FPT SOFTWARE QUY NHON

SPRING BOOT DATA JPA

KienNCT3 - Mentor HaiNV21

Content:

1) Spring Data JPA.

2) Testing Spring Data JPA with @DataJpaTest.

3) Spring Boot Data JPA with MySQL.

4) Spring Boot Data JPA with PostgreSQL.

5) Spring Data JPA Paging and Sorting.

6) JPA Specification join multiple tables.

7) Rest Query Language with Spring Data JPA Specification.

WHAT IS SPRING DATA JPA?

- Spring Data JPA is a part of Spring Framework, a project designed to reduce coding repetition when working with databases in Spring applications.
- **JPA** stands for **"Java Persistence API"**. It is a technology that allows to store and retrieve data from a database in a Java application more easily. Spring Data JPA uses **JPA** to interact with the database.
- Assume that we need to build a web application and store user information in database. Instead of writing repetitive code to perform operations like adding, editing, deleting, and retrieve data from the database, we can use Spring Data JPA to do this easily and more quickly through abstract method in interface that called repository like **findById(long id)**, **save(User user)...**

WHAT IS SPRING DATA JPA?

Here are some key features of Spring Data JPA:

- **Abstraction Layer**: Spring Data JPA provides an abstraction layer on JPA. Instead of having to write repetitive code to perform basic operations like CRUD, we can use the provided default methods like **findAll(), save(), update()...**

- **Query Methods:** Spring Data JPA allows we to define data query methods using naming rules.

- **Custom Queries:** If the default query methods do not satisfy needs, we can still write custom queries using annotations like **@Query**.

- **Paging and Sorting:** Spring Data JPA supports paging and sorting query results easily.

WHAT IS SPRING DATA JPA?

Here are some key features of Spring Data JPA:

- **Transaction Management:** Spring Data JPA integrates well with Spring's transaction management, automatically managing transactions when performing database operations.

For example of Spring Data JPA:

```java
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    4 usages
    private String name;
    4 usages
    private int age;
    4 usages
    private String address;
    4 usages
    private String phone;
```

Suppose we have a Java object "Author" and we want to save its information in the database. First, we will create an interface that inherits from **JpaRepository** interface:

```java
public interface AuthorRepository extends JpaRepository<Author, Long> {}
```

In this interface, "**Author**" is the data type of the object we want to store, and "**Long**" is the data type of the object's primary key. **JpaRepository** provides standard methods to perform basic operations such as **findAll(), findById(), save(), update(), and delete().**

For example of Spring Data JPA:

```java
@Service
public class AuthorService {
    5 usages
    private final AuthorRepository rep;
    new *
    @Autowired
    public AuthorService(AuthorRepository rep) {
        this.rep = rep;
    }
    no usages  new *
    public List<Author> findAll() {
        return rep.findAll();
    }
    no usages  new *
    public Author save(Author a) {
        return rep.save(a);
    }
    no usages  new *
    public void deleteById(Long id) {
        rep.deleteById(id);
    }
```

- Next, we can use **AuthorRepository** in other components or services to perform database operations.

- **AuthorService** uses **AuthorRepository** to save and query user information in the database easily and quickly.

BASIC FLOW OF SPRING DATA JPA

The basic flow of Spring Data JPA has the following steps:

1) Entity definition:

We often start by defining entities to represent the data in database. Entities usually correspond to tables in the database. Each field in an entity usually corresponds to a column in the table.

2) Create Repository Interface:

Next, we create interfaces called repositories to interact with database. Each repository usually corresponds to an entity in database like **BookRepository**, **ProductRepository**.

3) Query Method Definition:

In repositories, we define methods to perform database operations such as CRUD. Spring Data JPA uses naming conventions to automatically generate queries based on the names of methods like **save()**, **findById(long id), deleteById(long id), etc.**

BASIC FLOW OF SPRING DATA JPA

The basic flow of Spring Data JPA has the following steps:

4) <u>Automatic Repository Deployment:</u>
Spring Data JPA will automatically create implementations for methods in the repository based on naming conventions and defined annotations. This eliminates the need to write SQL queries.

5) <u>Using Repository in Application:</u>
Finally, we can use the repositories defined in application to perform database operations

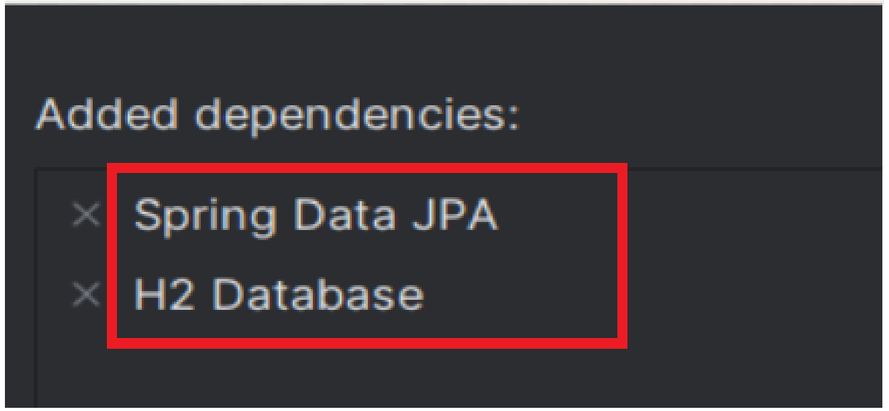
Testing Spring Data Jpa With @DataJpaTest

- **@DataJpaTest** is an annotation used in unit testing to test components related to JPA.

1) First, create new project with these dependencies:

```
<dependencies> Edit Starters...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
```



Testing Spring Data Jpa With @DataJpaTest

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

H2 is an embedded database and operates directly in the application's memory, requiring no specific configuration or an external database server like MySQL, PostgreSQL.

Testing Spring Data Jpa With @DataJpaTest

2) Create JPA Entity:

```
@Entity
@AllArgsConstructor
@NoArgsConstructor
@Data
@Builder
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;
    private String gmail;
}
```

Testing Spring Data Jpa With @DataJpaTest

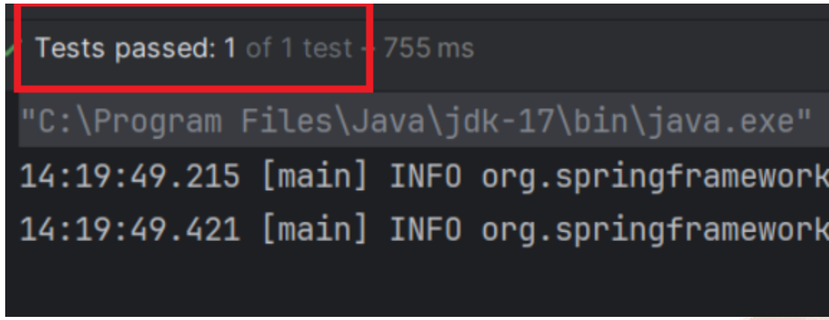
3) Create Spring Data JPA Repository:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {}
```

4) Create Testing class with @DataJpaTest:

```
@DataJpaTest
public class EmployeeRepositoryTests {
    @Autowired
    private EmployeeRepository rep;
    new *
    @Test
    public void testEmployeeSave() {
        Employee employee = Employee.builder().name("Trung Kien")
                .age(18).gmail("kien@gmail.com").build();
        rep.save(employee);
        assertThat(employee.getId()).isEqualTo( expected: 1L);
        Assertions.assertEquals( expected: "Trung Kien", employee.getName());
        Assertions.assertEquals( expected: 18, employee.getAge());
        Assertions.assertEquals( expected: "kien@gmail.com", employee.getGmail());
    }
}
```

Result:



```java
@Test
@Order(1)
@Rollback(value = false)
public void testEmployeeSave() {
    Employee employee = Employee.builder().name("Trung Kien")
            .age(18).gmail("kien@gmail.com").build();
    rep.save(employee);
    assertThat(employee.getId()).isEqualTo( expected: 1L);
    Assertions.assertEquals( expected: "Trung Kien", employee.getName());
    Assertions.assertEquals( expected: 18, employee.getAge());
    Assertions.assertEquals( expected: "kien@gmail.com", employee.getGmail());
}
```

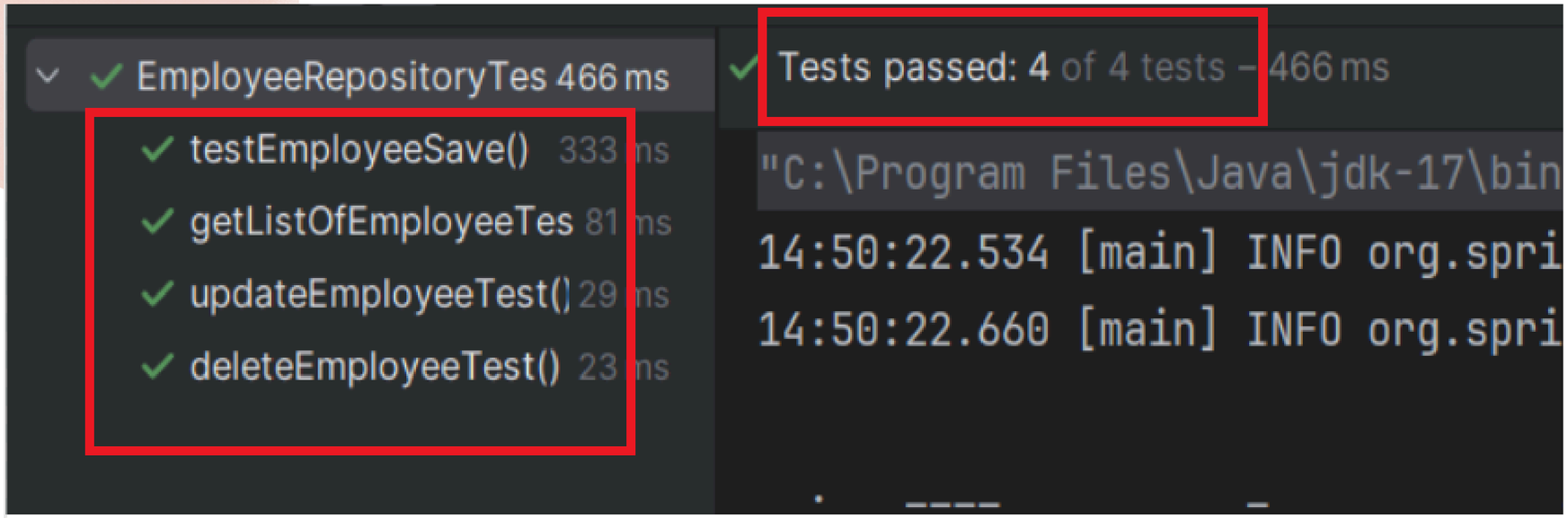
```java
@Test
@Order(2)
@Rollback(value = false)
public void getListOfEmployeeTest() {
    List<Employee> eList = rep.findAll();
    assertThat(eList.size()).isGreaterThan( other: 0);
}
```

```java
@Test
@Order(3)
@Rollback(value = false)
public void updateEmployeeTest() {
    Employee employee = rep.findById(1L).get();
    employee.setGmail("fpt@gmail.com");
    assertThat(rep.save(employee).getGmail()).isEqualTo( expected: "fpt@gmail.com");
}
```

```java
@Test
@Order(4)
@Rollback(value = false)
public void deleteEmployeeTest() {
    Employee employee = rep.findById(1L).get();
    rep.delete(employee);
    Employee employee1 = null;
    Optional<Employee> optionalEmployee = rep.findByGmail("fpt@gmail.com");
    if(optionalEmployee.isPresent()) {
        employee1 = optionalEmployee.get();
    }
    assertThat(employee1).isNull();
}
```

Testing Spring Data Jpa With @DataJpaTest

Result:



```java
@Test
@Order(1)
@Rollback(value = false)
public void testEmployeeSave() {
    Employee employee = Employee.builder().name("Trung Kien")
            .age(18).gmail("kien@gmail.com").build();
    rep.save(employee);
    assertThat(employee.getId()).isEqualTo( expected: 1L);
    Assertions.assertEquals( expected: "Trung Kien", employee.getName());
    Assertions.assertEquals( expected: 18, employee.getAge());
    Assertions.assertEquals( expected: "kien@gmail.com", employee.getGmail());
}
```

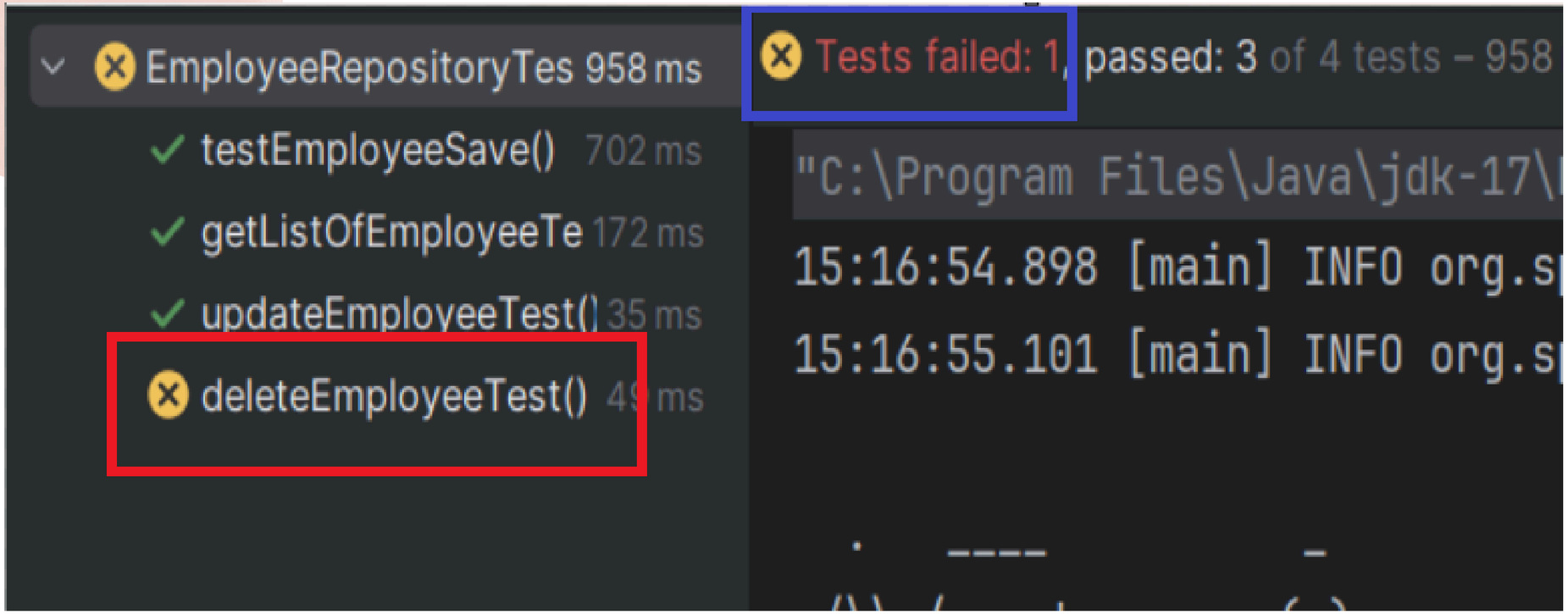
```java
@Test
@Order(2)
@Rollback(value = false)
public void getListOfEmployeeTest() {
    List<Employee> eList = rep.findAll();
    assertThat(eList.size()).isGreaterThan( other: 0);
}
```

