Unit 5 22-24

사용자 계정 인증

UT-NodeJS / 05.26.2023

ut-nodejs.github.io

Contents / 내용

22. 세션과 플래시 메시지의 추가

- 세션과쿠키 설정
- 컨트롤러 액션에서 플래시 메시지 생성
- 유입 데이터상에서 유효성 체크 미들웨어 설정

23. 사용자 로그인 폼 생성과 패스워드 해시

- 사용자로그인폼생성
- bcrypt를 통한 데이터베이스 내 데이터 해싱

24. 사용자 인증 추가

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

22

세션과 플래시 메시지의 추가

세션과쿠키 설정
컨트롤러 액션에서 플래시 메시지 생성
유입 데이터상에서 유효성 체크 미들웨어 설정

p. 323-332

22.1 플래시 메시지 모듈 설정

플래시 메시지 Flash Message는 애플리케이션 사용자에게 정보를 보여주는 반영구 데이터다. 이 메시지들은 애플리케이션 서버에서 생성돼 세션의 일부로서 사용자 브라우저에 전달된다.

세션은 최근 사용자와 애플리케이션 간 대부분의 연동 관련 데이터를 갖고 있다. 여기에는 현재 로그인 중인 사용자 페이지 타임아웃 전까지의 남은 시간, 또는 출력돼야 할 일회성 메시지 등이 있다.

```
npm install express-session connect-flash cookie-parser
```

express-session: 애플리케이션과 클라이언트사이에 메시지를 전달하기 위해서는 express-session 모듈을 설치해야 한다. 이 메시지들은 사용자의 브라우저에 유지되지만 궁극적으로는 서버에 저장된다.

cookie-parser: 쿠키는 세션 저장소 형태 중 하나이며 세션에서 브라우저에서 서버로 전달되는 쿠키
데이터를 해석하기 위해 cookie-parser 패키지가 필요하다. Express.js에 cookie-parser의 미들웨어로의 사용 및 여러분이 선택한 비밀 암호secret passcode의 사용을 선언한다. cookieparser는 이 코드를 브라우저로 전달되는 쿠키 데이터의 암호화에 사용한다.

connect-flash: 플래시 메시지를 생성하기 위해 connect-flash 패키지를 사용한다. 이 패키지는 세션과 쿠키에 종속되며 요청 간에 플래시 메시지를 전달한다.

22.1 플래시 메시지 모듈 설정

Listing 22.1 main.js에서의 플래시 메시징 요청

```
const expressSession = require("express-session"),
  cookieParser = require("cookie-parser"),
  connectFlash = require("connect-flash");
router.use(cookieParser("secret_passcode"));
router.use(expressSession({
  secret: "secret_passcode",
  cookie: {
    maxAge: 4000000 (1시간)
  },
  resave: false,
  saveUninitialized: false
}));
router.use(connectFlash());
```

← 3가지 모듈 요청

cookie-parser를 미들웨어로 사용하도록 Express.js를 설정

cookie-parser의 사용을 위한 express-session의 설정

connect-flash를 미들웨어로 사용하기 위한 애플리케이션 설정

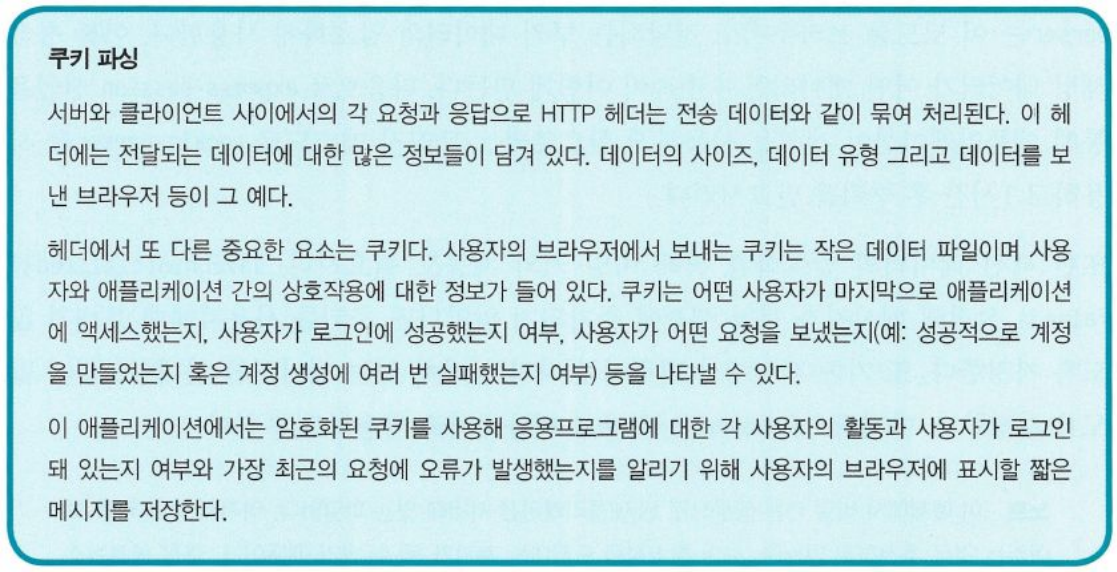
[노트] 이 예제에서 비밀 키는 평문으로 된 애플리케이션 서버에 있는 파일이다.

여기서 비밀 키를 보여주는 것은 추천하지 않는다. 보안 취약점을 드러내는 계기가 될 수 있기 때문이다.

대신 환경변수로 비밀 키를 저장하고 process.env로 해당 변수에 액세스한다.

자세한 것은 8부에서 다룬다.

22.1 플래시 메시지 모듈 설정



쿠키 파싱

서버와 클라이언트 사이에서의 각 요청과 응답으로 HTTP 헤더는 전송 데이터와 같이 묶여 처리된다. 이 헤더에는 전달되는 데이터에 대한 많은 정보들이 담겨 있다. 데이터의 사이즈, 데이터 유형 그리고 데이터를 보낸 브라우저 등이 그 예다.

헤더에서 또 다른 중요한 요소는 쿠키다. 사용자의 브라우저에서 보내는 쿠키는 작은 데이터 파일이며 사용자와 애플리케이션 간의 상호작용에 대한 정보가 들어 있다. 쿠키는 어떤 사용자가 마지막으로 애플리케이션에 액세스했는지, 사용자가 로그인에 성공했는지 여부, 사용자가 어떤 요청을 보냈는지(예: 성공적으로 계정을 만들었는지 혹은 계정 생성에 여러 번 실패했는지 여부) 등을 나타낼 수 있다.

