Ruth Nolytha Putri S
1606955006

Latihan Menambahkan Delete

1. Method **deleteStudent** yang ada di class **StudentMapper**

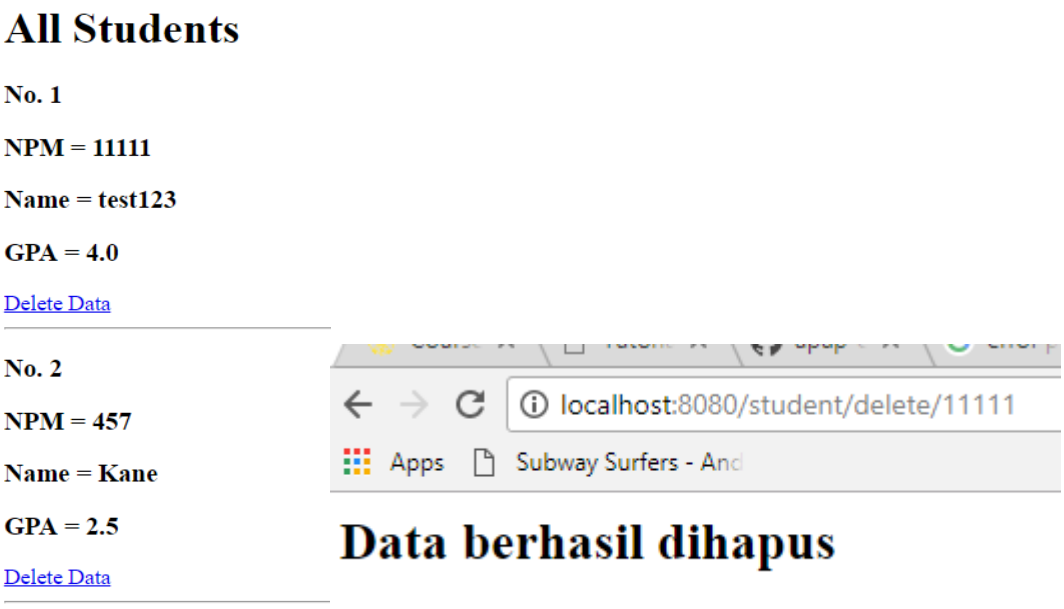
```
@Delete("DELETE FROM student where npm = #{npm}")
void deleteStudent(@Param("npm") String npm);
```

2. Method **deleteStudent** yang ada di class **StudentServiceDatabase**

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

3. Method **delete** pada class **StudentController**

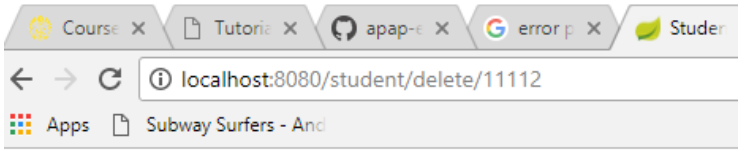
```
@RequestMapping("/student/delete/{npm}")
public String delete (Model model, @PathVariable(value = "npm") String npm)
{
    StudentModel student = studentDAO.selectStudent (npm);

    if (student != null) {
        studentDAO.deleteStudent (npm);
        return "delete";
    } else {
        model.addAttribute ("npm", npm);
        return "not-found";
    }

}
```




```
2018-03-10 00:37:49.831  INFO 9472 --- [nio-8080-exec-1] c.e.service.StudentServiceDatabase       : select all students
2018-03-10 00:43:02.458  INFO 9472 --- [nio-8080-exec-6] c.e.service.StudentServiceDatabase       : select all students
2018-03-10 00:47:04.317  INFO 9472 --- [nio-8080-exec-8] c.e.service.StudentServiceDatabase       : student 11111 deleted
```

Jika dilihat pada console IDE, akan terdapat log yang menyatakan bahwa student dengan npm yang dimaksud telah dihapus dari database. (Seperti pada gambar diatas)

Ketika npm yang menjadi parameter tidak ada pada database, maka controller akan mengembalikan view not-found. Seperti terlihat pada gambar dibawah

