성공했을때 받아온 토큰을 HTTP header에 아래와 같이 넣어서 보내고 있어요!
`Authorization: Bearer JWT토큰내용`

위와 같은 양식으로 보내는 이유는 HTTP 인증 유형중, Bearer 타입을 사용하여 토큰을 전달하기 위함입니다.

"Authorization" 헤더로 전달받는 토큰이 유효한지 검사하고, 만약 유효하다면 토큰 안에 있는 userId 데이터로 해당 사용자가 데이터베이스에 실제로 존재하는지 체크하면 됩니다.

▼ JWT + Bearer 인증 유형에 대한 T.M.I.

지금처럼 발급한 JWT 토큰은 OAuth 2.0 인증으로 발급한 액세스 토큰이 아니기 때문에 공식적으로 이 방법은 "비표준" 방식으로 볼 수 있습니다.
그렇지만 토큰을 헤더로 교환 할 목적으로 사용할 인증 유형이 Bearer가 제일 적절하다는 점에서 많이 사용되는 방식입니다.

비슷한 내용을 다룬 블로그 글이 있으니 참고해주세요!
→ https://velog.io/@city7310/백엔드가-이정도는-해줘야-함-5.-사용자-인증-방식-결정

▼ 2) 사용자 인증 미들웨어 구현 예시

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

module.exports = (req, res, next) => {
  const { authorization } = req.headers;
  const [authType, authToken] = (authorization || "").split(" ");

  if (!authToken || authType !== "Bearer") {
    res.status(401).send({
      errorMessage: "로그인 후 이용 가능한 기능입니다.",
    });
    return;
  }

  try {
    const { userId } = jwt.verify(authToken, "customized-secret-key");
    User.findById(userId).then((user) => {
      res.locals.user = user;
      next();
    });
  } catch (err) {
    res.status(401).send({
      errorMessage: "로그인 후 이용 가능한 기능입니다.",
    });
```

```
  }
};
```

▼ 3) ✅ `res.locals.user = user;` 는 무슨 코드인가요?

우리는 토큰에 담긴 userId로 해당 사용자가 실제로 존재하는지 확인했습니다.

이미 데이터베이스에서 사용자 정보를 가져온것이죠.
이 미들웨어를 사용하는 라우터에서는 굳이 데이터베이스에서 사용자 정보를 가져오지 않게 할 수 있도록 express가 제공하는 안전한 변수에 담아두고 언제나 꺼내서 사용할 수 있게 작성했습니다!

이렇게 담아둔 값은 정상적으로 응답 값을 보내고 나면 소멸하므로 해당 데이터가 어딘가에 남아있을 걱정의 여지를 남겨두지 않게 됩니다 😎

08. 내 정보 조회 API 구현하기

▼ 1) 내 정보 조회 API 구현하기

> 📌 GET 메서드, `/users/me` 경로로 요청 받을수 있도록 구현해보세요!

▼ 구현하려면 어떻게 해야 할까요?

우선 이 API는 사용자가 토큰을 헤더에 담아서 보내야 동작하는 API여야 합니다.
로그인을 해야 "내 정보"를 조회할 수 있게 되니까요 🙂

여기서 이미 구현해둔 사용자 인증 미들웨어를 사용하면 되겠죠?

아래 응답 값 예시를 참고하여 구현해주세요!

▼ 응답 값 예시

```
{
  "user": {
    "email": "test@email.com",
    "nickname": "test-nickname",
  }
}
```

▼ **[코드스니펫] 내 정보 조회 구현**

```
const authMiddleware = require("./middlewares/auth-middleware");

router.get("/users/me", authMiddleware, async (req, res) => {
  res.send({ user: res.locals.user });
});
```

▼ 2) 마무리 하면서...

지금까지 로그인 기능을 직접 구현해보았는데요, 사실 라이브러리를 사용하면 다른 사람이 이미 잘 구현해둔 코드를 이용하기 때문에 이번 수업처럼 우리가 굳이 작성하지 않아도 되는 코드가 많습니다!

다만, 그 동작 원리나 개념에 대한 이해가 없이 라이브러리를 사용하는 경우 나중에 독이 될 수 있기도 하죠 😈
그렇기 때문에 저는 여러분과 함께 로그인 기능을 직접 구현해보고, 라이브러리를 사용하더라도 이해한 상태로 사용할 수 있도록 여기까지 같이 배워봤습니다!

09. 2주차 끝 & 숙제 설명

- 이미 구현해 둔 API들의 입력 값을 Validation 라이브러리인 Joi를 이용해 검증하기
- 새로 구성한 프로젝트에 기존 기능들을 옮겨놓고 사용자 인증 기능 구현하기

  ▼ 어떤 API를 옮겨야 하는지 기억이 안나시나요?

  - 상품 관련 API

- 장바구니 관련 API

▼ 어떻게 사용자 인증 기능을 구현할까요?

> 👉 내 정보 조회 API를 구현할때 미들웨어를 사용했던 것처럼, 각 API 핸들러 옆에 미들웨어를 추가해주면 됩니다!

▼ ✅ 상품, 장바구니 관련 API 코드가 없으신가요?!

여러가지 이유로 이전에 개발했던 코드가 없으신 분들을 위해 제가 준비해두었습니다!

▼ app.js

