**mybatis 기초**

2020-09-22

이승진

**학습목표**

Spring boot + mybatis 기술을 사용하여 DB 조회, 수정, 삽입, 삭제 기능을 구현한다.

**목차**

[1. 배경지식 2](#_Toc116898300)

[1) ORM (Object Relational Mapping) 2](#_Toc116898301)

[2) JPA와 mybatis 비교 2](#_Toc116898302)

[3) mybatis mapper 4](#_Toc116898303)

[4) Auto Increment 필드 (Identity 필드) 4](#_Toc116898304)

[5) Referential Integrity Constraint 4](#_Toc116898305)

[6) 참조 무결성 제약조건 위반 피하기 7](#_Toc116898306)

[2. mybatis1 프로젝트 수정 8](#_Toc116898307)

[1) Student DTO 클래스 구현 8](#_Toc116898308)

[2) Student Mapper 구현 10](#_Toc116898309)

[3) auto increment 필드와 insert 14](#_Toc116898310)

[4) Student Controller 구현 15](#_Toc116898311)

[5) 액션 메소드의 URL 17](#_Toc116898312)

[6) context path 20](#_Toc116898313)

[7) context path 문제 21](#_Toc116898314)

[8) 정적 컨텐츠 22](#_Toc116898315)

[9) student/list 뷰 구현 24](#_Toc116898316)

[10) student/edit 뷰 구현 25](#_Toc116898317)

[3. 실행 27](#_Toc116898318)

[1) Run As Spring Boot App 27](#_Toc116898319)

[2) 참조 무결성 제약조건 위반 에러 28](#_Toc116898320)

[4. 연습문제 30](#_Toc116898321)

[1) StudentMapper 분리구현 30](#_Toc116898322)

# 배경지식

## ORM (Object Relational Mapping)

ORM 에서 Object 는 객체지향 언어의 객체를 의미한다.

Relational 은 관계형 데이터베이스(Relational Database)의 데이터를 의미한다.

Mapping이 의미하는 것은 객체지향 언어의 객체와 관계형 데이터를 서로 변환해 준다는 것이다.

ORM 이란?

관계형 데이터베이스에서 조회한 데이터를 Java 객체로 변환하여 리턴해 주고,

Java 객체를 관계형 데이터베이스에 저장해 주는

라이브러리 혹은 기술을 말한다.

Java ORM 기술로 유명한 것은

mybatis, Hibernate, JPA 이다.

mybatis와 Hibernate는 오픈소스 프로젝트이고 jar 라이브러리 형태로 제공된다.

JPA(Java Persistence API)는 제품의 이름이 아니고, API 표준의 이름이다.

JPA 표준 규격대로 만들어진 제품 중에서 유명한 것이 Hibernate 오픈소스 라이브러리이다.

우리가 사용하는 Spring JPA에 Hibernate 라이브러리가 포함되어 있다.

우리 나라의 전자 정부 표준 프레임웍에서 Spring mybatis를 채택하고 있기 때문에, 우리 나라 공공 프로젝트에서 mybatis를 사용하는 경우가 많다. 그렇지만 JPA가 좀 더 미래지향적인 기술이기 때문에 점점 JPA를 사용하는 경우가 늘어나고 있다.

## JPA와 mybatis 비교

MySQL, Oracle, SQL Server 등 DBMS 제품 마다 SQL 문법은 조금씩 다르다.

그래서 DBMS 제품을 교체하려면, SQL 문장도 수정해야 한다.

### JPA의 장점

SQL 명령을 구현할 필요가 없다. 그래서 DBMS 제품을 교체하더라도 소스코드를 수정할 필요가 없다.

자동으로 처리되는 부분이 많아서, 구현할 소스코드의 양이 상대적으로 적다.

관계형 데이터베이스가 아니더라도 적용할 수 있다.

### JPA의 단점

복잡한 조회 명령을 구현해야 할 때, 익숙한 SQL 명령으로 구현할 수가 없고, JPA의 고급 기능을 공부해야 한다.

### mybatis의 장점

익숙한 SQL 명령으로 구현할 수 있다.

SQL 문장을 그대로 사용하여 구현하기 때문에, SQL 문장에 익숙한 개발자에게 myBatis가 편하다.

DB 조회 결과를 복잡한 객체 구조로 변환해야 할 때 myBatis 기능이 좋다.

mybatis의 resultMap 기능이 바로 그것이다.

이 기능은 복잡한 보고서(report)를 출력해야 할 때, 특히 유용하다.

데이터베이스 성능 개선을 위해, 어떤 인덱스를 생성해야 하는지 파악하기 위해,

SQL 쿼리들을 분석해야 하는데, 이때 myBatis는 SQL 문장을 그대로 사용하기 때문에,

SQL 쿼리 분석하기 편하다.

### mybatis의 단점

구현할 소스코드의 양이 상대적으로 많다.

관계형 데이터베이스에만 적용할 수 있다.

DBMS 제품을 교체하면 SQL 소스코드를 수정해야 한다.

Oracle, MS SQL Server, mySQL 등 DBMS 마다 SQL 문법이 약간씩 차이가 있다.

그래서 DBMS를 바꾸면 SQL 문도 수정해야 하는 불편함이 있다.

SQL 문을 사용하지 않는 Hibernate, JPA에는 이런 문제가 없다.

## mybatis mapper

데이터베이스 테이블에 대한 SELECT/INSERT/UPDATE/DELETE SQL 명령들을 mybatis mapper에 구현한다.

보통 데이터베이스 테이블 한 개당 mybatis mapper를 한 개를 구현한다.

mybatis mapper 구현 방법은 다음과 같이 두 개이다.

(1) java interface 파일 한 개로 구현

SQL 명령이 비교적 간단한 경우에, SQL 명령을 애너테이션으로 구현한다. (지난 시간의 예제)

(2) Java Interface 파일 한 개와, XML 파일 한 개로 구현.

SQL 명령이 길고 복잡한 경우에,

SQL 명령을 XML 파일에 구현한다.

mapper 메소드를 호출하기 위한 Java 메소드를 Java Interface에 선언하기만 하면 된다.

이 메소드를 구현(implements)할 필요는 없다.

즉 StudentMapper Java Interface만 만들면 되고,

이 인터페이스를 구현(implements)한 Java 클래스를 구현할 필요는 없다.

StudentMapper 인터페이스를 구현한 Java 클래스를

mybatis spring이 자동으로 구현해 주기 때문이다.

## Auto Increment 필드 (Identity 필드)

Student 테이블의 기본키(primary key)는 id 필드이다.

MySQL에서 Student 테이블을 생성할 때, id 필드를 Auto Increment 필드로 지정하였다.

Auto Increment 필드의 값은 1부터 시작하는 일련번호이다.

테이블에 새 레코드를 insert 할 때, 이 필드의 값에 일련번호가 자동으로 부여된다.

Auto Increment 필드의 값이 자동으로 부여되기 때문에,

insert나 update SQL 문에서 이 필드의 값을 저장하지 않아야 한다.

