5장

5장

5.1 프로젝트 구조 갖추기

1. SNS 서비스 만들기

- 핵심 기능:
  - 1. 로그인
  - 2. 게시글 작성 (이미지 업로드)
  - 3. 해시태그 검색
  - 4. 팔로잉

2. 프로젝트 시작하기

- 폴더를 만들고 package.json 파일 생성
  - 노드 프로젝트의 기본

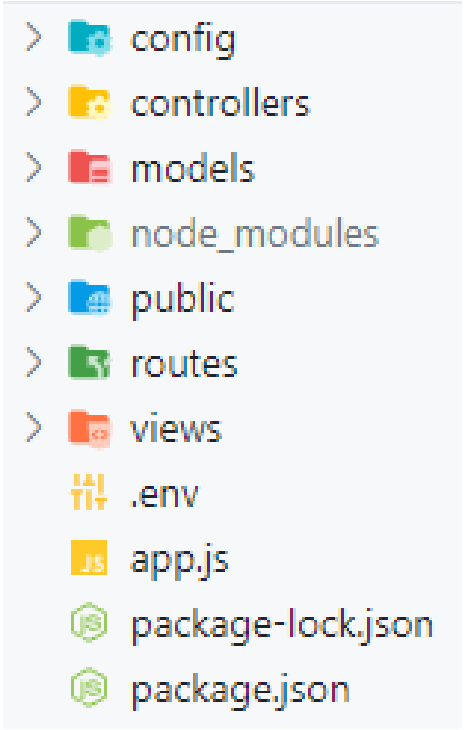
```
npm init
```

- 시퀄라이즈 폴더 구조 생성

```
npm i sequelize mysql2 sequelize-cli dotenv express cookie-parser express-session morgan multer nunjucks

npm i -D nodemon

npx sequelize init
```

3. 폴더 구조 설정

- 폴더 생성
  - views(템플릿 엔진), routes(라우터), public(정적 파일), passport(패스포트), controller(컨트롤러)
- 파일 생성
  - app.js & .env 파일 생성



4. app.js

app.js 이미지

```
require('dotenv').config();
const express = require('express');
const cookieParser = require('cookie-parser');
const morga = require('morgan');
const path = require('path');
const session = require('express-session');
const nunjucks = require('nunjucks');
const pageRouter = require('./routes/page');

const app = express();
app.set('port', process.env.PORT || 3001);
app.set('view engine', 'html');
nunjucks.configure('views', {
    express: app,
    watch: true,
});
```

```
app.use(
    morgan('dev'),
    express.static(path.join(__dirname, 'public')),
    express.json(),
    express.urlencoded({ extended: false }),
    cookieParser(process.env.COOKIE_SECRET),
    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(`404 ${req.method} ${req.url}`);
    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 포트 실행');
});
```

5. .env

- 포트 번호, 쿠키 서명, 개발 환경 등

```
PORT=3000
COOKIE_SECRET=cookiesecret
NODE_ENV=development
```

6. 라우터 생성

라우터 생성

- routes/page.js: 템플릿 엔진을 렌더링하는 라우터

```
const express = require('express');
const router = express.Router();
const {renderWrite, renderPost, renderJoin, renderMain, renderHashtag} = require('../controllers/page');

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

router.get('/write', renderWrite);
router.get('/post', renderPost);
router.get('/join', renderJoin);
router.get('/', renderMain);

module.exports = router;
```

7. 컨트롤러와 서비스

- 라우터 > 컨트롤러 > 서비스
- 컨트롤러는 라우터의 마지막에 위치하는 미들웨어로 요청을 받고, 응답을 보내는 역할
- 서비스는 핵심 로직을 담당하며, 요청과 응답에 대해 알지 못한다.