```
const express = require("express");
const mongoose = require("mongoose");
const jwt = require("jsonwebtoken");
const User = require("./models/user");
const Goods = require("./models/goods");
const Cart = require("./models/cart");
const authMiddleware = require("./middlewares/auth-middleware");

mongoose.connect("mongodb://localhost/shopping-demo", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
const db = mongoose.connection;
db.on("error", console.error.bind(console, "connection error:"));

const app = express();
const router = express.Router();

router.post("/users", async (req, res) => {
  const { nickname, email, password, confirmPassword } = req.body;

  if (password !== confirmPassword) {
    res.status(400).send({
      errorMessage: "패스워드가 패스워드 확인란과 동일하지 않습니다.",
    });
    return;
  }

  const existUsers = await User.find({
    $or: [{ email }, { nickname }],
  });
  if (existUsers.length) {
    res.status(400).send({
      errorMessage: "이미 가입된 이메일 또는 닉네임이 있습니다.",
    });
    return;
  }

  const user = new User({ email, nickname, password });
  await user.save();

  res.status(201).send({});
});

router.post("/auth", async (req, res) => {
  const { email, password } = req.body;

  const user = await User.findOne({ email, password }).exec();

  if (!user) {
    res.status(400).send({
      errorMessage: "이메일 또는 패스워드가 잘못됐습니다.",
    });
    return;
  }

  const token = jwt.sign({ userId: user.userId }, "my-secret-key");
  res.send({
    token,
  });
});

router.get("/users/me", authMiddleware, async (req, res) => {
  const { user } = res.locals;
  res.send({
    user,
  });
});
```

```
/**
 * 내가 가진 장바구니 목록을 전부 불러온다.
 */
router.get("/goods/cart", authMiddleware, async (req, res) => {
  const { userId } = res.locals.user;

  const cart = await Cart.find({
    userId,
  }).exec();

  const goodsIds = cart.map((c) => c.goodsId);

  // 루프 줄이기 위해 Mapping 가능한 객체로 만든것
  const goodsKeyById = await Goods.find({
    _id: { $in: goodsIds },
  })
    .exec()
    .then((goods) =>
      goods.reduce(
        (prev, g) => ({
          ...prev,
          [g.goodsId]: g,
        }),
        {}
      )
    );

  res.send({
    cart: cart.map((c) => ({
      quantity: c.quantity,
      goods: goodsKeyById[c.goodsId],
    })),
  });
});

/**
 * 장바구니에 상품 담기.
 * 장바구니에 상품이 이미 담겨있으면 갯수만 수정한다.
 */
router.put("/goods/:goodsId/cart", authMiddleware, async (req, res) => {
  const { userId } = res.locals.user;
  const { goodsId } = req.params;
  const { quantity } = req.body;

  const existsCart = await Cart.findOne({
    userId,
    goodsId,
  }).exec();

  if (existsCart) {
    existsCart.quantity = quantity;
    await existsCart.save();
  } else {
    const cart = new Cart({
      userId,
      goodsId,
      quantity,
    });
    await cart.save();
  }

  // NOTE: 성공했을때 응답 값을 클라이언트가 사용하지 않는다.
  res.send({});
});

/**
 * 장바구니 항목 삭제
 */
router.delete("/goods/:goodsId/cart", authMiddleware, async (req, res) => {
  const { userId } = res.locals.user;
  const { goodsId } = req.params;

  const existsCart = await Cart.findOne({
    userId,
    goodsId,
  }).exec();

  // 있든 말든 신경 안쓴다. 그냥 있으면 지운다.
  if (existsCart) {
    existsCart.delete();
  }

  // NOTE: 성공했을때 딱히 정해진 응답 값이 없다.
  res.send({});
});

/**
```

```
 * 모든 상품 가져오기
 * 상품도 몇개 없는 우리에겐 페이지네이션은 사치다.
 * @example
 * /api/goods
 * /api/goods?category=drink
 * /api/goods?category=drink2
 */
router.get("/goods", authMiddleware, async (req, res) => {
  const { category } = req.query;
  const goods = await Goods.find(category ? { category } : undefined)
    .sort("-date")
    .exec();

  res.send({ goods });
});

/**
 * 상품 하나만 가져오기
 */
router.get("/goods/:goodsId", authMiddleware, async (req, res) => {
  const { goodsId } = req.params;
  const goods = await Goods.findById(goodsId).exec();

  if (!goods) {
    res.status(404).send({});
  } else {
    res.send({ goods });
  }
});

app.use("/api", express.urlencoded({ extended: false }), router);
app.use(express.static("assets"));

app.listen(8080, () => {
  console.log("서버가 요청을 받을 준비가 됐어요");
});
```

▼ models/goods.js

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

const GoodsSchema = new mongoose.Schema({
  name: String,
  thumbnailUrl: String,
  category: String,
  price: Number,
});
GoodsSchema.virtual("goodsId").get(function () {
  return this._id.toHexString();
});
GoodsSchema.set("toJSON", {
  virtuals: true,
});
module.exports = mongoose.model("Goods", GoodsSchema);
```

▼ models/cart.js

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

const CartSchema = new mongoose.Schema({
  userId: String,
  goodsId: String,
  quantity: Number,
});
module.exports = mongoose.model("Cart", CartSchema);
```

▼ 만약 상품 데이터가 하나도 없다면? → MongoDB에 직접 추가하는 방법

1. 터미널 실행
2. `mongo` 입력하여 mongo client 실행
3. `use shopping-demo` 입력
4. `db.goods.insertOne({ "name": "상품", "thumbnailUrl": "` https://cdn.pixabay.com/photo/2014/08/26/19/19/wine-428316_1280.jpg `", "category": "drink", "price": 0.11 })` 입력
5. `exit` 입력해서 mongo client 종료

- Docker Desktop 프로그램 설치해두기 (이미 설치했다면 생략!)
- 암호화가 무엇인지 조사해보고, 실제로 우리가 자주 사용하게 될 암호화는 무엇인지 알아보기
  - 단방향 암호화와 양방향 암호화의 차이점도 함께!
- [선택] passport.js 라이브러리 사용하여 로그인 기능 다시 구현해보기