## Referential Integrity Constraint

Student 테이블의 departmentId 필드는 외래키(foreign key) 이다.

이 필드의 값은 department 테이블의 기본키인 id 필드값과 일치해야 한다.

Student 테이블과 Department 테이블을 조인할 때, departmentId 필드를 사용한다.

SELECT s.\*, d.departmentName

FROM student s LEFT JOIN department d ON s.departmentId = d.id

Department 테이블에서 레코드를 한 개 삭제 하려고 할 때,

만약 Student 테이블의 어떤 레코드의 departmentId 필드 값이,

그 삭제하려는 Department 레코드의 id 필드 값과 일치한다면,

삭제는 실패하고 에러가 발생한다.

이 에러를 참조 무결성 제약조건 위반(referential intergity constraint violation) 에러 이라고 부른다.

쉽게 표현하자면, 국어국문학과 소속 학생들이 존재한다면, 국어국문학과를 삭제할 수 없다는 얘기다.

Register 테이블에 외래키인 studentId 필드가 들어있다.

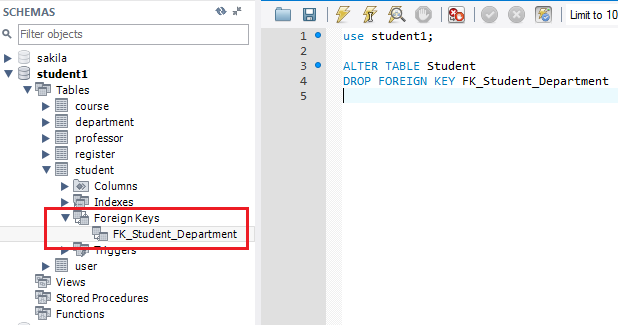
그래서 Student 테이블의 레코드를 삭제하려 할 때, 참조 무결성 제약조건 위반 에러가 발생할 수 있다.

쉽게 표현하자면, 201132050 학생의 수강신청 내역이 존재한다면, 그 학생을 삭제할 수 없다는 얘기다.

데이터베이스가 참조 무결성 제약조건을 실시간 검사해 준다.

참조 무결성을 제약조건을 외래키 제약조건이라고도 부른다.

### 외래키 제약조건 제거하기



Student 테이블에 FK\_Student\_Department 이름의 외래키 제약조건이 이미 설정되어 있다.

이 외래키 제약조건을 삭제하는 명령은 다음과 같다.

|  |
| --- |
| ALTER TABLE student  DROP FOREIGN KEY FK\_Student\_Department; |

### 외래키 제약조건 생성하기

Student 테이블의 departmentId 필드와 Department 테이블의 id 필드 사이에

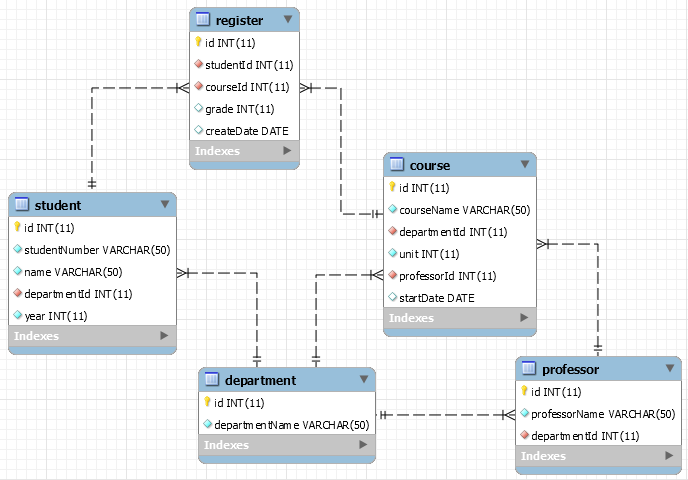
외래키 제약조건을 생성하는 명령은 다음과 같다.

|  |
| --- |
| ALTER TABLE student  ADD CONSTRAINT FK\_Student\_Department  FOREIGN KEY (departmentId) REFERENCES department(id); |

제약조건의 이름은 FK\_Student\_Department 이다.

외래키는 Student 테이블의 departmentId 필드이다.

이 필드는 department 테이블의 id 필드를 참조(references)한다.



register 테이블은 수강신청 레코드를 저장하는 테이블이다.

register 테이블의 studentId 필드는 foreign key 이다.

studentId 필드의 값은 이름 그대로 Student 테이블의 id 필드 값이다.

studentId 필드는 수강신청을 한 학생(student)을 가르킨다.

register 테이블의 courseId 도 foreign key 이다.

수강신청 대상 강좌(course)의 id 이다.

학생(student) 레코드를 삭제하려면, 먼저 그 학생의 수강신청(register) 레코드들부터 삭제해야 한다.

그렇지 않고 학생 레코드부터 삭제하면 참조 무결성 위반 에러가 발생한다.

## 참조 무결성 제약조건 위반 피하기

참조 무결성 제약조건 위반 에러를 피하는 방법은 다음과 같다.

### 먼저 삭제하기

Department 테이블의 레코드를 삭제하기 전에, 먼저 그 레코드를 참조하는 Student 레코드들을 전부 삭제한다.

예:

|  |
| --- |
| DELETE FROM student WHERE departmentId = 2;  DELETE FROM department WHERE id = 2; |

### Cascade Delete 옵션

외래키 제약 조건을 생성할 때, Casecade Delete 옵션을 지정할 수 있다.

이 옵션이 지정된 경우에는, Department 테이블의 레코드를 삭제할 때,

그 레코드를 참조하는 Student 레코드들이 전부 자동으로 삭제된다.

Casecade Delete 옵션 지정 방법

|  |
| --- |
| ALTER TABLE student  ADD CONSTRAINT FK\_Student\_Department  FOREIGN KEY (departmentId) REFERENCES department(id)  ON DELETE CASCADE; |

# mybatis1 프로젝트 수정

## Student DTO 클래스 구현

### src/main/java/net/skhu/dto/Student.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | package net.skhu.dto;  public class Student {  int id;  String studentNumber;  String name;  int departmentId;  int year;  String departmentName;  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getStudentNumber() {  return studentNumber;  }  public void setStudentNumber(String studentNumber) {  this.studentNumber = studentNumber;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getDepartmentId() {  return departmentId;  }  public void setDepartmentId(int departmentId) {  this.departmentId = departmentId;  }  public int getYear() {  return year;  }  public void setYear(int year) {  this.year = year;  }  public String getDepartmentName() {  return departmentName;  }  public void setDepartmentName(String departmentName) {  this.departmentName = departmentName;  }  } |

Student 테이블에서 조회한 데이터를 채울 DTO (Data Transfer Object) 이다.

