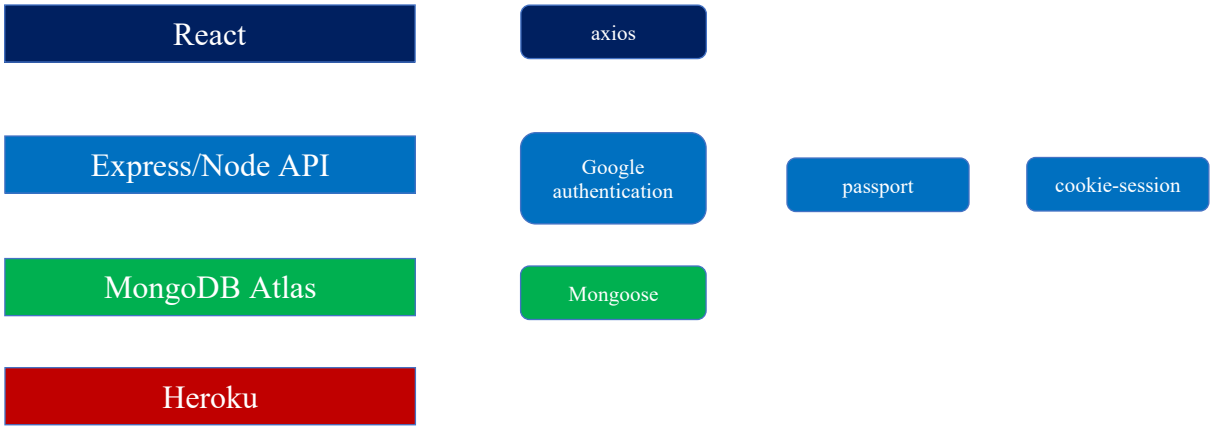
## Tech Stack




참조: https://github.com/StephenGrider/FullstackReactCode

# Lecture 1: setup – node/express & Google Authentication

## Making Node project

**node 프로젝트 만들기**

```
mkdir yu-nemr-air && cd yu-nemr-air
npm init -y
yarn add express
```

**index.js 생성**

yu-nemr-air/index.js

```
const express = require("express")

const app = express()

app.get('/', (req, res)=> {
   res.send({hi: 'there'})
})

const PORT = process.env.PORT || 5000
app.listen(PORT)
```

**package.json 수정**

다음을 main 밑에 추가

```
"engines": {
   "node": "16.8.0",
   "npm": "7.21.0"
 },
```

node, npm version number 는 여러분의 pc 에 설치된 version 을 써넣어야 함.

node -v  npm -v  를 통해 확인할 것

scripts 안에 다음 추가

```
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1",
   "start":"node index.js"
 },
```

**.gitignore 생성**

/node_modules 를 넣을 것

**실행**

yarn run start

# Heroku Deployment

**(1) Create an Heroku account**

Heroku account 를  우선 생성.

**(2) git commit**

server directory 에서…

```
git init
git add .
git config --global user.email "jisooyu@hotmail.com"
git commit -m “initial commit”
```

**(3) install Heroku CLI**

command line interface 를 설치함

and install the Heroku CLI by typing:

```
sudo snap install --classic heroku
```

for Mac: brew install heroku

**(4) Heroku signup/login**

signup site 에서 sign up

https://signup.heroku.com/

터미널에서 typing:

```
heroku login
```

일단 브라우저로 갔다가 로그인이 완료 되면 다시 터미널로 돌아 옴.

**(5) create Heroku app**
heroku create yu-nemr-air

**(6) Deploy git**
git remote add heroku https://git.heroku.com/yu-nemr-prod.git

git push heroku main

**(6-1) remove heroku**
git remote -v
git remote rm heroku
heroku git:remote -a yu-nemr-prod

**(7) execute the app**
heroku open

(8) subsequent deploy
to check the current status
git status

git add .
git commit -m “another commit”
git push Heroku main

## github repo 와 branch 만들기

- github 에서 repository 만들기

1. github 로 가서 처음이면 계정 생성. 계정이 있으면 로그인
2. “Repositories” click
3. “New” click
4. Repository 이름 입력 (예: my-repo)
5. terminal 로 돌아와서
6. git init
7. git add .
8. git commit -m “my first commit”
9. git remote add origin https://github.com/yourGithubUserName/my-repo.git
10. git branch -M main
11. git push origin main

- branch 만들기

1. git checkout -b yourbranch
2. 프로그램 내용 변경
3. git add .
4. git commit -m "some changes"
5. git push --set-upstream origin yourbranch

6. git checkout main
7. git merge yourbranch
8. git push origin main

## creating google project & application for development & production

1. Go to the Google Project Dashboard:

https://console.cloud.google.com

2. Click the "Create Project" button

3. Name the project and click the "Create" button

4. Click the menu button(Google Cloud Platform 왼쪽에 있는 평행선 3 개 모양), Select "APIs & Services", then "OAuth Consent Screen"

5. Select "External" and click the "Create" button

6. Fill out "Application Name", scroll to the bottom and click the "Save" button. No other info should be filled out in the consent screen at this time.

7. Click "Credentials" in the sidebar and then click the "Create Credentials" button

8. Select "OAuth Client ID"

9. Select Web Application and fill out the "Authorized JavaScript Origins"(http://localhost:5000) and "Authorized redirect URI"( http://localhost:5000/auth/google/callback) and click the "Create" button.

Note! Google has made a number of changes to the OAuth credential's restrictions, and no longer allows wildcards in the redirect URI field. If you follow along with the upcoming video lecture you will get this error: Invalid Redirect: cannot contain a wildcard (*)

Since the main goal of using http://localhost:5000/* was to show the redirect error a few lectures later, we entered the correct redirect as shown above since this is what it will be changed to anyway.

11. Copy your Client ID and Client Secret in a safe place so you can use it in your application in a future lecture. (ID and Secret were redacted from the screenshot)

**development 용**
project name: yu-nemr -air-dev
client ID:

**production 용**
project name: yu-nemr-air

# Lecture 2: authentication & passport middleware

## passport & passport-google-oauth20 설치

passport 와 passport strategy 는 authentication 프로세스를 용이하게 하는 library

```
yarn add passport passport-google-oauth20
```

client ID:

config/keys.js

```
module.exports = {
  googleClientID: '',
  googleClientSecret: ''
}
```

.gitignore

```
/config/keys.js
```

index.js

```
const express = require('express');
// const keys = require('./config/keys')

require('./services/passport');

const app = express();

require('./routes/authRoutes')(app);

const PORT = process.env.PORT || 5000;
app.listen(PORT);
```

services/passport.js

```
// passport 에 Google Strategy 를 등록
// passportjs.org 로 가서 strategy 열람(passport-facebook, passport-naver etc)

const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;
const keys = require('../config/keys');

passport.use(
new GoogleStrategy(
        {
                clientID: keys.googleClientID,
                clientSecret: keys.googleClientSecret,
                callbackURL: '/auth/google/callback'
        },
        // done 의 의미는 passport 에게 우리 일은 여기서 끝났다는 것을 알려주는 것
        (accessToken, refreshToken, profile, done)=> {
                console.log(profile)
        }
        )
);
```

routes/authRoutes.js

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

module.exports = app => {
  // request type 이 'get'이고 '/auth/google'로 들어오면 passport 가 처리하도록 함.
  app.get(
    '/auth/google',
    // GoogleStrategy 내에 internal identifier 로 'google' string 이 있음.
    // 그래서 여기서 'google'을 넣으면 Google Strategy 라는 것이 인지됨
    passport.authenticate('google', {
      // scope 에서 contact list, image 등을 요구 할 수도 있음. 여기서는 예로써 'profile', 'email'을 포함시킴 것임.
      scope: ['profile', 'email']
    })
  );

  app.get('/auth/google/callback', passport.authenticate('google'));
};
```

**nodemon 설치**

(**) d flag 를 사용하여 설치할 것. heroku 에 nodemon 을 설치할 필요가 없기때문에…

```
yarn add -d nodemon
npm i -D nodemon
```

in package.json

```
  "scripts": {
    "start": "node index.js",
    "dev":"nodemon index.js"
  },
```

**실행**

1. yarn run dev
2. localhost:5000/auth/google

# Lecture 3: Mongo DB Atlas, Mongoose

## mongo DB Atlas 계정 생성

**source: Stephen Grider, Node with React FullStack Web Development, Udemy lecture.**

MongoDB Atlas Production Setup and Configuration
***updated 6-10-2020***

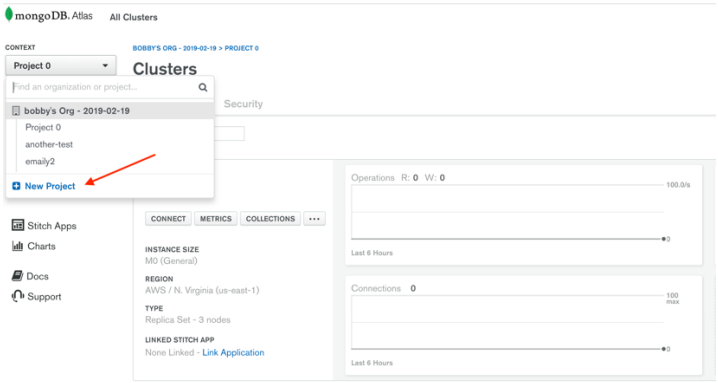
This note is for students who are using MongoDB Atlas per the course lecture note here:

https://www.udemy.com/node-with-react-fullstack-web-development/learn/v4/t/lecture/13733504

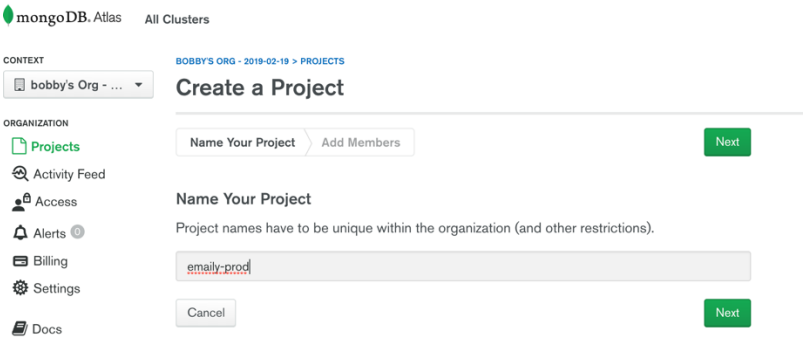
In the upcoming lecture, we are going to be creating a separate production database for use with Heroku. Here are the steps required to do this with MongoDB Atlas. (The most important part is the global whitelisting in step 9)

**1. Sign in to your MongoDB account**

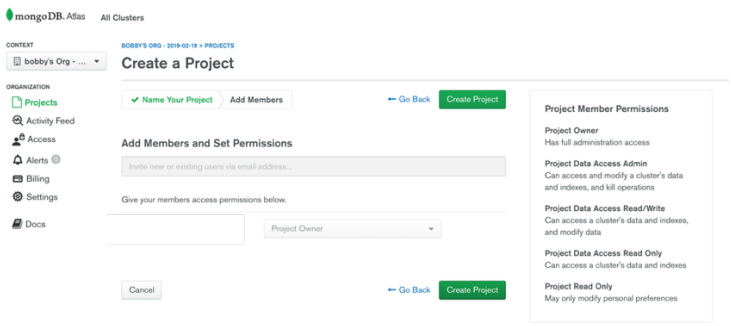
**2. Once logged in, you will be taken to your MongoDB Atlas Dashboard. Click the current project button listed under "Context" and click the "New Project" link. As a free tier user, you can have only one cluster per project, however, you can have as many projects as you wish.**



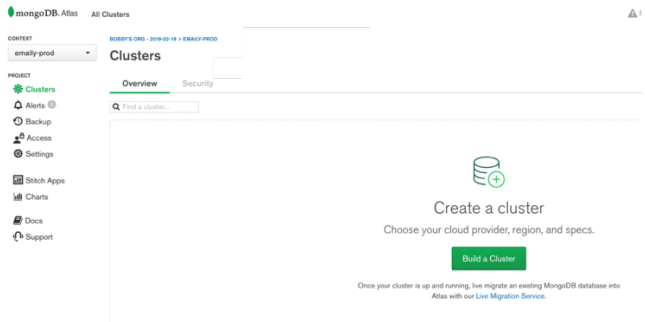

**3. Name your project and then click the "Next" Button:**




**4. Click the "Create Project" button:**

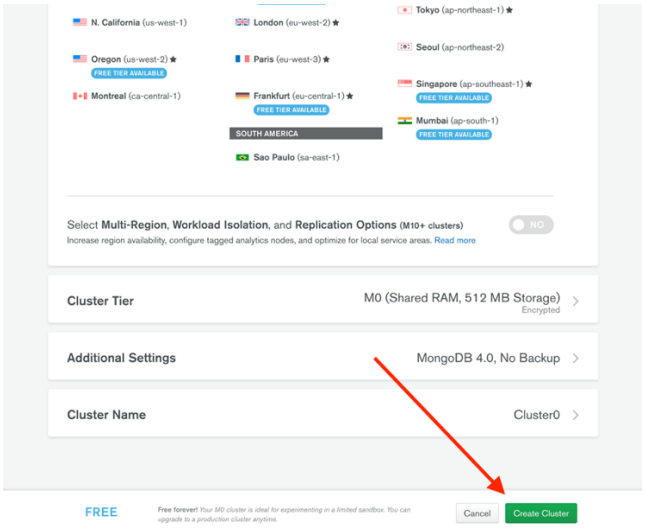


**5. After the project finishes initializing, you will be taken to the "Clusters" page of your Database. Click the "Build a Cluster" button**

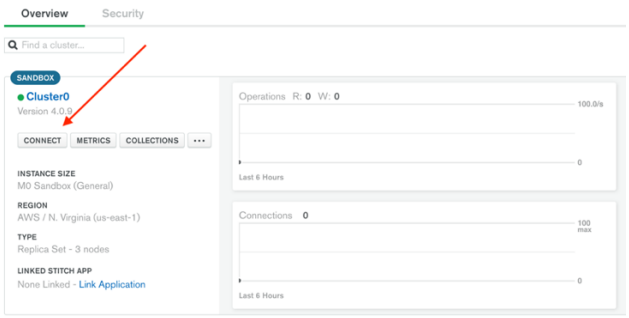



**6. Leave all of the Free Tier options selected and scroll down to the bottom. Click the "Create a Cluster" button:**

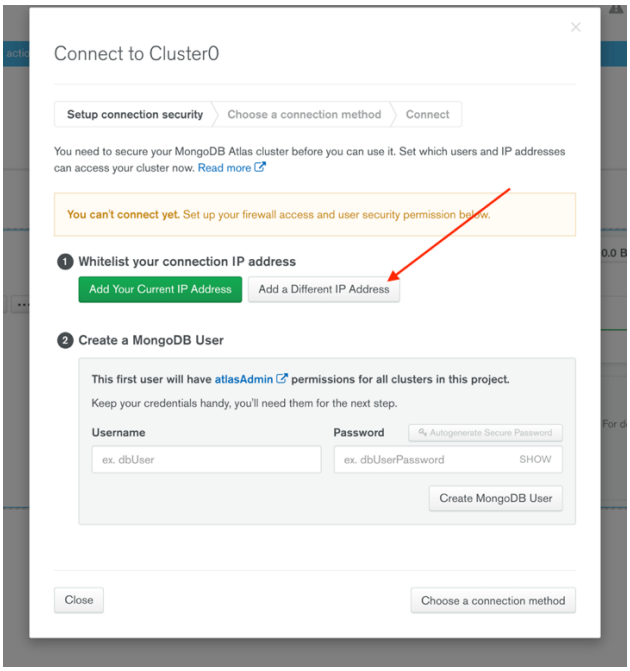
**"Create a Cluster"**를 누르기 전에 필요하다면 Cluster Name 으로 가서 이름을 변경할 것




**7. After the Cluster has been created, you will be taken back to the Clusters page of the database. Click the "Connect" button**



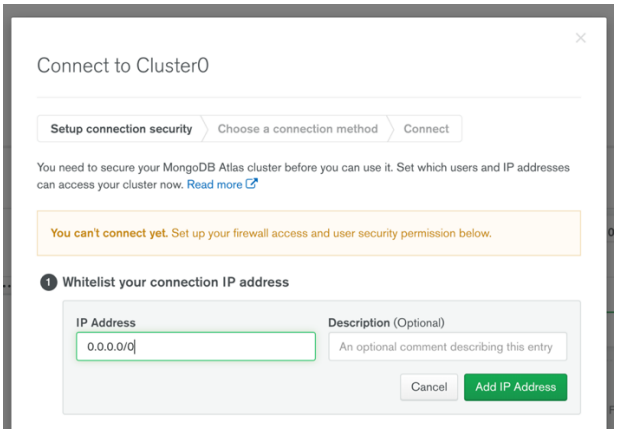

**8. Click the "Add a Different IP Address button:**



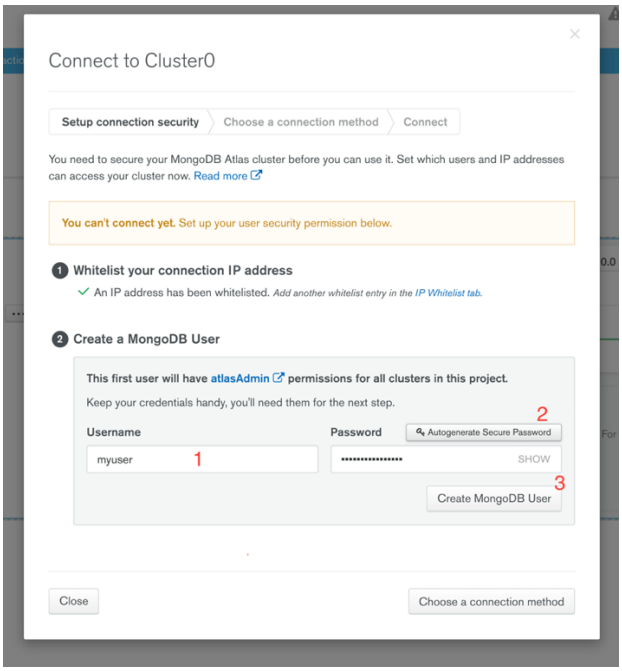

**9. Enter 0.0.0.0/0 into the IP Address field and click the "Add IP Address" button. This part is extremely important as our Heroku server will not be able to connect to our database unless we whitelist all IP addresses.**

**In a real production app, you would typically have a static IP and a Fully Qualified Domain Name. In this case, we would whitelist only the static IP. You can read up on this more here:**

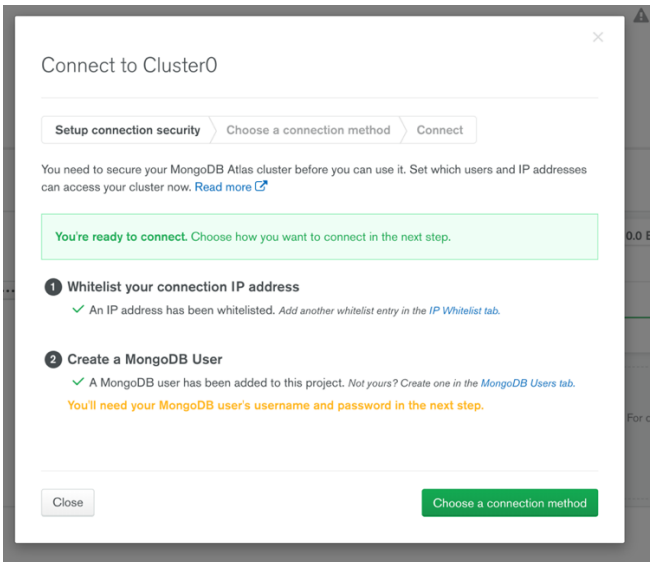
**https://help.heroku.com/JS13Y78I/i-need-to-whitelist-heroku-dynos-what-are-ip-address-ranges-in-use-at-heroku**



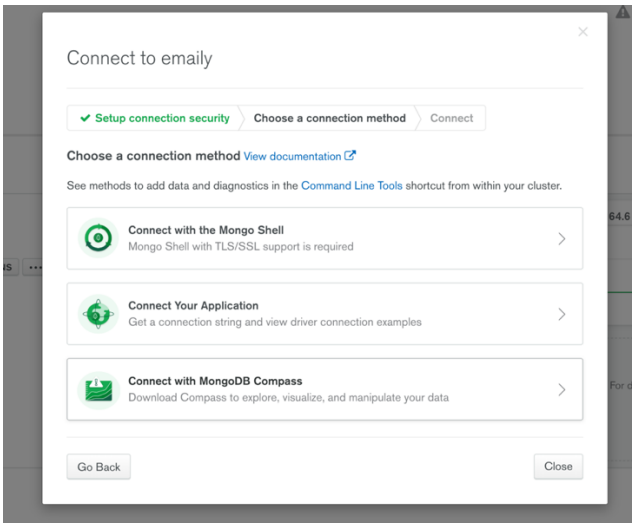

**10. Next, enter a new username and then click the "Autogenerate Secure Password" button. Then click "Create MongoDB User"**




**11. After the user is created, click the "Choose a Connection Method" button:**




**12. Select "Connect Your Application"**




**13. Copy the address under "Connection String Only"**

**When you paste this string into the Heroku Dashboard, you will need to replace <username> and <password> with the actual info created earlier and swap out the <dbname> placeholder with any arbitrary name.**

**14. Click the "Close" button and head back over to your application.**

## MongoDB Atlas User Password 설정

1. 왼쪽 메뉴 -> Database Access 클릭 -> 맨 오른쪽에 EDIT 클릭 -> Edit Password 클릭 -> Autogenerate Secure Password 클릭 -> Copy 클릭
2. 복사한 password 를 mongoURI 의 <password>에 paste 할 것

## App 변경

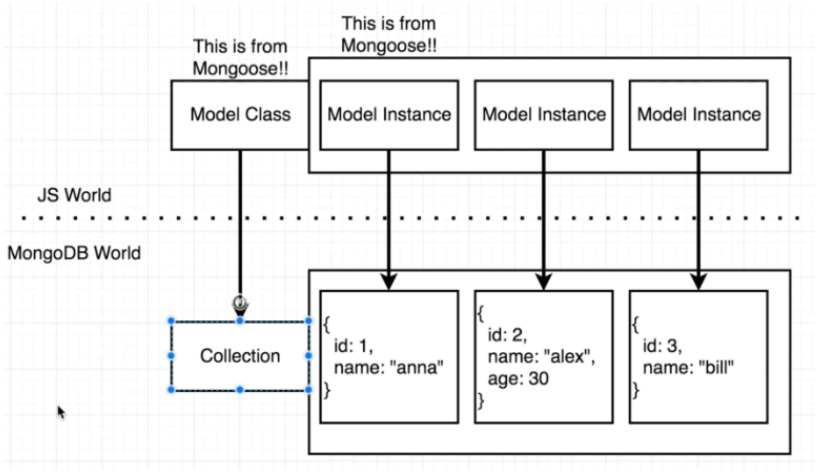
config/keys.js

