1. LATIHAN MENAMBAHKAN *DELETE*

Method yang Anda buat pada Latihan Menambahkan Delete, jelaskan

Penjelasan :

Method delete diimplementasikan pada kelas StudentController.java

- Pertama, buat *interface delete* di kelas StudentMapper.java
  StudentMapper ini terhubung langsung dengan *database*, jadi perlu dibuat *interface delete* yang untuk *query* data dari *database*.

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

  karena kita akan menghapus objek *student*, maka kondisi sql untuk ke database nya yaitu *delete* dari tabel *student* dimana npm sama dengan npm yang akan kita hapus. Lalu *void* karena *method* ini tidak mengembalikan suatu nilai. Dan kita butuh parameter npm, karena menghapus berdasarkan npm.

- Kedua, pindah ke *interface* StudentService.java
  StudentService merupakan *interface* untuk kodingan bukan database nya. Jadi hanya perlu dibuat :

```
void deleteStudent (String npm);
```

- Ketiga, pindah ke kelas StudentServiceDatabase.java
  Kelas ini mengimplementasi *behavior* dari *interface* StudentService. Sehingga perlu *override* nilai methodnya dengan cara :

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

  perlu dibuat *log.info* untuk *debugging* seperti *syso* di *java* dimana pesan akan tercetak di *console.* Jangan lupa memanggil *method deleteStudent(npm).*

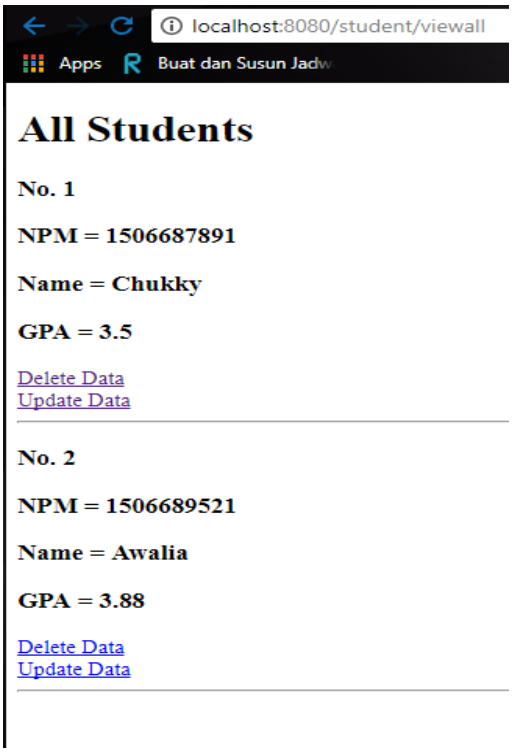
- Selanjutnya ke kelas StudentController.java
  Di kelas ini, *logic* untuk *delete*. Karena akan menghapus *student* dengan npm yang ditentukan maka perlu *value* npm pada annotasi *@PathVariable*
  Logic delete nya, cari *student* dengan npm yang ingin dihapus dengan memanfaatkan *method* selectStudent(). Ada kondisi,

Jika *student* dengan npm tersebut ada (berarti tidak *null*) maka manfaatkan *method* deleteStudent() dengan cara memanggil objek *studentDAO*. StudentController.java merefer ke *interface* StudentService dengan nama *studentDAO*. Jadi, harus panggil dari namanya.

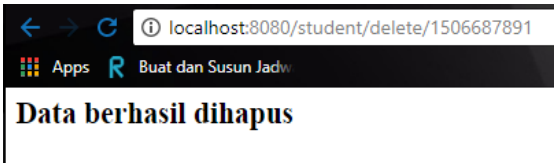
Selain itu berarti *student* dengan npm tersebut tidak ada. Maka akan ditampilkan pesan yang berada di halaman html *not-found.*

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

- Berikut *screenshoot* untuk latihan *delete*

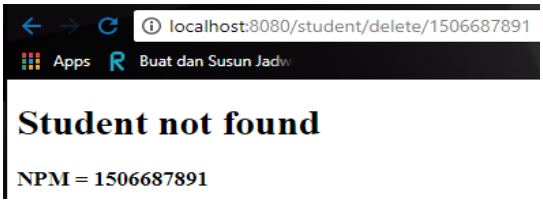


Lalu klik *link "Delete Data"*



Melihat apakah data *student* dengan npm 1506687891 berhasil dihapus

Contoh tampilan jika dilakukan *delete* npm 1596687891 yang kedua kalinya



2. LATIHAN MENAMBAHKAN *UPDATE*

Method yang Anda buat pada Latihan Menambahkan Update, jelaskan

Penjelasan :