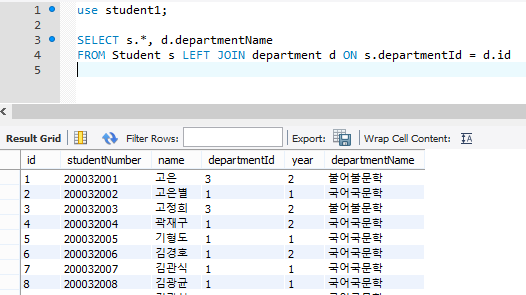
Student 테이블에 departmentName 필드는 없지만,

Student 테이블에서 조회할 때, Department 테이블과 조인하여 departmentName 필드도 조회하기 때문에,

그 조회 결과를 채우기, 위해 departmentName 멤버 변수가 Student 클래스에 추가되었다.

예;

|  |
| --- |
| SELECT s.\*, d.departmentName  FROM student s  JOIN department d ON s.departmentId = d.id |



## Student Mapper 구현

### src/main/java/net/skhu/mapper/StudentMapper.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | package net.skhu.mapper;  import java.util.List;  import org.apache.ibatis.annotations.Delete;  import org.apache.ibatis.annotations.Insert;  import org.apache.ibatis.annotations.Mapper;  import org.apache.ibatis.annotations.Options;  import org.apache.ibatis.annotations.Select;  import org.apache.ibatis.annotations.Update;  import net.skhu.dto.Student;  @Mapper  public interface StudentMapper {  @Select("SELECT \* FROM student WHERE id = #{id}")  Student findOne(int id);  @Select("SELECT \* FROM student WHERE studentNumber = #{studentNumber}")  Student findByStudentNumber(String studentNumber);  @Select("SELECT s.\*, d.departmentName " +  "FROM student s LEFT JOIN department d " +  " ON s.departmentId = d.id ")  List<Student> findAll();  @Insert("INSERT student (studentNumber, name, departmentId, year) " +  "VALUES (#{studentNumber}, #{name}, #{departmentId}, #{year}) ")  @Options(useGeneratedKeys=true, keyProperty="id")  void insert(Student student);  @Update("UPDATE student SET " +  " studentNumber = #{studentNumber}, " +  " name = #{name}, " +  " departmentId = #{departmentId}, " +  " year = #{year} " +  "WHERE id = #{id} ")  void update(Student student);  @Delete("DELETE FROM student WHERE id = #{id}")  void delete(int id);  } |

DB 의 Student 테이블에 대한 조회, 삽입, 수정, 삭제 SQL 명령을 StudentMapper에 구현한다.

StudentMapper는 클래스가 아니고 java interface임에 주목하자.

interface의 메소드는 강제로 public abstract 이다. 그래서 public을 붙이지 않아도 된다.

### findOne

|  |
| --- |
| @Select("SELECT \* FROM student WHERE id = #{id}")  Student findOne(int id); |

이 메소드의 리턴값이 Student 임에 주목하자.

"SELECT \* FROM student WHERE id = #{id}" SQL 명령이 조회한 데이터를 Student 객체에 채워서 리턴한다.

이때, 조회 결과 컬럼 이름과, Student 객체의 속성 이름이 일치해야 한다.

일치하는 이름이 없는 데이터들은 채워지지 않고 무시된다.

대소문자까지 일치해야 한다.

이 메소드의 파라미터 값이, SQL 문장의 #{id} 부분에 채워져서 실행된다.

### findAll

|  |
| --- |
| @Select("SELECT s.\*, d.departmentName " +  "FROM student s LEFT JOIN department d " +  " ON s.departmentId = d.id ")  List<Student> findAll(); |

이 메소드의 리턴값이 List<Student> 임에 주목하자.

@Select("SELECT s.\*, d.departmentName " +

"FROM student s LEFT JOIN department d " +

" ON s.departmentId = d.id ")

위 애너테이션에서 불필요한 공백을 제거하고 아래처럼 구현해도 된다.

@Select("SELECT s.\*, d.departmentName " +

"FROM student s LEFT JOIN department d" +

" ON s.departmentId = d.id")

(주의: 노란색으로 표시한 공백은 꼭 필요하다. 이 공백이 없으면 SQL 문법 오류가 발생한다.)

위 애너테이션의 값을 아래처럼 한 줄로 구현해도 된다.

@Select("SELECT s.\*, d.departmentName FROM student s LEFT JOIN department d ON s.departmentId = d.d")

위 SQL 명령이 조회한 레코드 각각을 Student 객체에 채우고,

그렇게 데이터가 채워진 Student 객체들을 List 객체에 채워서 리턴한다.

### insert

|  |
| --- |
| @Insert("INSERT student (studentNumber, name, departmentId, year) " +  "VALUES (#{studentNumber}, #{name}, #{departmentId}, #{year}) ")  @Options(useGeneratedKeys=true, keyProperty="id")  void insert(Student student); |

insert 메소드의 파라미터 변수가 Student 객체이다.

이 Java 객체의 속성값이, INSERT SQL 명령의 #{...} 부분에 채워져서 실행된다.

Java 객체의 속성명과, #{...} 부분의 이름이 일치해야 속성값이 채워질 수 있다.

#{...} 부분의 이름과 일치하는 Java 객체의 속성이 없을 경우에, 에러가 발생한다.

@Options(useGeneratedKeys=true, keyProperty="id")

INSERT할 테이블의 기본키(primary key) 필드 이름이 "id" 이고,

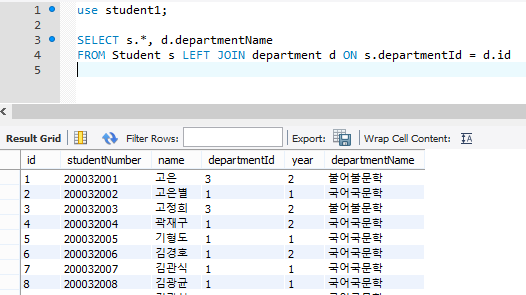
이 필드의 값은 자동으로 생성된다는 선언이다. (Auto Increment 필드)

### 조회 결과 컬럼명 일치

SELECT SQL 명령의 조회 결과가 Java 객체에 자동으로 채워질 때,

조회 결과 컬럼명과 Java 객체의 set 메소드 이름이 일치해야 한다.

조회 결과의 예



|  |  |
| --- | --- |
| 조회 결과 컬럼명 | 자바 객체의 set 메소드 이름 |
| id | setId |
| studentNumber | setStudentNumber |
| name | setName |
| departmentId | setDepartmentId |
| year | setYear |
| departmentName | setDepartmentName |

### mybatis 파라미터: 파라미터 한 개

StudentMapper.java

|  |
| --- |
| @Select("SELECT \* FROM student WHERE id = #{id}")  Student findOne(int id);  @Select("SELECT \* FROM student WHERE studentNumber = #{studentNumber}")  Student findByStudentNumber(String studentNumber);  @Delete("DELETE FROM student WHERE id = #{id}")  void delete(int id); |

#{id}, #{studentNumber} 부분이 mybatis 파라미터이다.