```java
@Test
@Order(3)
@Rollback(value = false)
public void updateEmployeeTest() {
    Employee employee = rep.findById(1L).get();
    employee.setGmail("fpt@gmail.com");
    assertThat(rep.save(employee).getGmail()).isEqualTo( expected: "fpt@gmail.com");
}
```

```java
@Test
@Order(4)
@Rollback(value = false)
public void deleteEmployeeTest() {
    Employee employee = rep.findById(1L).get();
    rep.delete(employee);
    Employee employee1 = null;
    Optional<Employee> optionalEmployee = rep.findByGmail("fpt@gmail.com");
    if(optionalEmployee.isPresent()) {
        employee1 = optionalEmployee.get();
    }
    assertThat(employee1).isNotNull();
```

Testing Spring Data Jpa With @DataJpaTest

Result:

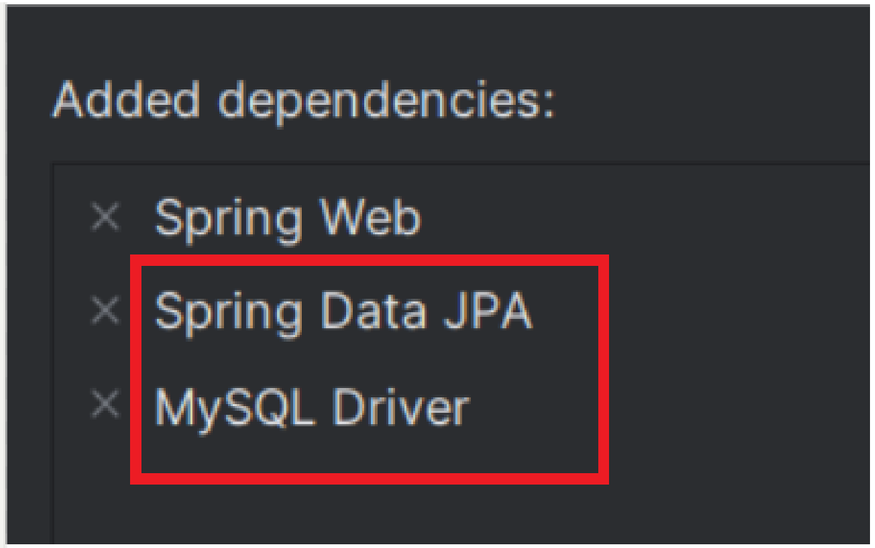


Spring Boot Data JPA with MySQL

1) Connect to MySQL:

- Add the necessary dependencies to **pom.xml(Maven project)** or **build.gradle(Gradle project)** file

```
<!-- Spring Boot Starter Data JPA-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- Spring Boot Starter Web-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- MySql Connector-->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```



Spring Boot Data JPA with MySQL

1) Connect to MySQL

- Configure connection information to **MySQL**: We need to provide configuration information such as the connection **URL**, **username**, and **password** of the MySQL database. This information is usually placed in the **application.properties** or **application.yml** file

```
spring.datasource.url=jdbc:mysql://localhost:3306/bookmanagement
spring.datasource.username=root
spring.datasource.password=trungkien


spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
```

Spring Boot Data JPA with MySQL

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

- **spring.jpa.properties.hibernate.dialect**: This is how Spring Boot provides configuration for Hibernate properties. In this case, it is setting up Hibernate Dialect.

+ **Hibernate** is one of the most popular frameworks used to interact with databases in Java applications.

+ **Hibernate Dialect** is one of the most important parameters to configure, as it determines how Hibernate interacts with the database. **Dialect** provides an interface for Hibernate to generate SQL statements appropriate to the database we are using.

- **org.hibernate.dialect.MySQL8Dialect**: This is the specific Hibernate Dialect used, which is MySQL8Dialect. In this case, MySQL8Dialect was chosen for compatibility with MySQL database version 8.

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
```

- **spring.jpa.hibernate.ddl-auto=update**: This configuration determines how Hibernate uses to create or update the structure of the database. In this case, the "**update**" value indicates that Hibernate will automatically update the database structure to accommodate changes in entity classes without losing existing data.
- Here, **spring.jpa.hibernate.ddl-auto** can be **none**, **update**, **create**, or **create-drop:**

+ **none**: The default for MySQL. No change is made to the database structure.
+ **update**: Hibernate changes the database according to the given entity structures.
+ **create**: Creates the database every time but does not drop it on close.
+ **create-drop**: Creates the database and drops it when application closes.

Spring Boot Data JPA with MySQL

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
```

- **spring.jpa.show-sql=true:** This configuration indicates whether Spring Boot should show SQL queries generated by Hibernate or not. When set to true, SQL queries will be displayed in the application log, making it easy to observe and debug queries.

- **spring.jpa.generate-ddl=true:** This configuration indicates whether Spring Boot should generate DDL (Data Definition Language) statements to generate the structure of the database. When set to true, Hibernate will generate DDL statements based on entity classes to create or update the structure of the database.

Spring Boot Data JPA with MySQL

2) Create the @Entity Model:

```
@Entity // This annotation specifies that class is an entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    no usages
    private String name;
    no usages
    private String gmail;
```

Hibernate automatically translates the entity into a table.

2) Create the @Entity Model:

```java
@Entity // This annotation specifies that class is an entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    no usages
    private String name;
    no usages
    private String gmail;
```

Before:

Result in console:

```
Hibernate: create table user (id bigint not null, gmail varchar(255), name varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table user_seq (next_val bigint) engine=InnoDB
Hibernate: insert into user_seq values ( 1 )
```

After:

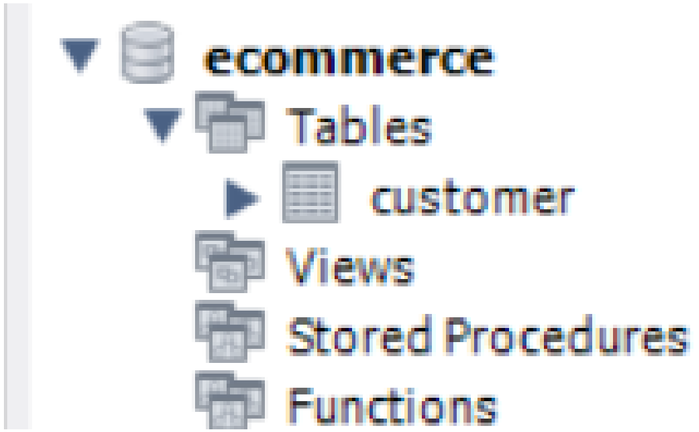
user_seq table:

	next_val
▶	1

JPA Annotation @Table, @Column

1) @Table: used to map an entity class to a table in the database.

```java
@Entity
@Table(name = "tb_customers", schema = "ecommerce")
public class Customer {
    @Id
    private int id;
    2 usages
    private String name;
    2 usages
    private int age;
    //constructor, getter, setter, toString method
```



The result in console:

```
Hibernate: create table tb_customers (id integer not null, age integer not null, name varchar(255), primary key (id)) engine=InnoDB
```

JPA Annotation @Table, @Column

1) @Table: used to map an entity class to a table in the database.

```java
@Entity
@Table(name = "tb_customers", schema = "ecommerce")
public class Customer {
    @Id
    private int id;
    2 usages
    private String name;
    2 usages
    private int age;
    //constructor, getter, setter, toString method
```

The result in console:

```
Hibernate: create table tb_customers (id integer not null, age integer not null, name varchar(255), primary key (id)) engine=InnoDB
```

JPA Annotation @Table, @Column

2) @Column: used to map an attribute of an entity class to a column in a table in the database.

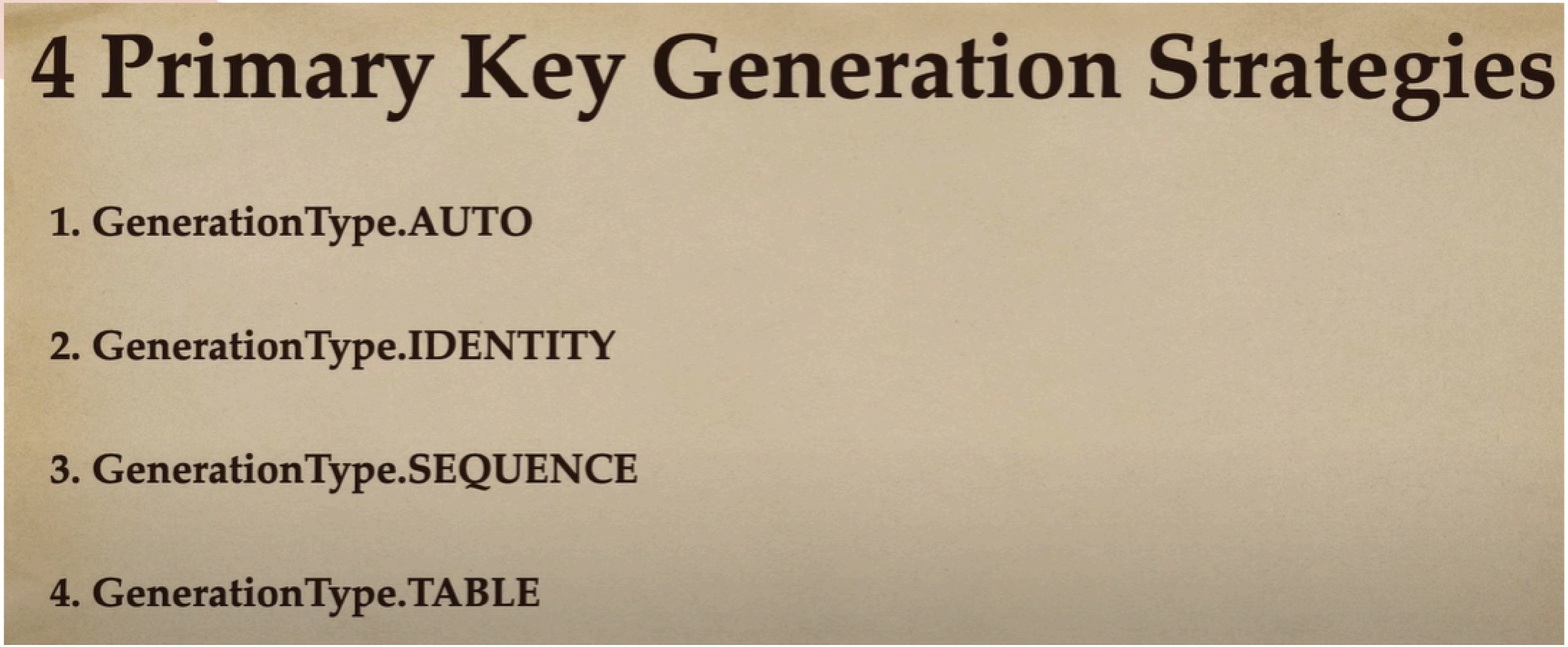
```java
@Entity
@Table(name = "tb_customers", schema = "ecommerce")
public class Customer {
    @Id
    private int id;
    2 usages
    private String name;
    2 usages
    private int age;
    no usages
    @Column(name = "cus_address", nullable = false)
    private String address;
    //constructor, getter, setter, toString method
```

Before:

After:

JPA Annotation @Generated Value

4) @GeneratedValue: used to specify how to generate values for the primary key of the entity class.



1) GenerationTYPE.INDENTITY

- In this strategy, the primary key is automatically generated by the database.
- MySQL uses **auto-increment** to generate primary key values.

```java
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
```

2) GenerationTYPE.SEQUENCE

- In this strategy, the primary key is generated by a database sequence generator.
- The database will generate a unique string of numbers to generate the primary key value.
- Oracle uses sequences to generate primary key values.

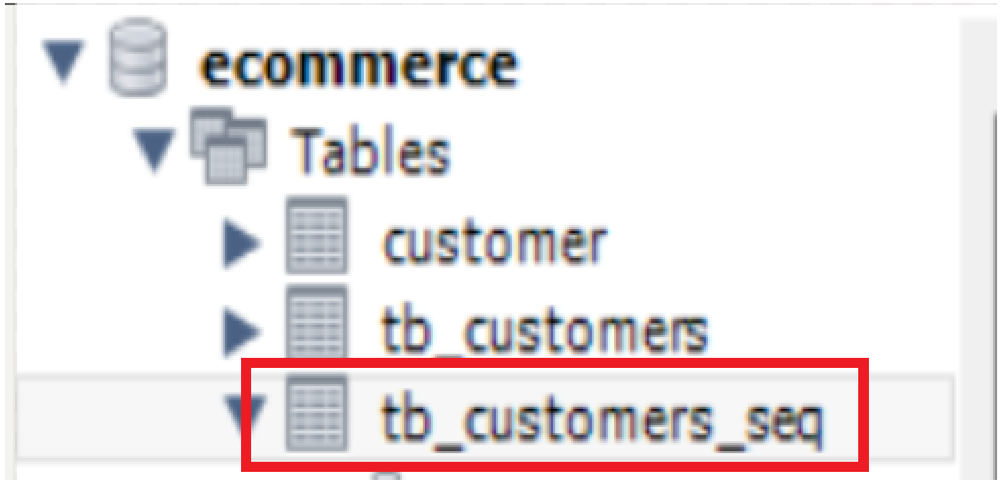
```
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private int id;
```

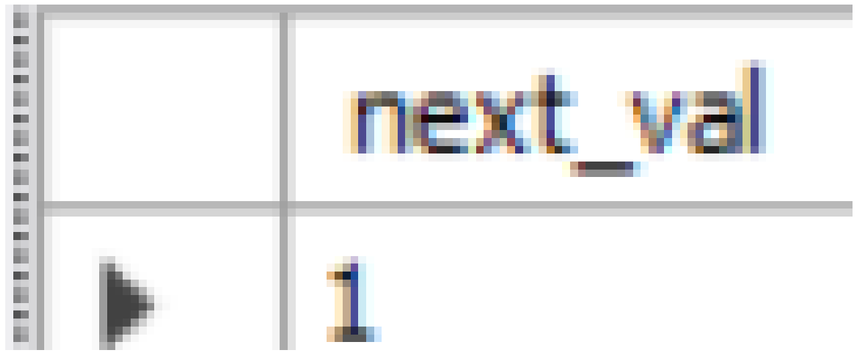
```
Hibernate: create table tb_customers_seq (next_val bigint) engine=InnoDB
Hibernate: insert into tb_customers_seq values ( 1 )
```

2) GenerationTYPE.SEQUENCE

```java
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private int id;
```

```
Hibernate: create table tb_customers_seq (next_val bigint) engine=InnoDB
Hibernate: insert into tb_customers_seq values ( 1 )
```





3) GenerationTYPE.TABLE

- In this strategy, Hibernate uses a special table in the database to generate primary key values.
- Every time a new key value needs to be generated, Hibernate will add a new row to this table and use the generated value.

