Unit 5 25
사용자 인증 추가
(최종 프로젝트의 사용자 인증)

UT-NodeJS / 06.02.2023

ut-nodejs.github.io

25

캡스톤 프로젝트 5

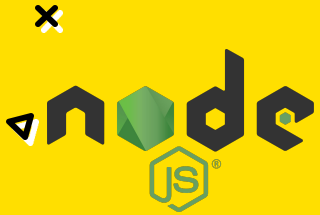
사용자 인증 추가
(최종 프로젝트의 사용자 인증)

p. 361-374

사용자 인증 추가

이 캡스톤 프로젝트에서 Passport.js를 사용하는 몇 가지 패키지를 추가해 사용자 로그인 프로세스의 보안을 설정하는 것이다. 또한 플래시 메시지를 추가해 사용자들이 리디렉션 및 페이지 렌더링 후 마지막 작업의 성공 여부를 알 수 있도록 할 것이다. 그 후 express-validator 미들웨어 패키지의 도움으로 몇 가지 유효성 체크를 추가할 것이다.

① 준비 작업



마지막 캡스톤 작업에서 작성한 코드(21 장) 에서 CRUD 작업으로 3가지 모델을 구현했다.

- **express-session**은 애플리케이션과 사용자의 연동에 대한 임시 데이터를 저장할 수 있게 한다. 여기서 저장된 세션을 통해 최근에 로그인한 사용자에 대한 정보를 알수있다 .
- **cookie-parser**는 클라이언트에 세션 데이터를 저장하게 한다. 결과 쿠키는 각 요청과 응답과 같이 보내지며 마지막으로 클라이언트를 사용한 사용자를 나타내는 데이터와 메시지 안에 담긴다.
- **connect-flash**는 사용자의 브라우저에서 플래시 메시지를 생성하는 쿠키나 세션을 쓸 수 있게 한다.
- **express-validator**는 미들웨어 함수를 통해 유입되는 데이터의 유효성 체크 계층을 추가 할 수 있게 한다.
- **passport**는 사용자 모델을 위한 암호화와 인증 프로세스를 설정한다.
- **passport-local-mongoose**는 사용자 모델에서 사용할 수 있는 플러그인을 통해 작성해야 하는 코드를 간결화시켜 passport와 더 깊숙이 연계되도록 한다.

```
npm install
express-session
cookie-parser
connect-flash
express-validator
passport
passport-local-mongoose
```

② 로그인 폼의 생성

만들어질 로그인 폼은 email과 password를 입력받게 하려고 한다. login.ejs 뷰를 users 폴더에 생성하고 다음 코드를 추가한다. 이 폼은 POST 요청을 /user/login 라우트로 보낸다.

Listing 25.1 /users/logine.js에 로그인 폼의 추가

```
<form class="form-signin" action="/users/login" method="POST">        ← 로그인 폼 생성
  <h2 class="form-signin-heading">Login:</h2>
  <label for="inputEmail" class="sr-only">Email</label>
  <input type="text" name="email" id="inputEmail" class="form-
➥   control" placeholder="Email" autofocus required>
  <label for="inputPassword" class="sr-only">Password</label>
  <input type="password" name="password" id="inputPassword"
➥   class="form-control" placeholder="Password" required>
  <button class="btn btn-lg btn-primary btn-block" type="submit">
➥   Login</button>
</form>
```

② 로그인 폼의 생성

[노트] router 객체에 모든 라우팅 관련 코드들을 추가한다.

