MYSQL 연동

#01. MySQL에 사용자 계정 추가하기

> 아래의 모든 명령어는 root로 수행해야 합니다.

1) 사용자 계정 생성하기

```
create user '아이디'@'접근허용호스트' identified by '비밀번호';
```

- 계정 생성은 mysql에 root로 로그인 한 상태에서만 가능하다.
- 보통 아이디는 사용하고자 하는 데이터베이스의 이름과 동일하게 맞춘다.
- 접근 허용 호스트는 mysql에 접속 가능한 Node.js가 구동중인 머신의 IP주소.
- Node.js와 MySQL이 같은 머신에 설치되어 있는 경우 localhost라고 기입
- 서로 다른 머신에 설치되어 있는 경우 접속 출발지의 IP주소(Node.js설치 장비)를 기입
- 접근허용호스트를 '%'로 지정할 경우 모든 곳에서의 접근을 허용하게 된다.
- 외부에서 접속 할 경우 MySQL이 설치된 운영체제 자체의 방화벽 설정에 따라 접근이 차단될 수 있다.

사용예시

```
create user 'myschool'@'localhost' identified by '123qwe!@#';
create user 'myschool'@'::1' identified by '123qwe!@#';
```

2) 데이터베이스에 대한 권한 부여

```
grant all privileges on 데이터베이스이름.* to '아이디'@'접근허용호스트';
```

- 데이터베이스이름 뒤의 *은 허용하고자 하는 테이블의 의미함.(여기서는 모든 테이블)

사용예시

```
grant all privileges on myschool.* to 'myschool'@'localhost';
```

#02. Node.js 데이터베이스 연동

1) 패키지 설치

```
$ yarn add mysql2
```

2) 데이터베이스 접속

```
import mysql from 'mysql2';

const connectionInfo = {
    host: MYSQL 서버 주소 (다른 PC인 경우 IP주소),
    port: MYSQL 포트번호,
    user: MYSQL의 로그인 할 수 있는 계정이름,
    password: 비밀번호,
    database: 사용하고자 하는 데이터베이스 이름
};

const dbcon = mysql.createConnection(connectionInfo);

dbcon.connect((error) => {
    if (error) {
        // ... 에러처리 ...
        return;
    }

    // ... 접속 성공시 SQL 처리
});
```

3) 접속 성공시 SQL 처리

```
 dbcon.query(sql, input_data, (error, result) => {
    if (error) {
        // ... 에러처리 ...
        dbcon.end(); // 데이터베이스 접속 해제(중요)
        return;
    }

    // 저장결과 확인
    console.log('반영된 데이터의 수: ' + result.affectedRows);
    // UPDATE,DELETE 쿼리의 경우 사용할 수 없는 값임
    console.log('생성된 PK값: ' + result.insertId);
    // 데이터베이스 접속 해제(중요)
    dbcon.end();
});
```

#03. 데이터베이스 접속 모듈 만들기

1) Helper 모듈 복사

```
└── helper
    ├── FileHelper.js
```

```
    ├── LogHelper.js
    └── UtilHelper.js
```

모듈의 정상 동작을 위해 아래 명령으로 패키지 설치

```
$ yarn add dotenv winston winston-daily-rotate-file multer node-thumbnail
```

2) /helper/DBPool.js 모듈 작성

싱글톤 디자인 패턴을 적용하여 모든 Express 컨트롤러들이 하나의 DBPool 객체를 공유하도록 구성한다.

#04. 객체지향 데이터베이스 프로그래밍

1) SQLMapper

- Object와 SQL의 필드를 매핑하여 데이터를 객체화 하는 기술
- 객체와 테이블 간의 관계를 매핑하는 것이 아님
- SQL문을 직접 작성하고 쿼리 수행 결과를 어떠한 객체에 매핑할지 바인딩 하는 방법

사용예시

아래와 같이 SQL 구문의 템플릿을 준비한다.

```
const sql = "INSERT INTO department (dname, loc) VALUES (#{dname}, #
{loc})";
```

템플릿의 치환자에 적용될 값을 객체로 구성한다.

```
const input = {dname: 'hello', loc: 'world'};
```

