23. JWT로 회원인증하기

22.mongodb를 23.jwt로 복사후 진행할 것

- yarn add koa
- yarn add --dev eslint
- yarn run eslint --init
- yarn add eslint-config-prettier
- yarn add --dev nodemon
- yarn add koa-router
- yarn add koa-bodyparser
- yarn add mongoose@6.10.5 dotenv
- yarn add esm
- yarn add joi

23.1 JWT 이해하기

- JWT : JSON Web Token의 약자로 데이터가 JSON으로 구성된 Token을 의미

23.1.1 세션기반 인증과 토큰가반 인증의 차이

- 사용자 로그인상태를 서버에서 처리하는 방법은 `세션기반인증과 JWT Token기반인증` 이 있다.
- 세션기반인증
  - 서버가 사용자 로그인중임을 기억하고 있다는 뜻
  - 서버의 세션저장소에 사용자의 정보를 관리
  - 단점은 서버확장시 모든 서버에서 동일 세션을 관리해야 한다(세션정용 DB가 필요할 수도 있다.)
- 토큰기반인증
  - 토큰은 로그인 후 서버가 만들어 준 문자열
  - 해당 문자열안에 사용자 로그인정보와 서버발급증명하는 서명이 저장
  - 서명 데이터는 `해싱알고리즘(HMAC SHA256, RSA SHA256)`을 통해 만들어 진다.
  - 서버에서 만들어준 토큰은 무결성(위조되지 않았다는)이 보장
  - 사용자가 로그인후 서버에서 토큰을 발급하고 사용자의 요청시 토큰과 함께 요청
  - 서버는 해당 토큰 유효여부를 검증하고 응답 _ 장점은 사용자쪽에서 로그인상태 토큰을 저장하기 때문에 서버에 부담이 감소

23.2 User스키마/모델 만들기

src/models/user.js

```
import mongoose, { Schema } from 'mongoose';

const UserSchema = new Schema({
```

```
  username: String,
  hashedPassword: String,
});

const User = mongoose.model('User', UserSchema);

export default User;
```

- 해시생성라이브러리설치 : `yarn add bcrypt`

23.2.1 모델메서드 만들기

- 모델메서드는 모델에서 사용하는 함수로서 두 가지 종류가 있다.
  - 첫 번째는 인스턴스메서드로 모델을 통해 만든 문서 인스턴스에서 사용할 수 있는 함수를 의미

    ```
    const user = new User({username: 'gilbaek'});
    use.setPasswrd('12345');
    ```

  - 두 번째는 스태틱(static) 메서드로 보델에서 바로 사용할 수 잇는 함수를 의미

    ```
    const user = User.findByUsername('gilbaek');
    ```

23.1.1 인스턴스 메서드 만들기

- setPassword() : 비밀번호를 전달받아 hashedPassword값을 설정
- checkPassword() : 전달 받은 비밀번호가 해당계정의 비밀번호와 일치 여부를 검증

src/models/user.js

- 화살표함수가 아닌 function을 사용하는 이유는 함수 내부에서 this에 접근해야 하기 때문이다.
  - 화살표함수는 this를 사용하지 못한다.

```
import mongoose, { Schema } from 'mongoose';
import bcrypt from 'bcrypt';

const UserSchema = new Schema({
  username: String,
  hashedPassword: String,
});

UserSchema.methods.setPassword = async function(password) {
  const hash = await bcrypt.hash(password, 10);
  this.hashedPassword = hash;
}

UserSchema.methods.checkPassword = async function(password) {
  const result = await bcrypt.compare(password, this.hashedPassword);
  return result;
}

const User = mongoose.model('User', UserSchema);

export default User;
```

23.2.2 스태틱메서드 만들기

- findByUsername메서드로 username으로 데이터를 검색

src/models/user.js

```
import mongoose, { Schema } from 'mongoose';
import bcrypt from 'bcrypyt';

const UserSchema = new Schema({
  username: String,
  hashedPassword: String,
});

UserSchema.methods.setPassword = async function(password) {
  const hash = await bcrypt.hash(password, 10);
  this.hashedPassword = hash;
}

UserSchema.methods.checkPassword = async function(password) {
  const result = await bcrypt.comare(password, this.hashedPassword);
  return result;
}

UserSchema.statics.findByUsername = function(username) {
  return this.findOne({ username });
}

const User = mongoose.model('User', UserSchema);

export default User;
```

23.3 회원인증 API 만들기

src/api/auth/auth.ctrl.js

```
export const register = async ctx => {
  // 회원가입
}

export const login = async ctx => {
  // 로글인
}

export const check = async ctx => {
  // 로그인상태확인
}

export const logout = async ctx => {
  // 로그아웃
}
```

src/api/auth/index.js

```
import Router from 'koa-router';
import * as authCtrl from './auth.ctrl';
```

```
const auth = new Router();

auth.post('/register', authCtrl.register);
auth.post('/login', authCtrl.login);
auth.post('/check', authCtrl.check);
auth.post('/logout', authCtrl.logout);

export default auth;
```

src/api/index.js

```
import Router from 'koa-router';
import posts from './posts';
import auth from './auth';

const api = new Router();

api.use('/posts', posts.routes());
api.use('/auth', auth.routes());

// 라우터 내보내기
export default api;
```