Listing 25.2 main.js에서 로그인 라우트 추가

```
router.get("/users/login", usersController.login);
router.post("/users/login", usersController.authenticate);
router.get("/users/logout", usersController.logout,
  usersController.redirectView );
```

login 라우트 추가

authenticate 액션으로 POST 데이터 전달

logout 라우트 추가 및 뷰로 리디렉션

Listing 25.3 userControllers.js에 login 액션 추가

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

브라우저에서 폼의 렌더링을 위한 액션 추가

③ **Passport**를 이용한 암호화 추가

Passport.js를 사용하려면 먼저 passport 모듈을 **main.js와 usersController.js**에서 **`const passport = require("passport")`** 로 요청해야 한다.
이 파일들은 해싱과 인증을 설정할 파일이다.

passport는 세션과 쿠키를 사용하기 때문에 **express-session과 cookie-parser**도 main.js에서 요청해야 하다.

passport를 사용하기 위해 먼저 cookieParser를 비밀 키로 설정해 클라이언트에 저장된 쿠키를 암호화해야 한다. 그런 다음 Express.js에서 세션을 사용하게 할 것이다.

기본 로그인 스트래티지를 설정한다. 이는 **passport-local-mongoose** 모듈을 통해 제공되며, passport로 사용자 인증을 하기 위해 나중에 사용자 모델이 추가될 것이다. 마지막 2 개 라인은 passport가 서버와 클라이언트 사이로 전달되는 사용자 데이터를 압축 및 암호화, 복호화를 하게 한다.

③ Passport를 이용한 암호화 추가

Listing 25.4 passport를 Express.js와 main.js에 추가

```
const passport = require("passport"),
  cookieParser = require("cookie-parser"),
  expressSession = require("express-session"),
  User = require("./models/user");

router.use(cookieParser("secretCuisine123"));
router.use(expressSession({
  secret: "secretCuisine123",
  cookie: {
    maxAge: 4000000
  },
  resave: false,
  saveUninitialized: false
}));

router.use(passport.initialize());
router.use(passport.session());
passport.use(User.createStrategy());
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

비밀 키로 cookieParser 설정

세션 사용을 위한 Express.js 설정

passport 사용 및 초기화를 위한 Express.js 설정

passport에게 세션을 사용하도록 함

기본 로그인 스트래티지 설정

사용자 데이터의 암호화, 복호화, 압출을 위한 passport의 설정

[노트] 여기서 주의해야 할 점은, 사용자 모델은 createStrategy 메소드를 사용할 수 있기 전에 main.js에서 요정돼야 한다는 것이다. 이 메소드는 passport-local-mongoose 모델로 사용자 모델이 설정된 후에 동작한다.

③ Passport를 이용한 암호화 추가

이 설정이 끝나면 사용자 모델을 user.js에서 삭제할 수 있어 passort-local-mongoose를 추가할 수 있다. user.js의 최상단에 **`const passportLocalMongoose = require("passport-local-mongoose")`** 를 추가해 사용자 모델에 passport-local-mongoose를 요청해야 한다.

이 파일에서 userSchema에 이 모듈을 플러그인으로서 추가하다. 이 코드는 passportLocalMongoose를 설정해 **salt**와 **hash** 필드를 데이터베이스 내 사용자 모델을 위해 만든다. 또 여기서는 email 속성을 로그인 시 인증을 위한 유효성 평가 필드로 사용한다.
이 코드는 module.export 라인 전에 위치해야 한다.

Listing 25.5 User 모델에 플러그인으로서 passport-local-mongoose의 추가

```
userSchema.plugin(passportLocalMongoose, {
  usernameField: "email"
});
```

사용자 스키마 플러그인으로서
passport-local-mongoose의 추가

[노트] User 모델 추가를 통해 더 이상 사용자 스키마에 평문 패스워드 속성을 사용할 필요가 없어졌다. 이에 사용자의 show 페이지에서 매스워드 행뿐만 아니라 속성 자체에 패스워드를 삭제하도록 하겠다.

④ 플래시 메시징 추가

사용자로의 응답과 요청에 데이터를 추가할 쿠키와 세션이 준비됨에 따라, 이를 connectflash를 사용해 플래시 메시지와 연결할 준비가 됐다.

main.js 내에 **`const connectFlash = require("connect-flash");`** 와 **`router.use(connectFlash());`** 를 추가하다.

이제 미들웨어가 세팅돼 애플리케이션에서 어떤 요청상에서도 플래시 메시지를 쓸 수 있게 됐다.

Express.js 앱이 이 사용자 정의 미들웨어를 갖고 컨트롤러 액션에서 생성된 플래시 메시지를 내포하고 있는 객체들에 flashMessages 라고 하는 로컬 변수를 할당할 수 있다. 그러면 뷰 페이지가 flashMessages 객체에 접근할 수 있게 된다.

Listing 25.6 main.js에서 플래시 메시지 사용을 위한 사용자 정의 미들웨어 추가