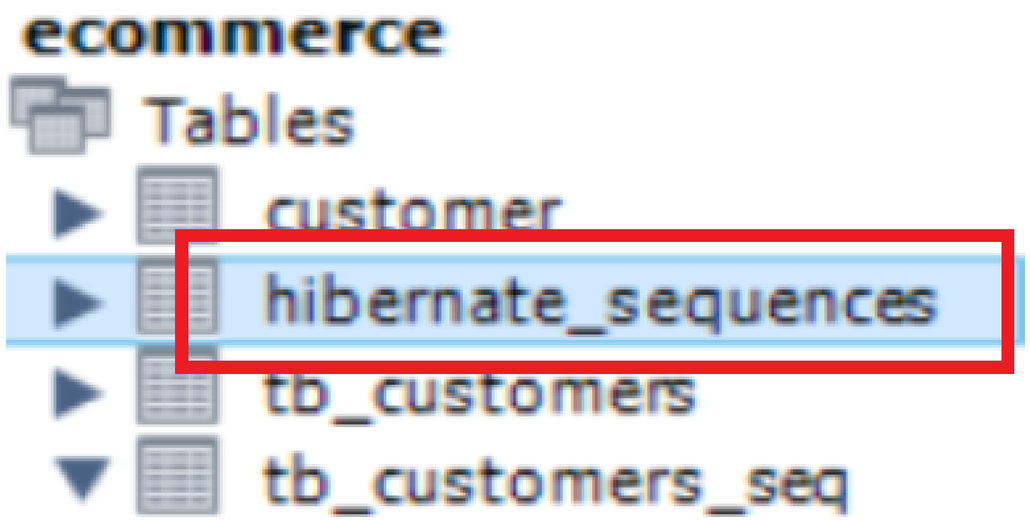
```
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
```

```
Hibernate: create table hibernate_sequences (sequence_name varchar(255) not null, next_val bigint, primary key (sequence_name)) engine=InnoDB
Hibernate: insert into hibernate_sequences(sequence_name, next_val) values ('default',0)
```

3) GenerationTYPE.TABLE

```java
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int id;
```

```
Hibernate: create table hibernate_sequences (sequence_name varchar(255) not null, next_val bigint, primary key (sequence_name)) engine=InnoDB
Hibernate: insert into hibernate_sequences(sequence_name, next_val) values ('default',0)
```



sequence_name	next_val
default	0
NULL	NULL

4) GenerationTYPE.AUTO

- In this strategy, Hibernate will automatically choose the primary key value generation strategy suitable for the database.
- Hibernate will prioritize using IDENTITY, then SEQUENCE, and finally TABLE.
- For example: If using MySQL, Hibernate will automatically use IDENTITY; If using Oracle, Hibernate will use SEQUENCE.

```
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
```

Some importants Repository Interfaces

In Spring Data JPA, Repository Interfaces play an important role in creating mechanisms to perform data queries and database operations. Below are some important interfaces:

1) CrudRepository:

Interface **CrudRepository** provides basic methods to perform CRUD (Create, Read, Update, Delete) operations with an entity.
Methods like save(), findById(), findAll(), delete() are typical examples.

2) PagingAndSortingRepository:

The **PagingAndSortingRepository** interface extends **CrudRepository** and provides support for paging and sorting results.
It adds methods like **findAll(Pageable pageable)** and **findAll(Sort sort)** to support paging and sorting.

Some importants Repository Interfaces

In Spring Data JPA, Repository Interfaces play an important role in creating mechanisms to perform data queries and database operations. Below are some important interfaces:

3) JpaRepository:

The **JpaRepository** interface extends **PagingAndSortingRepository** and provides some more flexible and convenient methods for manipulating data.

It provides methods like **findBy, findAll...()** to perform more complex queries.

4) JpaSpecificationExecutor:

This interface provides methods to perform queries based on criteria **(Specification)**.

3) Create the Repository:

```
4 usages   new *
public interface BookRepository extends JpaRepository<Book, Long>, JpaSpecificationExecutor<Book> {}
```

Book is the entity type that this repository will manipulate, and **Long** is the data type of the primary key of the Book entity. It also is the entity type to which this "**JpaSpecificationExecutor**" specification will be applied.

4) Create the Controller

```java
@RestController
@RequestMapping("/books")
public class BookController {
    3 usages
    private final BookRepository rep;

    new *
    @Autowired
    public BookController(BookRepository rep) { this.rep = rep; }
    new *
    @GetMapping("/getAll")
    public List<Book> getAll() {
        return rep.findAll();
    }
```

localhost:8080/books/getAll


```
[
    {
        "id": 1,
        "name": "Math",
        "authorID": {
            "id": 1,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 1,
            "name": "Comic"
        }
    },
    {
        "id": 2,
        "name": "Doraemon",
        "authorID": {
            "id": 1,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 2,
            "name": "Text Book"
        }
    },
```

CRUD In MySQL:

```java
@PostConstruct
public void generateCus() {
    try {
        Random random = new Random();
        List<Customer> listC = new ArrayList<>();
        for(int i = 0; i < 5; i++) {
            Customer cus = new Customer();
            cus.setName("Customer " + random.nextInt( bound: 100) + 1) ;
            cus.setEmail("customer" + random.nextInt( bound: 100) + 1 + "@gmail.com");
            cus.setPhone("09646412" + random.nextInt( bound: 900) + 1);
            cus.setAddress("Gia Lai, 23/" + random.nextInt( bound: 100) + 1 + " Hung Vuong");
            listC.add(cus);
        }
        rep.saveAll(listC);
    } catch(Exception e) {
        System.out.println("Fail to genetate: " + e.getMessage());
    }
```

Create Data

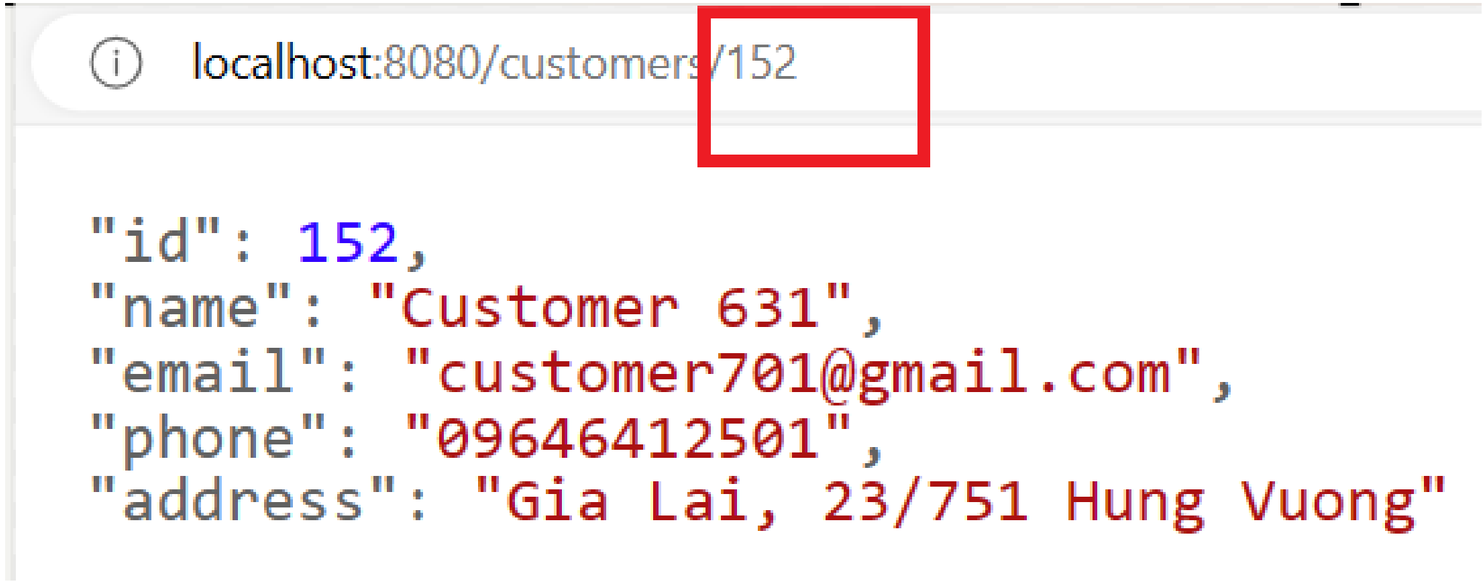
localhost:8080/customers


```
[
    {
        "id": 152,
        "name": "Customer 631",
        "email": "customer701@gmail.com",
        "phone": "09646412501",
        "address": "Gia Lai, 23/751 Hung Vuong"
    },
    {
        "id": 153,
        "name": "Customer 161",
        "email": "customer811@gmail.com",
        "phone": "096464128791",
        "address": "Gia Lai, 23/571 Hung Vuong"
    },
    {
        "id": 154,
        "name": "Customer 131",
        "email": "customer661@gmail.com",
        "phone": "096464122491",
        "address": "Gia Lai, 23/71 Hung Vuong"
    },
    {
        "id": 155,
        "name": "Customer 891",
        "email": "customer571@gmail.com",
        "phone": "096464128001",
        "address": "Gia Lai, 23/941 Hung Vuong"
    },
    {
        "id": 156,
        "name": "Customer 681",
        "email": "customer281@gmail.com",
        "phone": "09646412301",
        "address": "Gia Lai, 23/21 Hung Vuong"
    }
]
```


Get Data

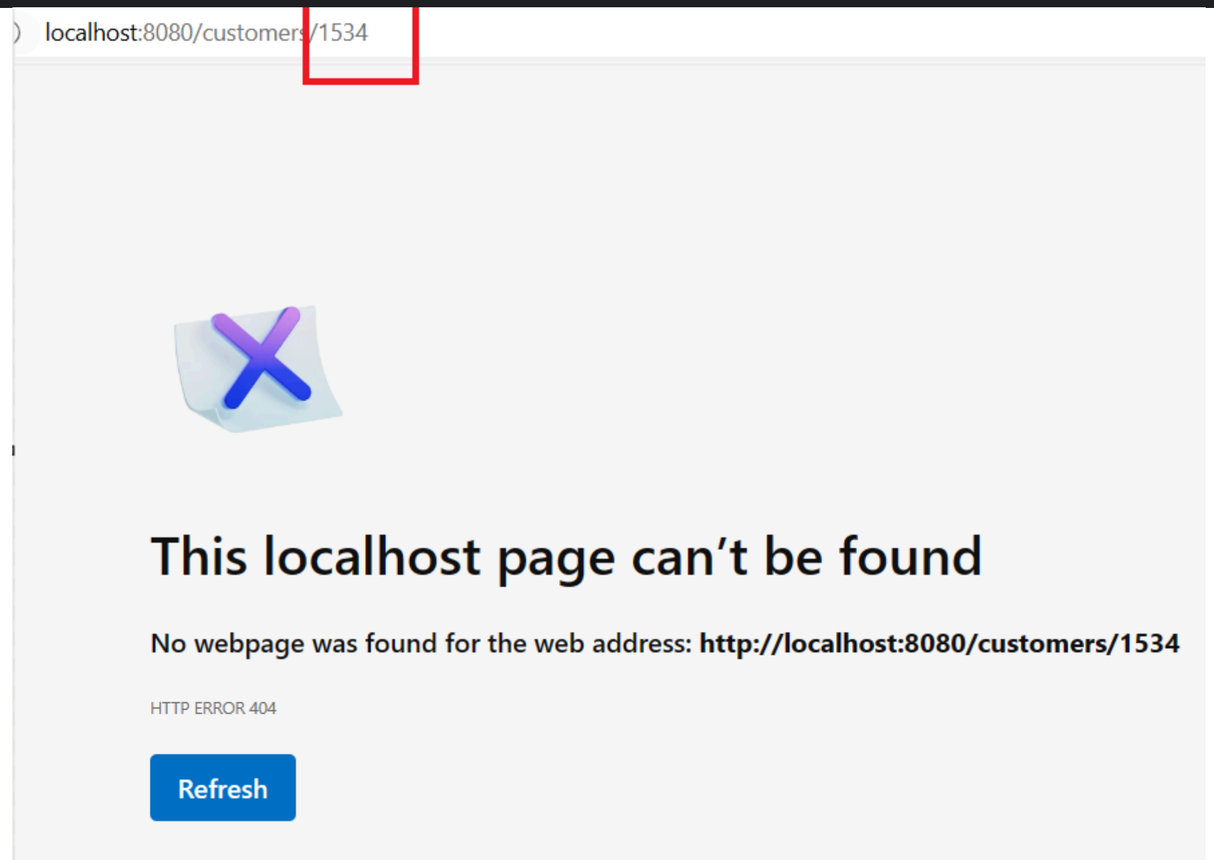
CRUD IN MySQL:

```java
@GetMapping("/{id}")
public ResponseEntity<Customer> getById(@PathVariable("id") Long id) {
    Optional<Customer> employeeOptional = rep.findById(id);
    return employeeOptional.map(ResponseEntity::ok)
            .orElseGet(() -> ResponseEntity.notFound().build());
}
```



CRUD IN MySQL:

```java
@GetMapping("/{id}")
public ResponseEntity<Customer> getById(@PathVariable("id") Long id) {
    Optional<Customer> employeeOptional = rep.findById(id);
    return employeeOptional.map(ResponseEntity::ok)
            .orElseGet(() -> ResponseEntity.notFound().build());
}
```

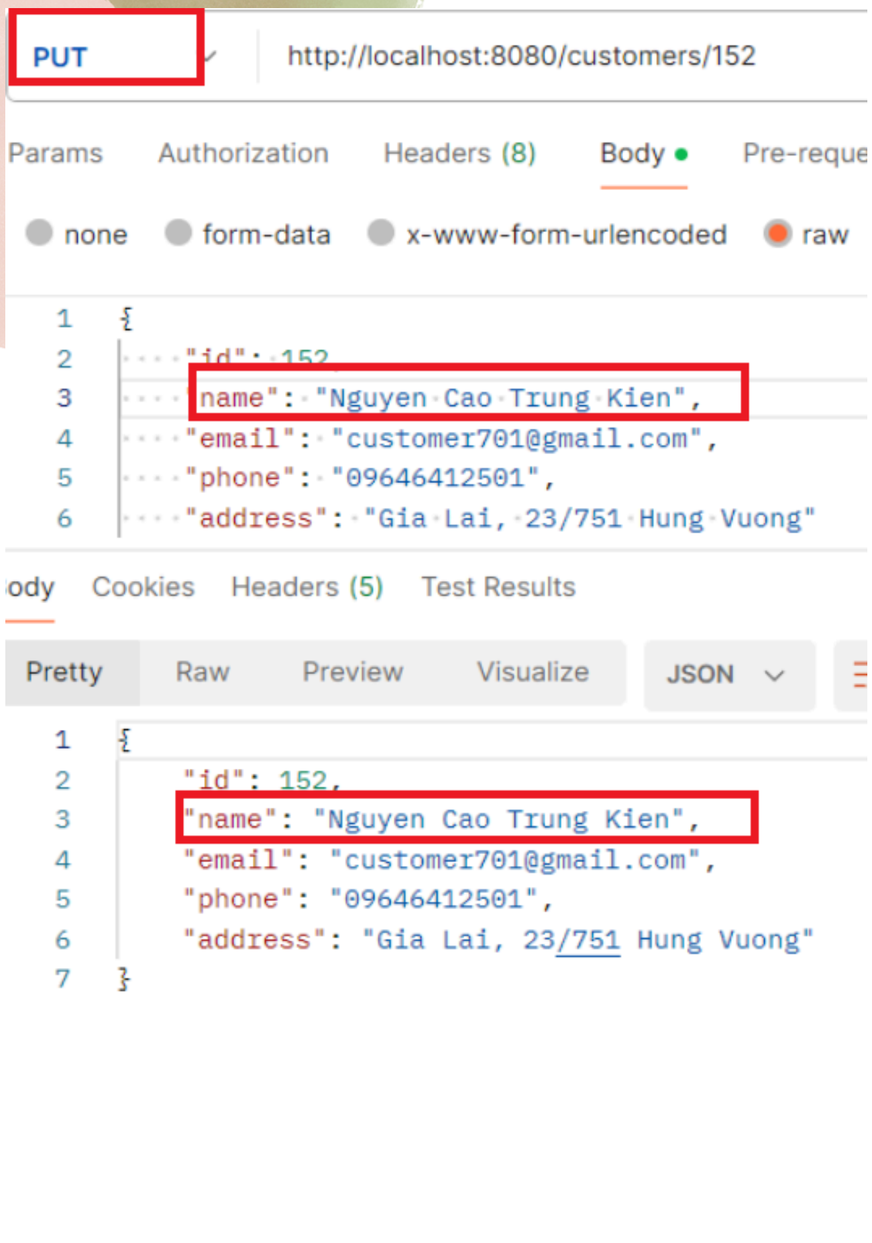


CRUD IN MySQL:

```
@PutMapping("/{id}")
public ResponseEntity<Customer> updateEmployee(@PathVariable("id") Long id, @RequestBody Customer cus) {
    if (!rep.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    cus.setId(id);
    Customer updatedCus= rep.save(cus);
    return ResponseEntity.ok(updatedCus);
}
```