이 애플리케이션에서는 암호화된 쿠키를 사용해 응용프로그램에 대한 각 사용자의 활동과 사용자가 로그인돼 있는지 여부와 가장 최근의 요청에 오류가 발생했는지를 알리기 위해 사용자의 브라우저에 표시할 짧은 메시지를 저장한다.

[노트] 요청들은 서로 독립적이기 때문에 만약 하나의 사용자 생성 요청이 실패한다면 홈페이지로 리디렉션되며, 리디렉션 자체가 또 하나의 요청이 된다. 하지만 사용자 생성이 실제로 성공했는지 여부는 사용자가 알 수가 없다. 이런 경우 쿠키는 상당한 도움이 된다.

22.2 컨트롤러 액션에 플래시 메시지 추가

플래시 메시지를 작동시키려면 사용자에게 뷰를 렌더링해주기 전에 만들어지는 요청에 추가해줘야 한다. 일반적으로 사용자가 페이지 **GET 요청**을 보내면 (예를 들어 홈페이지를 읽어들인다고 하면) **플래시 메시지를 보낼 필요는 없다.**

플래시 메시지는 사용자에게 성공 또는 실패를 알려주는 데 제일 유용한 도구이며, 데이터베이스와 연동된다.

플래시 메시지는 뷰를 위해 만들어진 로컬 변수와 다르지 않다. 이 때문에 Express를 위한 다른 미들웨어 설정을 해야 하며, 이를 통해 Listing 22.2에서처럼 응답상에서 connectFlash를 로컬 변수처럼 취급할 수 있다. 이 미들웨어 함수의 추가를 통해 Express가 flashMessages 라고 부르는 로컬 객체를 뷰로 전달시키고 있다.

Listing 22.2 응답상에서 connectFlash와 미들웨어와의 연계

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

응답 객체상에서 플래시 메시지의 로컬 flashMessages로의 할당

22.2 컨트롤러 액션에 플래시 메시지 추가

[노트] getUserParams는 이전 캡스톤 프로젝트 (21 장)에서 사용돼 왔다. 이 함수는 컨트롤러를 통해 재사용돼 사용자 속성을 하나의 객제로 구성한다. 동일한 함수를 다른 모델 컨트롤러에도 구성해야 한다.

```
/**
 * Listing 22.3 (p. 328)
 * userController.js에서 create액션이 플래시 메시지를 추가
 *
 * [노트] getUserParams는 이전 캡스톤 프로젝트 (21장)에서 사용돼 왔다.
 * 이 함수는 컨트롤러를 통해 재사용돼 사용자 속성을 하나의 객체로 구성한다.
 * 동일한 함수를 다른 모델 컨트롤러에도 구성해야 한다.
 */
const getUserParams = (body) => {
  return {
    name: {
      first: body.first,
      last: body.last,
    },
    email: body.email,
    username: body.username,
    password: body.password,
    profileImg: body.profileImg,
  };
};
```

22.2 컨트롤러 액션에 플래시 메시지 추가

Listing 22.3 userController.js에서 create 액션에 플래시 메시지 추가

```
create: (req, res, next) => {
  let userParams = getUserParams(req.body);
  User.create(userParams)
    .then(user => {
      req.flash("success", `${user.fullName}"s account created
➥ successfully!`);                                    ← 성공 플래시 메시지 출력
      res.locals.redirect = "/users";
      res.locals.user = user;
      next();
    })
    .catch(error => {
      console.log(`Error saving user: ${error.message}`);
      res.locals.redirect = "/users/new";
      req.flash(
        "error",
        `Failed to create user account because:
➥ ${error.message}`.                                  ← 실패 플래시 메시지 출력
      );
      next();
    });
},
```

[노트] 플래시 메시지를 임시로 저장하기 위해 요청 객체를 사용했지만, 응답에서의 로컬 변수와 이 메시지들을 연결했기 때문에 메시지들은 결국 응답 객체로 연결된다.

[노트] error와 success는 플래시 메시지를 위해 만든 2가지 타입이다. 이 타입들은 원하는 대로 커스터마이징이 가능하다. 만일 superUrgent 타입이 필요하면 req.flash("superUrgent", "Read this message ASAP!")를 사용할 수 있다. 그러면 superUrgent는 여러분이 무슨 메시지를 추가하든지 이를 얻기 위한 키로 사용될 것이다.

22.2 컨트롤러 액션에 플래시 메시지 추가

플래시 메시지 작업의 마지막 단계는 뷰에서 이를 수신하고 표현하기 위한 코드 추가다.

[노트] 만일 어떤 메시지도 화면에 보이지 않는다면 메시지에 적용한 스타일링을 제거한 평문(plain text) 메시지로 우선 출력되는지 확인하라.

Listing 22.4 layout.ejs에서 플래시 메시지 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>
```

flashMessages가 있는지 체크

success 메시지 출력

error 메시지 출력

22.2 컨트롤러 액션에 플래시 메시지 추가

페이지를 리프레시하거나 다른 새로운 요청을 만들 때 이 메시지는 사라진다. 여러 개의 success나 error 메시지를 보여주려고 하는 경우에는 뷰에 있는 모든 error나 success 키들을 한 번에 보여주기보다는 루프를 돌면서 하나씩 보여주는 게 더 유용할 것이다.

이 플래시 메시지를 렌더링하는 뷰에서 보여주고 싶다면 메시지를 로컬 변수로 바로 전달한다.

Listing 22.5 렌더링된 인덱스 뷰에서 플래시 메시지 추가