```
module.exports = {
  googleClientID: '',
  googleClientSecret: '',
  mongoURI: ''
}
```

## mongoose 설치

yarn add mongoose

## mongoose 기능



source: Stephen Grider, React and Node Fullstack Web Development, Udemy lecture.

models/User.js
```
const mongoose = require('mongoose');
const { Schema } = mongoose;

const userSchema = new Schema({
        googleId: String,
});

// create a new collection, users
mongoose.model('users', userSchema);
```

**mongoDB 에서는 왜 schema 를 정의해야 하는 가?**
mongo db 는 각 record 에 다른 fields 혹은 properties 를 허용.
(e.g.) 레코드 1 : {name: jfkaj, age: 23} 레코드 2: {name: djkflaj, weight: 73}

따라서 schema 에 모든 properties 를 기재해야 함.

index.js
```
const express = require('express');
const mongoose = require('mongoose')
const keys = require('./config/keys')

// User 를 passport 보다 먼저 import 해야 함
require('./models/User')
require('./services/passport');

mongoose.connect(keys.mongoURI)

const app = express();

require('./routes/authRoutes')(app);

const PORT = process.env.PORT || 5000;
app.listen(PORT);
```

services/passport.js

```
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;
const mongoose = require('mongoose')
const keys = require('../config/keys');

const User = mongoose.model('users')

passport.use(
new GoogleStrategy(
        {
                clientID: keys.googleClientID,
                clientSecret: keys.googleClientSecret,
                callbackURL: '/auth/google/callback'
        },
        // done 의 의미는 passport 에게 우리 일은 여기서 끝났다는 것을 알려주는 것
        (accessToken, refreshToken, profile, done)=> {
                // database(yu-react-mongo)의 collection(users)에 google id 를 저장
                // 그러나 추후 동일한 user 가 두번 저장되지 않도록 해야 함
                new User({googleId: profile.id}).save()
        }
        )
);
```

routes/authRoutes.js

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

module.exports = app => {
  // request type 이 'get'이고 '/auth/google'로 들어오면 passport 가 처리하도록 함.
  app.get(
    '/auth/google',
    // GoogleStrategy 내에 internal identifier 로 'google' string 이 있음.
    // 그래서 여기서 'google'을 넣으면 Google Strategy 라는 것이 인지됨
    passport.authenticate('google', {
      // scope 에서 contact list, image 등을 요구 할 수도 있음. 여기서는 예로써 'profile', 'email'을 포함시킴 것임.
      scope: ['profile', 'email']
    })
  );

  app.get('/auth/google/callback', passport.authenticate('google'));
};
```

문제: 같은 google profile id 를 갖은 user record 가 중복으로 mongo db 에 저장됨.
해결: services/passport.js 를 수정

```
User.findOne({googleId: profile.id}).then(existingUser => {
        if (existingUser){
          done(null, existingUser)
        } else {
          new User({googleId: profile.id})
          .save()
          .then(user => done(null, user))
        }
```

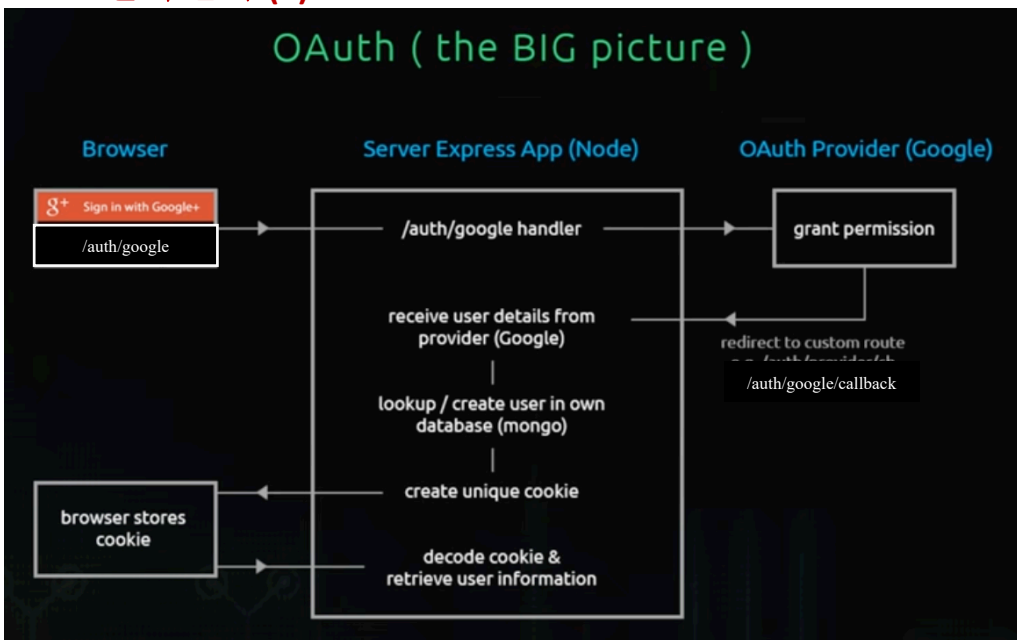
```
})
```

**실행**

1. yarn run dev
2. localhost:5000/auth/google
3. Mongo Atlas 로 가서 user record 가 생긴 것 확인

# Lecture 4: Processes after the authentication

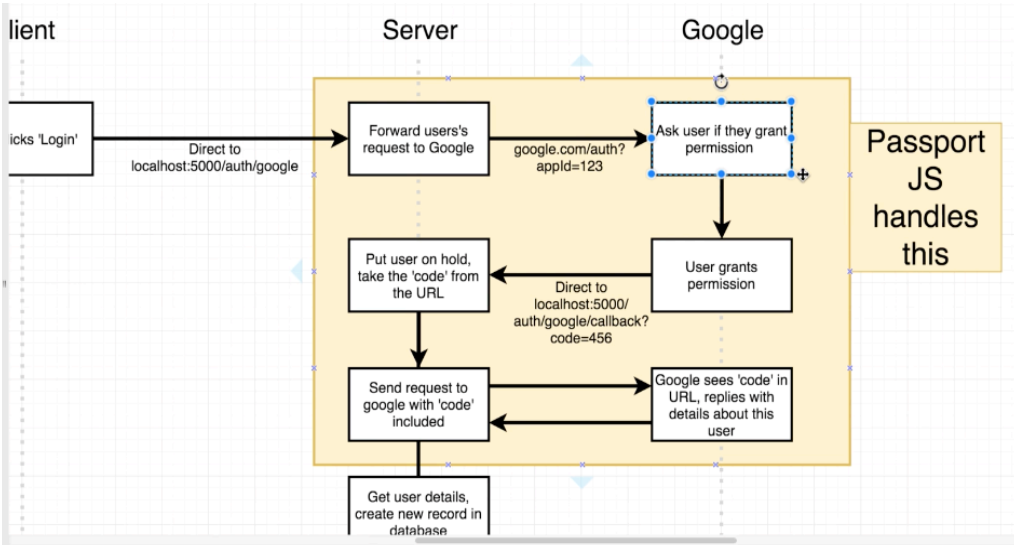
## 1. 전체 윤곽 (1)




source: OAuth Login (Passport.js) Tutorial #2 - The OAuth Flow, https://www.youtube.com/watch?v=CHodPpqLqG8

cookie 의 사용은 database 에서 가져온 user id (mongo db 가 만든 것)로 session token 이 만들어 진 다음에 시작됨.

## 2. 윤곽(2): Authentication process 와 passport 역할




source: Stephen Grider, React and Node Fullstack Web Development, Udemy lecture.

routes/authRoutes.js

```
 app.get(
   '/auth/google',
   passport.authenticate('google', {
     scope: ['profile', 'email']
   })
 );
 app.get('/auth/google/callback', passport.authenticate('google'));
```

services/passport.js

```
passport.use(
new GoogleStrategy(
{
   clientID: keys.googleClientID,
   clientSecret: keys.googleClientSecret,
   callbackURL: '/auth/google/callback'
},

(accessToken, refreshToken, profile, done)=> {
      User.findOne({googleId: profile.id}).then(existingUser => {
         if (existingUser){
         done(null, existingUser)
         } else {
         new User({googleId: profile.id})
         .save()
         .then(user => done(null, user))
         }
      })
   }
   )
);
```

강의내용

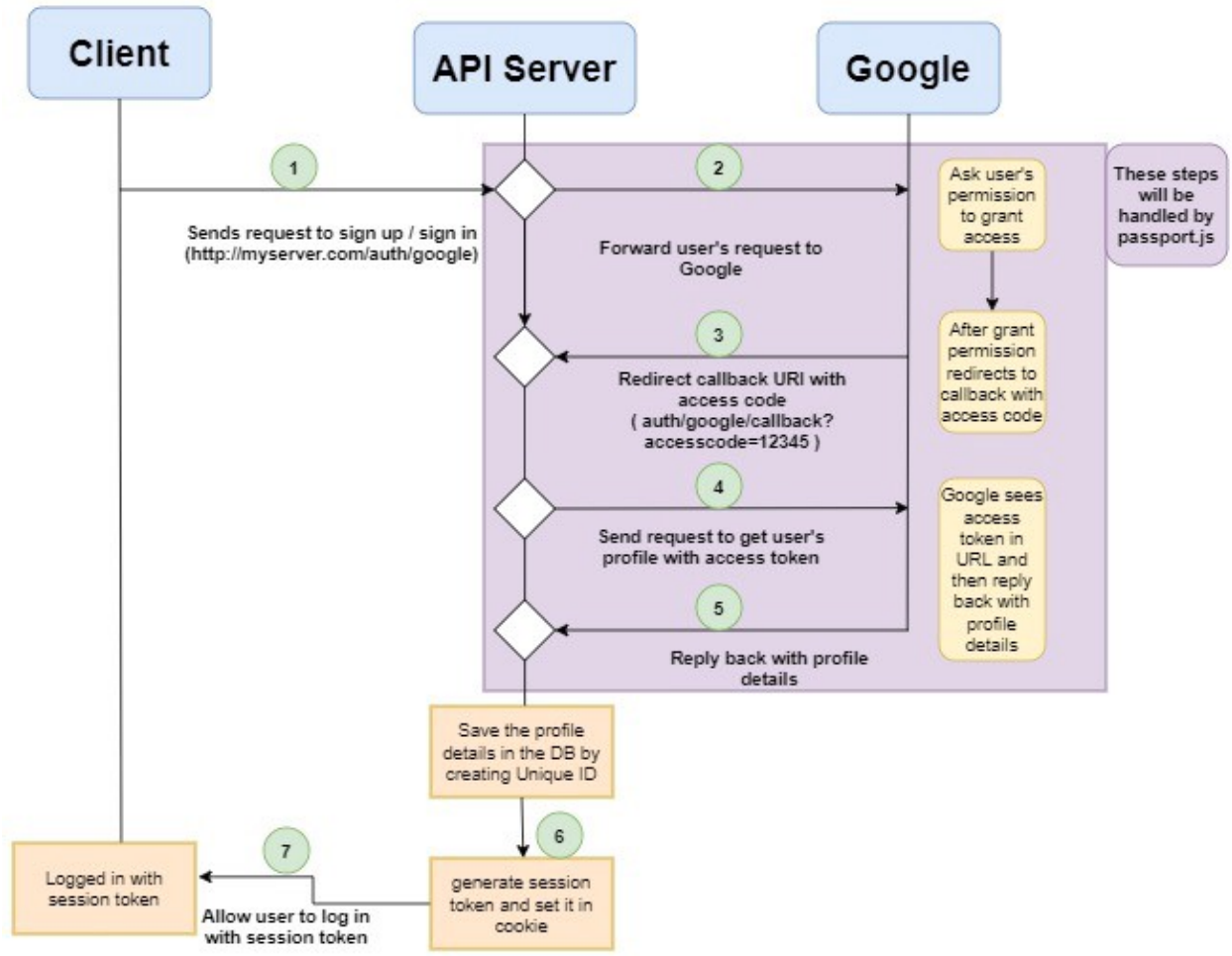
authentication 이후의 프로세스를 설명. credentials(client ID, client Secret, access code )를 제공하고 authentication 이 끝난 이후 취득한 google profile 로 client 와 server 간의 프로세스 설명

a. Authentication
   i. signup/sign in request (client ID)->구글이 response 로 node/express server 로 access code 를 보냄
   ii. node/express server 는 access code+client ID+client Secret 로 access token 을 request -> 구글이 access token 로 response
   iii. node/express server 가 access token 으로 profile request -> 구글이 profile 로 response(profile 을 보내 줌)

b. Post-authentication
   i. database 에 profile id 가 없으면 저장. 새로 user record 를 mongo db 에 저장. mongo db 가 user id 생성. 앞으로는 profile id 가 아니라 user id 로 user 확인.
   ii. mongo db 에서 불러낸 user id 를 serializeUser 에게 보냄
   iii. serializeUser 는 user id 가 들어가 있는 session token 을 만듬. passport 는 user id 를 req.session.passport 에 저장