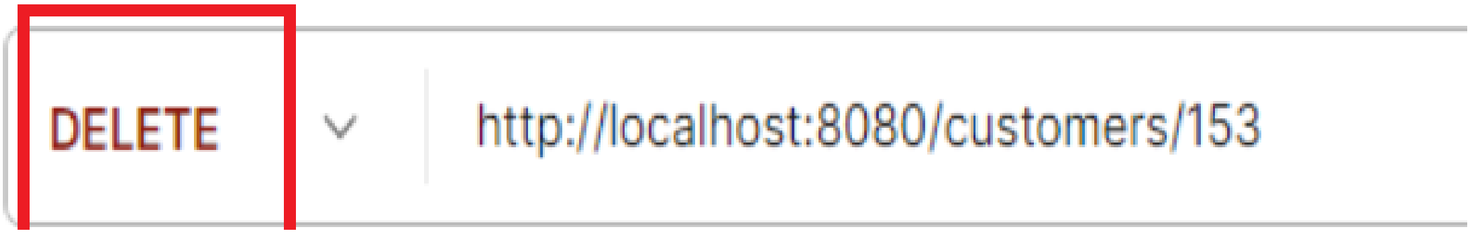
Update Data

CRUD IN MySQL:



CRUD IN MySQL:

```java
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteCus(@PathVariable("id") Long id) {
    if (!rep.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    rep.deleteById(id);
    return ResponseEntity.noContent().build();
}
```



CRUD IN MySQL:

DELETE http://localhost:8080/customers/153

localhost:8080/customers/153

This localhost page can't be found

No webpage was found for the web address: **http://localhost:8080/customers/153**

HTTP ERROR 404

Refresh

localhost:8080/customers


```
{
    "id": 152,
    "name": "Nguyen Cao Trung Kien",
    "email": "customer701@gmail.com",
    "phone": "09646412501",
    "address": "Gia Lai, 23/751 Hung Vuong"
},
{
    "id": 154,
    "name": "Customer 131",
    "email": "customer661@gmail.com",
    "phone": "096464122491",
    "address": "Gia Lai, 23/71 Hung Vuong"
},
{
    "id": 155,
    "name": "Customer 891",
    "email": "customer571@gmail.com",
    "phone": "096464128001",
    "address": "Gia Lai, 23/941 Hung Vuong"
},
{
    "id": 156,
    "name": "Customer 681",
    "email": "customer281@gmail.com",
    "phone": "09646412301",
    "address": "Gia Lai, 23/21 Hung Vuong"
}
```

After all. here is the data in MySQL:

localhost:8080/customers


```
[
    {
        "id": 152,
        "name": "Nguyen Cao Trung Kien",
        "email": "customer701@gmail.com",
        "phone": "09646412501",
        "address": "Gia Lai, 23/751 Hung Vuong"
    },
    {
        "id": 154,
        "name": "Customer 131",
        "email": "customer661@gmail.com",
        "phone": "096464122491",
        "address": "Gia Lai, 23/71 Hung Vuong"
    },
    {
        "id": 155,
        "name": "Customer 891",
        "email": "customer571@gmail.com",
        "phone": "096464128001",
        "address": "Gia Lai, 23/941 Hung Vuong"
    },
    {
        "id": 156,
        "name": "Customer 681",
        "email": "customer281@gmail.com",
        "phone": "09646412301",
        "address": "Gia Lai, 23/21 Hung Vuong"
    }
```


id	address	email	name	phone
152	Gia Lai, 23/751 Hung Vuong	customer701@gmail.com	Nguyen Cao Trung Kien	09646412501
154	Gia Lai, 23/71 Hung Vuong	customer661@gmail.com	Customer 131	096464122491
155	Gia Lai, 23/941 Hung Vuong	customer571@gmail.com	Customer 891	096464128001
156	Gia Lai, 23/21 Hung Vuong	customer281@gmail.com	Customer 681	09646412301
NULL	NULL	NULL	NULL	NULL

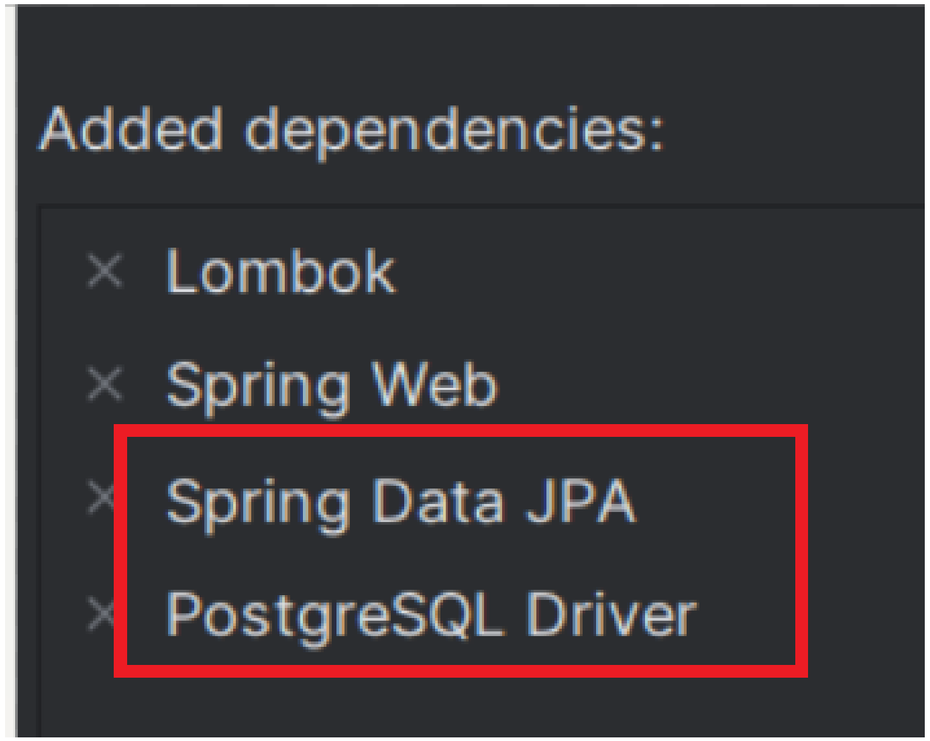
Spring Boot Data JPA with PostgreSQL

1) Connect to PostgreSQL:

- Add the necessary dependencies to **pom.xml(Maven project)** or **build.gradle(Gradle project)** file

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```



Spring Boot Data JPA with PostgreSQL

1) Connect to PostgreSQL

- Configure connection information to PostgreSQL: We need to provide configuration information such as the connection URL, username, and password of the postgreSQL database. This information is usually placed in the **application.properties** or **application.yml** file

```
#connect to database:
spring.datasource.url=jdbc:postgresql://localhost:5432/EmployeeManagement
spring.datasource.username=postgres
spring.datasource.password=trungkien

##JPA Config
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Spring Boot Data JPA with PostgreSQL

2) Create the @Entity Model:

```java
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "em_id")
    private Long id;
    @Column(name = "em_name")
    private String name;
    @Column(name = "em_age")
    private int age;
    @Column(name = "em_dob")
    private Date dob;
    @Column(name = "em_email")
    private String email;
```

Result in console:

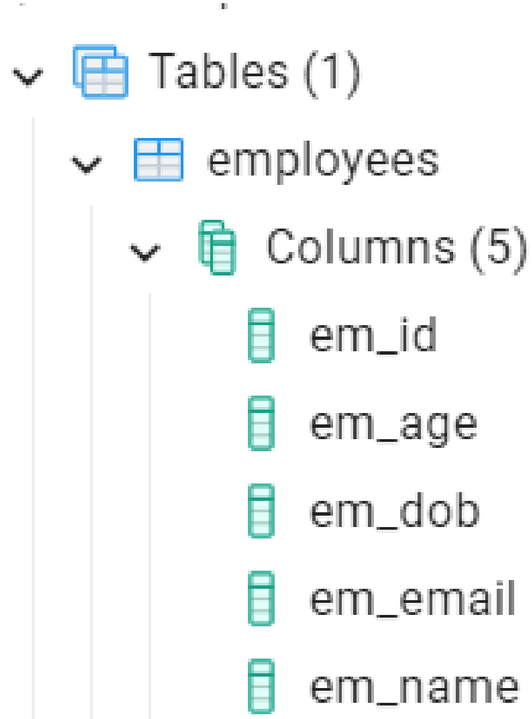
```
Hibernate:
    create table employees (
        em_id bigserial not null,
        em_age integer,
        em_dob timestamp(6),
        em_email varchar(255),
        em_name varchar(255),
        primary key (em_id)
    )
```

Spring Boot Data JPA with PostgreSQL

2) Create the @Entity Model:

```java
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "em_id")
    private Long id;
    @Column(name = "em_name")
    private String name;
    @Column(name = "em_age")
    private int age;
    @Column(name = "em_dob")
    private Date dob;
    @Column(name = "em_email")
    private String email;
```

Result in PostgreSQL:



Spring Boot Data JPA with PostgreSQL

3) Create Spring Data Repository:

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    //Optional<Employee> findById(long id);
    //Optional<Employee> updateById();
}
```

Spring Boot Data JPA with PostgreSQL

4) Use Repository in Controller:

```
@PostMapping
public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee) {
    Employee savedEmployee = rep.save(employee);
    return ResponseEntity.status(HttpStatus.CREATED).body(savedEmployee);
}
```

Create data

```
@GetMapping
public List<Employee> getList() { return rep.findAll(); }
new *
@GetMapping("/{id}")
public ResponseEntity<Employee> getById(@PathVariable("id") Long id) {
    Optional<Employee> employeeOptional = rep.findById(id);
    return employeeOptional.map(ResponseEntity::ok)
            .orElseGet(() -> ResponseEntity.notFound().build());
}
```

Get data

Spring Boot Data JPA with PostgreSQL

4) Use Repository in Controller:

```
@PutMapping("/{id}")
public ResponseEntity<Employee> updateEmployee(@PathVariable("id") Long id, @RequestBody Employee employee) {
    if (!rep.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    employee.setId(id);
    Employee updatedEmployee = rep.save(employee);
    return ResponseEntity.ok(updatedEmployee);
}
```

Update data

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteEmployee(@PathVariable("id") Long id) {
    if (!rep.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    rep.deleteById(id);
    return ResponseEntity.noContent().build();
}
```

Delete data

Testing API in Postman - Post method:

http://localhost:2902/employees

POST http://localhost:2902/employees

Params Authorization Headers (8) Body Pre-reque

none form-data x-www-form-urlencoded raw

```
{
    "name": "Trung Kien",
    "age": 18,
    "dob": "2003-11-27",
    "email": "kientrung@gmail.com"

}
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "id": 1,
    "name": "Trung Kien",
    "age": 18,
    "dob": "2003-11-27T00:00:00.000+00:00",
    "email": "kientrung@gmail.com"
}
```

POST http://localhost:2902/employees

Params Authorization Headers (8) Body Pre-request Scr

none form-data x-www-form-urlencoded raw b

```
{
    "name": "Trung Dung",
    "age": 18,
    "dob": "2003-11-27",
    "email": "kientrung@gmail.com"

}
```

ody Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "id": 2,
    "name": "Trung Dung",
    "age": 18,
    "dob": "2003-11-27T00:00:00.000+00:00",
    "email": "kientrung@gmail.com"
}
```

