`@Controller`

- This annotation marks a class as a **Spring MVC controller**.
- It is used to **handle web requests** and return **views (like HTML pages)** in traditional web applications (like JSP or Thymeleaf).
- Methods in this class typically return the **name of a view**, not raw data.

`@RestController`

- A specialized version of `@Controller`.
- Combines `@Controller` and `@ResponseBody`.
- Used in **REST APIs** where you want to return **JSON or XML** instead of views. ▯ All methods in this class will return **data directly to the client**.

`@SpringBootApplication`

- A **meta-annotation** that includes:
  - `@Configuration`: Marks the class as a Spring config class. ○ `@EnableAutoConfiguration`: Tells Spring Boot to configure the app automatically.
  - `@ComponentScan`: Automatically scans and registers beans in the package.
- It is placed on the **main class** to mark the entry point of a Spring Boot app.

`@RequestMapping`

- Used to **map web requests to specific handler methods** or controller classes.
- Can be used at the class level (base path) or method level (specific paths).
- Supports all HTTP methods (GET, POST, PUT, DELETE, etc.).

`@RequestParam`

- Binds a **URL query parameter** to a method argument.
- Useful for reading parameters like `/search?name=hari` or `/page?size=10`.

`@ResponseBody`

- Tells Spring to **return the method result directly as the HTTP response body**.
- Used in REST APIs to send data (like JSON) instead of view names.
- Automatically serializes Java objects to JSON (or XML).

`@RequestBody`

- Binds the **HTTP request body** to a method argument.
- Useful when sending **JSON or XML in a POST/PUT request**.
- Spring automatically **converts JSON into Java objects**.

`@GetMapping`

- Shortcut for `@RequestMapping(method = RequestMethod.GET)`
- Used for handling **GET requests**, which are used to **retrieve data**.

`@PostMapping`

- Shortcut for `@RequestMapping(method = RequestMethod.POST)`
- Used for handling **POST requests**, which are used to **submit or create data**.

What is a REST API?

REST API stands for **Representational State Transfer Application Programming Interface**.

It is a **way for different software systems to communicate with each other** over the internet using **standard HTTP methods** like `GET`, `POST`, `PUT`, `DELETE`, etc.

A **REST API** is like a **messenger** that allows your **frontend** (like a mobile app or website) to **talk to a backend server** to **get or send data**.

Concept	Meaning
Client	The app or browser that sends requests (e.g., a mobile app)
Server	The backend service that processes and sends responses
Resource	Any data object — e.g., user, product, file
Endpoint	A URL where the resource can be accessed
HTTP Methods	Actions that can be performed (GET, POST, PUT, DELETE)
JSON/XML	Common data formats used in REST communication

What is JSON?

JSON stands for **JavaScript Object Notation**.

It is a **lightweight, text-based format** for **storing and exchanging data** between systems — especially between a **client (like a web browser or mobile app)** and a **server**.

JSON is just **data in a text format** that looks like a **JavaScript object**. It is easy for both **humans to read** and **machines to parse**.

Where is JSON Used?

- In **REST APIs** to send/receive data
- In **AJAX** calls for dynamic web pages
- In **configuration files** (`package.json`, `appsettings.json`)

- In **database storage** (MongoDB uses JSON-like documents) **What is a**

Web Service?

A **web service** is a **software system** designed to **communicate with other systems** over a **network** (usually the internet) using **standard web protocols** like **HTTP**.
A **web service** is like an **online function** that one application can call over the internet to **send or receive data** from another application.

Type	Description
SOAP (Simple Object Access Protocol)	Uses XML, very strict structure, heavier and older
REST (Representational State Transfer)	Uses HTTP, JSON/XML, lightweight and commonly used

What is Spring Boot ?

Spring Boot is an **open-source Java-based framework** built on top of the Spring Framework that is designed to **simplify the development of stand-alone, production-grade Spring applications**, including **web applications and RESTful web services**, by providing features like **auto-configuration**, **starter dependencies**, and an **embedded web server** with **minimal setup and configuration**.

Features of Spring Boot

- **Auto-Configuration**

Spring Boot provides **intelligent auto-configuration**, meaning it automatically configures your application based on the libraries you include. For example, if you add `spring-bootstarter-web`, Spring Boot will automatically set up the embedded Tomcat server, Spring MVC, and basic configurations for handling HTTP requests — without needing you to define anything manually.

- **Embedded Web Servers**

Spring Boot comes with **built-in web servers** like Tomcat, Jetty, or Undertow. This means you don't need to deploy your application as a WAR file to an external server. Instead, your application can run as a simple Java program (`java -jar app.jar`) and handle web requests directly. This simplifies deployment and testing.