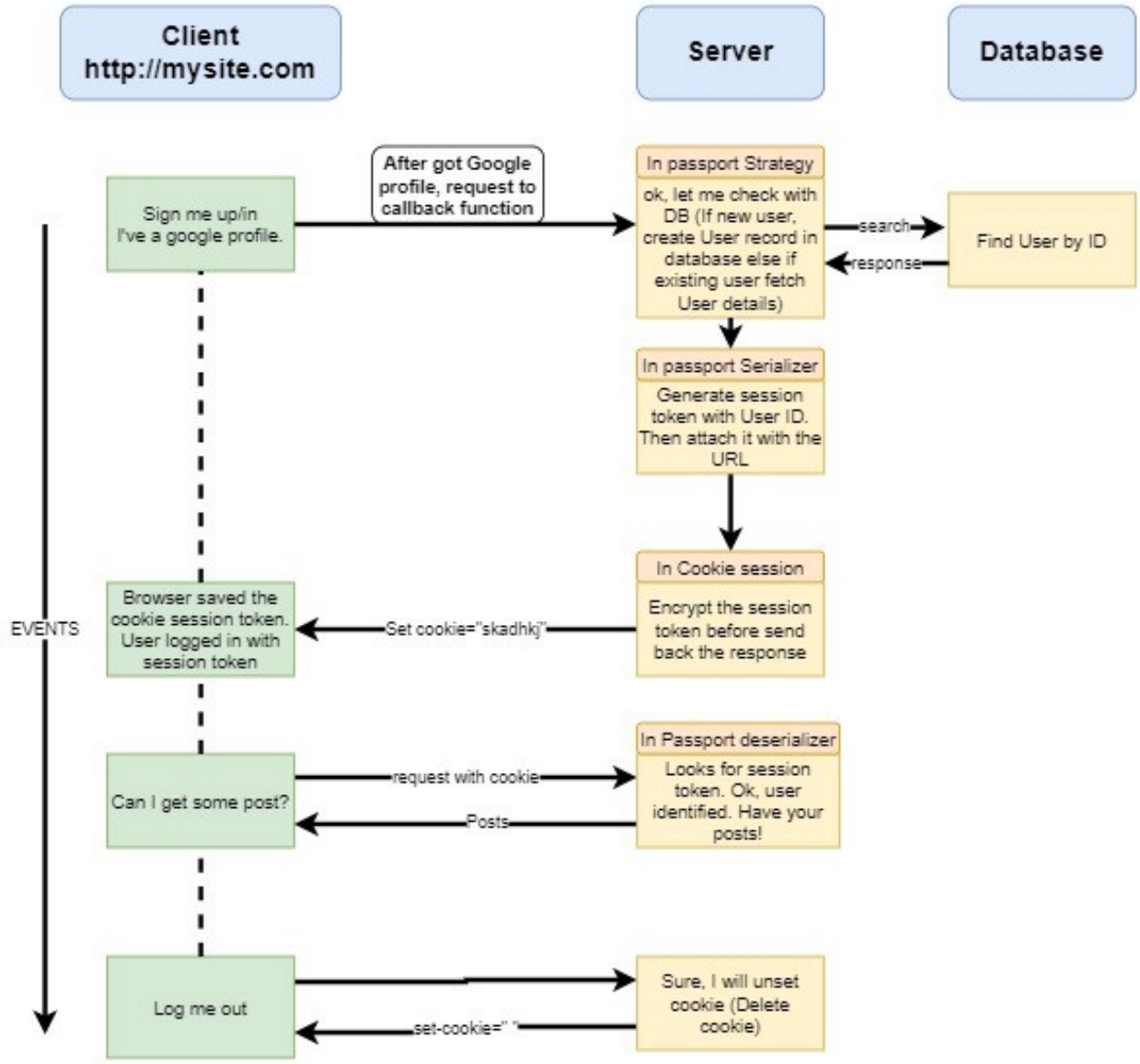
iv. cookie-session 은 session token 을 encrypt 하여 passport 에게 줌
v. passport 는 setCookie 로 session token 이 cookie 에 포함되도록 하고 browser 가 cookie 를 저장할 수 있게 함
vi. browser 가 cookie 를 browser 에 저장
vii. browser 가 이제 부터는 server 에 request 할 때 마다 cookie 를 보냄.
viii. browser 가 server 로 보낸 cookie 를 cookie-session 이 de-encrypt 하고 session token 을 추출하고 session token 에 있는 user object 를 deserializeUser 에 보냄.
ix. deserializeUser 는 user model instance 를 만들어 req.user 에 추가함.

## 3. 윤곽(3): Authentication process 와 passport 역할 상세




source: OAuth Authentication System- Behind the scenes,
https://medium.com/geekculture/oauth-authentication-system-behind-the-scenes-e0720a3c31ab

## 4. 윤곽(4): post authentication




source: OAuth Authentication System- Behind the scenes,
https://medium.com/geekculture/oauth-authentication-system-behind-the-scenes-e0720a3c31ab

services/passport.js

```
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;
const mongoose = require('mongoose')
const keys = require('../config/keys');

const User = mongoose.model('users')

// database 에서 불러온 user record(user)을 변환시키는 과정.
```

// passport.serializeUser method 는 database 에서 불러온 user record 로 부터 session token 을 만들어 냄. session token 은 user 를 식별할 수 있는 user.id(monogo db 가 부여한 record id)를 담고 있음.
// session token 은 req.user 와 req.session.passport.user 로 가게 됨
// passport 는 session token 을 browser 에게 보내면서 cookie 에 담을 것을 browser 에게 요청(setCookies:'session token'). 이후 부터 browser 는 server 에 request 를 보낼 때 cookie 를 첨부

```
passport.serializeUser((user, done)=> { // 첫번째 argument 인 user 는 database 에서 찾아온 user record 임.
  done(null, user.id)
})
```

// browser 가 session token 을 cookie 에 담아서 이 cookie 를 server 에 보내어 user data 를 database 에서 불러달라고 요청.
// session token 에 있는 user data(user id)을 passport.deserializeUser 는 받게 됨. 이 user data 를 사용하여 database 에 있는 user 의 모든 정보를 받게 됨. 우리의 경우는 database 에 id 와 google id 만이 있음. 이를 모두 받게 됨.

```
  // Whatever gets passed to done here will be available in req.user.
passport.deserializeUser((id, done) =>{
  User.findById(id)
    .then(user => {
      done(null, user)
    })
})

passport.use(
  new GoogleStrategy(
// authentication process I: passport.authenticate()이 호출됨.
    {
      clientID: keys.googleClientID,
      clientSecret: keys.googleClientSecret,
      callbackURL: '/auth/google/callback'
    },
          // authentication process II: passport strategy 는 I 에서 받은 temp code 를 갖고 OAuth token 과 profile 을 요청

    (accessToken, refreshToken, profile, done)=> {

          // profile id 를 이용하여 database 에 있는 user model(user record)을 불러냄. 이 user record 는 serializeUser 의 첫번째 argument 로 사용됨

      User.findOne({googleId: profile.id}).then(existingUser => {
        if (existingUser){
          done(null, existingUser)
        } else {
          new User({googleId: profile.id})
          .save()
          .then(user => done(null, user))
        }
      })
    }
  )
);
```

cookie-session library 설치

- add cookie-session
  - The express doesn't know how to manage the cookie. It needs cookie-session
- install cookie-session
  - yarn add cookie-session

index.js

```
// express 는 cookie 로 어떻게 할 지를 모름. index.js 파일에 cookie 를 만들어서 express 가 cookie 를 알게 하는 것이 최상.
const express = require('express');
const mongoose = require('mongoose')
// cookie 에 access 하는 데 cookie-session 이 필요
const cookieSession = require('cookie-session')
// Passport 에게 cookie 를 사용하여 user status, authentication status 를 추적하도록 지시해야 함.
const passport = require('passport')      s
const keys = require('./config/keys');
const { cookieKey } = require('./config/keys');

// User 를 passport 보다 먼저 import 해야 함
require('./models/User')
require('./services/passport');

mongoose.connect(keys.mongoURI)

const app = express();

// register cookieSession
// configuration 하는 데 2 개가 필요: cookie 의 최대 유효기간, cookie keys
app.use(
  cookieSession({
    // expiration dates of cookie: 2 days
    maxAge: 2 * 24 * 60 * 60 * 1000,
    keys:[keys.cookieKey] // multiple keys are allowed. cookie 를 encrypt 하는 데 사용함.
  })
)

/* Connect or Express-based application 에서는 passport.initialize() middleware 가 초기화 되어야 함.
authentication 이 완료되면 session 이 만들어지고 session 은 browser 에 있는 cookie set 을 통해 유지됨.
passport.session()은 middleware 로써 req object 를 변경하고 user value 를 deserialized user object 로 변경함.

// passport 가 cookie 를 사용하여 authentication 을 하도록 하게 하기 위해.. passport.initialize(), passport.session()
app.use(passport.initialize())
app.use(passport.session())

require('./routes/authRoutes')(app);

const PORT = process.env.PORT || 5000;
app.listen(PORT);
```

config/keys.js

```
module.exports = {
   googleClientID: '',
   googleClientSecret:'',
   mongoURI: '',
   cookieKey: ''
 };
```

## testing authentication

change the authRoutes.js

routes/authRoutes.js

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

module.exports = app => {
 app.get(
  '/auth/google',
  passport.authenticate('google', {
   scope: ['profile', 'email']
  })
 );

 app.get('/auth/google/callback', passport.authenticate('google'));

 // logout()을 부르면 cookie 안에 있는 user.id 를 없앰. 결국 user 는 더 이상 request 를 실행할 수 없음
 app.get('/api/logout', (req, res)=> {
  req.logout()
  res.redirect('/');
 })

 app.get('/api/current_user', (req, res) => {
  res.send(req.user);
 });
};
```

what does app.use() do?

app.use is wiring up the middleware. The middleware is the small function that can be used to modify the incoming requests to our web before they are sent off to their route handlers.

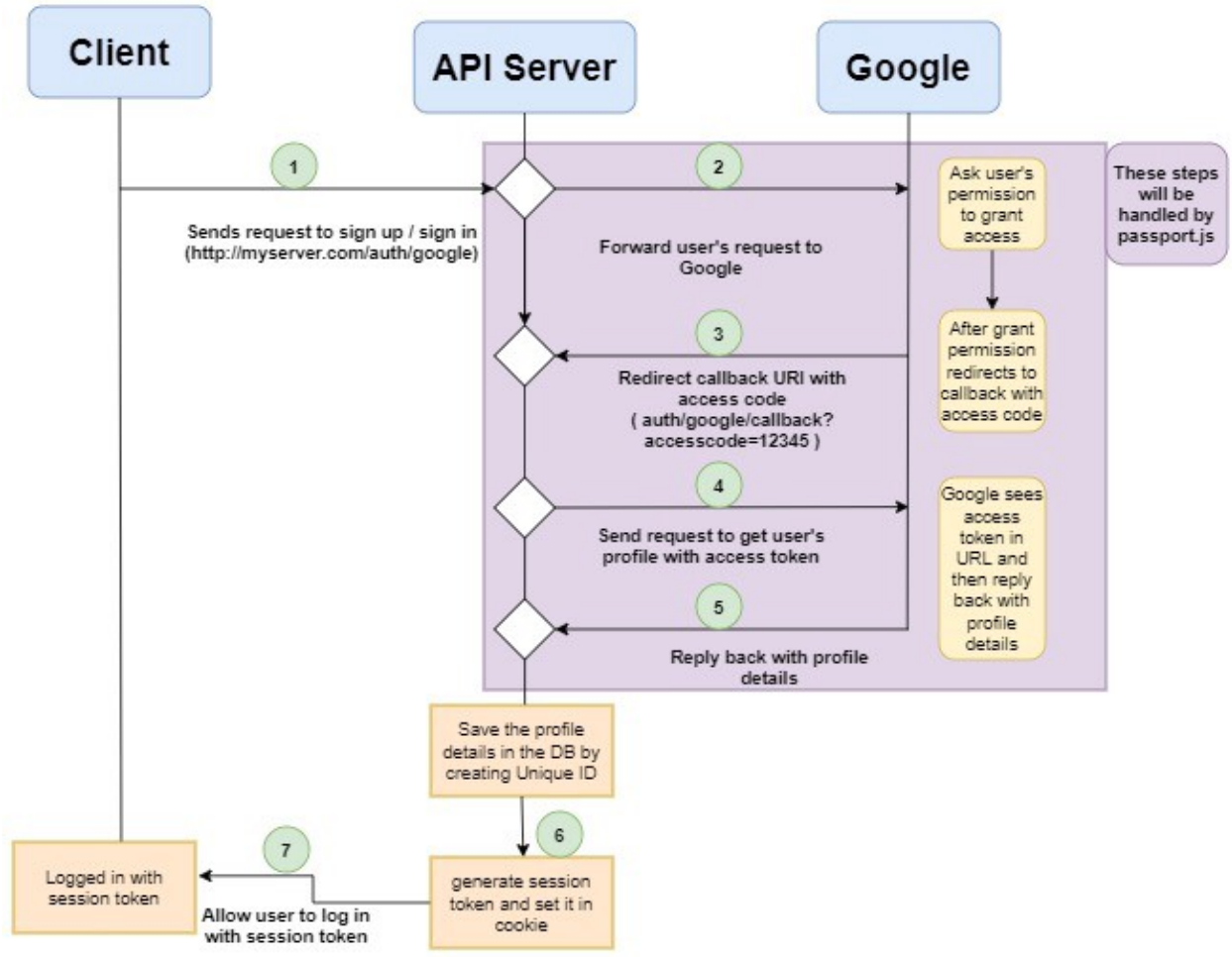
# Lecture 5: Theory: session, passport , serializeUser, deserializeUser, and cookie

recap

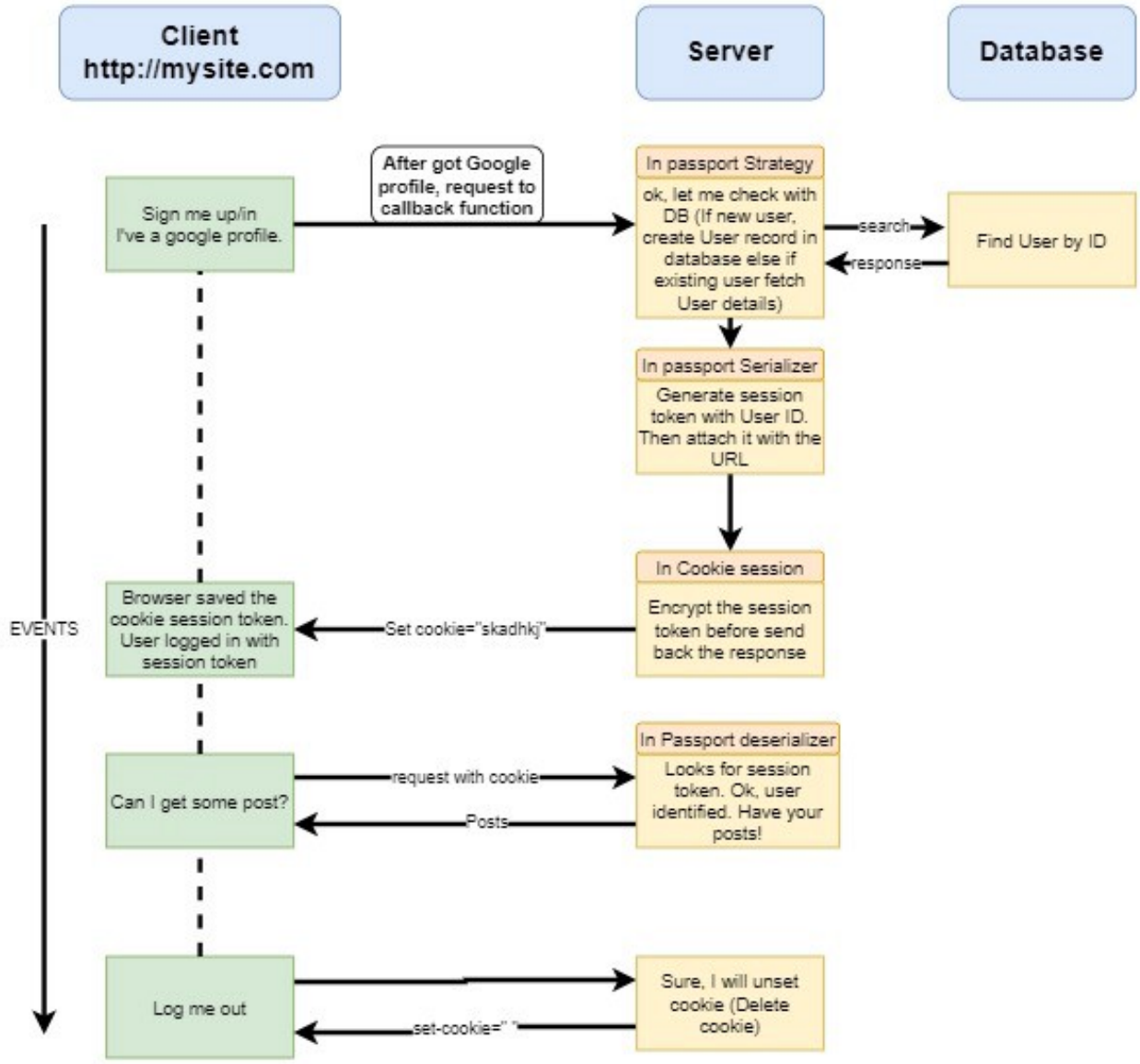
1. express/node js 만들
2. google project 생성 및 OAuth client ID 생성
    a. 승인된 자바스크립트 원본에 URI 입력 (예: /auth/google 혹은 /google 등등)
    b. 승인된 리디렉션 URI 입력 (예: /auth/google/callback 혹은 /google/callback 등들)
3. passport 사용
    a. passport 는 middleware

b. passport strategy 는 여러 strategy 를 보유(twitter, facebook, naver, github, etc)하고 있어서 다양한 login 가능

## passport의 역할




source: OAuth Authentication System- Behind the scenes,
https://medium.com/geekculture/oauth-authentication-system-behind-the-scenes-e0720a3c31ab




source: OAuth Authentication System- Behind the scenes,
https://medium.com/geekculture/oauth-authentication-system-behind-the-scenes-e0720a3c31ab

- **cookie 확인**

  1. browser 의 inspect open, localhost:5000/auth/google 로 감. response header 의 set cookie 를 볼 것

  2. new browser tab 을 열고, inspect open, localhost:5000/auth/google/callback 로 감. request header 의 cookie 를 볼 것.

  https://codeburst.io/how-do-express-sessions-work-c465e0488af4 도 읽어 볼 것.

- **session 은 무엇인가?**
  - Http protocol 은 stateless. 따라서 web browser page 를 refresh 할 때 마다 다시 authentication(email, pw 입력)을 실행해야 하는 불편. 마찬가지로 browser 의 다른 page(tab)을 열 때 마다 authentication 을 해야 하는 불 편. 이 문제를 해결하기 위해 session 을 사용.

- session 은 request 할 때 사용할 데이터를 저장하는 장소.
- websites(e.g. Google, Twitter, Instagram, Naver, etc)를 방문할 때 사용자에게 unique session 이 부여됨.
- 일반적으로 web application 에서는 login request 을 사용자가 할 때 credentials(email, password, phone number, etc)보내어 authentication 을 하게 됨. authentication 이 성공적으로 완료되면 session 이 만들어지고 (session 은 cookie 에 설정이 됨).
- 이후로는 다시 credential 을 이용한 authentication 을 하지 않고 session 을 식별하는 cookie 를 통해 사용자의 login state(상태)를 확인하는 것임.
- Login state 를 보여주는 login session 을 지원하기 위해 passport 는 session 의 user instance 를 serialize & deserialize 를 하여 session 과 주고 받음. .

source: https://codeburst.io/how-do-express-sessions-work-c465e0488af4
source: http://www.passportjs.org/docs/configure/

- **session 을 저장하는 장소**
  - In application memory
  - In a cookie
  - In a memory cache
  - In a database

  source: https://codeburst.io/how-do-express-sessions-work-c465e0488af4

- **cookie 와 req, session, passport 의 관계**

눈으로 확인 하려면…authRoutes.js 에서 다음을 변경..

