

9.1

프로젝트 구조 갖추기

SNS 중에는 140자의 단문 메시지를 보내고 사람들이 메시지의 내용을 공유할 수 있는 서비스가 있습니다. 이와 유사한 서비스를 노드로 만들어보겠습니다. 프런트엔드 쪽 코드가 많이 들어가지만, 노드와 익스프레스 코드 위주로 보면 됩니다.

먼저 nodebird라는 폴더를 만듭니다. 항상 package.json을 제일 먼저 생성해야 합니다. package.json을 생성해주는 npm init 명령어를 콘솔에서 호출해도 되고 직접 만들어도 됩니다. version이나 description, author, license는 원하는 대로 자유롭게 수정해도 괜찮습니다. scripts 부분에 start 속성은 잊지 말고 넣어줘야 합니다.

package.json

```
{
  "name": "nodebird",
  "version": "0.0.1",
  "description": "익스프레스로 만드는 SNS 서비스",
  "main": "app.js",
  "scripts": {
    "start": "nodemon app"
  },
  "author": "ZeroCho",
  "license": "MIT"
}
```

nodebird 폴더 안에 package.json을 생성했다면 이제 시퀄라이즈를 설치합니다. 이 프로젝트에서는 NoSQL 대신 SQL(MySQL)을 데이터베이스로 사용할 것입니다. 사용자와 게시물 간, 게시물과 해시태그 간의 관계가 중요하므로 관계형 데이터베이스인 MySQL을 선택했습니다.

콘솔

```
$ npm i sequelize mysql2 sequelize-cli
$ npx sequelize init
```

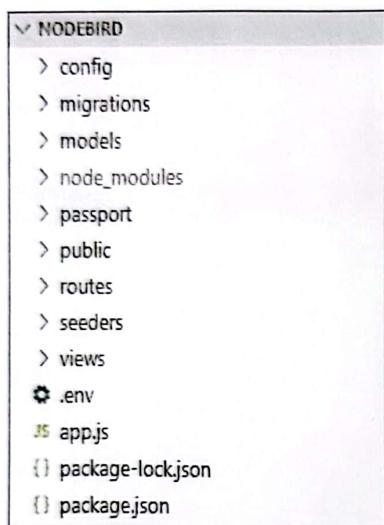
npm i sequelize mysql2 sequelize-cli 명령어를 호출하면 node_modules 폴더와 package-lock.json이 생성됩니다. 또한, npx sequelize init 명령어를 호출하면 config, migrations,

models, seeders 폴더가 생성됩니다. npx 명령어를 사용하는 이유는 전역 설치(npm i -g)를 피하기 위해서입니다.

이제 다른 폴더도 생성합니다. 템플릿 파일을 넣을 views 폴더, 라우터를 넣을 routes 폴더, 정적 파일을 넣을 public 폴더가 필요합니다. 9.3절에서 설명할 passport 패키지를 위한 passport 폴더도 만듭니다.

마지막으로, 익스프레스 서버 코드가 담길 app.js와 설정값들을 담을 .env 파일을 nodebird 폴더 안에 생성합니다. 폴더 구조는 다음과 같습니다.

▼ 그림 9-2 nodebird 폴더 구조



이와 같은 구조라면 폴더 구조가 올바르게 설정된 것입니다. 여기에 몇 개의 폴더가 추가되기는 하지만 이 구조를 크게 벗어나지 않을 것입니다.

이 구조는 고정된 구조가 아니므로 편의에 따라 바꿔도 됩니다. 서비스가 성장하고 규모가 커질수록 폴더 구조도 복잡해지므로 각자 서비스에 맞는 구조를 적용해야 합니다.

먼저 필요한 npm 패키지들을 설치하고 app.js를 작성합니다. 템플릿 엔진은 넌저스를 사용할 것입니다.

콘솔

```
$ npm i express cookie-parser express-session morgan multer dotenv nunjucks
$ npm i -D nodemon
```

모두 6장에서 설명한 패키지들입니다. app.js와 .env는 다음과 같이 작성합니다.

app.js

```
const express = require('express');
const cookieParser = require('cookie-parser');
const morgan = require('morgan');
const path = require('path');
const session = require('express-session');
const nunjucks = require('nunjucks');
const dotenv = require('dotenv');

dotenv.config();
const pageRouter = require('./routes/page');

const app = express();
app.set('port', process.env.PORT || 8001);
app.set('view engine', 'html');
nunjucks.configure('views', {
  express: app,
  watch: true,
});

app.use(morgan('dev'));
app.use(express.static(path.join(__dirname, 'public')));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser(process.env.COOKIE_SECRET));
app.use(session({
  resave: false,
  saveUninitialized: false,
  secret: process.env.COOKIE_SECRET,
  cookie: {
    httpOnly: true,
    secure: false,
  },
}));
app.use('/', pageRouter);

app.use((req, res, next) => {
  const error = new Error(`${req.method} ${req.url} 라우터가 없습니다.`);
  error.status = 404;
  next(error);
});
```

```
app.use((err, req, res, next) => {
  res.locals.message = err.message;
  res.locals.error = process.env.NODE_ENV !== 'production' ? err : {};
  res.status(err.status || 500);
  res.render('error');
});

app.listen(app.get('port'), () => {
  console.log(app.get('port'), '번 포트에서 대기 중');
});
```

라우터로는 현재 pageRouter만 있지만, 추후에 더 추가할 예정입니다. 라우터 이후에는 404 응답 미들웨어와 에러 처리 미들웨어가 있습니다. 마지막으로, 앱을 8001번 포트에 연결했습니다.

.env

```
COOKIE_SECRET=cookiesecret
```

하드 코딩된 비밀번호가 유일하게 남아 있는 파일이 있습니다. 시퀄라이즈 설정을 담아둔 config.json입니다. JSON 파일이라 process.env를 사용할 수 없습니다. 시퀄라이즈의 비밀번호를 숨기는 방법은 15.1.2절에서 알아봅니다.

기본적인 라우터와 템플릿 엔진도 만들어봅시다. routes 폴더 안에는 page.js를, views 폴더 안에는 layout.html, main.html, profile.html, join.html, error.html을 생성합니다. 약간의 디자인을 위해 main.css를 public 폴더 안에 생성합니다.

routes/page.js

```
const express = require('express');
const { renderProfile, renderJoin, renderMain } = require('../controllers/page');

const router = express.Router();

router.use((req, res, next) => {
  res.locals.user = null;
  res.locals.followerCount = 0;
  res.locals.followingCount = 0;
  res.locals.followingIdList = [];
  next();
});

router.get('/profile', renderProfile);
```

```
router.get('/join', renderJoin);

router.get('/', renderMain);

module.exports = router;
```

GET /profile, GET /join, GET /까지 페이지 세 개로 구성되어 있습니다. router.use로 라우터용 미들웨어를 만들어 템플릿 엔진에서 사용할 user, followingCount, followerCount, followingIdList 변수를 res.locals로 설정했습니다. 지금은 각각 null, 0, 0, []이지만 나중에 값을 넣을 것입니다. res.locals로 값을 설정하는 이유는 user와 followingCount, followerCount, followingIdList 변수는 모든 템플릿 엔진에서 공통으로 사용하기 때문입니다.

여기서 특이한 점은 컨트롤러의 존재입니다. 이전과는 다르게 라우터의 미들웨어를 다른 곳에서 불러오고 있습니다. renderProfile, renderJoin, renderMain과 같이 라우터 마지막에 위치해 클라이언트에 응답을 보내는 미들웨어를 컨트롤러라고 합니다. 우리는 아직 컨트롤러를 작성하지 않았습니다. 프로젝트에 controllers 폴더를 만들고 그 안에 page.js를 만듭니다.

9

위스프레스로 SNS 서비스 만들기

controllers/page.js

```
exports.renderProfile = (req, res) => {
  res.render('profile', { title: '내 정보 - NodeBird' });
};

exports.renderJoin = (req, res) => {
  res.render('join', { title: '회원 가입 - NodeBird' });
};

exports.renderMain = (req, res, next) => {
  const twits = [];
  res.render('main', {
    title: 'NodeBird',
    twits,
  });
};
```

컨트롤러라고 해서 특별한 것은 아니고 res.send, res.json, res.redirect, res.render 등이 존재하는 미들웨어일 뿐입니다. 다만 컨트롤러를 분리하면 좋은 점이 있는데, 11장에서 테스트를 진행할 때 분리한 이유를 알게 됩니다. 지금은 실무에서 코드를 편하게 관리하기 위해 컨트롤러를 따로 분리한다고 알아두면 됩니다.

`renderProfile`과 `renderJoin`은 각각 내 정보 페이지와 회원가입 페이지를 화면에 렌더링합니다. `renderMain` 컨트롤러는 메인 페이지를 렌더링하면서 네이티브에 `twits`(게시글 목록)를 전달합니다. `twits`는 지금은 빈 배열이지만 나중에 값을 넣습니다.

Note ≡ 컨트롤러와 서비스

컨트롤러에서 비즈니스 로직을 서비스(service)라는 개념으로 한 번 더 따로 분리하는 경우도 많습니다. 서비스는 해당 컨트롤러의 핵심 비즈니스 로직을 담당하면서 요청(req)이나 응답(res)에 대해 모른다고 보면 됩니다. 요청이나 응답을 몰라야 하는 이유는 서버가 항상 HTTP 요청만 받는 것은 아니기 때문입니다. 서버는 12장에서 배운 웹 소켓 요청을 받을 수도 있고, RPC라는 HTTP와는 다른 프로토콜의 요청을 받을 수도 있습니다. 어떠한 요청이 모든 동일한 비즈니스 로직을 수행해야 하는 것이 서비스의 역할입니다. 11장에서는 컨트롤러에서 서비스를 분리하는 방법을 알아봅니다. 12장에서는 서비스의 필요성을 실제 예제로 배워봅니다.

그다음은 클라이언트 코드입니다. css나 html 파일들은 그리 중요하지 않으니 <https://github.com/zerocho/nodejs-book>에서 직접 코드를 복사하는 것을 권장합니다.

views/layout.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{title}}</title>
    <meta name="viewport" content="width=device-width, user-scalable=no">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <link rel="stylesheet" href="/main.css">
  </head>
  <body>
    <div class="container">
      <div class="profile-wrap">
        <div class="profile">
          {% if user and user.id %}
            <div class="user-name">{'안녕하세요! ' + user.nick + '님'}</div>
            <div class="half">
              <div>팔로잉</div>
              <div class="count following-count">{{followingCount}}</div>
            </div>
            <div class="half">
              <div>팔로워</div>
              <div class="count follower-count">{{followerCount}}</div>
            </div>
          {% endif %}
        </div>
      </div>
    </div>
  </body>
</html>
```

```

<input id="my-id" type="hidden" value="{{user.id}}">
<a id="my-profile" href="/profile" class="btn">내 프로필</a>
<a id="logout" href="/auth/logout" class="btn">로그아웃</a>
{% else %}
<form id="login-form" action="/auth/login" method="post">
<div class="input-group">
<label for="email">이메일</label>
<input id="email" type="email" name="email" required autofocus>
</div>
<div class="input-group">
<label for="password">비밀번호</label>
<input id="password" type="password" name="password" required>
</div>
<a id="join" href="/join" class="btn">회원 가입</a>
<button id="login" type="submit" class="btn">로그인</button>
<a id="kakao" href="/auth/kakao" class="btn">카카오톡</a>
</form>
{% endif %}
</div>
<footer>
Made by ;
<a href="https://www.zerocho.com" target="_blank">ZeroCho</a>
</footer>
</div>
{% block content %}
{% endblock %}
</div>
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
window.onload = () => {
if (new URL(location.href).searchParams.get('error')) {
alert(new URL(location.href).searchParams.get('error'));
}
};
</script>
{% block script %}
{% endblock %}
</body>
</html>

```

layout.html에서는 if문을 중점적으로 보면 됩니다. 렌더링할 때 user가 존재하면 사용자 정보와 팔로잉, 팔로워 수를 보여주고, 존재하지 않으면 로그인 메뉴를 보여줍니다.

