230104_2반_실습

bcrypt

암호화 방식

Bcrypt

```
bcrypt.hashpw(password, bcrypt.gensalt())
```

- 1999년에 publish된 password-hashing function이다.
- Blowfish 암호를 기반으로 설계된 암호화 함수이며 현재까지 사용중인 **가장 강력한 해시 메커니즘** 중 하나이다.
- 보안에 집착하기로 유명한 OpenBSD에서 사용하고 있다.
- .NET 및 Java를 포함한 많은 플랫폼,언어에서 사용할 수 있다.
- 반복횟수를 늘려 연산속도를 늦출 수 있으므로 연산 능력이 증가하더라도 brute-forece 공격에 대비할 수 있다.

요약

암호화 사용 정리

- ISO-27001 보안 규정을 준수해야하는 상황이면 **PBKDF2**를 사용하자.
- 일반적으로 규정을 준수해야할 상황이 아니면 구현이 쉽고 비교적 강력한 **Bcrypt**를 사용하자.
- 보안 시스템을 구현하는데 많은 비용을 투자할 수 있다면, **Scrpyt**를 사용하자.

Bcrypt 모듈 사용법

Bcrypt 모듈 설치

```
npm install bcrypt
```

비밀번호 암호화하기

- hash는 동기, hashSync는 비동기 방식
- 파라미터로 넣은 숫자 12은 암호화에 사용되는 Salt로, 값이 높을 수록 암호화 연산이 증가한다. 하지만 암호화하는데 속도가 느려진다.

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

const PW = 'abcd1234'
const salt = 12;

// hash 비동기콜백
bcrypt.hash(PW, salt, (err, encryptedPW) => {

})

// hashSync 동기
const hash = bcrypt.hashSync(PW, salt);

// async/await 사용
const hash = await bcrypt.hash(PW, salt)
```

비밀번호 검증하기

```
const PW = 'abcd1234';
const hash = bcrypt.hashSync(PW, 12);

// 비동기 콜백
bcrypt.compare(PW , hash, (err, same) => {
  console.log(same);  //=> true
})