```
// deserializeUser 가 보낸 req.user
app.get('/api/current_user', (req, res) => {
  res.send(req.user)
});
// serializeUser 가 보낸 req.session.passport
app.get('/api/session/passport', (req, res)=> {
  res.send(req.session.passport)
})
// cookie-session 이 de-encrypt 하여 넘겨 준 req.session
app.get('/api/session', (req, res)=>{
  res.send(req.session)
})
```

으로 변경하고 localhost:5000/api/session 로 가볼 것. 여기 req.session 에는 passport 와 id 가 있는 것을 알 수 있음. 이 user id 는 passport 가 serializeUser()에서 받은 것임. 받아서 req.session.passport 에 저장.

{"passport": {"user": "5gxa2ab2a0bad26b004b602e"}}

- **passport, session, cookie, serializeUser, and deserializeUser**

express 는 cookie 를 갖고 무엇을 해야 하는 지 알지를 못함. 그래서 cookie-session 이라는 helper function 을 사용하여 express 가 cookie 를 사용하도록 함.

passport 가 cookie 를 사용하여 user session, user authentication state 를 keep track 하도록 지시해야 함.

그렇게 하려면 우선 cookie session 을 사용하여 cookie 를 handle 하도록 해야 함. cookie-session 이 cookie 를 access 하는 것임. cookie session 을 사용하려면 라이브러리를 설치해야 함.

1. google profile 을 얻은 후에 browser 로 부터 request 가 들어오면 우선 new user 인지를 확인하고 new user 이면 database 에 user record 를 생성. new user 가 아니면 profile id 로 database 에서 user id 를 불러냄
2. user id 를 serializeUser 로 보내어 session token 을 만듬. session token 에는 물론 user 를 확인 하는 user.id 가 들어가 있음.
3. cookie-session 이 session token 을 encrytion 하여 passport 에 주고 passport 는 cookieSet: 'session token'을 통해 browser 가 request header 에 session token 을 넣어서 cookie 를 만들 것을 지시
4. browser 는 passport 가 지시한 대로 cookie 를 만들어 저장. cookie 에 user id 가 있게 됨.
5. browser 는 cookie 와 함께 request 를 server 로 보냄
6. browser 로 부터 cookie 와 함께 request 가 들어오면 server 에서는
7. cookie-session 이 cookie 에서 user info 를 extract 하여
8. 이를 de-encrypt 하여 passport 에 넘겨줌. (구체적으로 말하면 cookie-session 은 req.session 에 user id 를 넣어서 이를 passport 에 넘겨주는 것임.)
9. passport 는 이를 deserializeUser 에게 보냄. deserializeUser 는 passport 가 준 user id 가 database 에 있는 지를 확인하고 있으면 user model instance(user 의 complete info)를 만듬. 그리고 이를 req.user 에 추가.
10. req.user 는 client 의 request 에 부합하는 express router 에 보내져 request 를 처리 하는 데 사용됨.

**PassportJS — The Confusing Parts Explained**
https://hackernoon.com/passportjs-the-confusing-parts-explained-edca874ebead

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});          |
             |
             |
             └──────────────┬──→ saved to session
                            |   req.session.passport.user = {id: '..'}
                            |
                            ↓
passport.deserializeUser(function(id, done) {
             ┌──────────────┘
             |
             ↓
  User.findById(id, function(err, user) {
    done(err, user);
  });        └──────────────→ user object attaches to the request as req.user
});
```

source: https://stackoverflow.com/questions/27637609/understanding-passport-serialize-deserialize?answertab=votes#tab-top

**serializeUser**

- authentication 을 한 후 google profile 을 받으면 mongo db 에 저장
  - authRoutes.js 에서 strategy 를 세팅할 때 done()를 callback 으로 포함시켜서 user profile object 를 done()에 pass 하였음..

    authRoutes.js 에서 다음 부분을 볼 것. existingUser 혹은 user 라는 profile object 를 pass.

    (accessToken, refreshToken, profile, **done**) => {

```
User.findOne({googleId: profile.id}).then(existingUser => {
  if (existingUser){
    done(null, existingUser)
  } else {
    new User({googleId: profile.id})
    .save()
    .then(user => done(null, user))
  }
})
```

- 위의 과정을 하는 이유는 위에서 pass 한 user 가 serializeUser()로 가게 하기 위한 것임. serializeUser()는 done(null, user.id)를 불러서 user.id 를 확보.
- passport 는 user id 를 받아서 req.session.passport 에 저장함.
- passport 는 req.session.passport 를 사용하여 현재 상황을 추적함.

**deserialUser**

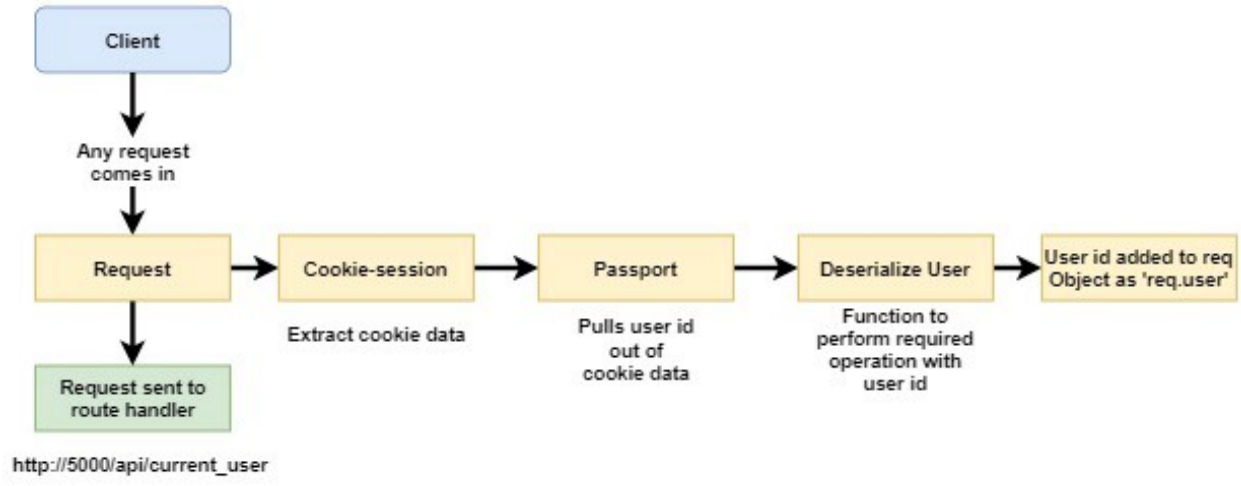
- browser 에서 request 가 들어오면(cookie 가 들어오면) cookie 안의 session token 에서 user id 를 추출하여 deserializeUser()로 가게되면 database 에서 user 의 모든 정보(user full profile info)를 찾어내어 done(null, user)를 호출. done()은 user profile 을 req.user 에 첨부함.

- **cookie-session 과 passport 의 프로세스**

1. cookie-session: 필요한 id 를 extract 하여 req.session 에 넣어서, 이를 passport 에게 전달
2. passport 는 deserializeUser 를 불러서 req.session 에 있는 user id 로 db 에 user 정보를 불러내어 user object 를 만듬. user object 에는 user 의 full profile info 가 포함되어 있음. 이 user object 를 req.user 에 첨부

## 이런 복잡한 절차가 필요?

http protocol 은 stateless. 방문자를 기억하지 못함. request, response 를 시행하면 모든 것을 잊어 먹음. 그래서 code, cookie, session 과 같은 방법을 이용하여 방문자가 지난 번 방문했던 자라는 것을 증명하도록 하는 것임.




source: OAuth Authentication System- Behind the scenes, https://medium.com/geekculture/oauth-authentication-system-behind-the-scenes-e0720a3c31ab

## Cookies vs JWT

*Cookies vs JWT's are two very different things that do not serve the same purpose.*

**Cookies 는 운송 수단**

*Cookies are a transport mechanism. It is a piece of data that is managed automatically by the browser, and get added on to every request issued to server at some particular domain. Although they are native to the browser, mobile apps, desktop apps, etc can use cookies as well.*

**JWT 는 authentication mechanisam. 정보를 저장하는 방법**

*JWT's are an authentication mechanism. It is a token that encodes some piece of information.*
*To be clear: cookies are a tool to move data around. JWT's are a tool to store some information. They don't serve the same purpose.*

차이점:

a. **cookie 가 server 와의 교신과 소통에 관해서는 browser 가 알아서 처리**
b. **JWT 를 사용하면 JWT 의 이동에 관한 것을 개발자가 만들어야 함. cookie 를 사용하면 개발자 일이 적어짐.**

c. **우리 강의에서는 cookie-session 을 사용하여 user 에 관한 정보를 저장하는 방법을 배웠음. 이는 JWT 가 정보를 저장하는 방식과 유사. 즉, cookie 에서 정보를 저장하는 것은 JWT 와 유사. 다만 cookie 는 client(brower)와 server 간의 이동도 담당.**

d. **cookie 의 크기는 14k 로 제약이 되어 있어 cookie 의 크기가 크다면 JWT 가 대안.**

e. **모바일에 경우에는 cookie 의 사용에 제약이 있음(못 사용하는 것은 아니지만)**

*With all this said, the auth setup in the course very closely mimics exactly what a JWT does. The course shows how to store some encoded piece of information in a string via the cookie-session library. This encoded string has some information that identifies the user. Although we only store the user's ID in that string, we could just as easily add in some more information, such as the user's email address, full name, etc. A JWT functions in the same way - we can store some information in it.*

# cookie 에 대한 이해

1. /api/logout -> browser 의 network -> logout -> Headers 를 클릭하면

Set-Cookie:express:sess=eyJwYXNacG9ydCI6w319 가 보임. == 이후에 있는 문자가 encrypted cookie 임. 이 cookie 는 안에 아무 것도 없음. logout 되었으므로 user id 와 같은 정보가 없는 empty cookie 임

2. /auth/google/을 통해 로그인해서 network -> callback 으로 가면

그러나 로그인 상태에서는 cookie 에 user id 가 포함되어 있음. 이를 확인 하려면…

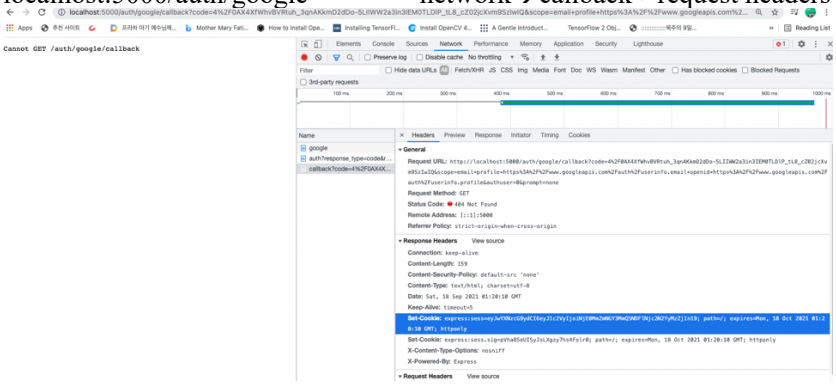
Response Headers 에 ..
Set-Cookie: express:sess=xxxxxxxxxxxxxxxx; path=/; expires=Mon, 18 Oct 2021 01:20:10 GMT; httponly

3. /api/current_user 로 가서 network -> current_user -> Headers 로 가면

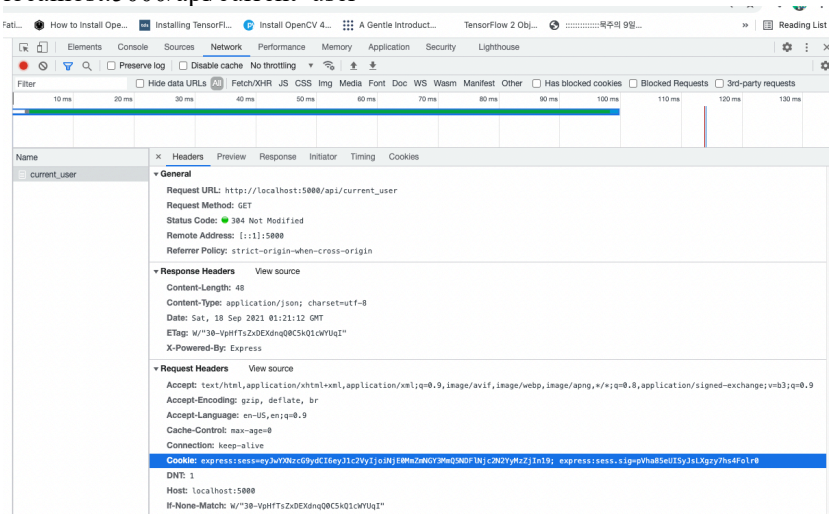
Request Headers 에 …
Cookie: express:sess=xxxxxxxxxxxxxxxxxxx; express:sess.sig=pVha85eUISyJsLXgzy7hs4Folr0 가 나옴.