```

{% extends 'layout.html' %}

{% block content %}
    <div class="timeline">
        {% if user %}
            <div>
                <form id="twit-form" action="/post" method="post" enctype=
                    multipart/form-data>
                    <div class="input-group">
                        <textarea id="twit" name="content" maxlength="140"></textarea>
                    </div>
                    <div class="img-preview">
                        <img id="img-preview" src="" style="display: none;" width="250"
                            alt="미리 보기">
                        <input id="img-url" type="hidden" name="url">
                    </div>
                    <div>
                        <label id="img-label" for="img">사진 업로드</label>
                        <input id="img" type="file" accept="image/*">
                        <button id="twit-btn" type="submit" class="btn">짹짹</button>
                    </div>
                </form>
            </div>
        {% endif %}
        <div class="twits">
            <form id="hashtag-form" action="/hashtag">
                <input type="text" name="hashtag" placeholder="태그 검색">
                <button class="btn">검색</button>
            </form>
            {% for twit in twits %}
                <div class="twit">
                    <input type="hidden" value="{{twit.User.id}}" class="twit-user-id">
                    <input type="hidden" value="{{twit.id}}" class="twit-id">
                    <div class="twit-author">{{twit.User.nick}}</div>
                    {% if not followingIdList.includes(twit.User.id) and twit.User.id !=
                        user.id %}
                        <button class="twit-follow">팔로우하기</button>
                    {% endif %}
                    <div class="twit-content">{{twit.content}}</div>
                    {% if twit.img %}
                        <div class="twit-img"></div>
                    
```

```

        {% endif %}
    </div>
    {% endfor %}
</div>
</div>
{% endblock %}

{% block script %}
<script>
if (document.getElementById('img')) {
    document.getElementById('img').addEventListener('change', function(e) {
        const formData = new FormData();
        console.log(this, this.files);
        formData.append('img', this.files[0]);
        axios.post('/post/img', formData)
            .then((res) => {
                document.getElementById('img-url').value = res.data.url;
                document.getElementById('img-preview').src = res.data.url;
                document.getElementById('img-preview').style.display = 'inline';
            })
            .catch((err) => {
                console.error(err);
            });
    });
}
document.querySelectorAll('.twit-follow').forEach(function(tag) {
    tag.addEventListener('click', function() {
        const myId = document.querySelector('#my-id');
        if (myId) {
            const userId = tag.parentNode.querySelector('.twit-user-id').value;
            if (userId !== myId.value) {
                if (confirm('팔로잉하시겠습니까?')) {
                    axios.post(`/user/${userId}/follow`)
                        .then(() => {
                            location.reload();
                        })
                        .catch((err) => {
                            console.error(err);
                        });
                }
            }
        }
    });
});

```

```
});  
</script>  
{% endblock %}
```

main.html에서는 user 변수가 존재할 때 게시글 업로드 폼을 보여줍니다. for문도 추가되었으며, 렌더링 시 twits 배열 안의 요소들을 읽어서 게시글로 만듭니다. 지금은 빈 배열이지만 나중에 twits에 게시글 데이터를 넣으면 됩니다.

if not followingIdList.includes(twit.User.id) and twit.User.id !== user.id는 나의 팔로잉 아이디 목록에 게시글 작성자의 아이디가 없으면 팔로우 버튼을 보여주기 위한 구문입니다. 또한, 게시글 작성자가 나인 경우 나를 팔로우할 수는 없게 했습니다. if not과 and를 써서 여러 가지 조건을 조합했으며, 넘적스 문법입니다.

views/profile.html

```
{% extends 'layout.html' %}  
  
{% block content %}  
  <div class="timeline">  
    <div class="followings half">  
      <h2>팔로잉 목록</h2>  
      {% if user.Followings %}  
        {% for following in user.Followings %}  
          <div>{{following.nick}}</div>  
        {% endfor %}  
      {% endif %}  
    </div>  
    <div class="followers half">  
      <h2>팔로워 목록</h2>  
      {% if user.Followers %}  
        {% for follower in user.Followers %}  
          <div>{{follower.nick}}</div>  
        {% endfor %}  
      {% endif %}  
    </div>  
  </div>  
{% endblock %}
```

profile.html은 사용자의 팔로워와 사용자가 팔로잉 중인 목록을 보여줍니다.

views/join.html

```
{% extends 'layout.html' %}

{% block content %}
<div class="timeline">
<form id="join-form" action="/auth/join" method="post">
<div class="input-group">
<label for="join-email">이메일</label>
<input id="join-email" type="email" name="email"></div>
<div class="input-group">
<label for="join-nick">닉네임</label>
<input id="join-nick" type="text" name="nick"></div>
<div class="input-group">
<label for="join-password">비밀번호</label>
<input id="join-password" type="password" name="password">
</div>
<button id="join-btn" type="submit" class="btn">회원 가입</button>
</form>
</div>
{% endblock %}

{% block script %}
<script>
window.onload = () => {
    if (new URL(location.href).searchParams.get('error')) {
        alert('이미 존재하는 이메일입니다.');
    }
};
</script>
{% endblock %}
```

9

익스프레스로 SNS 서비스 만들기

join.html은 회원 가입하는 폼을 보여줍니다.

views/error.html

```
{% extends 'layout.html' %}

{% block content %}
<h1>{{message}}</h1>
<h2>{{error.status}}</h2>
<pre>{{error.stack}}</pre>
{% endblock %}
```

error.html은 서버에 에러가 발생했을 때 에러 내역을 보여줍니다. 에러는 콘솔로 봐도 되지만 브라우저 화면으로 보면 좀 더 편리합니다. 단, 배포 시에는 에러 내용을 보여주지 않는 게 보안상 좋습니다.

마지막으로, 디자인을 위한 CSS 파일입니다.

public/main.css

```
* { box-sizing: border-box; }

html, body { margin: 0; padding: 0; height: 100%; }

.btn {
    display: inline-block;
    padding: 0 5px;
    text-decoration: none;
    cursor: pointer;
    border-radius: 4px;
    background: white;
    border: 1px solid silver;
    color: crimson;
    height: 37px;
    line-height: 37px;
    vertical-align: top;
    font-size: 12px;
}

input[type='text'], input[type='email'], input[type='password'], textarea {
    border-radius: 4px;
    height: 37px;
    padding: 10px;
    border: 1px solid silver;
}

.container { width: 100%; height: 100%; }

@media screen and (min-width: 800px) {
    .container { width: 800px; margin: 0 auto; }
}

.input-group { margin-bottom: 15px; }

.input-group label { width: 25%; display: inline-block; }

.input-group input { width: 70%; }

.half { float: left; width: 50%; margin: 10px 0; }

#join { float: right; }

.profile-wrap {
    width: 100%;
    display: inline-block;
    vertical-align: top;
```

```
margin: 10px 0;
}
@media screen and (min-width: 800px) {
    .profile-wrap { width: 290px; margin-bottom: 0; }
}
.profile {
    text-align: left;
    padding: 10px;
    margin-right: 10px;
    border-radius: 4px;
    border: 1px solid silver;
    background: lightcoral;
}
.user-name { font-weight: bold; font-size: 18px; }
.count { font-weight: bold; color: crimson; font-size: 18px; }
.timeline {
    margin-top: 10px;
    width: 100%;
    display: inline-block;
    border-radius: 4px;
    vertical-align: top;
}
@media screen and (min-width: 800px) { .timeline { width: 500px; } }
#twit-form {
    border-bottom: 1px solid silver;
    padding: 10px;
    background: lightcoral;
    overflow: hidden;
}
#img-preview { max-width: 100%; }
#img-label {
    float: left;
    cursor: pointer;
    border-radius: 4px;
    border: 1px solid crimson;
    padding: 0 10px;
    color: white;
    font-size: 12px;
    height: 37px;
    line-height: 37px;
}
#img { display: none; }
#twit { width: 100%; min-height: 72px; }
```

```

#twit-btn {
    float: right;
    color: white;
    background: crimson;
    border: none;
}
.twit {
    border: 1px solid silver;
    border-radius: 4px;
    padding: 10px;
    position: relative;
    margin-bottom: 10px;
}
.twit-author { display: inline-block; font-weight: bold; margin-right: 10px; }
.twit-follow {
    padding: 1px 5px;
    background: #fff;
    border: 1px solid silver;
    border-radius: 5px;
    color: crimson;
    font-size: 12px;
    cursor: pointer;
}
.twit-img { text-align: center; }
.twit-img img { max-width: 75%; }
.error-message { color: red; font-weight: bold; }
#search-form { text-align: right; }
#join-form { padding: 10px; text-align: center; }
#hashtag-form { text-align: right; }
.footer { text-align: center; }

```

이제 npm start로 서버를 실행하고 http://localhost:8001에 접속하면 다음과 같은 화면이 나타날 것입니다.

▼ 그림 9-3 NodeBird 메인 화면

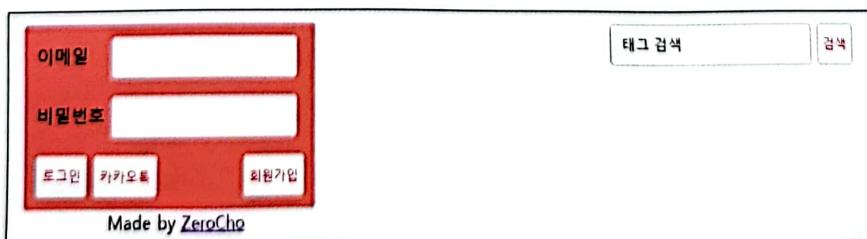


그림 9-4 NodeBird 회원가입 화면

프런트 구성이 완료되었습니다. 다음 절에서는 데이터베이스를 세팅하겠습니다.

9.2 데이터베이스 세팅하기

이번 절에서는 MySQL과 시퀄라이즈로 데이터베이스를 설정합니다.

로그인 기능이 있으므로 사용자 테이블이 필요하고, 게시글을 저장할 게시글 테이블도 필요합니다. 해시태그를 사용하므로 해시태그 테이블도 만들어야 합니다. 팔로잉 기능도 있는데, 이는 조금 뒤에 설명합니다.

models 폴더 안에 user.js와 post.js, hashtag.js를 생성합니다.

models/user.js

```
const Sequelize = require('sequelize');

class User extends Sequelize.Model {
  static initiate(sequelize) {
    User.init({
      email: {
        type: Sequelize.STRING(40),
        allowNull: true,
        unique: true,
      },
      nick: {
        type: Sequelize.STRING(15),
        allowNull: false,
      },
    },
  }
}
```

```

    password: {
      type: Sequelize.STRING(100),
      allowNull: true,
    },
    provider: {
      type: Sequelize.ENUM('local', 'kakao'),
      allowNull: false,
      defaultValue: 'local',
    },
    snsId: {
      type: Sequelize.STRING(30),
      allowNull: true,
    },
  },
  {
    sequelize,
    timestamps: true,
    underscored: false,
    modelName: 'User',
    tableName: 'users',
    paranoid: true,
    charset: 'utf8',
    collate: 'utf8_general_ci',
  });
}

static associate(db) {}

module.exports = User;

```

사용자 정보를 저장하는 모델입니다. 이메일, 닉네임, 비밀번호를 저장하고, SNS 로그인을 했을 경우에는 provider와 snsId를 저장합니다. provider 컬럼을 보면 처음 보는 ENUM이라는 속성을 갖고 있습니다. 이는 넣을 수 있는 값을 제한하는 데이터 형식입니다. 종류로는 이메일/비밀번호 로그인(local)이나 카카오 로그인(kakao) 둘 중 하나만 선택할 수 있게 했으며, 이를 어겼을 때 에러가 발생합니다. 기본적으로 이메일/비밀번호 로그인이라 가정해서 defaultValue를 local로 주었습니다.

테이블 옵션으로 timestamps와 paranoid가 true로 주어졌으므로 createdAt, updatedAt, deletedAt 컬럼도 생성됩니다.