```
res.render("users/index", {
  flashMessages: {
    success: "Loaded all users!"
  }
});          ←—— 렌더링된 뷰로 플래시 메시지 전달
```

22.2 컨트롤러 액션에 플래시 메시지 추가

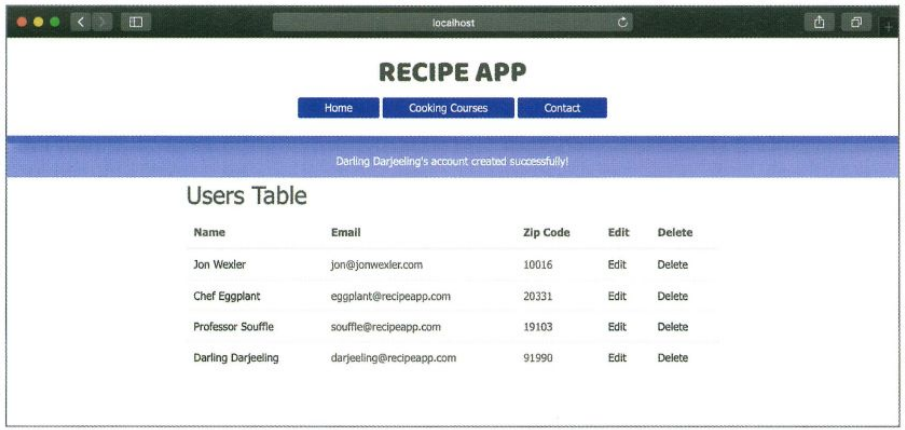


그림 22.1 /users 페이지에서 성공 플래시 메시지

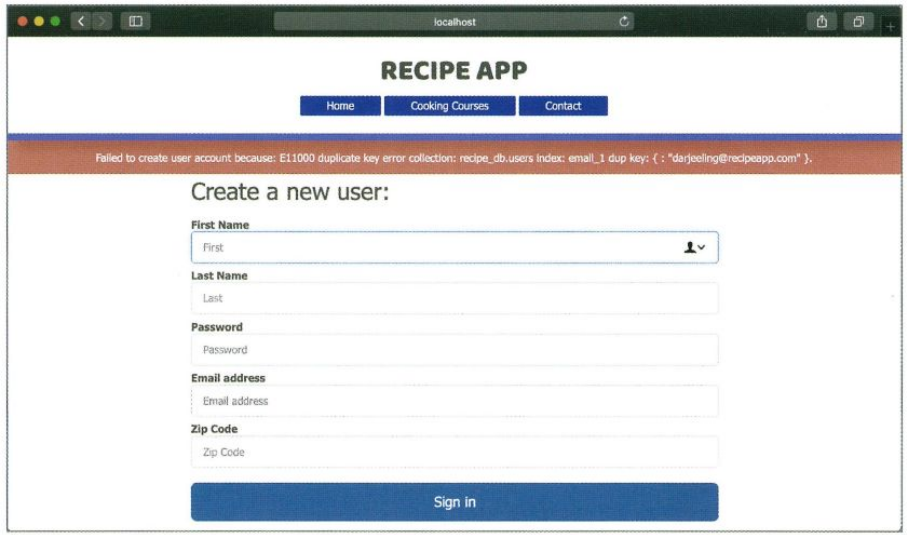
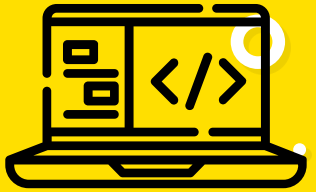


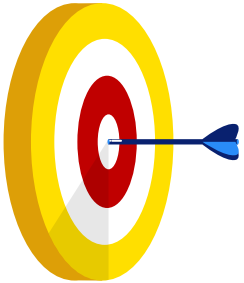
그림 22.2 홈페이지상 보이는 에러 플래시 메시지



Coding!

Listing 22.1~

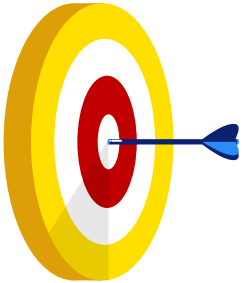
p. 323-332



퀵 체크 22.1

쿠키의 비밀 키는 브라우저에 데이터를 보내고 저장하는 방법을 어떻게 바꾸는가?

쿠키와 같이 사용되는 비밀 키는 데이터를 암호화하는 데 사용된다. 암호화는 데이터 보안을 유지하며 인터넷상으로 전달하는 데 중요하며 전달 도중 변형 없이 사용자의 브라우저로 도달하는 것을 보장한다.



퀵 체크 22.2

req.flash 메소드를 위해 필요한 2개의
매개변수는 무엇일까?

req.flash는 플래시 메시지 타입과 메시지가 필요하다.

23

사용자 로그인 폼 생성과 패스워드 해시

사용자로그인폼생성
bcrypt를 통한 데이터베이스 내 데이터 해싱

p. 333-348

23.1 사용자 로그인 폼

애플리케이션에 사용자 로그인 처리 로직을 알아보기 전에, 등록과 로그인 폼의 형태를먼저 확정하도록하자.

주목해야 할 중요한 점은 폼 태그 내의 /users/login 액션이다. 이 경로에의 POST 요청의 처리를 위한 라우트를 만들어야 한다.

Listing 23.1 login.ejs에서의 로그인 폼 생성

```
<form action="/users/login" method="POST">
  <h2>Login:</h2>
  <label for="inputEmail">Email address</label>
  <input type="email" name="email" id="inputEmail"
➥ placeholder="Email address" required>
  <label for="inputPassword">Password</label>
  <input type="password" name="password" id="inputPassword"
➥ placeholder="Password" required>
  <button type="submit">Login</button>
</form>
```

사용자 로그인을 위한 로그인 폼 추가

23.1 사용자 로그인 폼

[노트] show 및 edit 라우트 라인 위에 login 라우트를 추가하라 순서가 바뀌면 Express.js는 경로에 있는 login이라는 단어를 사용자 ID로 인식하고 이에 해당하는 사용자를 찾으려 할 것이다. 바른 순서로 라우트를 추가하면 애플리케이션은 사용자 ID를 URL에서 찾기 전에 login 라우트를 전체 경로로 인식한다.

Listing 23.2 main.js로 로그인 라우트 추가

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

/users/login으로 GET 요청 처리를 위한 라우트 추가

동일 경로로 POST 요청 처리를 위한 라우트 추가

23.1 사용자 로그인 폼

Listing 23.3 userController.js로의 로그인과 인증 액션 추가

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

authenticate: (req, res, next) => {             ← 인증 액션 추가
  User.findOne({
    email: req.body.email                       ← 데이터베이스의 패스워드와 비교
  })
    .then(user => {
      if (user && user.password === req.body.password){
        res.locals.redirect = `/users/${user._id}`;
        req.flash("success", `${user.fullName}'s logged in successfully!`);
        res.locals.user = user;
        next();
    } else {
      req.flash("error", "Your account or password is incorrect.
Please try again or contact your system administrator!");
      res.locals.redirect = "/users/login";
      next();
    }
  })                                            ← 에러를 콘솔에 로깅하고 리디렉션
    .catch(error => {
      console.log(`Error logging in user: ${error.message}`);
      next(error);
    });
}
```

login 액션은 사용자 로그인을 위한 login 뷰를 렌더링한다.

authenticate 액션은 이메일 주소와 일치하는 사용자를 찾는다. 이 속성은 데이터베이스에서 고유하기 때문에 검색 결과는 한 명이거나 없어야 한다.