이 cookie 는 encrypt 된 것이며 여기에 user id 가 포함되어 있음. user 가 request 를 보낼 때 browser 가 자동으로 첨부한 cookie 임

localhost:5000/auth/google network→callback->request headers




localhost:5000/api/current user



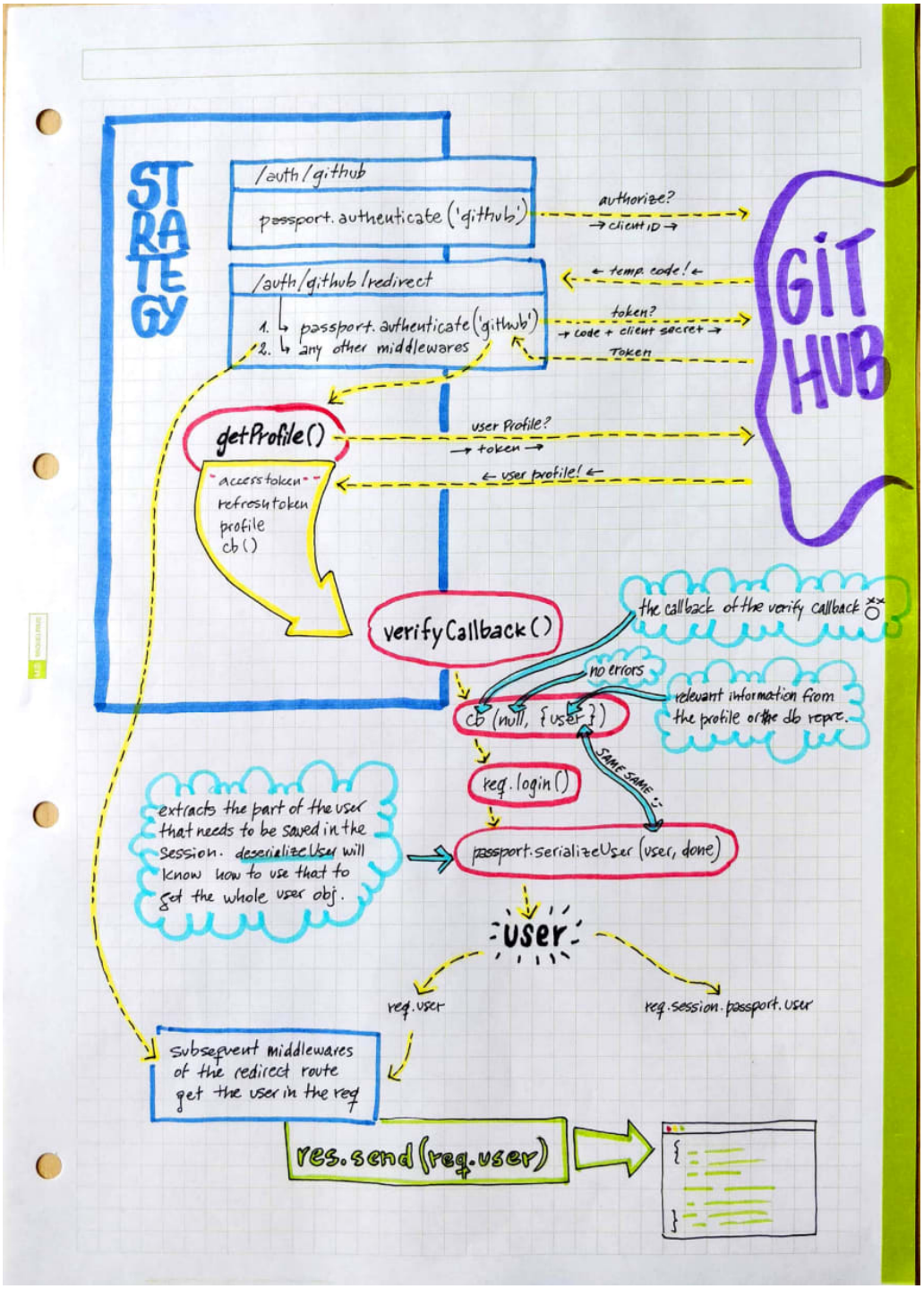

- **cookie-session 과 express-session 의 차이**

두 라이브러리 모두 같은 기능을 수행. 그러나 cookie 에 어떻게 정보를 저장하는 방식에서 두 라이브러리는 차이가 남. 둘다 cookie 를 사용하는 것은 동일.
cookie-session 에서는 user 가 곧 cookie 임 user id 가 cookie 에 들어가 있음. 둘이 동일. cookie 에 user 에 관한 모든 정보가 들어가 있음. current user 를 인식하려면 cookie 을 들여다 보면 됨. cookie 에 있는 user id 로 user 를 찾아내고 이를 req.session 에 설정함.

cookie-session 에서는 user id 가 cookie 에 있음.

express-session 은 cookie 에 user id 가 없고 session id 로 session 을 reference 함. express-session 은 cookie 에 있는 session id 로 session 관련 데이터가 저장되어 있는 'session store'에서 필요한 정보를 찾아냄. 보통 'session store'는 database 임.
따라서 express-session 에서는 cookie 에 아주 작은 정보 즉, session id 만 저장함.
cookie 의 허용최대 크기는 4kb. cookie 가 이를 초과하면 express-session 을 사용.

## another diagram for authentication process




source: https://dev.to/anabella/a-peep-beneath-the-hood-of-passportjs-oauth-flow-eb5

# passport.js 에 async/await 적용하기

passport.js

```
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;
const mongoose = require('mongoose');
const keys = require('../config/keys');

const User = mongoose.model('users');

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

passport.deserializeUser((id, done) => {
  User.findById(id).then(user => {
    done(null, user);
  });
});

passport.use(
  new GoogleStrategy(
    {
      clientID: keys.googleClientID,
      clientSecret: keys.googleClientSecret,
      callbackURL: '/auth/google/callback'
    },
    // (accessToken, refreshToken, profile, done) => {
    //   User.findOne({ googleId: profile.id }).then(existingUser => {
    //     if (existingUser) {
    //       // we already have a record with the given profile ID
    //       done(null, existingUser);
    //     } else {
    //       // we don't have a user record with this ID, make a new record!
    //       new User({ googleId: profile.id })
    //         .save()
    //         .then(user => done(null, user));
    //     }
    //   });
    // }
     async ( accessToken, refreshToken, profile, done ) => {
       try {
         const existingUser = await User.findOne( { googleId: profile.id } )

         if ( !existingUser ) {
           const user = await new User( { googleId: profile.id } )
           user.save()
           return done(null, user)
         }
         done( null, existingUser )
```

```
    } catch (error) {
      res.status(500).send()
    }
  }
 )
);
```

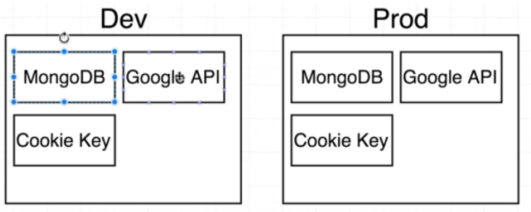
# Lecture 6. dev vs prod

개발용과 production 용의 2 개를 별도로 준비

local computer 에서 돌릴 때는 development 용으로 key 를 만들고
heroku 같이 production 용으로는 따로 key 를 만들어야 함.

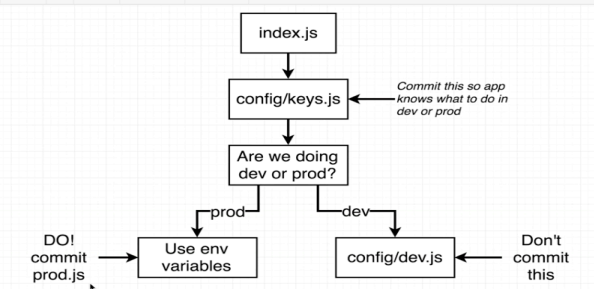
production 의 경우에는 heroku 와 같은 remote site 에 key 를 저장.

- **중요**
  - mongo db 도 dev 용, prod 용 2 개를 준비해야 함. 만일 여러분의 회사에서 dev, prod 용 database 를 따로 만들지 않고 있다면 매우 위험한 접근을 하고 있는 것임.
  - 우리 application 에서는 google api, cookie key 는 다 dev, prod 별도로 만들 것임.
  - mongo database 는 dev 에서 사용하던 것을 그대로 production 에서도 사용. 실제 현업에서는 대개 별도로 만들어서 사용.
  - prod key 는 heroku 에 저장될 것 임. dev key 는 여러분 pc 에 저장되어 있음. 따라서 heroku 에 저장되어 있는 key 가 더 안전.
  - production 용 mongo db 는 임의로 변경해서는 안될 경우가 많음. 그래서 development 용 mongo db 를 만들어서 여러 테스트를 실행해야 하는 것임.




- **할일**
  - config/keys.js 의 내용을 변경하고 config/prod.js 와 config/dev.js 를 별도로 만듬

## Version Control Scheme




- keys.js

```
// process.env.NODE_ENV === 'production"은 heroku 에 의해 자동으로 설정됨
if ( process.env.NODE_ENV === 'production' ) {
   module.exports = require('./prod')
} else {
   module.exports = require('./dev')
}
```

- prod.js

```
// prod.js --- production keys are here!!!

module.exports = {
  googleClientID: process.env.GOOGLE_CLIENT_ID,
  googleClientSecret: process.env.GOOGLE_CLIENT_SECRET,
  mongoURI: process.env.MONGO_URI,
  cookieKey: process.env.COOKIE_KEY
};
```

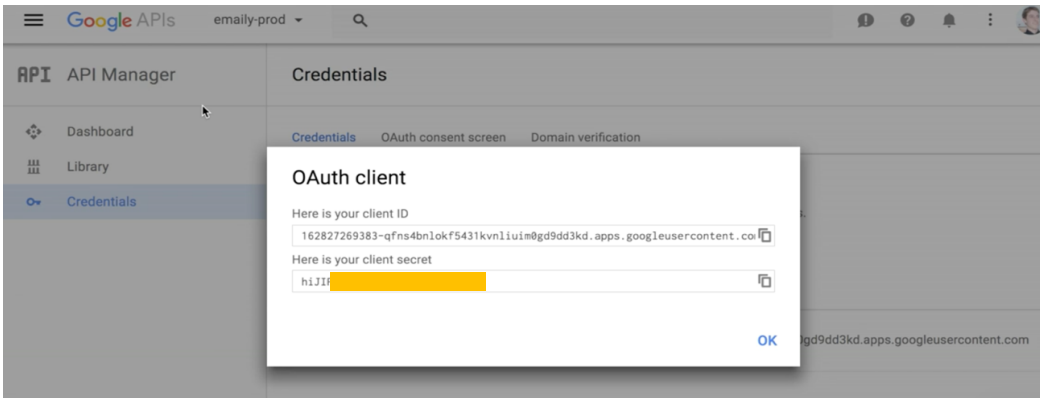
- .gitignore 에서 confing file 변경
  .gitignore 로 가서 config/index.js 대신 config/dev.js 를 포함 시킴
  config/dev.js 는 commit 하면 안됨. 그러나 config/prod.js 는 commit 해야 함.

- committing to Heroku

```
git add .
git commit -m "modifying the config files"
git push heroku main
```

- console.cloud.google.com 으로 가서 production 용 project 를 만들고 setting 할 것
  - 승인된 자바스크립트 출처의 URI 추가에 https://my-mern-app.herokuapp.com 입력
  - 승인된 리디렉션 URI 에 https:// my-mern-app.herokuapp.com/auth/google/callback 입력
  - 만들기 클릭
  - 클라이언트 ID 와 클라이언트 보안 비밀 번호를 다른 곳에 기록하여 둘 것

## creating Google project

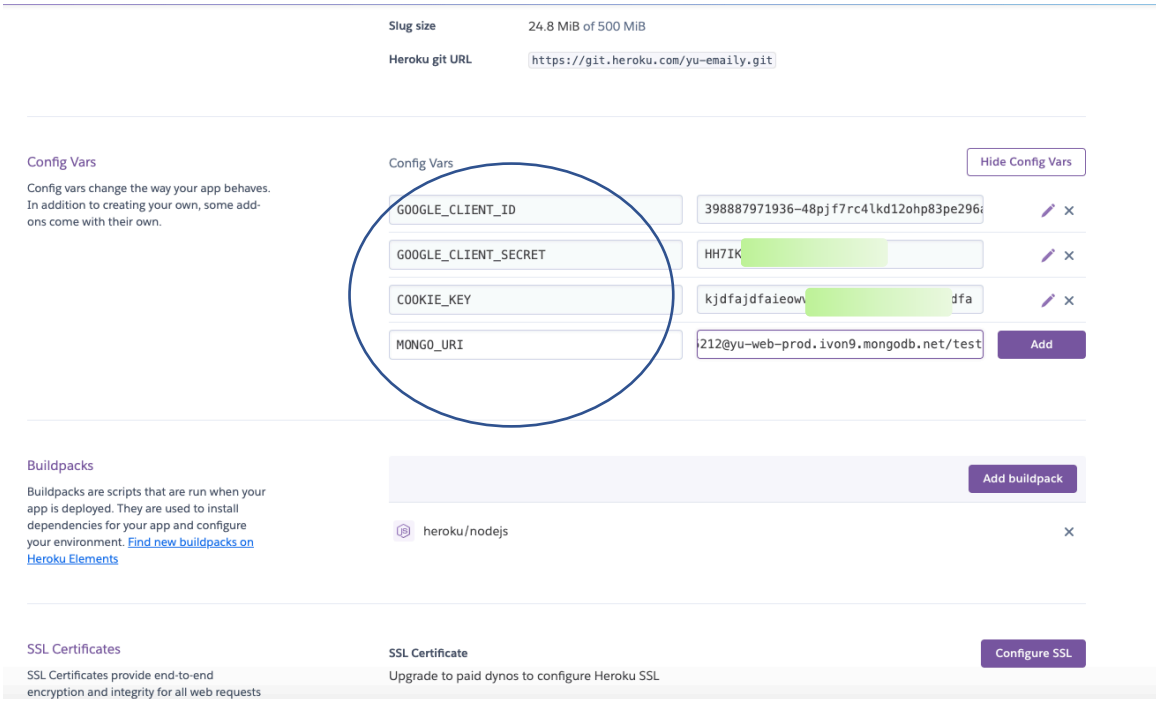



## Heroku Env Variables

go to Heroku dashboard -> click settings -> Reveal Config Vars

prod.js 에 있는 **env var names** 와 똑 같이 Reveal Config Vars 에 입력해야 함.

나의 경우는.... 이렇게 setting 을 하면 됨. Heroku 에서는 aqidb mongo db(clusters: air-quality-prod)에 연결하였음.




now try Heroku open

and in the url, try https://yu-nemr-air.herokuapp.com/auth/google ====> still errors broke out.

## Fixing Heroku Proxy Issues

**400 오류: redirect_uri_mismatch**

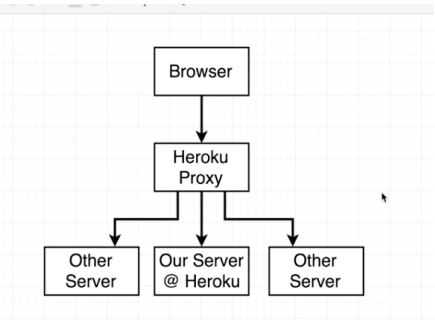
The redirect URI in the request, http://yu-emaily.herokuapp.com/auth/google/callback, does not match the ones authorized for the OAuth client. To update the authorized redirect URIs, visit: https://console.developers.google.com/apis/credentials/oauthclient/${your_client_id}?project=${your_project_number}

여기에서 보면 https 가 아니라 http 로 되어 있는 것을 알 수 있음. 이를 fix 할 것임.

http are generated by two factors.
그리고 passport.js 안 에 있는 google strategy 를 다시 보기로 하자.

```
passport.use(
  new GoogleStrategy(
    {
      clientID: keys.googleClientID,
      clientSecret: keys.googleClientSecret,
      callbackURL: '/auth/google/callback'
    },
```

위에서 callbackURL 을 보면 relative url 이 적혀 있는 것을 알 수 있음. 이 상대적 url 때문에 GoogleStrategy 은 우리가 https 가 아니라 http 로 가기를 원하는 지로 착각함.
잠깐 어떤 식으로 browser 에서 Heroku 로 연결되는 지를 살펴 봄.