// 동기
const match = bcrypt.compareSync(PW, hash);
if(match) {
  //login
}

// async/await
const match = await bcrypt.compare(PW , hash)
if(match) {
  //login
}
```

passport

passport 모듈

- Passport.js는 Node.js를 위한 **인증 미들웨어**이다.
- OAuth를 이용한 구글, 페이스북, 트위터, 혹은 전통적인 로컬 인증 방법(DB에 저장된 계정 정보로 로그인) 등 여러 가지 인증 방법을 제공하는데 이 개별 인증 방법을 **Strategy**라고 부른다.
- 현재 500개가 넘는 인증 방법을 제공하고 있다.
- 네이버, 카카오도 가능하다.
- Strategy는 passport 의존성과 별개로 설치를 해줘야 한다.

> ℹ️ Passport 공식 웹 사이트
> http://www.passportjs.org/

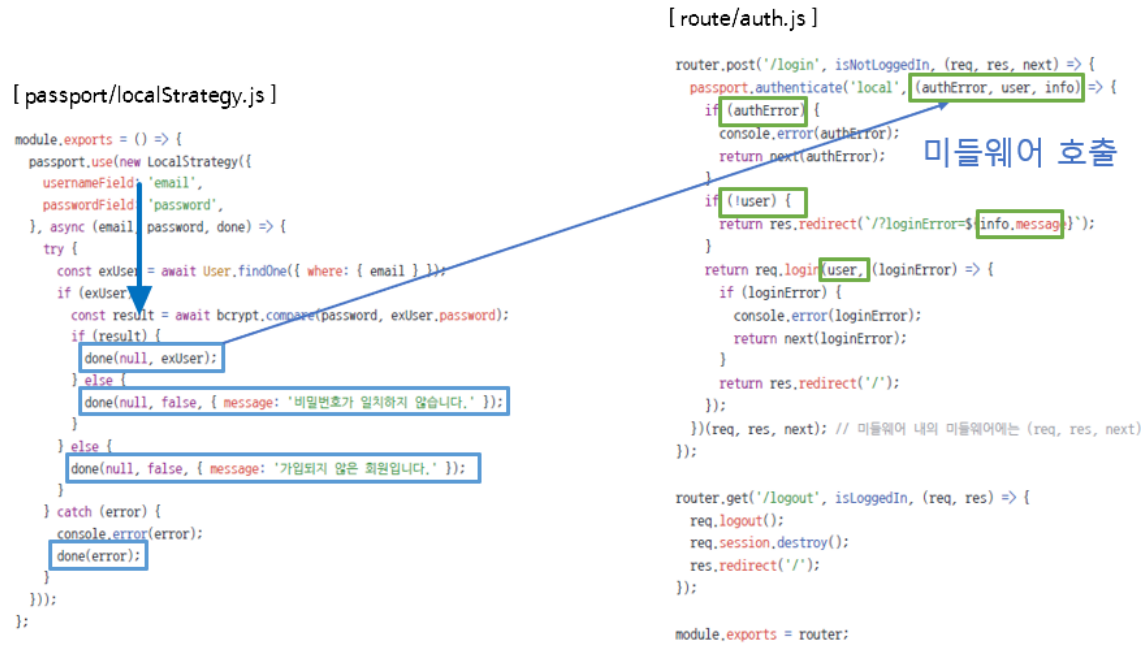
- strategy(전략)에 따른 요청으로 인증하기 위한 목적으로 사용
- **strategy 종류 (로그인 인증 방식)** :
  - Local Strategy(passport-local) : 로컬 DB에서 로그인 인증 방식
  - Social Authentication (passport-kakao, passport-twitter 등) : 소셜 네트워크 로그인 인증 방식
- • API 동작 : 사용자가 passport에 인증 요청 -> passport는 인증 성공/실패시 어떤 제어를 할지 결정

passport 처리 과정

passport 초기 로그인 과정

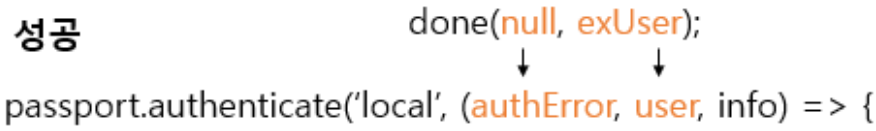
1. 로그인 요청이 라우터로 들어옴.
2. 미들웨어를 거치고, passport.authenticate() 호출
3. authenticate에서 passport/locaStrategy.js 호출