```
router.use((req, res, next) => {
  res.locals.flashMessages = req.flash();
  next();
});
```

로컬 변수로
플래시 메시지 추가

④ 플래시 메시징 추가

모든 페이지에 플래시 메시지를 보여주고 싶기 때문에 메시지가 존재한다면 이를 찾아 보여주기 위해 layout.ejs 에 코드를 추가할 것이다.

메시지는 모든 메시지가 아닌 **success** 와 **error** 메시지만 보여주려 한다.

Listing 25.7 layout.ejs에서 플래시 메시지를 사용하기 위한 로직 추가

```
<div class="flashes">
<% if (flashMessages) { %>
  <% if (flashMessages.success) { %>
    <div class="flash success"><%= flashMessages.success %></div>
  <% } else if (flashMessages.error) { %>
    <div class="flash error"><%= flashMessages.error %></div>
  <% } %>
<% } %>
</div>
```

뷰에서
플래시 메시지 출력

④ 플래시 메시징 추가

Listing 25.8 create 액션에서 passport 등록과 플래시 메시지 추가

```
create: (req, res, next) => {
  if (req.skip) next();

  let newUser = new User(getUserParams(req.body));

  User.register(newUser, req.body.password, (e, user) => {
    if (user) {
      req.flash("success", `${user.fullName}'s account
➥ created successfully!`);
        res.locals.redirect = "/users";
      next();
    } else {
      req.flash("error", `Failed to create user account
➥ because: ${e.message}.`);
      res.locals.redirect = "/users/new";
      next();
    }
  });
}
```

사용자 등록을 위한 create 액션 추가

플래시 메시지로 응답

마지 막으로 usersController.js 에 다음 코드를 추가해 사용자의 create 액션을 수정, passport와 플래시 메시지를 사용해 추가된 코드를 테스트한다. create 액션은 Passport.js에서 제공하는 register 메소드를 사용해 새로운 사용자 계정을 만든다. 그 결과는 해싱된 패스워드와 솔트가 같이 데이터베이스에 사용자 도큐먼트로 저장된다.

⑤ 유효성 체크 미들웨어 추가

express-validator 모듈은 데이터가 애플리케이션에 쓰이기 전에 유효성 체크 및 세니타이정을 위한 쓸 만한 메소드를 제공한다.

우선 main.js 에 **`const expressValidator = require("express-validator");`**를 추가하고 Express 애플리케이션으로 하여금 이 모듈을 미들웨어로 사용하도록 하기 위해 같은 파일에 **`router.use(expressValidator());`** 를 추가한다.

데이터가 userController에 있는 create 에 도달하기 전에 몇 개의 미들웨어를 통과시켜 유효성 체크를 하려 한다. 따라서 /users/create 라우트를 해당 요구 사항을 고려해 변경한다. 이 validate 액션은 userController에 존재하며 create 액션 전에 실행된다. 이 사용자 정의 유효성 체크 미들웨어는 잘못된 데이터가 사용자모델에 도달하기 전에 필터링을해준다.

Listing 25.9 main.js에서 create 전에 유효성 체크 추가

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

사용자 생성 라우트에
유효성 체크 미들웨어 추가

⑤ 유효성 체크 미들웨어 추가

Listing 25.10 userController.js에서 validate 액션의 추가

