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

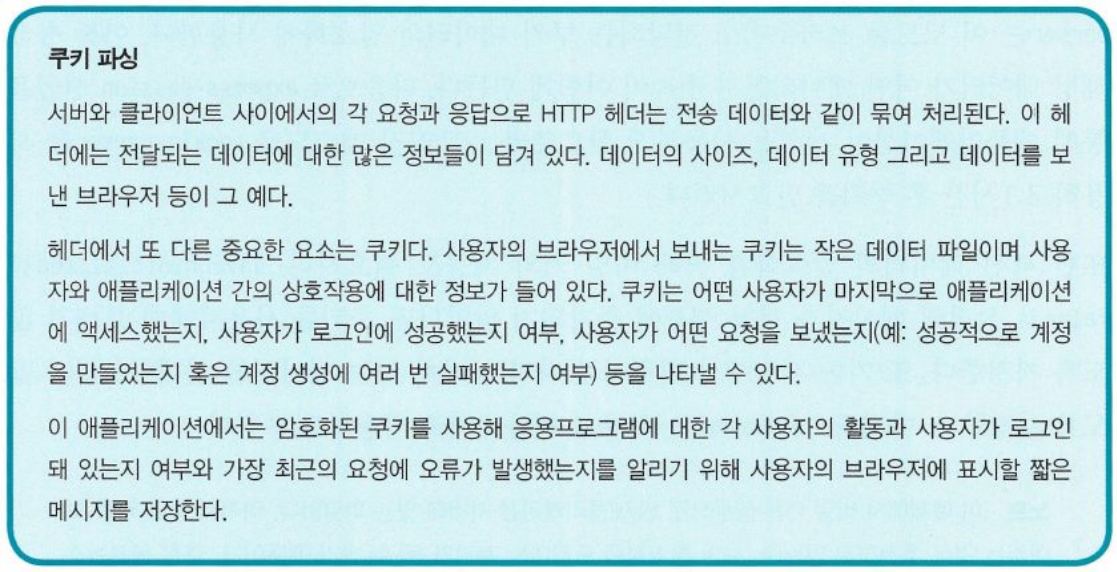
[노트] 이 예제에서 비밀 키는 평문으로 된 애플리케이션 서버에 있는 파일이다.

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.

```
10  /**
11   * Listing 22.3 (p. 328)
12   * userController.js에서 create액션이 플래시 메시지를 추가
13   *
14   * [노트] getUserParams는 이전 캡스톤 프로젝트 (21장)에서 사용돼 왔다.
15   * 이 함수는 컨트롤러를 통해 재사용돼 사용자 속성을 하나의 객체로 구성한다.
16   * 동일한 함수를 다른 모델 컨트롤러에도 구성해야 한다.
17   */
18  const getUserParams = (body) => {
19    return {
20      name: {
21        first: body.first,
22        last: body.last,
23      },
24      email: body.email,
25      username: body.username,
26      password: body.password,
27      profileImg: body.profileImg,
28    };
29  };
```