- **Starter Dependencies**

To simplify dependency management, Spring Boot provides **starter packages**. These are pre-configured collections of commonly used libraries grouped together. For example, `spring-boot-starter-web` includes everything you need to build a web application, like

Spring MVC, Jackson for JSON, and an embedded server. This avoids the hassle of manually adding multiple dependencies.

- **Production-Ready Features**

Spring Boot includes **Actuator**, which gives you built-in endpoints to monitor and manage your application in production. These endpoints can expose health status, metrics, application environment details, and more, helping with diagnostics and operations.

- **Spring Boot CLI**

Spring Boot includes a **Command-Line Interface (CLI)** that lets you run Spring

applications using simple scripts without writing full Java classes. It's useful for quick prototyping or scripting.

- **Spring Initializr**

Spring Boot provides a web tool called **Spring Initializr** (https://start.spring.io), which helps you quickly generate a project with the required dependencies. You can choose your language, dependencies, and project type, and download a ready-to-use template in seconds.

- **No XML Configuration**

Unlike older Spring applications, Spring Boot avoids XML configuration. Instead, it relies on **annotations and Java-based configuration**, which makes the code cleaner and more readable. This reduces complexity and makes your application easier to maintain.

- **Fast Development**

Thanks to auto-configuration, starter dependencies, and embedded servers, Spring Boot allows you to **develop applications faster**. You can focus more on writing business logic instead of handling configuration, setup, or deployment issues.

- **Microservices Support**

Spring Boot is designed to support **microservice architecture**. It's lightweight and modular, which makes it perfect for building independent services that communicate with each other. When combined with **Spring Cloud**, it becomes a powerful toolkit for microservice development.

- **Custom Configuration**

Spring Boot allows you to easily customize application settings using `application.properties` or `application.yml` files. You can change server ports, database connections, logging levels, and more — all through simple key-value pairs, without changing your code.

- **Seamless Integration with Spring Ecosystem**

Spring Boot works perfectly with other Spring projects like **Spring Security**, **Spring Data JPA**, **Spring Cloud**, and more. You can plug them in easily and they'll be auto-configured based on your needs.

- **DevTools for Developer Experience**

Spring Boot provides **DevTools**, a set of developer-focused features like **automatic restarts**, **live reload**, and **debugging support**. These features help you test changes instantly during development without restarting the whole application manually.

Create a REST API Using Spring Boot

Step 1: Create a Spring Boot Project

You can use **Spring Initializr**:

- Go to: https://start.spring.io
- Select:
  - Project: Maven ○ Language: Java ○ Spring Boot Version: (latest stable) ○ Project Metadata:
    - Group: `com.example`
    - Artifact: `demo` □ Add Dependencies:
  - `Spring Web`
- Click **"Generate"** to download the project ZIP
- Extract it and open in your IDE (IntelliJ, Eclipse, or VS Code)

Step 2:Understand the Project Structure src/

```
└── main/
    ├── java/
    │   └── com/example/demo/
    │       └── DemoApplication.java
    └── resources/
        └── application.properties
```

Step 3: Create a REST Controller Class

Create a new class inside `com.example.demo` named `HelloController`.

Use the annotation `@RestController` and define a simple method using `@GetMapping`.

Example: (Conceptual - No code here)

- Mark the class as a REST controller.
- Define a method that handles a GET request (like `/hello`).

- The method returns a plain string or a JSON response.

Step 4: Run the Application

- Run the `DemoApplication` class.
- Spring Boot starts an embedded Tomcat server.
- You'll see something like:

  `Tomcat started on port(s): 8080` **Step**

5: Test the API

- Open your browser or use **Postman**.

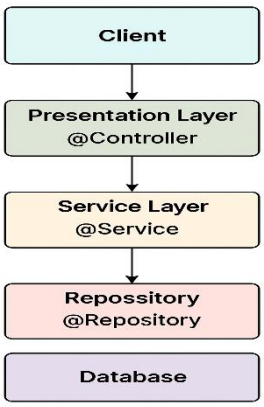
- Access:
  `http://localhost:8080/hello`
- You should see the response returned from your method. **Step 6: Add More API**

  Endpoints You can add more methods using:

- `@PostMapping` – for creating data
- `@PutMapping` – for updating data `@DeleteMapping` – for deleting data
- Use `@RequestParam` for query parameters and `@RequestBody` for JSON input

Step 7: Customize with application.properties (Optional) server.port=9090

Spring Boot Architecture



Client Layer

What It Is:

- External entities that interact with your application.
- This could be a browser, mobile app, Postman, or any other front-end or API client.

Usage:

- Sends HTTP requests (e.g., GET, POST) to the server.
- Receives and displays the response.

Presentation Layer (Controller Layer)

What It Is:

- Entry point for the application logic.
- Handles HTTP requests and maps them to service methods.

Usage:

- Annotated with `@Controller` or `@RestController`.
- Defines endpoints using `@GetMapping`, `@PostMapping`, etc.
- Calls the service layer and returns responses.

