

Andre Gema Syahputra
1506689282
ADPAP – B

Refleksi

Pada *tutorial* ini, saya mempelajari cara menggunakan *database* dengan aplikasi Spring, serta mengintegrasikannya dengan menggunakan komponen-komponen MVC, terutama Mapper.

Tutorial 4 ini juga memperkenalkan tentang cara untuk melakukan *debugging* pada Spring Boot yang lebih baik yaitu dengan menggunakan *logging*. *Logging* dapat dilakukan dengan memanfaatkan sebuah *library* bernama Slf4J yang ada pada *library* eksternal Lombok. Untuk menggunakan *library* tersebut, kita hanya perlu menambahkan anotasi @Slf4j pada suatu *class* yang ingin dilakukan *logging*. Anotasi tersebut memungkinkan untuk menggunakan variabel *log* yang memiliki *method* seperti log.info, log.debug, dan log.error.

Selain menggunakan *database*, saya juga mempelajari cara menggunakan *method* POST dalam *data passing*.

Delete

1. Mapper deleteStudent untuk fungsi Delete:

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

Gambar 1. Method deleteStudent pada StudentMapper

Method inilah yang bertugas dalam berinteraksi dengan *database* untuk menjalankan *query* DELETE dan melakukan *deletion* sesuai NPM pada *parameter*.

2. *Method* pada ServiceStudent dan ServiceStudentDatabase:

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

Gambar 2. Method deleteStudent pada StudentService

Andre Gema Syahputra
1506689282
ADPAP – B

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

Gambar 3. Implementasi *method* deleteStudent pada StudentServiceDatabase

Pada ServiceStudentDatabase, deleteStudent mengimplementasi *method* dari *interface* ServiceStudent dan memanggil deleteStudent pada StudentMapper yang akan menjalankan *query*. Logging juga diterapkan pada *method* ini.

3. *Method* ini menerima *parameter* sebuah Model dan sebuah string npm.

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

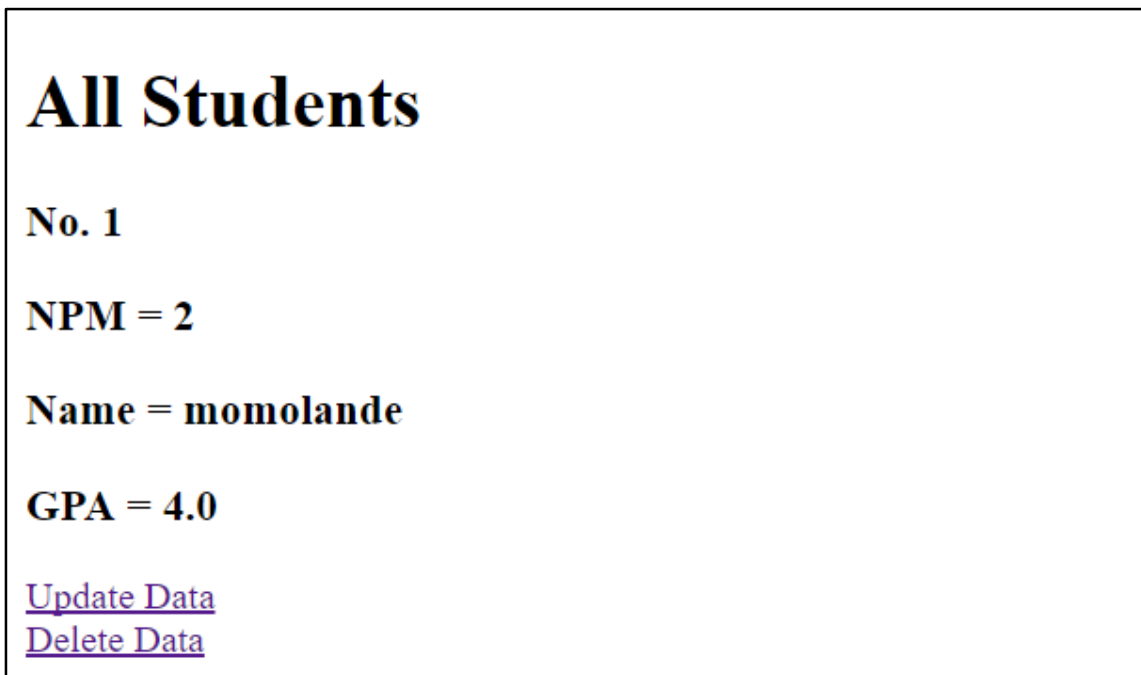
    if(student == null) {
        model.addAttribute("npm", npm);
        return "not-found";
    }
    studentDAO.deleteStudent (npm);