8. 컨트롤러 생성

- 컨트롤러 생성

```
exports.renderWrite = (req, res) => {
    res.render("write", { title: "글쓰기" });
};

exports.renderJoin = (req, res) => {
    res.render("join", { title: "회원가입" });
};

exports.renderJoin = (req, res) => {
    res.render('post', { title: '게시글' });
};

exports.renderMain = (req, res) => {
    res.render('main', { title: '메인페이지' });
};
```

9. 뷰 작성

- 뷰 작성
  - views/error.html: 에러 발생 시 에러가 표시될 화면
  - views/join.html: 회원가입 화면
  - views/layout.html: 프론트엔드 화면 레이아웃
  - views/main.html: 메인 화면
  - views/post.html: 게시글 화면
  - views/profile.html: 프로필 화면
  - views/write.html: 글쓰기 화면
  - public/index.css: 화면 CSS

10. 모델 생성

- User, Post, Hashtag 모델 생성

```
const Sequelize = require('sequelize');
class User extends Sequelize.Model {
    static initiate(sequelize) {

    }
    static associate(db) {

    }
}
module.exports = User;
```

```
const Sequelize = require('sequelize');
class Post extends Sequelize.Model {
    static initiate(sequelize) {

    }
    static associate(db) {

    }
}
module.exports = Post;
```

```
const Sequelize = require('sequelize');
class Hashtag extends Sequelize.Model {
    static initiate(sequelize) {

    }
    static associate(db) {

    }
}
module.exports = Hashtag;
```

5.2 데이터베이스 세팅하기

1. 모델 생성

모델 생성

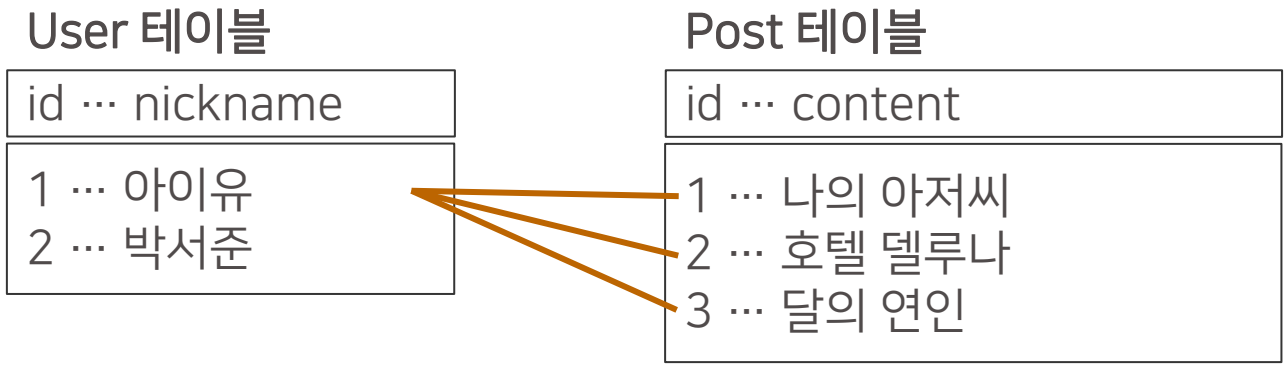
- models/user.js: 사용자 테이블과 연결됨
  - email : 일반 회원가입의 경우, 작성한 이메일 주소
  - kakaoId: 카카오톡 회원가입인 경우, 주어지는 값
  - provider: 카카오톡 회원가입인 경우, "kakao", 일반 회원가입인 경우 "local"
  - nickname : 사용자 별명
  - password : 사용자 암호 (암호화 예정)
- models/post.js: 게시글 내용과 이미지 경로를 저장(이미지는 파일로 저장)
  - content: : 게시글 내용
  - img : 게시 이미지 파일명 (다중파일 불가)
- models/hashtag.js: 해시태그 이름을 저장(나중에 태그로 검색하기 위해서)
  - name : 태그명

2. models/index.js

- 시퀄라이즈가 자동으로 생성해주는 코드를 변경
  - 모델들을 models 폴더에서 자동으로 불러옴(readdirSync)
  - 모델 간 관계가 있는 경우 관계 설정
  - User(1):Post(다)
  - Post(다):Hashtag(다)
  - User(다):User(다)

3. 사용자-게시물 일대다 관계

- User(1) : Post(다)
  - 일대다 관계 [사용자는 여러 개의 게시물을 작성할 수 있다.]




```
static associate(db) {
    db.User.hasMany(db.Post);
}
```

```
static associate(db) {
    db.Post.belongsTo(db.User);
}
```