Service Layer (Business Logic Layer)

What It Is:

- Contains business logic.
- Acts as a middle layer between the controller and repository.

Usage:

- Annotated with `@Service`.
- Performs validation, calculations, and other operations.
- Calls repository methods to fetch/save data.

Repository Layer (Data Access Layer / DAO)

What It Is:

- Handles data operations (CRUD) on the database.
- Uses Spring Data JPA or JDBC to interact with the DB.

Usage:

- Annotated with `@Repository`.
- Extends interfaces like `JpaRepository`, `CrudRepository`.
- Automatically implements data access logic.

Database Layer

What It Is:

- The actual database (MySQL, PostgreSQL, MongoDB, etc.).

- Stores persistent data (user data, orders, etc.).

Usage:

- Spring Boot uses `application.properties` to configure DB

  spring.datasource.url=jdbc:postgresql://localhost:8080/testdb

  spring.datasource.username=root

  spring.datasource.password=admin

  spring.jpa.hibernate.ddl-auto=update

Layer	Annotation	Responsibility
Client	—	Sends HTTP requests
Controller	`@RestController`	Handles requests, returns responses
Service	`@Service`	Processes business logic
Repository	`@Repository`	Performs CRUD operations
Database	—	Stores and retrieves persistent data

Steps to Create a REST Controller in Spring Boot

Create a POJO Class

```
import java.util.List;

public class Student {
    private int rol;
    private String name;
    private double height;

    //Getter Setter
}
```

Create the REST Controller

```
@RestController
public class StudentController {

    @PostMapping("/mystudent")
    public String recive(@RequestBody Student student) {

        return "Student  is : "+ student;
    }
}
```

Mark the class with @RestContoller

Run Your Application

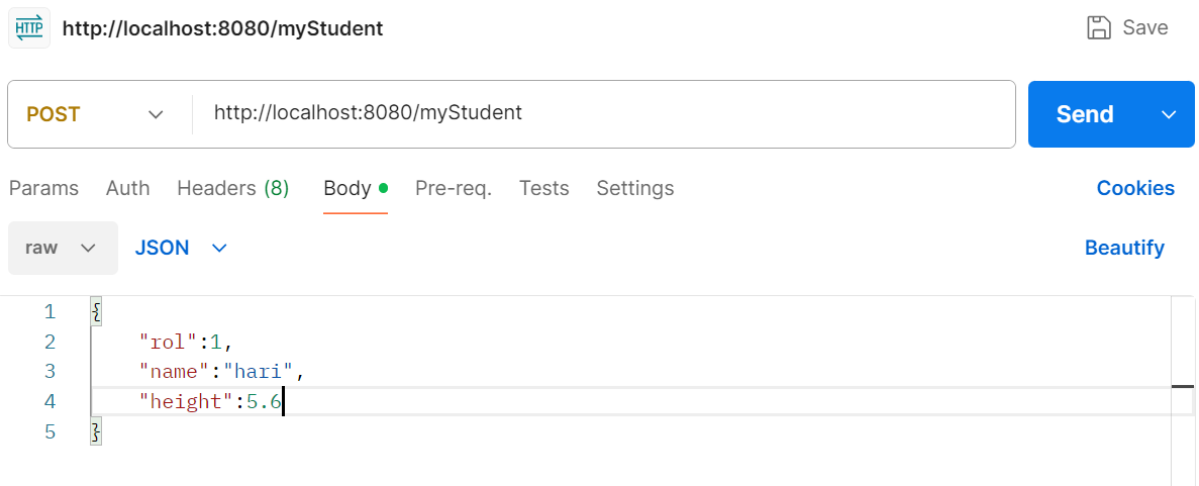
```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

Run the app → It will start on port `8080`.

Open Post man tool and enter the url with the end point and JSON Object



All Mapping Annotations

`@GetMapping`

- **Used for:** Handling `GET` requests (fetching data)

```
@RestController
public class StudentController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }

