Node.js와 MongoDB II

03 Express.js와 MongoDB로 웹서비스 만들기 2

/* elice */

목차

01. 회원가입 구현하기

02. Passport.js와 로그인

03. Session Store

04. 회원과 게시글의 연동

05. CSR로 댓글 기능 구현하기

06. 추가 - MongoDB Aggregation

🎯 수강목표

1. 회원가입, 로그인 이해하기

웹 서비스에서의 회원가입과 로그인의 플로우를 이해하고
이를 구현하는 방법에 대해 학습한다.

2. Session과 Session Store 이해하기

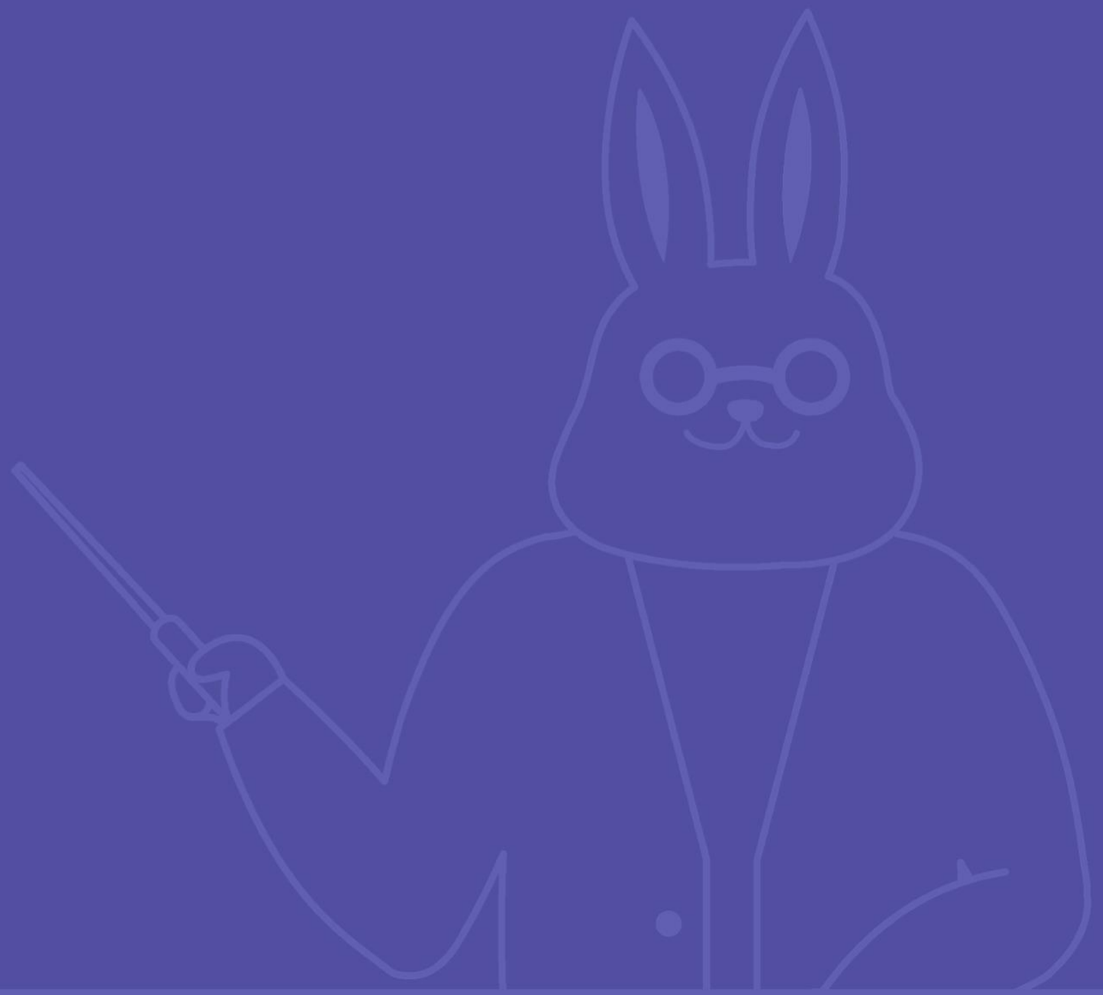
웹 서비스에서의 Session과 Session Store의 동작 방식을 이해하고
이를 프로젝트에 적용하는 방법에 대해 학습한다.

3. Client-Side Rendering 구현하기

간단한 Client-Side Rendering 방식에 대해 알아보고
이를 구현하며 CSR의 동작 원리를 이해한다.

01

회원가입 구현하기



회원가입 설명

이메일, 이름, 패스워드의 간단한 정보만 사용

- 이메일의 형식이 올바른지 확인
- 비밀번호 최소 길이 확인
- 패스워드와 패스워드 확인 문자가 일치하는지 확인

회원 정보를 데이터베이스에 저장하기

회원의 비밀번호를 **그대로 저장**한다면?

관리자가 모든 회원의 비밀번호를 알 수 있음 → **보안 취약점** 발생

비밀번호 저장 방법 - Hash

Hash는 문자열을 **되돌릴 수 없는 방식**으로 암호화하는 방법
→ hash 출력값을 이용해 사용자의 **비밀번호를 알아낼 수 없음**

비밀번호의 Hash 값을 데이터베이스에 저장하고,
로그인 시 **전달된 비밀번호를 Hash** 하여 **저장된 값과 비교**해 로그인을 처리

SHA1 – 사용 방법

crypto sha1

```
const hash = crypto.createHash('sha1');
hash.update(password);
hash.digest("hex");
```

Node.js 의 기본제공 모듈인 **crypto 모듈**을 사용하여 hash 값을 얻을 수 있음
간단하게 **sha1** 알고리즘을 사용하거나
보다 강력한 sha224, sha256 등의 알고리즘도 사용할 수 있음

회원가입 구현하기

1. 회원가입 페이지 구현
2. script를 이용해 이메일 형식, 비밀번호 확인 문자 확인
3. form을 이용해 post 요청 전송
4. 회원가입 처리 및 redirect