4. 게시물-해시태그 다대다 관계

Post(다):HashTag(다)

- 다대다 관계를 맺어주면 중간 테이블(PostHashTag)이 자동으로 생성
- 시퀄라이즈는 모델 이름을 바탕으로 자동으로 addPost, getPosts, addUser, getUser 메서드가 생성됨

Post 테이블

id ··· content
1 ··· 나의 아저씨
2 ··· 호텔 델루나
3 ··· 달의 연인

PostHashtag 테이블

id	postid	hashtagid
1	1	1
2	1	2
3	1	3
4	2	1
5	2	2
6	2	4
7	3	1
8	3	4

Hashtag 테이블

id name
1 드라마
2 tvN
3 힐링
4 판타지

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

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

5. 팔로잉-팔로워 다대다 관계

- 팔로잉(다) : 팔로워(다)
  - 다대다 관계를 맺어주면 중간 테이블(Follow)이 자동으로 생성
  - 모델 이름이 같으므로 구분이 반드시 필요!
    - as가 구분자 역할
    - foreignKey는 반대 테이블 컬럼의 프라이머리 키 컬럼)
  - 시퀄라이즈는 as 이름을 바탕으로 자동으로 addFollower, getFollowers, addFollowing, getFollowings 메서드 생성

User 테이블 (follower)

id ··· nickname
1 ··· 아이유
2 ··· 박서준
3 ··· 유재석
4 ··· 박명수

Follow 테이블

id	followerid	followingid
1	1	2
2	2	1
3	4	3

User 테이블 (following)

id ··· nickname
1 ··· 아이유
2 ··· 박서준
3 ··· 유재석
4 ··· 박명수

```
static associate(db) {
    db.User.hasMany(db.Post);
    db.User.belongsToMany(db.User, { foreignKey: 'followingId', as: 'Follower', through: 'Follow' });
    db.User.belongsToMany(db.User, { foreignKey: 'followerId', as: 'Following', through: 'Follow' });
}
```

6. associate 작성하기

- 모델 간의 관계들 associate에 작성
  - 일대다: hasMany와 belongsTo
  - 다대다: belongsToMany
    - foreignKey: 외래키
    - as: 컬럼에 대한 별명
    - through: 중간 테이블명

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

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

static associate(db) {
    db.User.hasMany(db.Post);
    db.User.belongsToMany(db.User, { foreignKey: 'followingId', as: 'Follower', through: 'Follow' });
    db.User.belongsToMany(db.User, { foreignKey: 'followerId', as: 'Following', through: 'Follow' });
}
```

7. 모델과 서버 연결하기

- sequelize.sync()가 테이블 생성
  - false 옵션을 true로 하면 서버 실행 시마다 테이블 재생성
  - false로 하면 IF NOT EXIST(SQL문)으로 테이블이 없을 때만 생성

```
const { sequelize } = require('./models');
sequelize.sync({ force: false })
    .then(() => {
        console.log('데이터베이스 연결 성공');
    })
    .catch((err) => {
        console.error(err);
    });
```

8. 시퀄라이즈 설정하기

- 시퀄라이즈 설정은 config/config.json
  - 개발환경용 설정은 development 아래에

```
"development": {
    "username": "root",
    "password": "1234",
    "database": "데이터베이스명",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
```

- 설정 파일 작성 후 데이터베이스 생성

```
npx sequelize db:create
```

6. 모델과 서버 연결하기

- npm start로 서버 실행

```
npm start
```

5.3 Passport 모듈로 로그인

1. 패스포트 설치하기

- 로그인 과정을 쉽게 처리할 수 있게 도와주는 Passport 설치
  - 비밀번호 암호화를 위한 bcrypt도 같이 설치

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

2. 회원가입 기능 구현을 위한 라우터 및 컨트롤러 작성

- 라우터 및 컨트롤러 작성 routes/auth.js, controllers/auth.js 작성
  - bcrypt.hash로 비밀번호 암호화
  - hash의 두 번째 인수는 암호화 라운드 (라운드가 높을수록 안전하지만 오래 걸림)