Testing API in Postman - Get method

GET http://localhost:2902/employees

Params Authorization Headers (6) Body Pre-request Scri

Query Params

	Key	Value
	Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "id": 1,
        "name": "Trung Kien",
        "age": 18,
        "dob": "2003-11-27T00:00:00.000+00:00",
        "email": "kientrung@gmail.com"
    },
    {
        "id": 2,
        "name": "Trung Dung",
        "age": 18,
        "dob": "2003-11-27T00:00:00.000+00:00",
        "email": "kientrung@gmail.com"
    }
]
```


http://localhost:2902/employees/1

GET http://localhost:2902/employees/1

Params Authorization Headers (6) Body Pre-requ

Query Params

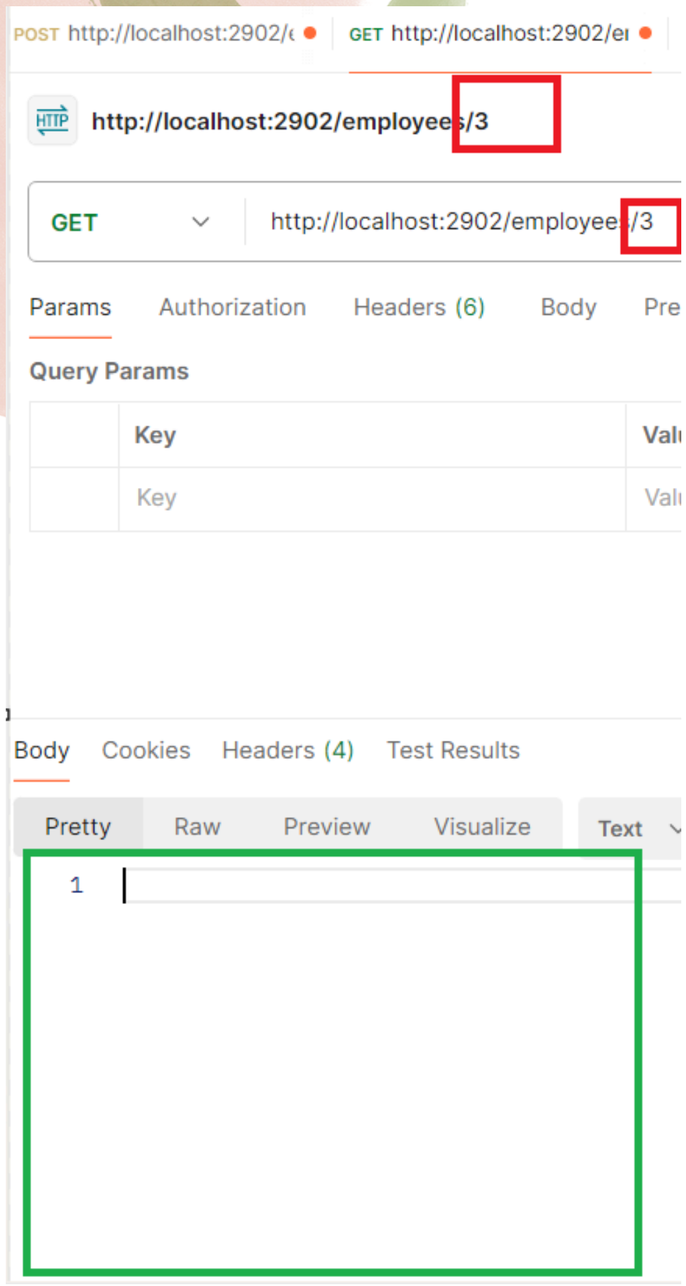
	Key	Value
	Key	Value

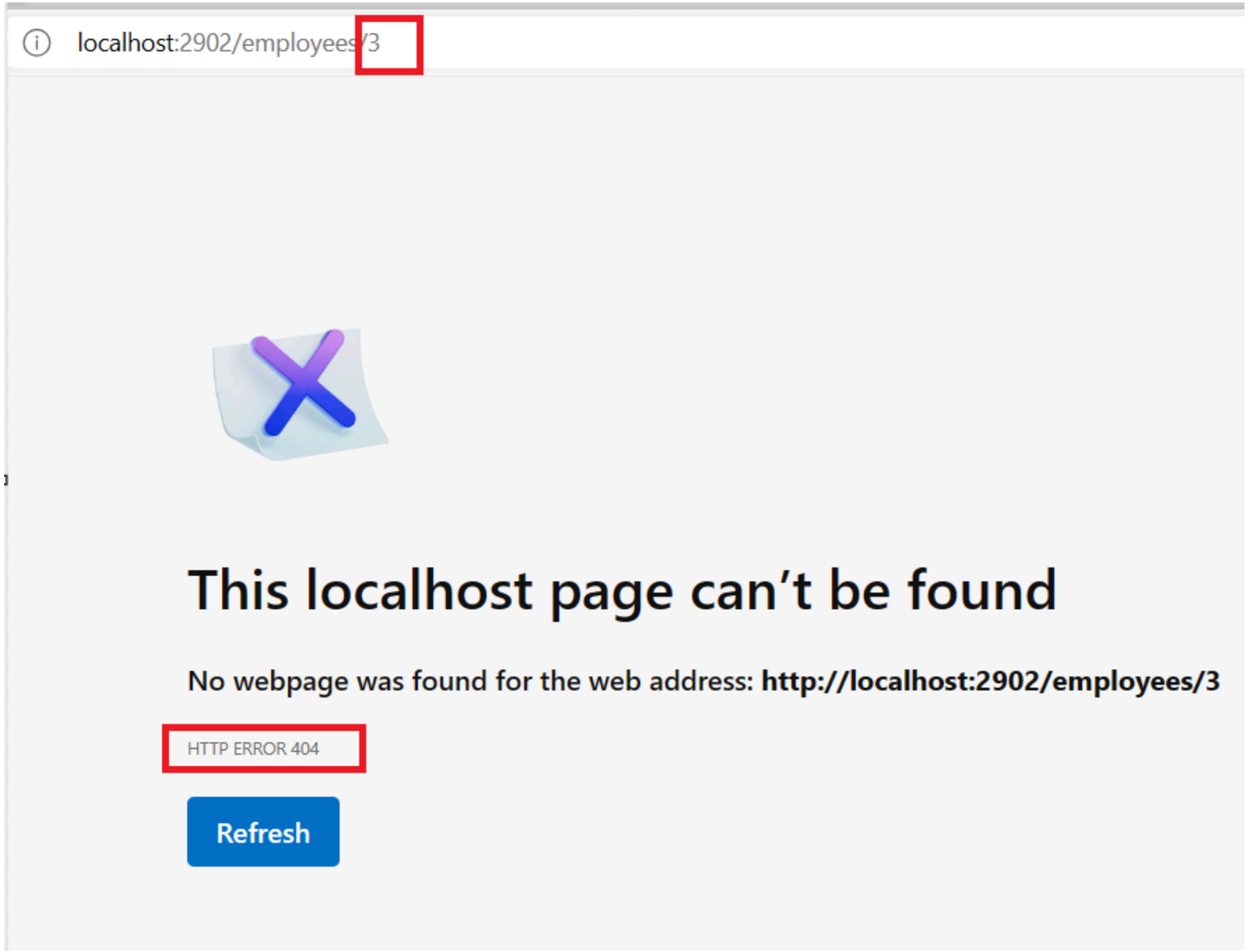
Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

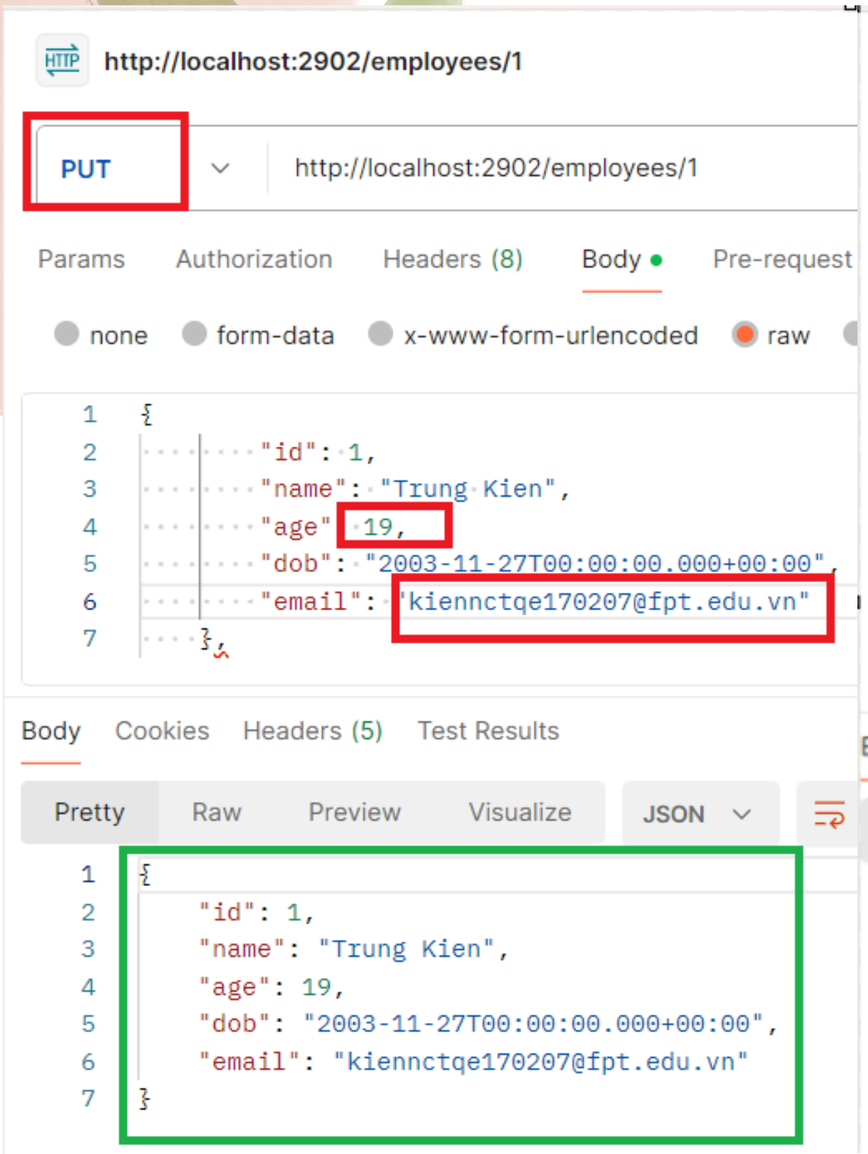

```
{
    "id": 1,
    "name": "Trung Kien",
    "age": 18,
    "dob": "2003-11-27T00:00:00.000+00:00",
    "email": "kientrung@gmail.com"
}
```

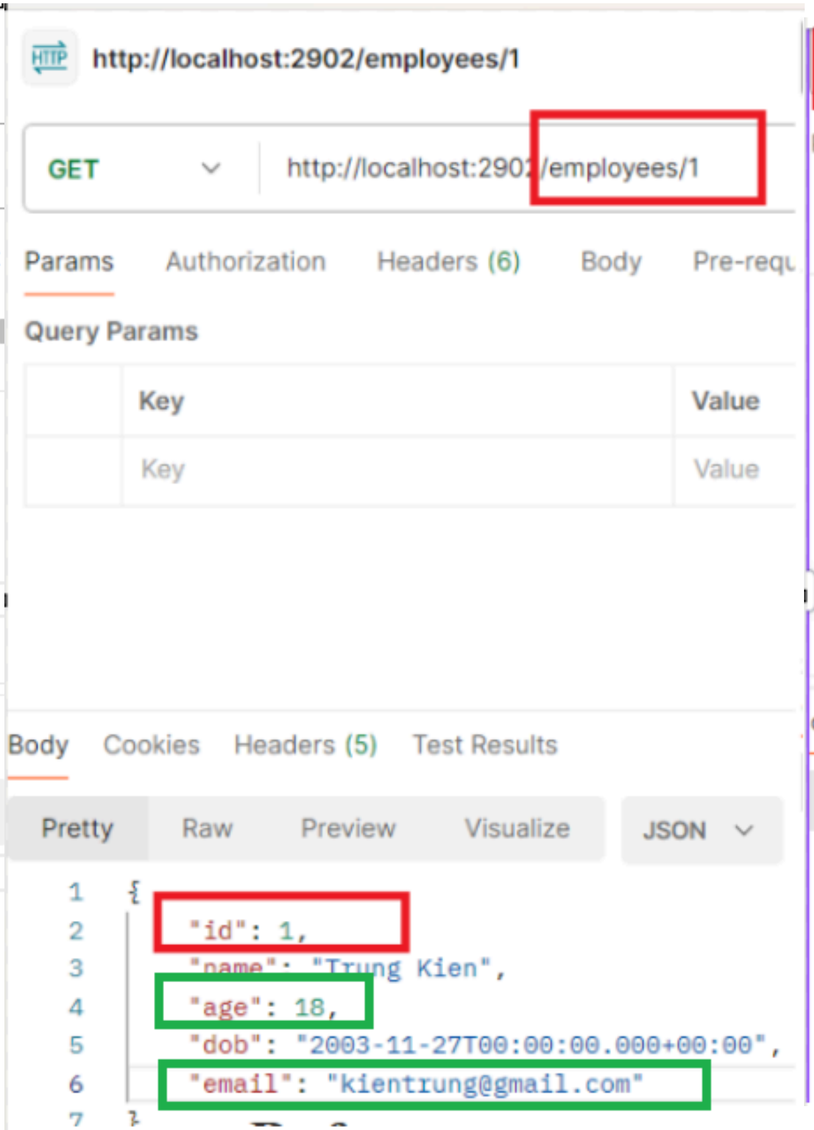
Testing API in Postman - Get method:



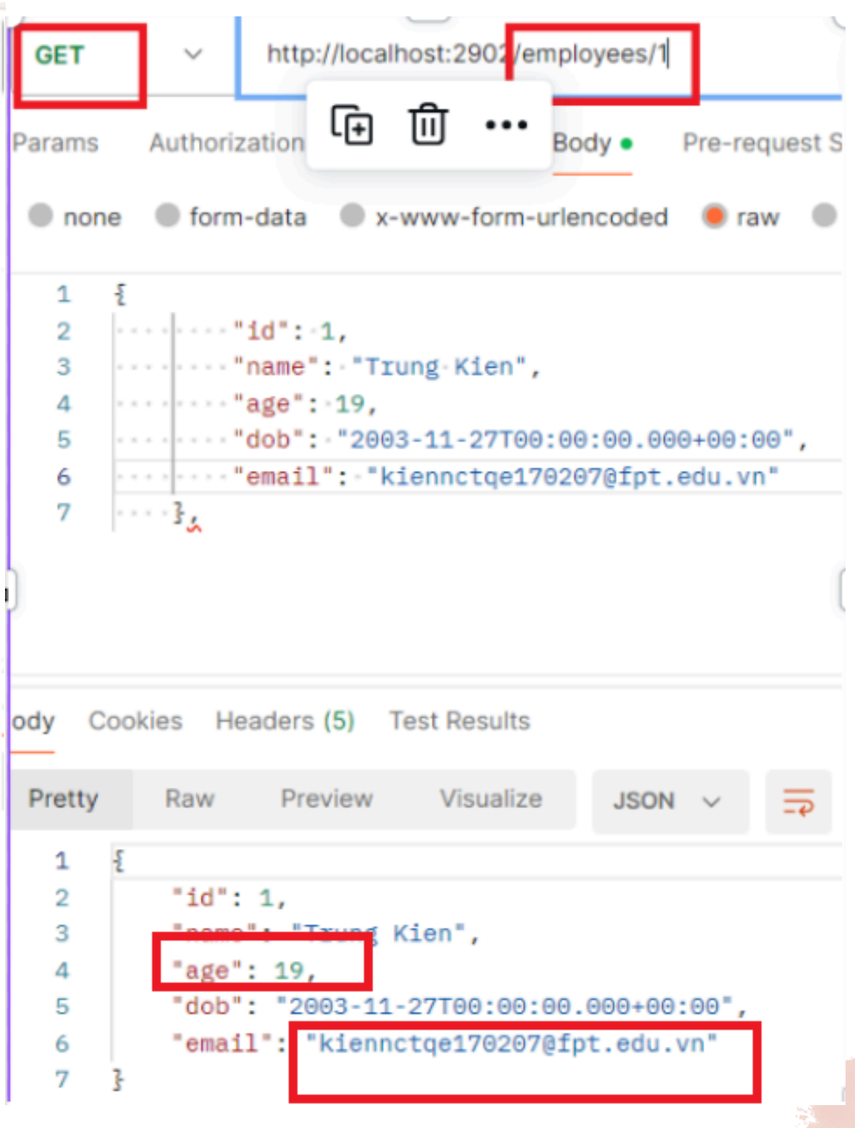


Testing API in Postman - Put method:





Before



After

Testing API in Postman - Delete Method

DELETE http://localhost:2902/employees/1

Params Authorization Headers (8) Body Pre-reques

none form-data x-www-form-urlencoded raw

```
{
    "id": 1,
    "name": "Trung Kien",
    "age": 19,
    "dob": "2003-11-27T00:00:00.000+00:00",
    "email": "kiennctqe170207@fpt.edu.vn"
},
```

Result:

http://localhost:2902/employees

GET http://localhost:2902/employees

Params Authorization Headers (8) Body Pre-request Script

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
[
    {
        "id": 2,
        "name": "Trung Dung",
        "age": 18,
        "dob": "2003-11-27T00:00:00.000+00:00",
        "email": "kientrung@gmail.com"
    },
    {
        "id": 3,
        "name": "Trung Dung",
        "age": 18,
        "dob": "2003-11-27T00:00:00.000+00:00",
        "email": "kientrung@gmail.com"
    }
]
```