회원가입 페이지 만들기

회원가입 페이지

```
...
form(action="/join" method="post" onsubmit="return check()")
 table
  tbody
   tr
    td 이메일
    td: input(type="text" name="email")
   tr
    td 이름
    td: input(type="text" name="name")
   tr
    td 비밀번호
    td: input(type="password" name="password")
   tr
    td 비밀번호 확인
    td: input(type="password" name="password_confirm")
   tr
    td(colspan="2")
     input(type="submit" value="가입하기")
...
```

```
...
script.
 function check() {
  const email = document.querySelector('[name="email"]').value;
  if (!/^\S+@\S+\.\S+$/.test(email)) {
   alert('이메일 형식이 올바르지 않습니다.');
   return false;
  }

  const password = document
   .querySelector('[name="password"]')
   .value;
  if (password.length < 8) {
   alert("최소 8자리 이상의 비밀번호를 설정해 주세요.");
   return false;
  }

  const passwordConfirm = document
   .querySelector('[name="password_confirm"]')
   .value;
  if (password != passwordConfirm) {
   alert('비밀번호 확인이 일치하지 않습니다.');
   return true;
  }
  return false;
 }
```

회원가입 요청 처리하기

가입 요청 처리

```
router.post(... => {
  const { email, name, password } = req.body;
  const pwHash = getHash(password);
  const exists = await User.findOne({
    email,
  });

  if (exists) {
    throw new Error('이미 가입된 메일입니다');
  }

  await User.create({
    email,
    name,
    password: pwHash,
  });

  res.redirect('/');
});
```

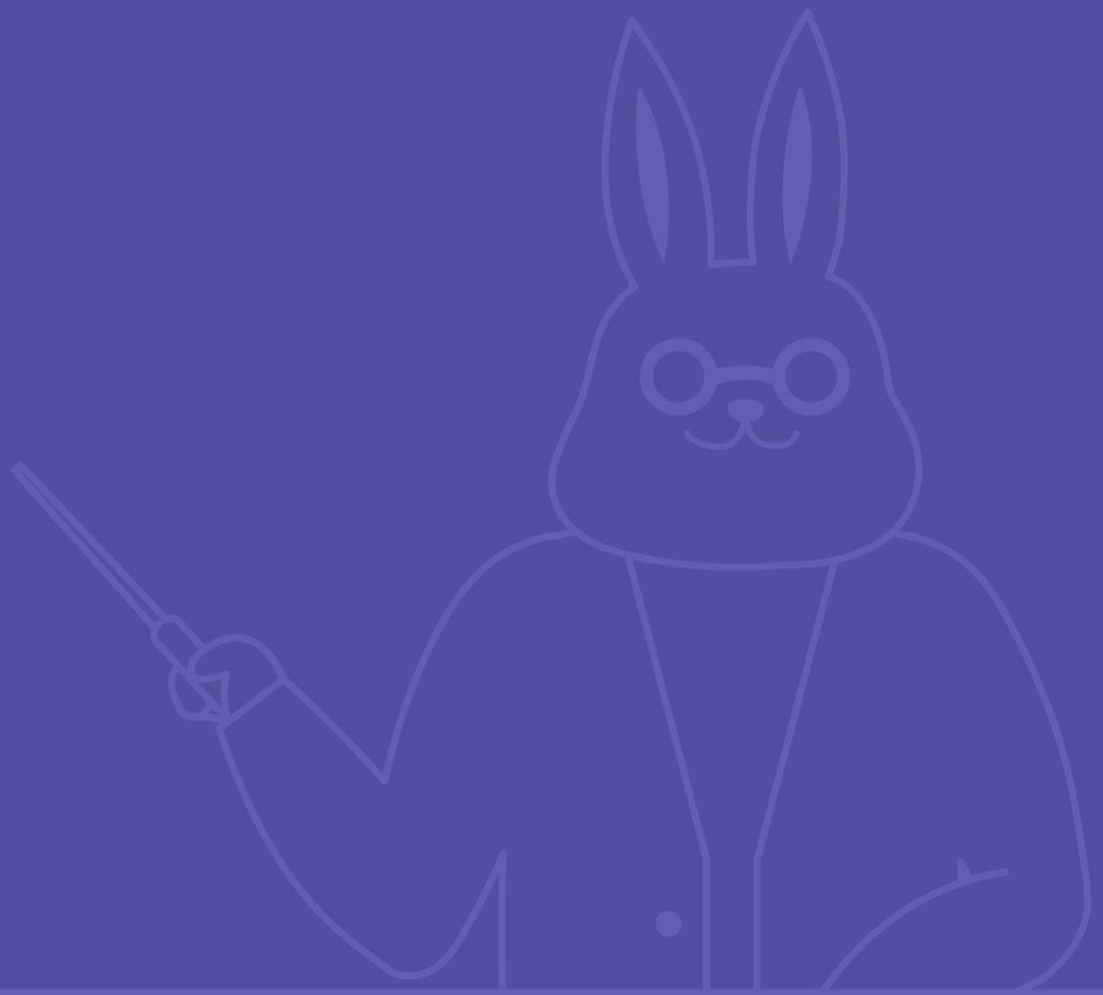
비밀번호 hash 값 저장

이미 존재하는 회원인지 체크

가입 후 메인화면으로 **redirect**

02

Passport.js와 로그인



Passport.js란?

Express.js 어플리케이션에 간단하게 **사용자 인증 기능**을 구현하게 도와주는 패키지
유저 **세션 관리** 및 **다양한 로그인 방식** 추가 가능

passport-local

passport는 **다양한 로그인 방식**을 구현하기 위해 **strategy라는 인터페이스**를 제공
strategy **인터페이스에 맞게 설계**된 **다양한 구현체**들이 있음 (facebook, google, ...)
passport-local은 **username, password**를 사용하는 로그인의 구현체

로그인 기능 구현하기

1. 로그인 화면 구성하기
2. passport-local strategy로 로그인 구현하기
3. passport.js 설정하기
4. passport로 요청 처리하기

로그인 기능 구현하기 - 로그인 화면 구성하기

로그인 화면

```
...

form(action="/auth" method="post" onsubmit="return check()")
  table
    tbody
      tr
        td 이메일
        td: input(type="text" name="email")
      tr
        td 비밀번호
        td: input(type="password" name="password")
      tr
        td(colspan="2")
          td: input(type="submit" name="로그인")

...
```

```
...

script.
  function check() {
    const email = document
      .querySelector('[name="email"]')
      .value
    if (!email) {
       alert("이메일을 입력해 주세요.");
       return false;
    }
    const password = document
      .querySelector('[name="password"]')
      .value
    if (!email) {
       alert("비밀번호를 입력해 주세요.");
       return false;
    }
    return true;
  }
```