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

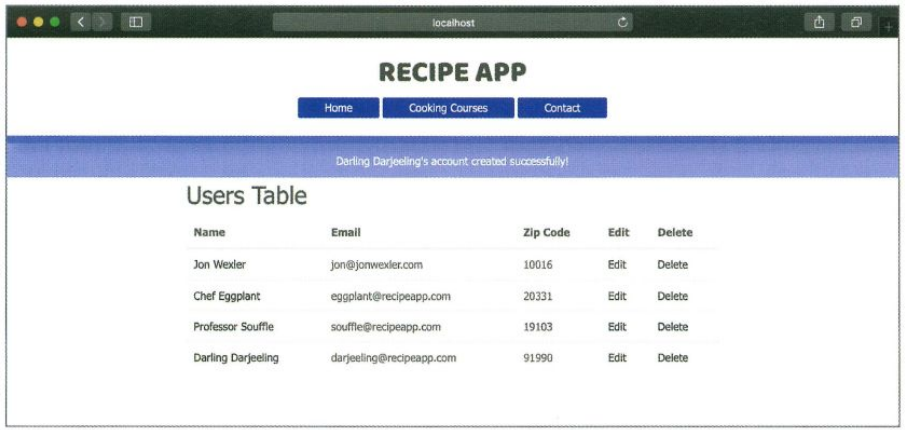


그림 22.1 /users 페이지에서 성공 플래시 메시지

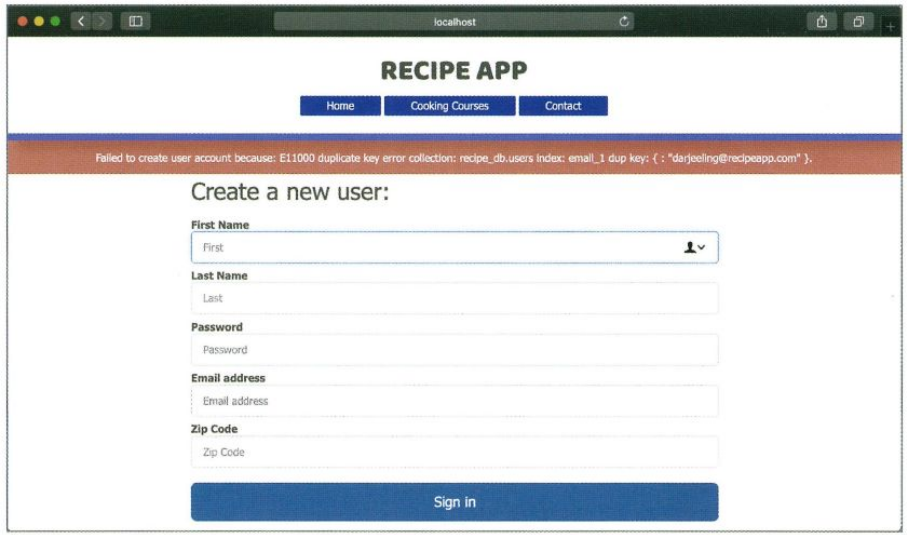
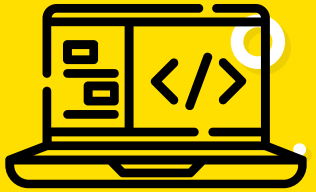


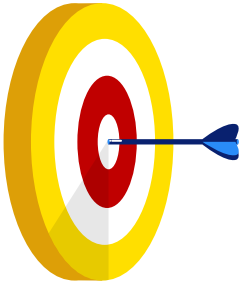
그림 22.2 홈페이지상 보이는 에러 플래시 메시지



Coding!

Listing 22.1~

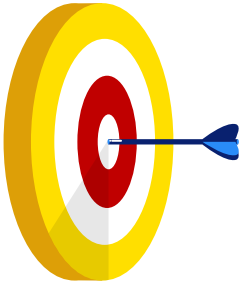
p. 323-332



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

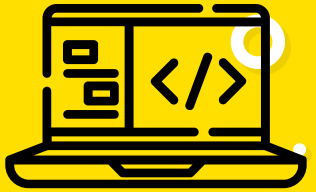
```
validate: (req, res, next) => {          ← validate 함수 추가
  req.sanitizeBody("email").normalizeEmail({
    all_lowercase: true                  ← trim()으로 whitespace 제거
    }).trim();
  req.check("email", "Email is invalid").isEmail();
  req.check("zipCode", "Zip code is invalid")
.notEmpty().isInt().isLength({
    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.

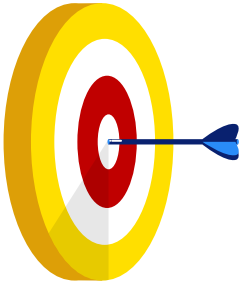
- **밸리데이터 Validator:** 유입 데이터가 일정 기준을 만족하는지 체크
- **새니타이저 Sanitizer:** 데이터베이스로 저장하기 전에 불필요한 데이터의 삭제나 데이터 타입 캐스팅 수행

[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!

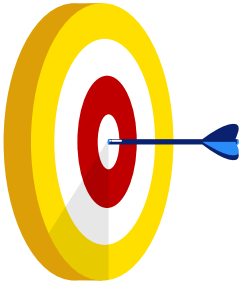
Listing 23.1~
p. 333-348



퀵 체크 23.1

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

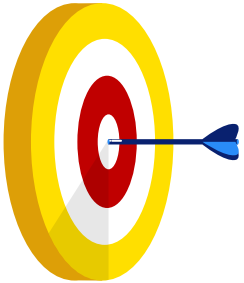
URL 내 파라미터를 처리하는 라우트들을 갖고 있어서, 만일 이 라우트들(예를 들어 /users/:id)가 먼저 온다면 Express.js는 /users/login으로의 요청을 사용자의 show 페이지로 처리할 것이며 여기에서 로그인은 :id로 인식된다. 순서는 중요하다. /users/login 라우트가 먼저 온다면 Express.js는 파라미터를 처리하는 라우트를 체킹하기 전에 그 라우트를 매칭할 것이다.



퀵 체크 18.2

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.

과제 타임!

한번 해 보자!~