Finally, here is the result in database after using CRUD method

GET http://localhost:2902/employees

Params Authorization Headers (8) Body Pre-request

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "id": 2,
        "name": "Trung Dung",
        "age": 18,
        "dob": "2003-11-27T00:00:00.000+00:00",
        "email": "kientrung@gmail.com"
    },
    {
        "id": 3,
        "name": "Trung Dung",
        "age": 18,
        "dob": "2003-11-27T00:00:00.000+00:00",
        "email": "kientrung@gmail.com"
    }
]
```


Data Output Messages Notifications

	em_id [PK] bigint	em_age integer	em_dob timestamp without time zone (6)	em_email character varying (255)	em_name character varying
1	2	18	2003-11-27 07:00:00	kientrung@gmail.com	Trung Dung
2	3	18	2003-11-27 07:00:00	kientrung@gmail.com	Trung Dung

Spring Data JPA with Sorting and Paging

1) **Sorting:** The process of arranging the results of a query according to one or more fields in the database.

2) **Paging:** The process of dividing query results into small sections called pages, helping to display data efficiently on the user interface.

1.1 Spring Data JPA with Sorting

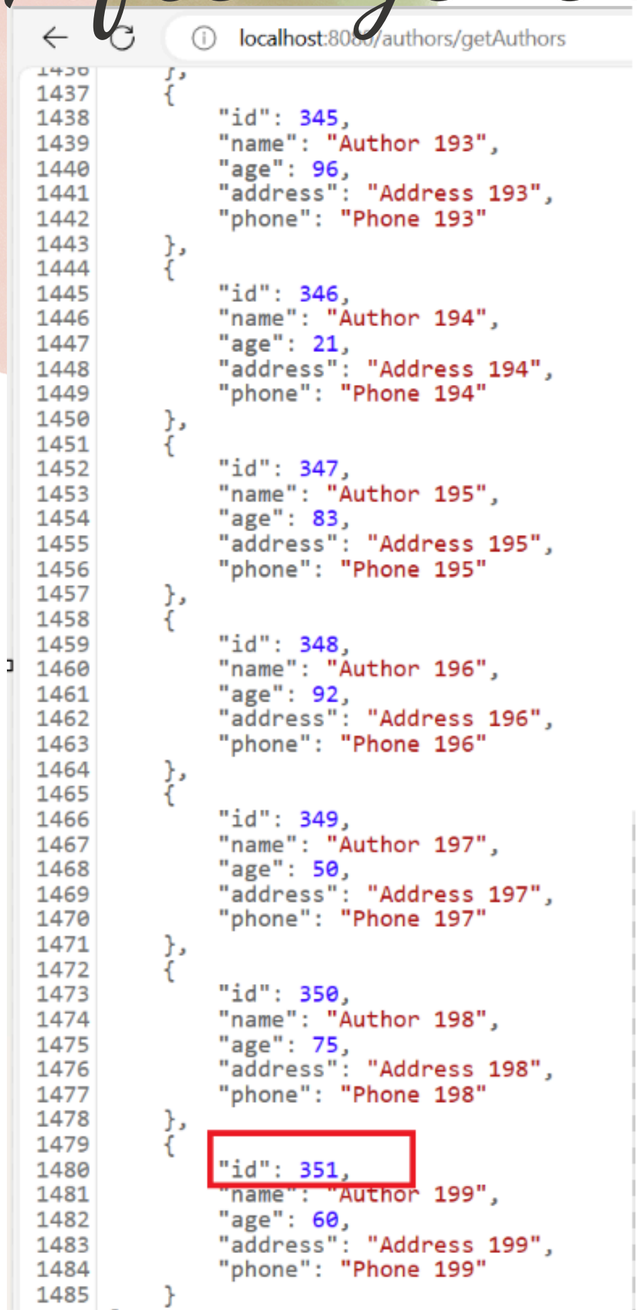
First we have a Author entity:

```java
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    4 usages
    private String name;
    4 usages
    private int age;
    4 usages
    private String address;
    4 usages
    private String phone;
```

Then we generate data for this entity:

```java
@PostConstruct
public void init() {
    for(int i = 0; i < 200; ++i) {
        rep.save(new Author( name: "Author " + i,  age: new Random().nextInt( bound: 80) + 20,
                address: "Address " + i,  phone: "Phone " + i));
    }
}
```

After generating:



In database:

id	address	age	name	phone
335	Address 183	37	Author 183	Phone 183
336	Address 184	31	Author 184	Phone 184
337	Address 185	42	Author 185	Phone 185
338	Address 186	27	Author 186	Phone 186
339	Address 187	45	Author 187	Phone 187
340	Address 188	60	Author 188	Phone 188
341	Address 189	97	Author 189	Phone 189
342	Address 190	70	Author 190	Phone 190
343	Address 191	80	Author 191	Phone 191
344	Address 192	25	Author 192	Phone 192
345	Address 193	96	Author 193	Phone 193
346	Address 194	21	Author 194	Phone 194
347	Address 195	83	Author 195	Phone 195
348	Address 196	92	Author 196	Phone 196
349	Address 197	50	Author 197	Phone 197
350	Address 198	75	Author 198	Phone 198
351	Address 199	60	Author 199	Phone 199
NULL	NULL	NULL	NULL	NULL

1.1.1. Spring Data JPA sorting with multiply parameter field

```java
@NonNullApi
public interface AuthorRepository extends PagingAndSortingRepository<Author, Long> {
    /* By having extend PagingAndSortingRepository, we get findAll(Sort sort)
       and findAll(Pageable pageable) method for sorting and paging
     */
}
```

First way

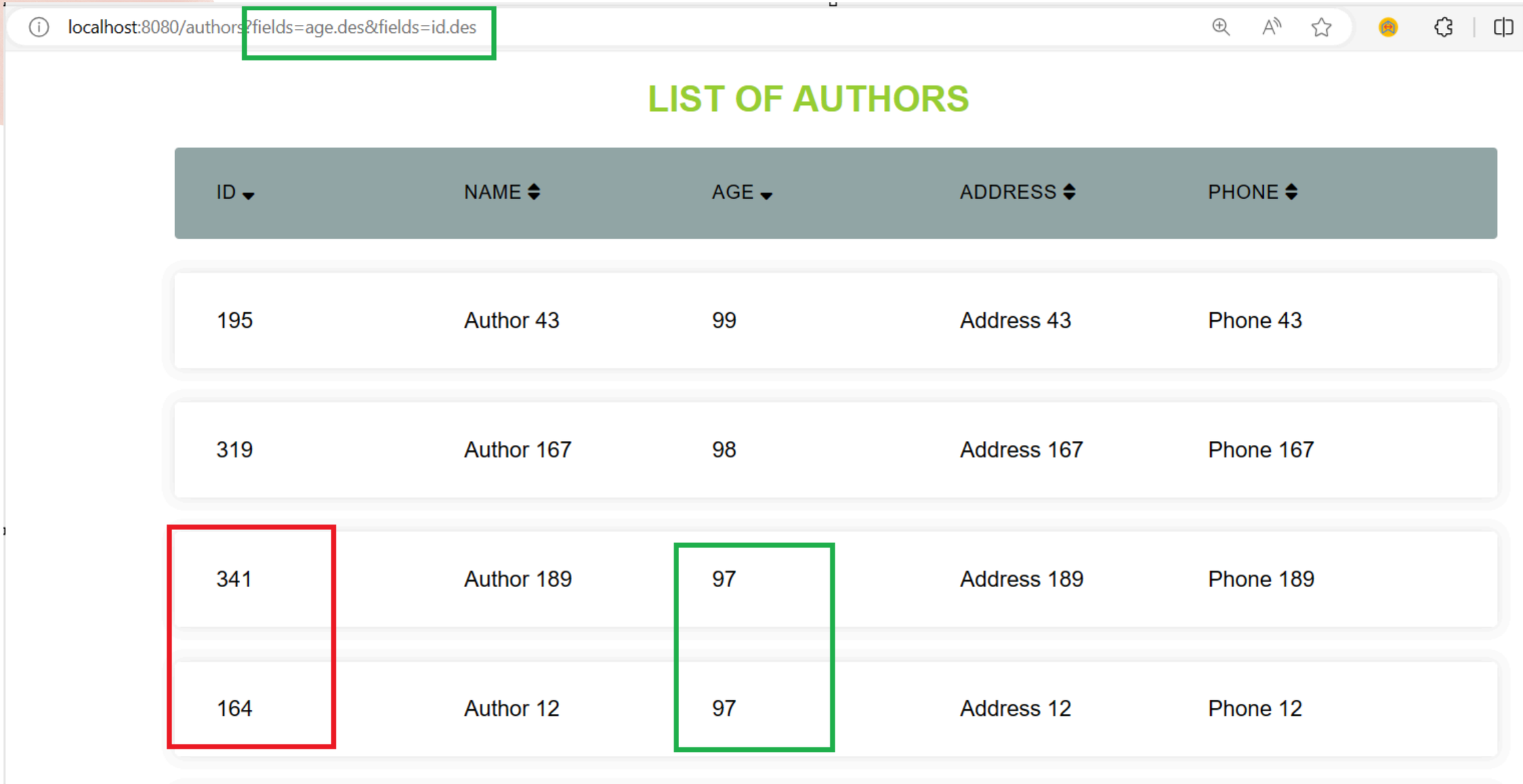
```java
@NonNullApi
public interface AuthorRepository extends JpaRepository<Author, Long> {
    /* or we can also choose extends JpaRepository instead, as it
       extends PagingAndSortingRepository too
     */
}
```

Second way

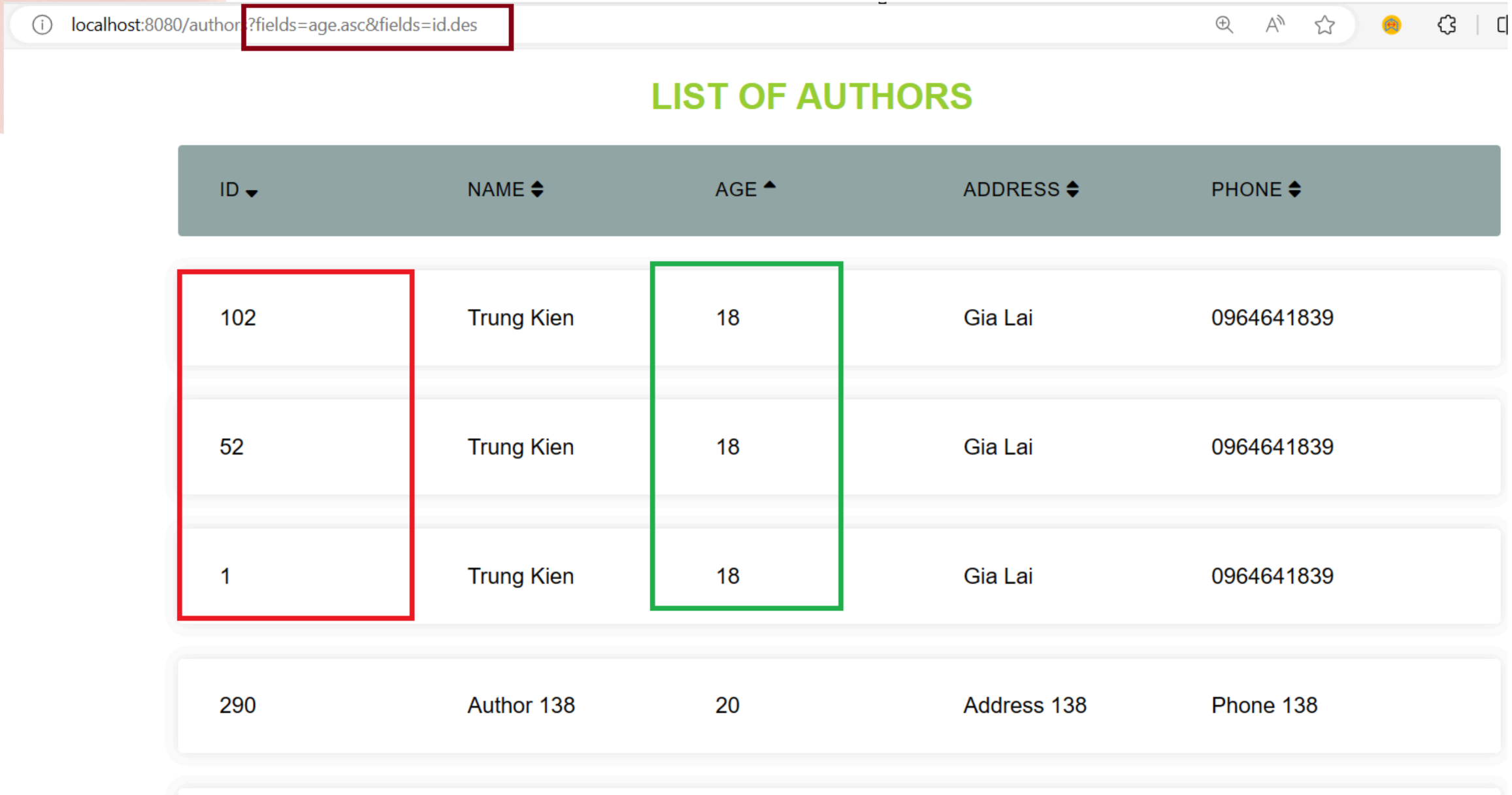
1.1.1. Spring Data JPA sorting with multiply parameter field

```java
@GetMapping("")
public ModelAndView author(@RequestParam(defaultValue = "id.asc") String[] fields,
                           @RequestParam(defaultValue = "0") int page, Model model) {
    // Parse fields and directions
    Sort.Order[] orders = new Sort.Order[fields.length];//create array to save order sort
    for (int i = 0; i < fields.length; i++) {
        String[] parts = fields[i].split( regex: "\\."); //parse part in field into 2 part, split by "."
        Sort.Direction direction = parts[1].equalsIgnoreCase( anotherString: "asc") ?
                                   Sort.Direction.ASC : Sort.Direction.DESC;
        //cf the sort direction based on the second value in the parts array, ex: id.asc
        orders[i] = new Sort.Order(direction, parts[0]); //contains fields with their direction
    }
    Sort sort = Sort.by(orders);
```

1.1.1. Spring Data JPA sorting with multiply parameter field



1.1.1. Spring Data JPA sorting with multiply parameter field



1.1.1. Spring Data JPA sorting with multiply parameter field

localhost:8080/authors?fields=age.asc&fields=id.des&fields=address.asc&fields=name.asc&fields=phone.asc

LIST OF AUTHORS

ID	NAME	AGE	ADDRESS	PHONE
102	Trung Kien	18	Gia Lai	0964641839
52	Trung Kien	18	Gia Lai	0964641839
1	Trung Kien	18	Gia Lai	0964641839
290	Author 138	20	Address 138	Phone 138

1.2. Spring Data JPA with Paging

First we have a Author entity:

```java
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    4 usages
    private String name;
    4 usages
    private int age;
    4 usages
    private String address;
    4 usages
    private String phone;
```

Then we generate data for this entity:

```java
@PostConstruct
public void init() {
    for(int i = 0; i < 200; ++i) {
        rep.save(new Author( name: "Author " + i,  age: new Random().nextInt( bound: 80) + 20,
                address: "Address " + i,  phone: "Phone " + i));
    }
}
```

1.2. Spring Data JPA with Paging