그런 다음 패스워드 폼의 내용을 데이터베이스 패스워드와 비교하고 결과가 일치하면 사용자의 show 페이지로 리디렉션시킨다.

또 플래시 메시지를 사용자가 로그인에 성공함을 알려주도록 설정하고 사용자 객체를 로컬 변수로서 사용자의 show 페이지로 전달한다.

하지만 이 것으로 끝난 게 아니다. 여전히 패스워드는 평문으로 저장되고 있다.

23.2 패스워드의 해싱

암호화는 민감한 데이터를 고유 키나 암호를 결합해 오리지널 데이터를 표현하지만 사용할 수 없는 값을 만드는 것이다. 이 과정에는 데이터 해성이 포함돼 있으며, 해싱된 데이터는 해시 함수를 위한 개인 키를 사용하면 데이터를 읽어들일 수 있다.

이렇게 해싱된 값은 데이터베이스 내 공개에 민감한 데이터 대신 저장된다. **새로운 데이터를 암호화**
하려면 암호 알고리즘에 데이터를 통과시켜야 한다. 데이터를 읽어들이거나 비교 하려면 (사용자 패스워 드의 경 우를 생각하면) 애플리케이션은 동일한 고유 키와 알고리즘을 복호화 데이터에 사용할 수 있다. **bcrypt**는 잘 만들어진 해시 함수이며 패스워드 등의 데이터를 데이터베이스에 저장 시 고유 키와 조합하게 해준다.

```
npm install bcrypt
```

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

- **[노트]** 패스워드의 원래 값을 기술적으로 끄집어낼 필요는 없기 때문에 패스워드는 해싱만 하며 암호화는 하지 않는다. 사실 애플리케이션은 사용자가 어떻게 패스워드를 정했는지 알 수는 없다. 애플리케이션은 해싱된 패스워드만 갖고 있다. 나중에 (로그인 등에서) 패스워드 값을 입력받으면 이를 해싱하고 (애플리케이션이 갖고 있는) 해싱된 값과 비교하게 된다.

23.2 패스워드의 해싱

Listing 23.4 user.js에서의 pre 훅 해싱

```
userSchema.pre("save", function(next) {
  let user = this;

  bcrypt.hash(user.password, 10).then(hash => {
    user.password = hash;
    next();
  })
    .catch(error => {
      console.log(`Error in hashing password: ${error.message}`);
      next(error);
    });
});

userSchema.methods.passwordComparison = function(inputPassword){
  let user = this;
  return bcrypt.compare(inputPassword, user.password);
};
```

사용자 스키마에 pre hook 추가

사용자 패스워드의 해싱

해싱된 패스워드와 비교하는 함수 추가

저장된 패스워드와의 비교

[노트] save에서의 pre 훅은 사용자가 저장될 때마다 실행된다. 다시 말하면 Mongoose의 save 메소드를 통해 생성 또는 업데이트 후에 실행된다.

Mongoose의 pre 훅은 post 훅은 사용자가 데이터베이스에 저장되기 전후로 User 인터페이스에 코드를 실행시키기 좋은 방법을 제공한다.

bcrypt.hash 메소드는 패스워드와 숫자를 파라미터로 받는다. 이는 패스워드 해싱에서의 복잡도를 의미하며 보통 10을 준다.

[노트] bcrpyt.hash 실행 시 이 pre-hook에서는 context를 잃어버리기 때문에 this를 변수에 보존시켜 해시 함수 내에서 액세스할 수 있도록 한다.

passwordComparison은 userSchema에서의 시용자 정의 메소드이며, 폼의 input에서 받은 패스워드 값과 해싱된 패스워드를 비교한다. 이 체크를 비동기로 수행하려면 bcrypt로 프라미스 라이브러리를 사용한다.

23.2 패스워드의 해싱

Listing 23.5 userController.js에서 authenticate 액션 수정

```
authenticate: (req, res, next) => {
  User.findOne({email: req.body.email})
      .then(user => {
        if (user) {
          user.passwordComparison(req.body.password)
              .then(passwordsMatch => {
                if (passwordsMatch) {
                  res.locals.redirect = `/users/${user._id}`;
                  req.flash("success", `${user.fullName}'s logged in
➥ successfully!`);
                  res.locals.user = user;
                } else {
                  req.flash("error", "Failed to log in user account:
➥ Incorrect Password.");
                  res.locals.redirect = "/users/login";
                }
                next();
              });
        } else {
          req.flash("error", "Failed to log in user account: User
➥ account not found.");
          res.locals.redirect = "/users/login";
          next();
    }
  })
      .catch(error => {
        console.log(`Error logging in user: ${error.message}`);
        next(error);
      });
}
```

이메일로 사용자를 찾는 쿼리

사용자를 찾았는지 체크

User 모델에서 패스워드 비교 메소드 호출

패스워드가 일치하는지 체크

리디렉션될 경로와 플래시 메시지 셋을 포함한 next 미들웨어 함수의 호출

에러의 콘솔 로깅 및 next 에러 핸들러 미들웨어로 전달

마지막 단계는 usersController.js에 있는 authenticate 액션을 재작성해 패스워드와 bcrypt.compare의 비교를 하는 것이다.

이전 계정에서의 패스워드는 bcrypt로 해싱이 돼 있지 않기 때문에 테스트 전에 새로 사용자를 만들어야 한다 (이렇게 되면 기존 평문으로 저장된 패스워드와 해싱 된 입력 패스워드와 비교하기 때문이다). 계정이 생성된 다음 동일한 패스워드로 로그인을 /users/login 에서 시도한다.

