Tutorial 4

Ringkasan Materi

- String pada annotation (eg: @Delete) merupakan string query database biasa (sql) dimana bisa mendapatkan parameter dengan cara #{namaVariablePadaMethod}
- Untuk memudahkan dalam debuging, kita bisa menggunakan log.info("string log"). Cara kerja hampir sama dengan printf pada bahasa C. Yang membedakannya adalah cara pengaksesan variable. Jika C menggunakan %d atau yang lain (sesuai dengan tipe variable), pada log kita bisa menggunakan {} dan berlaku untuk semua jenis variable.
- Langkah langkah dalam pengerjaan tutorial ini merupakan bagian dari ringkasan materi.

Latihan Menambahkan Delete

Tambahkan method deleteStudent yang ada di class StudentMapper.

Untuk menambahkan fitur delete student berdasarkan npm, maka perlu ditambahkan method deleteStudent terlebih dahulu pada StudentMapper. Masukkan string query delete kedalam annotation @Delete seperti gambar dibawah ini.

```
@Mapper
public interface StudentMapper {
    @Select("select npm, name, gpa from student where npm = #{npm}")
    StudentModel selectStudent (@Param("npm") String npm);

    @Select("select npm, name, gpa from student")
    List<StudentModel> selectAllStudents ();

    @Insert("INSERT INTO student (npm, name, gpa) VALUES (#{npm}, #{name}, #{gpa})")
    void addStudent (StudentModel student);

    @Delete("DELETE FROM student where npm = #{npm}")
    void deleteStudent (@Param("npm")String npm);
}
```

Setelah itu kita bisa menambahkan fungsi deleteStudent pada StudentServiceDatabase seperti gambar dibawah ini.

```
@Override
public void deleteStudent (String npm)
{
    log.info ("student {} deleted", npm);
    studentMapper.deleteStudent(npm);
}
```

Tambahkan validasi agar jika mahasiswa tidak ditemukan tampilkan view not-found

Untuk mencapai hal ini, maka perlu dilakukan pengecekan terhadap data student apakan exist atau tidak. Untuk itu, langkah pertama adalah selectStudent terlebih dahulu dengan npm yang diberikan oleh parameter, kemudian lakukan pengecekan apakah model student tersebut null atau tidak, jika null maka return false.

Fungsi deleteStudent juga harus diubah agar mengembalikan boolean, dimana akan mengembalikan true jika student ditemukan dan false jika tidak ditemukan. Untuk lebih jelasnya dapat dilihat pada gambar dibawah ini.

```
boolean deleteStudent(String npm);
```
perubahan pada interface StudentService

```
@Override
public boolean deleteStudent (String npm)
{
    StudentModel student = selectStudent(npm);
    if(student != null) {
        log.info ("student {} deleted", npm);
        studentMapper.deleteStudent(npm);
        return true;
    } else {
        return false;
    }
}
```
Perubahan pada class StudentServiceDatabase.

Untuk menghandle student yang tidak ditemukan, kita akan menampilkan halaman not-found beserta NPM dari student yang tidak ditemukan tersebut. Detail pengimplementasiannya dapat dilihat pada gambar dibawah ini.

```
@RequestMapping("/student/delete/{npm}")
public String delete (Model model, @PathVariable(value = "npm") String npm)
{
    if (studentDAO.deleteStudent(npm)) {
        return "delete";
    } else {
        model.addAttribute("npm",npm);
        return "not-found";
    }

}
```
Method deleteStudent yang dijalankan mengembalikan boolean yang langsung diproses sebagai conditional untuk menampilkan halaman delete atau not-found.