```
models/post.js
const Sequelize = require('sequelize');

class Post extends Sequelize.Model {
  static initiate(sequelize) {
    Post.init({
      content: {
        type: Sequelize.STRING(140),
        allowNull: false,
      },
      img: {
        type: Sequelize.STRING(200),
        allowNull: true,
      },
    }, {
      sequelize,
      timestamps: true,
      underscored: false,
      modelName: 'Post',
      tableName: 'posts',
      paranoid: false,
      charset: 'utf8mb4',
      collate: 'utf8mb4_general_ci',
    });
  }
  static associate(db) {}
}

module.exports = Post;
```

게시글 모델은 게시글 내용과 이미지 경로를 저장합니다. 게시글 등록자의 아이디를 담은 컬럼은 나중에 관계를 설정할 때 시퀄라이즈가 알아서 생성합니다.

models/hashtag.js

```
const Sequelize = require('sequelize');

class Hashtag extends Sequelize.Model {
  static initiate(sequelize) {
    Hashtag.init({
      title: {
        type: Sequelize.STRING(15),
```

```

        allowNull: false,
        unique: true,
    },
}, {
    sequelize,
    timestamps: true,
    underscored: false,
    modelName: 'Hashtag',
    tableName: 'hashtags',
    paranoid: false,
    charset: 'utf8mb4',
    collate: 'utf8mb4_general_ci',
});
}

static associate(db) {}

module.exports = Hashtag;

```

해시태그 모델은 태그 이름을 저장합니다. 해시태그 모델을 따로 두는 것은 나중에 태그로 검색하기 위해서입니다.

이제 생성한 모델들을 시퀄라이즈에 등록합니다. `models/index.js`에는 시퀄라이즈가 자동으로 생성한 코드들이 들어 있을 것입니다. 그것을 다음과 같이 통째로 바꿉니다.

models/index.js

```

const Sequelize = require('sequelize');
const User = require('./user');
const Post = require('./post');
const Hashtag = require('./hashtag');
const env = process.env.NODE_ENV || 'development';
const config = require('../config/config')[env];

const db = {};
const sequelize = new Sequelize(
    config.database, config.username, config.password, config,
);

db.sequelize = sequelize;
db.User = User;
db.Post = Post;

```

```

db.Hashtag = Hashtag;

User.initiate(sequelize);
Post.initiate(sequelize);
Hashtag.initiate(sequelize);

User.associate(db);
Post.associate(db);
Hashtag.associate(db);

module.exports = db;

```

각각의 모델들을 시퀄라이즈 객체에 연결했습니다. 다만, 모델이 많이 늘어나면 `initiate`와 `associate` 부분도 따라서 늘어날 수 있습니다. 실무에서는 모델이 100개가 넘어가는 경우도 흔하므로 그럴 때는 `models/index.js`를 다음과 같이 작성하면 좋습니다.

models/index.js

```

const Sequelize = require('sequelize');
const fs = require('fs');
const path = require('path');
const env = process.env.NODE_ENV || 'development';
const config = require('../config/config')[env];

const db = {};
const sequelize = new Sequelize(
  config.database, config.username, config.password, config,
);

db.sequelize = sequelize;

const basename = path.basename(__filename);
fs
  .readdirSync(__dirname) // 현재 폴더의 모든 파일을 조회
  .filter(file => { // 숨김 파일, index.js, js 확장자가 아닌 파일 필터링
    return (file.indexOf('.') !== 0) && (file !== basename) && (file.slice(-3) ===
'.js');
  })
  .forEach(file => { // 해당 파일의 모델을 불러와서 init
    const model = require(path.join(__dirname, file));
    console.log(file, model.name);
    db[model.name] = model;
  });

```

```
model.initiate(sequelize);
});

Object.keys(db).forEach(modelName => { // associate 호출
  if (db[modelName].associate) {
    db[modelName].associate(db);
  }
});

module.exports = db;
```

사실 이 코드는 `npx sequelize init` 명령어를 수행했을 때 자동으로 생성되는 `models/index.js` 와 거의 비슷합니다. 이제 모델이 무수히 많더라도 자동으로 시퀄라이즈가 모델을 파악할 수 있게 됩니다. 다만 이 코드에는 단점도 있습니다. `models` 폴더에 미완성인 모델이 있을 때 해당 모델도 시퀄라이즈가 읽어들여 연결해버립니다. 시퀄라이즈가 모델을 읽어 테이블을 자동으로 생성한다는 점을 고려할 때, 미완성 테이블이 생길 수도 있는 것입니다. 또한, `models` 폴더에 모델이 아닌 다른 파일을 넣지 않도록 주의해야 합니다. 이 경우에는 `model.initiate`나 `model.associate` 메서드가 존재하지 않아 에러가 발생하게 됩니다. 수동 연결과 자동 연결 모두 장단점이 있으므로, 여러분의 선택에 맡기겠습니다.

이번에는 각 모델 간의 관계를 `associate` 함수 내에 정의해보겠습니다.

models/user.js

```
...
static associate(db) {
  db.UserhasMany(db.Post);
  db.User.belongsToMany(db.User, {
    foreignKey: 'followingId',
    as: 'Followers',
    through: 'Follow',
  });
  db.User.belongsToMany(db.User, {
    foreignKey: 'followerId',
    as: 'Followings',
    through: 'Follow',
  });
}
```

User 모델과 Post 모델은 1(User):N(Post) 관계에 있으므로 hasMany로 연결되어 있습니다. user.getPosts, user.addPosts 같은 관계 메서드들이 생성됩니다.

같은 모델끼리도 N:M 관계를 가질 수 있습니다. 팔로잉 기능이 대표적인 N:M 관계입니다. 사용자 한 명이 팔로워를 여러 명 가질 수도 있고, 한 사람이 여러 명을 팔로잉할 수도 있습니다. User 모델과 User 모델 간에 N:M 관계가 있는 것입니다.

▼ 그림 9-5 같은 테이블 간 N:M 관계

같은 테이블 간 N:M

User(Follower)		Follow		User(Following)	
id	nick	followerId	followingId	id	nick
1	zero	1	2	1	zero
2	nero	1	3	2	nero
3	hero	3	1	3	hero

9

오피스텔로 SNS 서비스 만들기

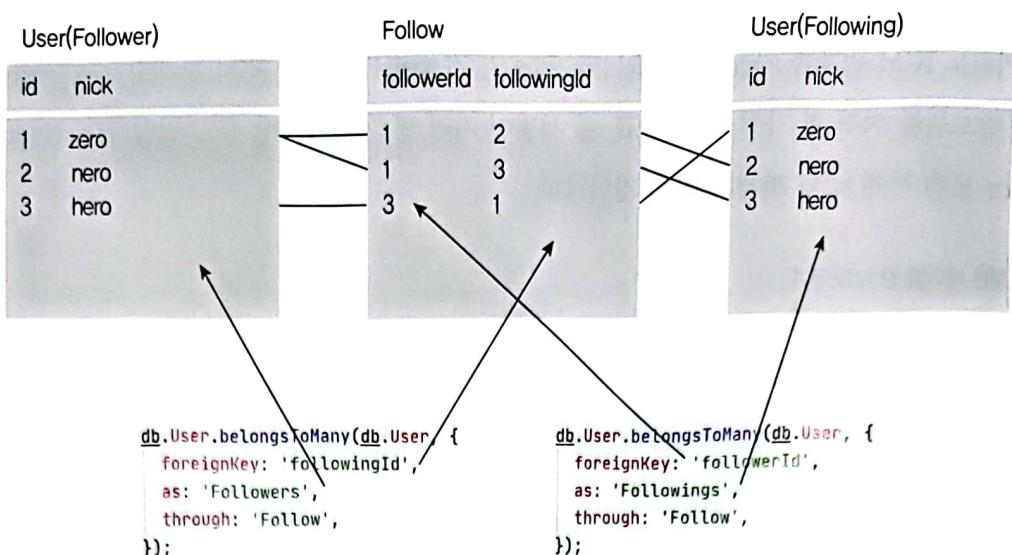
같은 테이블 간 N:M 관계에서는 모델 이름과 컬럼 이름을 따로 정해야 합니다. 모델 이름이 UserUser일 수는 없으니까요. through 옵션을 사용해 생성할 모델 이름을 Follow로 정했습니다.

Follow 모델에서 사용자 아이디를 저장하는 컬럼 이름이 둘 다 userId이면 누가 팔로워이고 누가 팔로잉 중인지 구분되지 않으므로 따로 설정해야 합니다. foreignKey 옵션에 각각 followerId, followingId를 넣어줘서 두 사용자 아이디를 구별했습니다.

같은 테이블 간의 N:M 관계에서는 as 옵션도 넣어야 합니다. 둘 다 User 모델이라 구분되지 않기 때문입니다. 주의할 점은 as는 foreignKey와 반대되는 모델을 가리킨다는 것입니다. foreignKey가 followerId(팔로워 아이디)이면 as는 Followings(팔로잉)이고, foreignKey가 followingId(팔로잉 아이디)이면 as는 Followers(팔로워)여야 합니다. 팔로워(Followers)를 찾으려면 먼저 팔로잉하는 사람의 아이디(followingId)를 찾아야 하는 것이라고 생각하면 됩니다.

▼ 그림 9-6 N:M에서의 as

같은 테이블 간 N:M



as에 특정한 이름을 지정했으니 user.getFollowers, user.getFollowings 같은 관계 메서드를 사용할 수 있습니다. include할 때도 as에 같은 값을 넣으면 관계 쿼리가 작동합니다.

Post 모델도 작성해봅시다.

models/post.js

```

...
static associate(db) {
  db.Post.belongsTo(db.User);
  db.Post.belongsToMany(db.Hashtag, { through: 'PostHashtag' });
}
};

```

User 모델과 Post 모델은 1(User):N(Post) 관계이므로 belongsTo로 연결되어 있습니다. 시퀄라 이즈는 Post 모델에 User 모델의 id를 가리키는 UserId 컬럼을 추가합니다. 어디에 컬럼이 추가되는 것인지는 관계를 생각해보면 쉽습니다. 사용자가 한 명이고 그에 속한 게시글이 여러 개이므로 각각의 게시글에 게시글의 주인이 누구인지 넣어야 합니다. belongsTo는 게시글에 붙습니다. post.getUser, post.addUser와 같은 관계 메서드가 생성됩니다.

Post 모델과 Hashtag 모델은 N:M 관계입니다. 7.6.3.3절에서 설명한 것과 동일합니다. N:M 관계이므로 PostHashtag라는 중간 모델이 생기고, 각각 postId와 hashtagId라는 foreignKey도 추가됩니다. as는 따로 지정하지 않았으니 post.getHashtags, post.addHashtags, hashtags.getPosts 같은 기본 이름의 관계 메서드들이 생성됩니다.

models/hashtag.js

```
...
static associate(db) {
  db.Hashtag.belongsToMany(db.Post, { through: 'PostHashtag' });
}
};
```

Hashtag 모델은 Post 모델과 N:M 관계이므로 관계를 설정했습니다. 이에 대한 설명은 Post 모델과 같습니다.

NodeBird의 모델은 총 다섯 개, 즉 직접 생성한 User, Hashtag, Post와 시퀄라이즈가 관계를 파악해 생성한 PostHashtag, Follow까지입니다.

자동으로 생성된 모델도 다음과 같이 접근할 수 있습니다. 다음 모델을 통해 쿼리 호출이나 관계 메서드 사용도 가능합니다.