    return "delete";
}
```

Gambar 4. *Method* delete pada StudentController

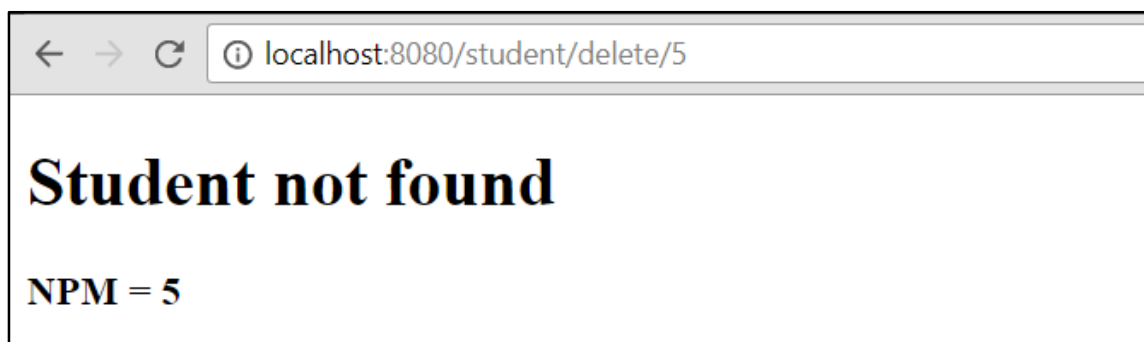
Cara kerja *method* ini adalah dengan pertama-tama mencari StudentModel dengan NPM yang berasal dari PathVariable dengan menggunakan *method* selectStudent pada *object* studentDAO dari *class* StudentServiceDatabase. Lalu, akan dilakukan pengecekan ada atau tidaknya *student* dengan NPM tersebut dengan cara mengecek apakah *object* student hasil pencarian merupakan null atau tidak.

Andre Gema Syahputra
1506689282
ADPAP – B



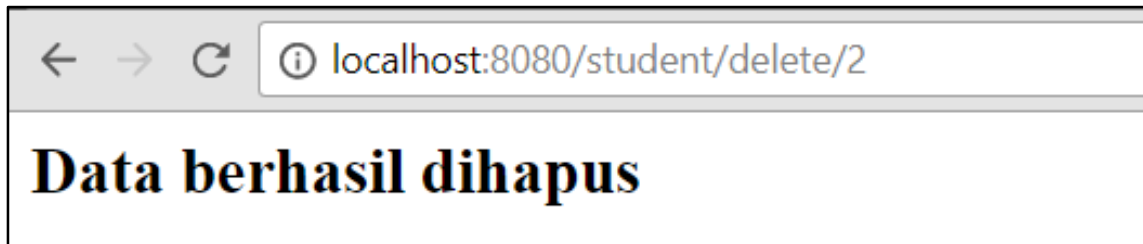
Gambar 5. Tampilan sebuah *student* dengan NPM 2

Apabila null, berarti tidak ditemukan *student* dengan NPM tersebut, sehingga akan ditampilkan View not-found.



Gambar 6. Student dengan NPM 5 tidak ditemukan ketika delete

Apabila bukan null, berarti *student* ditemukan, sehingga akan dipanggil *method* deleteStudent dengan *parameter* NPM, lalu akan ditampilkan View delete.



Gambar 7. Student dengan NPM 2 berhasil di-delete

Update

1. Mapper untuk Update:

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

Gambar 8. Method updateStudent pada StudentMapper

Method inilah yang bertugas dalam berinteraksi dengan *database* untuk menjalankan *query* UPDATE dan melakukan *update* sesuai dengan *object* StudentModel pada *parameter*.

2. Method Service pada StudentService dan StudentServiceDatabase:

```
void updateStudent (StudentModel student);
```

Gambar 9. Method updateStudent pada StudentService