로그인 기능 구현하기 - passport-local strategy

passport-local

```
const config = {
  usernameField: 'email',
  passwordField: 'password',
};
// 아이디 패스워드 필드 설정 필수!

const local = new LocalStrategy(config,
...
```

```
..., async (email, password, done) => {
  try {
    const user = await User.findOne({ email });
    if (!user) {
      throw new Error('회원을 찾을 수 없습니다.');
    }
    if (user.password !== password) {
      throw new Error('비밀번호가 일치하지 않습니다.');
    }

    // 세션에 저장되는 유저 정보의 최소화
    done (null, {
      shortId: user.shortId,
      email: user.email,
      name: user.name,
    });
  } catch (err) {
    done(err, null);
  }
})
```

로그인 기능 구현하기 - Passport.js 설정하기

passport.use

```
const local = require('./strategies/local');
passport.use(local);
```

작성한 strategy를 **passport.use를 이용해 사용하도록 선언**해야 함

passport.use를 이용해 strategy를 사용하도록 선언한 후 **passport.authenticate**를 사용해 해당 strategy를 이용해 요청을 처리할 수 있음

로그인 기능 구현하기 - Passport.js 로 post 요청 처리하기

passport 적용

```
--- routes/auth.js ---
router.post('/',
  passport.authenticate('local');
--- app.js ---
const session =
  require('express-session');
app.use(session({
  secret: 'secret',
  resave: false,
  saveUninitialized: true
}));
app.use(passport.initialize());
app.use(passport.session());
app.use('/auth', authRouter);
```

passport.authenticate 함수를 **http 라우팅에 연결**하면 passport가 자동으로 해당하는 strategy를 사용하는 **request handler를 자동 생성**

express-session과 passport.session()을 사용하면 passport가 로그인 시 **유저 정보를 세션에 저장하고 가져오는 동작**을 자동으로 수행해 줌

로그인 기능 구현하기 - session 유저 활용하기

passport 적용

```
passport.serializeUser((user, callback) => {
  callback(null, user);
});

passport.deserializeUser((obj, callback) => {
  callback(null, obj);
});
```

session을 이용해 user를 사용할 때에는
serializeUser 와 **deserializeUser** 를
설정해 주어야 함
이는 세션에 user 정보를
변환하여 저장하고 가져오는 기능을 제공
ex) **회원 id 만 세션에 저장**하고,
사용 시 회원 정보를 디비에서 찾아서 사용

※ 세션 사용 시 위 두 함수를 작성하지 않으면
passport 로그인이 동작하지 않음

로그아웃

logout

```
router.get('/logout', ... {
  req.logout();
  res.redirect('/');
});
```

passport는 **req.logout 함수**를 통해 **세션의 로그인 정보를 삭제**하여, 로그아웃 기능을 구현할 수 있음

로그인 확인 미들웨어

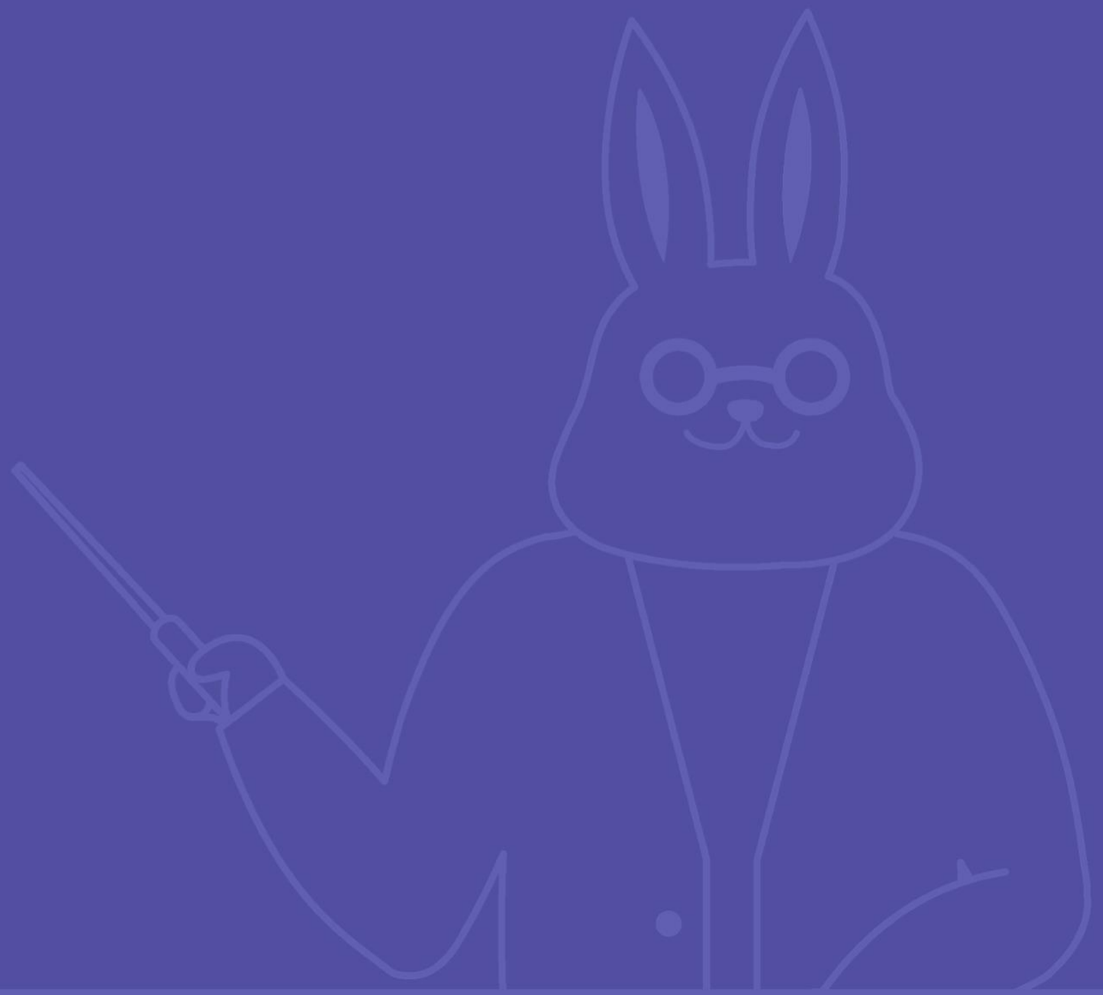
logout

```
function loginRequired(req, res, next) {
  if (!req.user) {
    res.redirect('/');
    return;
  }
  next();
}

app.use('/posts', loginRequired, postsRouter);
```

로그인을 필수로 설정하고 싶을 경우,
미들웨어를 사용하여 체크할 수 있음.