4. 로그인 전략을 실행하고, done()을 호출하면, 다시 passport.authenticate() 라우터로 돌아가 다음 미들웨어를 실행

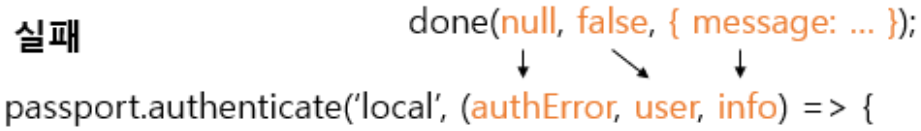


+ done() 함수 인자값에 따른 여러 동작

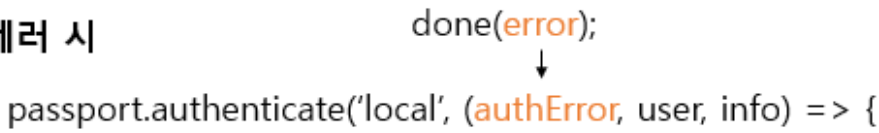
로그인 성공



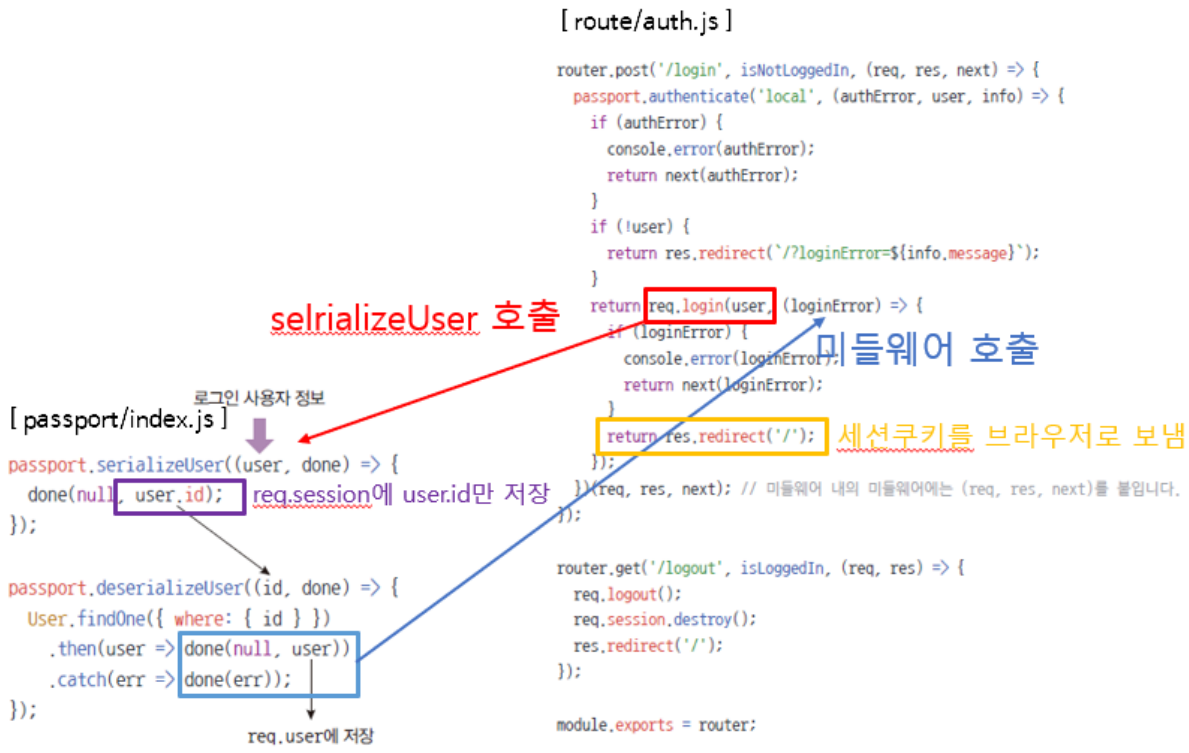
로그인 실패



서버 에러 시



5. done()정보를 토대로, 로그인 성공 시 사용자 정보 객체와 함께 req.login()를 자동으로 호출
6. req.login 메서드가 passport.serializeUser() 호출 (passport/index.js)
7. req.session에 사용자 아이디 키값만 저장 (메모리 최적화를 위해서)
8. passport.deserializeUser()로 바로 넘어가서 sql조회후 req.user 객체를 등록후, done() 반환하여 req.login 미들웨어로 다시 되돌아감.
9. 미들웨어 처리후, res.redirect('/')을 응답하면, 세션쿠키를 브라우저에 보내게 된다.
10. 로그인 완료 처리 (이제 세션쿠키를 통해서 통신하며 로그인됨 상태를 알 수 있다.)



passport 로그인 이후 과정

1. 모든 요청에 passport.session() 미들웨어가 passport.deserializeUser() 메서드를 매번 호출한다.
2. deserializeUser에서 req.session에 저장된 아이디로 데이터베이스에서 사용자 조회
3. 조회된 사용자 전체 정보를 req.user 객체에 저장
4. 이제부터 라우터에서 req.user를 공용적으로 사용 가능하게 된다.

passport 사용법

passport 설치

- 먼저 passport 관련 패키지들을 설치한다.
- 비밀번호 암호화를 위해 bcrypt도 같이 설치 한다.