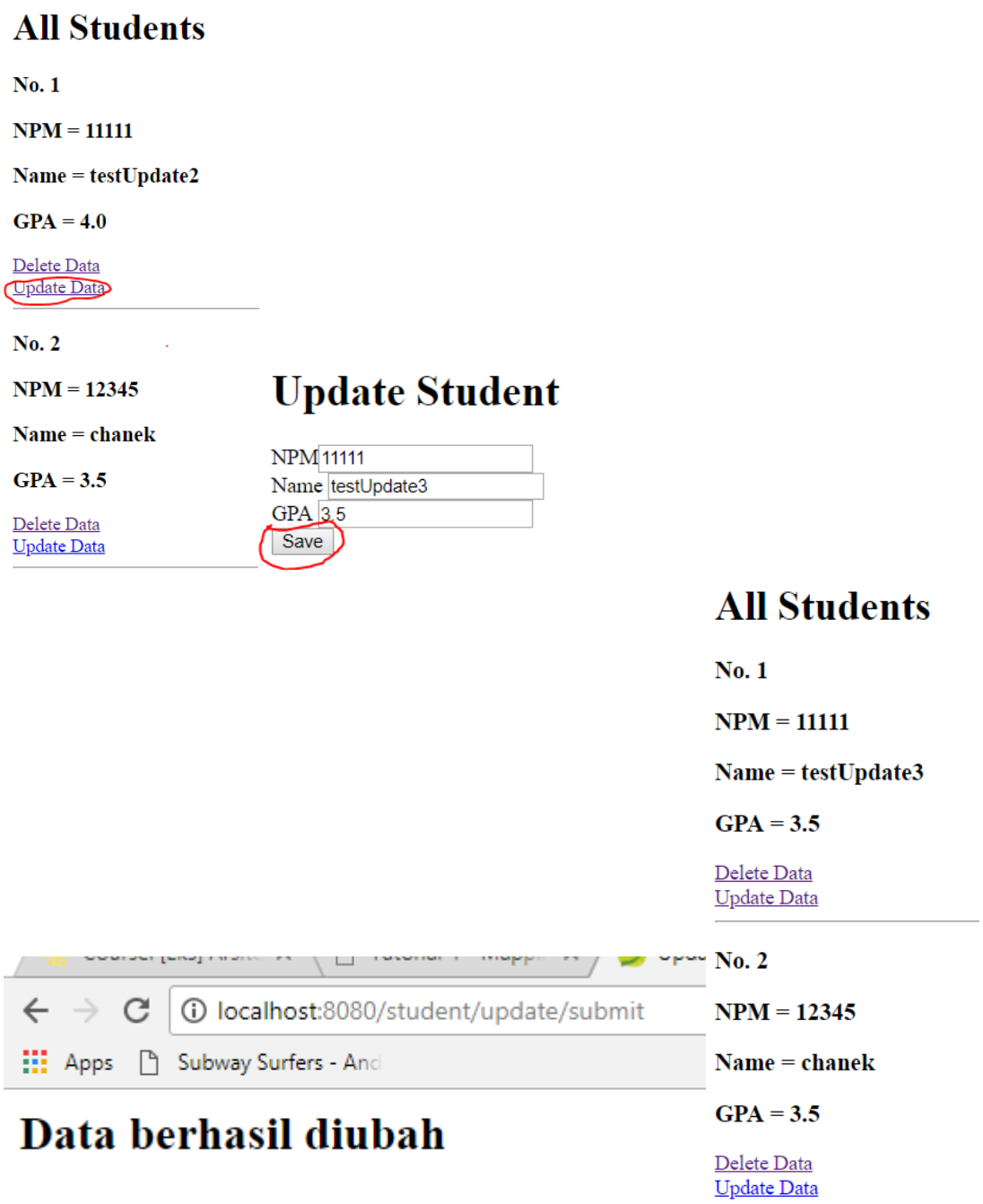


Student not found

NPM = 11112

Latihan Menambahkan Update

1. Method **updateStudent** pada class **StudentMapper**
```
@Update("UPDATE student SET name = #{name}, gpa = #{gpa} where npm = #{npm}")
void updateStudent(StudentModel student);
```
2. Method **updateStudent** pada interface **StudentService**
```
void updateStudent (StudentModel student);
```
3. Tambahkan implementasi method **updateStudent** pada class **StudentServiceDatabase**
```
@Override
public void updateStudent (StudentModel student)
{
    log.info ("student " + student.getNpm() + " updated");
    studentMapper.updateStudent(student);
}
```
4. Tambahkan link Update Data pada viewall.html
```
<a th:href="'/student/update/' + ${student.npm}" > Update Data</a><br/>
```
5. Method **update** 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";
}
```
6. Jalankan program



Cara untuk melakukan update student dilakukan berdasarkan langkah-langkah diatas. NPM student yang ingin diupdate di parse menjadi parameter ketika dilakukan klik pada link **update data**. Tombol save akan men-trigger aksi updateSubmit pada studentController untuk menyimpan data yang ada pada form-update kedalam database. Setelah data tersimpan, controller akan me-render view ke view success-update yang memberikan informasi bahwa data mahasiswa yang dimaksud telah berhasil diubah. Method viewAll akan mengembalikan semua list student yang ada pada database

Latihan Menggunakan Object Sebagai Parameter

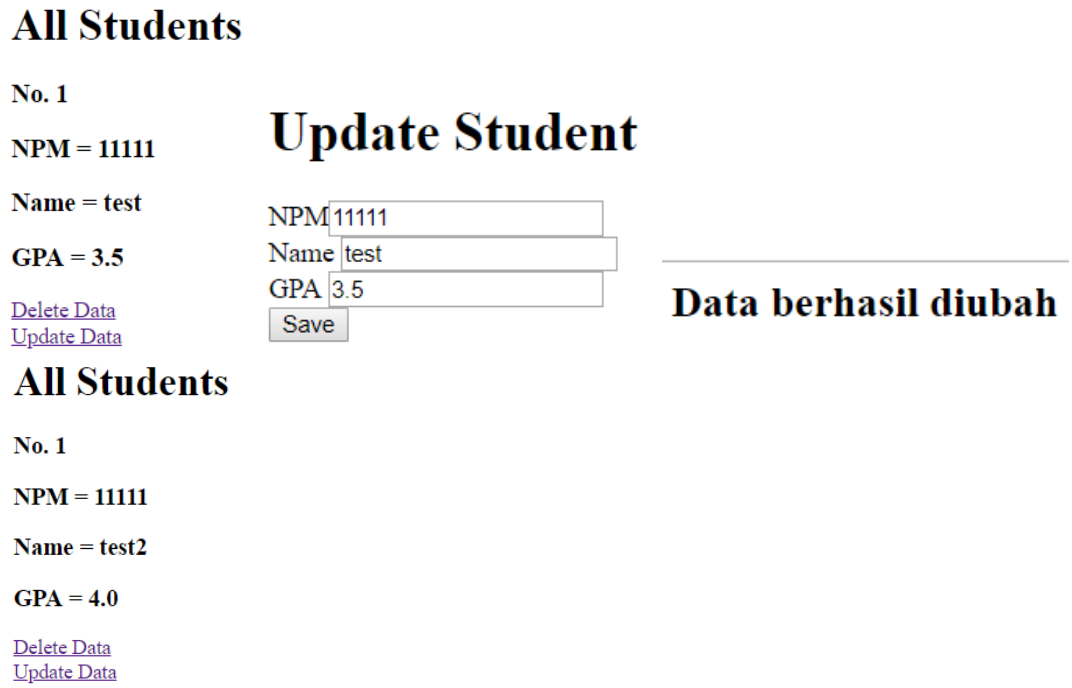
1. Method **updateSubmit** menerima parameter berupa model **StudentModel**

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

2. Tambahan atribut thymeleaf pada form-update view

```
<form action="/student/update/submit" method="post" th:object="${student}">

<label for="npm">NPM</label><input type="text" name="npm" readonly="true" th:value="${student.npm}" th:field="${student.npm}" /

<label for="name">Name</label> <input type="text" name="name" th:value="${student.name}" th:field="${student.name}" />