메소드를 호출할 때 전달된 파라미터 값이, SQL 문장의 mybatis 파라미터 부분에 채워져서 SQL 문장이 실행된다.

mybatis 파라미터로 전달할 값이 한 개이고,

값의 타입이 int, long, float, double, boolean 등 기본 자료형이나

String, Date, Time, Timestamp 클래스 객체인 경우에는

위와 같은 방법으로 구현한다.

여기서 Java 파라미터 변수 이름은 중요하지 않다.

중요한 것은 파라미터가 한 개이고,

파라미터 타입이 int, long, float, double, boolean 등 기본 자료형이거나

String, Date, Time, Timestamp 클래스 중 하나의 객체이어야 한다는 점이다.

### mybatis 파라미터: 파라미터 여러 개

StudentMapper.java

|  |
| --- |
| @Insert("INSERT Student (studentNumber, name, departmentId, year) " +  "VALUES (#{studentNumber}, #{name}, #{departmentId}, #{year}) ")  @Options(useGeneratedKeys=true, keyProperty="id")  void insert(Student student);  @Update("UPDATE Student SET " +  " studentNumber = #{studentNumber}, " +  " name = #{name}, " +  " departmentId = #{departmentId}, " +  " year = #{year} " +  "WHERE id = #{id} ")  void update(Student student); |

#{...} 부분이 mybatis 파라미터이다.

여기서 Java 파라미터 변수 이름은 중요하지 않다.

중요한 것은 Java 파라미터 변수의 타입이 Java 클래스이어야 하고

이 클래스의 get 메소드 이름과 mybatis 파라미터의 이름이 일치해야 한다는 점이다.

|  |  |
| --- | --- |
| Student 클래스의 get 메소드 | mybatis 파라미터 |
| getId | #{id} |
| getStudentNumber | #{studentNumber} |
| getName | #{name} |
| getDepartmentId | #{departmentId} |
| getYear | #{year} |

## auto increment 필드와 insert

Student 테이블에 새 레코드를 insert 할 때, auto increment 필드인 id 필드 값은, 자동으로 부여된다.

그래서 insert SQL 문에 id 필드값은 지정하지 않았다.

|  |
| --- |
| @Insert("INSERT Student (studentNumber, name, departmentId, year) " +  "VALUES (#{studentNumber}, #{name}, #{departmentId}, #{year}) ")  @Options(useGeneratedKeys=true, keyProperty="id")  void insert(Student student); |

@Options(useGeneratedKeys=true, keyProperty="id")

애너테이션은, insert SQL 문이 실행되어 새 레코드가 저장된 후에,

값이 자동 부여된, 그 새 레코드의 id 필드값을 Student 객체의 id 속성에 자동으로 채워달라는 선언이다.

예:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Student student = new Student();  student.setStudentNumber("201132091");  student.setName("홍길동");  student.setDepartmentId(1);  student.setYear(1);  System.out.println(student.getId());  studentMapper.insert(student);  System.out.println(student.getId()); |

줄7에서 출력되는 id 값은 0 이다.

Student 객체의 id 멤버 변수에 아직 아무것도 대입되지 않았기 때문이다.

줄9에서 출력되는 id 값은, 줄8에서 저장된 새 레코드의 id 필드값이다.

이 값은 0 이 아니다.

## Student Controller 구현

### src/main/java/net/skhu/controller/StudentController.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | package net.skhu.controller;  import java.util.List;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import org.springframework.ui.Model;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.PostMapping;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestParam;  import net.skhu.dto.Department;  import net.skhu.dto.Student;  import net.skhu.mapper.DepartmentMapper;  import net.skhu.mapper.StudentMapper;  @Controller  @RequestMapping("student")  public class StudentController {  @Autowired StudentMapper studentMapper;  @Autowired DepartmentMapper departmentMapper;  @RequestMapping("list")  public String list(Model model) {  List<Student> students = studentMapper.findAll();  model.addAttribute("students", students);  return "student/list";  }  @GetMapping("create")  public String create(Model model) {  Student student = new Student();  List<Department> departments = departmentMapper.findAll();  model.addAttribute("student", student);  model.addAttribute("departments", departments);  return "student/edit";  }  @PostMapping("create")  public String create(Model model, Student student) {  studentMapper.insert(student);  return "redirect:list";  }  @GetMapping("edit")  public String edit(Model model, @RequestParam("id") int id) {  Student student = studentMapper.findOne(id);  List<Department> departments = departmentMapper.findAll();  model.addAttribute("student", student);  model.addAttribute("departments", departments);  return "student/edit";  }  @PostMapping("edit")  public String edit(Model model, Student student) {  studentMapper.update(student);  return "redirect:list";  }  @RequestMapping("delete")  public String delete(Model model, @RequestParam("id") int id) {  studentMapper.delete(id);  return "redirect:list";  }  } |

## 액션 메소드의 URL

### 학생 편집 화면의 URL

context path가 mybatis1 인 경우 (Run on Server로 실행할 때, 외부 톰캣 서버에서 실행할 때)

http://localhost:8080/mybatis1/student/edit?id=2

context path가 / 인 경우 (Run as Spring Boot App으로 실행할 때, 내장된 톰캣 서버에서 실행할 때)

http://localhost:8080/student/edit?id=2

### 학생 목록 화면의 URL

context path가 mybatis1 인 경우 (Run on Server로 실행할 때)

http://localhost:8080/mybatis1/student/list

context path가 / 인 경우 (Run as Spring Boot App으로 실행할 때)

http://localhost:8080/student/list

### 컨트롤러의 request mapping

|  |
| --- |
| @Controller  @RequestMapping("student")  public class StudentController {  @RequestMapping("list")  public String list(Model model) {  @GetMapping("edit")  public String edit(Model model, @RequestParam("id") int id) { |

컨트롤러 클래스를 구현할 때, URL에서 context path 부분을 생략한다. (하늘색 부분 생략)

context path가 / 이라면

list 액션 메소드를 호출하기 위한 URL => http://서버주소:포트번호/student/list

edit 액션 메소드를 호출하기 위한 URL => http://서버주소:포트번호/student/edit

컨트롤러 클래스에 @RequestMapping을 붙이지 않고

액션 메소드에 URL 전체를 @RequestMapping 값으로 선언할 수도 있다.

|  |
| --- |
| @Controller  public class StudentController {  @RequestMapping("student/list")  public String list(Model model) {  @GetMapping("student/edit")  public String edit(Model model, @RequestParam("id") int id) { |

### 상대 URL, 절대 URL

context path가 / 이고,

학생 목록 페이지의 URL이 http://localhost:8080/student/list 일 때,

즉 현재 URL이 http://localhost:8080/student/list 일 때,

학생 목록 페이지에서 학생 수정 페이지로 넘어가기 위한 URL은?