Browser 에서 heroku 로 연결될 때 heroku 는 heroku proxy 혹은 load balancer 를 사용. to make sure the browser request is connected to the Heroku. The issue here is by default the Google Strategy does not set it as https if the request is via the Heroku proxy.
To fix the problem, one option is to change the configuration in the Google Strategy.

passport.js

```
passport.use(
  new GoogleStrategy(
    {
      clientID: keys.googleClientID,
      clientSecret: keys.googleClientSecret,
      callbackURL: '/auth/google/callback',
      proxy: true
```

},

그리고 Heroku 에 deploy

git add .
git commit -m “tweak the Google Strategy proxy”
git push heroku master

======
error

400 오류: redirect_uri_mismatch

The redirect URI in the request, https://yu-emaily.herokuapp.com/auth/google/callback, does not match the ones authorized for the OAuth client. To update the authorized redirect URIs, visit: https://console.developers.google.com/apis/credentials/oauthclient/${your_client_id}?project=${your_project_number}

https://yu-nemr-air.heroku.com/auth/google/callback

# Lecture 7. Working w. React

- 학습 주요 포인트
  - React client 와 Node/Express server 를 실행하기
  - Development version 에서는 proxy 사용
  - Production version 에서는 통합 architecture 사용

## Development

### React App Generation

at myServer(혹은 yu-nemr-air)에 React 설치
npx create-react-app client

node 의 하위 directory 로 client(React)를 만드는 것임.

### Running the Client and Server

- 두개의 server 를 돌리려면 두개의 터미널에서 명령문 집행

  at myServer( 혹은 yu-nemr-air)
  npm run dev

  at myServer(혹은 yu-nemr-air)/client
  npm run start

- 두개의 server 를 하나의 명령문으로 실행하는 방법
  우선 concurrently library 설치
  npm i -D --save concurrently

myServer 에 있는 package.json 을 수정

```
"scripts": {
   "start": "node index.js",
   "server": "nodemon index.js",
   "client": "npm run start --prefix client",
   "dev": "concurrently  \"npm run server\"  \"npm run client\""
 },
```

실행

myServer 에서
npm run dev  // 이렇게 하면 server, server/client 에서 두개의 명령문을 동시에 집행

**Create React App Proxy Update**

*Updated 3-22-2021*
proxy 를 client-side package.json file 에 추가해야 함.

**1. client/** directory 에 the http-proxy-middleware library 설치

npm install http-proxy-middleware@1.0.6

**2. client/src 에 setupProxy.js** 생성. 이 파일을 import 할 필요는 없음. CRA 자동으로 이 파일 이름을 찾아서 load 함.

**3. setupProxy.js** file 에 우리의 proxy 를 추가

```
const { createProxyMiddleware } = require( "http-proxy-middleware" );

module.exports = function (app) {
  app.use(
    ["/api/*", "/auth/google"],
    createProxyMiddleware({
      target: "http://localhost:5000",
    })
  );
};
```

**4.** at the myServer directory 에서 npm run dev 를 restart 할 것. setupProxy file 내용을 변경할 때 마다 restart 할 것.

**Log in from App.js**

App.js

```
import './App.css';

function App() {
 return (
```

```
    <div className="App">
      <header className="App-header">
         welcom to React and Node JS
        <a href="/auth/google">Sign In With Google</a>
      </header>
    </div>
  );
}

export default App;
```

error 발생. 이 에러를 수정하는 것은 google oauth redirect configuration 단원에서 설명할 것임.

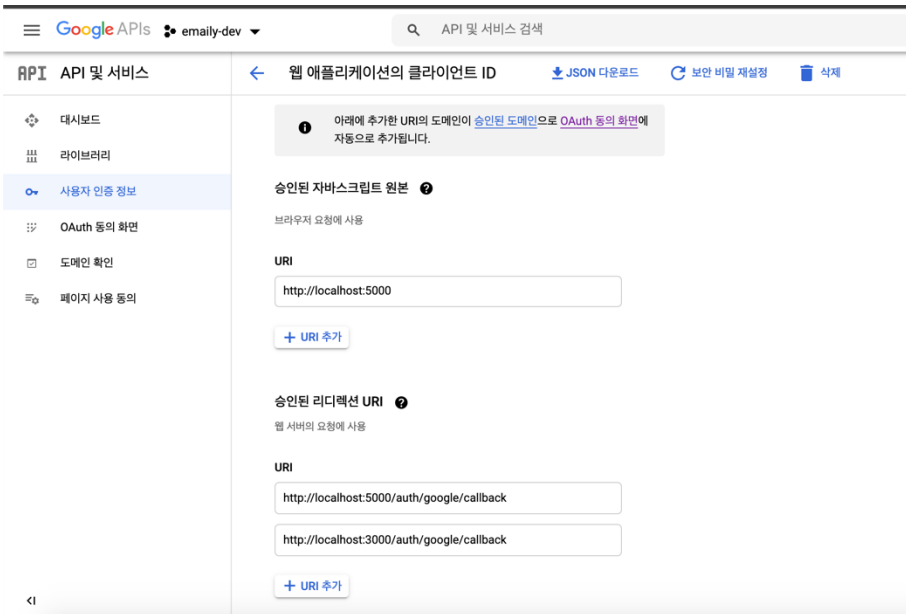
**Proxy 설정 설명**

App.js 에서 <a href="/auth/google">Sign In With Google</a> 로 herf 를 /auth/google 로 지정하면 브라우저는 현재 localhost:3000/auth/google 로 이동함.
허지만 우리는 localhost:5000/auth/google 로 이동을 해야 함. 이를 가능하게 하기 위해 createProxyMiddleWare 에 target: http://localhost:5000 이라는 proxy 를 설정하는 것임.

**google oauth redirect configuration**

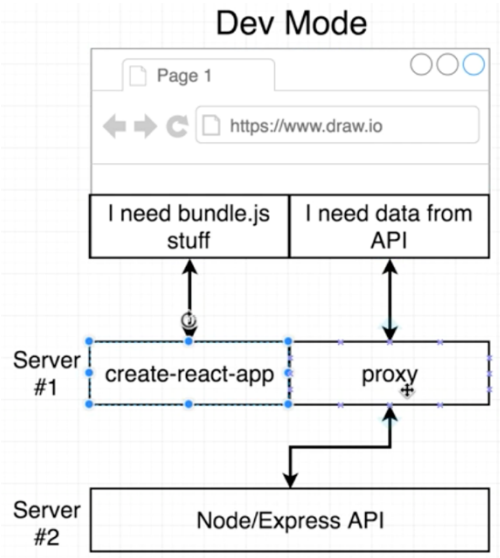
yu-air-quality 로 가서 승인된 리디렉트 URI 에 http://localhost:3000/auth/google/callback 추가

이제 npm run dev 를 실행하면 작동하는 것을 알 수 있을 것.



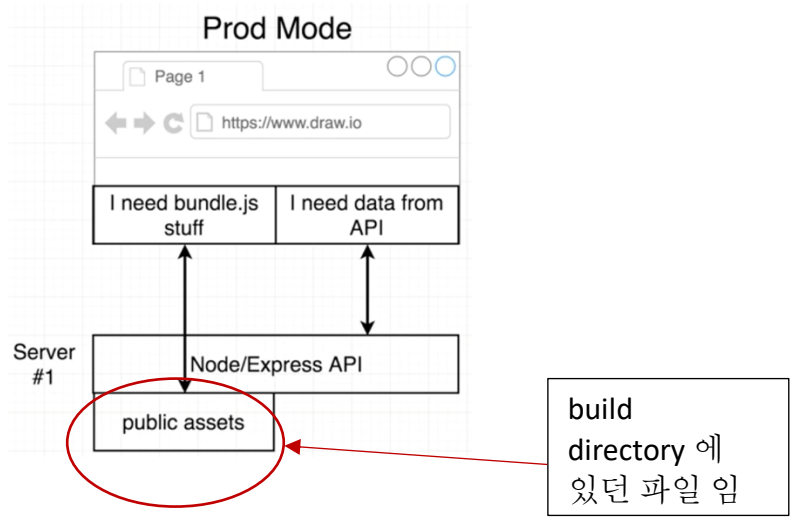

**Create React App's Proxy 의 장점**

Dev mode 에서는 node/express api 가 react server 를 직접 거치지 않고 proxy 를 통해 user(browser)에게 접근. 그러나 Prod mode 에서는 node/express 만 단독으로 server 역할을 하게 됨. react server 는 존재하지 않게 됨.




## Production

Heroku 에 production version 을 upload 하기 전에 모든 자바스크립트/css 파일을 webpack 과 Babel 를 이용해서 최종 production file 로 변환하고 이 final build file 을 build folder 에 저장하여 Heroku 에 upload 함 .




**Heroku Deployment**

- creating build directory

myServer/client 에서

npm run build

위를 실행하면 모든 javascript, css, html file 이 build 되어 build directory 에 저장됨.

.gitignore 에 client/build 가 들어가 있으면 안됨. client/build 는 heroku 에 deploy(upload) 되어야 함.

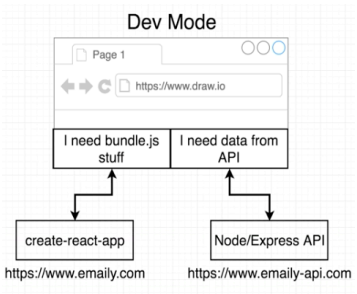
- yu-nemr-air/index.js 수정

yu-nemr-air/index.js 에 require('./routes/authRoutes')(app) 밑에 다음을 추가해야 함.

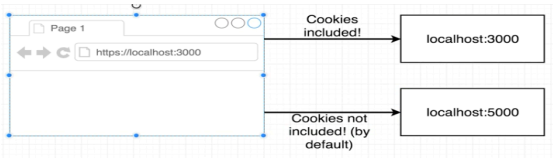
```
if ( process.env.NODE_ENV = 'production' ) {
  /*
  Express will server up production assets
  like our main.js or main.css files
  */
  app.use( express.static('client/build' ) );

  // Express will serve up index.html file
  // if it doen't recognize the route
  const path = require( 'path' );
  app.get( '*', ( req, res ) => {
    res.sendFile( path.resolve( __dirname, 'client', 'build', 'index.html' ) );
  } );
}
```

**dev 와 prod 의 다른 architecture**

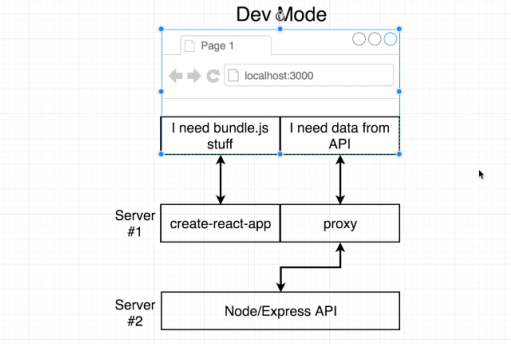
- dev 의 경우




왜냐하면?

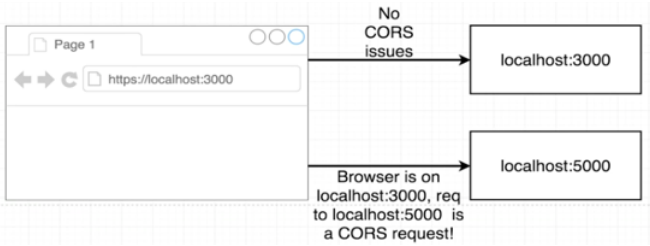



첫째, 우리는 authentication 을 위해 cookie 를 사용하고 있음. create-react-app 에서는 port 3000 를 사용. 그러나 node js 에서는 port 5000 을 사용하고 여기에는 cookie 가 포함되어 있지 않음. 이 문제를 해결하기 위해 다음 그림에 나와 있는 방법을 사용하는 것임.



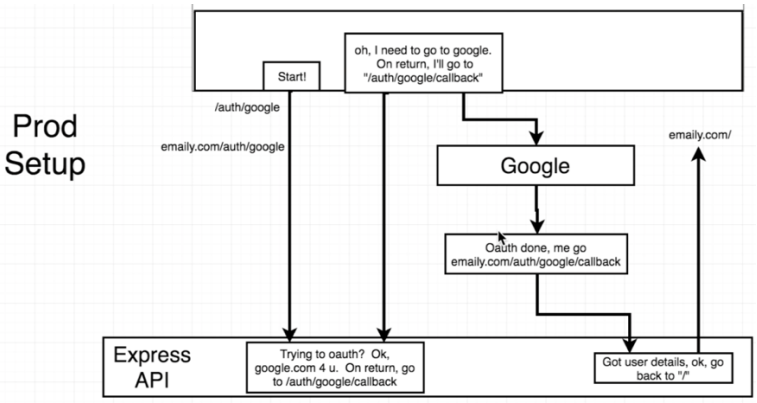

localhost:3000 에서 로그인을 하면 브라우저는 proxy 만을 보고 cookie 를 제공. 그러면 proxy 가 이 cookie 를 node/express api 에게 전달. 따라서 비록 node/express api 의 port number 가 5000 이라고 해도 cookie 가 있으므로 브라우저에 접근이 가능한 것임.
security issue 때문에 cookie 가 주어지지 않고 그렇게 때문에 proxy 를 사용하여 cookie 를 받는 것임.

두번째 이유는 다음 그림에 나와 있음.



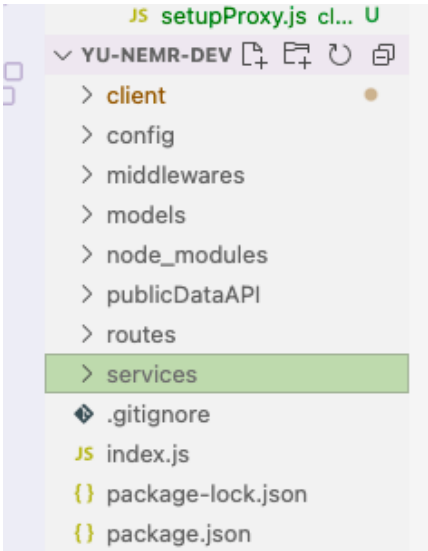

이도 마찬가지로 다른 port 를 사용하면 CORS 로 의심. 그렇기 때문에 다음과 같이 proxy 를 사용하는 architecture 를 채택

- prod 의 경우



# Lecture 8. 공공데이터포털에서 데이터 처리

## folder structure




## 공동데이터 포털의 공기 오염 데이터 처리 준비 작업

- **library installation**

at myServer (or whatever your root directory for node/express)
npm i axios

- **air quality data 를 mongod db 에 저장**

node/express 의 model 에 Air.js 추가

myServer/models/Air.js

```
const mongoose = require( 'mongoose' )
const { Schema } = mongoose;

const airSchema = new Schema({

  location: {
    type: String,
    required: true,
    trim: true
  },
  time: {
    type: String,
    required: true,
    trim: false
  },
  pm10: {
```

```
    type: String,
    required: true,
    trim: true
  },
  pm25: {
    type: String,
    required: true,
    trim: true
  },
  no2: {
    type: String,
    required: false,
    trim: true
  },
  _user: { type: Schema.Types.ObjectId, ref: 'User' }
})

mongoose.model('airs', airSchema)
```

- **login 상태 일 때 만 이용 허용**

myServer/middlewares/requirelogin.js

```
module.exports = (req, res, next) => {
  if(!req.user){
    return res.status(401).send({error:'You must log in'})
  }
  next()
}
```