03

Session Store



Session 이란?

웹 서버가 **클라이언트의 정보**를 클라이언트별로 구분하여 **서버에 저장**하고
클라이언트 요청 시 **Session ID를 사용**하여 클라이언트의 **정보를 다시 확인**하는 기술

※ 클라이언트가 정보를 저장하고, 요청 시 정보를 보내는 Cookie와 대조됨

Session 작동 방식

서버는 세션을 생성하여 세션의 구분자인 **Session ID를 클라이언트에 전달**함
클라이언트는 요청 시 **session id를 함께 요청에 담아서 전송**
서버는 전달받은 session id로 해당하는 세션을 찾아 **클라이언트 정보를 확인**

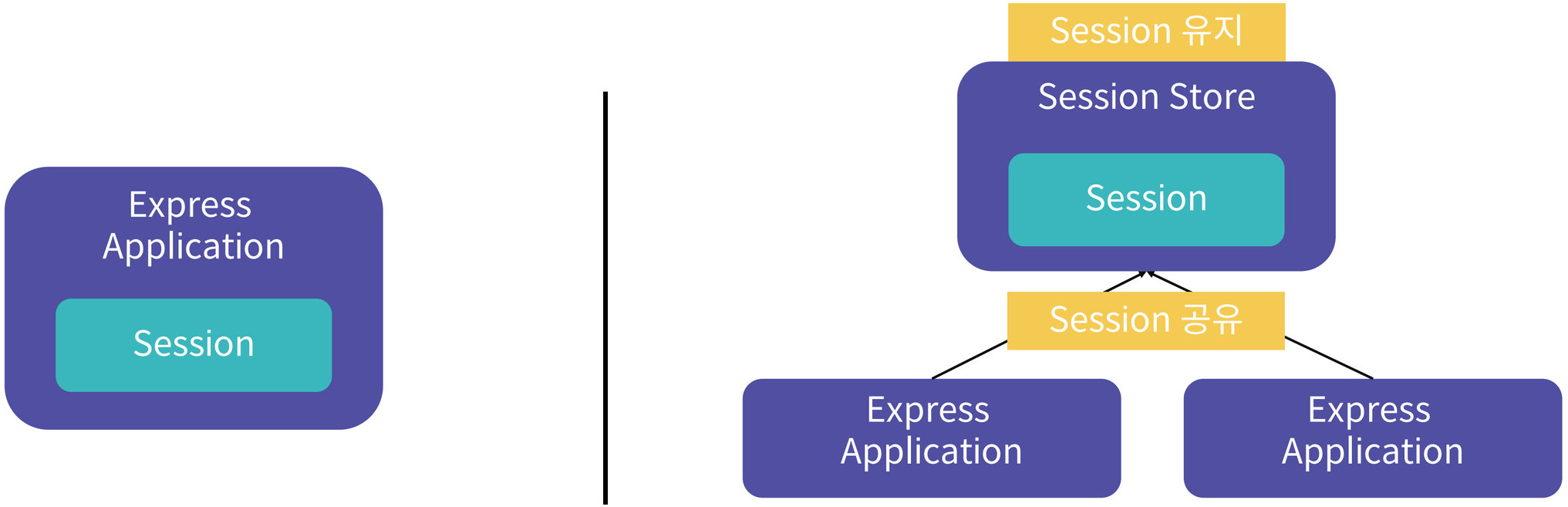
Express.js의 session

express-session 패키지를 사용하여 간단하게 **session 동작을 구현**할 수 있음
특별한 설정 없이, **자동으로 session 동작을 구현**해 줌
→ 자동으로 session id를 클라이언트에 전달, session id로 클라이언트 정보 확인

Session Store를 사용하는 이유

express-session 패키지는 session을 기본적으로 **메모리에 저장**함
따라서 현재 구현된 어플리케이션을 **종료 후 다시 실행**하면,
모든 유저의 로그인이 해제됨
혹은 서버가 여러 대가 있을 경우, **서버 간 세션 정보 공유할 수 없음**

Session Store 구성



MongoDB를 Session Store로 사용하기

connect-mongo 패키지를 이용해, **MongoDB를 session store로** 사용 할 수 있음
connect-mongo 패키지는 **express-session 패키지의 옵션**으로 전달 가능
자동으로 session 값이 변경될 때 update되고, session이 호출될 때 find 함