Setelah dilakukan perubahan, maka akan didapat hasil sebagai berikut

- Ketika delete dengan npm tidak ditemukan

- Ketika delete dengan npm ditemukan dan berhasil delete

Latihan Menambahkan Update

1. Tambahkan method updateStudent pada class StudentMapper

```
    @Update("UPDATE student set name = #{name}, gpa = #{gpa} where npm = #{npm}")
    void updateStudent (StudentModel student);
```

Kita dapat langsung mengakses attribute pada model student tanpa harus melakukan seperti student.getNpm()

2. Tambahkan method updateStudent pada interface StudentService

```
public interface StudentService {
    StudentModel selectStudent(String npm);
    List<StudentModel> selectAllStudents();
    void addStudent(StudentModel student);
    boolean deleteStudent(String npm);
    void updateStudent(StudentModel student);
}
```

3. Tambahkan implementasi method updateStudent pada class StudentServiceDatabase. Jangan lupa tambahkan logging pada method ini.

```
@Override
public void updateStudent(StudentModel student) {
    log.info("updating student with {}, {}", student.getName(), student.getGpa());
    studentMapper.updateStudent(student);
}
```

4. Tambahkan link Update Data pada viewall.html

```
<div th:each="student,iterationStatus: ${students}">
        <h3 th:text="'No. ' + ${iterationStatus.count}">No. 1</h3>
        <h3 th:text="'NPM = ' + ${student.npm}">Student NPM</h3>
        <h3 th:text="'Name = ' + ${student.name}">Student Name</h3>
        <h3 th:text="'GPA = ' + ${student.gpa}">Student GPA</h3>
        <a th:href="'delete/' + ${student.npm}">Delete Data</a>
        <br/>
        <a th:href="'update/' + ${student.npm}">Update Data</a>
        <hr/>
    </div>
```

5. Copy view form-add.html menjadi form-update.html

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>

<title>Update student</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>

<body>

    <h1 class="page-header">Problem Editor</h1>

    <form action="/student/update/submit" method="post">
        <div>
            <label for="npm">NPM</label> <input type="text" readonly="true" name="npm" th:value="${student.npm}" /
        </div>
        <div>
            <label for="name">Name</label> <input type="text" name="name" th:value="${student.name}" />
        </div>
        <div>
            <label for="gpa">GPA</label> <input type="text" name="gpa" th:value="${student.gpa}"/>
        </div>

        <div>
            <button type="submit" name="action" value="save">Save</button>
        </div>
    </form>

</body>

</html>
```

6. Copy view success-add.html menjadi success-update.html.

```
<html>
    <head>
        <title>Update</title>
    </head>
    <body>
        <h2>Data berhasil diupdate</h2>
    </body>
</html>
```

7. Tambahkan method update pada class StudentController

```
@RequestMapping("/student/update/{npm}")
public String update (Model model, @PathVariable(value = "npm") String npm) {
    StudentModel student = studentDAO.selectStudent(npm);
    if (student != null) {
        model.addAttribute("student",student);
        return "form-update";
    } else {
        model.addAttribute("npm",npm);
        return "not-found";
    }
}
```

Pada method update ini dilakukan pengecekan apakah student dengan npm tersebut exist or not, dilakukan dengan cara memanfaatkan method selectStudent(npm) dan melakukan pengecekan apakah variable student tersebut null atau tidak. Jika tidak null, maka akan menampilkan form-update dan membawa model student sebagai initial data pada field

form-update. Jika null, maka akan menampilkan halaman not-found dengan membawa npm sebagai attribut untuk ditampilkan pada halaman not-found.

8. Tambahkan method updateSubmit pada class StudentController

```
@RequestMapping(value = "/student/update/submit", method = RequestMethod.POST)
public String updateSubmit (
        @RequestParam(value = "npm", required = false) String npm,
        @RequestParam(value = "name", required = false) String name,
        @RequestParam(value = "gpa", required = false) double gpa)
{
    StudentModel student = new StudentModel(npm, name, gpa);
    studentDAO.updateStudent(student);
    return "success-update";
}
```

Method updateSubmit ini akan menghandle action dari button submit pada form-update. Disini data pada form akan diolah dan dijadikan instance dari StudentModel yang kemudian digunakan dalam service updateStudent. Kemudian akan ditampilkan halaman success-update.

Latihan Menggunakan Object Sebagai Parameter

Untuk membuat method updateSubmit menjadi lebih rapih, instead of using @RequestParam kita menggunakan @ModelAttribute dimana method updateSubmit hanya menerima parameter dalam bentuk model, yang dalam kasus kita ini adalah StudentModel. Langkah-langkahnya adalah sebagai berikut.

Ubah method updateSubmit menjadi terlihat sebagai berikut