23.3.1 회원가입구현하기

src/models/user.js - serialize

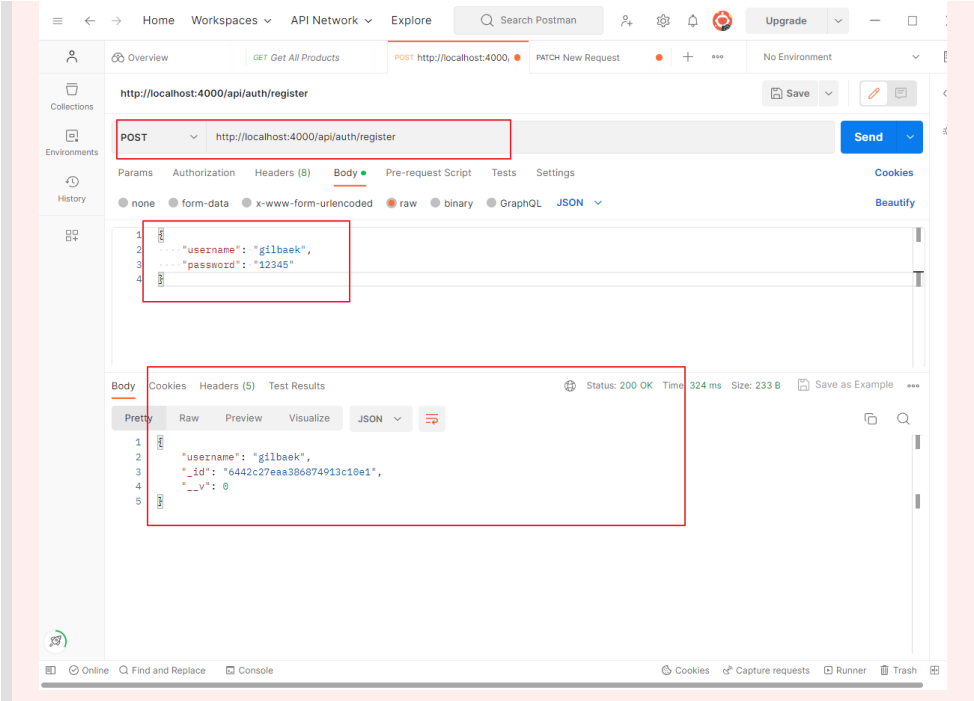
```
// 중략
UserSchema.methods.serialize = function() {
  const data = this.toJSON();
  delete data.hashedPassword;
  return data;
}
```

src/api/auth/auth.ctrl.js - register

```
import Joi from 'joi';
import User from '../../models/user';

/*
  POST /api/auth/register
  {
    username: 'gilbaek',
    password: '12345'
  }
*/
export const register = async ctx => {
  // request 검증하기
  const schema = Joi.object().keys({
    username: Joi.string().alphanum().min(3).max(20).required(),
    password: Joi.string().required()
  });
  const result = schema.validate(ctx.request.body);
  if(result.error) {
    ctx.status = 400;
    ctx.body = result.error;
    return;
  }
```

```
  const {username, password} = ctx.request.body;
  try {
    // username 존재확인
    const exists = await User.findByUsername(username);
    if(exists) {
      ctx.status = 400; // Conflict
      return;
    }

    const user = new User({
      username,
    });
    await user.setPassword(password); // 비밀번호설정
    await user.save(); // DB에 저장

    // 응답데이터에서 hashedPassword 필드 제거
    ctx.body = data.serialize();
  } catch (e) {
    ctx.throw(500, e);
  }
}
```

- POST http://localhost:4000/api/auth/register



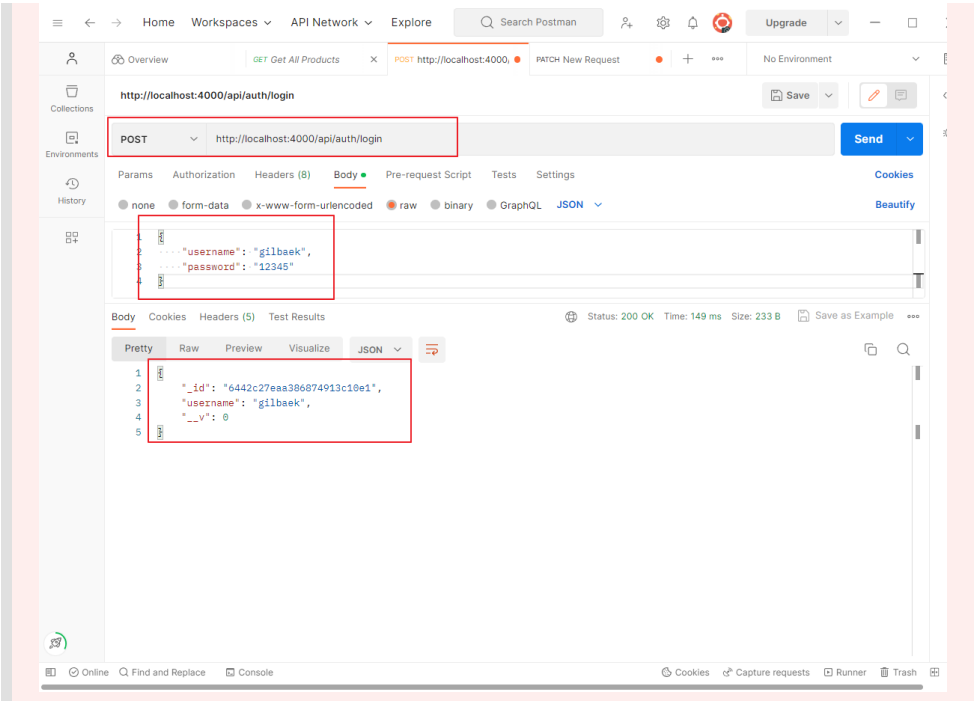
23.3.2 로그인 구현하기

src/api/auth/auth.ctrl.js - login

```
/*
  POST /api/auth/login
  {
    username: 'gilbaek',
    password: '12345'
  }
*/
export const login = async ctx => {
```

```
  const {username, password} = ctx.request.body;

  // username, password가 없은 에러
  if(!username || !password) {
    ctx.status = 401; // unauthorized
    return;
  }

  try {
    const user = await User.findByUsername(username);
    // 계정이 존재하지 않으면 에러
    if(!user) {
      ctx.status = 401;
      return;
    }

    const valid = await user.checkPassword(password);
    // 잘못된 비밀번호
    if(!valid) {
      ctx.status = 401;
      return;
    }
    ctx.body = user.serialize();

  } catch (e) {
    ctx.throw(500, e);
  }
}
```

- POST http://localhost:4000/api/auth/login

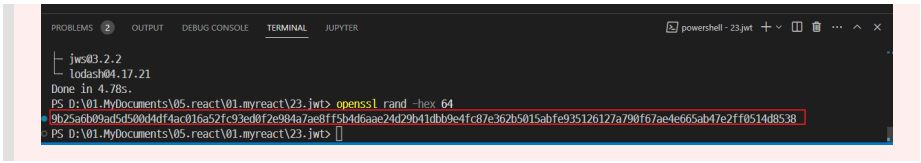


23.4 토큰 발급 및 검증하기

- JWT토큰을 만들기 우해 jsonwebtoken 설치 `yarn add jsonwebtoken`

23.4.1 비밀키 설정하기

- .env파일에 토큰생성시 사용할 비밀키를 설정(임의의 문자열을 설정)
  - windows에 openssl 설치
    - download & install : https://slproweb.com/products/Win32OpenSSL.html
      - 마지막 donattion 체크 모도두 해제
      - 참고 사이트 : https://aspdotnet.tistory.com/2653
    - cmd창 : `openssl rand -hex 64` 실행
      - 비밀키 복사:
      - linux/mac 에서 동일명령
  - 또는 임의의 문자열 입력
  - 토큰발급사이트 : http://jwt.io