connect-mongo

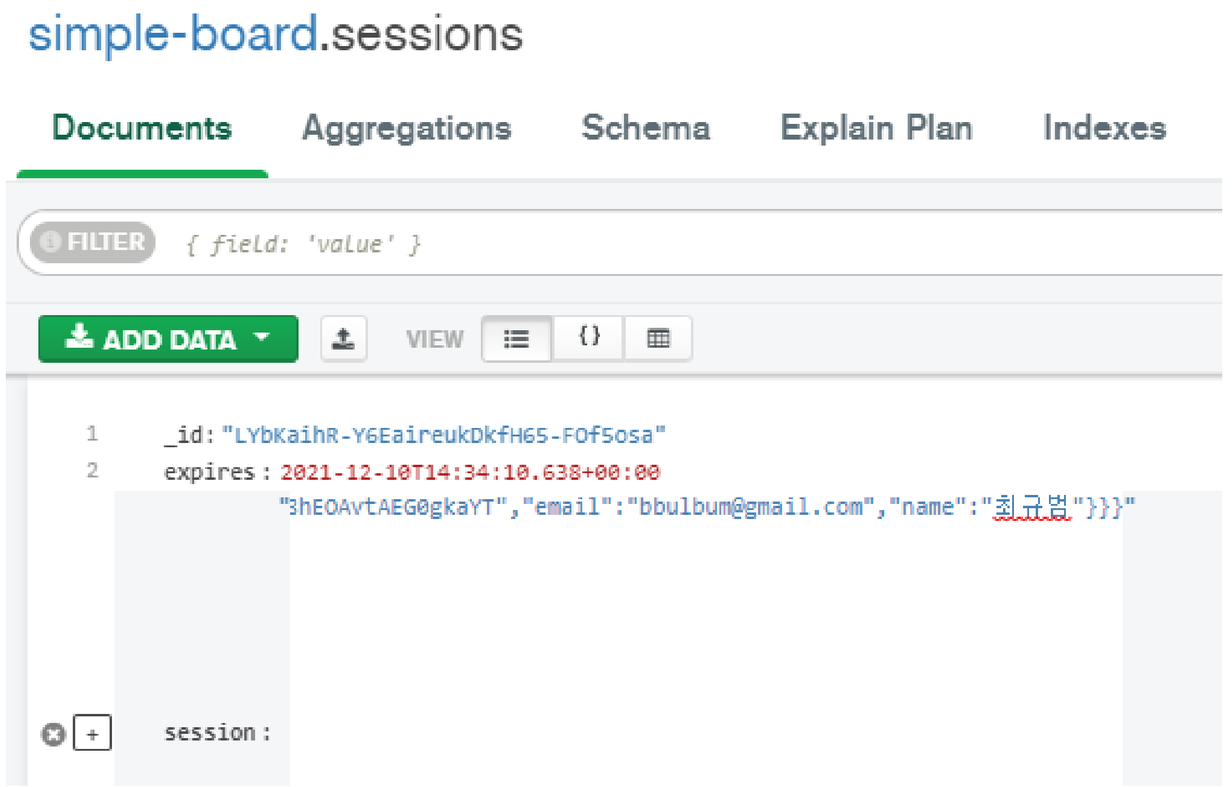
connect-mongo

```
const MongoStore =
  require('connect-mongo');
app.use(session({
  secret: 'SeCrEt',
  resave: false,
  saveUninitialized: true,
  store: MongoStore.create({
    mongoUrl: 'mongoUrl',
  }),
}));
```

connect-mongo 패키지를 사용해
express-session 설정 시
store 옵션에 전달하고, mongoUrl을 설정

세션데이터를 **몽고디비에 저장하고 관리**하는
기능을 **자동으로 수행**해 줌

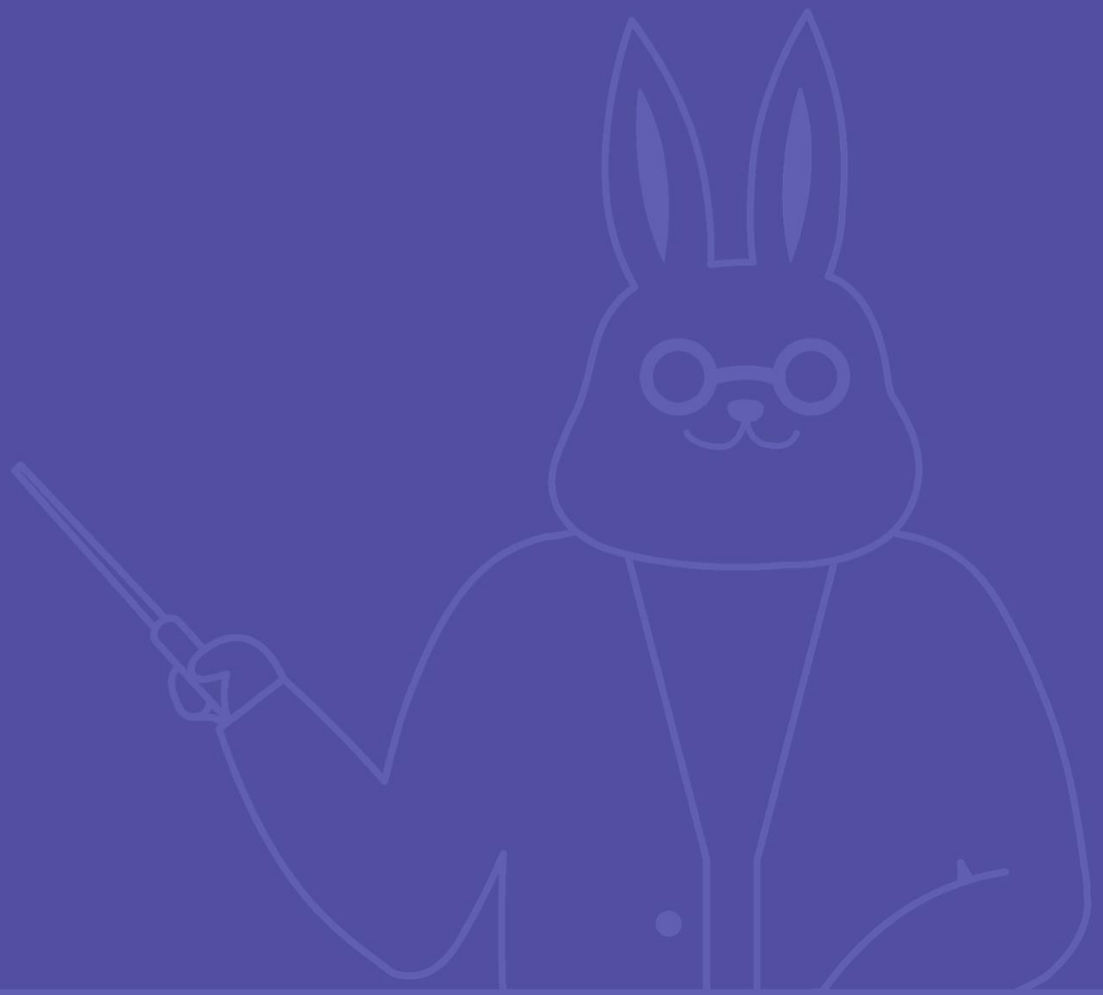
세션 확인 해 보기



MongoDB Compass를 이용해 확인한 session

04

회원과 게시글의 연동



회원과 게시글 연동 기능 설명

게시글 작성 시 **로그인된 회원 정보**를 **작성자**로 추가
게시글 - 작성자는 **populate하여 사용**하도록 구현
게시글 수정, 삭제 시 **로그인된 유저와 작성자가 일치**하는지 확인
작성자의 게시글 모아 보기 기능 구현