Method *update* dibuat di kelas StudentController.java

```
@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";
            }
    }
```

Logic nya mirip dengan *delete*, hanya saja kondisi jika *student* nya ada (tidak *null*) dia memanggil *page* html *form-update*. Jadi *method update* ini hanya untuk menampilkan *form* untuk *update*. Belum

submit *value*. Sementara kalau *student* nya tidak ditemukan, maka yang ditampilkan adalah *page* html *not-found*.

Langkah awalnya sama dengan *delete*:

Buat di StudentMapper untuk *query* ke *database*

```
@Update("UPDATE student s SET name = #{name}, gpa = #{gpa} WHERE s.npm = #{npm}")
        void updateStudent (StudentModel student);
```

Kemudian, buat *method* kosong *update* di *interface* StudentService, lalu *override* nilainya di kelas StudentServiceDatabase.java.

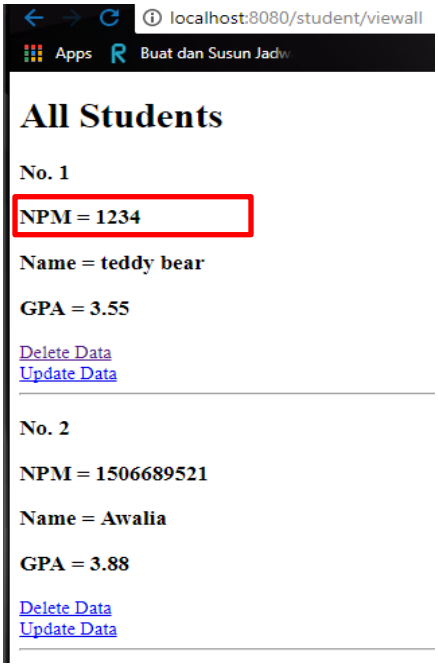
Selanjutnya ke kelas StudentController.java

```
@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";
    }
```

Nah, *method* di atas adalah *method* updateSubmit() yang diisi dengan *logic* submit ketika ada parameter yang akan di *update*. Parameter dibuat dengan annotasi *@RequestParam. Logic* nya, ketika ingin meng-*update* data *student*, maka perlu dibuat objek *student* baru dengan parameter yang dibutuhkan. Lalu panggil *method* updateStudent() dengan parameter si *student* baru. Lalu *return* nya untuk memanggil *page* html yang berisi pesan bahwa *update* berhasil dilakukan. Jadi, *method* ini melakukan pengecekan terhadap *input* berupa npm, nama, dan gpa. Lalu jika semua terisi maka *method* ini akan melakukan *update* dan dimasukkan ke dalam *database.*

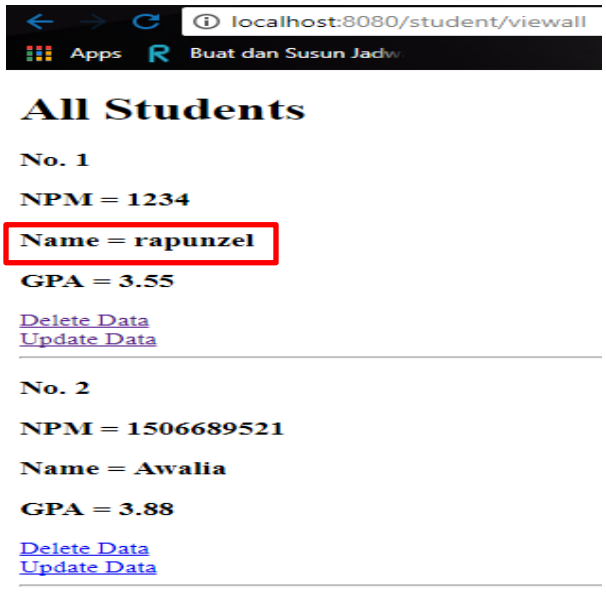
Berikut *screenshoot update student:*



Lalu klik *link "Update Data"* pada *student dengan npm = 1234*

Klik *button update*

Melihat seluruh data untuk memastikan data berhasil di *update (*data dengan npm 123, nama menjadi rapunzel*)*