.env

```
PORT=4000
MONGO_URI=mongodb://127.0.0.1:27017/blog
JWT_SECRET=9b25a6b09ad5d500d4df4ac016a52fc93ed0f2e984a7ae8ff5b4d6aae24d29b41dbb9e4fc
```

- 이 키는 JWT토큰 서명 생성과정에서 사용, 비밀키는 외부공개불가, 공개시 JWT토큰을 마음대로 사용가능

23.4.2 토큰 발급하기

src/models/user.js - generateToken

```
import mongoose, { Schema } from 'mongoose';
import bcrypt from 'bcrypt';
import jwt from 'jsonerbtoken';

const UserSchema = new Schema({
  username: String,
  hashedPassword: String,
});

UserSchema.methods.setPassword = async function(password) {
  const hash = await bcrypt.hash(password, 10);
  this.hashedPassword = hash;
}

UserSchema.methods.checkPassword = async function(password) {
  const result = await bcrypt.compare(password, this.hashedPassword);
  return result;
}

UserSchema.statics.findByUsername = function(username) {
  return this.findOne({ username });
}

UserSchema.methods.serialize = function() {
```

```
  const data = this.toJSON();
  delete data.hashedPassword;
  return data;
}

UserSchema.methods.generateToken = function() {
  const token = jwt.sign(
    // 첫번째 파라미터엔 토큰 안에 집어넣고 싶은 데이터를 넣습니다
    {
      _id: this.id,
      username: this.username,
    },
    process.env.JWT_SECRET, // 두번째 파라미터에는 JWT 암호를 넣습니다
    {
      expiresIn: '7d', // 7일동안 유효함
    },
  );
  return token;
};

const User = mongoose.model('User', UserSchema);

export default User;
```

- 회원가입과 로그인성공시 토큰 발급, 브라우저에서 토큰 사용시 2 가지 방법을 사용
  1. 브러우저의 localStorage or sessionStorage에 저장후 사용,
     - 사용 및 구현은 편리하나 쉽게 토큰탈취가 용이 이러한 공격을 XSS(Cross Site Sciiping)라고 한다
  2. 브러우저 쿠키에 저장후 사용
     - 쿠키에 저장해도 같은 문제가 발생할 수 있지만 `httpOnly 속성 활성화`하면 JS으로 쿠키조회불가
     - 그대신 CSRF(Cross Site Request Forery)공격에 취약가능성 상존
     - 쿠키는 서버요청시마나 무조건 토큰이 함께 저달되기 때문에 사용자 모르게 API요청을 하게 한다.
     - 단, CSRF는 `CSRF토큰사용 및 Rdferer검증등의 방식으로 해킹방지가능`
     - 반면 XSS는 보안장치가 있어도 다양한 취약점을 통해 공격당할 수 있다.
- 여기서는 토큰을 쿠키에 담아서 사용한다.

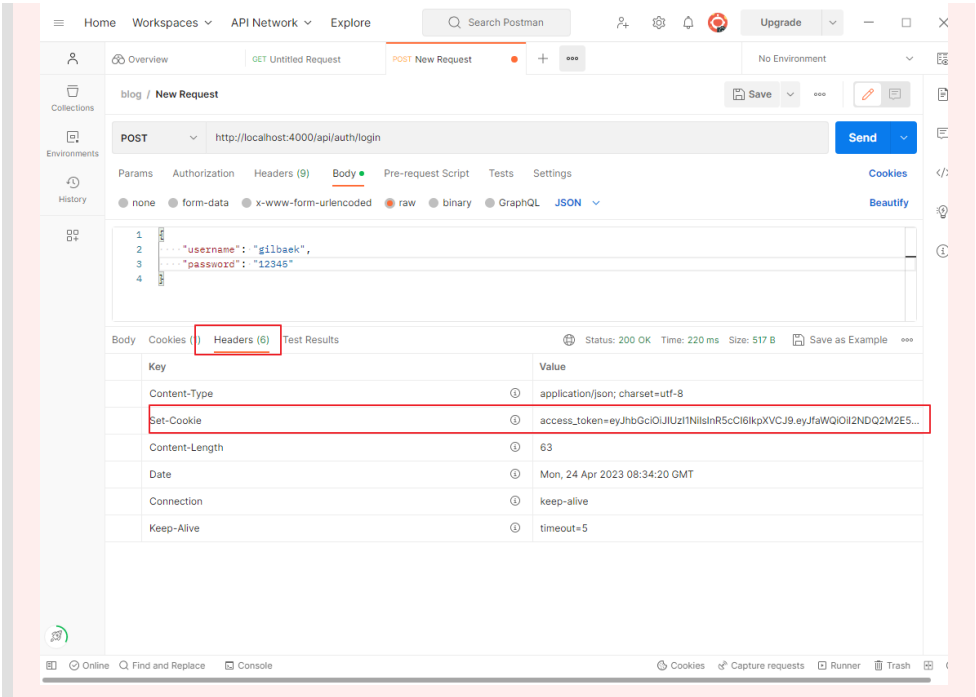
src/api/auth/auth.ctrl.js - register, login

```
import Joi from 'joi';
import User from '../../models/user';

/*
  POST /api/auth/register
  {
    username: 'gilbaek',
    password: '12345'
  }
*/
export const register = async ctx => {
  // request 검증하기
  const schema = Joi.object().keys({
```

```
    username: Joi.string().alphanum().min(3).max(20).required(),
    password: Joi.string().required()
  });
  const result = schema.validate(ctx.request.body);
  if(result.error) {
    ctx.status = 400;
    ctx.body = result.error;
    return;
  }

  const {username, password} = ctx.request.body;
  try {
    // username 존재확인
    const exists = await User.findByUsername(username);
    if(exists) {
      ctx.status = 400; // Conflict
      return;
    }

    const user = new User({
      username,
    });
    await user.setPassword(password); // 비밀번호설정
    await user.save(); // DB에 저장

    // 응답데이터에서 hashedPassword 필드 제거
    ctx.body = user.serialize();

    const token = user.generateToken();
    ctx.cookies.set('access_token', token, {
      maxAge: 1000 * 60 * 60 * 24 * 7,   // 7일
      httpOnli: true,
    });

  } catch (e) {
    ctx.throw(500, e);
  }
}

/*
  POST /api/auth/login
  {
    username: 'gilbaek',
    password: '12345'
  }
*/
export const login = async ctx => {
  const {username, password} = ctx.request.body;

  // username, password가 없은 에러
  if(!username || !password) {
    ctx.status = 401; // unauthorized
    return;
  }

  try {
    const user = await User.findByUsername(username);
```

```
    // 계정이 존재하지 않으면 에러
    if(!user) {
      ctx.status = 401;
      return;
    }

    const valid = await user.checkPassword(password);
    // 잘못된 비밀번호
    if(!valid) {
      ctx.status = 401;
      return;
    }
    ctx.body = user.serialize();

    const token = user.generateToken();
    ctx.cookies.set('access_token', token, {
      maxAge: 1000 * 60 * 60 * 24 * 7,    // 7일
      httpOnli: true,
    });

  } catch (e) {
    ctx.throw(500, e);
  }
}

export const check = async ctx => {
  // 로그인상태확인
}

export const logout = async ctx => {
  // 로그아웃
}
```