}
```

URL to test: `GET http://localhost:8080/hello`

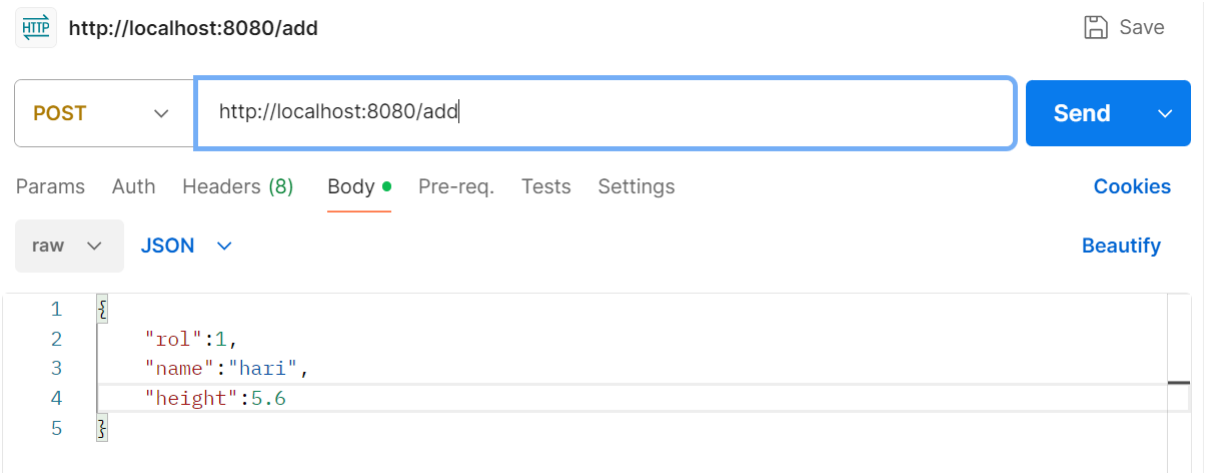
`@PostMapping`

- **Used for:** Handling `POST` requests (creating data)

```
@PostMapping("/add")
public String addStudent(@RequestBody Student s) {
    // Assume user gets added to a list or DB
    return "User " + s + " added successfully";
}
```

URL to test: `POST http://localhost:8080/add`

Send JSON body



`@PutMapping`

- **Used for:** Handling `PUT` requests (updating existing data)

```
@PutMapping("/students/{id}")
public String updateStudent(@PathVariable int id, @RequestBody Student updatedStudent) {
    for (Student student : studentList) {
        if (student.getId() == id) {
            student.setName(updatedStudent.getName());
            student.setEmail(updatedStudent.getEmail());
            return "Student updated";
        }
    }
    return "Student not found";
}
```

`@DeleteMapping`

- **Used for:** Handling `DELETE` requests (deleting data)

```
@DeleteMapping("/students/{id}")
public String deleteStudent(@PathVariable int id) {
    //logic to delete element
    return "";
}
```

URL to test: `DELETE http://localhost:8080/deleteUser/1`

`@RequestMapping` (Generic)

- **Used for:** Mapping any HTTP method; often used for base path or custom mapping.

```
@RequestMapping("/custom")
public String customMapping() {
    return "This is a custom GET mapping using @RequestMapping";
}
```

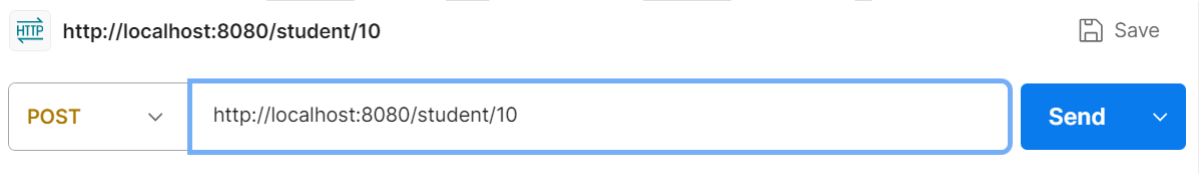
URL to test: `GET http://localhost:8080/custom`

To Read Data from URL Path in Spring Boot

You can read data from the URL using **`@PathVariable`** or **`@RequestParam`**.

@PathVariable

```
@GetMapping("/student/{id}")
    public String getStudentById(@PathVariable int id) {
        return "Student ID received: " + id;
    }
```



@RequestParam

```
@GetMapping("/student")
    public String getStudentByRequestParam(@RequestParam int id) {
        return "Student ID via query param: " + id;
    }
```

Test URL:
`http://localhost:8080/student?id=10`

Method	Used For	Example URL
`@PathVariable`	Path values in URL	`/student/10`
`@RequestParam`	Query parameters in URL	`/student?id=10`

How to **Send an Object** in Spring Boot (From Client → REST Controller)

In Spring Boot, to **send a Java object** from the client (e.g., Postman or frontend) to the backend, you use:

- **`@RequestBody`** in the controller method to map the JSON to a Java object.