```
const bcrypt = require("bcrypt");
const User = require("../models/user");
exports.join = async (req, res, next) => {
    const { email, nickname, password } = req.body;
    try {
        const exUser = await User.findOne({ where: { email } });
        if (exUser) {
            throw new Error("이미 가입된 이메일입니다.");
        }
        const hash = await bcrypt.hash(password, 10);
        await User.create({
            email,
            nickname,
            password: hash,
        });
        return res.redirect("/");
    } catch (error) {
        console.error(error);
        return next(error);
    }
};
```

```
const express = require('express');
const { join } = require('../controllers/auth');
const router = express.Router();

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

module.exports = router;
```

```
const authRouter = require('./routes/auth');

app.use('/auth', authRouter);
```

3. passport 적용

- passport를 app.js와 연결

```
const passport = require('passport');
const passportConfig = require('./passport');
passportConfig(); // 패스포트 설정
```

- passport.initialize(): 요청 객체에 passport 설정을 심음
- passport.session(): req.session 객체에 passport 정보를 저장
- express-session 미들웨어에 의존하므로 이보다 더 뒤에 위치해야 함

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

3. passport 적용

- passport/index.js 파일 작성

```
module.exports = () => {

}
```

4. 로컬 로그인 구현

- 라우터 및 컨트롤러에 로그인 동작 추가
  - routes/auth.js, controllers/auth.js 작성

  - passport.authenticate('local') 함수로 로그인 전략이 수행됨
  - 전략을 수행 후, authenticate의 콜백 함수가 호출됨
  - err : 인증 과정 중 에러
  - user : 인증에 성공한 유저 정보
  - info : 인증 오류 메시지

```
const express = require('express');
const { join, login } = require('../controllers/auth');
const router = express.Router();

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

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

module.exports = router;
```

```
exports.login = (req, res, next) => {
    passport.authenticate('local', (err, user, info) => {
        console.log(err, user, info);
    })(req, res, next);
}
```

5. 로컬 로그인 전략 구현

passport/index.js 작성

```
const local = require('./local');
module.exports = () => {
    local();
}
```

passport/local.js 작성

- usernameField와 passwordField가 input 태그의 email, password (body-parser의 req.body)

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

module.exports = () => {
    passport.use(new LocalStrategy({
        usernameField: 'email',
        passwordField: 'password',
        passReqToCallback: false,
    }, async (email, password, done) => {
        console.log(email, password);
    }));
};
```

6. 로컬 로그인 전략 구현

- 실제 전략을 수행하는 done 함수를 작성
  - 사용자가 DB에 저장되어 있는지 확인한 후, 있다면 비밀번호 비교 (bcrypt.compare)
  - 비밀번호까지 일치한다면 로그인

```
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);
        }
    }));
};
```

7. 컨트롤러에서 로그인 응답 처리

- 로그인 전략을 통해 인증이 성공했다면 req.login에 의해 세션에 유저 정보를 저장한다.

```
exports.login = (req, res, next) => {
    passport.authenticate('local', (err, user, info) => {
        console.log("authenticate 완료", err, user, info);
        if (err) {
            console.error(err);
            return next(err);
        }
        if (!user) {
            throw new Error(info.message);
        }
        return req.login(user, (err) => {
            console.log('login 실행', err);
            if (err) {
                console.error(err);
                return next(err);
            }
            return res.redirect('/');
        });
    })(req, res, next);
};
```

8. serializeUser, deserializeUser

- passport의 serializeUser : 세션 객체(req.session)에 어떤 데이터를 저장할 지 선택
  - 모든 사용자 정보를 다 가지고 있는 것은 비효율적. 따라서 사용자의 아이디만 저장

- passport의 deserializeUser : 세션 객체(req.session)에 저장된 사용자 아이디를 이용
  - 사용자 정보를 얻어낸 후 req.user에 저장

로그인 사용자 정보

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

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

사용자 아이디

req.user에 저장

9. passport의 처리 과정

로그인 과정

- 1. 로그인 요청이 들어옴
- 2. passport.authenticate 메서드 호출
- 3. 로그인 전략 수행
- 4. 로그인 성공 시 사용자 정보 객체와 함께 req.login 호출
- 5. req.login 메서드가 passport.serializeUser 호출
- 6. req.session에 사용자 아이디만 저장
- 7. 로그인 완료