PostSchema 수정

author 타입 추가

```
author: {
  type: Schema.Types.ObjectId,
  ref: 'User',
  required: true,
},
```

PostSchema에 **author 추가**

populate를 사용하기 위해 **ObjectID 사용**

ref를 **유저 모델의 이름**인 **'User'**로 선언

게시글 등록 요청 수정

게시글에 작성자 추가

```
const author = await User.find({
  shortId: req.user.shortId,
});
if (!author) {
  throw new Error('No User');
}

await Post.create({
  title,
  content,
  author,
});
```

req.user에는 strategy에서 **최소한의 정보**로 저장한 shortId, email, username만 가지고 있음

Post 생성 시 user의 **ObjectID를 전달**해야 하는데, 이를 위해 **User에서 shortId로 회원을 검색**하여 한 번 더 검증

type: ObjectID로 선언한 필드에 객체가 주어지면 **자동으로 ObjectID** 사용

게시글에 작성자 연동

populate

```
--- ./routes/posts.js ---
router.get('/', ... {
  ...
  const posts = await Post
    .find({})
    ...
    .populate('author');

res.render('posts/list', { posts });

--- ./views/posts/list.pug ---
...
  td post.author.name
```

게시글 find 시 **populate를 추가**하여
ObjectID로 저장된 author를
각 게시글에 주입

사용 시 **post.author.{field}**로 사용 가능

게시글 수정, 삭제 시 유저 확인

수정, 삭제 시 유저 확인

```
const post = await Post.find({
  shortId,
}).populate('author');

if (post.author.shortId !== req.user.shortId) {
  throw new Error('Not Authorized');
}
```

게시글 수정, 삭제 시 **작성자를 populate**하여
로그인된 사용자와 일치하는지 확인

작성자 게시글 모아 보기 기능구현

기본적으로 MongoDB는 Document 검색 시, **전체 문서를 하나씩 확인**함
하나씩 확인하기 때문에 **매우 비효율적인 검색** 수행
데이터가 많아질 경우 **속도 저하의 가장 큰 원인**이 됨

Index

MongoDB는 **검색을 위해 Document를 정렬**하여 저장하는 기능을 제공함
Index를 설정하면 주어진 **쿼리를 효율적으로 수행**하여 **성능을 향상**시킬 수 있음

※ 다중 키, 좌표, 텍스트 등의 특별한 값으로 정리되는 인덱스도 제공

author에 index 설정하기

index 사용

```
author: {
  type: Schema.Types.ObjectId,
  ref: 'User',
  required: true,
  index: true,
},
```

PostSchema의 author 속성에 **index: true 옵션**을 사용하면 mongoose가 **자동으로 MongoDB에 인덱스를 생성**해 줌

이미 데이터가 많은 상태에서 인덱스를 추가할 시 작업 시간이 길어져, **MongoDB가 응답하지 않을 수 있음**
→ **예상되는 인덱스를 미리 추가**하는 것이 좋음

회원 게시글 라우팅 추가하기

회원 게시글

```
--- ./routes/users.js ---
...
router.get('/:shortId/posts', ... => {
   ...
   const { shortId } = req.params;
   const user = await User.find({ shortId });
   const posts = await Post
     .find({ author: user })
     .populate('author');
   res.render('posts/list', { posts, user });
});
...
```

RESTful 한 구성을 위해,
회원 → 게시글의 경로를
/users/{userId}/posts로 구성

게시글 목록 view는
기존에 작성한 posts/list.pug를 **재활용**

게시글 목록 화면 수정

게시글 목록 화면

```
h2= user ? `${user.name}의 게시글`: "전체 게시글"
...

td: a(href=`/users/${post.author.shortId}`)
  = post.author.name
```