<label for="gpa">GPA</label> <input type="text" name="gpa" th:value="${student.gpa}" th:field="${student.gpa}" />
```

3. Contoh tampilan



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?
   Jawab:
   Melakukan validasi input yang required dan optional bisa dilakukan pada view yang menggunakan template view Thymeleaf. Misalnya dengan atribut **th:required** untuk setiap field yang diperlukan. Menggunakan object sebagai parameter pada form POST juga tidak mempersulit developer dalam memilah-milah field yang ingin digunakan, karena pada Thymeleaf banyak atribut seperti **th:field** yang memungkinkan kita untuk menggunakan atribut tertentu pada objek yang di-parse dari controller dan diterima dalam form view menggunakan **th:object** milik Thymeleaf. Validasi input tetap perlu dilakukan.

   Asumsikan input pada form Anda tidak menggunakan attribute required sehingga butuh validasi di backend.
   Jawab:
   Jika input pada form tidak menggunakan atribut required dan membutuhkan validasi di backend, maka bisa menggunakan anotasi @Valid atau @Notnull milik Hibernate validator pada atribut yang diinginkan.

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:
   Karena form submit biasanya berisi data-data yang bersifat private yang tidak sebaiknya tersimpan di cache ataupun url browser setiap environment. Ketika kita menggunakan method GET (seperti pada tutorial sebelumnya/method add pada tutorial ini), value dari setiap parameter akan terlihat pada url browser ketika method dijalankan, sehingga akan tersimpan sebagai cache pada browser. Hal ini tidak baik untuk dilakukan. Tidak ada penanganan yang berbeda pada body method di controller.

3. Apakah mungkin satu method menerima lebih dari satu jenis request method, misalkan menerima GET sekaligus POST?
   Jawab:
   Satu method mungkin untuk menerima lebih dari satu jenis request method. Misal GET dan POST sekaligus. Caranya adalah dengan menggunakan method = {RequestMethod.POST, RequestMethod.GET}. Namun perlu diperhatikan parameter yang digunakan pada method tersebut

Penjelasan

- Method pada latihan menambahkan delete