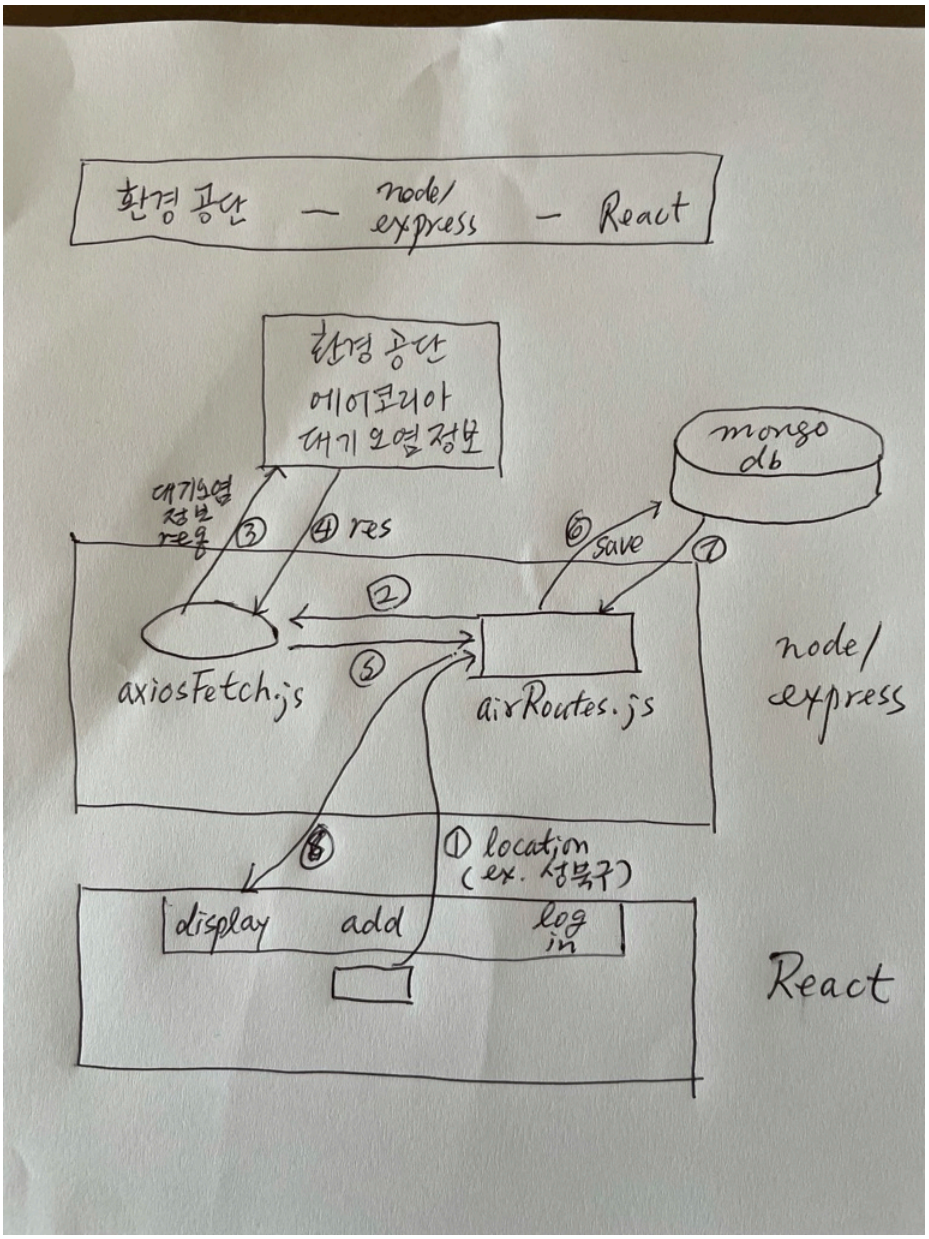
- **localhost:3000/auth/google/callback 이후 /airdata 로 이동**

myServer/routes/authRoutes.js

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

module.exports = app => {
 app.get(
  '/auth/google',
  passport.authenticate('google', {
   scope: ['profile', 'email']
  })
 );

 // google authenticate 가 끝나면 /airdata 로 돌아감.
 app.get(
  '/auth/google/callback',
  passport.authenticate( 'google' ),
  ( req, res ) => {
   res.redirect('/airdata')
  }
 );
```

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

  // serializeUser 가 보낸 req.user
  app.get('/api/current_user', (req, res) => {
    res.send(req.user)
  });
  // serializeUser 가 보낸 req.session.passport
  app.get('/api/session/passport', (req, res)=> {
    res.send(req.session.passport)
  })
  // cookie-session 이 de-encrypt 하여 넘겨 준 req.session
  app.get('/api/session', (req, res)=>{
    res.send(req.session)
  })

};
```

- 공공데이터 포털 – node/express – React 의 연결구조
  전형적인 MERN 구조



- **공공데이터 포털 api 에서 data fetch**

myServer/publicDataAPI/axiosFetch.js

```
const axios = require('axios');
const serviceKey = require( '../config/publicDataPortalKey' )
const aqiUrl = require('../config/url')

const axdata = async (stationName, callback) => {
 const url = aqiUrl.airUrl
 let ServiceKey = decodeURIComponent(serviceKey.publicPortalkey)

 // axios 로 한국환경공단_ 에어코리아_ 대기오염정보 api 에 연결하여 데이터를 불러옴.
 try {
  const response = await axios.get(url,
   { params:
    {
     dataTerm: 'DAILY',
     stationName: stationName,
     pageNo: '1',
     numOfRows: '1',
     ver:'1.3',
     returnType: 'json',
     ServiceKey: ServiceKey}
   }
  )

  const {dataTime,pm10Value,pm25Value,no2Value} = response.data.response.body.items[0]
  const airdata = {
   location: stationName,
   time: dataTime,
   pm10: pm10Value,
   pm25: pm25Value,
   no2: no2Value
  }
  callback(undefined, {airquality:airdata})
 } catch (error) {
  console.log('error broke out:  ', error)
 }
}

module.exports = axdata;
```

- **config 에 public data portal 의 url 및 service key 추가**

myServer/config/url.js

```
module.exports = {
  airUrl: 'http://apis.data.go.kr/B552584/ArpltnInforInqireSvc/getMsrstnAcctoRltmMesureDnsty?'
}
```

myServer/config/publicDataPortalKey.js

```
module.exports = {
  publicPortalkey:
'g9faQvlMMszwEA639lFX1Y%2BcimFiC1qn8FkPQQ9FJQX3F0Ft%2FrLsYC%2BZrPTaRsKnrirK0lyGW%2BgI2DmLhgu
KQA=='
}
```

- **air quality data 의 endpoint 만들기**

myServer/routes/airRoutes.js

```
const mongoose = require('mongoose')
const requireLogin = require('../middlewares/requireLogin')
const Air = mongoose.model('airs')

const axdata = require( '../publicDataAPI/axiosFetch' )

module.exports = app => {
  app.post( '/airdata/update', requireLogin, async ( req, res ) => {
    try {
      await axdata(req.body.station, ( error, { airquality } = {} ) => {

        Air.findOne( { time: airquality.time, location:airquality.location }, ( err, doc ) => {
          if ( doc ) {
            res.status(302).send( "Duplicate document exists" )
          } else {
            const air = new Air( {
              location, time, pm10, pm25, no2
            } = airquality  )
            air.save()
            // res.send('Air quality data are saved.')
            res.send(air)
          }
        } )
      } )
    } catch ( e ) {
      res.status( 400 ).send( e )
    }
  })

  app.get('/airdata/display', requireLogin, async (req, res) => {
    try {
      const airdata = await Air.find({})
      res.send(airdata)
    } catch (e) {
      res.status(500).send()
    }
  })

  app.get('/airdata/Date/:date', async (req, res) => {
    const measureDate = req.params.date
    try {
      const air = await Air.find({time:{$regex:measureDate}})
```

```
        if (!air) {
            return res.status(404).send()
        }
        res.send(air)
    } catch (e) {
        res.status(500).send()
    }
} )

app.get('/airdata/Time/:time', async (req, res) => {
    const measureTime = req.params.time

    try {
        const air = await Air.find( { time: measureTime })
        if (!air) {
            return res.status(404).send()
        }
        res.send(air)
    } catch (e) {
        res.status(500).send()
    }
} )

app.get('/airdata/:id', async (req, res) => {
    const dataID = req.params.id
    console.log("dataID from /airdata/:id", dataID)
    try {
        const air = await Air.find( { _id: dataID })
        console.log("air from /airdata/:id", air)
        if (!air) {
            return res.status(404).send()
        }
        res.send(air)
    } catch (e) {
        res.status(500).send()
    }
} )

app.delete('/airdata/:time', async (req, res) => {
    try {
        const air = await Air.findByIdAndDelete(req.params.time)

        if (!air) {
            res.status(404).send()
        }

        res.send(air)
    } catch (e) {
        res.status(500).send()
    }
})
}
```

- **setupProxy 에 airdata route 추가**

client/src/setupProxy.js

```
const { createProxyMiddleware } = require( "http-proxy-middleware" )

module.exports = function ( app ) {
   app.use(
      ["/api/*", "/auth/google", "/airdata/display", "/airdata/update"],
      createProxyMiddleware( {
         target: "http://localhost:5000",
      })
   )
}
```

- **index.js 수정**

myServer/index.js

```
const express = require('express');
const mongoose = require('mongoose');
const cookieSession = require('cookie-session');
const passport = require('passport');
const keys = require('./config/keys');
require('./models/User');
require('./models/Air')
require('./services/passport');

mongoose.connect(keys.mongoURI)

const app = express();

app.use(
   cookieSession({
      maxAge: 30 * 24 * 60 * 60 * 1000,
      keys: [keys.cookieKey],
   })
);
app.use(express.json()) // req.body 를 json 으로  parse 하는 데 필요. express v4.16 이후 부터 사용
app.use(passport.initialize());
app.use(passport.session());
require('./routes/authRoutes')(app)
require('./routes/airRoutes')(app)

if (process.env.NODE_ENV = 'production') {
   app.use(express.static('client/build'));
   const path = require('path');
   app.get('*', (req, res) => {
      res.sendFile(path.resolve(__dirname, 'client', 'build', 'index.html'));
   });
```

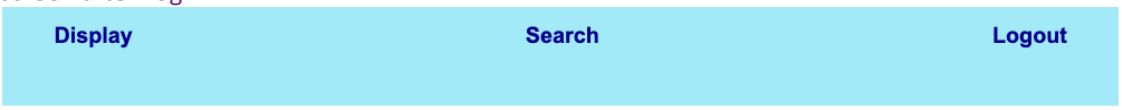
```
}

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
   console.log(`The app is running on ${PORT}`)
});
```

# **React** 에서 공동 데이터 포털의 공기 오염 데이터 처리

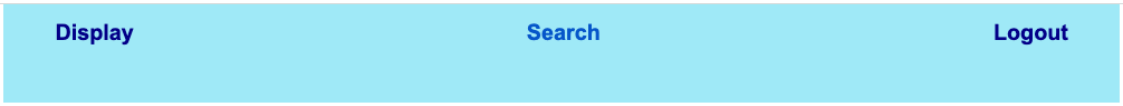
- **React Screen**

screen before Log In


screen after Log In




screen after Search clicked




Enter the location:

UpdateDB Status:

Update Data

screen after Display clicked

- **reactstrap 사용**

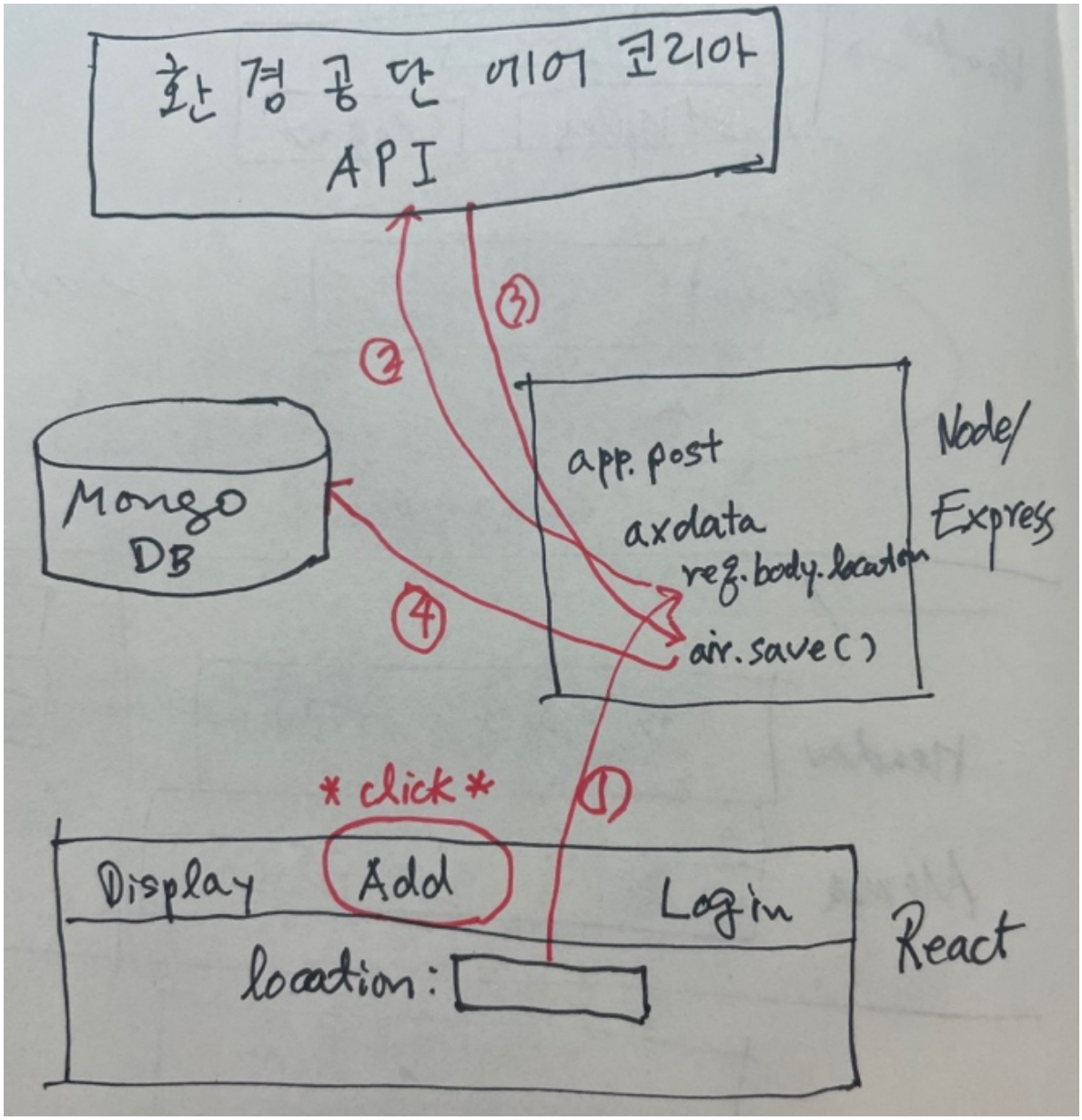
react-bootstrap 과 80~90% 동일. components 이름만 다른 경우가 많음.

```
npm install  bootstrap
npm install  reactstrap react react-dom
```

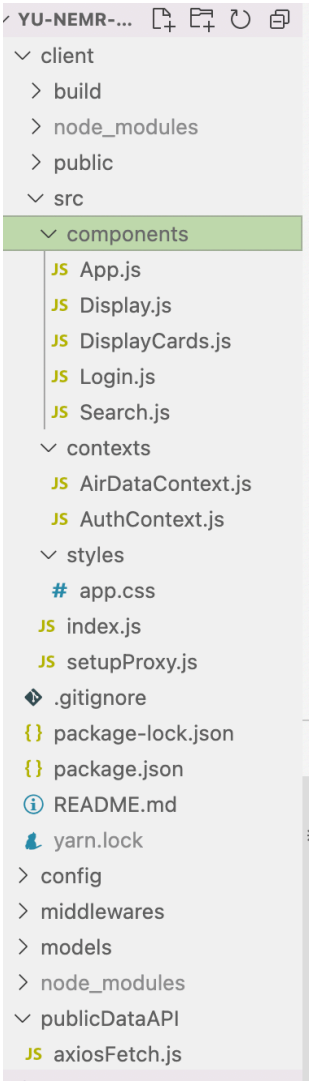
at client/src/index.js

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

- **req/res among React/Express/Mongo**



- **File Structure**




- **client/src/styles/app.css**

```
.app {
  margin: 0 2rem;
  padding: 0 1.6rem;
  max-width: 60rem;
}

.header {
  background: #a2eaf8;
  margin: 1.2rem, 0;
  height: 5rem;
  font-family: Calibri, Helvetica, Arial, sans-serif;
  font-size: large;
  align-items: center;
  display: flex;
  justify-content: space-around;
```

```
 /* flex-direction: row; */
 padding: 0.6rem;
 color: #03045e;
 font-weight: bold;
}

a {
 text-decoration: none;
 margin: 10rem;
 color: darkblue;
}

.heading-menu {
 text-align: center;
 font-family: "굴림", Arial, Helvetica, sans-serif;
 color: #03045e;
 margin: 0.5rem 0;
}

.heading-menu .heading-h1 {
 font-size: 1.6rem;
}

.heading-menu .heading-p {
 font-size: 0.9rem;
}

.formLayout {
 margin: 2rem 0;
}

.formLayout .formSub {
 margin: 0.1rem 0;
}

.card-image-resizing {
 border-top-width: 1em;
 min-height: 300px;
 height: 100%;
 object-fit: cover;
}

#renderBlue {
 color: blue;
 margin: 0 0.4rem 0;
}

#renderRed {
 color: red;
 margin: 0 0.4rem 0;
```

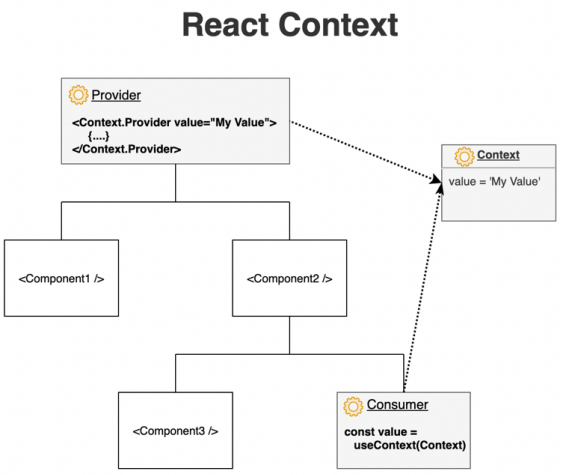
```
}
```

- **index.js 의 생성과 내용**

client/src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './components/App'
import 'bootstrap/dist/css/bootstrap.min.css';
import './styles/app.css';