(1) <a href="edit?id=2"> Ok

이 상대 URL은 현재 URL과 결합해서 아래의 절대 URL 이 된다.

http://localhost:8080/student/edit?id=2

(2) <a href="/edit?id=2"> 틀림

이 절대 URL은 현재 URL과 결합해서 아래의 절대 URL 이 된다.

http://localhost:8080/edit?id=2

(3) <a href="student/edit?id=2"> 틀림

이 상대 URL은 현재 URL과 결합해서 아래의 절대 URL 이 된다.

http://localhost:8080/student/student/edit?id=2

(4) <a href="/student/edit?id=2"> Ok

이 절대 URL은 현재 URL과 결합해서 아래의 절대 URL 이 된다.

http://localhost:8080/student/edit?id=2

위 a 태그가 클릭되면,

웹브라우저는 a 태그의 href 값과, 현재 URL(웹브라우저창의 URL)을 결합하여

full URL을 만들어서, 그 full URL을 서버에 요청한다.

이때, href 값이 상대 URL이면, 현재 URL에서 파일명 부분만 교체된다.

"/" 문자로 시작하는 URL은 절대 URL 이다.

href 값이 절대 URL이면, 현재 URL에서 context path 부분부터 교체된다.

(1) "edit?id=2" 상대URL

현재URL http://localhost:8080/student/list

full URL http://localhost:8080/student/edit?id=2 Ok

(2) "/edit?id=2" 절대URL

현재URL http://localhost:8080/student/list

full URL http://localhost:8080/edit?id=2 틀림

(3) "student/edit?id=2" 상대url

현재URL http://localhost:8080/student/list

full URL http://localhost:8080/student/student/edit?id=2 틀림

(4) "/student/edit?id=2" 절대url

현재URL http://localhost:8080/student/list

full URL http://localhost:8080/student/edit?id=2 Ok

### 설명 생략 (학습 목표가 아님)

context path가 /mybatis1 이고,

학생 목록 페이지의 URL이 http://localhost:8080/mybatis1/student/list 일 때,

학생 목록 페이지에서 학생 수정 페이지로 넘어가기 위한 URL은?

(1) <a href="edit?id=2"> Ok

(2) <a href="/edit?id=2"> 틀림

(3) <a href="/mybatis1/edit?id=2"> 틀림

(4) <a href="student/edit?id=2 }"> 틀림

(5) <a href="mybtis1/student/edit?id=2 }"> 틀림

(6) <a href="/mybatis1/student/edit?id=2 }"> Ok

(1) <a href="edit?id=2">

http://localhost:8080/mybatis1/student/list

=> http://localhost:8080/mybatis1/student/edit?id=2 Ok

(2) <a href="/edit?id=2">

http://localhost:8080/mybatis1/student/list

=> http://localhost:8080/edit?id=2 틀림

(3) <a href="/mybatis1/edit?id=2">

http://localhost:8080/mybatis1/student/list

=> http://localhost:8080/mybatis1/edit?id=2 틀림

(4) <a href="student/edit?id=2 }">

http://localhost:8080/mybatis1/student/list

=> http://localhost:8080/mybatis1/student/student/edit?id=2 틀림

(5) <a href="mybtis1/student/edit?id=2 }">

http://localhost:8080/mybatis1/student/list

=> http://localhost:8080/mybatis1/student/mybtis1/student/edit?id=2 틀림

(6) <a href="/mybatis1/student/edit?id=2 }"> Ok

http://localhost:8080/mybatis1/student/list

=> http://localhost:8080/mybatis1/student/edit?id=2 Ok

## context path

### context path 하드 코딩이 나쁜 이유

학생 수정 페이지의 올바른 절대 URL은

context path가 / 일 때 => <a href="/student/edit?id=2"> 이고,

context path가 /mybatis1 일 때 => <a href="/mybatis1/student/edit?id=2"> 이다.

서버의 설정에 따라 context path가 가끔 바뀌는데,

context path가 바뀌면 소스코드에서 절대 URL을 전부 찾아서 수정해야 하는 번거로움이 있다.

따라서, 소스코드에 절대 URL 값을 사용할 때,

context path 값을 소스코드에 하드 코딩하지 않고, context path 값을 조회해서 출력해야 한다.

|  |
| --- |
| 참고  하드 코딩 => 종종 바뀔 수 있는 값을 소스 코드에 타이핑해서 입력하는 행위.  하드 코딩된 값을 수정하기 위해 소스 코드를 종종 수정해야 하니,  유지보수성이 나빠진다. |

### context path가 종종 바뀌는 이유

서버 하나에 웹 사이트 한 개만 운영할 때 context path는 / 이다.

서버에 하나에 여러 웹 사이트를 운영할 때 context path는 /이름 이다. (예: /mybatis1)

### @RequestingMapping과 context path

@RequestingMapping 어노테이션의 URL에서 context path 부분은

입력하지 않거나, / 으로 입력해야 한다.

예:

@RequestMapping("student/list") Ok

@RequestMapping("/student/list") Ok

@RequestMapping("mybatis1/student/list") 틀림 (context path를 적으면 안 됨)

@RequestMapping("/mybatis1/student/list") 틀림 (context path를 적으면 안 됨)

액션 메소드의 URL을 @RequestMapping, @GetMapping, @PostMapping 어노테이션에 입력할 때는

context path를 생략해야 한다.

## context path 문제

<link rel="stylesheet" type="text/css" href="/common.css" />

지난 강의에서 구현한 department/list.jsp 파일에 포함된 위 link 태그의 href URL에 문제가 있다.

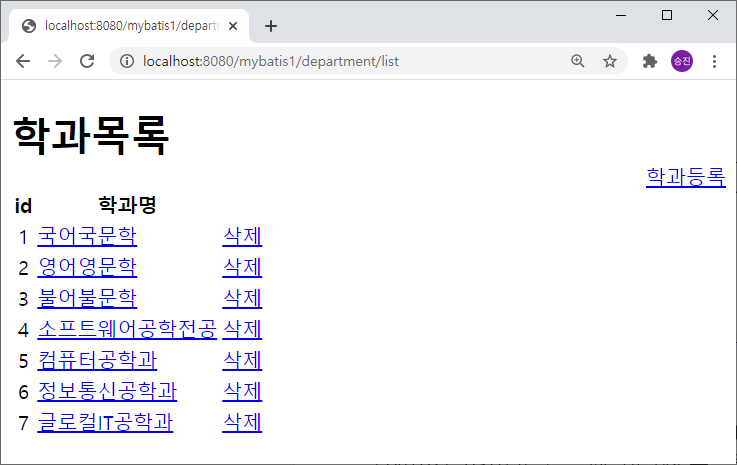
"/common.css" 이것은 절대 URL 이다.

context path가 / 일때 이 URL은 올바르지만,

context path가 /mybatis1 이면 이 URL은 "/mybatis1/common.css" 이어야 한다.

Run As Spring Boot App 메뉴로 실행할 때, context path는 / 이지만,

Run On Server 메뉴로 실행하면, context path는 /mybatis1 이다.