```
validate: (req, res, next) => {          ← validate 액션 추가
  req
    .sanitizeBody("email")
    .normalizeEmail({
      all_lowercase: true
    })
    .trim();
  req.check("email", "Email is invalid").isEmail();
  req
    .check("zipCode", "Zip code is invalid")
    .notEmpty()
    .isInt()
    .isLength({
      min: 5,
      max: 5
    })
    .equals(req.body.zipCode);           ← input 필드 데이터 체크 및 새니타이징
  req.check("password", "Password cannot be empty").notEmpty();
  req.getValidationResult().then((error) => {
    if (!error.isEmpty()) {
      let messages = error.array().map(e => e.msg);
      req.skip = true;
      req.flash("error", messages.join(" and "));
      res.locals.redirect = '/users/new';   ← 에러를 수집하고 플래시 메시지로 출력
      next();
    } else {
      next();
    }
  });
}
```

이 validate 액션은 유입 요청들을 파싱하고 요청 본문 내 데이터를 체크한다. 이번 경우에는 first 및 last 이름 필드 값 내에 있는 공백을 삭제한다.

이메일의 데이터베이스와의 일관성과 우편번호에 요구되는 길이 유지를 위해 expressValidator에서 제공하는 메소드 일부를 사용한다. 사용자가 등록 시 패스워드 입력 여부를 체크할 것이다. 그리고 유효성 체크 과정에서 발생하는 오류들을 모은다.

그런 다음 에러 메시지들을 **하나의 문자열로** 이어 붙인다. 요청 객체의 설정을 **`req.skip = true`**로 해 create 액션을 건너뛰고 바로 뷰로 되돌아간다.

모든 플래시 메시지는 users/new 뷰에서 나타난다.

⑥ **Passport.js**로 인증 추가

passport-local-mongoose를 추가하면 사용자 모델은 passport가 단독으로 제공했던 것보다 더 유용한 메소드를 상속받는다. passport-local-mongoose 모듈이 플러그인으로서 사용자 모델에 추가됐기 때문에 많은 인증 설정이 보이지 않게 수행할 수 있다.

register 메소드는 passport가 제공하는 메소드들 중 가장 직관적이고 파워풀한 것 중 하나다. 이를 사용하기 위해 **passport.register**를 호출해야 하며 사용하기로 한 로그인 스트래티지를 전달해야 한다. **passport.authenticate** 메소드를 사용하기 위한 authenticate 액션을 userController.js.

[노트] **`const passport = require("passport")`**가 사용자 컨트롤러의 제일 위에 위치하는지 확인해야 한다.

이 액션은 곧바로 passport.register 메소드를 가리킨다. 이미 main.js에서 사용자 모델을 위한 로컬 스트래티지를 생성했고 passport가 사용자 데이터를 인증 성공 여부에 따라 직렬화와 역직렬화를 하도록 했다. 여기서 추가한 옵션들은 플래시 메시지와 함께 인증이 성공 또는 실패했는지에 따라 경로를 결정한다.

⑥ Passport.js로 인증 추가

passport-local-mongoose를 추가하면 사용자 모델은 passport가 단독으로 제공했던 것보다 더 유용한 메소드를 상속받는다. passport-local-mongoose 모듈이 플러그인으로서 사용자 모델에 추가됐기 때문에 많은 인증 설정이 보이지 않게 수행할 수 있다.

register 메소드는 passport가 제공하는 메소드들 중 가장 직관적이고 파워풀한 것 중 하나다. 이를 사용하기 위해 **passport.register**를 호출해야 하며 사용하기로 한 로그인 스트래티지를 전달해야 한다. **passport.authenticate** 메소드를 사용하기 위한 authenticate 액션을 userController.js.

[노트] **`const passport = require("passport")`**가 사용자 컨트롤러의 제일 위에 위치하는지 확인해야 한다.

Listing 25.11 userController.js에서 authenticate 액션의 추가