ReactDOM.render(
  <App />,
  document.querySelector( '#root' )
)
```

- **Context Provider & useContext**




source: https://dmitripavlutin.com/react-context-and-usecontext/

- **In Our Web Application**

  - AuthContext, AirDataContext 를 정의
  - Login.js, Display.js, 그리고 Search.js 를 AuthContextProvider 와 AirDataContextProvide 로 wrap
  - Login.js 와 Display.js 에 value 을 통해 state, setState 를 전달
  - Login 에게는 state, setState 를 전달하고
  - Display 에게는 airData, setAirData 를 전달

- **AuthContext 정의 하기**

client/src/contexts/AuthContext.js

```
import React, {useState, createContext} from 'react'

export const AuthContext = createContext()

const initState ={
   auth: false
}

export const AuthContextProvider = props => {
   const [state, setState] = useState(initState)
   return <AuthContext.Provider value={[state, setState]}>{props.children}</AuthContext.Provider>
}
```

- **AirDataContext 정의**

```
import React, {useState, createContext} from 'react'

export const AirDataContext = createContext()

export const AirDataContextProvider = props => {
   const [airData, setAirData] = useState ("")
   return <AirDataContext.Provider value={[airData,
setAirData]}>{props.children}</AirDataContext.Provider>
}
```

- **App 에서 ContextProvider wrapping**

하위 component 에서 Context 의 value 에 정의 된 state 와 useState 를 사용하도록 함.
만약 useReducer 를 사용할 경우는 state 와 dispatch 를 사용함. (하단에 useReducer 방법예제 참조)

client/src/components/App.js

```
import React from 'react';
import { BrowserRouter, Route } from 'react-router-dom';
import {AuthContextProvider} from '../contexts/AuthContext'
import {AirDataContextProvider} from '../contexts/AirDataContext'
import Login from './Login'
import Display from './Display'
import Search from './Search'

const App = () => {
 return (
   <div className="app">
    <BrowserRouter>
     <AuthContextProvider>
      <AirDataContextProvider>
         <Login />
```

```
          <Route exact path="/display" component={ Display } />
          <Route exact path="/search" component={ Search }  />
        </AirDataContextProvider>
      </AuthContextProvider>
    </BrowserRouter>
  </div>
 )
}

export default App;
```

- **Login.js: useContext, state, setState**

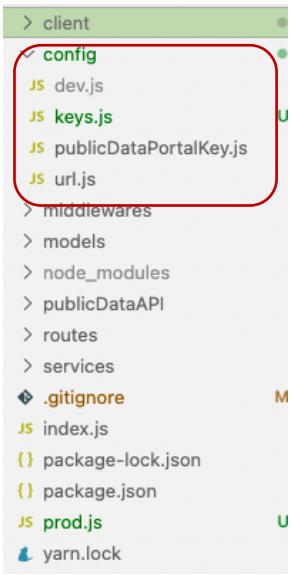
client/src/components/Login.js

```
import React, {useEffect, useContext} from 'react'
import { Link } from 'react-router-dom'
import { AuthContext } from '../contexts/AuthContext'

import axios from 'axios'

const Login = () => {

  const [state, setState] = useContext(AuthContext)

  useEffect (()=>{
    async function fetchUser (){
      const res = await axios.get('/api/current_user')
      console.log("res inside useEffect of Login", res)
      res.data === "" ? setState(false) : setState(res.data)
    }
    fetchUser()
  },[])
  const auth = state

  const renderContent = () =>{
    switch ( auth ) {
      case false:
        return <a href="/auth/google">Login With Google</a>

      default:
        return (
          <>
            <Link to={"/display"}>Display</Link>
            <Link to={"/search"}>Search</Link>
            <a href="/api/logout">Logout</a>
          </>
        )
    }
```

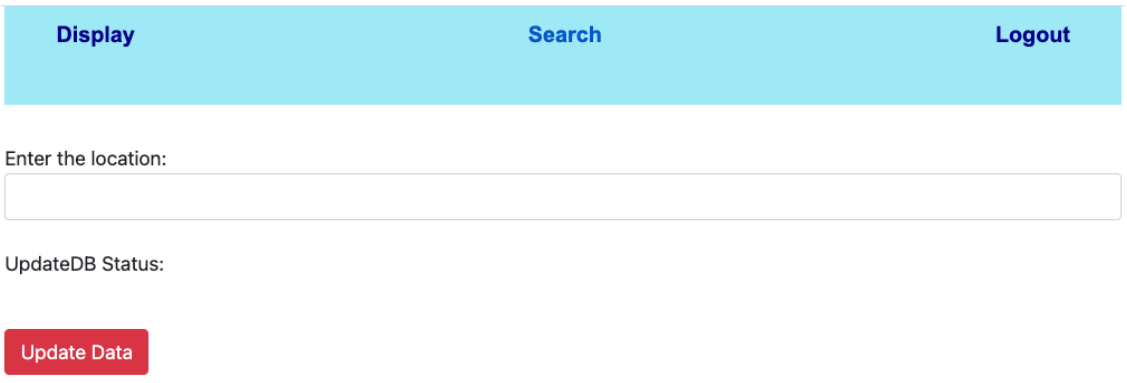
```
  }
  return (
        <div className="header" >
          <p>{renderContent()}</p>
        </div>
  )
}

export default Login;
```

# Lecture 9: Completion of Search and Display Components

- **config file 내용**




- **Search: location 에 따라 air data 불러오기**




client/src/components/Search.js

```
import React, { useState} from 'react';
import { Link } from 'react-router-dom'
import { Button, Form, FormGroup, Label, Input } from 'reactstrap';
```

```
import axios from 'axios'

const Search = () => {
   const [location, setLocation] = useState("")
   const [dataStatus, setDataStatus] = useState("")

   const onLocationChange = ( e ) => {
      setDataStatus("")
      setLocation( e.target.value)
   }

   const onUpdateSubmit = async (e) => {
      e.preventDefault();
      setDataStatus(".....Loading")
      try {
         const res = await axios.post('/airdata/update', { station: location } )
         if (res){
            setDataStatus("Data updated! Click Update Button.")
            return res
         }
      } catch ( err ) {
         setDataStatus("No data to update!")
         return err
      }
   }

   const onAddClick = async () => {
      setDataStatus("Loading.....")
   }

   return (
      <div className="formLayout">
         <Form onSubmit={onUpdateSubmit}>
            <FormGroup>
            <Label >Enter the location:{location}</Label>
            <Input onChange={onLocationChange} />
            <br></br>
            <p>UpdateDB Status:
               <span id="renderRed">{dataStatus}</span>
            </p>
            </FormGroup>
            {
               <Link to={'/display'}>
                  <div className="formSub">
                     <Button color="danger" onClick={onAddClick} >Update Data</Button>{' '}
                  </div>
               </Link>
            }
         </Form>
```

```
      </div>
    )
}

export default  Search;
```

- **Display: air data 를 불러와서 display 하기**


client/src/components/Display.js

```
import React, { useEffect, useContext } from 'react';

import { Container, Row } from 'reactstrap';
import DisplayCards from './DisplayCards';

import axios from 'axios'
import { AirDataContext } from '../contexts/AirDataContext';

const Display = () => {

   const [airData, setAirData] = useContext(AirDataContext)
   useEffect( ()=> {
      async function fetchData (){
         const res = await axios.get('/airdata/display')
         setAirData(res.data)
      }
      fetchData()
   },[])

   let air = airData
   if (air){
      return (
```

```
        <div className="heading-menu">
          <h1 className="heading-h1">서울시 공기 정보</h1>
          <p className="heading-p">공기청정도 정보는 매시간마다 갱신됩니다. </p>
          <p className="heading-p">출처: 공공데이터포털</p>
            <Container>
              <Row>
                {
                  air.sort((a, b) => a.time <= b.time ? 1:-1).map(
                    (air) =><DisplayCards { ...air } key={air._id} />
                  )
                }
              </Row>
            </Container>
        </div>
      )
    }
    return <div> Loading... </div>
}

export default Display;
```

- **DisplayCards: where the air data are displayed**

client/src/components/DisplayCards

```
import React from 'react';
import { Col, Card, CardImg,
 CardTitle, ListGroup, ListGroupItem } from 'reactstrap';

const DisplayCards = ({ location, time, pm10, pm25, no2 }) => {

 const clearSky = 'images/clean-air.jpeg'
 const graySky = "images/dirty_air.jpeg"
 const perplexed = "images/perplexed.jpeg"


 let cardImage =  ( pm10 <=55 || pm25 <= 25) ? clearSky : graySky
 if (pm10 ==="-" || pm25 ==="-"){
  cardImage = perplexed
 }
 const bg = (pm10 !== "-" || pm25 !== "-") && pm25 <= 35 ? '#a2eaf8': "#dbd8e3"

 return (
  <Col sm="4" >
   <Card body style={{ width: '18rem', backgroundColor: bg  }} key={time.toString()}>
     <CardImg className="card-image-resizing" src={cardImage} top width="100%"/>
       <CardTitle className="title text-center">{ location } 측정치</CardTitle>
     <ListGroup variant="flush">
      <ListGroupItem> { time } </ListGroupItem>
      <ListGroupItem>pm10: { pm10 } 𝜇g/m³</ListGroupItem>
      <ListGroupItem>pm2.5: { pm25 } 𝜇g/m³</ListGroupItem>
      <ListGroupItem>no2: { no2 } 𝜇g/m³</ListGroupItem>
```

```
        </ListGroup>
      </Card>
    </Col>
  )
}

export default DisplayCards;
```

- **Landing.js: nothing special**

client/src/components/Landing.js

```
import React from 'react'

const Landing = () => {
  return (
    <div className="heading-menu">
      <h1 className="heading-h1">공공데이터 포털</h1>
      <p className="heading-p">시간별/구별 미세먼지 정보</p>
    </div>
  )
}
export default Landing;
```

- **Mongo Database 확인**

mongo db atlas 로 이동.

- **Heroku dispatch**

점검사항

**console.cloud.google.com 으로 가서…**
승인된 자바스크립트 원본에 heroku app 주소 (예: https://yu-nemr-prod.herokuapp.com)
승인된 리디렉션 URI 에 여러분의 주소 입력(예: https://yu-nemr-prod.herokuapp.com/auth/google/callback)

**Heroku setting 으로 가서…**
google authentication project 의 google client id, google client secret, mongo_uri 가 맞는 지를 재확인 할 것

**server 의 package.json 로 가서…**
build 와 install 을 추

```
"scripts": {
"build": "cd client && npm run build",
"install": "cd client && npm install",
"test": "echo \"Error: no test specified\" && exit 1",
"start": "node index.js",
"server": "nodemon index.js",
"client": "npm run start --prefix client",
"dev": "concurrently \"npm run server\" \"npm run client\""
},
```

위를 다 확인하고 다음을 실행...

```
git add .
git commit -m “dispatch to Heroku”
git push heroku main
```

- **(참고자료) useReducer 사용방법 예제**

client/src/contexts/AuthContext.js

```
import React, {useReducer, createContext} from 'react'

export const AuthContext = createContext()

const initState ={
   auth: false
}

const reducer = (state=null, action) =>{

   switch(action.type){
      case "FETCH_USER":
         return action.payload || false
      default:
         throw new Error()
   }
}

export const AuthContextProvider = props => {
   const [state, dispatch] = useReducer(reducer, initState)
   return <AuthContext.Provider value={[state, dispatch]}>{props.children}</AuthContext.Provider>
}
```

client/src/contexts/AirDataContex.js

```
import React, {useReducer, createContext} from 'react'

export const AirDataContext = createContext()

const state = {
   airData:''
}

const reducer = (state, action) => {
   switch(action.type){
      case "FETCH_DATA":
         return {
            airData:action.payload
         }
      default:
         throw new Error()
   }
}
```

```
export const AirDataContextProvider = props => {
   const [airData, airDispatch] = useReducer(reducer, state)
   return <AirDataContext.Provider value={[airData,
airDispatch]}>{props.children}</AirDataContext.Provider>
}
```

client/src/components/App.js

```
import React from 'react';
import { BrowserRouter, Route } from 'react-router-dom';
import {AuthContextProvider} from '../contexts/AuthContext'
import {AirDataContextProvider} from '../contexts/AirDataContext'

import Header from './Header';
import Display from './Display';
import Landing from './Landing';
import AddForm from './AddForm';

const App = () => {
 return (
   <div className="app">
    <BrowserRouter>
     <AuthContextProvider>
      <AirDataContextProvider>
         <Header />
         <Route exact path="/display" component={ Display } />
         <Route exact path="/add" component={ AddForm }  />
         <Route exact path="/" component={ Landing } />
      </AirDataContextProvider>
     </AuthContextProvider>
    </BrowserRouter>
   </div>
 )
}

export default App;
```

client/src/components/Header.js

```
import React, {useEffect, useContext} from 'react'
import { Link } from 'react-router-dom'
import { AuthContext } from '../contexts/AuthContext'

import axios from 'axios'

const Header = () => {
   const [state, dispatch] = useContext(AuthContext)

   useEffect (()=>{
      async function fetchUser (){
         const res = await axios.get('/api/current_user')
```

```
        dispatch({
        type:"FETCH_USER",
        payload:res.data
        })
      }
      fetchUser()
    },[])

    let auth = state

    const renderContent = () =>{
      switch ( auth ) {
        case false:
          return <a key="1" href="/auth/google">Login With Google</a>
        default:
          return [
            <a key="2" href="/api/logout">Logout</a>
          ]
      }
    }
    return (
      <div className="header">
        <div className="subheader">
          <Link to={auth ?  '/display' : '/'}>
            <p>Display</p>
          </Link>
          <Link to={ auth ? '/add' : '/' }>
            <p>Add</p>
          </Link>
          <Link to={'/'}>
            <p className="para">AQI Information Site</p>
          </Link>
          <p>{renderContent()}</p>

        </div>
      </div>
    )
}

export default Header;
```

client/src/components/Display.js

```
import React, { useEffect, useContext } from 'react';

import { Container, Row } from 'reactstrap';
import DisplayCards from './DisplayCards';

import axios from 'axios'
import { AirDataContext } from '../contexts/AirDataContext';
```

```
const Display = () => {

   const [airData, airDispatch] = useContext(AirDataContext)

   useEffect( ()=> {
      async function fetchData (){
         const res = await axios.get('/airdata/display')
         airDispatch({
            type:'FETCH_DATA',
            payload: res.data
         })
      }
      fetchData()
   },[])

   let air = airData['airData']
   if (air){
      return (
         <div className="heading-menu">
            <h1 className="heading-h1">서울시 공기 정보</h1>
            <p className="heading-p">공기청정도 정보는 매시간마다 갱신됩니다. </p>
            <p className="heading-p">출처: 공공데이터포털</p>
               <Container>
                  <Row>
                     {
                        air.sort((a, b) => a.time <= b.time ? 1:-1).map(
                           (air) =><DisplayCards { ...air } key={air._id} />
                        )
                     }
                  </Row>
               </Container>
         </div>
      )
   }
   return <div> Loading... </div>
}

export default Display;
```

client/src/components/AddForm.js

```
import React, { useState} from 'react';
import { Link } from 'react-router-dom'
import { Button, Form, FormGroup, Label, Input } from 'reactstrap';

import axios from 'axios'

const AddForm = () => {
   const [location, setLocation] = useState("")
   const [dataStatus, setDataStatus] = useState("")
```

```
  const onLocationChange = ( e ) => {
    setDataStatus("")
    setLocation( e.target.value)
  }

  const onUpdateSubmit = async (e) => {
    e.preventDefault();
    setDataStatus(".............Loading")
    try {
      const res = await axios.post('/airdata/update', { station: location } )
      if (res){
        setDataStatus("Data updated! Click Update Button.")
        return res
      }
    } catch ( err ) {
      setDataStatus("No data to update!")
      return err
    }
  }

  const onAddClick = async () => {
    setDataStatus("Loading.....")
  }

  return (
    <div className="addForm">
      <Form onSubmit={onUpdateSubmit}>
        <FormGroup>
        <Label >Enter the location:{location}</Label>
        <Input onChange={onLocationChange} />
        <br></br>

        <p>UpdateDB Status:
          <span id="renderRed">{dataStatus}</span>
        </p>
        </FormGroup>
        {
          <Link to={'/display'}>
            <Button  color="danger" onClick={onAddClick} >Update Data</Button>{' '}
          </Link>
        }
      </Form>
    </div>
  )
}

export default  AddForm;
```

client/src/components/Landing.js

```
import React from 'react'

const Landing = () => {
   return (
      <div className="heading-menu">
         <h1 className="heading-h1">공공데이터 포털</h1>
         <p className="heading-p">시간별/구별 미세먼지 정보</p>
      </div>
   )
}
export default Landing;
```