```
db.sequelize.models.PostHashtag
db.sequelize.models.Follow
```

이제 생성한 모델을 데이터베이스 및 서버와 연결합니다. 아직 데이터베이스를 만들지 않았으므로 데이터베이스부터 만들겠습니다. 데이터베이스의 이름은 nodebird입니다.

7장에서는 MySQL 프롬프트를 통해 SQL문으로 데이터베이스를 만들었습니다. 하지만 시퀄라이즈는 config.json을 읽어 데이터베이스를 생성해주는 기능이 있습니다. 따라서 config.json을 먼저 수정합니다. MySQL 비밀번호를 password에 넣고 데이터베이스 이름을 nodebird로 바꿉니다. 자동으로 생성한 config.json에 operatorAliases 속성이 들어 있다면 삭제합니다.

config/config.json

```
{
  "development": {
    "username": "root",
    "password": "[root 비밀번호]",
    "database": "nodebird",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
}
```

콘솔에서 npx sequelize db:create 명령어를 입력하면 데이터베이스가 생성됩니다.

콘솔

```
$ npx sequelize db:create
Sequelize CLI [Node: 18.0.0, CLI: 6.4.1, ORM: 6.19.0]
```

```
Loaded configuration file "config\config.json".
Using environment "development".
Database nodebird created.
```

데이터베이스를 생성했으니 모델을 서버와 연결합니다.

app.js

```
...
dotenv.config();
const pageRouter = require('./routes/page');
const { sequelize } = require('./models');

const app = express();
app.set('port', process.env.PORT || 8001);
app.set('view engine', 'html');
nunjucks.configure('views', {
  express: app,
  watch: true,
});
sequelize.sync({ force: false })
  .then(() => {
    console.log('데이터베이스 연결 성공');
  })
  .catch((err) => {
    console.error(err);
  });
app.use(morgan('dev'));
...
```

서버 쪽 세팅이 완료되었습니다. 이제 서버를 실행합니다. 시퀄라이즈는 테이블 생성 쿼리문에 IF NOT EXISTS를 넣어주므로 테이블이 없을 때 테이블을 자동으로 생성합니다.

콘솔

```
$ npm start
> nodebird@0.0.1 start
> nodemon app
```

```
[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
8001 번 포트에서 대기 중
Executing (default): CREATE TABLE IF NOT EXISTS `users` (`id` INTEGER NOT NULL auto_increment , `title` VARCHAR(15) NOT NULL UNIQUE, `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_general_ci;
...
데이터베이스 연결 성공
```

9.3

Passport 모듈로 로그인 구현하기

NODE.JS

SNS 서비스이므로 로그인이 필요합니다. 회원 가입과 로그인을 직접 구현할 수도 있지만, 세션과 쿠키 처리 등 복잡한 작업이 많으므로 검증된 모듈을 사용하는 것이 좋습니다. 바로 Passport를 사용하는 것입니다. 이 모듈은 이름처럼 우리의 서비스를 사용할 수 있게 해주는 여권 같은 역할을 합니다.

요즘에는 서비스에 로그인할 때 아이디와 비밀번호를 사용하지 않고 구글, 페이스북, 카카오톡 같은 기존의 SNS 서비스 계정으로 로그인하기도 합니다. 이 또한 Passport를 사용해서 해결할 수 있습니다. 이번 절에서는 자체 회원 가입 및 로그인을 하는 방법뿐만 아니라 한국에서 많이 사용하는 SNS인 카카오톡을 이용해 로그인하는 방법도 알아봅니다.

먼저 Passport 관련 패키지들을 설치합니다.

콘솔

```
$ npm i passport passport-local passport-kakao bcrypt
```

설치 후 Passport 모듈을 미리 app.js와 연결합시다. Passport 모듈은 조금 뒤에 만듭니다.

app.js

```
...
const dotenv = require('dotenv');
const passport = require('passport');

dotenv.config();
const pageRouter = require('./routes/page');
const { Sequelize } = require('./models');
const passportConfig = require('./passport');

const app = express();
passportConfig(); // 패스포트 설정
app.set('port', process.env.PORT || 8001);
app.set('view engine', 'html');

...
app.use(session({
    resave: false,
    saveUninitialized: false,
    secret: process.env.COOKIE_SECRET,
    cookie: {
        httpOnly: true,
        secure: false,
    },
}));
app.use(passport.initialize());
app.use(passport.session());

app.use('/', pageRouter);
...
```

`require('./passport')`는 `require('./passport/index.js')`와 같습니다. 폴더 내의 `index.js` 파일은 `require`할 때 이름을 생략할 수 있습니다.

`passport.initialize` 미들웨어는 요청(req 객체)에 passport 설정을 심고, `passport.session` 미들웨어는 `req.session` 객체에 passport 정보를 저장합니다. `req.session` 객체는 `express-session`에서 생성하는 것이므로 passport 미들웨이는 `express-session` 미들웨어보다 뒤에 연결해야 합니다.

passport 폴더 내부에 `index.js` 파일을 만들고 Passport 관련 코드를 작성해봅시다.

```

const passport = require('passport');
const local = require('./localStrategy');
const kakao = require('./kakaoStrategy');
const User = require('../models/user');

module.exports = () => {
  passport.serializeUser((user, done) => {
    done(null, user.id);
  });

  passport.deserializeUser((id, done) => {
    User.findOne({ where: { id } })
      .then(user => done(null, user))
      .catch(err => done(err));
  });

  local();
  kakao();
};

```

모듈 내부를 보면 `passport.serializeUser`와 `passport.deserializeUser`가 있습니다. 이 부분이 Passport를 이해하는 핵심입니다.

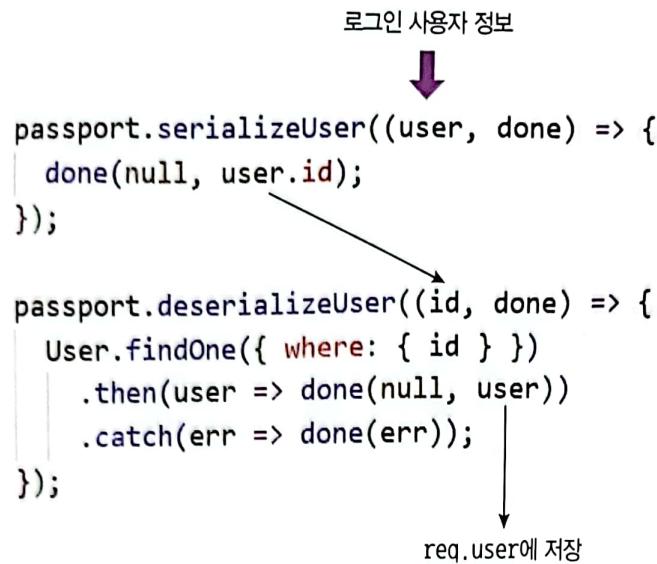
`serializeUser`는 로그인 시 실행되며, `req.session`(세션) 객체에 어떤 데이터를 저장할지 정하는 메서드입니다. 매개변수로 `user`를 받고 나서 `done` 함수에 두 번째 인수로 `user.id`를 넘기고 있으며, 매개변수 `user`가 어디서 오는지는 나중에 설명합니다. 지금은 그냥 사용자 정보가 들어 있다고 생각하면 됩니다.

`done` 함수의 첫 번째 인수는 에러가 발생할 때 사용하는 것이고, 두 번째 인수에는 저장하고 싶은 데이터를 넣습니다. 로그인 시 사용자 데이터를 세션에 저장하는데(4.3절을 떠올려보세요), 세션에 사용자 정보를 모두 저장하면 세션의 용량이 커지고 데이터 일관성에 문제가 발생하므로 사용자의 아이디만 저장하라고 명령한 것입니다.

`serializeUser`가 로그인할 때만 실행된다면 `deserializeUser`는 각 요청마다 실행됩니다. `passport.session` 미들웨어가 이 메서드를 호출합니다. `serializeUser`의 `done`의 두 번째 인수로 넣었던 데이터가 `deserializeUser`의 매개변수가 됩니다. 여기서는 사용자의 아이디입니다. 조금 전에 `serializeUser`에서 세션에 저장했던 아이디를 받아 데이터베이스에서 사용자 정보를 조회

합니다. 조회한 정보를 req.user에 저장하므로 앞으로 req.user를 통해 로그인한 사용자의 정보를 가져올 수 있습니다.

▼ 그림 9-7 serializeUser와 deserializeUser의 관계



즉, `serializeUser`는 사용자 정보 객체에서 아이디만 추려 세션에 저장하는 것이고, `deserializeUser`는 세션에 저장한 아이디를 통해 사용자 정보 객체를 불러오는 것입니다. 이는 세션에 불필요한 데이터를 담아두지 않기 위한 과정입니다.

전체 과정은 다음과 같습니다.

1. /auth/login 라우터를 통해 로그인 요청이 들어옴
2. 라우터에서 `passport.authenticate` 메서드 호출
3. 로그인 전략(LocalStrategy) 수행
4. 로그인 성공 시 사용자 정보 객체와 함께 `req.login` 호출
5. `req.login` 메서드가 `passport.serializeUser` 호출
6. `req.session`에 사용자 아이디만 저장해서 세션 생성
7. express-session에 설정한 대로 브라우저에 `connect.sid` 세션 쿠키 전송
8. 로그인 완료

1~4번은 아직 구현하지 않았으며, 로컬 로그인을 구현하면서 상응하는 코드를 보게 될 것입니다. 다음은 로그인 이후의 과정입니다.

1. 요청이 들어옴(어떠한 요청이든 상관없음)
2. 라우터에 요청이 도달하기 전에 passport.session 미들웨어가 passport.deserializeUser 메서드 호출
3. connect.sid 세션 쿠키를 읽고 세션 객체를 찾아서 req.session으로 만들
4. req.session에 저장된 아이디로 데이터베이스에서 사용자 조회
5. 조회된 사용자 정보를 req.user에 저장
6. 라우터에서 req.user 객체 사용 가능

passport/index.js의 localStrategy와 kakaoStrategy 파일은 각각 로컬 로그인과 카카오 로그인 전략에 대한 파일입니다. Passport는 로그인 시의 동작을 전략(strategy)이라는 용어로 표현하고 있습니다. 전략이라는 이름이 거창하긴 하지만, 그냥 로그인 과정을 어떻게 처리할지 설명하는 파일이라고 생각하면 됩니다. 9.3.1절과 9.3.2절에서 이 파일들을 작성합니다.

9.3.1 로컬 로그인 구현하기

로컬 로그인이란 다른 SNS 서비스를 통해 로그인하지 않고 자체적으로 회원 가입 후 로그인하는 것을 의미합니다. 즉, 아이디/비밀번호 또는 이메일/비밀번호를 통해 로그인하는 것입니다.

Passport에서 이를 구현하려면 passport-local 모듈이 필요한데, 이미 설치했으므로 로컬 로그인 전략만 세우면 됩니다. 로그인에만 해당하는 전략이므로 회원 가입은 따로 만들어야 합니다.

회원 가입, 로그인, 로그아웃 라우터를 먼저 만들어봅시다. 이러한 라우터에는 접근 조건이 있습니다. 로그인한 사용자는 회원 가입과 로그인 라우터에 접근하면 안 됩니다. 이미 로그인을 했으니까요. 마찬가지로 로그인하지 않은 사용자는 로그아웃 라우터에 접근하면 안 됩니다. 따라서 라우터에 접근 권한을 제어하는 미들웨어가 필요합니다. 미들웨어를 만들어보며 Passport가 req 객체에 추가해주는 req.isAuthenticated 메서드를 사용해봅시다.

middlewares 폴더를 만들고 그 안에 index.js를 작성합니다.

middlewares/index.js

```
exports.isLoggedIn = (req, res, next) => {
  if (req.isAuthenticated()) {
    next();
  } else {
```

```
    res.status(403).send('로그인 필요');
}

};

exports.isNotLoggedIn = (req, res, next) => {
  if (!req.isAuthenticated()) {
    next();
  } else {
    const message = encodeURIComponent('로그인한 상태입니다.');
    res.redirect(`/?error=${message}`);
  }
};
```

Passport는 req 객체에 isAuthenticated 메서드를 추가합니다. 로그인 중이면 req.isAuthenticated() 가 true이고, 그렇지 않으면 false입니다. 따라서 로그인 여부를 이 메서드로 파악할 수 있습니다. 라우터 중에 로그아웃 라우터나 이미지 업로드 라우터 등은 로그인한 사람만 접근할 수 있게 해야 하고, 회원가입 라우터나 로그인 라우터는 로그인하지 않은 사람만 접근할 수 있게 해야 합니다. 이럴 때 라우터에 로그인 여부를 검사하는 미들웨어를 넣어 걸러낼 수 있습니다.

isLoggedIn과 isNotLoggedIn 미들웨어를 만들었습니다. 이 미들웨어가 page 라우터에 어떻게 사용되는지 봅시다.

routes/page.js

```
const express = require('express');
const { isLoggedIn, isNotLoggedIn } = require('../middlewares');
const { renderProfile, renderJoin, renderMain } = require('../controllers/page');

const router = express.Router();

router.use((req, res, next) => {
  res.locals.user = req.user;
  res.locals.followerCount = 0;
  res.locals.followingCount = 0;
  res.locals.followingIdList = [];
  next();
});

router.get('/profile', isLoggedIn, renderProfile);
```

```
router.get('/join', isNotLoggedIn, renderJoin);
```

```
...
```

자신의 프로필은 로그인해야 볼 수 있으므로 `isLoggedin` 미들웨어를 사용합니다. `req.isAuthenticated()`가 `true`여야 `next`가 호출되어 `res.render`가 있는 미들웨어로 넘어갈 수 있습니다. `false`라면 로그인 창이 있는 메인 페이지로 리다이렉트됩니다.

회원 가입 페이지는 로그인하지 않은 사람에게만 보여야 합니다. 따라서 `isNotLoggedIn` 미들웨어로 `req.isAuthenticated()`가 `false`일 때만 `next`를 호출하도록 했습니다.

로그인 여부로만 미들웨어를 만들 수 있는 것이 아니라 팔로잉 여부, 관리자 여부 등의 미들웨어를 만들 수 있으므로 다양하게 활용할 수 있습니다. `res.locals.user` 속성에 `req.user`를 넣은 점에 주목해주세요. 네이티브에서 `user` 객체를 통해 사용자 정보에 접근할 수 있게 되었습니다.

이제 회원 가입, 로그인, 로그아웃 라우터와 컨트롤러를 작성해봅시다.

routes/auth.js