- POST http://localhost:4000/api/auth/login



22.4.3 토큰검증하기

src/lib/jwtMiddleware.js

```
import jwt from 'jsonwebtoken';

const jwtMiddleware = async (ctx, next) => {
  const token = ctx.cookies.get('access_token');
  if (!token) return next(); // 토큰이 없음
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    console.log(decoded);
    return next();
  } catch (e) {
    // 토큰 검증 실패
    return next();
  }
};

export default jwtMiddleware;
```

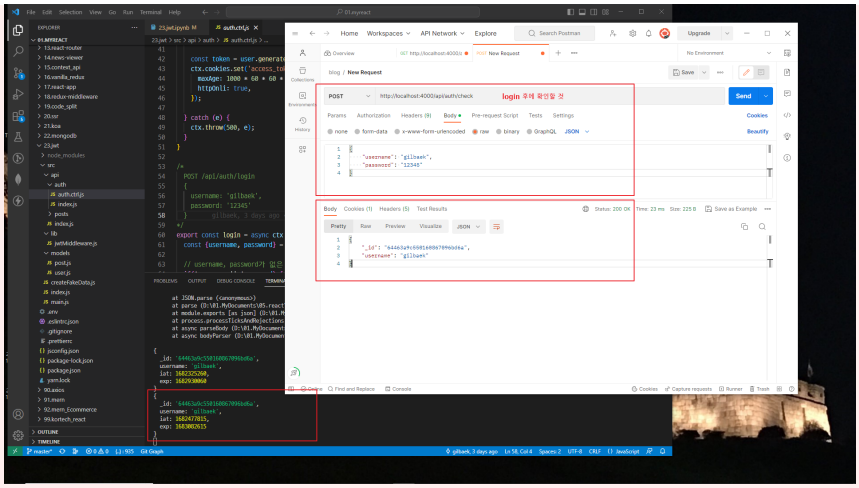
- jwtMiddleware를 적용하는 작업은 app에 router미들웨어 적용하기 전에 이루어 져야한다.
- 즉, main.js코드 상단에 작성되어야 한다.

src/main.js

```
require('dotenv').config();
import Koa from 'koa';
import Router from 'koa-router';
import bodyParser from 'koa-bodyparser';
import mongoose from 'mongoose';

import api from './api';
import jwtMiddleware from './lib/jwtMiddleware';

import createFakeData from './createFakeData';

// 비구조화 할당을 통해 process.env내부값에 대한 레퍼런스 만들기
const { PORT, MONGO_URI } = process.env;

mongoose
  .connect(MONGO_URI)
  .then(() => {
    console.log("Conntected to MongoDB");
    createFakeData();
  })
  .catch(e => {
    console.error(e);
})

const app = new Koa();
const router = new Router();

// 라우터설정
router.use('/api', api.routes()); // api 라우트 적용

// 라우터적용전에 bodyParser적용
app.use(bodyParser());
app.use(jwtMiddleware);
```

```
// app 인스턴스에 라우터 적용
app.use(router.routes()).use(router.allowedMethods());

// PORT가 지정되어 있지 않다면 4000사용
const port = PORT || 4000;

app.listen(port, () => {
  console.log('Listening to port %d', port)
});
```

- GET http://localhost:4000/api/auth/check
  - postman으로 요청을 하면 'Method Not Allowed' 메시지
  - 확인은 Terminal에서 아래화면처럼 검증확인할 것



- 검증한 결과를 `jwtMiddleware`에 작성

src/lib/jwtMiddleware.js

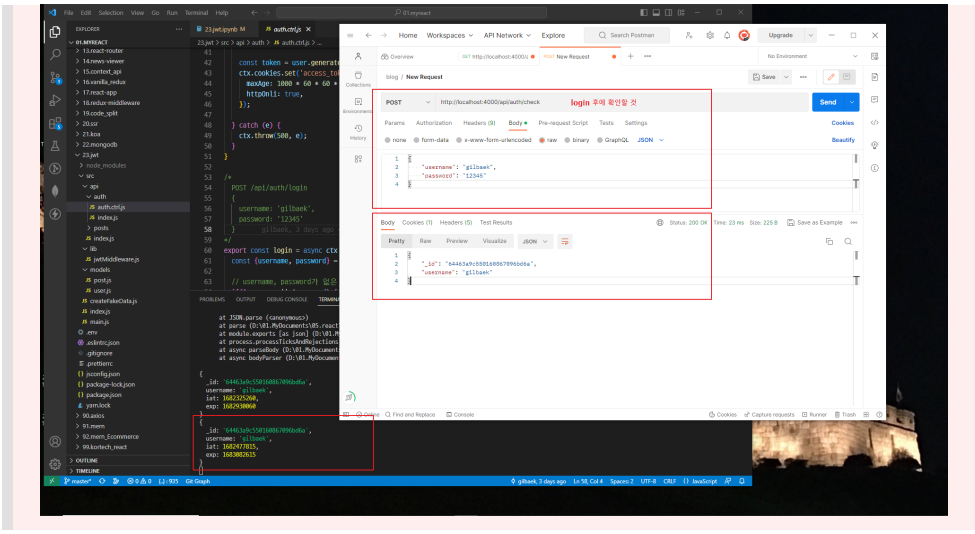
```
import jwt from 'jsonwebtoken';

const jwtMiddleware = async (ctx, next) => {
  const token = ctx.cookies.get('access_token');
  if (!token) return next(); // 토큰이 없음
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    ctx.state.user = {
      _id: decoded._id,
      username: decoded.username,
    };

    console.log(decoded);

    return next();
  } catch (e) {
    // 토큰 검증 실패
    return next();
  }
};

export default jwtMiddleware;
```