2. LATIHAN MENGGUNAKAN OBJEK SEBAGAI PARAMETER

Method yang Anda buat pada Latihan Menggunakan Object Sebagai Parameter, jelaskan

Penjelasan :

Latihan ini, *method* updateSubmit() pada kelas StudentController.java diubah

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

Awalnya, menggunakan annotasi *@RequestParam* sebanyak parameter yang dibutuhkan. Namun ada kendala ketika terdapat suatu *form* dengan *field* yang banyak. Tentu membutuhkan parameter yang sangat banyak menyesuaikan kebutuhan sehingga kode menjadi tampak berantakan dan kurang efisien. Solusi yang telah dipelajari pada tutorial kali ini, *SpringBoot* dan *Thymeleaf* memungkinkan agar *method* updateSubmit menerima parameter berupa model *StudentModel*. Jadi, semua *field* dalam *form* dibungkus ke dalam *StudentModel* tersebut.

Lalu pindah ke kelas form-update.html
Di halaman html ini, ditambahkan beberapa keperluan untuk menyesuaikan *StudentModel* pada method updateSubmit. Berikut kode nya :

```
<!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">Update Student</h1>

  <form action="/student/update/submit" method="post" th:action="@{/student/update/submit}"
th:object="${student}" >
            <div>
                  <label for="npm">NPM</label> <input type="text" name="npm" readonly="true"
th:value="${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">Update</button>
            </div>

    </form>

</body>

</html>
```

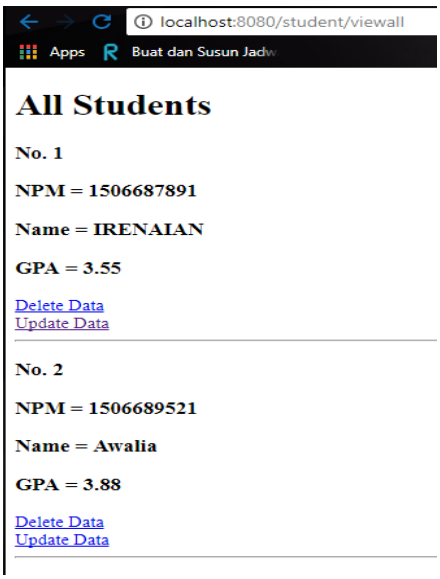
Yang diubah:

1. Bagian *tag form,* tambahkan
   th:action="@{/student/update/submit}" th:object="${student}"

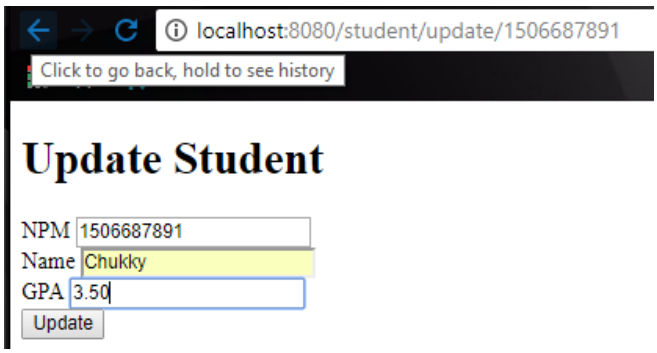
   th:action bertugas *page* ini akan merefer ke alamat mana.
   th:object sebagai pembungkusnya yang dibuat pada *method* updateSubmit, digunakan lagi di *form* ini.

2. Di setiap bagian label tambahkan kode:
   th:field="*{[nama]}"
   artinya *field* ini untuk mengisi objek *Student* berdasarkan *value* nya (npm,name, gpa).