```
const express = require('express');
const passport = require('passport');

const { isLoggedIn, isNotLoggedIn } = require('../middlewares');
const { join, login, logout } = require('../controllers/auth');

const router = express.Router();

// POST /auth/join
router.post('/join', isNotLoggedIn, join);

// POST /auth/login
router.post('/login', isNotLoggedIn, login);

// GET /auth/logout
router.get('/logout', isLoggedIn, logout);

module.exports = router;
```

controllers/auth.js

```
const bcrypt = require('bcrypt');
const passport = require('passport');
const User = require('../models/user');
```

```
exports.join = async (req, res, next) => {
  const { email, nick, password } = req.body;
  try {
    const exUser = await User.findOne({ where: { email } });
    if (exUser) {
      return res.redirect('/join?error=exist');
    }
    const hash = await bcrypt.hash(password, 12);
    await User.create({
      email,
      nick,
      password: hash,
    });
    return res.redirect('/');
  } catch (error) {
    console.error(error);
    return next(error);
  }
}
```

```
exports.login = (req, res, next) => {
  passport.authenticate('local', (authError, user, info) => {
    if (authError) {
      console.error(authError);
      return next(authError);
    }
    if (!user) {
      return res.redirect(`/?error=${info.message}`);
    }
    return req.login(user, (loginError) => {
      if (loginError) {
        console.error(loginError);
        return next(loginError);
      }
      return res.redirect('/');
    });
  })(req, res, next); // 미들웨어 내의 미들웨어에는 (req, res, next)를 불입니다
};
```

```
exports.logout = (req, res) => {
  req.logout(() => {
    res.redirect('/');
  });
};
```

나중에 app.js와 연결할 때 /auth 접두사를 붙일 것이므로 라우터의 주소는 각각 /auth/join, /auth/login, /auth/logout이 됩니다.

- ① 회원 가입 컨트롤러입니다.** 기존에 같은 이메일로 가입한 사용자가 있는지 조회한 뒤, 있다면 회원 가입 페이지로 되돌려보냅니다. 단, 주소 뒤에 에러를 쿼리스트링으로 표시합니다. 없다면 비밀번호를 암호화하고 사용자 정보를 생성합니다.

회원 가입 시 비밀번호는 암호화해서 저장해야 합니다. 이번에는 bcrypt 모듈을 사용했습니다(crypto 모듈의 pbkdf2 메서드를 사용해서 암호화할 수도 있습니다). bcrypt 모듈의 hash 메서드를 사용하면 손쉽게 비밀번호를 암호화할 수 있습니다. bcrypt의 두 번째 인수는 pbkdf2의 반복 횟수와 비슷한 기능을 합니다. 숫자가 커질수록 비밀번호를 알아내기 어려워지지만 암호화 시간도 오래 걸립니다. 12 이상을 추천하며, 31까지 사용할 수 있습니다. 프로미스를 지원하는 함수이므로 await을 사용했습니다.

- ② 로그인 컨트롤러입니다.** 로그인 요청이 들어오면 passport.authenticate('local') 미들웨어가 로컬 로그인 전략을 수행합니다. 미들웨어인데 라우터 미들웨어 안에 들어 있으며, 미들웨어에 사용자 정의 기능을 추가하고 싶을 때 이렇게 할 수 있습니다. 이럴 때는 내부 미들웨어에 (req, res, next)를 인수로 제공해서 호출하면 됩니다.

전략 코드는 잠시 후에 작성합니다. 전략이 성공하거나 실패하면 authenticate 메서드의 콜백 함수가 실행됩니다. 콜백 함수의 첫 번째 매개변수(authErr) 값이 있다면 실패한 것입니다. 두 번째 매개변수 자리는 사용자 정보입니다. 이 자리에 값이 있다면 성공한 것이고, 이 값으로 req.login 메서드를 호출합니다. Passport는 req 객체에 login과 logout 메서드를 추가합니다. req.login은 passport.serializeUser를 호출하고, req.login에 제공하는 user 객체가 serializeUser로 넘어가게 됩니다. 또한, 이때 connect.sid 세션 쿠키가 브라우저에 전송됩니다.

- ③ 로그아웃 컨트롤러입니다.** req.logout 메서드는 req.user 객체와 req.session 객체를 제거합니다. req.logout 메서드는 콜백 함수를 인수로 받고, 세션 정보를 지운 후 콜백 함수가 실행됩니다. 콜백 함수에서는 메인 페이지로 되돌아가면 됩니다. 로그인이 해제되어 있을 것입니다.

로그인 전략을 구현했습니다. passport-local 모듈에서 Strategy 생성자를 불러와 그 안에 전략을 구현하면 됩니다.

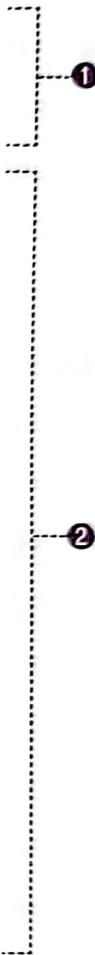
```

const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const bcrypt = require('bcrypt');

const User = require('../models/user');

module.exports = () => {
  passport.use(new LocalStrategy({
    usernameField: 'email',
    passwordField: 'password',
    passReqToCallback: false,
  }, async (email, password, done) => {
    try {
      const exUser = await User.findOne({ where: { email } });
      if (exUser) {
        const result = await bcrypt.compare(password, exUser.password);
        if (result) {
          done(null, exUser);
        } else {
          done(null, false, { message: '비밀번호가 일치하지 않습니다.' });
        }
      } else {
        done(null, false, { message: '가입되지 않은 회원입니다.' });
      }
    } catch (error) {
      console.error(error);
      done(error);
    }
  }));
};

```



- ①** LocalStrategy 생성자의 첫 번째 인수로 주어진 객체는 전략에 관한 설정을 하는 곳입니다. usernameField와 passwordField에는 일치하는 로그인 라우터의 req.body 속성명을 적으면 됩니다. req.body.email에 이메일이, req.body.password에 비밀번호가 담겨 들어오므로 email과 password를 각각 넣었습니다.
- ②** 실제 전략을 수행하는 async 함수입니다. LocalStrategy 생성자의 두 번째 인수로 들어갑니다. 첫 번째 인수에서 넣어준 email과 password는 각각 async 함수의 첫 번째와 두 번째 매개 변수가 됩니다. 세 번째 매개변수인 done 함수는 passport.authenticate의 콜백 함수입니다.

전략의 내용은 다음과 같습니다. 먼저 사용자 데이터베이스에서 일치하는 이메일이 있는지 찾은 후, 있다면 bcrypt의 compare 함수로 비밀번호를 비교합니다. 비밀번호까지 일치한다면 done 함수의 두 번째 인수로 사용자 정보를 넣어 보냅니다. 두 번째 인수를 사용하지 않는 경우는 로그인에 실패했을 때뿐입니다. done 함수의 첫 번째 인수를 사용하는 경우는 서버 쪽에서 에러가 발생했을 때이고, 세 번째 인수를 사용하는 경우는 로그인 처리 과정에서 비밀번호가 일치하지 않거나 존재하지 않는 회원인 경우와 같은 사용자 정의 에러가 발생했을 때입니다.

▼ 그림 9-8 done과 passport.authenticate의 관계

1. 로그인 성공 시

```
done(null, exUser);
  ↓   ↓
passport.authenticate('local', (authError, user, info) => {
```

2. 로그인 실패 시

```
done(null, false, { message: '비밀번호가 일치하지 않습니다.' });
  ↓   ↓   ↓
passport.authenticate('local', (authError, user, info) => {
```

3. 서버 에러 시

```
done(error);
  ↓
passport.authenticate('local', (authError, user, info) => {
```

done이 호출된 후에는 다시 passport.authenticate의 콜백 함수에서 나머지 로직이 실행됩니다. 로그인에 성공했다면 메인 페이지로 리다이렉트되면서 로그인 폼 대신 회원 정보가 뜰 것입니다. 아직 auth 라우터를 연결하지 않았으므로 코드가 동작하지 않습니다. 카카오 로그인까지 구현한 후 연결해봅시다.

9.3.2 카카오 로그인 구현하기

카카오 로그인이란 로그인 인증 과정을 카카오에 맡기는 것을 뜻합니다. 사용자는 번거롭게 새로운 사이트에 회원 가입하지 않아도 돼서 좋고, 서비스 제공자는 로그인 과정을 안심하고 검증된 SNS에 맡길 수 있어 좋습니다.

SNS 로그인의 특징은 회원 가입 절차가 따로 없다는 것입니다. 처음 로그인할 때는 회원 가입 처리를 해야 하고, 두 번째 로그인부터는 로그인 처리를 해야 합니다. 따라서 SNS 로그인 전략은 로컬 로그인 전략보다 다소 복잡합니다.

passport-kakao 모듈로부터 Strategy 생성자를 불러와 전략을 구현합니다.

passport/kakaoStrategy.js

```
const passport = require('passport');
const KakaoStrategy = require('passport-kakao').Strategy;

const User = require('../models/user');

module.exports = () => {
  passport.use(new KakaoStrategy({
    clientID: process.env.KAKAO_ID,
    callbackURL: '/auth/kakao/callback',
  }, async (accessToken, refreshToken, profile, done) => {
    console.log('kakao profile', profile);
    try {
      const exUser = await User.findOne({
        where: { snsId: profile.id, provider: 'kakao' },
      });
      if (exUser) {
        done(null, exUser);
      } else {
        const newUser = await User.create({
          email: profile._json?.kakao_account?.email,
          nick: profile.displayName,
          snsId: profile.id,
          provider: 'kakao',
        });
        done(null, newUser);
      }
    } catch (error) {
      console.error(error);
      done(error);
    }
  }));
};


```



- ① 로컬 로그인과 마찬가지로 카카오 로그인에 대한 설정을 합니다. clientID는 카카오에서 발급 해주는 아이디입니다. 노출되지 않아야 하므로 process.env.KAKAO_ID로 설정했으며, 나중에 아이디를 발급받아 .env 파일에 넣을 것입니다. callbackURL은 카카오로부터 인증 결과를 받을 라우터 주소입니다. 아래에서 라우터를 작성할 때 이 주소를 사용합니다.

- ② 먼저 기존에 카카오를 통해 회원 가입한 사용자가 있는지 조회합니다. 있다면 이미 회원 가입되어 있는 경우이므로 사용자 정보와 함께 done 함수를 호출하고 전략을 종료합니다.
- ③ 없다면 회원 가입을 진행합니다. 카카오에서는 인증 후 callbackURL에 적힌 주소로 accessToken, refreshToken과 profile을 보냅니다. profile에 사용자 정보들이 들어 있습니다. 카카오에서 보내주는 것이므로 데이터는 console.log 메서드로 확인해보는 것이 좋습니다. profile 객체에서 원하는 정보를 꺼내와 회원 가입을 하면 됩니다. email의 경우 profile의 속성이 undefined일 수도 있어 옵셔널 체이닝 문법을 사용했습니다. 사용자를 생성한 뒤 done 함수를 호출합니다.

이제 카카오 로그인 라우터를 만들어봅시다. 로그아웃 라우터 아래에 추가하면 됩니다. 회원 가입을 따로 코딩할 필요가 없고 카카오 로그인 전략이 대부분의 로직을 처리하므로 라우터가 상대적으로 간단합니다. 코드가 매우 간단하므로 컨트롤러도 굳이 다른 파일에 분리하지 않았습니다.