```
@PostMapping("/save")
public String saveEmployee(@RequestBody Employee e) {
    return e.getName()+"";
}
```

- `@RequestBody` automatically maps the **JSON request body** to the `Student` object.

  URL:
  `POST` http://localhost:8080/save

  JSON Format

  {

  "id": 101,

  "name": "Hari",

  "email": "hari@example.com"

  }

What	Use
Java object	Create a POJO class
Read object	`@RequestBody`
Send object	JSON body via HTTP

Connecting Spring Boot with PostgreSQL Using Eclipse

how to **store and retrieve data** using **Spring Boot and PostgreSQL**. We will create a REST API for `Student` and connect it to a PostgreSQL database.

Tool	Purpose
Eclipse IDE	Java development
PostgreSQL	Database
Postman	API Testing

Tool	Purpose
Spring Initializr	Project generation

Step 1: Create Database in PostgreSQL

Open **pgAdmin**

A new database named `studentsdb` is created.

Generate Spring Boot Project from Spring Initializr

1. Go to https://start.spring.io
2. Choose:
   - **Project**: Maven
   - **Language**: Java
   - **Group**: `com.example`
   - **Artifact**: `studentdemo`
3. Add dependencies:
   - Spring Web
   - Spring Data JPA
   - PostgreSQL Driver
4. Click **Generate**, then extract and **import in Eclipse**:
   - `File > Import > Existing Maven Projects`

Step 3: Configure Database in `application.properties`

Navigate to:
`src/main/resources/application.properties`

```
spring.datasource.url=jdbc:postgresql://localhost:5432/studentsdb

spring.datasource.username=postgres

spring.datasource.password=your_password



spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

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

Step 4: Create `Student` Entity Class

```java
@Entity
public class Student {
    @Id
    private int id;
    private String name;
    private String email;

    // Getters and setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

Step 5: Create `StudentRepository` Interface

```java
public interface StudentRepository extends JpaRepository<Student, Integer> {

}
```

Step 6: Create REST Controller

```java
@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentRepository studentRepository;

    @PostMapping("/add")
    public String addStudent(@RequestBody Student student) {
        studentRepository.save(student);
        return "Student saved successfully: " + student.getName();
    }

    @GetMapping
    public List<Student> getAllStudents() {
        return studentRepository.findAll();
    }
}
```

Step 7: Run the Application

```
@SpringBootApplication
public class StudentdemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(StudentdemoApplication.class, args);
    }
}
```

Step 8: Test API in Postman

To Save a Student:

- **URL**: `POST http://localhost:8080/students/add`

```
{
  "id": 1,
  "name": "Hari",
  "email": "hari@example.com"
}
```

To Get All Students:

- **URL**: `GET http://localhost:8080/students`

Component	Purpose
`@Entity`	Maps class to DB table
`JpaRepository`	Provides CRUD operations
`@RestController`	Exposes REST endpoints
PostgreSQL + Hibernate	Manages DB and ORM
`@RequestBody`	Accepts object from client

What is the Use of `JpaRepository` in Spring Boot?

`JpaRepository` is an interface provided by **Spring Data JPA** that allows you to perform **database operations** without writing any SQL or HQL queries manually.

To **simplify CRUD operations** (Create, Read, Update, Delete) and provide powerful **built-in query methods** for working with database entities.

How It Works

When you create a repository interface like this:

```
public interface StudentRepository extends JpaRepository<Student, Integer> {

}
```

- `Student` → Your entity class (mapped to DB table)
- `Integer` → Type of the primary key (`@Id` field)

Spring automatically creates a **proxy implementation** of this interface at runtime.

1. `save(entity)`

Used to **insert** a new record or **update** an existing record in the database.

```
studentRepository.save(student);
```

---

2. `saveAll(listOfEntities)`

Used to save **multiple entities** in a single call.

```
studentRepository.saveAll(List.of(student1, student2));
```

---

3. `findById(id)`

Fetches a **single record** using its primary key.

```
Optional<Student> student = studentRepository.findById(1);
```

---

4. `findAll()`

Returns **all records** from the table.

```
List<Student> allStudents = studentRepository.findAll();
```

---

5. `deleteById(id)`

Deletes a record using its primary key.

```
studentRepository.deleteById(1);
```

---

6. `delete(entity)`

Deletes a specific entity object.

```
studentRepository.delete(student);
```

---

7. `deleteAll()`

Deletes **all records** in the table.

```
studentRepository.deleteAll();
```

---

8. `existsById(id)`

Checks if a record **exists** by its ID.

```
boolean exists = studentRepository.existsById(1);
```

---

9. `count()`

Returns the **total number of records** in the table.

```
long total = studentRepository.count();
```

Why Do We Create an Interface That Extends `JpaRepository` in Spring Boot?

To allow Spring Data JPA to **automatically generate the code** for all common database operations like `save()`, `findAll()`, `deleteById()`, etc., **without writing any implementation code**.

Why Interface and Not a Class?

- Because **Spring handles the implementation** at runtime.
- You just **declare the methods**, and Spring will generate the body.

- If you use a class, **you would have to manually implement all methods** (like `findAll`, `save`, etc

  Spring Boot, using **Spring Data JPA**, provides the implementation at runtime.

Benefits of Using Interface for `JpaRepository`

1. **No boilerplate code**: No need to write SQL or DAO code manually.
2. **Built-in methods**: Get access to `save()`, `findById()`, `findAll()`, etc.
3. **Custom methods**: You can define custom finders like `findByName()` and Spring will still generate them.
4. **Cleaner architecture**: Keeps your code modular and easy to manage.

How to Write Custom Queries in JPA Repository (Spring Data JPA)

In Spring Boot with Spring Data JPA, you can write custom queries in your `JpaRepository` using:

1. **Derived Query Methods** (No query string needed)

   You simply **name the method** following a pattern, and Spring will create the query for you