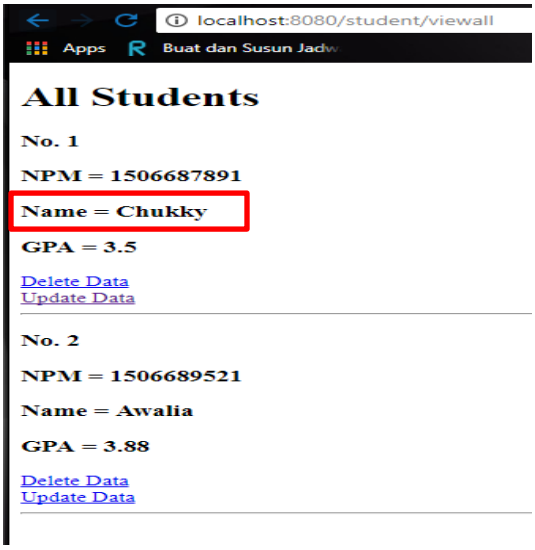
Berikut *screenshoot* latihan menambahkan objek sebagai parameter:



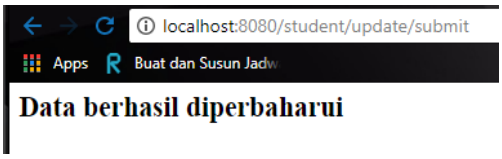
edit student dengan npm 1506687891 Dari GPA = 3.55 menjadi 3.50 & Name = IRENAIAN menjadi Chukky



Lihat semua data



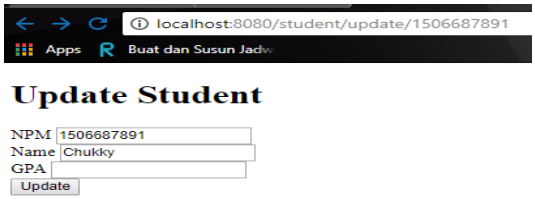
Klik *button update*



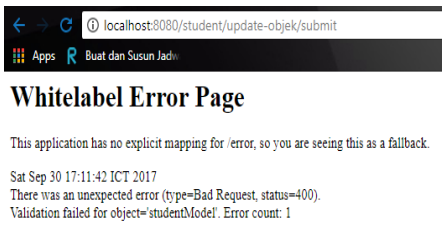
3. PERTANYAAN

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? Asumsikan input pada form Anda tidak menggunakan attribute required sehingga butuh validasi di backend.
   Jawab :

   Validasi dibutuhkan, karena *input* tidak boleh ada yang kosong. Baik itu nama, gpa bahkan npm. Semua harus terisi. Jika salah satu *input* ada yang kosong maka akan mengembalikan halaman *error*. Berikut *screenshoot* jika *field gpa* kosong *:*



   Maka hasilnya :



```
Caranya bisa dengan th:if="${usernameExists}"
 atau th:text="${#strings.isEmpty(string)}"
```

Source : http://www.baeldung.com/spring-thymeleaf-3-expressions

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?
   Jawab :
   Menurut saya, *input form* submit biasanya memang menggunakan POST. Kedua *method* tersebut memiliki perbedaan yang tampak yakni pada url. Pada *method POST* tidak ditampilkan nilai variabel pada url sementara *method GET* menampilkan nilai variable yang dikirimkan.
   Tidak diperlukan penanganan berbeda dalam implementasi kedua *method* tersebut.

3. Apakah mungkin satu method menerima lebih dari satu jenis request method, misalkan menerima GET sekaligus POST?
   Menurut saya, hal tersebut tidak bisa. Karena ketika menggunakan GET untuk mengambil nilai dari variabel di url, tidak bisa sekaligus melakukan POST. Keduanya memiliki tugas hamper mirip namun berbeda.

4. WRITE-UP MATERI

Pada tutorial 4, saya mempelajari kembali terkait GET dan POST walaupun sudah pernah dibahas pada mata kuliah PPW. Tutorial kali ini membahas tentang *library* yang dipakai yaitu Lombok, MyBatis, dan MySQL. Salah satu fungsi *ibrary* Lombok yaitu sebagai *helper annotation* pada *project*. Kemudian *library* MyBatis memiliki fungsi untuk menghubungkan *project* dengan MySQL. MyBatis membantu untuk melakukan koneksi dan *generate query* dengan *helper annotation*. Kita juga bisa *debugging* dengan menggunakan log dan selebihnya tidak jauh berbeda dengan tutorial 3, Alhamdulillah. Menambah wawasan terkait penggunaan controller, interface, mapper, dan database. Tutorial 3 tidak menggunakan *database,* sekarang belajar menggunakan *database*. Untuk mengakses *database* diperlukan bantuan dari mapper dan *interface service* yang kemudian diteruskan ke *controller.*