```
npm install passport passport-local passport-kakao bcrypt
```

passport 예제

app.js

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

...
const passportConfig = require('./passport');

...
```

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

const app = express();
passportConfig(); // 패스포트 설정

...
app.use(cookieParser(process.env.COOKIE_SECRET));
app.use(
   session({
      resave: false,
      saveUninitialized: false,
      secret: process.env.COOKIE_SECRET,
      cookie: {
         httpOnly: true,
         secure: false,
      },
   }),
);
//! express-session에 의존하므로 뒤에 위치해야 함
app.use(passport.initialize()); // 요청 객체에 passport 설정을 심음
app.use(passport.session()); // req.session 객체에 passport정보를 추가 저장
// passport.session()이 실행되면, 세션쿠키 정보를 바탕으로 해서 passport/index.js의 deserialize
User()가 실행하게 한다.

//* 라우터
app.use('/auth', authRouter);

...
```

- passport.initialize 미들웨어는 요청 (req 객체) 에 passport 설정을 심고,
- passport.session 미들웨어는 req.session 객체에 passport 인증 완료 정보를 저장한다.

req.session 객체는 express-session에서 생성하는 것이므로, 따라서 passport 미들웨어는 express-session 미들웨어보다 뒤에 연결해야 한다.

route/middlewares.js

: 자신이 만든 간단한 사용자 미들웨어 함수 파일.

- 로그인을 꼭 해야되는 페이지와, 하지 않아도 되는 페이지 구분하는 미들웨어 작성
- **req.isAuthenticated** 함수를 이용하여 요청에 인증여부 확인

```
//* 사용자 미들웨어를 직접 구현

exports.isLoggedIn = (req, res, next) => {
   // isAuthenticated()로 검사해 로그인이 되어있으면
   if (req.isAuthenticated()) {
      next(); // 다음 미들웨어
   } else {
      res.status(403).send('로그인 필요');
```

```
   }
};

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

로그인에 해당하는 전략을 짜야하는데,

로그인한 사용자는 회원가입과 로그인 라우터에 접근하면 안되며,

로그인을 하지 않은 사용자는 로그아웃 라우터에 접근하면 안된다.

따라서 라우터에 접근 권한을 제어하는 미들웨어가 필요하다.

Passport는 req객체에 isAuthenticated라는 메서드를 자동으로 만들어준다.

로그인이 되어있다면 req.isAuthenticated()가 true일 것이고, 그렇지 않다면 false일 것이다.

따라서 이 메서드를 통해 로그인 여부를 파악할 수 있다.

auth.js 라우터에 로그인 여부를 검사하는 위 미들웨어들을 넣어 원하지 않는 상황들을 방지할 수 있다.

route/auth.js

: 회원가입, 로그인, 로그아웃 처리를 담당하는 라우터. 로그인 인증에 관해 passport폴더로 인증전략을 요청한다.