```
@Override
public void updateStudent (StudentModel student)
{
    Log.info ("student " + student.getNpm() + " updated name to " + student.getName() +
        " and gpa to " + student.getGpa());
    studentMapper.updateStudent(student);
}
```

Gambar 10. Implementasi *method* updateStudent pada StudentService

Pada ServiceStudentDatabase, updateStudent mengimplementasi *method* dari *interface* ServiceStudent dan memanggil updateStudent pada StudentMapper yang akan menjalankan *query*. *Logging* juga diterapkan pada *method* ini.

3. *Method* pada StudentController:

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

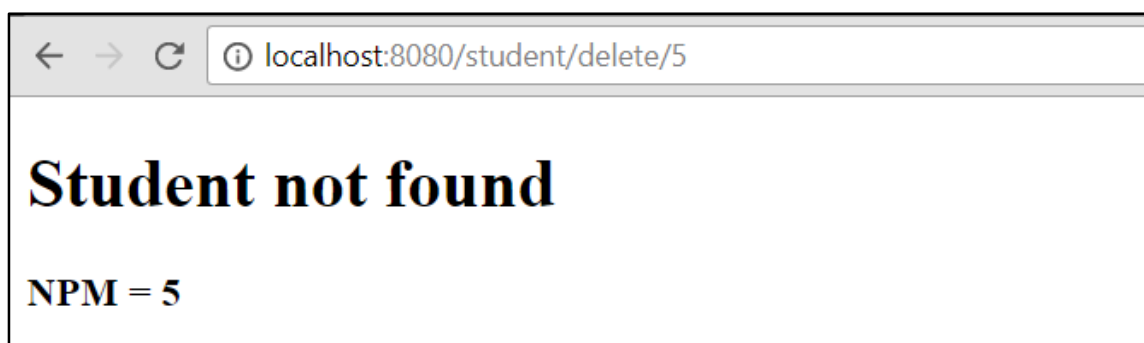
    if(student == null) {
        model.addAttribute("npm", npm);
        return "not-found";
    }

    model.addAttribute("student", student);
    return "form-update";
}
```

Gambar 11. *Method* update pada StudentController

Cara kerja *method* ini adalah dengan pertama-tama mencari StudentModel dengan NPM yang berasal dari PathVariable dengan menggunakan *method* selectStudent pada *object* studentDAO dari *class* StudentServiceDatabase. Lalu, akan dilakukan pengecekan ada atau tidaknya *student* dengan NPM tersebut dengan cara mengecek apakah *object* student hasil pencarian merupakan null atau tidak.

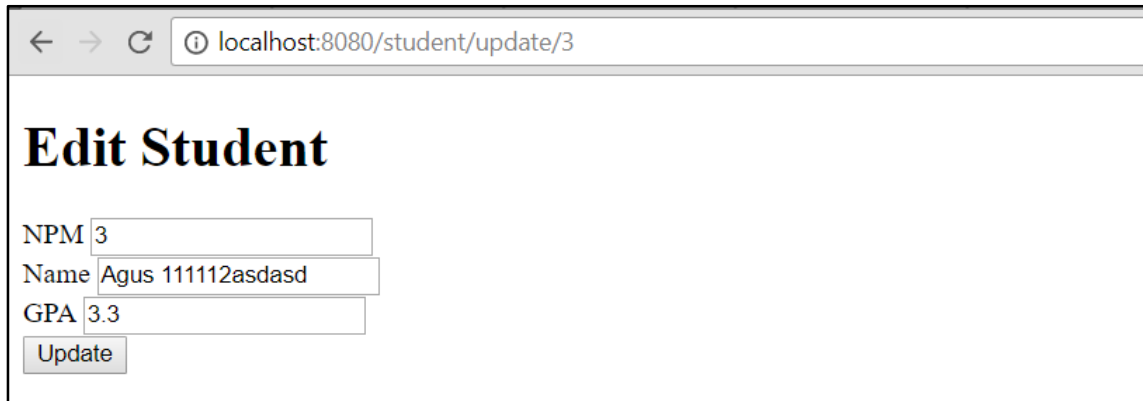
Apabila null, akan ditampilkan View not-found:



Gambar 12. Student dengan NPM 5 tidak ditemukan ketika update

Andre Gema Syahputra
1506689282
ADPAP – B

Apabila null, maka akan ditampilkan form-update dengan data-data *student* dengan NPM tersebut:



The screenshot shows a web browser window with the address bar displaying `localhost:8080/student/update/3`. The main content area has the heading **Edit Student**. Below the heading, there are three input fields: **NPM** with the value `3`, **Name** with the value `Agus 111112asdasd`, and **GPA** with the value `3.3`. At the bottom of the form is an **Update** button.