- 콘솔에 토큰정보를 출력하는 코드 이후 토큰이 만료전에 재발급해 주는 기능을 구현한 후 삭제

src/api/auth/auth.ctrl.js - check

```
/*
  GET /api/auth/check
*/
export const check = async (ctx) => {
  const { user } = ctx.state;
  if (!user) {
    // 로그인중 아님
    ctx.status = 401; // Unauthorized
    return;
  }
  ctx.body = user;
};
```

- GET http://localhost:4000/api/auth/check



22.4.4 토큰재발급하기

- iat : 토큰 생성일, exp : 토큰 만료일
- exp 만료일 3.5미만일 경우 토큰 재발급 기능을 구현

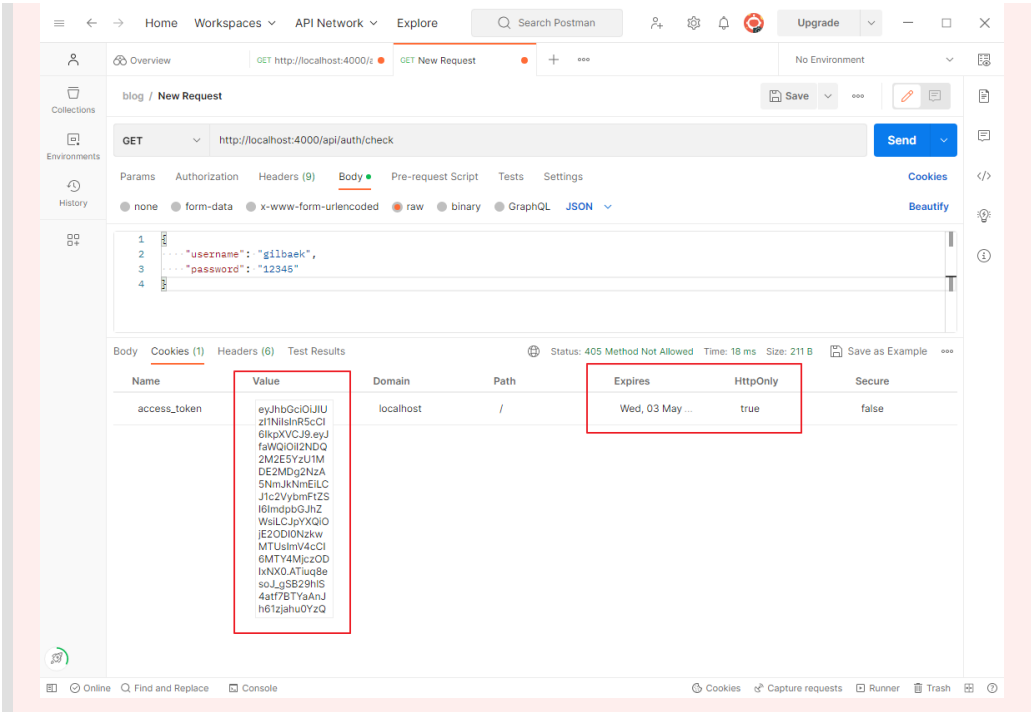
```
{
_id: '64463a9c550160867096bd6a',
username: 'gilbaek',
iat: 1682477815,
exp: 1683082615
}
```

src/lib/jwtMiddleware.js

```
import jwt from 'jsonwebtoken';

const jwtMiddleware = async (ctx, next) => {
  const token = ctx.cookies.get('access_token');
  if (!token) return next(); // 토큰이 없음
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
```

```
      ctx.state.user = {
        _id: decoded._id,
        username: decoded.username,
      };

      // 토큰 3.5일 미만 남으면 재발급
      const now = Math.floor(Date.now() / 1000);
      if (decoded.exp - now < 60 * 60 * 24 * 3.5) {
        const user = await User.findById(decoded._id);
        const token = user.generateToken();
        ctx.cookies.set('access_token', token, {
          maxAge: 1000 * 60 * 60 * 24 * 7, // 7일
          httpOnly: true,
        });
      }


      return next();
    } catch (e) {
      // 토큰 검증 실패
      return next();
    }
};

export default jwtMiddleware;
```

- 재발급 확인을 위해 user.js의 generateToken의 expireIn을 3d로 수정후 다시 login을 하여 check로 API요청
- 재발급확인 후에 7d 로 되돌리기



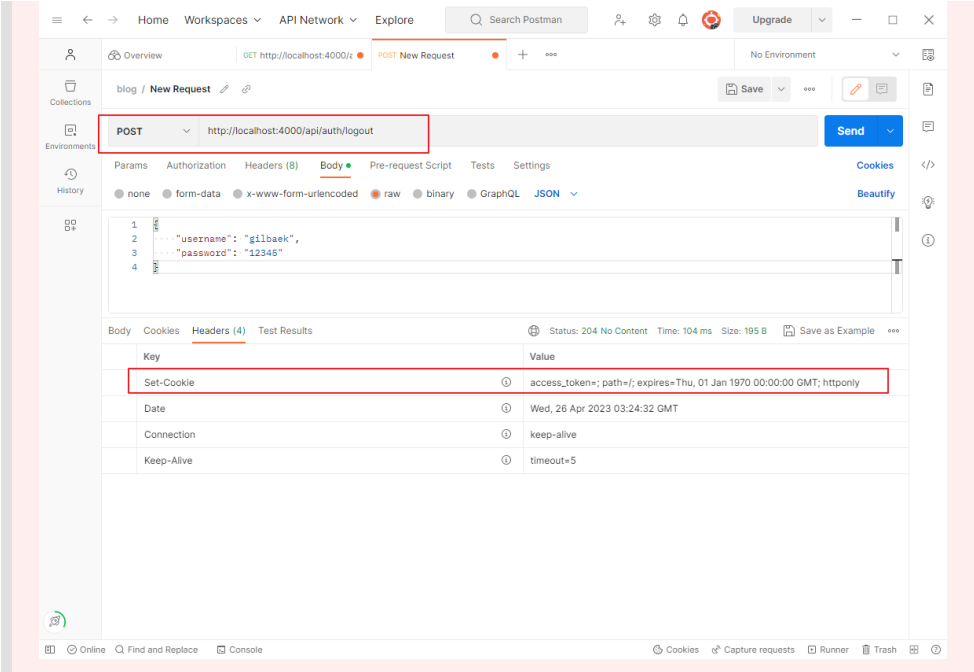
23.4.5 로그아웃 구현하기

src/api/auth/auth.ctrl.js - logout

```
/*
  POST /api/auth/logout
```

```
*/
export const logout = async (ctx) => {
  ctx.cookies.set('access_token');
  ctx.status = 204; // No Content
};
```