- /logout 경로로 접근하였을 때 로그인이 되어있지 않다면 접근할 수 없도록 해야 할 것이기 때문에 isLoggedIn을 미들웨어로 넣어주었고,
- /join 경로로 접근하였을 때 로그아웃이 되어있지 않다면 (즉, 현재 로그인한 상태라면) 접근할 수 없도록 해야하기 때문에 isNotLoggedIn을 미들웨어로 넣어주었다.

```
const express = require('express');
const passport = require('passport');
const bcrypt = require('bcrypt');
const { isLoggedIn, isNotLoggedIn } = require('./middlewares'); // 내가 만든 사용자 미들웨
어
const User = require('../models/user');

const router = express.Router();
```

```
//* 회원 가입
// 사용자 미들웨어 isNotLoggedIn을 통과해야 async (req, res, next) => 미들웨어 실행
router.post('/join', isNotLoggedIn, async (req, res, next) => {
   const { email, nick, password } = req.body; // 프론트에서 보낸 폼데이터를 받는다.

   try {
      // 기존에 이메일로 가입한 사람이 있나 검사 (중복 가입 방지)
      const exUser = await User.findOne({ where: { email } });
      if (exUser) {
         return res.redirect('/join?error=exist'); // 에러페이지로 바로 리다이렉트
      }

      // 정상적인 회원가입 절차면 해시화
      const hash = await bcrypt.hash(password, 12);

      // DB에 해당 회원정보 생성
      await User.create({
         email,
         nick,
         password: hash, // 비밀번호에 해시문자를 넣어준다.
      });

      return res.redirect('/');
   } catch (error) {
      console.error(error);
      return next(error);
   }
});


/* ************************************************************************************
***** */


//* 로그인 요청
// 사용자 미들웨어 isNotLoggedIn 통과해야 async (req, res, next) => 미들웨어 실행
router.post('/login', isNotLoggedIn, (req, res, next) => {
   //? local로 실행이 되면 localstrategy.js를 찾아 실행한다.
   passport.authenticate('local', (authError, user, info) => {
      //? (authError, user, info) => 이 콜백 미들웨어는 localstrategy에서 done()이 호출되면 실
행된다.
      //? localstrategy에 done()함수에 로직 처리에 따라 1,2,3번째 인자에 넣는 순서가 달랐는데 그 이
유가 바로 이것이다.

      // done(err)가 처리된 경우
      if (authError) {
         console.error(authError);
         return next(authError); // 에러처리 미들웨어로 보낸다.
      }
      // done(null, false, { message: '비밀번호가 일치하지 않습니다.' }) 가 처리된 경우
      if (!user) {
         // done()의 3번째 인자 { message: '비밀번호가 일치하지 않습니다.' }가 실행
         return res.redirect(`/?loginError=${info.message}`);
      }

      //? done(null, exUser)가 처리된경우, 즉 로그인이 성공(user가 false가 아닌 경우), passpor
t/index.js로 가서 실행시킨다.
```

```
      return req.login(user, loginError => {
         //? loginError => 미들웨어는 passport/index.js의 passport.deserializeUser((id, d
one) => 가 done()이 되면 실행하게 된다.
         // 만일 done(err) 가 됐다면,
         if (loginError) {
            console.error(loginError);
            return next(loginError);
         }
         // done(null, user)로 로직이 성공적이라면, 세션에 사용자 정보를 저장해놔서 로그인 상태가 된
다.
         return res.redirect('/');
      });
   })(req, res, next); //! 미들웨어 내의 미들웨어에는 콜백을 실행시키기위해 (req, res, next)를 붙
인다.
});


/* ************************************************************************************
***** */


//* 로그아웃 (isLoggedIn 상태일 경우)
router.get('/logout', isLoggedIn, (req, res) => {
   // req.user (사용자 정보가 안에 들어있다. 당연히 로그인되어있으니 로그아웃하려는 거니까)
   req.logout();
   req.session.destroy(); // 로그인인증 수단으로 사용한 세션쿠키를 지우고 파괴한다. 세션쿠키가 없다는
 말은 즉 로그아웃 인 말.
   res.redirect('/');
});

module.exports = router;
```

passport/index.js

: 인증 전략을 등록하고, 데이터를 저장하거나 불러올때 이용되는 파일

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

module.exports = () => {
   /*
    * ㅇ 직렬화(Serialization) : 객체를 직렬화하여 전송 가능한 형태로 만드는 것.
    * ㅇ 역직렬화(Deserialization) : 직렬화된 파일 등을 역으로 직렬화하여 다시 객체의 형태로 만드는
 것.
    */

   //? req.login(user, ...) 가 실행되면, serializeUser가 실행된다.
   //? 즉 로그인 과정을 할때만 실행
   passport.serializeUser((user, done) => {
      // req.login(user, ...)의 user가 일로 와서 값을 이용할수 있는 것이다.
      done(null, user.id);
      // req.session객체에 어떤 데이터를 저장할 지 선택.
```