로그인 이후 과정

- 1. 모든 요청에 passport.session() 미들웨어가 passport.deserializeUser 메서드 호출
- 2. req.session에 저장된 아이디로 데이터베이스에서 사용자 조회
- 3. 조회된 사용자 정보를 req.user에 저장
- 4. 라우터에서 req.user 객체 사용 가능

10. 사용자 로그인 여부 체크하는 미들웨어 추가

- 라우터 중간에 로그인 여부를 체크하는 미들웨어를 추가 (middlewares/index.js)

```
exports.isLogIn = (req, res, next) => {
    if (req.isAuthenticated()) {
        next();
    } else {
        res.status(403).send("로그인 필요");
    }
};
exports.isNotLogIn = (req, res, next) => {
    if (!req.isAuthenticated()) {
        next();
    } else {
        throw new Error("로그인한 상태입니다.");
    }
};
```

- routes/page.js 에 적용
  - req.user 값 사용

```
const { isLogIn, isNotLogIn } = require('../middlewares');

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

router.get('/write', isLogIn, renderWrite);
router.get('/post', isLogIn, renderPost);
router.get('/join', isNotLogIn, renderJoin);
router.get('/', renderMain);
```

11. 로그아웃 기능 구현

- routes/auth.js에 미들웨어 적용 및 로그아웃 라우터 추가

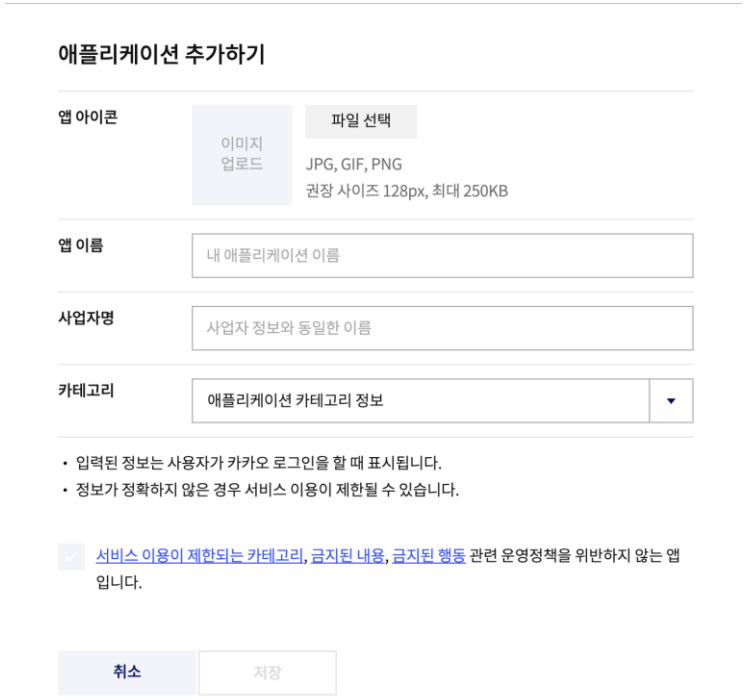
```
const express = require('express');
const { join, login, logout } = require('../ controllers / auth');
const { isLogIn, isNotLogIn } = require('../middlewares');
const router = express.Router();

// POST /auth/join
router.post('/join', isNotLogIn, join);
// POST /auth/login
router.post('/login', isNotLogIn, login);
// GET /auth/logout
router.get('/logout', isLogIn, logout);

module.exports = router;
```

- 컨트롤러에 logout 기능 추가

```
exports.logout = (req, res) => {
    console.log('로그아웃', req);
    req.logout(() => {
        req.session.destroy();
        res.redirect('/');
    });
};
```

12. 카카오 로그인 앱 만들기

- https://developers.kakao.com 에 접속



- .env 파일에 REST API 키 저장 (.gitignore)