```
final int PAGE_SIZE = 20;
// Parse fields and directions
Sort.Order[] orders = new Sort.Order[fields.length];//create array to save order sort
for (int i = 0; i < fields.length; i++) {
    String[] parts = fields[i].split( regex: "\\."); //parse part in field into 2 part, split by
    Sort.Direction direction = parts[1].equalsIgnoreCase( anotherString: "asc") ?
                                    Sort.Direction.ASC : Sort.Direction.DESC;
    //cf the sort direction based on the second value in the parts array, ex: id.asc
    orders[i] = new Sort.Order(direction, parts[0]); //contains fields with their direction
}
Sort sort = Sort.by(orders);
PageRequest pageRequest = PageRequest.of(page, PAGE_SIZE, sort);
Page<Author> authorPage = rep.findAll(pageRequest);


// Fetch authors with sorting
//List<Author> authors = rep.findAll(sort);
//model.addAttribute("authors", authors);
model.addAttribute( attributeName: "authors", authorPage.getContent());
model.addAttribute( attributeName: "fields", fields);
model.addAttribute( attributeName: "currentPage", page);
model.addAttribute( attributeName: "totalPages", authorPage.getTotalPages());
return new ModelAndView( viewName: "author");
```

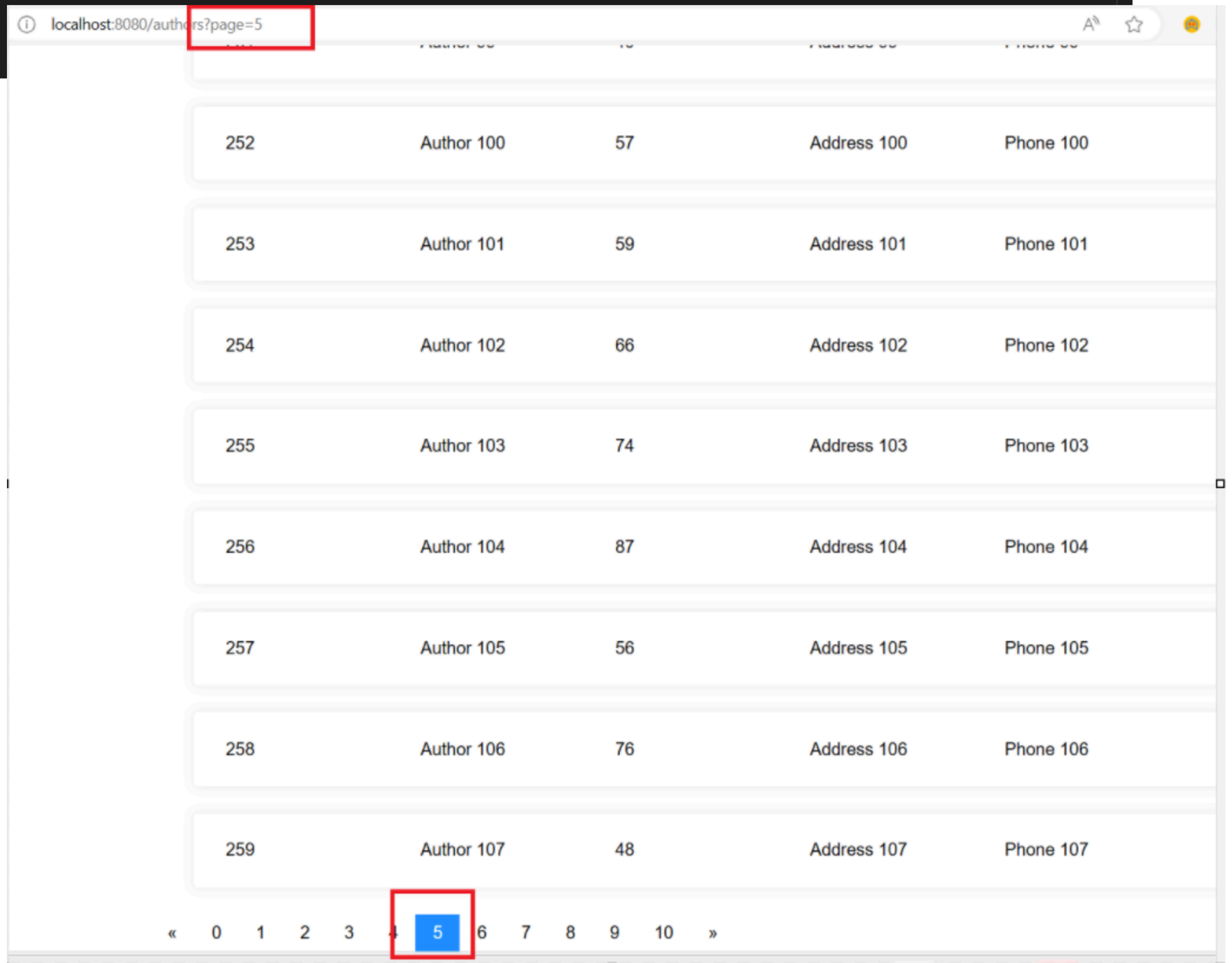
```
<a href="#">&laquo;</a>
<th:block th:each="pageNumber: ${#numbers.sequence(0, totalPages - 1)}">
    <a href="#" th:href="@{/authors(page=${pageNumber})}" th:class="${pageNumber == currentPage ? 'active' : ''}"
       th:text="${pageNumber}"></a>
</th:block>
<a href="#">&raquo;</a>
```

localhost:8080/authors?page=0

152	Author 0	51	Address 0	Phone 0
153	Author 1	75	Address 1	Phone 1
154	Author 2	48	Address 2	Phone 2
155	Author 3	60	Address 3	Phone 3
156	Author 4	69	Address 4	Phone 4
157	Author 5	57	Address 5	Phone 5
158	Author 6	84	Address 6	Phone 6
159	Author 7	37	Address 7	Phone 7

« 0 1 2 3 4 5 6 7 8 9 10 »

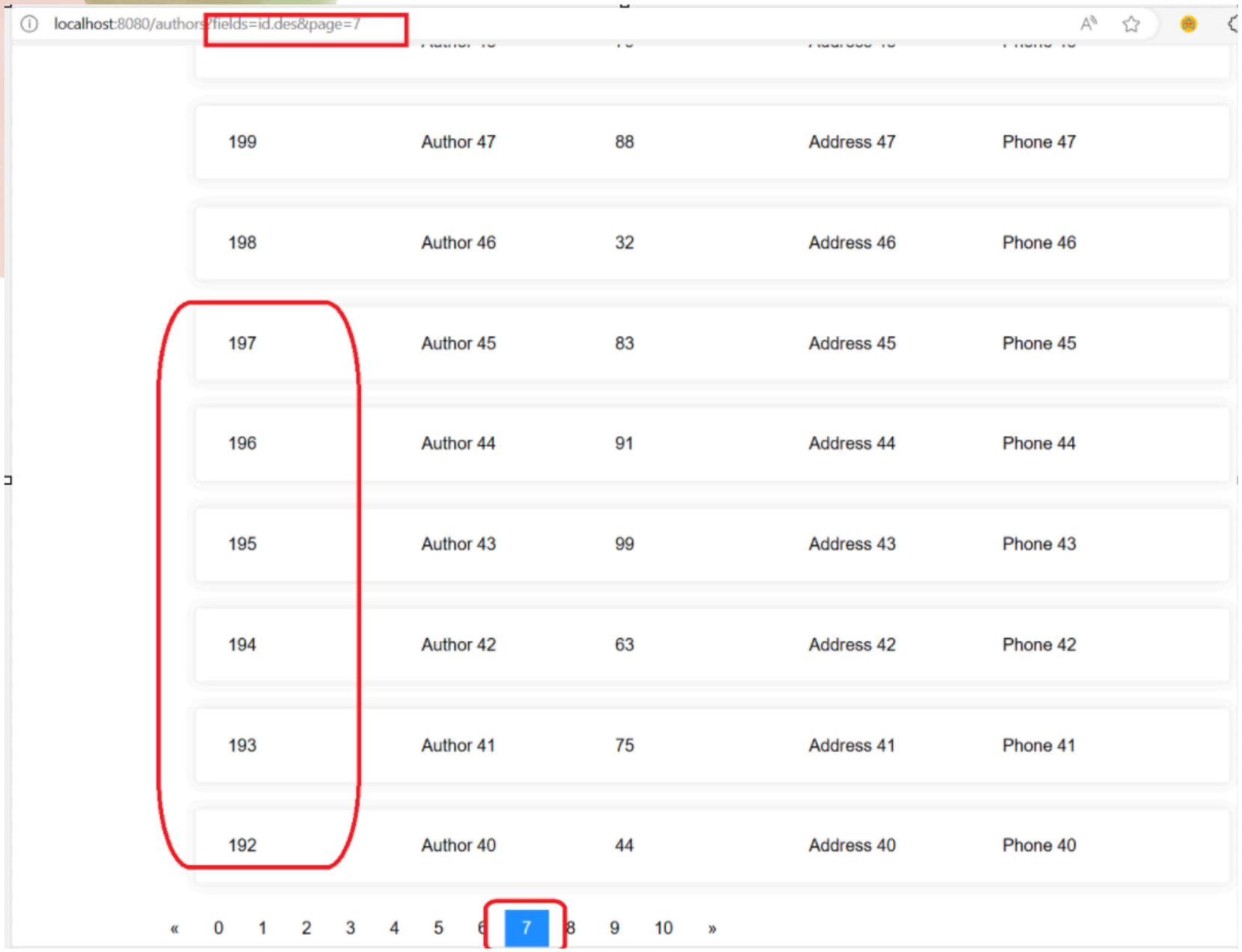
```
<a href="#">&laquo;</a>
<th:block th:each="pageNumber: ${#numbers.sequence(0, totalPages - 1)}">
    <a href="#" th:href="@{/authors(page=${pageNumber})}" th:class="${pageNumber == currentPage ? 'active' : ''}"
       th:text="${pageNumber}"></a>
</th:block>
<a href="#">&raquo;</a>
```



1.3. Spring Data JPA with Paging and Sorting with multiple criteria:

```
public ModelAndView author(@RequestParam(defaultValue = "id.asc") String[] fields,
                           @RequestParam(defaultValue = "0") int page, Model model) {
    final int PAGE_SIZE = 20;
    // Parse fields and directions
    Sort.Order[] orders = new Sort.Order[fields.length];//create array to save order sort
    for (int i = 0; i < fields.length; i++) {
        String[] parts = fields[i].split( regex: "\\."); //parse part in field into 2 part, sp
        Sort.Direction direction = parts[1].equalsIgnoreCase( anotherString: "asc") ?
                                   Sort.Direction.ASC : Sort.Direction.DESC;
        //cf the sort direction based on the second value in the parts array, ex: id.asc
        orders[i] = new Sort.Order(direction, parts[0]); //contains fields with their direct
    }
    Sort sort = Sort.by(orders);
    PageRequest pageRequest = PageRequest.of(page, PAGE_SIZE, sort);
    Page<Author> authorPage = rep.findAll(pageRequest);
    // Fetch authors with sorting
    //List<Author> authors = rep.findAll(sort);
    //model.addAttribute("authors", authors);
    model.addAttribute( attributeName: "authors", authorPage.getContent());
    model.addAttribute( attributeName: "fields", fields);
    model.addAttribute( attributeName: "currentPage", page);
    model.addAttribute( attributeName: "totalPages", authorPage.getTotalPages());
    return new ModelAndView( viewName: "author");
```

The result on localhost:



The result on localhost:

localhost:8080/authors?fields=age.des&fields=id.asc&page=7

LIST OF AUTHORS

ID ▲	NAME ⇕	AGE ▼	ADDRESS ⇕	PHONE ⇕
312	Author 160	43	Address 160	Phone 160
218	Author 66	42	Address 66	Phone 66
236	Author 84	42	Address 84	Phone 84
308	Author 156	42	Address 156	Phone 156
310	Author 158	42	Address 158	Phone 158

JPA Specification in Spring Boot

- **JPA Specification** in the Spring Boot refers to the use of JPA standards to build flexible and diverse queries in Spring Boot applications.

- **JPA Specification** provides a flexible approach for creating dynamic query conditions based on different criteria.

JPA Specification

- JPA Specification in the Spring Boot refers to the use of JPA standards to build flexible and diverse queries in Spring Boot applications.
- JPA Specification provides a flexible approach for creating dynamic query conditions based on different criteria.

```java
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    4 usages
    private String name;
    4 usages
    private int age;
    4 usages
    private String address;
    4 usages
    private String phone;
```

Problem: We want to create a dynamic query to retrieve all users whose age is greater than a certain value(for example > 20)
-> We can use the JPA Specification to do this.

JPA Specification in Spring Boot

First, we define JPA Specification to fetch all author with **age > certain value**.

```
public static Specification<Author> getAllGreaterThan(int age) {
    new *
    return new Specification<Author>() {
        no usages   new *
        @Override
        public Predicate toPredicate(Root<Author> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {
            return criteriaBuilder.greaterThan(root.get("age"), age);
        }
    };
}
```

- **public Predicate toPredicate(Root<Author> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder)**: This is an implementation of the **toPredicate()** method. This method is override from the **Specification interface** and is where we define query conditions. In this case, we want to fetch all authors whose age is greater than the given age value.
- **criteriaBuilder** was used to create a greater than comparison expression.

JPA Specification in Spring Boot

After we have defined the Specification, we can use it in repository methods to perform queries:

```
public interface AuthorRepository extends JpaRepository<Author, Long> {
    1 usage  new *
    List<Author> findAll(Specification<Author> spec);
}
```

After that, we can then use UserRepository to retrieve all users whose age is greater than a certain value.

JPA Specification in Spring Boot

After we have defined the Specification, we can use it in repository methods to perform queries:

```
public interface AuthorRepository extends JpaRepository<Author, Long> {
    1 usage  new *
    List<Author> findAll(Specification<Author> spec);
}
```

After that, we can then use UserRepository to retrieve all users whose age is greater than a certain value.

```
@GetMapping("/getGreaterThan")
public List<Author> getGreaterThan() {
    return rep.findAll(AuthorService.getAllGreaterThan( age: 20));
}
```

Here is the result in localhost:

Before:

localhost:8080/authors/getAuthors


```
[
    {
        "id": 1,
        "name": "Trung Kien",
        "age": 18,
        "address": "Gia Lai",
        "phone": "0964641839"
    },
    {
        "id": 2,
        "name": "Trung Hieu",
        "age": 20,
        "address": "Phu Yen",
        "phone": "0943781244"
    },
    {
        "id": 3,
        "name": "Trung Dung",
        "age": 21,
        "address": "Quang Ngai",
        "phone": "0671345673"
    },
    {
        "id": 4,
        "name": "Trung Nguyen",
        "age": 22,
        "address": "Binh Dinh",
        "phone": "0913467845"
    },
    {
```


After:

localhost:8080/authors/getGreaterThan


```
    {
        "id": 3,
        "name": "Trung Dung",
        "age": 21,
        "address": "Quang Ngai",
        "phone": "0671345673"
    },
    {
        "id": 4,
        "name": "Trung Nguyen",
        "age": 22,
        "address": "Binh Dinh",
        "phone": "0913467845"
    },
    {
        "id": 54,
        "name": "Trung Dung",
        "age": 21,
        "address": "Quang Ngai",
        "phone": "0671345673"
    },
    {
        "id": 55,
        "name": "Trung Nguyen",
        "age": 22,
        "address": "Binh Dinh",
        "phone": "0913467845"
    },
    {
        "id": 104,
        "name": "Trung Dung",
        "age": 21,
```

JPA Specification join multiple tables

Suppose we have two objects **“Author”** and **“Book”**, and the relationship between them is **one-to-many.** At the same time, each book has a corresponding category.