```
@RequestMapping(value = "/student/update/submit", method = RequestMethod.POST)
public String updateSubmit (@ModelAttribute StudentModel student) {
    studentDAO.updateStudent(student);
    return "success-update";
}
```

Pada method sebelumnya, kita perlu membuat instance dari StudentModel dengan menggunakan data dari parameter-parameter npm, name dan gpa. Setelah melakukan perubahan seperti diatas, kita tidak perlu melakukan hal tersebut karena parameter sudah langsung dalam bentuk StudentModel, jadi bisa langsung dijalankan updateStudent.

Ubah form-update menjadi sebagai berikut

Pertama pada tag form tambahkan th:object=”${namaModel}”, kemudian pada setiap tag input, tambahkan th:field=”*{namaAttributeModel}”. Bentuk penerapannya bisa dilihat pada gambar dibawah ini.

```
<h1 class="page-header">Problem Editor</h1>

<form action="/student/update/submit" th:object="${student}" method="post">
    <div>
        <label for="npm">NPM</label> <input type="text" readonly="true" name="npm" th:value="${student.npm}" th:field="*{npm}" />
    </div>
    <div>
        <label for="name">Name</label> <input type="text" name="name" th:value="${student.name}" th:field="*{name}" />
    </div>
    <div>
        <label for="gpa">GPA</label> <input type="text" name="gpa" th:value="${student.gpa}" th:field="*{gpa}"/>
    </div>

    <div>
        <button type="submit" name="action" value="save">Save</button>
    </div>
</form>
```

Pertanyaan

1. Jika menggunakan Object sebagai parameter pada form POST, bagaimana caranya melakukan validasi input yang optional dan input yang required seperti jika menggunakan RequestParam? Apakah validasi diperlukan?

   Answer: terkait validasi, yang dibutuhkan. Pada dasarnya object yang diterima pada method updateSubmit tidak akan pernah null walaupun kita mengosongkan field, karena akan dianggap string “” kosong. Jadi jika yang diinginkan adalah validasi terhadap field yang tidak boleh string “”, maka harus dilakukan pengecekan pada backend seperti gambar berikut

   ```
   @RequestMapping(value = "/student/update/submit", method = RequestMethod.POST)
   public String updateSubmit (@ModelAttribute StudentModel student) {
       if (student.getName() == "" || student.getGpa() == 0) {
           //do your stuff
       }
       studentDAO.updateStudent(student);
       return "success-update";
   }
   ```

2. Menurut Anda, mengapa form submit biasanya menggunakan POST method dibanding GET method? Apakah perlu penanganan berbeda di header atau body method di controller jika form di post dikirim menggunakan method berbeda?

   Answer: karena ketika kita submit, kita akan meng-alter data pada server dan berdasarkan ketentuan yang telah ditentukan, bahwa ketika kita ingin meng-alter data pada server kita harus menggunakan POST instead of GET. Disamping itu, jika menggunakan GET maka data yang dikirim akan terekspose pada url sehingga tidak akan aman.

   Ref:
   https://stackoverflow.com/questions/1254132/so-why-should-we-use-post-instead-of-get-for-posting-data?lq=1

3. Apakah mungkin satu method menerima lebih dari satu jenis request method, misalkan menerima GET sekaligus POST? Tidak, karena pada @RequestMapping, method hanya menerima “sebuah value RequestMethod”, bukan “array of RequestMethod”. Namun jika yang ditanya apakah satu request bisa melakukan GET dan POST sekaligus, itu mungkin terjadi namun hal ini cukup tricky.