[노트] 화면이 아닌 데이터베이스 레벨에서도 몽고 DB 셀의 mongo를 통해 해싱된 패스워드를 확인할 수 있다. 새로운 터미널에서 mongo 명령 입력 후에 use db와 db.users.find({})를 입력한다. 다른 방법으로 몽고 DB의 Compass를 사용해 데이터베이스의 레코드를 확인할 수도 있다.

23.3 express-validator로 유효성 체크 추가

(나중에 마침)

지금까지 애플라케이션은 뷰 및 모델 수준에서 유효성 검사를 제공했다 이메일 주소 없이 사용자 계정을 만들려고 하면 경고의 HTML 폼이 뜨며 진행이 되지 않았다.

6부에서도 볼 수 있듯이 폼을 우회하거나 누군가가 API를 통해 계정을 만들려고 하면 모델 스키마의 제한으로 더 많은 유효성 체크가 작동하기 전에 데이터가 데이터베이스에 유입되는 것을 막을 수 있다 실제로 애플리케이션에서 모델에 도달하기 전에 더 많은 유효성 검사를 추가할 수 있으면 Mongoose 쿼리 및 페이지 리디렉션에 소요되는 많은 컴퓨팅 시간과 에너지를 절약할 수 있다. **이러한 이유로 미들웨어의 유효성을 체크할 것이다.**

```
npm install express-validator@5.3.1
```

패키지가 설치되면 main.js 에서 **const expressValidator = require (" expressvalidator");** 를 통해 요청하고 **router.use(expressValidator())** 로 Express.js에 사용을 선언한다. 이는 express.json() 및 express.urlencoded() 다음에 추가해야 한다. 유효성 체크전에 요청 본문에 먼저 파싱돼야하기 때문이다.

Listing 23.6 main.js에서 사용자 생성 라우트에 유효성 체크 미들웨어 추가

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

사용자 생성 라우트에
유효성 체크 미들웨어 추가

23.3 express-validator로 유효성 체크 추가

(나중에 마침)

Listing 23.7 userController.js에서 validate 컨트롤러 생성

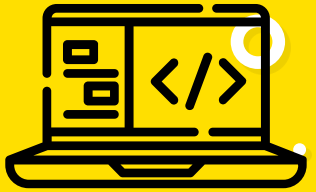
```
validate: (req, res, next) => {          ← validate 함수 추가
  req.sanitizeBody("email").normalizeEmail({
    all_lowercase: true                  ← trim()으로 whitespace 제거
    }).trim();
  req.check("email", "Email is invalid").isEmail();
  req.check("zipCode", "Zip code is invalid")
.notEmpty().isInt().isLength({
    min: 5,
    max: 5                               ← zipCode 값의 유효성 체크
  }).equals(req.body.zipCode);
  req.check("password", "Password cannot be empty").notEmpty();   ← password 필드 유효성 체크

  req.getValidationResult().then((error) => {   ← 앞에서의 유효성 체크 결과 수집
    if (!error.isEmpty()) {
      let messages = error.array().map(e => e.msg);
      req.skip = true;                   ← skip 속성을 true로 설정
      req.flash("error", messages.join(" and "));   ← 에러를 플래시 메시지로 추가
      res.locals.redirect = "/users/new";
      next();                            ← new 뷰로 리디렉션 설정
    } else {
      next();
    }                                    ← 다음 미들웨어 함수 호출
  });
}
```

마지막으로 create 액션에 도달하기 전에 요청을 처리하기 위한 validate 액션을 userController.js에서 만들어야한다. 이 액션에서 다음사항을추가한다.

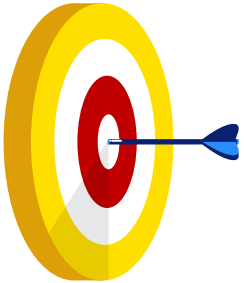
- **밸리데이터 Validator:** 유입 데이터가 일정 기준을 만족하는지 체크
- **새니타이저 Sanitizer:** 데이터베이스로 저장하기 전에 불필요한 데이터의 삭제나 데이터 타입 캐스팅 수행

[노트] 폼 데이터를 다시 채우기 위해 다양한 방법을 선택할 수 있다 (연구해보면) 어떤 매키지가 효과적인지 알게 될 것이다 자신에게 가장 적합한 방법을 찾으면 데이터를 다시 처리하도록 애플리케이션의 모든 폼을 변경하라.



Coding!

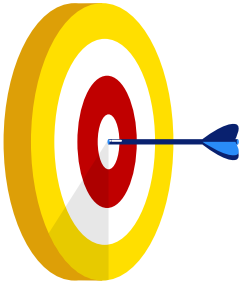
Listing 23.1~
p. 333-348



퀵 체크 23.1

왜 main.js에서 /users/login 라우트의
위치가 문제될까?

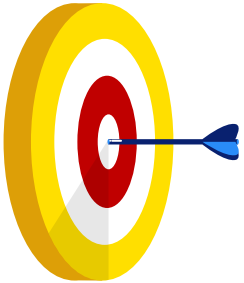
URL 내 파라미터를 처리하는 라우트들을 갖고 있어서, 만일 이 라우트들(예를 들어 /users/:id)가 먼저 온다면 Express.js는 /users/login으로의 요청을 사용자의 show 페이지로 처리할 것이며 여기에서 로그인은 :id로 인식된다. 순서는 중요하다. /users/login 라우트가 먼저 온다면 Express.js는 파라미터를 처리하는 라우트를 체킹하기 전에 그 라우트를 매칭할 것이다.



퀵 체크 23.2

참 또는 거짓! bcrypt의 compare 메소드는 데이터베이스의 평문 패스워드와 사용자 input에서의 평문을 비교한다.

거짓이다! 데이터베이스에 저장된 매스윈드의 값은 해싱된 것밖에 없다. 따라서 비교할 평문 값은 존재하지 않는다. 비교 작업은 사용자 입력 값의 해싱을 통해 시작되며 새로 데이터베이스에 저장된 해싱된 값과 이를 비교한다. 이렇게 때문에 애플리케이션은 사용자의 실제 패스워드를 알 수는 없지만 해싱된 값들이 일치하면 매스워드가 안전하게 일치한다고 볼 수 있다.



퀵 체크 23.3

밸리데이터 (validator)와 새니타이저(sanitizer)의 차이점은 무엇인가?