- POST http://localhost:4000/api/auth/logout
- access_token이 비워지는 Set-Cookie 헤더 확인



23.5 posts API에 회원인증 하기

- 새 포스트는 로그인을 해야 등록제한, 작성자만 수정, 삭제가능한 기능 구현

23.5.1 Post스키마 수정

src/models/post.js

```
import mongoose, { Schema } from 'mongoose';

const PostSchema = new Schema({
  title: String,
  body: String,
  tags: [String], // 문자열로 이루어진 배열
  publishedDate: {
    type: Date,
    default: Date.now, // 현재 날짜를 기본 값으로 지정
  },
  user: {
    _id: mongoose.Types.ObjectId,
    username: String,
  }
});

const Post = mongoose.model('Post', PostSchema);
```

```
export default Post;
```

23.5.2 posts 콜렉션 비우기

- posts콜렉션 삭제 - db.posts.drop();
  - or 자료삭제 : db.posts.remove({});

23.5.3 로그인했을 때만 API 사용하기

src/lib/checkLoggedIn.js

```
const checkLoggedIn = (ctx, next) => {
  if (!ctx.state.user) {
    ctx.status = 401; // Unauthorized
    return;
  }
  return next();
};

export default checkLoggedIn;
```

src/api/posts/index.js

```
import Router from 'koa-router';
import * as postsCtrl from './posts.ctrl';
import checkLoggedIn from '../../lib/checkLoggedIn'

const posts = new Router();

posts.get('/', postsCtrl.list);
posts.post('/', checkLoggedIn, postsCtrl.write);

const post = new Router(); // /api/posts/:id
post.get('/', postsCtrl.read);
post.delete('/', checkLoggedIn, postsCtrl.remove);
post.patch('/', checkLoggedIn, postsCtrl.update);

posts.use('/:id', postsCtrl.checkObjectId, post.routes());

export default posts;
```

23.5.4 포스트작성시 사용자 정보 넣기

src/api/posts.ctrl.js - write

- post등록할 때 사용자정보 추가하기

```
import Post from '../../models/post';
import mongoose from 'mongoose';
import Joi from 'joi';

const { ObjectId } = mongoose.Types;

export const checkObjectId = (ctx, next) => {
  const { id } = ctx.params;
  if (!ObjectId.isValid(id)) {
    ctx.status = 400; // Bad Request
    return;
```

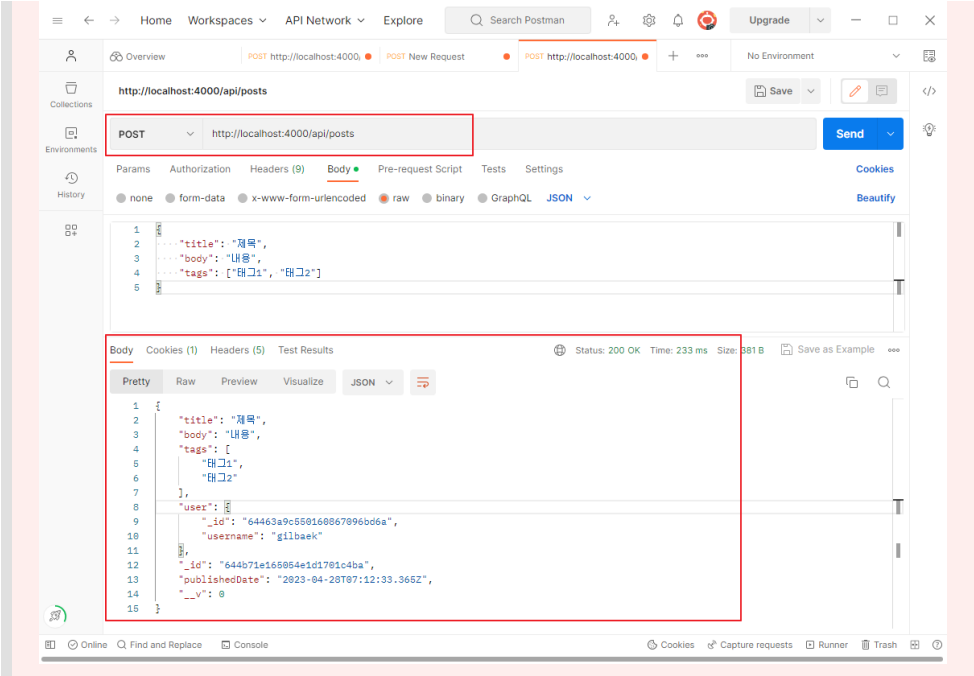
```
  }
  return next();
};

/*
  POST /api/posts
  {
    title: '제목',
    body: '내용',
    tags: ['태그1', '태그2']
  }
*/
export const write = async ctx => {
  const schema = Joi.object().keys({
    // 객체가 다음 필드를 가지고 있음을 검증
    title: Joi.string().required(), // required() 가 있으면 필수 항목
    body: Joi.string().required(),
    tags: Joi.array()
      .items(Joi.string())
      .required(), // 문자열로 이루어진 배열
  });

  // 검증 후, 검증 실패시 에러처리
  const result = schema.validate(ctx.request.body);
  if (result.error) {
    ctx.status = 400; // Bad Request
    ctx.body = result.error;
    return;
  }

  const { title, body, tags } = ctx.request.body;
  const post = new Post({
    title,
    body,
    tags,
    // ---------------------- 추가
    user: ctx.state.user,
    // ---------------------- 추가
  });
  try {
    await post.save();
    ctx.body = post;
  } catch (e) {
    ctx.throw(500, e);
  }
};

/*
  GET /api/posts
*/
export const list = async ctx => {
  // query 는 문자열이기 때문에 숫자로 변환해주어야합니다.
  // 값이 주어지지 않았다면 1 을 기본으로 사용합니다.
  const page = parseInt(ctx.query.page || '1', 10);

  if (page < 1) {
    ctx.status = 400;
```

```
    return;
  }

  try {
    const posts = await Post.find()
      .sort({ _id: -1 })
      .limit(10)
      .skip((page - 1) * 10)
      .lean()
      .exec();
    const postCount = await Post.countDocuments().exec();
    ctx.set('Last-Page', Math.ceil(postCount / 10));
    ctx.body = posts.map(post => ({
      ...post,
      body:
        post.body.length < 200 ? post.body : `${post.body.slice(0, 200)}...`,
    }));
  } catch (e) {
    ctx.throw(500, e);
  }
};

/*
  GET /api/posts/:id
*/
export const read = async ctx => {
  const {id} = ctx.params;
  try {
    const post = await Post.findById(id).exec();
    if(!post) {
      ctx.status = 404;
      return;
    }
    ctx.body = post;
  } catch(e) {
    ctx.throw(500, e);
  }
};

/*
  DELETE /api/posts/:id
*/

export const remove = async ctx => {
  const {id} = ctx.params;
  try {
    const post = await Post.findByIdAndRemove(id).exec();
    ctx.status = 204; // no content 성공했지만 응답데이터 없음
    ctx.body = post;
  } catch(e) {
    ctx.throw(500, e);
  }
};

/*
  PATCH /api/posts/:id
  {
```

```
    title: '수정',
    body: '수정 내용',
    tags: ['수정', '태그']
  }
*/
export const update = async ctx => {

  const { id } = ctx.params;
  // write 에서 사용한 schema 와 비슷한데, required() 가 없습니다.
  const schema = Joi.object().keys({
    title: Joi.string(),
    body: Joi.string(),
    tags: Joi.array().items(Joi.string()),
  });

  // 검증 후, 검증 실패시 에러처리
  const result = schema.validate(ctx.request.body);
  if (result.error) {
    ctx.status = 400; // Bad Request
    ctx.body = result.error;
    return;
  }
  try {
    const post = await Post.findByIdAndUpdate(id, ctx.request.body, {
      new: true, // 이 값을 설정하면 업데이트된 데이터를 반환한다, false일 경우
업데이트전 값을 리턴
    }).exec();
    if(!post) {
      ctx.status = 404;
      return;
    }
    ctx.status = 204; // no content 성공했지만 응답데이터 없음
    ctx.body = post;
  } catch(e) {
    ctx.throw(500, e);
  }
};
```