```
public interface StudentRepository extends JpaRepository<Student, Integer> {

        List<Student> findByName(String name);

        Student findByEmail(String email);

        void deleteByName(String name);
}
```

   Custom Method

Rules to Follow

- Method name must start with `findBy`, `readBy`, `getBy`
- Use **camelCase** that matches **entity field names**
- Combine field names using `And`, `Or`, etc.
- Use operators like `Containing`, `GreaterThan`, `IsNull`, etc

```java
public interface StudentRepository extends JpaRepository<Student, Integer> {

    Student findByEmail(String email);

    Student findByNameAndEmail(String name, String email);

    Student findByNameOrEmail(String name, String email);

    List<Student> findByIdBetween(int start, int end);

    List<Student> findByAgeGreaterThan(int age);

    List<Student> findByIdLessThan(int id);

    List<Student> findByEmailIsNull();

    List<Student> findByEmailIsNotNull();

    List<Student> findByNameContaining(String keyword); // LIKE %keyword%

    List<Student> findByNameStartingWith(String prefix); // LIKE prefix%

    List<Student> findByNameEndingWith(String suffix); // LIKE %suffix

    List<Student> findByNameIgnoreCase(String name);

    Student findTop1ByOrderByIdDesc();

    List<Student> findFirst3ByName(String name);

}
```

Custom Method Naming Tips

1. Always start with `findBy`, `getBy`, or `readBy`.
2. Match exact field names in your entity.
3. Combine conditions using `And`, `Or`.
4. Use `Containing`, `Between`, `LessThan`, etc., for special queries.
5. No need to write SQL – Spring builds the query for you.

How to Write a Custom Query in `JpaRepository` (Spring Data JPA)

Spring Data JPA allows you to define **custom queries** using the `@Query` annotation inside your repository interface.

You can write:

- JPQL (Java Persistence Query Language)
- Native SQL
- Modifying queries (for `UPDATE` or `DELETE`)

1. Custom Query Using JPQL and sql

JPQL works on **entity and field names**, not table or column names.

```java
public interface StudentRepository extends JpaRepository<Student, Integer> {

    /*
     * 1. Custom Query Using JPQL JPQL works on entity and field names, not table or
     * column names. JPQL uses entity class Student and its field email. The query
     * will return a Student object with the given email.
     */

    @Query("SELECT s FROM Student s WHERE s.email = :email")
    Student findByEmail(@Param("email") String email);

    /*
     * 2. Custom Query Using Native SQL If you want to use raw SQL, set nativeQuery
     * = true.
     *
     * Uses the actual table name student. ✔ ILIKE makes it case-insensitive
     * (PostgreSQL only).
     */

    @Query(value = "SELECT * FROM student WHERE name ILIKE %:name%", nativeQuery = true)
    List<Student> searchByName(@Param("name") String name);


    @Query("SELECT s FROM Student s WHERE s.name = :name")
    List<Student> findByName(@Param("name") String name);

    // Native Query
    @Query(value = "SELECT * FROM student WHERE email = :email", nativeQuery = true)
    Student getStudentByEmail(@Param("email") String email);

    // Update Query
    @Modifying
    @Transactional
    @Query("UPDATE Student s SET s.name = :name WHERE s.id = :id")
    int updateStudentName(@Param("id") int id, @Param("name") String name);