```
      // user.id만을 세션객체에 넣음. 사용자의 온갖 정보를 모두 들고있으면,
      // 서버 자원낭비기 때문에 사용자 아이디만 저장 그리고 데이터를 deserializeUser애 전달함
      // 세션에는 { id: 3, 'connect.sid' : s%23842309482 } 가 저장됨
   });

   //? deserializeUser는 serializeUser()가 done하거나 passport.session()이 실행되면 실행된다.
   //? 즉, 서버 요청이 올때마다 항상 실행하여 로그인 유저 정보를 불러와 이용한다.
   passport.deserializeUser((id, done) => {
      // req.session에 저장된 사용자 아이디를 바탕으로 DB 조회로 사용자 정보를 얻어낸 후 req.user에
 저장.
      // 즉, id를 sql로 조회해서 전체 정보를 가져오는 복구 로직이다.
      User.findOne({ where: { id } })
         .then(user => done(null, user)) //? done()이 되면 이제 다시 req.login(user, ...)
 쪽으로 되돌아가 다음 미들웨어를 실행하게 된다.
         .catch(err => done(err));
   });

   //^ 위의 이러한 일련의 과정은, 그냥 처음부터 user객체를 통째로 주면 될껄 뭘 직렬화/역직렬화를 하는 이유
는
   //^ 세션 메모리가 한정되어있기때문에 효율적으로 하기위해, user.id값 하나만으로 받아와서,
   //^ 이를 deserialize 복구해서 사용하는 식으로 하기 위해서다.

   /* ------------------------------------------------------------------------ */

   local();
   //kakao();
};
```

passport.serializeUser

- strategy에서 로그인 성공시 호출하는 done(null, user) 함수의 두 번째 인자 user를 전달 받아 세션(req.session.passport.user)에 저장
- 보통 세션의 무게를 줄이기 위해, user의 id만 세션에 저장

passport.deserializeUser

- 서버로 들어오는 요청마다 세션정보를 실제 DB와 비교
- 해당 유저 정보가 있으면 done을 통해 req.user에 사용자 전체 정보를 저장 (그러면 다른 미들웨어에서 req.user를 공통적으로 사용 가능)
- serializeUser에서 done으로 넘겨주는 user가 deserializeUser의 첫 번째 매개변수로 전달되기 때문에 둘의 타입은 항상 일치 필요

passport/localStrategy.js

: 로컬 인증전략 절차 코드가 있는 파일이며, 라우터에서 요청이 들어오면 실행된다.

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

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

module.exports = () => {
   //? auth 라우터에서 /login 요청이 오면 local설정대로 이쪽이 실행되게 된다.
   passport.use(
      new LocalStrategy(
         {
            //* req.body 객체인자 하고 키값이 일치해야 한다.
            usernameField: 'email', // req.body.email
            passwordField: 'password', // req.body.password
            /*
            session: true, // 세션에 저장 여부
            passReqToCallback: false,
              express의 req 객체에 접근 가능 여부. true일 때, 뒤의 callback 함수에서 req 인자가
 더 붙음.

              async (req, email, password, done) => { } 가 됨
            */
         },
         //* 콜백함수의  email과 password는 위에서 설정한 필드이다. 위에서 객체가 전송되면 콜백이 실행
된다.
         async (email, password, done) => {
            try {
               // 가입된 회원인지 아닌지 확인
               const exUser = await User.findOne({ where: { email } });
               // 만일 가입된 회원이면
               if (exUser) {
                  // 해시비번을 비교
                  const result = await bcrypt.compare(password, exUser.password);
                  if (result) {
                     done(null, exUser); //? 성공이면 done()의 2번째 인수에 선언
                  } else {
                     done(null, false, { message: '비밀번호가 일치하지 않습니다.' }); //? 실
패면 done()의 2번째 인수는 false로 주고 3번째 인수에 선언
                  }
                  //? done()을 호출하면, /login 요청온 auth 라우터로 다시 돌아가서 미들웨어 콜백을
실행하게 된다.
               }
               // DB에 해당 이메일이 없다면, 회원 가입 한적이 없다.
               else {
                  done(null, false, { message: '가입되지 않은 회원입니다.' });
               }
            } catch (error) {
               console.error(error);
               done(error); //? done()의 첫번째 함수는 err용. 특별한것 없는 평소에는 null로 처
리.
            }
         },
      ),
   );
};
```