```java
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    4 usages
    private String name;
    no usages
    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author authorID;

    no usages
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category categoryID;
```

id	name	author_id	category_id
1	Math	1	1
2	Doraemon	1	2
3	Conan	1	2
4	Fire	52	2
5	Harry Potter	54	1
6	One Piece	54	1
NULL	NULL	NULL	NULL

id	name
1	Comic
2	Text Book

Problem: Show all books that was written by an author and its category

MySQL syntax jo **JOIN** multiply tables:

```
select author.*, book.name, book.category_id, category.name
from author join book on author.id = book.author_id
join category on book.category_id = category.id;
```

Result table:

id	address	age	name	phone	name	category_id	name
1	Gia Lai	18	Trung Kien	0964641839	Math	1	Comic
1	Gia Lai	18	Trung Kien	0964641839	Doraemon	2	Text Book
1	Gia Lai	18	Trung Kien	0964641839	Conan	2	Text Book
52	Gia Lai	18	Trung Kien	0964641839	Fire	2	Text Book
54	Quang Ngai	21	Trung Dung	0671345673	Harry Potter	1	Comic
54	Quang Ngai	21	Trung Dung	0671345673	One Piece	1	Comic

JPA Specification join multiple tables

1) Create a Specification class to featch all books that was written by an author

```
public class BookSpecification {
    1 usage  new *
    public static Specification<Book> authoredBy(Author author) {
        return((root, query, criteriaBuilder) -> {
            Join<Book, Author> authorJoin = root.join( s: "authorID", JoinType.INNER);
            return criteriaBuilder.equal(authorJoin.get("id"), author.getId());
        });
    }
}
```

2) Use the repository to search through the defined specification

```
7 usages  new *
public interface BookRepository extends JpaRepository<Book, Long>, JpaSpecificationExecutor<Book> {}
```

JPA Specification join multiple tables

3) Use Specification in Controller:

```
@GetMapping("/{authorId}/books")
public List<Book> getBooksByAuthor(@PathVariable Long authorId) {
    Author author = rep.findById(authorId).orElse( other: null);
    if (author == null) {
        System.out.println("No books was found with author ID " + authorId);
    }

    Specification<Book> spec = BookSpecification.authoredBy(author);
    return bookRepository.findAll(spec);
}
```

JPA Specification join multiple tables

Result:

localhost:8080/authors/1/books


```
{
    "id": 1,
    "name": "Math",
    "authorID": {
        "id": 1,
        "name": "Trung Kien",
        "age": 18,
        "address": "Gia Lai",
        "phone": "0964641839"
    },
    "categoryID": {
        "id": 1,
        "name": "Comic"
    }
},
{
    "id": 2,
    "name": "Doraemon",
    "authorID": {
        "id": 1,
        "name": "Trung Kien",
        "age": 18,
        "address": "Gia Lai",
        "phone": "0964641839"
    },
    "categoryID": {
        "id": 2,
        "name": "Text Book"
    }
},
{
    "id": 3,
    "name": "Conan",
    "authorID": {
        "id": 1,
        "name": "Trung Kien",
        "age": 18,
        "address": "Gia Lai",
        "phone": "0964641839"
    },
    "categoryID": {
        "id": 2,
        "name": "Text Book"
    }
}
```


id	address	age	name	phone	name	category_id	name
1	Gia Lai	18	Trung Kien	0964641839	Math	1	Comic
1	Gia Lai	18	Trung Kien	0964641839	Doraemon	2	Text Book
1	Gia Lai	18	Trung Kien	0964641839	Conan	2	Text Book

localhost:8080/authors/6/books

OOPS!

SOMETHING WENT WRONG!

GO TO HOMEPAGE

Rest query languare with JPA Specification

Rest Query Language is a method that allows to create RESTful queries easily using Spring Data JPA Specification in Spring Boot application.

For example, user can pass search conditions like "**name=Kien&age=20**" to get a list of objects with the name "**Kien**" and age **20.**

For instance of RESTful queries:

- **GET /products: returns a list of all products currently in the database.**
- **GET /products/123: return information about the product with ID 123**
- **PUT /products/{id}: updates the information of the product whose ID is {id} with the new information provided in the body of the PUT request.**
- **PATCH/products{id}{price: 123}**
- **DELETE /products/{id}: delete the product with ID 123**

Rest query languare with JPA Specification

Rest Query Language is a method that allows to create RESTful queries easily using Spring Data JPA Specification in Spring Boot application.

1) First, we have “Book” entity

```
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    4 usages
    private String name;
    2 usages
    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author authorID;
    2 usages
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category categoryID;
```

2) Using Specification:

spec = spec.and: Build the filter condition

```
@GetMapping("/getBooks")
public ResponseEntity<Object> getBooks(@RequestParam(required = false) String authorName,
                                       @RequestParam(required = false) String categoryName,
                                       @RequestParam(required = false) Integer greaterAuthorAge) {
    Specification<Book> spec = Specification.where( spec: null);
    if (authorName != null) {
        spec = spec.and((root, query, criteriaBuilder) -> {
            Join<Book, Author> authorJoin = root.join( s: "authorID");
            return criteriaBuilder.equal(authorJoin.get("name"), authorName);
        });
    }
if (categoryName != null) {
    spec = spec.and((root, query, criteriaBuilder) -> {
        Join<Book, Category> categoryJoin = root.join( s: "categoryID");
        return criteriaBuilder.equal(categoryJoin.get("name"), categoryName);
    });
}
if (greaterAuthorAge != null) {
    spec = spec.and((root, query, criteriaBuilder) -> {
        Join<Book, Author> authorJoin = root.join( s: "authorID");
        return criteriaBuilder.gt(authorJoin.get("age"), greaterAuthorAge); // filter books
    });
}
```

id	name	author_id	category_id
1	Math	1	1
2	Doraemon	1	2
3	Conan	1	2
4	Fire	52	2
5	Harry Potter	54	1
6	One Piece	54	1
NULL	NULL	NULL	NULL

The result:

localhost:8080/books


```
[
    {
        "id": 1,
        "name": "Math",
        "authorID": {
            "id": 1,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 1,
            "name": "Comic"
        }
    },
    {
        "id": 2,
        "name": "Doraemon",
        "authorID": {
            "id": 1,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 2,
            "name": "Text Book"
        }
    },
    {
        "id": 3,
        "name": "Conan",
        "authorID": {
            "id": 1,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 2,
            "name": "Text Book"
        }
    },
    {
        "id": 4,
        "name": "Fire",
        "authorID": {
            "id": 52,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 2,
            "name": "Text Book"
        }
    },
    {
        "id": 5,
        "name": "Harry Potter",
        "authorID": {
            "id": 54,
            "name": "Trung Dung",
            "age": 21,
            "address": "Quang Ngai",
            "phone": "0671345673"
        },
        "categoryID": {
            "id": 1,
            "name": "Comic"
        }
    },
    {
        "id": 6,
        "name": "One Piece",
        "authorID": {
            "id": 54,
            "name": "Trung Dung",
            "age": 21,
            "address": "Quang Ngai",
            "phone": "0671345673"
        },
```


	id	name	author_id	category_id
▶	1	Math	1	1
	2	Doraemon	1	2
	3	Conan	1	2
	4	Fire	52	2
	5	Harry Potter	54	1
	6	One Piece	54	1
*	NULL	NULL	NULL	NULL

Now, when we access to “/books/?authorName=**Trung%20Kien**” or “**Trung Dung**”

localhost:8080/books?authorName=Trung%20Kien

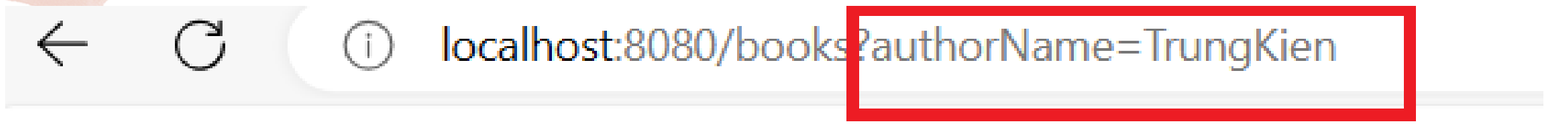

```
{
    "id": 1,
    "name": "Math",
    "authorID": {
        "id": 1,
        "name": "Trung Kien",
        "age": 18,
        "address": "Gia Lai",
        "phone": "0964641839"
    },
    "categoryID": {
        "id": 1,
        "name": "Comic"
    }
},
{
    "id": 2,
    "name": "Doraemon",
    "authorID": {
        "id": 1,
        "name": "Trung Kien",
        "age": 18,
        "address": "Gia Lai",
        "phone": "0964641839"
    },
    "categoryID": {
        "id": 2,
        "name": "Text Book"
    }
},
{
    "id": 3,
    "name": "Conan",
    "authorID": {
        "id": 1
        "name": "Trung Kien",
        "age": 18,
```


localhost:8080/books/getBooks?authorName=Trung%20Dung

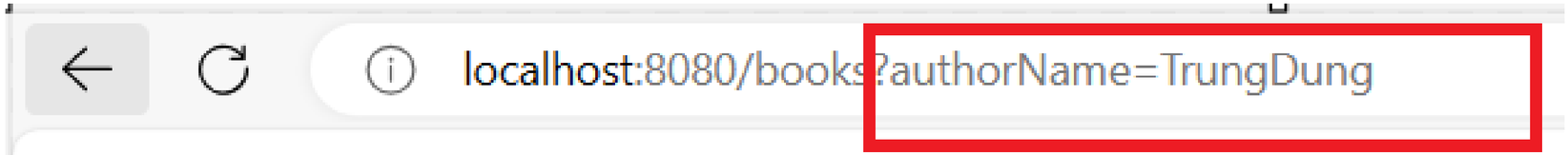

```
[
    {
        "id": 5,
        "name": "Harry Potter",
        "authorID": {
            "id": 54,
            "name": "Trung Dung",
            "age": 21,
            "address": "Quang Ngai",
            "phone": "0671345673"
        },
        "categoryID": {
            "id": 1,
            "name": "Comic"
        }
    },
    {
        "id": 6,
        "name": "One Piece",
        "authorID": {
            "id": 54,
            "name": "Trung Dung",
            "age": 21,
            "address": "Quang Ngai",
            "phone": "0671345673"
        },
        "categoryID": {
            "id": 1,
            "name": "Comic"
        }
    }
```

Now, when we access to “/books/?authorName=**TrungKien**” or “**TrungDung**” (Which have not been in database)

```
List<Book> books = rep.findAll(spec);
if (books.isEmpty()) {
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body("No books found based on the provided criteria!.");
}
```



No books found based on the provided criteria!.



No books found based on the provided criteria!.

Now, when we access to “/books/getBooks?**authorName**=**Trung%20Kien**” and **categoryName** = “**Text Book**”

localhost:8080/books/getBooks?authorName=Trung%20Kien&categoryName=Text%20Book


```
[
    {
        "id": 2,
        "name": "Doraemon",
        "authorID": {
            "id": 1,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 2,
            "name": 'Text Book"
        }
    },
    {
        "id": 3,
        "name": "Conan",
        "authorID": {
            "id": 1,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 2,
            "name": 'Text Book"
        }
    {
        "id": 4,
        "name": "Fire",
        "authorID": {
            "id": 52,
            "name": "Trung Kien",
            "age": 18,
            "address": "Gia Lai",
            "phone": "0964641839"
        },
        "categoryID": {
            "id": 2,
            "name": "Text Book"
        }
    }
```

Now, when we access to “/books/getBooks?**greaterAuthorAge=20**”

localhost:8080/books/getBooks?greaterAuthorAge=20


```
{
    "id": 5,
    "name": "Harry Potter",
    "authorID": {
        "id": 54,
        "name": "Trung Dung",
        "age": 21,
        "address": "Quang Ngai",
        "phone": "0671345673"
    },
    "categoryID": {
        "id": 1,
        "name": "Comic"
    }
},
{
    "id": 6,
    "name": "One Piece",
    "authorID": {
        "id": 54,
        "name": "Trung Dung",
        "age": 21,
        "address": "Quang Ngai",
        "phone": "0671345673"
    },
    "categoryID": {
        "id": 1,
        "name": "Comic"
    }
}
```


localhost:8080/books/getBooks?greaterAuthorAge=30

No books found based on the provided criteria!.

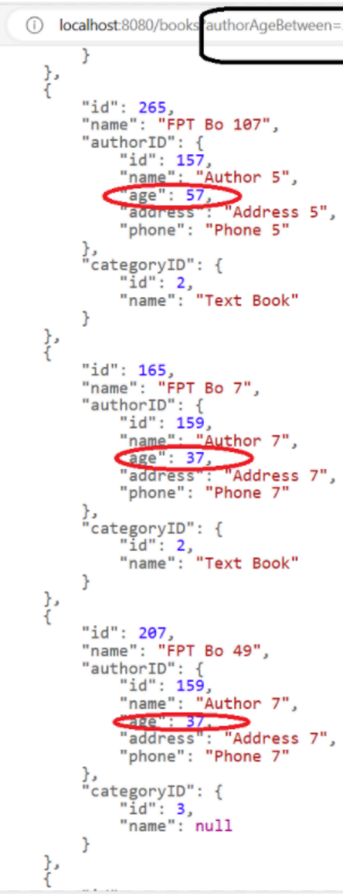
Now, when we access to “/books?**authorName=Trung%20Dung**&**greaterAuthorAge=20**&**categoryName=Comic**”

localhost:8080/books?authorName=Trung%20Dung&greaterAuthorAge=20&categoryName=Comic


```
{
    "id": 5,
    "name": "Harry Potter",
    "authorID": {
        "id": 54,
        "name": "Trung Dung",
        "age": 21,
        "address": "Quang Ngai",
        "phone": "0671345673"
    },
    "categoryID": {
        "id": 1,
        "name": "Comic"
    }
},
{
    "id": 6,
    "name": "One Piece",
    "authorID": {
        "id": 54,
        "name": "Trung Dung",
        "age": 21,
        "address": "Quang Ngai",
        "phone": "0671345673"
    },
    "categoryID": {
        "id": 1,
        "name": "Comic"
    }
}
```

```java
public static Specification<Book> searchBooks(Map<String, String> searchParams) {
    return (root, query, criteriaBuilder) -> {
        List<Predicate> predicates = new ArrayList<>();
        Join<Book, Author> authorJoin = root.join( s: "authorID");
        if (searchParams.containsKey("name")) {
            predicates.add(criteriaBuilder.like(root.get("name"), s: "%" + searchParams.get("name") + "%"));
        }
        if(searchParams.containsKey("id")) {
            predicates.add(criteriaBuilder.equal(root.get("id"), searchParams.get("id")));
        }
        if (searchParams.containsKey("authorName")) {
            predicates.add(criteriaBuilder.like(authorJoin.get("name"), s: "%" + searchParams.get("authorName") + "%"));
        }

        if (searchParams.containsKey("authorAge")) {
            predicates.add(criteriaBuilder.equal(authorJoin.get("age"), Integer.parseInt(searchParams.get("authorAge"))));
        }
        if (searchParams.containsKey("authorAddress")) {
            predicates.add(criteriaBuilder.like(authorJoin.get("address"), s: "%" + searchParams.get("authorAddress") + "%"));
```



FPT SOFTWARE QUY NHON

Greatful Thank You for your listening!

KienNCT3 - Mentor HaiNV21