- POST http://localhost:4000/api/auth/login
- POST http://localhost:4000/api/posts

```
{
    "title": "제목",
    "body": "내용",
    "tags": ["태그1", "태그2"]
}
```



23.5.5 포스트 수정 및 삭제시 권한 확인하기

- 미들웨어에서 처리할 경우 기존 checkObjectId를 getPostById로 변경후 해당 id의 포스트를 ctx.state로 저장

src/posts/posts.ctrl.js - getPostById, read, checkOwnPost

- getPostById : checkObjectId를 변경(예제 소스는 별도로 작성함)
- read : getPostById 작성함으로 코드가 간결해 짐
- checkOwnPost : 로그인중인 사용자가 작성한 포스트여부 확인
  - mongoDB의 id값을 문자열로 비교하기 위해 .toString()으로 처리

```
import Post from '../../models/post';
import mongoose from 'mongoose';
import Joi from 'joi';

const { ObjectId } = mongoose.Types;

export const checkObjectId = (ctx, next) => {

  const { id } = ctx.params;
  if (!ObjectId.isValid(id)) {
    ctx.status = 400; // Bad Request
    return;
  }
  return next();
};

export const getPostById = async (ctx, next) => {

  const { id } = ctx.params;
  if (!ObjectId.isValid(id)) {
    ctx.status = 400; // Bad Request
    return;
  }
```

```
  try {
    const post = await Post.findById(id);
    // 포스트가 존재하지 않을 때
    if (!post) {
      ctx.status = 404; // Not Found
      return;
    }
    ctx.state.post = post;
    return next();
  } catch (e) {
    ctx.throw(500, e);
  }
};

export const checkOwnPost = (ctx, next) => {
  const { user, post } = ctx.state;
  if (post.user._id.toString() !== user._id) {
    ctx.status = 403;
    return;
  }
  return next();
};

/*
  POST /api/posts
  {
    title: '제목',
    body: '내용',
    tags: ['태그1', '태그2']
  }
*/
export const write = async ctx => {

  const schema = Joi.object().keys({
    // 객체가 다음 필드를 가지고 있음을 검증
    title: Joi.string().required(), // required() 가 있으면 필수 항목
    body: Joi.string().required(),
    tags: Joi.array()
      .items(Joi.string())
      .required(), // 문자열로 이루어진 배열
  });

  // 검증 후, 검증 실패시 에러처리
  const result = schema.validate(ctx.request.body);
  if (result.error) {
    ctx.status = 400; // Bad Request
    ctx.body = result.error;
    return;
  }

  const { title, body, tags } = ctx.request.body;

  const post = new Post({
    title,
    body,
    tags,
    user: ctx.state.user,
```

```
  });
  try {
    await post.save();
    ctx.body = post;
  } catch (e) {
    ctx.throw(500, e);
  }
};

/*
  GET /api/posts
*/
export const list = async ctx => {

  // query 는 문자열이기 때문에 숫자로 변환해주어야합니다.
  // 값이 주어지지 않았다면 1 을 기본으로 사용합니다.
  const page = parseInt(ctx.query.page || '1', 10);

  if (page < 1) {
    ctx.status = 400;
    return;
  }

  try {
    const posts = await Post.find()
      .sort({ _id: -1 })
      .limit(10)
      .skip((page - 1) * 10)
      .lean()
      .exec();
    const postCount = await Post.countDocuments().exec();
    ctx.set('Last-Page', Math.ceil(postCount / 10));
    ctx.body = posts.map(post => ({
      ...post,
      body:
        post.body.length < 200 ? post.body : `${post.body.slice(0, 200)}...`,
    }));
  } catch (e) {
    ctx.throw(500, e);
  }
};

/*
  GET /api/posts/:id
*/
export const read = async ctx => {
  ctx.body = ctx.state.post;
};

/*
  DELETE /api/posts/:id
*/

export const remove = async ctx => {
  const {id} = ctx.params;
  try {
    const post = await Post.findByIdAndRemove(id).exec();
```

```
    ctx.status = 204; // no content 성공했지만 응답데이터 없음
    ctx.body = post;
  } catch(e) {
    ctx.throw(500, e);
  }
};

/*
  PATCH /api/posts/:id
  {
    title: '수정',
    body: '수정 내용',
    tags: ['수정', '태그']
  }
*/
export const update = async ctx => {

  const { id } = ctx.params;
  // write 에서 사용한 schema 와 비슷한데, required() 가 없습니다.
  const schema = Joi.object().keys({
    title: Joi.string(),
    body: Joi.string(),
    tags: Joi.array().items(Joi.string()),
  });

  // 검증 후, 검증 실패시 에러처리
  const result = schema.validate(ctx.request.body);
  if (result.error) {
    ctx.status = 400; // Bad Request
    ctx.body = result.error;
    return;
  }
  try {
    const post = await Post.findByIdAndUpdate(id, ctx.request.body, {
      new: true, // 이 값을 설정하면 업데이트된 데이터를 반환한다, false일 경우
업데이트전 값을 리턴
    }).exec();
    if(!post) {
      ctx.status = 404;
      return;
    }
    ctx.status = 204; // no content 성공했지만 응답데이터 없음
    ctx.body = post;
  } catch(e) {
    ctx.throw(500, e);
  }
};
```