13. 카카오 웹 플랫폼 추가 및 Redirect URI 등록

웹 플랫폼을 추가해야 Redirect URI 등록할 수 있음

Web 플랫폼 등록

사이트 도메인

JavaScript SDK, 카카오톡 공유, 카카오맵, 메시지 API 사용시 등록이 필요합니다.
여러개의 도메인은 줄바꿈으로 추가해주세요. 최대 9개까지 등록 가능합니다. 추가 등록은 포럼(데브톡)으로 문의주세요.
예시: (O) https://example.com (X) https://www.example.com

http://localhost:3001

기본 도메인

기본 도메인은 첫 번째 사이트 도메인으로, 카카오톡 공유와 카카오톡 메시지 API를 통해 발송되는 메시지의 Web 링크 기본값으로 사용됩니다.

http://localhost:3001

취소 저장

Redirect URI

Redirect URI

카카오 로그인에서 사용할 OAuth Redirect URI를 설정합니다.
여러개의 URI를 줄바꿈으로 추가해주세요. (최대 10개)
REST API로 개발하는 경우 필수로 설정해야 합니다.
예시: (O) https://example.com/oauth (X) https://www.example.com/oauth

http://localhost:3001/auth/kakao/login

취소 저장

14. 카카오 동의항목 설정

닉네임 정보를 얻기 위해 동의항목 설정

15. 카카오 로그인용 라우터 만들기

- 회원가입과 로그인이 전략에서 동시에 수행됨
  - passport.authenticate('kakao')만 하면 됨
  - /kakao/callback 라우터에서는 인증 성공 시(res.redirect)와 실패 시(failureRedirect) 리다이렉트할 경로 지정

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

// GET /auth/kakao/callback
router.get('/kakao/login', passport.authenticate('kakao', {
    failureRedirect: '/',
}), (req, res) => {
    res.redirect('/');
});

module.exports = router;
```

16. 카카오 로그인 전략 구현

passport/kakao.js 작성

- clientID에 카카오 앱 아이디 추가
- callbackURL: 카카오 로그인 후 카카오가 결과를 전송해줄 URL
- accessToken, refreshToken: 로그인 성공 후 카카오가 보내준 토큰
- profile: 카카오가 보내준 유저 정보
- profile의 정보를 바탕으로 회원가입

```
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/login',
    }, async (accessToken, refreshToken, profile, done) => {
        try {
            const exUser = await User.findOne({
                where: {
                    kakaoId: profile.id,
                    provider: 'kakao'
                }
            });
            if (exUser) {
                done(null, exUser);
            } else {
                const newUser = await User.create({
                    nickname: profile.displayName,
                    kakaoId: profile.id,
                    provider: 'kakao',
                });
                done(null, newUser);
            }
        } catch (error) {
            console.error(error);
            done(error);
        }
    }));
};
```

5.4 Multer 모듈로 이미지 업로드 구현하기

1. 이미지 업로드 구현

- form 태그의 enctype이 multipart/form-data
  - body-parser로는 요청 본문을 해석할 수 없음
  - multer 패키지 필요

  - 이미지를 먼저 업로드하고, 이미지가 저장된 경로를 반환
  - 게시글 form을 submit할 때는 이미지 자체 대신 경로를 전송

2. 이미지 업로드 라우터 구현

- fs.readdir, fs.mkdirSync로 upload 폴더가 없으면 생성
- multer() 함수로 업로드 미들웨어 생성
  - storage: diskStorage는 이미지를 서버 디스크에 저장 (destination은 저장 경로, filename은 저장 파일명)
  - limits는 파일 최대 용량(5MB)
  - imgUpload.single('img'): 요청 본문의 img에 담긴 이미지 하나를 읽어 설정대로 저장하는 미들웨어
  - 저장된 파일에 대한 정보는 req.file 객체에 담김

```
const express = require("express");
const multer = require("multer");
const path = require("path");
const fs = require("fs");
const { afterUploadImage, uploadPost } = require("../controllers/post");
const { isLogIn } = require("../middlewares");
const router = express.Router();

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

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

// POST /post/img
router.post("/img", isLogIn, imgUpload.single("img"), afterUploadImage);