```
routes/auth.js
...
router.get('/logout', isLoggedIn, logout);

// GET /auth/kakao
router.get('/kakao', passport.authenticate('kakao'));

// GET /auth/kakao/callback
router.get('/kakao/callback', passport.authenticate('kakao', {
  failureRedirect: '/?error=카카오로그인 실패',
}), (req, res) => {
  res.redirect('/'); // 성공 시에는 /로 이동
});

module.exports = router;
```

GET /auth/kakao로 접근하면 카카오 로그인 과정이 시작됩니다. layout.html의 카카오톡 버튼에 /auth/kakao 링크가 붙어 있습니다. GET /auth/kakao에서 로그인 전략(KakaoStrategy)을 수행하는데, 처음에는 카카오 로그인 창으로 리다이렉트합니다. 그 창에서 로그인 후 성공 여부 결과를 GET /auth/kakao/callback으로 받습니다. 이 라우터에서는 카카오 로그인 전략(KakaoStrategy)을 다시 수행합니다.

로컬 로그인과 다른 점은 passport.authenticate 메서드에 콜백 함수를 제공하지 않는다는 점입니다. 카카오 로그인은 로그인 성공 시 내부적으로 req.login을 호출하므로 우리가 직접 호출할

필요가 없습니다. 콜백 함수 대신 로그인에 실패했을 때 어디로 이동할지를 `failureRedirect` 속성에 적고, 성공 시에도 어디로 이동할지를 다음 미들웨어에 적습니다.

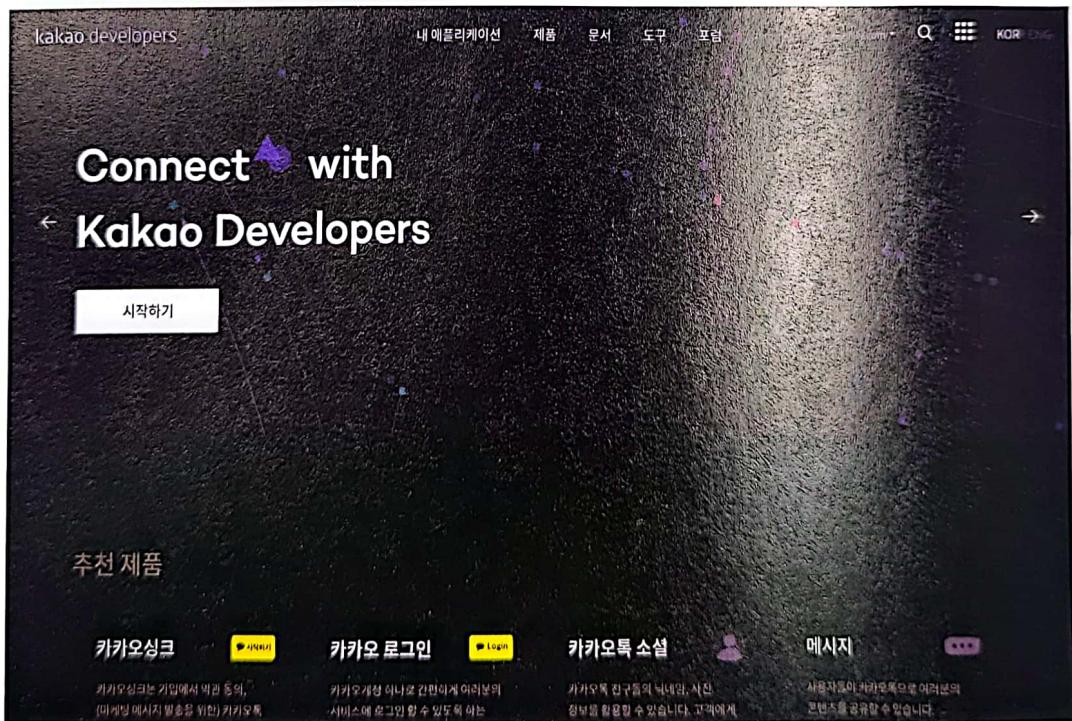
추가한 auth 라우터를 `app.js`에 연결합니다.

app.js

```
...  
const pageRouter = require('./routes/page');  
const authRouter = require('./routes/auth');  
const { sequelize } = require('./models');  
  
...  
app.use('/', pageRouter);  
app.use('/auth', authRouter);  
...
```

아직 끝난 것이 아닙니다. `kakaoStrategy.js`에서 사용하는 `clientID`를 발급받아야 합니다. 카카오 로그인을 위해서는 카카오 개발자 계정과 카카오 로그인용 애플리케이션 등록이 필요합니다. <https://developers.kakao.com>에 접속해 카카오 회원 가입 또는 로그인을 합니다.

▼ 그림 9-9 <https://developers.kakao.com> 접속



로그인 후 내 애플리케이션 메뉴에 가서 애플리케이션 추가하기 버튼을 누릅니다.

▼ 그림 9-10 애플리케이션 추가하기 버튼 클릭



다음 그림과 같이 카카오용 NodeBird 앱을 만듭니다. 앱 아이콘은 등록하지 않아도 되고, 앱 이름과 회사 이름은 여러분이 원하는 대로 입력하면 됩니다.

▼ 그림 9-11 앱 정보 작성 화면

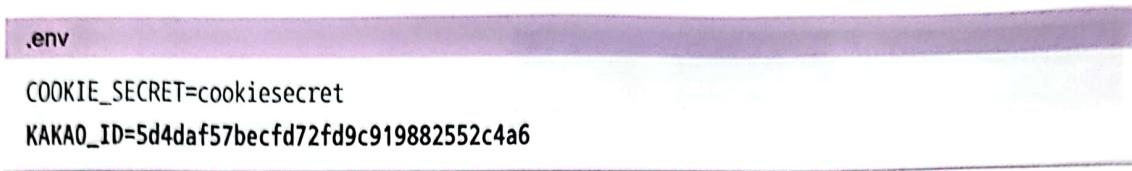
 A screenshot of the 'Add Application' form. It has fields for 'App Icon' (with placeholder '업로드'), 'App Name' (input field containing 'NodeBird'), and 'Company Name' (input field containing 'NodeBird'). Below these are two bullet points: '입력된 정보는 사용자가 카카오 로그인을 할 때 표시됩니다.' and '정보가 정확하지 않은 경우 서비스 이용이 제한될 수 있습니다.'. At the bottom are '취소' and '저장' buttons.

생성한 NodeBird 앱으로 들어가면 앱 키가 보입니다.

▼ 그림 9-12 앱 생성 후 화면

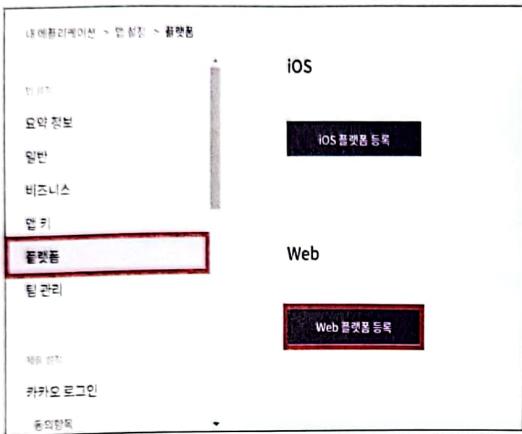
A screenshot of the 'App Details' page for 'NodeBird'. It shows the '요약 정보' section with 'ID 380815', 'OWNER', and 'Web'. Below that is the '앱 키' section, which lists several API keys: '네이티브 앱 키' (505ec0d52f2d7cf7d4cdab03ba648f46), 'REST API 키' (5d4daf57becfd72fd9c919882552c4a6), 'JavaScript 키' (f4c05b99504a5e8204576bccddf591bf), and 'Admin 키' (5b4c85dab468d78a571b449b0426e292).

그림 9-11의 REST API 키를 복사해 .env 파일에 넣습니다. 여러분의 키는 이 책의 키와 다릅니다. 따라서 이 책의 키 대신 여러분의 키를 넣어야 합니다.



앱 설정 > 플랫폼에서 Web 플랫폼 등록 메뉴를 선택합니다.

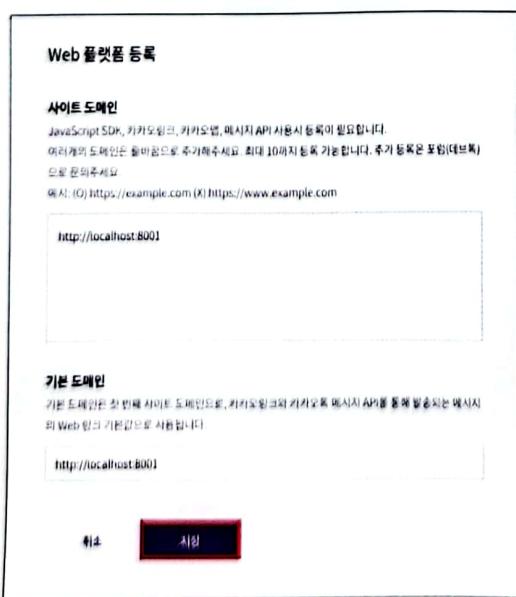
▼ 그림 9-13 플랫폼 등록 버튼 클릭



사이트 도메인에는 `http://localhost:8001`을 입력합니다. 만약 8001 외의 다른 포트를 사용하고 있다면 해당 포트를 적어야 합니다.

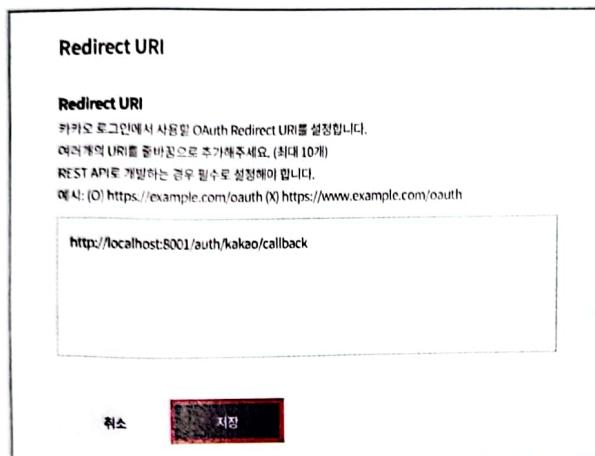
`Enter`를 눌러 여러 개의 주소를 입력할 수도 있습니다. 입력 후 저장 버튼을 누릅니다.

▼ 그림 9-14 사이트 도메인 추가



제품 설정 > 카카오 로그인 메뉴에서 OFF 상태의 활성화 설정 상태 스위치를 클릭해 ON 상태로 활성화시킵니다. 그 후 Redirect URL 등록 버튼을 클릭해 Redirect URI를 수정합니다. `http://localhost:8001/auth/kakao/callback`을 입력한 후 저장 버튼을 누릅니다. `/auth/kakao/callback` 부분은 kakaoStrategy.js의 callbackURL과 일치해야 합니다.

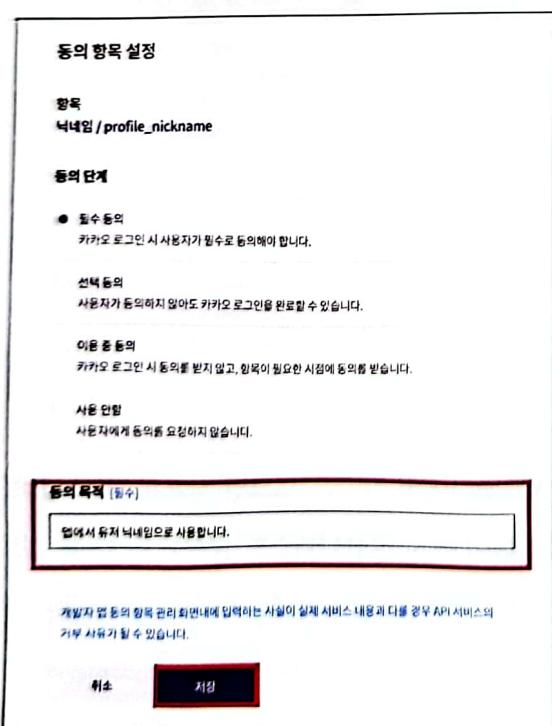
▼ 그림 9-15 Redirect URI 등록



제품 설정 > 카카오 로그인 > 동의 항목 메뉴로 가서 로그인 동의 항목을 작성합니다. 원하는 정보가 있다면 설정 버튼을 누르고 동의 단계와 동의 목적을 입력하면 됩니다.

예제에서는 닉네임과 카카오 계정(이메일)이 반드시 필요합니다. 닉네임 동의 단계에서는 필수 동의를 선택하고 동의 목적을 입력한 후 저장 버튼을 누릅니다.

▼ 그림 9-16 로그인 동의 항목(닉네임)



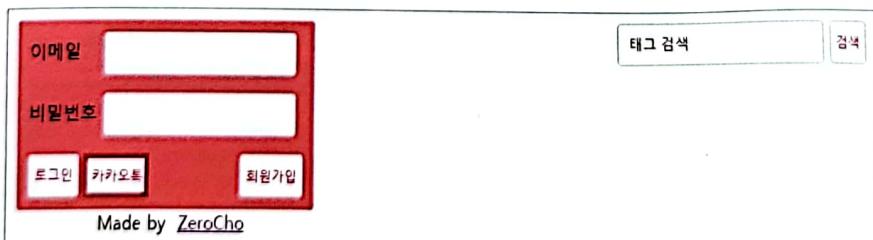
이메일 동의 단계에서는 선택 동의를 선택하고, 혹시나 값이 없는 경우를 대비해 카카오 계정으로 정보 수집 후 제공 체크박스에 체크 표시를 한 후 동의 목적을 입력합니다. 입력 완료 후 저장 버튼을 누릅니다.

▼ 그림 9-17 로그인 동의 항목(이메일)

The screenshot shows the 'Consent Form' for 'Email'. It includes sections for 'Email' (Email), 'Consent Type' (동의 타입), 'Consent Purpose' (동의 목적), and 'Consent Details' (동의 내용). A red box highlights the 'Kakao Account Information Collection After Consent' section, which contains a checked checkbox for 'Requesting consent information from users who have not yet provided it' (사용자에게 길이 없는 경우 카카오 계정 정보 입력을 요청하여 수집). Below this, there is a note about using the user's Kakao ID for login ('업의 로그인 아이디로 이메일을 사용합니다.') and a note about app usage ('개발자 앱 등의 창작 관리 화면내에 입력하는 사실이 실제 서비스 내용과 다를 경우 API 서비스의 거부 사용가 될 수 있습니다.'). At the bottom are '취소' (Cancel) and '저장' (Save) buttons.

이제 NodeBird 서비스에서 카카오톡 버튼을 눌러 GET /auth/kakao 라우터로 요청을 보내면 카카오 인증이 시작됩니다.

▼ 그림 9-18 카카오톡 버튼 클릭



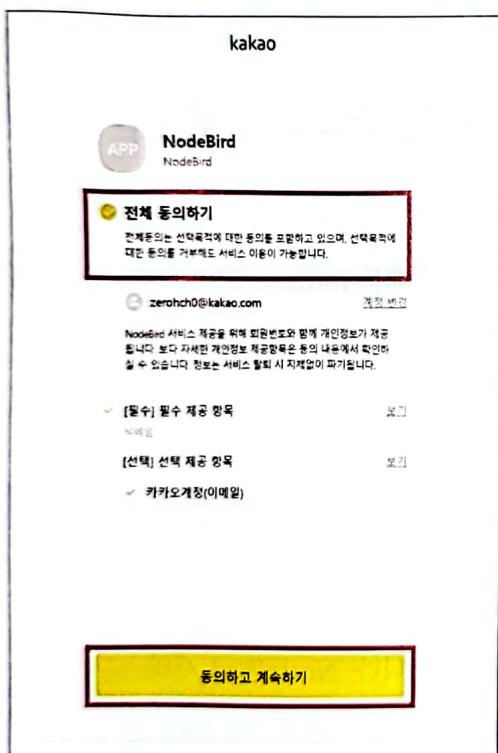
GET /auth/kakao 라우터의 passport.authenticate('kakao')에서 카카오 로그인 창으로 리다이렉트합니다. 이미 카카오에 로그인되어 있다면 로그인 화면이 뜨지 않습니다.

▼ 그림 9-19 카카오 로그인 화면



카카오 계정으로 로그인하면 NodeBird 애플리케이션에 카카오 닉네임과 이메일을 제공할지 묻는 화면으로 넘어갑니다. 전체 동의하기에 체크하고 동의하고 계속하기 버튼을 누릅니다.

▼ 그림 9-20 애플리케이션 제공 항목 동의 화면



로컬 로그인과 카카오 로그인을 모두 해보면서 Passport의 인증 과정을 다시 한 번 되짚어보세요. 로컬 로그인한 계정은 10장에서도 사용하므로 기억하고 있어야 합니다. 카카오 로그인 외에 구글(passport-google-oauth2), 페이스북(passport-facebook), 네이버(passport-naver), 트위터(passport-twitter) 로그인도 가능합니다. 따라서 npm에서 찾아 사용하면 됩니다.

▼ 그림 9-21 로그인 후 화면



Note 三 카카오 로그아웃

로그아웃 버튼을 누르면 NodeBird 서비스에서 로그아웃됩니다. 다만, 카카오에서 로그아웃되는 것은 아니므로 다음 번에 카카오를 통해 NodeBird에 로그인할 때는 카카오 로그인 없이 바로 서비스에 로그인됩니다. 카카오에서도 로그아웃하고 싶다면 공식 문서(<https://developers.kakao.com/docs/latest/ko/kakaologin/rest-api#logout>)를 참조하면 됩니다.

9.4 multer 패키지로 이미지 업로드 구현하기

N O D E . J S

SNS 서비스인 만큼 이미지 업로드도 중요합니다. 6.2.7절에서 배운 multer 모듈을 사용해 멀티파트 형식의 이미지를 업로드합니다.

패키지를 먼저 설치합니다.

콘솔