src/posts/index.js

```
import Router from 'koa-router';
import * as postsCtrl from './posts.ctrl';
import checkLoggedIn from '../../lib/checkLoggedIn'

const posts = new Router();

posts.get('/', postsCtrl.list);
```

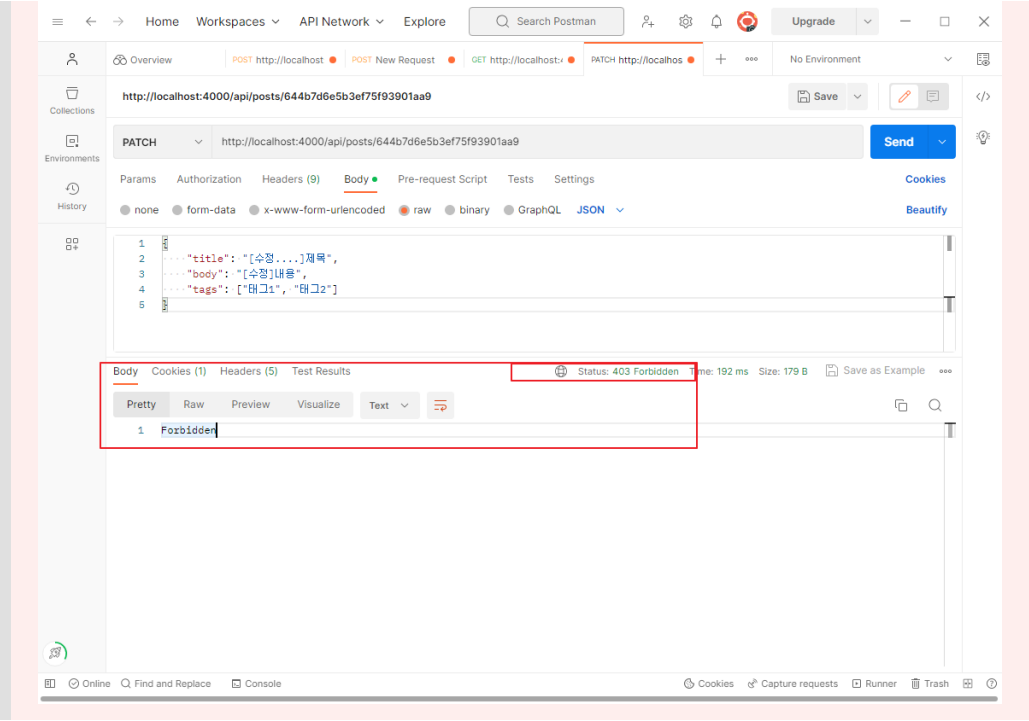
```
posts.post('/', checkLoggedIn, postsCtrl.write);

const post = new Router(); // /api/posts/:id
post.get('/', postsCtrl.read);
// post.delete('/', checkLoggedIn, postsCtrl.remove);
// post.patch('/', checkLoggedIn, postsCtrl.update);
post.delete('/', checkLoggedIn, postsCtrl.checkOwnPost, postsCtrl.remove);
post.patch('/', checkLoggedIn, postsCtrl.checkOwnPost, postsCtrl.update);

// posts.use('/:id', postsCtrl.checkObjectId, post.routes());
posts.use('/:id', postsCtrl.getPostById, post.routes());

export default posts;
```

신규회원등록후 post수정 및 삭제 테스트하기

- POST http://localhost:4000/api/auth/register

```
{
  "username": "iwbaek",
  "password": "12345"
}
```

- POST http://localhost:4000/api/auth/login
- GET http://localhost:4000/api/posts or http://localhost:4000/api/posts/644b7c8b5b3ef75f93901a96
- DELETE http://localhost:4000/api/posts/644b7c8b5b3ef75f93901a96



23.6 username / tags로 포스트 필터링하기

src/posts/posts.ctrl.js - list

```
// 중략
/*
  GET /api/posts?username=&tag=&page=
*/
export const list = async (ctx) => {
```

```
  // query 는 문자열이기 때문에 숫자로 변환해주어야합니다.
  // 값이 주어지지 않았다면 1 을 기본으로 사용합니다.
  const page = parseInt(ctx.query.page || '1', 10);

  if (page < 1) {
    ctx.status = 400;
    return;
  }

  const { tag, username } = ctx.query;
  // tag, username 값이 유효하면 객체 안에 넣고, 그렇지 않으면 넣지 않음
  const query = {
    ...(username ? { 'user.username': username } : {}),
    ...(tag ? { tags: tag } : {}),
  };

  try {
    const posts = await Post.find(query)
      .sort({ _id: -1 })
      .limit(10)
      .skip((page - 1) * 10)
      .lean()
      .exec();
    const postCount = await Post.countDocuments(query).exec();
    ctx.set('Last-Page', Math.ceil(postCount / 10));
    ctx.body = posts.map((post) => ({
      ...post,
      body:
        post.body.length < 200 ? post.body : `${post.body.slice(0, 200)}...`,
    }));
  } catch (e) {
    ctx.throw(500, e);
  }
};
```

- GET http://localhost:4000/api/posts?username=gilbaek&tags=태그1