```
@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";
    }
}
```

Method ini digunakan untuk menghapus student dari database dengan parameter npm student tersebut. NPM yang menjadi parameter dicek ke database melalui method selectStudent pada studentService. Ketika student dengan npm tersebut benar ada pada database, maka student tersebut akan dihapus dang browser akan menampilkan view delete. Sebaliknya, jika student dengan npm tersebut tidak ditemukan, maka browser akan menampilkan view not-found

- Method pada latihan menambahkan update

```
@RequestMapping("/student/view/{npm}")
public String viewPath (Model model,
        @PathVariable(value = "npm") String npm)
{
    StudentModel student = studentDAO.selectStudent (npm);

    if (student != null) {
        model.addAttribute ("student", student);
        return "view";
    } else {
        model.addAttribute ("npm", npm);
        return "not-found";
    }
}
```

Method diatas digunakan untuk menampilkan form-update ketika student dengan npm yang diterima melalui parameter ada pada database (Hal ini dicek dengan menjalankan method selectStudent pada studentService yang mengembalikan objek student). Ketika student tidak ada, maka browser akan menampilkan view not-found.

```
@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 diatas digunakan ketika submit terhadap update dilakukan. Data student yang ada form-update akan disimpan ke database dengan menjalankan method updateStudent pada

studentService. Aplikasi akan menampilkan view success-update pada browser sebagai informasi bahwa update data student telah berhasil dilakukan.

- Method pada latihan menambahkan object sebagai parameter

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

Method ini menggunakan objek sebagai parameter agar ketika ingin menampilkan suatu objek pada view, tidak perlu panjang dalam menuliskan semua atribut yang ingin di-parse ke view dengan anotasi request parameter-nya. Method ini juga menggunakan POST agar semua atributnya tidak terlihat pada url browsernya. Method ini menerima objek student sebagai parameternya dan melakukan update data student tersebut ke database dengan menjalankan method updateStudent pada studentService. Controller akan mengembalikan view success-update yang memberikan informasi bahwa update data student telah berhasil dilakukan.

Ringkasan

- Logging menggunakan library **Slf4j** yang di-implementasikan pada **StudentServiceDatabase**
  Hal ini memudahkan dan membantu developer dalam melakukan debugging. Pesan yang ditulis melalui method dengan syntax log.info akan ditampilkan pada console saat method tersebut dijalankan. Contoh:

```
log.info ("select all students");
c.e.service.StudentServiceDatabase       : select all students
```

  Cara penggunaannya adalah dengan menambahkan anotasi **@Slf4j** di atas kelas yang ingin diberikan log. Variable dengan nama log akan bisa digunakan didalam lingkup kelas tersebut. Misal: **log.info**, **log.debug**, dan **log.error**.
- Pengaturan konfigurasi aplikasi yang diletakkan pada file **application.properties**. Konfigurasi yang bisa diatur diantaranya port, database configuration, API Key service, dll. Namun file application.properties unik untuk setiap environment. Perlu dicatat untuk tetap memastikan konfigurasi aplikasi kita sesuai dengan yang kita tulis dalam file application.properties.
- Penggunaan **Ibatis** sebagai data mapper framework yang memudahkan developer dalam menggunakan database relational dengan aplikasi berorientasi objek. Ibatis menyediakan fungsi-fungsi JDBC untuk bisa mengurangi penulisan code yang biasa dilakukan secara manual. Misalnya dengan menggunakan anotasi. Contoh:

```
import org.apache.ibatis.annotations.Select;
@Select("select npm, name, gpa from student where npm = #{npm}")
StudentModel selectStudent (@Param("npm") String npm);
```