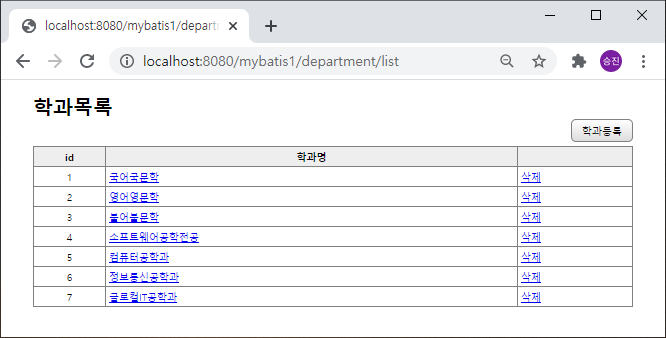


Run On Server 메뉴로 실행한 화면이다.

common.css 서식이 적용되고 있지 않다.

<link rel="stylesheet" type="text/css" href="/mybatis1/common.css" />

위와 같이 수정하고 Run On Server 메뉴로 실행하면,

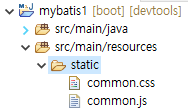


서식이 적용된다.

그런데 이번에는 Run As Spring Boot App 메뉴로 실행하면, 서식이 적용되지 않을 것이다.

이때는 context path가 / 이기 때문이다.

## 정적 컨텐츠



### common.js

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | $(function() {    $("[data-url]").click(function() {  var url = $(this).attr("data-url");  location.href = url;  })  $("[data-confirm-delete]").click(function() {  return confirm("삭제하시겠습니까?");  })  }) |

여러 JSP 파일에 필요한 javascript 코드를 구현 함

|  |
| --- |
| $(function() {  ...  }) |

웹브라우저 창에 현재 페이지 로딩이 완료된 후 ... 부분을 실행한다.

|  |
| --- |
| $("[data-url]").click( ... ) |

... 부분은 함수이어야 한다.

data-url 애트리뷰트가 있는 모든 태그에, 그 태그가 클릭되었을 때 실행될 함수를 등록한다.

클릭되었을 때 실행될 함수를 click event handler 라고 부른다.

|  |
| --- |
| function() {  var url = $(this).attr("data-url");  location.href = url;  } |

data-url 애트리뷰트가 있는 태그가 클릭되면, 이 함수가 실행된다.

|  |
| --- |
| var url = $(this).attr("data-url"); |

클릭된 태그의 data-url 애트리뷰트 값을, url 지역 변수에 대입한다.

|  |
| --- |
| location.href = url; |

url 지역 변수의 값을 location 객체의 href 속성에 대입한다.

웹브라우저 창은 즉시 서버에 url을 요청하게 된다.

즉, 웹브라우저 창의 주소칸에 url 값을 입력하고 엔터를 누른 것과 같다.

|  |
| --- |
| $("[data-confirm-delete]").click(function() {  return confirm("삭제하시겠습니까?");  }) |

data-confirm-delete 애트리뷰트가 있는 모든 태그가 틀릭되면,

아래의 함수가 실행된다.

|  |
| --- |
| function() {  return confirm("삭제하시겠습니까?");  } |

"삭제하시겠습니까?" 라고 묻는 대화상자가 나타난다.

"확인" 버튼을 누르면, 클릭 작업이 계속 진행되고,

"취소" 버튼을 누르면, 클릭 작업이 취소된다.

위 javascript 코드는 jquery 라이브러를 이용한 구현이므로,

jquery 라이브러리가 현재 페이지에 포함되어 있어야 한다.

jquery 라이브러리를 현재 페이지에 포함하기 위한 태그가 필요하다.

|  |
| --- |
| <script src="http://ajax.googleapis.com/ajax/libs/jquery/버전/jquery.min.js"></script> |

### common.css 수정

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | div.container { width: 800px; margin: 10px auto; font-size: 10pt; }  .btn { padding: 0.4em 1em; border: 1px solid gray;  border-radius: 0.5em; background: linear-gradient(#fff, #ddd);  text-decoration: none; color: black;  display: inline-block; }  .btn:active {  -ms-transform: translateY(2px);  -webkit-transform: translateY(2px);  transform: translateY(2px);  background: #ccc; }  table.list { border-collapse: collapse; width: 100%; }  table.list td { padding: 4px; border: 1px solid gray; }  table.list th { padding: 4px; border: 1px solid gray; background-color: #eee; }  input { padding: 4px; }  select { padding: 4px; }  tr[data-url]:hover { cursor: pointer; background-color: #ffe; } |

tr[data-url]:hover { cursor: pointer; background-color: #ffe; }

data-url 애트리뷰트를 포함하는 tr 태그위에 마우스가 위치하면,

마우스가 손가락 모양이 되고, tr 태그 배경을 밝은 노란색이 된다.

## student/list 뷰 구현

### src/main/webapp/WEB-INF/views/student/list.jsp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  <!DOCTYPE html>  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>  <script src="/common.js"></script>  <link rel="stylesheet" type="text/css" href="/common.css" />  <style>  a.btn { float: right; margin: -20px 0 5px 0; }  td:nth-child(1), td:nth-child(5) { text-align: center; }  </style>  </head>  <body>  <div class="container">  <h1>학생목록</h1>  <a href="create" class="btn">학생등록</a>  <table class="list">  <thead>  <tr>  <th>id</th>  <th>학번</th>  <th>이름</th>  <th>학과</th>  <th>학년</th>  </tr>  </thead>  <tbody>  <c:forEach var="student" items="${ students }">  <tr data-url="edit?id=${ student.id }">  <td>${ student.id }</td>  <td>${ student.studentNumber }</td>  <td>${ student.name }</td>  <td>${ student.departmentName }</td>  <td>${ student.year }</td>  </tr>  </c:forEach>  </tbody>  </table>  </div>  </body>  </html> |

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

c: 으로 시작하는 JSTL 확장 태그를 사용하기 위해 필요한 선언.

<c:forEach var="student" items="${ students }">

모델에서 "students" 이름으로 등록된 학생 목록의 학생 객체 각각에 대해서,

학생 객체를 student 변수에 대입하고,

<c:forEach>...</c:forEach> 태그 사이의 내용을 출력한다.

<tr data-url="edit?id=${ student.id }">

이 태그를 클릭하면, "edit?id=..." 상대 URL을 서버에 요청하게 된다.

즉 그 상대 URL로 넘어간다.