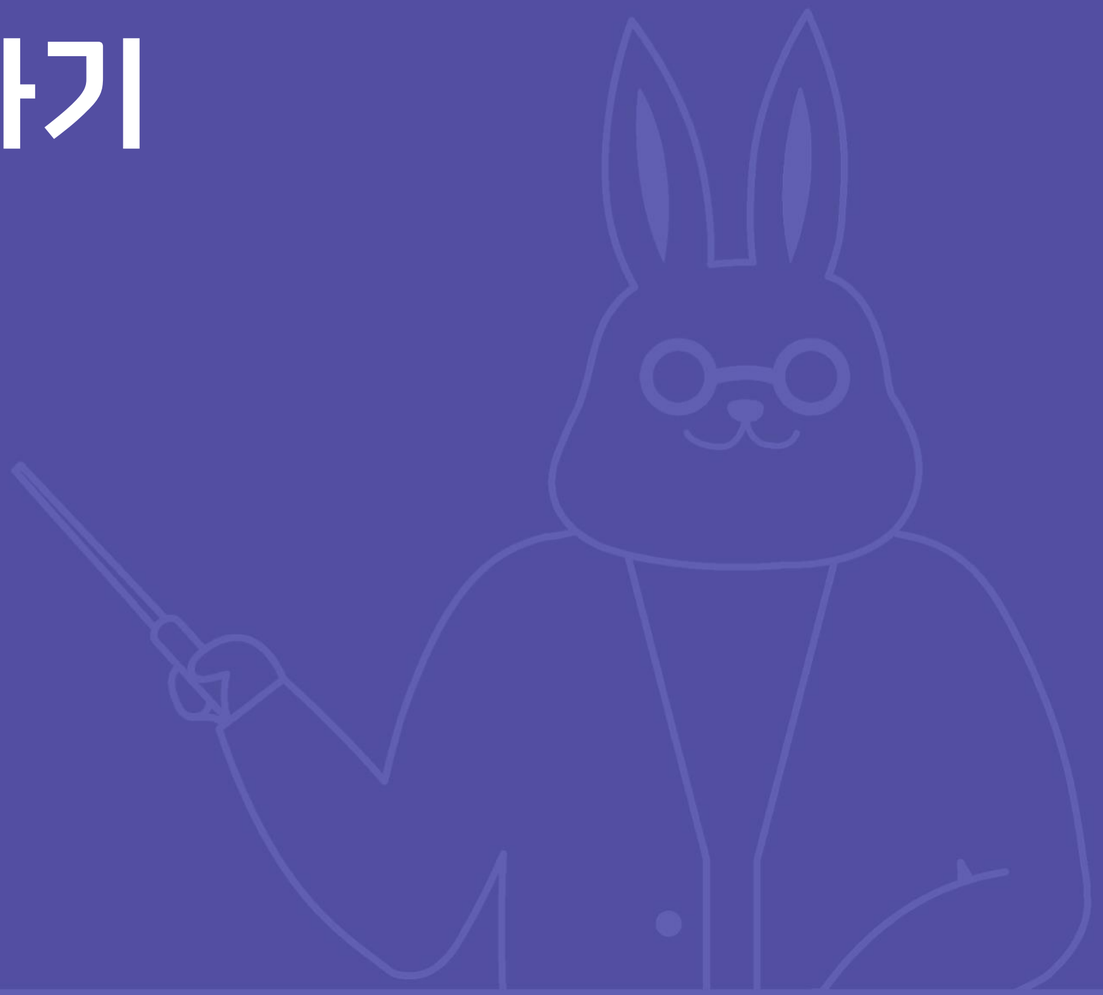
게시글 **목록 화면을 재활용**하기 위해 수정

유저의 게시글인 경우
"###의 게시글"이라는 제목 사용

게시글의 사용자 이름에
유저의 게시글 link 추가

05

CSR로 댓글 기능 구현하기



CSR을 구현하는 방법

1. 페이지 로드 시 **필요한 리소스**를 **클라이언트에 선언**
2. 클라이언트에서 필요한 **데이터**를 **비동기 호출**
3. 클라이언트가 전달받은 **데이터를 가공**, **리소스를 사용**하여 **화면에 표시**

클라이언트에 리소스 선언 - HTML Template

클라이언트에 리소스를 선언하기 위한 다양한 방법이 존재. (React.js, Vue.js 등)
본 강의에서는 간단하게 **HTML Template 기능**을 사용
HTML Template은 **브라우저에 표시되지 않는** HTML Element를 작성해 두고,
JavaScript로 이를 화면에 **반복적으로 그릴 수 있게 하는** 기술

댓글 화면 작성하기

posts/view.pug

```
...
table
  thead
    tr
      td(colspan="2")
        input#content(type="text")
      td: button(onclick="writeComment()")
          댓글 작성
  tbody#comments
template#comment-template
  tr
    td.content
    td.author
    td.createdAt
```

게시글 상세 화면 하단에
댓글작성, 목록 화면 추가

HTML Template 사용하여
한 개의 댓글이 표시될 모양을 선언

JavaScript 로 조작하기 위해
id, class를 선언하는 것이 유용함

데이터 비동기 호출 - API 작성하기

지금까지의 구현들은 **HTTP 응답으로 HTML을 전송**하는 방식
CSR을 구현하기 위해서는 HTM이 아닌,
데이터만 주고받을 수 있는 API를 구성해야 함 (JSON 사용)

- 댓글 작성 API와 댓글 목록 API만 구현
- 댓글 작성 시 댓글목록을 다시 불러와 그리는 형식으로 구현

게시글에 댓글 추가하기

PostSchema

```
const CommentSchema = new Schema({
  content: String,
  author: {
    type: Schema.Types.ObjectId,
    ref: 'User',
  },
}, {
  timestamps: true,
});

const PostSchema = new Schema({
  ...
  comments: [CommehtSchema],
  ...
```

mongoose의 **sub-schema**를 이용하여
Post 스키마에 **Comment를 배열로 추가**

populate를 사용할 때,
ObjectID만 저장하는 것과는 다르게
Comment의 내용을 게시글이 포함하게 됨

※ sub-schema 내부에서도 populate 가능

API 작성하기 - 댓글 작성

routes/api.js

```
...
router.post('/posts/:shortId/comments', ... {
  const { shortId } = req.params;
  const { content } = req.body;
  const author = await User
    .findOne({ shortId: req.user.shortId });

  await Post.updateOne({ shortId }, {
    $push: { comments: {
      content,
      author,
    }},
  });

  res.json({ result: 'success' });
});
...
```

api 라우터를 추가하고, RESTful하게
api/posts/{postId}/comment 경로로
댓글 작성 기능 구현

게시글 업데이트 시 **$push**를 사용하여
comments 배열에 새로 작성된 **댓글 추가**
→**동시에 들어오는 요청**에 대해 정확하게 처리

api는 render 하지 않고 **json으로 응답**

API 작성하기 - 댓글 목록

routes/api.js

```
...
router.get('/posts/:shortId/comments', ... {
  const { shortId } = req.params;
  const post = await Post
    .findOne({ shortId });

  await User.populate(post.comments, {
    path: 'author'
  });

  res.json(post.comments);
});
...
```

/api/posts/{postId}/comments로
RESTful 경로 설정

find에 populate하지 않고
User (model)의 populate를 사용하는
방법도 가능

데이터 비동기 호출 - fetch로 클라이언트에서 api 호출하기

브라우저는 **비동기 HTTP 요청**을 fetch 함수를 이용해 제공 함
jQuery의 ajax와 유사한 기능, **jQuery를 사용하지 않고도** HTTP 요청 구현 가능
fetch를 이용하면 간단하게 **JavaScript로 HTTP 요청을 비동기 처리**할 수 있음

fetch로 API 호출하고 처리하기 - 댓글 작성하기

posts/view.pug

```
...
script.
  function writeComment() {
    const input = document.querySelector('#content')
    const content = input.value;
    fetch('/api/posts/#{post.shortId}/comments', {
      method: 'post',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ content }),
    })
    .then(() => {
      if (res.ok) {
        input.value = '';
          loadComments();
        } else {
          alert('오류가 발생했습니다.');
        }
    });
```