LocalStrategy

- local 로그인을 위한 전략

- usernameField / passwordField : 프론트단의 폼태그에서 요청된 값들이 (req.body.*) 오게 된다.
- session: 세션 저장여부
- passReqToCallback: express의 req 객체에 접근 가능 여부true일 때, 뒤의 callback 함수에서 req 인자가 더 붙음 (req, email, password, done) => {}
- 첫 번째 인자에서 id, pw가 전송되면, 두번 째 인자 콜백함수 실행. 실제 전략은 async 함수에서 실행
- DB에서 비교하여 done 함수를 이용해 user 객체 전송, 또는 에러 리턴
- done(에러, 성공, 실패값)
    - 첫 번째 인자: DB조회시 발생하는 서버 에러. 무조건 실패하는 경우에만 사용
    - 두 번째 인자: 성공했을 때 return할 값
    - 세 번째 인자: 사용자가 임의로 실패를 만들고 싶을 때 사용ex) 위에서 비밀번호가 틀렸다는 에러

Session Store

Session Store 개요

Session이란?

- 웹 서버가 클라이언트의 정보를 클라이언트 별로 구분하여 서버에 저장하고 클라이언트 요청 시 Session ID를 사용하여 클라이언트의 정보를 다시 확인하는 기술
- 클라이언트가 정보를 저장하고, 요청 시 정보를 보내는 Cookie와 대조된다.

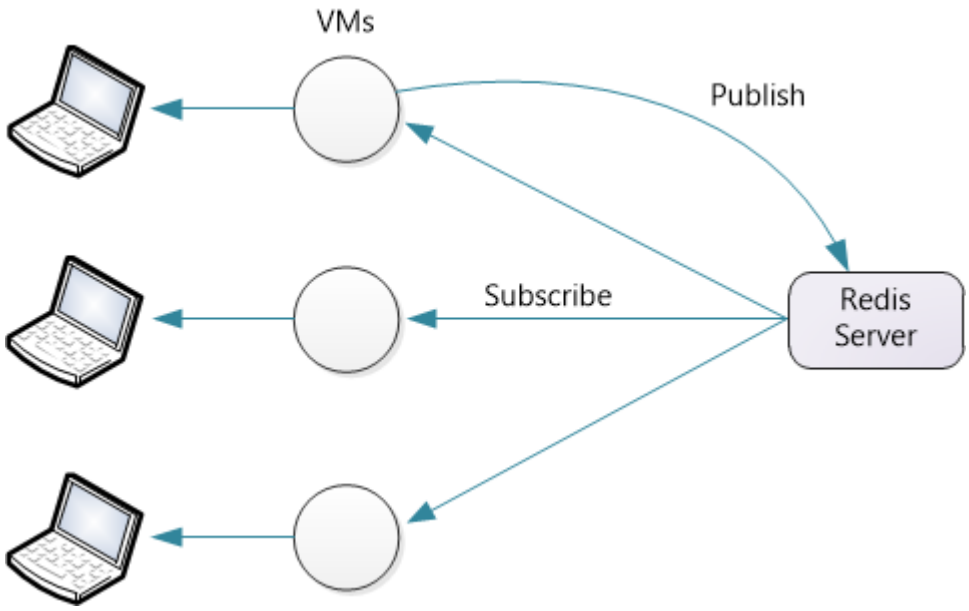
Session 작동 방식

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

Express.js의 session

- express-session 패키지를 사용하여 간단하게 session 동작을 구현할 수 있다.
- 특별한 설정 없이 **자동으로 session 동작을 구현** ⇒ 자동으로 session id를 클라이언트에 전달, session id로 클라이언트 정보 확인