// POST /post
const noImg = multer();
router.post("/", isLogIn, noImg.none(), uploadPost);

module.exports = router;
```

3. 게시글 등록

- noimg.none()은 multipart/form-data 타입의 요청이지만 이미지는 없을 때 사용
  - 게시글 등록 시 아까 받은 이미지 경로 저장
  - 게시글에서 해시태그를 찾아서 게시글과 연결(post.addHashtags)
  - findOrCreate는 기존에 해시태그가 존재하면 그걸 사용하고, 없다면 생성하는 시퀄라이즈 메서드

['#일상', '#데일리', '#맛집']

↓ hashtags.map

[findOrCreate, findOrCreate, findOrCreate]

↓ Promise.all

[[모델, bool], [모델, bool], [모델, bool]]

↓ result.map(r => r[0]

[모델, 모델, 모델]

↓

post.addHashtag( )

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

exports.afterUploadImage = (req, res) => {
    console.log(req.file);
    res.json({ url: `/uploads/${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#]*/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);
    }
};
```

4. 메인 페이지에 게시글 보여주기

- 메인 페이지(/) 요청 시 게시글을 먼저 조회한 후 템플릿 엔진 렌더링
  - include로 관계가 있는 모델을 합쳐서 가져올 수 있음
  - Post와 User는 관계가 있음 (1대다)
  - 게시글을 가져올 때 게시글 작성자까지 같이 가져온다.

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

exports.renderJoin = (req, res) => { res.render("join", { title: "회원가입" }); };

exports.renderWrite = (req, res) => { res.render("write", { title: "글쓰기" }); };

exports.renderPost = async (req, res) => {
    try {
        const posts = await Post.findAll({
            include: {
                model: User,
                attributes: ["id", "nickname"],
            },
            order: [["createdAt", "DESC"]],
        });
        res.render("post", { title: "게시글", posts });
    } catch (err) {
        console.error(err);
        next(err);
    }
};

exports.renderMain = (req, res) => {
    try {
        if (req.user) {
            res.render("profile", { title: "메인페이지" });
            return;
        } else {
            res.render("main", { title: "메인페이지" });
        }
    } catch (err) {
        console.error(err);
        next(err);
    }
};
```

5.5 프로젝트 마무리하기

1. 팔로잉 기능 구현

- POST /follow/:id 라우터 추가
  - /follow/:id
  - 사용자 아이디는 req.params.id로 접근
  - user.addFollowing(사용자아이디)로 팔로잉하는 사람 추가

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

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

const router = express.Router();

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

module.exports = router;
```

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

exports.follow = async (req, res, next) => {
    try {
        const user = await User.findOne({ where: { id: req.user.id } });
        if (user) {
            await user.addFollowing(parseInt(req.params.id));
            res.send("success");
        } else {
            res.status(404).send("없는 사용자");
        }
    } catch (error) {
        console.error(error);
        next(error);
    }
};
```

2. 팔로잉 기능 구현

deserializeUser

- req.user.Followers로 팔로워 접근 가능
- req.user.Followings로 팔로잉 접근
- 단, 목록이 유출되면 안 되므로 팔로워/팔로잉 숫자만 프런트로 전달

```
const express = require('express');
const router = express.Router();
const {renderWrite, renderPost, renderJoin, renderMain } = require('../controllers/page');
const { isLogIn, isNotLogIn } = require('../middlewares');

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();
});

router.get('/write', isLogIn, renderWrite);
router.get('/post', isLogIn, renderPost);
router.get('/join', isNotLogIn, renderJoin);
router.get('/', renderMain);

module.exports = router;
```

3. 해시태그 검색 기능 추가

GET /hashtag 라우터 추가

```
const router = express.Router();
const {renderWrite, renderPost, renderJoin, renderMain, renderHashtag} = require('../controllers/page');
const { isLogIn, isNotLogIn } = require('../middlewares');

...

router.get('/write', isLogIn, renderWrite);
router.get('/post', isLogIn, renderPost);
router.get('/join', isNotLogIn, renderJoin);
router.get('/', renderMain);
router.get('/hashtag', renderHashtag);

module.exports = router;
```

3. 해시태그 검색 기능 추가

- renderHashtag 컨트롤러 추가

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

...

exports.renderHashtag = async (req, res, next) => {
    const query = req.query.tag;
    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 }] });
        }
        console.log(posts);
        return res.render("post", {
            title: `${query} 검색 결과 | 게시글`,
            posts,
        });
    } catch (error) {
        console.error(error);
        return next(error);
    }
};
```