댓글 작성 버튼 클릭 시
writeComment() 실행

input#content에서 내용을 읽어
fetch로 댓글 작성 api 호출

호출 결과의 성공 여부를 확인하여,
댓글 다시 불러오기 실행

fetch로 API 호출하고 처리하기 - 댓글 목록 불러오기

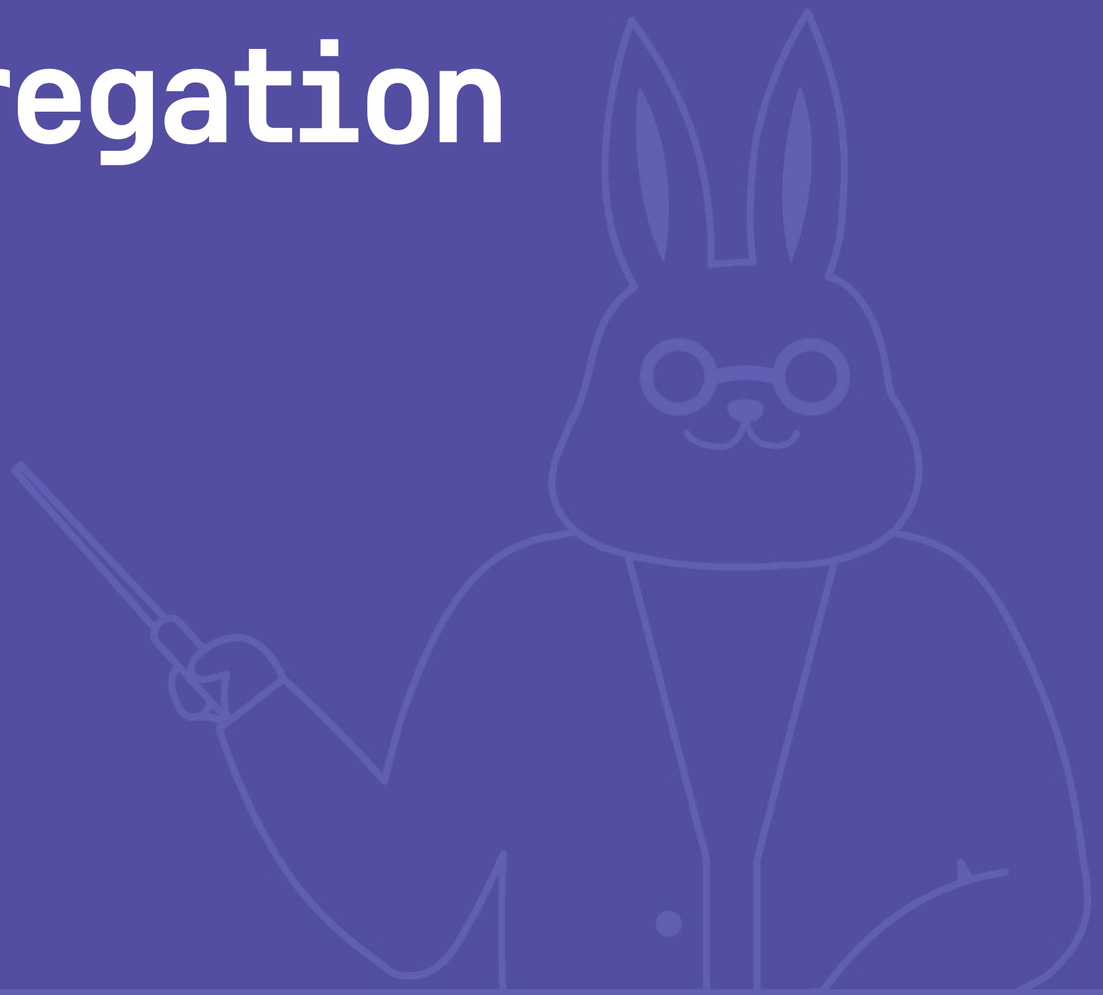
posts/view.pug

```
// 댓글 목록 api 호출하기
script.
  loadComments();
  function loadComments() {
    document
      .querySelector('#comments')
      .innerHTML = ''; // 이전 목록 삭제
    fetch('/api/posts/#{post.shortId}/comments')
      .then((res) => {
        if (res.ok) {
          return res.json();
        } else {
          throw new Error('댓글을 불러오지 못했습니다');
        }
      })
      .then((comments) => {
        comments.forEach(addComment);
      });
      .catch((err) => alert(err.message));
  }
```

```
// HTML Template 사용하여 댓글 화면에 표시하기
function addComment(comment) {
  const template = document
    .querySelector('#comment-template');
  const node = document
    .importNode(template.content, true);

  node.querySelector('.content')
    .textContent = comment.content;

  node.querySelector('.author')
    .textContent = comment.author.name;

  node.querySelector('.createdAt')
    .textContent = comment.createdAt;

  document.querySelector('#comments')
    .appendChild(node);
}
```

06

추가 - MongoDB Aggregation



Aggregation이란?

MongoDB에서 **Document 들을 가공하고, 연산**하는 기능
RDBMS에서 SQL로 수행할 수 있는 기능들을 **유사하게 구현**할 수 있음

ex) SQL의 GROUP BY, DISTINCT, COUNT, JOIN 등

Aggregation 을 사용하는 이유

MongoDB의 find는 **검색 필터링과 정렬** 이외의 기능을 제공하지 않음
다른 Collection에서 데이터를 가져오거나,
검색된 **데이터를 그룹화**하는 등의 작업이 필요한 경우
Aggregation을 통해 이를 수행할 수 있음.

간단한 Aggregation 예제

aggregation

```
db.posts.aggregate([
  { $group: { _id: '$author', count: { $sum: 1 } } },
  { $match: { sum: { $gt: 10 } } },
  { $lookup: { from: 'users', localField: '_id', foreignField: '_id', as: 'users } },
]);
```

aggregation은 **Stage들의 배열**로 이루어지고 각 Stage는 **순차적으로 수행**됨

1. 작성자별 게시글 수를 취합하고
2. 게시글 수가 10개보다 많은 작성자를 찾아서
3. 해당 작성자를 회원 collection에서 검색 함

Aggregation Reference

Aggregation의 종류는 **너무 다양하고 복잡**하기 때문에
전부 외워서 사용할 필요는 없음
MongoDB 홈페이지에 **Stage들의 설명**과 **예제 코드까지 잘 정리**되어 있음

https://docs.mongodb.com/manual/meta/aggregation-quick-reference/#stages

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
최규범

강사
최규범

감수자
최규범

디자이너
강혜정

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io