밸리데이터는 데이터베이스의 요구에 데이터가 부합하는지 여부 판단을 위해 데이터 품질을 체크한다. 새니타이저는 스페이스를 트리밍이나 대소문자의 변환, 필요 없는 캐릭터의 삭제를 통해 데이터를 정리한다.

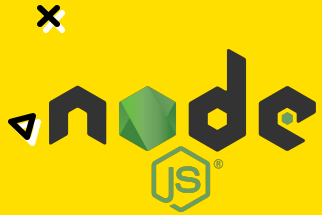
24

사용자 인증 추가

passport 패키지를사용해 앱에서 사용자 인증
모델에 passport-local-mongoose 플러그인 적용
사용자 로그인 전에 인증 액션 추가

p. 349-360

24 사용자 인증 추가



23장에서 사용자 데이터 보안의 중요성과 패스워드 해싱 사용법을 배웠다.
24장에서는 해성 프로세스를 **조금 더 쉽게** 해주는 도구를 알아본다.

먼저 해성 메소드를 **passport-local-mongoose 패키지**를 사용하도록 수정한다. 이 패키지는 passport와 mongoose를 같이 사용해 보이지 않는 곳에서 해싱을 수행한다. 다음으로 Passport.js가 어떻게 사용자 계정을 애플리케이션상에서 인증하는지 알아볼 것이다. 이 프로세스는 **세션 쿠키를 포함하며 플래시 메시지가 사용하는 방식과 비슷하다.**

24.1 Passport.js의 실행

Passport.js는 Node.js에서 사용되는 미들웨어이며 새로운 사용자의 패스워드 해싱과 애플리케이션상에서의 인증 활동을 수행한다. Passport.js는 사용자 계정에서 생성과 로그인에 다른 메소드를 사용하며 이는 사용자 이름과 패스워드로 하는 기본 로그인부터 페이스북과 같은 제삼자 로그인 서비스를 활용하는 것까지 가능하다. 이들 로그인 메소드를 **스트래티지 Strategy**라고 부르며 우리 애플리케이션에서 사용하는 스트래티지는 외부 서비스를 이용하지 않기 때문에 **로컬 스트래티지**다.

이 스트래티지는 사용자 로그인 상태와 관련된 데이터와 패스워드의 비교와 **해싱 관리**를 통해 유입 데이터가 인증됐는지 체크한다. **local 스트래티지는 사용자명과 패스워드를 시용하는 로그인 메소드를 참조한다.**

```
npm install passport passport-local-mongoose
```

Listing 24.1 main.js에서 passport의 요청과 초기화

```
const passport = require("passport");          ← passport 모듈 요청
router.use(passport.initialize());             ← passport 초기화
router.use(passport.session());
                                               ← passport가 Express.js 내 세션을 사용하도록 설정
```

[노트] passport.session은 passport가 사전에 Express.js에서 사용된 세션들을 사용하게 한다. 세션은 Listing 24.2 **전에** 정의돼야 한다.

24.1 Passport.js의 실행

그다음 두 개 라인은 passport가 사용자 모델을 통해 사용자를 **직렬화**serialize 또는 **역직렬화**deserialize하도록 한다. 이 라인들은 프로세스가 세션에 저장된 시용자 데이터를 암호화 또는 복호화하도록 지시한다.