SQL에 데이터 객체를 맵핑하면 아래와 같이 완성된 SQL문을 얻을 수 있다.

```
INSERT INTO department (dname, loc) VALUES ('hello', 'world')
```

MyBatis

- 원래는 자바에서 사용하는 가장 대표적인 SQL Mapper Framework
- 데이터를 데이터 전달을 위한 객체(DTO, Data Transfer Object)에 담아서 전달하고, SQL 수행 결과를 미리 정의해 놓은 클래스에 대한 객체에 매핑된 상태로 반환받는다.
- 쿼리문을 xml로 분리 가능
- 복잡한 쿼리문 작성 가능
- 데이터 캐싱 기능으로 성능 향상

- but 객체와 쿼리문 모두 관리해야함, CRUD 메소드를 직접 다 구현해야함.

2) ORM

- Object와 DB테이블을 매핑하여 데이터를 객체화하는 기술
- DBMS에 종속적이지 않음
- 복잡한 쿼리문은 처리하기 어려움 (SQL Mapper와 혼용하여 처리 가능함)
- 테이블 구조를 미리 클래스로 정의해 놓아야 하는 번거로움이 있음.

Sequelize

- Node에서 사용하는 대표적인 ORM Framework
- CRUD 메소드 기본 제공
- 쿼리를 만들지 않아도 됨
- 1차 캐싱, 쓰기지연, 변경감지, 지연로딩 제공
- 객체 중심으로 개발 가능

```
const Sequelize = require('sequelize');

class Department extends Sequelize.Model {
    static init(sequelize) {
        return super.init({
            deptno: {
                type: Sequelize.INTEGER,
                allowNull: false,
                unique: true,
            },
            dname: {
                type: Sequelize.STRING(50),
                allowNull: false,
            },
            loc: {
                type: Sequelize.STRING(50),
                allowNull: true,
            }
        }, {
            sequelize,
            timestamps: false,
            underscored: false,
            modelName: 'Department',
            tableName: 'department',
            paranoid: false,
            charset: 'utf8',
            collate: 'utf8_general_ci'
        });
    }
    static associate(db) {}
};

module.exports = Department;
```

#05. MyBatis

SQL문을 XML파일에 미리 정의해 두고, 치환자에 적용될 변수를 전달하여 완성된 SQL문을 반환받는 Framework

Mapper XML 작성 구조

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="Mapper의_식별자">
    <select id="SQL문_식별값">
        SELECT deptno, dname, loc FROM department WHWERE deptno=#{deptno}
    </select>

    <insert id="SQL문_식별값">
        ...
    </insert>

    <update id="SQL문_식별값">
        ...
    </update>

    <delete id="SQL문_식별값">
        ...
    </delete>
</mapper>
```

SQL Mapper에서 변수로 치환되는 부분의 경우 #{변수명} 방식과 ${변수명} 방식이 있음.

#{변수명}경우는 SQL Injection 공격에 대응하기 위해 모든 변수값에 홑따옴표를 강제로 적용하기 때문에 LIMIT절에서 사용할 수 없다.

${변수명}은 전달된 변수값을 원본 그대로 적용하기 때문에 SQL Injection 공격에 노출되어 있지만 원문을 그대로 핸들링 할 수 있으므로 복잡한 SQL 작성이 가능하다.

패키지 참조

```
$ yarn add mybatis-mapper
```

```
import mybatisMapper from 'mybatis-mapper';
```

Mapper 불러오기

mapper는 용도에 따라 여러 개의 파일로 나누어 작성할 수 있다.

주로 단위 기능이나 테이블 단위로 분할하여 작성한다.

```
mybatisMapper.createMapper([
    'mapper-1의 경로',
    'mapper-2의 경로',
    ...
    'mapper-n의 경로'
]);
```

SQL문 조합하기

```
let params = {deptno: 201};
let query = mybatisMapper.getStatement('Mapper의_식별자', 'SQL문_식별값',
params);
console.log(query);
```

조합된 SQL문은 일반 db처리 방식으로 수행한다.

동적 SQL 작성하기

> https://mybatis.org/mybatis-3/ko/dynamic-sql.html