```
$ npm i multer
```

이미지를 어떻게 저장할 것인지는 서비스의 특성에 따라 달라집니다. NodeBird 서비스는 `input` 태그를 통해 이미지를 선택할 때 바로 업로드를 진행하고, 업로드된 사진 주소를 다시 클라이언트에 알릴 것입니다. 게시글을 저장할 때는 데이터베이스에 직접 이미지 데이터를 넣는 대신 이미지 경로만 저장합니다. 이미지는 서버 디스크(uploads 폴더)에 저장됩니다.

그럼 `post` 라우터와 컨트롤러를 작성해보겠습니다.

```

const express = require('express');
const multer = require('multer');
const path = require('path');
const fs = require('fs');

const { afterUploadImage, uploadPost } = require('../controllers/post');
const { isLoggedIn } = require('../middlewares');

const router = express.Router();

try {
  fs.readdirSync('uploads');
} catch (error) {
  console.error('uploads 폴더가 없어 uploads 폴더를 생성합니다.');
  fs.mkdirSync('uploads');
}

const upload = multer({
  storage: multer.diskStorage({
    destination(req, file, cb) {
      cb(null, 'uploads/');
    },
    filename(req, file, cb) {
      const ext = path.extname(file.originalname);
      cb(null, path.basename(file.originalname, ext) + Date.now() + ext);
    },
  }),
  limits: { fileSize: 5 * 1024 * 1024 },
});

// POST /post/img
router.post('/img', isLoggedIn, upload.single('img'), afterUploadImage);

// POST /post
const upload2 = multer();
router.post('/', isLoggedIn, upload2.none(), uploadPost);

module.exports = router;

```

controllers/post.js

```
const { Post, Hashtag } = require('../models');

exports.afterUploadImage = (req, res) => {
    console.log(req.file);
    res.json({ url: `/img/${req.file.filename}` });
};

exports.uploadPost = async (req, res, next) => {
    try {
        const post = await Post.create({
            content: req.body.content,
            img: req.body.url,
            UserId: req.user.id,
        });
        const hashtags = req.body.content.match(/#[^\s#]*\w/g);
        if (hashtags) {
            const result = await Promise.all(
                hashtags.map(tag => {
                    return Hashtag.findOrCreate({
                        where: { title: tag.slice(1).toLowerCase() },
                    })
                }),
            );
            await post.addHashtags(result.map(r => r[0]));
        }
        res.redirect('/');
    } catch (error) {
        console.error(error);
        next(error);
    }
};
```

multer 부분은 6.2.7절의 코드와 거의 유사합니다. POST /post/img 라우터와 POST /post 라우터 두 개를 만듭니다. app.use('/post')를 할 것이므로 앞에 /post 경로가 붙었습니다.

POST /post/img 라우터에서는 이미지 하나를 업로드받은 뒤 이미지의 저장 경로를 클라이언트로 응답합니다. static 미들웨어가 /img 경로의 정적 파일을 제공하므로 클라이언트에서 업로드한 이미지에 접근할 수 있습니다.

POST /post 라우터는 게시글 업로드를 처리하는 라우터입니다. 이전 라우터에서 이미지를 업로드 했다면 이미지 주소도 req.body.url로 전송됩니다. 데이터 형식이 multipart이긴 하지만, 이미지

데이터가 들어 있지 않으므로 `none` 메서드를 사용했습니다. 이미지 주소가 온 것이지, 이미지 데이터 자체가 온 것은 아닙니다. 이미지는 이미 POST /post/img 라우터에서 저장되었습니다.

게시글을 데이터베이스에 저장한 후, 게시글 내용에서 해시태그를 정규표현식(`/#[^\s#]+/g`)으로 추출해냅니다. 추출한 해시태그는 데이터베이스에 저장하는데, 먼저 `slice(1).toLowerCase()`를 사용해 해시태그에서 #을 빼고 소문자로 바꿉니다. 저장할 때는 `findOneOrCreate` 메서드를 사용했습니다. 이 시퀄라이즈 메서드는 데이터베이스에 해시태그가 존재하면 가져오고, 존재하지 않으면 생성한 후 가져옵니다. 결괏값으로 [모델, 생성 여부]를 반환하므로 `result.map(r => r[0])`으로 모델만 추출해냅니다. 마지막으로, 해시태그 모델들을 `post.addHashtags` 메서드로 게시글과 연결합니다.

그림 9-22 해시태그와 게시글 연결

`['#노드', '#익스프레스', '#제로초']`

`hashtags.map`

`[findOneOrCreate, findOneOrCreate, findOneOrCreate]`

`Promise.all`로 모두 실행

`[[모델, bool], [모델, bool], [모델, bool]]`

`result.map(r => r[0])`

`[모델, 모델, 모델]`

`post.addHashtags()`

Note ≡ 실제 서버 운영 시

현재 `multer` 패키지는 이미지를 서버 디스크에 저장합니다. 디스크에 저장하면 간단하기는 하지만, 서버에 문제가 생겼을 때 이미지가 제공되지 않거나 손실될 수도 있습니다. 따라서 AWS S3나 클라우드 스토리지(Cloud Storage) 같은 정적 파일 제공 서비스를 사용해 이미지를 따로 저장하고 제공하는 것이 좋습니다. 이러한 서비스를 사용하고 싶다면 `multer-s3`나 `multer-google-storage` 같은 패키지를 찾아보면 됩니다. 이에 대해서는 16장에서 알아봅니다.

게시글 작성 기능이 추가되었으므로 이제부터 메인 페이지 로딩 시 메인 페이지와 게시글을 함께 로딩하도록 하겠습니다.

controllers/page.js

```
const { User, Post } = require('../models');

exports.renderProfile = (req, res) => {
    res.render('profile', { title: '내 정보 - NodeBird' });
};

exports.renderJoin = (req, res) => {
    res.render('join', { title: '회원 가입 - NodeBird' });
};

exports.renderMain = async (req, res, next) => {
    try {
        const posts = await Post.findAll({
            include: {
                model: User,
                attributes: ['id', 'nick'],
            },
            order: [['createdAt', 'DESC']],
        });
        res.render('main', {
            title: 'NodeBird',
            twits: posts,
        });
    } catch (err) {
        console.error(err);
        next(err);
    }
}
```

먼저 데이터베이스에서 게시글을 조회한 뒤 결과를 twits에 넣어 렌더링합니다. 조회할 때 게시글 작성자의 아이디와 닉네임을 JOIN해서 제공하고, 게시글의 순서는 최신순으로 정렬했습니다. 지금까지 이미지 업로드 기능을 만들었습니다. 그럼 남은 기능들을 마저 추가하고 서버를 실행해봅시다.

9.5

프로젝트 마무리하기

이미지 업로드까지 마무리되었으니 이제 팔로잉 기능과 해시태그 검색 기능만 추가하면 됩니다.

다른 사용자를 팔로우하는 기능을 만들기 위해 routes/user.js와 controllers/user.js를 작성합니다.

routes/user.js

```
const express = require('express');

const { isLoggedIn } = require('../middlewares');
const { follow } = require('../controllers/user');

const router = express.Router();

// POST /user/:id/follow
router.post('/:id/follow', isLoggedIn, follow);

module.exports = router;
```

controllers/user.js

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

exports.follow = async (req, res, next) => {
  try {
    const user = await User.findOne({ where: { id: req.user.id } });
    if (user) { // req.user.id가 followerId, req.params.id가 followingId
      await user.addFollowing(parseInt(req.params.id, 10));
      res.send('success');
    } else {
      res.status(404).send('no user');
    }
  } catch (error) {
    console.error(error);
    next(error);
  }
};
```

POST /user/:id/follow 라우터입니다. :id 부분이 req.params.id가 됩니다. 먼저 팔로우할 사용자를 데이터베이스에서 조회한 후, 시퀄라이즈에서 추가한 addFollowing 메서드로 현재 로그인한 사용자와의 관계를 지정합니다.