Listing 24.2 main.js에서 passport 직렬화 설정

```
const User = require("./models/user");
passport.use(User.createStrategy());
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

User 모델의 요청

사용자의 로그인 스트래티지 설정

passport가 사용자 데이터의 직렬화 및 역직렬화 작업을 하도록 설정

패스포트는 사용자 데이터를 직렬화하거나 역직렬화해 세션에 전달한다. 세션은 압축된 형태의 직렬화된 사용자 데이터를 저장하고 클라이언트로부터 마지막에 로그인한 사용자를 확인하기 위해 이 데이터는 다시 서버로 이동한다. 역직렬화는 사용자 데이터를 압축 버전으로부터 복구하며 이 데이터를 통해 사용자 정보를 확인할 수 있다.

24.1 Passport.js의 실행

패스포트는 사용자 데이터를 직렬화하거나 역직렬화해 세션에 전달한다. 세션은 압축된 형태의 직렬화된 사용자 데이터를 저장하고 클라이언트로부터 마지막에 로그인한 사용자를 확인하기 위해 이 데이터는 다시 서버로 이동한다. 역직렬화는 사용자 데이터를 압축 버전으로부터 복구하며 이 데이터를 통해 사용자 정보를 확인할 수 있다.

데이터의 직렬화

애플리케이션 내 데이터를 갖고 작업할 때, 액세스 및 수정이 쉬운 구조가 편하다. 예를 들어 사용자 객체의 경우 이메일 같은 정보를 찾아주며 사용자 모델의 가상 속성인 fullName도 사용하게 해야 한다. 이런 모델이 애플리케이션에서는 특히 유용하지만 이 사용자 객체를 메소드와 Mongoose ODM과 함께 보내기는 쉽지 않다. 따라서 사용자 데이터를 직렬화시킬 필요가 있는 것이다.

직렬화(Serialization)는 압축된 가독 형식으로 데이터 구조를 바꾸는 프로세스다. 이 데이터는 JSON, YAML, XML과 같은 다양한 포맷을 취할 수 있다. 사용자 데이터는 단일 데이터(때로는 문자열)로 변환돼 HTTP 트랜잭션 내에서 전달될 수 있다

Passport.js는 이 직렬화 프로세스를 수행하고 사용자 데이터를 암호화해 클라이언트 브라우저에 이 데이터가 세션 쿠키 중 일부로서 저장시킨다. 이 쿠키는 사용자 정보를 포함하고 있기 때문에 다음번에 요청이 발생할 때, 이 사용자가 이전에 로그인했다는 것을 (애플리케이션에서 결정한 사용자 상태의 유효성 검증 방식으로) 애플리케이션 서버에 알려준다.

동일한 사용자가 애플리케이션에서 다른 요청을 보냈다면 Passport.js는 사용자를 원래 모델 객체 형식으로 복원하기 위해 역직렬화를 시도한다. 작업이 완료되고 사용자가 유효한지 확인되면 이전과 같이 사용자 객체를 사용해 모델 메소드를 적용하고 Mongoose 쿼리를 사용할 수 있다.

24.1 Passport.js의 실행

User.js 첫 부분에
`const passportLocalMongoose = require("passport-local-mongoose");`
를 추가한다.

이곳에서 보는 것과 같이 사용자 스키마에 Passport.js 플러그인 추가가 일어난다. Mongoose 플러그인 메소드를 사용해 userSchema가 passportLocalMongoose를 패스워드의 해성과 저장에 사용한다. 또한 passportLocalMongoose가 이메일 필드를 사용자 이름 대신 로그인 파라미터로 사용하게 한다. 사용자 이름이 이 모듈에서 기본 필드이기 때문이다.

[노트] 이 라인은 사용자 모델의 등록 전에 위치해야 한다.

Listing 24.3 passport-local-mongoose 플러그인을 사용자 스키마에 추가

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

passport-local-mongoose
플러그인을 사용자 스키마에 추가

이 코드가 추가되면 Passport.js는 자동적으로 패스워드 저장에 관여를 하며 **userSchema에서 패스워드 속성은 없애버릴 수 있다.** 이 플러그인은 스키마를 수정해 hash와 salt 필드를 사용자 모델이 일반 password 필드 대신 추가시킨다.

24.1 Passport.js의 실행

해시와 솔트

24장에서 해싱은 배웠으나 bcrypt가 알고리즘을 통해 수행하는 해싱 프로세스 수행까지 이해할 필요는 없었다. 정확히 어떻게 bcrypt와 Passport.js가 사용자 패스워드를 해싱하는 것일까?

현대 해싱 기술은 사용자 패스워드를 받아 이를 원래로 복원할 수 없는 해시(hash)로 변환한다. 이 해시는 무작위의 숫자와 문자로 돼 있어 평문 데이터보다 더 안전하게 데이터베이스에 저장할 수 있다. 누군가가 데이터베이스를 해킹해도 이 해싱된 데이터만 가져갈 수 있다. 해커가 이 데이터로 할 수 있는 일은 그가 예상한 패스워드를 해싱해 그가 빼낸 해시 데이터와 일치하는지 비교하는 것밖에 없다. 이는 상당히 지겹고 오래 걸리는 작업이지만, 언젠가는 오리지널 패스워드를 알아낼 가능성이 있다는 것은 부정할 수 없다. 솔트는 이런 취약점과 싸우기 위해 도입됐다.

솔트(salt)는 짧은 랜덤 문자열이며 평문 패스워드가 해싱되기 전에 패스워드 뒤에 덧붙여진다. 이렇게 하면 해커가 독자의 패스워드를 정확히 추측했다고 해도 오리지널 패스워드에 추가된 솔트 값도 역시 같이 추측을 해야 하는 수고가 발생한다. 해킹은 더 어려워진다.

Passport.js는 해싱된 패스워드와 솔트 값을 데이터베이스에 저장해 애플리케이션 내에서 해싱을 항상 수행할 수 있도록 한다. Passport.js를 사용해 첫 번째 사용자를 등록할 때 몽고DB에서 어떻게 저장되는지 다음 단계를 따라 살펴보자.

- 터미널을 띄워 mongo 실행
- 레시피 데이터베이스를 읽어들이기 위한 use recipe_db 수행
- 모든 사용자의 패스워드를 열람하기 위한 db.users.find({}, {password: 1}) 수행
- 해싱된 패스워드와 해싱되지 않는 패스워드의 비교

[노트] 애플리케이션에서 password 속성으로의 참조가 제거됐는지 확인하라 passport-local-mongoose가 새로운 패스워드 필드를 사용자 모델에 추가하기 때문에 이들 속성은 더 이상 사용하지 않는다.

24.2 패스포트 등록 사용을 위한 **Create** 액션 수정

다음 단계는 **create 액션의 수정**이며 사용자 계정 생성 전에 수행했던 bcrypt 해싱 함수 대신에 Passport.js를 사용할 것이다. Passport.js 모듈과 통합하면 계정 등록 과정을 간소화하기 위한 메소드 라이브러리에 액세스할 수 있다.

Listing 24.4 main.js 내 create 액션에서의 새로운 사용자 등록

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

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

  User.register(newUser, req.body.password, (error, 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:
➥ ${error.message}.`);
      res.locals.redirect = "/users/new";
      next();
    }
  });
}
```

← 새로운 사용자 등록

사용자 등록이 성공할 경우를 위한 리디렉션 설정

실패 시 리디렉션 설정 및 에러를 플래시 메시지로 로깅

[노트] 사용자 모델에서 bcrypt를 위한 pre(“ save") 혹과 userSchema.methods.password Comparison을 각주 처리나 삭제를 해야 한다. 이 흑들을 삭제하지 않으면 bcrypt는 passport가 할 수 있는 사용자 패스워드의 해싱을 수행하기 때문에 처리되지 않은 프라미스로 에러가 발생한다.

[팁] (선택) seed.js 파일을 업데이트해 사용자 계정을 Mongoose의 create 메소드 대신 passport로 생성하자. 이 작업은 개발 과정에서 애플리케이션의 사이즈가 커질 때 쉽게 데이터를 다시 채울 수 있게한다.

24.3 로그인 시 사용자 인증

또한 사용자 컨트롤러에 **`const passport = require("passport");`**를 파일 최상단에 추가해 passport를 요청하도록 한다.

usersController.authenticate를 호출하면 passport.authenticate를 호출하는 것이다.
이 함수에서 passport는 유입 요청 데이터와 사용자에 대한 정보를 데이터베이스의 레코드와 비교를 시도한다. 만일 사용자 계정이 존재하고 입력된 패스워드가 해싱된 패스워드와 일치하면 이 액션으로부터 리디렉션된다.

Listing 24.5 userController.js에서 passport 인증 미들웨어 추가

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

local strategy로 사용자를 인증하기 위해 passport 호출

성공, 실패의 플래시 메시지를 설정하고 사용자의 인증 상태에 따라 리디렉션할 경로를 지정한다.

다음에 이어지는 **usersController.redirectView는 로그인 라우트에서 더 이상 쓰이지 않는다.**
23장에서 설정된 라우트 router.post("/users/login", usersController.authenticate) 를 통해 애플리케이션은 기존 사용자를 인증할 준비가 됐다.

24.3 로그인 시 사용자 인증

로그인 사용자가 로그인돼 있다는 것을 보여주는 표시와 로그아웃하는 방법이 있다면 좋을 것이다.