## student/edit 뷰 구현

### src/main/webapp/WEB-INF/views/student/edit.jsp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  <%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>  <c:url var="R" value="/" />  <!DOCTYPE html>  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <script src="http://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>  <script src="${R}common.js"></script>  <link rel="stylesheet" href="${R}common.css">  <style>  form { width: 600px; margin: auto;  box-shadow: 0 0 4px lightgray, 2px 2px 4px gray; overflow: auto; }  div.title { font-size: 20pt; padding: 10px; background-color: #eee; }  table { margin: 20px; }  td { min-width: 100; padding: 5px; }  td:nth-child(1) { text-align: right; }  button { margin: 5px 0 20px 20px; }  </style>  </head>  <body>  <div class="container">  <form:form method="post" modelAttribute="student">  <div class="title">학생 ${ student.id > 0 ? "수정" : "등록" }</div>  <table>  <tr>  <td>학번:</td>  <td><form:input path="studentNumber" /></td>  </tr>  <tr>  <td>이름:</td>  <td><form:input path="name" /></td>  </tr>  <tr>  <td>학과:</td>  <td><form:select path="departmentId"  itemValue="id" itemLabel="departmentName" items="${ departments }" /></td>  </tr>  <tr>  <td>학년:</td>  <td><form:input path="year" /></td>  </tr>  </table>  <hr />  <div>  <button type="submit" class="btn">저장</button>  <c:if test="${ student.id > 0 }">  <a href="delete?id=${ student.id }" class="btn" data-confirm-delete>삭제</a>  </c:if>  <a href="list" class="btn">목록으로</a>  </div>  </form:form>  </div>  </body>  </html> |

<c:url var="R" value="/" />

context path를 변수 R에 대입한다.

context path가 / 일 때, R 변수에 대입되는 값은 "/" 이고,

context path가 /mybatis 일 때, R 변수에 대입되는 값은 "/mybatis1/" 이다.

"${R}common.js"

R 변수의 값이 common.js 앞에 출력된다.

R 변수의 값은 context path 이다.

context path가 / 일 때, 위 URL은 "/common.js" 형태로 출력되고,

context path가 /mybatis 일 때, 위 URL은 "/mybatis1/common.js" 형태로 출력되고,

${ student.id > 0 ? "수정" : "등록" }

model에 들어있는 "student" 이름의 객체의,

id 속성 값이 0 보다 크면, "수정" 문자열을 출력하고, 그렇지 않으면 "등록" 문자열이 출력된다.

학생 등록의 경우, "student" 이름의 model attribute 객체의 id 속성 값은 0 이고,

학생 수정의 경우, "student" 이름의 model attribute 객체의 id 속성 값은, 수정할 학생 테이블의 레코드의 id 값이다.

<form:form method="post" modelAttribute="student">

이 확장 태그는, form 태그를 출력한다.

이 입력 폼에서 편집할 데이터 객체는, model에 들어있는 "student" 이름의 객체이다. (model attribute)

<form:input path="studentNumber" />

이 확장 태그는 아래 코드와 동일하다.

<input type="text" name="studentNumber" value="${ student.studentNumber }" />

- form:form 태그에 선언된 modelAttribute 객체는 student 이다. modelAttribute="student"

student model attribute 객체의 studentNumber 속성값이 input 태그의 value에 채워진다.

- input 태그의 name 값은 studentNumber 이다.

form이 submit 될 때, 이 input 태그에 입력된 값이 request parameter로 전달된다.

이 request parameter의 이름은 studentNumber 이다.

<form:select path="departmentId"

itemValue="id" itemLabel="departmentName" items="${ departments }" />

이 확장 태그가 하는 일은 다음과 같다.

- select 태그와 option 태그를 출력한다.

- (items 값) departments 목록의 Department 객체 각각을, option 태그로 출력한다.

(itemValue 값) option 태그의 value는 Department 객체의 id 속성값이다.

(itemLabel 값) option 태그의 텍스트는 Department 객체의 departmentName 속성값이다.

- form:form 태그에 선언된 modelAttribute 객체는 student 이다. modelAttribute="student"

이 student 객체의 departmentId 속성값과 일치하는 option 태그에 selected를 붙인다.

- select 태그의 name 값은 departmenId 이다.

form이 submit 될 때, 선택된 option 태그의 값이 request parameter로 전달된다.

이 request parameter의 이름은 departmentId 이다.

<c:if test="${ student.id > 0 }">

<a href="delete?id=${ student.id }" class="btn" data-confirm-delete>삭제</a>

</c:if>

수정 화면에서만 삭제 버튼이 출력된다.

삭제 a 태그에는 data-confirm-delete 애트리뷰트가 포함되어 있다.

common.js 파일의 javascript 코드에 의해서, 이 a 태그에 click event handler가 등록된다.

이 a 태그를 클릭하면, 아래 대화상자가 나타난다.



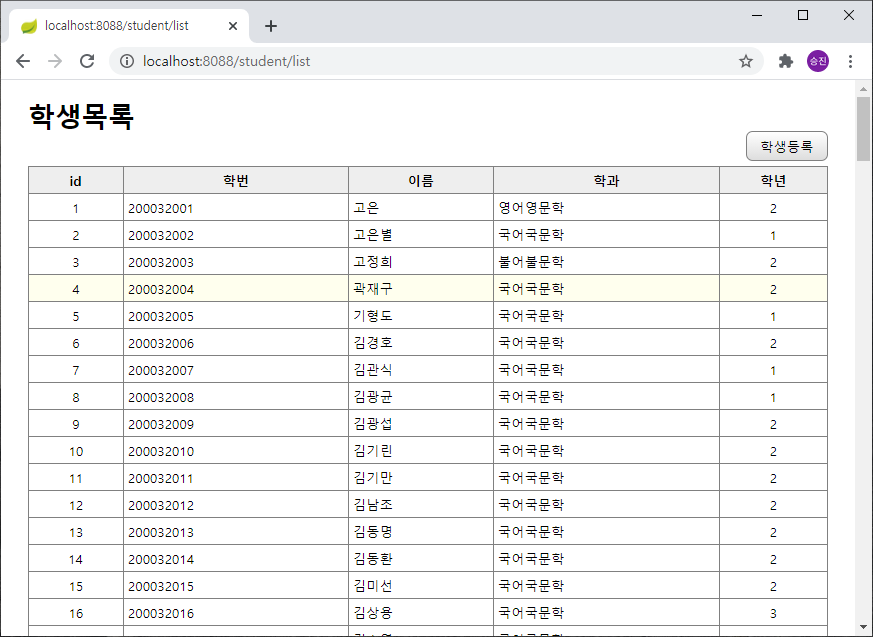
확인 버튼을 클릭하면, a 태그의 href URL이 서버에 요청된다. **"delete?id=6"**

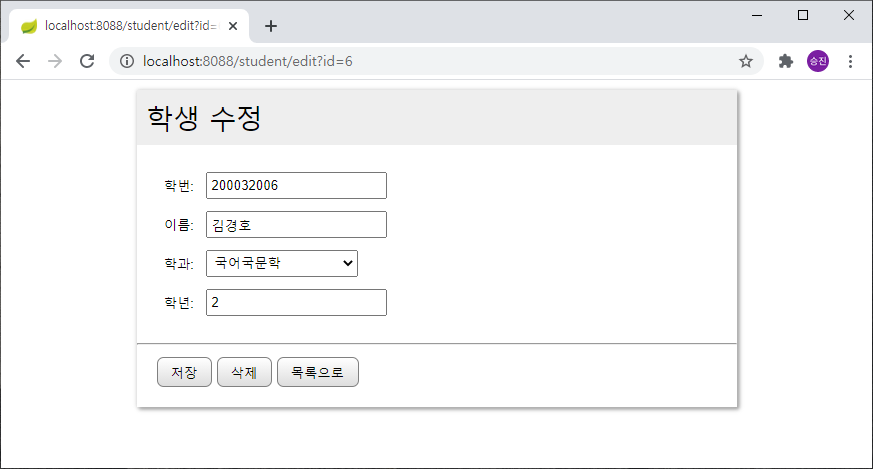
취소 버튼을 클릭하면, a 태그 클릭이 취소되어 아무일도 일어나지 않는다.