Gambar 13. Form-update *student* dengan NPM 3

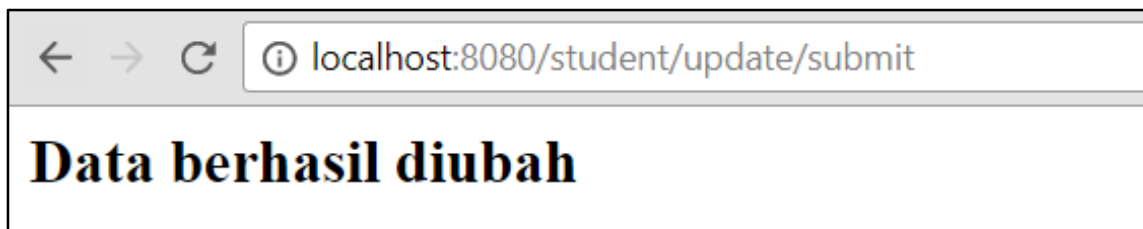
4. *Method* updateSubmit:

```
@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)
{
    studentDAO.updateStudent(new StudentModel(npm, name, gpa));
    return "success-update";
}
```

Gambar 14. *Method* updateSubmit pada StudentController

Method ini berjalan ketika *form* untuk melakukan *update* di-submit, data-data hasil POST dari *form* tersebut akan didapatkan. Lalu, *method* akan memanggil *method* updateStudent dengan *parameter* sebuah *object* StudentModel baru berdasarkan data yang di-submit.

Setelah itu, akan ditampilkan *view* success-update:



The screenshot shows a web browser window with the address bar displaying `localhost:8080/student/update/submit`. The main content area displays the message **Data berhasil diubah** in a large, bold font.

Gambar 15. *View* success-update

Object Sebagai *Parameter*

1. *Method* yang diubah:

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

Gambar 16. *Method* updateSubmit pada StudentController dengan *parameter object*

Method ini menerima *object* StudentModel hasil dari *submit form*.

Perubahan pada *method* ini adalah *parameter* yang hanya menerima StudentModel, dan pemanggilan updateStudent tanpa perlu membuat *object* StudentModel baru sebagai *parameter*. Setelah pemanggilan *method*, akan ditampilkan View success-update.

Pertanyaan

1. Jika menggunakan Object sebagai parameter pada form POST, bagaimana caranya melakukan validasi input yang optional dan input yang required? Apakah validasi di perlukan?

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

- Validasi dapat dilakukan dengan mengimplementasikan *interface* Validator pada sebuah *class* untuk melakukan validasi. Selain itu, dengan mengimplementasikan Validator, kita dapat menggunakan BindingResult untuk melakukan tes dan mendapatkan informasi mengenai *error* pada validasi.

Andre Gema Syahputra
1506689282
ADPAP – B

- Menambahkan *annotation* pada variabel di *class* yang mengandung constraints untuk melakukan validasi juga dapat dilakukan. *Annotation* seperti `@Size`, `@NotNull`, `@Min`, dan `@Max` dapat digunakan. Lalu, pada *parameter* ketika *object* dari *class* tersebut digunakan, tambahkan *annotation* `@Valid` untuk melakukan validasi
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?
- Karena GET seharusnya tidak digunakan untuk mengubah kondisi *server* dan database. Hal ini dikarenakan masalah *security* yang dapat terjadi apabila menggunakan GET, seperti terjadi *delete* tidak sengaja, gangguan *search engine bot*, dan banyak lainnya. Selain itu, data GET ditampilkan pada URL, sehingga user dapat melihat data yang mungkin tidak seharusnya dilihat. Sehingga, GET lebih baik digunakan untuk *viewing*, *search*, *select*, dan semacamnya.
 - Penanganan pada *controller* hanyalah dengan mengubah *method* menjadi *RequestMethod.POST*
3. Apakah mungkin satu method menerima lebih dari satu jenis request method, misalkan menerima GET sekaligus POST?
- Mungkin, dengan menggunakan
`method = { RequestMethod.GET, RequestMethod.POST }`
- ```
@RequestMapping(value = "/student/update/submit", method = { RequestMethod.GET, RequestMethod.POST })
```
- Source
- <https://spring.io/guides/gs/validating-form-input/>



Andre Gema Syahputra  
1506689282  
ADPAP – B

- <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/validation/BindingResult.html>
- <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#validation>
- <http://stephenwalther.com/archive/2009/01/21/asp-net-mvc-tip-46-ndash-donrsquot-use-delete-links-because>
- <https://stackoverflow.com/questions/17987380/combine-get-and-post-request-methods-in-spring>