}
```

- Use `@Query` for **custom SELECT queries**.
- Use `@Modifying` + `@Transactional` for **update/delete**.
- Use `:param` and `@Param` to bind values.
- Choose between JPQL (entity-based) and native SQL (table-based).

What is Pagination in Spring Boot?

Pagination is the process of **dividing large amounts of data into smaller, manageable chunks called pages**. Instead of loading all records at once, we load a specific number of records per request (e.g., 10 at a time).

- Improves **performance** by loading fewer records at once.
- Reduces **memory usage**.
- Makes it easier to **navigate large datasets** (like search results or product listings).
- Avoids long response times.

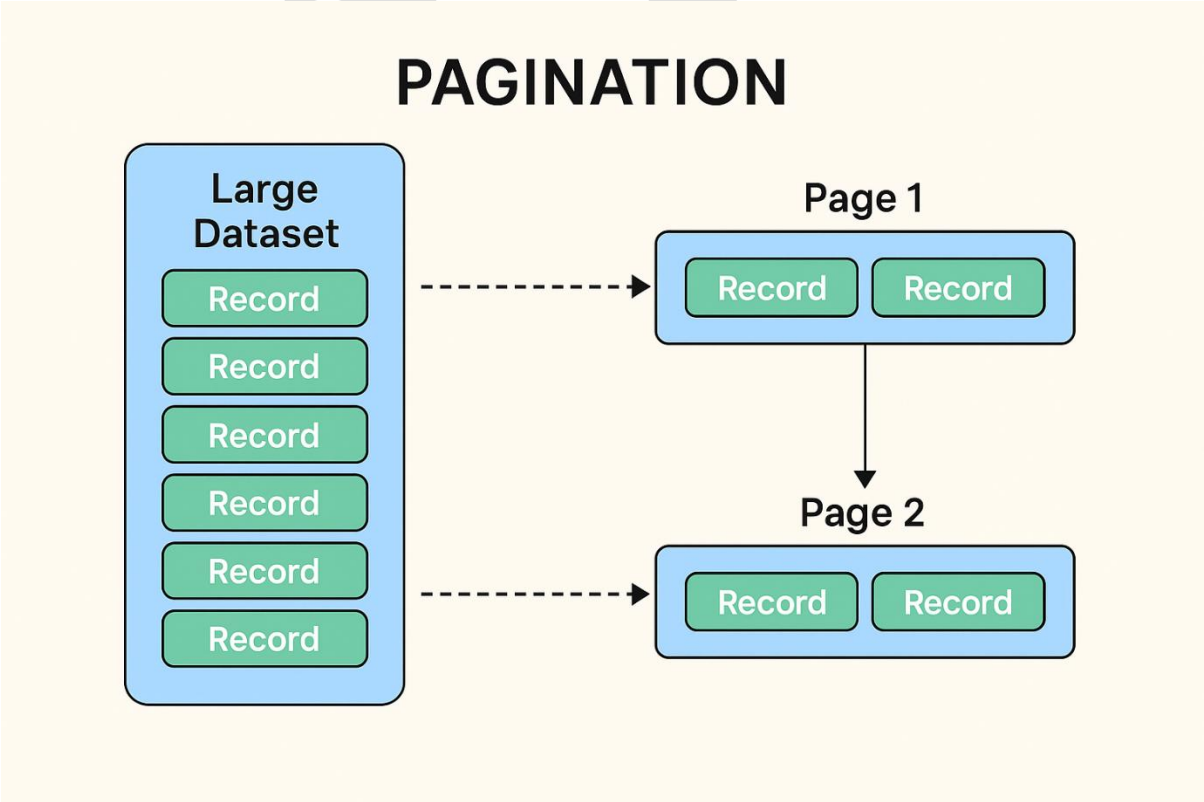
Imagine you have 10,000 students in your database.

Without Pagination:

- All 10,000 are fetched at once → slow, memory-heavy

With Pagination:

- First request loads only 10 students
- Next page loads the next 10, and so on...



In Spring Boot with JPA

Spring provides built-in support using:

- `Pageable` – tells which page and how many records per page.

- `Page<T>` – returns the data along with page info like total pages, current page, etc

```
@GetMapping("/page")
public Page<Student> getStudentsWithPagination(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "5") int size) {
    PageRequest pageable = PageRequest.of(page, size);
    return studentRepository.findAll(pageable);
}
```

`PageRequest.of(page, size)`:

- A static method that returns a `PageRequest` object.
- It specifies which page you want and how many records should be on that page.

- **`page`**:
  - The **page number** you want to retrieve.
  - It is **zero-based** — so:
    - Page 0 = 1st page
    - Page 1 = 2nd page
    - and so on...

- **`size`**:
  - The **number of records** per page.
  - Example: if `size = 5`, it means "5 students per page".

- **`Pageable`**:
  - An **interface** in Spring Data that represents pagination information (page number, size, and sorting).
  - `PageRequest` is an implementation of `Pageable`.

```
@GetMapping("/paged")
public List<Student> getStudentsWithPaginationAndSorting(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "5") int size,
        @RequestParam(defaultValue = "name") String sortBy
) {
    Pageable pageable = PageRequest.of(page, size, Sort.by(sortBy).ascending());
    Page<Student> studentPage = studentRepository.findAll(pageable);
    return studentPage.getContent();
}
```

http://localhost:8080/students/paged?page=0&size=5&sortBy=name

What is Response Structure in Spring Boot?

The **response structure** in Spring Boot refers to the **format of data returned** to the client (browser, frontend, Postman, etc.) when an API endpoint is called.

Spring Boot uses **Jackson (JSON parser)** by default to automatically **convert Java objects into JSON** in the HTTP response.

Step 1: Create a Spring Boot Project in Eclipse

Dependencies:

- Spring Web
- Spring Data JPA
- PostgreSQL Driver (if connecting DB, but optional here)