# 실행

## Run As Spring Boot App

http://localhost:8088/student/list





## 참조 무결성 제약조건 위반 에러

위 화면에서 **삭제** 버튼을 클릭하면,

StudentMapper에 정의된 아래 SQL 명령이 실행된다.

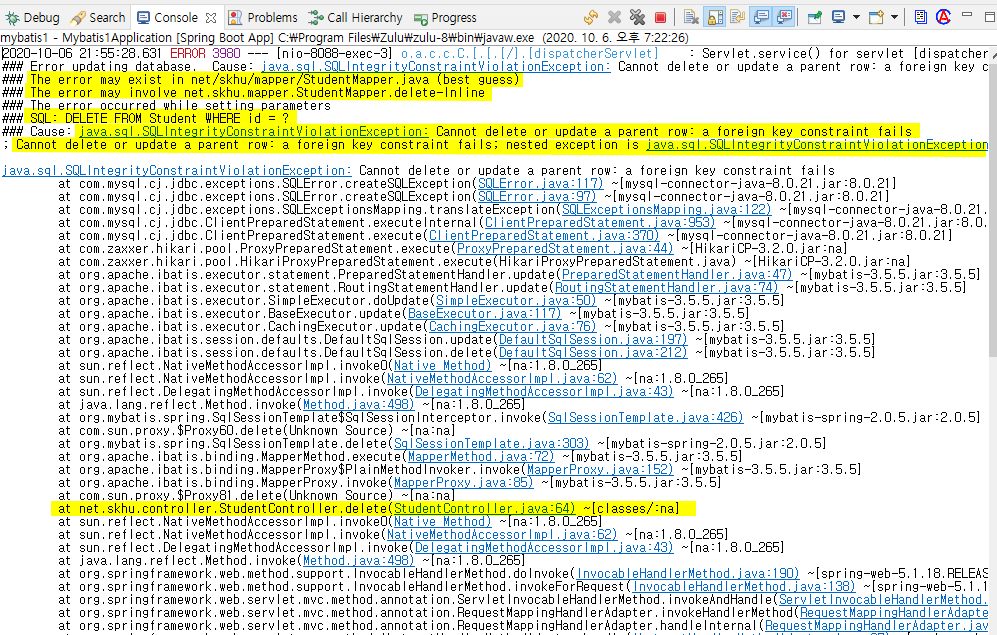
DELETE FROM student WHERE id = #{id}

그런데 '김경호' 학생을 참조하는 수강(register) 테이블의 레코드가 존재하기 때문에,

참조 무결성 제약조건 위반 에러가 발생한다.

이클립스 console 창에 자세한 에러 메시지가 출력된다.

이 에러 메시지의 시작 부분을 자세히 살펴보자.



에러가 발생한 소스코드 위치를 에러 메시지에서 볼 수 있다.

StudentController.java 파일의 줄 64 에서 에러가 발생했다.

|  |
| --- |
| studentMapper.delete(id); |



StudentMapper의 delete 메소드에서 에러가 발생함



**### SQL: DELETE FROM student WHERE id = ?**

에러가 발생한 SQL 명령

**SQLIntegrityConstraintViolationException:**

무결성 제약조건 위반 에러가 발생했다

**Cannot delete or update a parent row: a foreign key constraint fails**

외래키(foreign key) 제약조건 때문에 레코드를 삭제할 수 없다

# 연습문제

## StudentMapper 분리구현

### 수정 전

src/main/java/net/skhu/mapper/StudentMapper.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | package net.skhu.mapper;  import java.util.List;  import org.apache.ibatis.annotations.Delete;  import org.apache.ibatis.annotations.Insert;  import org.apache.ibatis.annotations.Mapper;  import org.apache.ibatis.annotations.Options;  import org.apache.ibatis.annotations.Select;  import org.apache.ibatis.annotations.Update;  import net.skhu.dto.Student;  @Mapper  public interface StudentMapper {  @Select("SELECT \* FROM student WHERE id = #{id}")  Student findOne(int id);  @Select("SELECT \* FROM student WHERE studentNumber = #{studentNumber}")  Student findByStudentNumber(String studentNumber);  @Select("SELECT s.\*, d.departmentName " +  "FROM student s LEFT JOIN department d " +  " ON s.departmentId = d.id ")  List<Student> findAll();  @Insert("INSERT student (studentNumber, name, departmentId, year) " +  "VALUES (#{studentNumber}, #{name}, #{departmentId}, #{year}) ")  @Options(useGeneratedKeys=true, keyProperty="id")  void insert(Student student);  @Update("UPDATE student SET " +  " studentNumber = #{studentNumber}, " +  " name = #{name}, " +  " departmentId = #{departmentId}, " +  " year = #{year} " +  "WHERE id = #{id} ")  void update(Student student);  @Delete("DELETE FROM student WHERE id = #{id}")  void delete(int id);  } |

### 수정 후

src/main/java/net/skhu/mapper/StudentMapper.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package net.skhu.mapper;  import java.util.List;  import org.apache.ibatis.annotations.Delete;  import org.apache.ibatis.annotations.Mapper;  import org.apache.ibatis.annotations.Select;  import net.skhu.dto.Student;  @Mapper  public interface StudentMapper {  @Select("SELECT \* FROM student WHERE id = #{id}")  Student findOne(int id);  @Select("SELECT \* FROM student WHERE studentNumber = #{studentNumber}")  Student findByStudentNumber(String studentNumber);  List<Student> findAll();  void insert(Student student);  void update(Student student);  @Delete("DELETE FROM student WHERE id = #{id}")  void delete(int id);  } |

src/main/java/net/skhu/mapper/StudentMapper.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <?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="net.skhu.mapper.StudentMapper">  <select id="findAll" resultType="net.skhu.dto.Student">  SELECT s.\*, d.departmentName  FROM student s LEFT JOIN department d  ON s.departmentId = d.id  </select>    <insert id="insert" useGeneratedKeys="true" keyProperty="id">  INSERT student (studentNumber, name, departmentId, year)  VALUES (#{studentNumber}, #{name}, #{departmentId}, #{year})  </insert>    <update id="update">  UPDATE student SET  studentNumber = #{studentNumber},  name = #{name},  departmentId = #{departmentId},  year = #{year}  WHERE id = #{id}  </update>  </mapper> |