팔로잉 관계가 생겼으므로 req.user에도 팔로워와 팔로잉 목록을 저장합니다. 앞으로 사용자 정보를 불러올 때는 팔로워와 팔로잉 목록도 같이 불러오게 됩니다. req.user를 바꾸려면 deserializeUser를 조작해야 합니다.

passport/index.js

```
...
  passport.deserializeUser((id, done) => {
    User.findOne({
      where: { id },
      include: [
        {
          model: User,
          attributes: ['id', 'nick'],
          as: 'Followers',
        },
        {
          model: User,
          attributes: ['id', 'nick'],
          as: 'Followings',
        },
      ],
    })
      .then(user => done(null, user))
      .catch(err => done(err));
  });
...
}
```

세션에 저장된 아이디로 사용자 정보를 조회할 때 팔로잉 목록과 팔로워 목록도 같이 조회합니다. include에서 계속 attributes를 지정하고 있는데, 이는 실수로 비밀번호를 조회하는 것을 방지하기 위해서입니다. 브라우저에 회원의 비밀번호가 전송돼서는 안 됩니다.

Note ≡ deserializeUser 캐싱하기

라우터가 실행되기 전에 deserializeUser가 먼저 실행됩니다. 따라서 모든 요청이 들어올 때마다 매번 사용자 정보를 조회하게 됩니다. 서비스의 규모가 커질수록 더 많은 요청이 들어오게 되고, 그에 따라 데이터베이스에도 더 큰 부담이 주어집니다. 사용자 정보가 빈번하게 바뀌는 것이 아니라면 캐싱을 해두는 것이 좋습니다. 다만, 캐싱이 유지되는 동안 팔로워와 팔로잉 정보가 갱신되지 않는 단점이 있으므로 캐싱 시간은 서비스 정책에 따라 조절해야 합니다.

실제 서비스에서는 메모리에 캐싱하기보다는 레디스 같은 데이터베이스에 사용자 정보를 캐싱합니다.

팔로잉/팔로워 숫자와 팔로우 버튼을 표시하기 위해 routes/page.js를 수정합니다.

routes/page.js

```
...
router.use((req, res, next) => {
  res.locals.user = req.user;
  res.locals.followerCount = req.user?.Followers?.length || 0;
  res.locals.followingCount = req.user?.Followings?.length || 0;
  res.locals.followingIdList = req.user?.Followings?.map(f => f.id) || [];
  next();
});
...

```

로그인한 경우에는 req.user가 존재하므로 팔로잉/팔로워 수와 팔로워 아이디 리스트를 넣습니다. 팔로워 아이디 리스트를 넣는 이유는 팔로워 아이디 리스트에 게시글 작성자의 아이디가 존재하지 않으면 팔로우 버튼을 보여주기 위해서입니다.

routes/page.js

```
const express = require('express');
const { isLoggedIn, isNotLoggedIn } = require('../middlewares');
const {
  renderProfile, renderJoin, renderMain, renderHashtag,
} = require('../controllers/page');

const router = express.Router();
...
router.get('/hashtag', renderHashtag);

module.exports = router;
```

해시태그로 조회하는 GET /hashtag 라우터입니다. 쿼리스트링으로 해시태그 이름을 받고 해시태그 값이 없는 경우 메인 페이지로 돌려보냅니다. 데이터베이스에서 해당 해시태그가 존재하는지 검색한 후, 해시태그가 있다면 시퀄라이즈에서 제공하는 getPosts 메서드로 모든 게시글을 가져옵니다. 가져올 때는 include를 통해 작성자 정보를 합칩니다. 조회 후 메인 페이지를 렌더링하면서 전체 게시글 대신 조회된 게시글만 twits에 넣어 렌더링합니다.

controllers/page.js

```
const { User, Post, Hashtag } = require('../models');
...
exports.renderHashtag = async (req, res, next) => {
  const query = req.query.hashtag;
  if (!query) {
    return res.redirect('/');
  }
  try {
    const hashtag = await Hashtag.findOne({ where: { title: query } });
    let posts = [];
    if (hashtag) {
      posts = await hashtag.getPosts({ include: [{ model: User }] });
    }

    return res.render('main', {
      title: `${query} | NodeBird`,
      twits: posts,
    });
  } catch (error) {
    console.error(error);
    return next(error);
  }
};
```

마지막으로, routes/post.js와 routes/user.js를 app.js에 연결합니다. 업로드한 이미지를 제공할 라우터(/img)도 express.static 미들웨어로 uploads 폴더와 연결합니다. express.static을 여러 번 쓸 수 있다는 사실을 기억해주세요. 이제 uploads 폴더 내 사진들이 /img 주소로 제공됩니다.

app.js

```
...
const pageRouter = require('./routes/page');
const authRouter = require('./routes/auth');
const postRouter = require('./routes/post');
const userRouter = require('./routes/user');
const { sequelize } = require('../models');
const passportConfig = require('./passport');

...
app.use(morgan('dev'));
app.use(express.static(path.join(__dirname, 'public')));
```

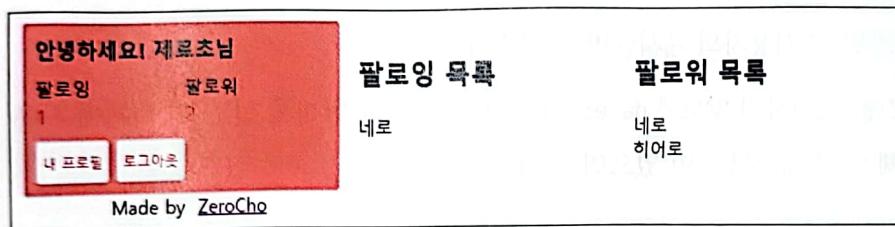
```

app.use('/img', express.static(path.join(__dirname, 'uploads')));
app.use(express.json());
...
app.use('/', pageRouter);
app.use('/auth', authRouter);
app.use('/post', postRouter);
app.use('/user', userRouter);
...

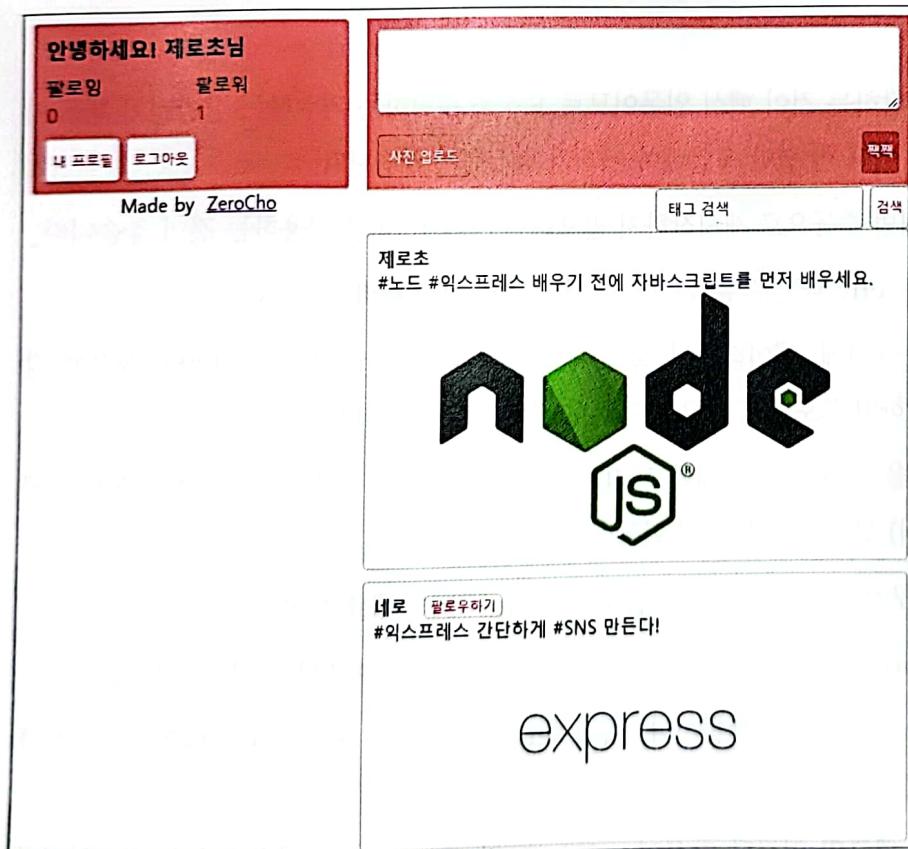
```

서버를 실행하고 NodeBird에 접속해 로그인, 게시글 작성, 팔로잉, 해시태그 검색 등의 기능을 사용해보세요.

▼ 그림 9-23 프로필 페이지 화면



▼ 그림 9-24 해시태그 검색 화면



9.5.1 스스로 해보기

여기서는 추가로 해볼 만한 작업을 몇 가지 소개하니 스스로 해보길 바랍니다. 지금은 기본 기능만 있지만, 살을 붙여나가다 보면 점점 완성도 있는 SNS 앱이 될 것입니다.

- 팔로잉 끊기(시퀄라이즈의 `destroy` 메서드와 라우터 활용)
- 프로필 정보 변경하기(시퀄라이즈의 `update` 메서드와 라우터 활용)
- 게시글 좋아요 누르기 및 좋아요 취소하기(사용자 - 게시글 모델 간 N:M 관계 정립 후 라우터 활용)
- 게시글 삭제하기(등록자와 현재 로그인한 사용자가 같을 때, 시퀄라이즈의 `destroy` 메서드와 라우터 활용)
- 사용자 이름을 누르면 그 사용자의 게시글만 보여주기
- 매번 데이터베이스를 조회하지 않도록 `deserializeUser` 캐싱하기(객체 선언 후 객체에 사용자 정보 저장, 객체 안에 캐싱된 값이 있으면 조회)

9.5.2 핵심 정리

- 서버는 요청에 응답하는 것이 핵심 임무이므로 요청을 수락하든 거절하든 상관없이 반드시 응답해야 합니다. 이때 한 번만 응답해야 에러가 발생하지 않습니다.
- 개발 시 서버를 매번 수동으로 재시작하지 않으려면 `nodemon`을 사용하는 것이 좋습니다.
- `dotenv` 패키지와 `.env` 파일로 유출되면 안 되는 비밀 키를 관리합니다.
- 라우터는 `routes` 폴더에, 데이터베이스는 `models` 폴더에, `html` 파일은 `views` 폴더에 각각 구분해서 저장하면 프로젝트 규모가 커져도 관리하기 쉽습니다.
- 라우터에서 응답을 보내는 미들웨어를 컨트롤러라고 합니다. 컨트롤러도 따로 분리하면 (`controllers` 폴더) 코드를 관리할 때 편합니다.
- 데이터베이스를 구성하기 전에 데이터 간 1:1, 1:N, N:M 관계를 잘 파악합니다.
- `middlewares/index.js`처럼 라우터 내에 미들웨어를 사용할 수 있다는 것을 기억합니다.
- `Passport`의 인증 과정을 기억해둡시다. 특히 `serializeUser`와 `deserializeUser`가 언제 호출되는지 파악하고 있어야 합니다.
- 프런트엔드 `form` 태그의 인코딩 방식이 `multipart`일 때는 `multer` 같은 `multipart` 처리용 패키지를 사용하는 것이 좋습니다.