```java
@Entity
public class Student {
    @Id
    private int id;
    private String name;
    private String email;

    // Getters and setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

Create a Custom Response Wrapper Class

```java
public class ApiResponse<T> {

        private String status;
         private String message;
         private T data;

         public ApiResponse(String status, String message, T data) {
             super();
             this.status = status;
             this.message = message;
             this.data = data;
         }
         public String getStatus() {
             return status;
         }
         public void setStatus(String status) {
             this.status = status;
         }
         public String getMessage() {
             return message;
         }
         public void setMessage(String message) {
             this.message = message;
         }
         public T getData() {
             return data;
         }
         public void setData(T data) {
             this.data = data;
         }


 }
```

```java
@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentRepository studentRepository;

    @GetMapping("/{id}")
    public ApiResponse<Student> getStudentById(@PathVariable int id) {
        Optional<Student> student = studentRepository.findById(id);

        if (student.isPresent()) {
            return new ApiResponse<Student>("success", "Student found", student.get());
        } else {
            return new ApiResponse<Student>("error", "Student with ID " + id + " not found", null);
        }
    }

}
```

http://localhost:8080/students/1

- You now return a clean and uniform JSON response even for errors.
- This makes your APIs **frontend-friendly** and **standardized**

What is `ResponseEntity` in Spring Boot?

`ResponseEntity` is a **class in Spring** used to **customize HTTP responses** completely — including the **body**, **status code**, and **headers**.

When to Use `ResponseEntity`

Use it when:

- You need to **set status code explicitly** (like 201, 404, 500)
- You want to return **custom messages**
- You want to **return or manipulate headers**
- You want to build a **consistent API structure**

```java
@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentRepository studentRepository;

    @GetMapping("/{id}")
    public ResponseEntity<ApiResponse<Student>> getStudentById(@PathVariable int id) {
        Optional<Student> student = studentRepository.findById(id);

        if (student.isPresent()) {
            ApiResponse<Student> response = new ApiResponse<>(
                "success",
                "Student found",
                student.get()
            );
            return new ResponseEntity<>(response, HttpStatus.OK);
        } else {
            ApiResponse<Student> response = new ApiResponse<>(
                "error",
                "Student with ID " + id + " not found",
                null
            );
            return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
        }
    }
}
```

200 OK
This means the request was successful. It is used for most successful `GET`, `PUT`, and `DELETE` requests.
Example: When a student is found or updated.

201 Created
This means a new resource was created successfully.
Example: When a new student is saved using a `POST` request.

204 No Content
This means the operation was successful but there's nothing to return in the response body.
Example: When a student is deleted and you don't want to return any message

400 Bad Request
This means the client sent an invalid or malformed request.
Example: Required fields like `name` or `email` are missing in the request body.

401 Unauthorized
This means the client is not authenticated.
Example: You tried accessing a secured API without providing a token or login credentials.

403 Forbidden
This means you're authenticated but you're not allowed to access this resource.
Example: A regular user tries to access an admin-only API.

404 Not Found
This means the requested resource does not exist.
Example: You try to find a student with ID `100`, but no such student exists.

500 Internal Server Error
This means something went wrong on the server — typically an unhandled exception in the code.
Example: A null pointer exception or database connection failure.

502 Bad Gateway
This means the server received an invalid response from another server it's trying to communicate with.

503 Service Unavailable
This means the server is temporarily down or under maintenance.

Custom exception in Spring Boot

Creating and handling **custom exceptions** in Spring Boot is very important for clean, readable, and user-friendly API responses.

Step 1: Create a Custom Exception Class

Create a Global Exception Handler Class

This will **catch** your custom exception and return a proper structured response.

```java
@ControllerAdvice
public class MyAppExceptionController {

//  public ResponseEntity<ResponseStructure<String>> handleNullPointerException(NullPointerException nullPointerException){
//
//
//  }
//  @ExceptionHandler(NullPointerException.class)
//  public void handleNullPointer() {
//      System.out.println("-------Hi i am called----------------");
//  }
    @ExceptionHandler(NullPointerException.class)
    public ResponseEntity<ResponseStructure<String>> handleNullPointerException(NullPointerException npe){
        ResponseStructure<String> rs = new ResponseStructure<String>();
        rs.setStatusCode(HttpStatus.BAD_REQUEST.value());
        rs.setMessage("Message : "+npe.getMessage());
        rs.setData("Dont deal with null");
        return new ResponseEntity<ResponseStructure<String>>(rs,HttpStatus.BAD_REQUEST);

    }
}
```

1. **Create custom exception** (`StudentNotFoundException`)
2. **Throw it** inside your controller/service
3. **Handle it** using `@ControllerAdvice` and `@ExceptionHandler`
4. Return **clean, user-friendly, structured** error responses