```
authenticate: passport.authenticate("local", {
  failureRedirect: "/users/login",
  failureFlash: "Failed to login.",
  successRedirect: "/",
  successFlash: "Logged in!"
})
```

플래시 메시지와 리디렉션을 포함한 인증 미들웨어 추가

이 액션은 곧바로 passport.register 메소드를 가리킨다. 이미 main.js에서 사용자 모델을 위한 로컬 스트래티지를 생성했고 passport가 사용자 데이터를 인증 성공 여부에 따라 직렬화와 역직렬화를 하도록 했다. 여기서 추가한 옵션들은 플래시 메시지와 함께 인증이 성공 또는 실패했는지에 따라 경로를 결정한다.

⑦ 로그인과 로그아웃

우리는 이미 로그인 프로세스를 정상 동작시켰고 이제 사용자가 로그인됐다는 시각적 표시를 추가하려 한다. 먼저 로그인된 사용자를 위한 만료되지 않은 세션이 있는지 알 수 있는 변수들을 설정한다.

이 미들웨어 함수로 **loggedIn** 에 액세스해 요청이 전달된 클라이언트로부터 로그인된 계정인지를 결정한다 **isAuthenticated**는 사용자의 활성화된 세션이 있는지 여부를 알려준다. **currentUser**는 사용자가 존재한다면 로그인된 사용자로 설정된다.

Listing 25.12 미들웨어를 통한 로컬 변수를 응답으로 추가

```
res.locals.loggedIn = req.isAuthenticated();
res.locals.currentUser = req.user;
```

로그인 상태를 나타내기 위한 loggedIn 변수의 설정

로그인한 사용자를 나타내기 위한 currentUser 변수의 설정

⑦ 로그인과 로그아웃

이제 레이아웃에 있는 내비게이션 바에 다음 코드를 추가해 이들 변수를 사용할 수 있게 됐다. loggedIn 이 true 인지를 체크해 사용자가 로그인 중이라면 해당사용자의 show 페이지의 링크가 걸려 있는 currentUser의 fullName를 출력한다. 그렇지 않으면 로그인 링크를 출력한다.

Listing 25.13 layout.ejs 내 내비게이션 바에 로그인 상태 추가

```
<div class="login">
  <% if (loggedIn) { %>          ← 사용자의 로그인 여부 체크
    <p>Logged in as
      <a href="<%=`/users/${currentUser._id}`%>">
  <%= currentUser.fullName %></a>
      <a href="/users/logout">Log out</a>
    </p>
  <%} else {%>
    <a href="/users/login">Log In</a>
  <% } %>
</div>
```

사용자의 fullName과 로그아웃 링크 출력

⑦ 로그인과 로그아웃

마지막으로 /users/logout 라우트가 이미 설정돼 있으면 logout 액션을 userController 에 추가해야 한다. 이 액션은 유입 요청상에서 logout 메소드를 사용한다. passport가 제공하는 이 메소드는 활성 중인 사용자 세션을 삭제한다.

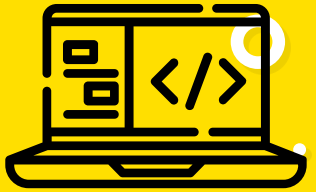
Listing 25.14 userController.js에 로그아웃 액션 추가

```
logout: (req, res, next) => {
  req.logout();
  req.flash("success", "You have been logged out!");
  res.locals.redirect = "/";
  next();
}
```

사용자 로그아웃을 위한 액션 추가

⑦ 로그인과 로그아웃

그림 25.1 Confetti Cuisine의 로그인 후 화면



Coding 과제!

캡스톤 프로젝트
(최종 프로젝트의 사용자 인증)

p. 361-374

과제 타임!

한번 해 보자!~