Session과 Session Store



- express-session 패키지는 session을 기본적으로 메모리에 저장함
  - 따라서 현재 구현된 애플리케이션을 종료 후 다시 실행하면, 모든 유저의 로그인이 해제된다.
- 혹은 서버가 여러 대가 있을 경우, 서버 간 세션 정보 공유할 수 없다. ⇒ 세션의 문제점

MongoDB를 Session Store로 사용하기

- connect-mongo 패키지를 이용하면 MongoDB 를 session store로 사용할 수 있다.
- connect-mongo 패키지는 express-session 패키지의 옵션으로 전달 가능
- 자동으로 session 값이 변경될 때 update 되고, session이 호출될 때 find 한다.

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, mongoUrl를 설정해주면 된다.
- 세션 데이터를 몽고 디비를 통해 저장하고 관리하는 기능을 자동으로 수행해 준다.

JWT

JWT(JSON Web Token) 개요

- `JWT`는 유저를 인증(`authentication`)하고 식별(`identification`)하기 위한 `Token`기반 인증방식
- RFC 7519 참고
- `Token`은 `Session`과 달리 서버가 아닌 클라이언트에 저장되기 때문에 메모리나 스토리지 등을 통해 세션을 관리했던 서버의 부담 감소
- `JWT`는 Token 자체에 사용자의 권한 정보나 서비스를 사용하기 위한 정보가 포함(Self-contained)
- 데이터가 많아지면 토큰도 커지며, 토큰이 한 번 발급된 이후 사용자의 정보가 변경되어도 토큰을 재발급 하지 않는 이상 반영되지 않음
- JWT를 사용하면 RESTful과 같은 Stateless인 환경에서 사용자 데이터 통신 가능
- `Session`사용 시 쿠키등을 통해 식별하고 서버에 세션을 저장
- `Token`사용 시 클라이언트에 저장하고 요청시 HTTP 헤더에 토큰을 첨부하는 것만으로 통신 가능
- JWT 사용되는 과정
  1. client가 id, password를 통해 웹서비스 인증
  2. server에 서명된(signed) JWT를 생성하여 client에 응답으로 반환
  3. client가 server에 데이터를 추가적으로 요구할 때 JWT를 HTTP Header에 첨부
  4. server에서 client로부터 수신한 JWT 검증
- JWT는 `JSON` 데이터를 Base64 URL-safe Encode를 통해 인코딩하여 직렬화한 것이 포함
- Token 내부에는 위변조 방지를 위해 개인키를 통한 `전자서명` 존재

- Base64 URL-safe Encode : 일반적인 Base64 Encode에서 URL에서 오류없이 사용하도록 '+', '/'를 각각 '-', '_'로 표현한 방식

JWT(JSON Web Token) 구조

- `JWT`는 `Header`, `Payload`, `Signature`로 구성
- 각 요소는 `.`으로 구분

- `Header`에는 JWT에서 사용할 `Type`, `Hash 알고리즘 종류`

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- `Payload`에는 서버에서 첨부한 `사용자 권한 정보와 데이터`

```
{
  "email": "sample@domain.com",
  "profile": "http://domain.com/image.png",
  "http://domain.com/xxx/yyy/is_admin": true
}
```

- `Signature`에는 Header, Payload를 Base64 URL-safe Encode를 한 후 Header에서 명시한 Hash함수를 적용하고, 개인키(Private Key)로 서명한 `전자서명`이 존재

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

JWT(JSON Web Token) 구현

- jwt 모듈 설치

```
npm install jsonwebtoken
```

- jwt 모듈 불러오기

```
const jwt = require('jsonwebtoken');
```

- 토큰생성(`sign`)

```
const token = jwt.sign(payload, secretKey, option)
```

- decode payload(`verify`)

```
const decoded_token = jwt.verify(token, secretKey);
```