Listing 24.6 layout.ejs에서 내비게이션 바에 로그인 상태 추가

```
<% if (loggedIn) { %>
  Logged in as  <a href="<%=`/users/${currentUser._id}`%>">
  <%= currentUser.fullName %></a>
<%} else {%>
  <a href="/users/login">Log In</a>
<% } %>
```

사용자의 로그인 상태 체크

로그인을 할 수 있는 링크 표시

애플리케이션을 다시 시작해도 아직 내비게이션 바에 어떤 변화도 보이지 않을 것이다. loggedIn 및 currentUser 변수를 생성해 이들이 각 뷰 페이지에서 보이도록 해야 한다.

24.3 로그인 시 사용자 인증

애플리케이션을 다시 시작해도 아직 내비게이션 바에 어떤 변화도 보이지 않을 것이다. loggedIn 및 currentUser 변수를 생성해 이들이 각 뷰 페이지에서 보이도록 해야 한다.

flashMessages를 로컬 객체로 설정하기 위한 미들웨어를 이미 만들었으므로 main.js에서 미들웨어 함수 내에 아래의 코드를 추가할 수 있다. isAuthenticated는 Passport.j s에서 제공되며 이를 통해 유입된 요청상에서 존재하는 사용자가 요청 쿠키에 저장돼 있는지 확인할 수 있다.

Listing 24.7 사용자 정의 미들웨어로 로컬 변수 추가

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

passport login status를 나타내기 위한 loggedIn 변수의 설정

로그인된 사용자를 나타내기 위한 currentUserd의 설정

24.3 로그인 시 사용자 인증

마지막으로 **Log out**를 currentUser의 이름을 나타내는 부분 아래에 추가한다. 그다음 **router.get("/users/logout", usersController.logout, usersController.redirectView)**를 main.js 에 로그인 라우트가 위치한 곳 다음으로 추가한다.

이 액션은 요청에서 Passport.js가 제공하는 logout 메소드를 사용하며 이를 통해 사용자 세션을 지워버린다. 사용자 정의 미들웨어를 통한 다음 전달을 통해 isAuthenticated는 false를 돌려주고 현재 로그인된 사용자는 더 이상 존재하지 않게 된다. 이 동작은 플래시 메시지를 통해 사용자는 로그아웃 됐다는 것을 나타내며 redirectView 액션을 통해 홈 페이지로 리디렉션하게 된다.

Listing 24.8 userController.js에서 로그아웃 액션의 추가

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

24.3 로그인 시 사용자 인증

그림 24.1 로그인 성공 페이지

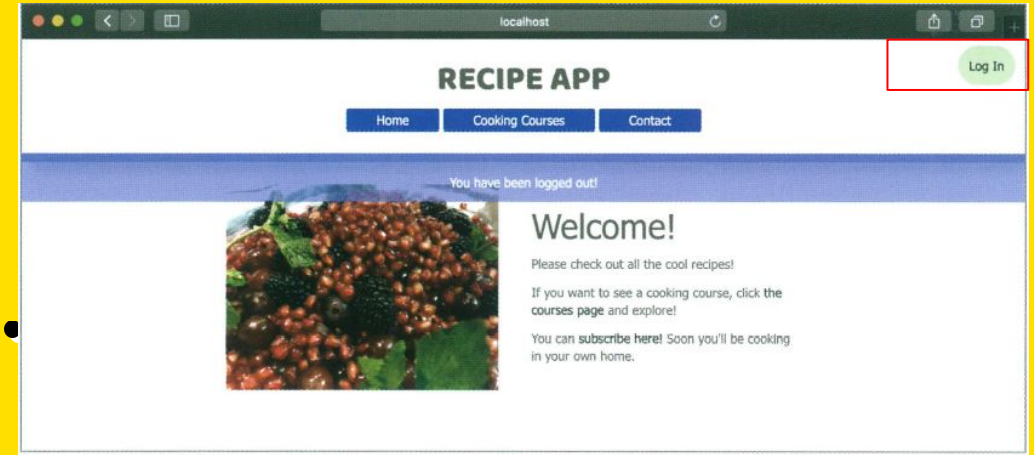
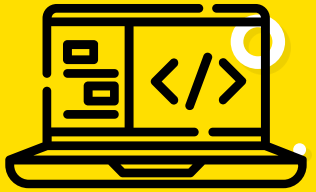
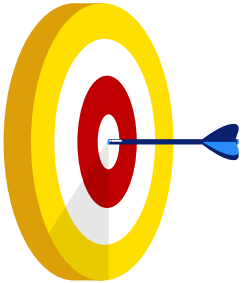


그림 24.2 로그아웃 성공 페이지



Coding!

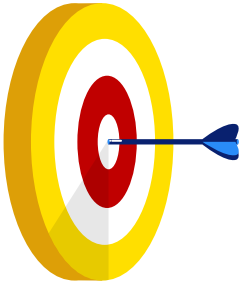
Listing 24.1~
p. 349-360



퀵 체크 24.1

참 또는 거짓! 솔트는 패스원드 해싱을 위해 필요하다.

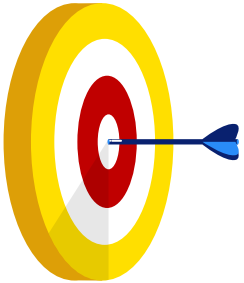
거짓이다! 솔트는 패스워드 해싱을 좀 더 강력하게 하기 위해 평문 패스원드와 섞는 랜덤 텍스트이며 반드시 필요한 것은 아니다.



퀵 체크 24.2

왜 Passport.js는 hash와 salt를
데이터베이스에 저장하게 할까?

Passport.js가 salt와 hash를 저장함으로써 각 사용자들은
자신들의 고유한 해싱 팩터를 가질 수 있다. 모든 사용자들이
동일한 salt 값을 가질 수도 있지만 보안적인 면에서는 덜 안전하다.



퀵 체크 24.3

애플리케이션 내에서는 어떻게 Passport.js 메소드에 액세스를 할까?

Express.js 내의 미들웨어로서 passport 모듈을 추가했기 때문에 Passport.js에서 제공하는 라이브러리에 액세스할 수 있다. 이 메소드들은 애플리케이션에 진입 시 요청으로 유입된다. 이 요청들이 미들웨어 체인을 통해 전달되면 어디에서나 이 passport 메소드를 호출할 수 있다.

과제 타임!

한번 해 보자!~