Difference between RestController and Controller?

Great question! Both @RestController and @Controller are used in Spring MVC, but they serve **different purposes**, especially in how they return data.

**✅ Quick Summary**

| **Feature** | **@RestController** | **@Controller** |
| --- | --- | --- |
| Purpose | REST APIs (JSON/XML responses) | Web pages (HTML + Thymeleaf/JSP) |
| Returns | JSON, XML, or any serialized object | View name (like a JSP or Thymeleaf page) |
| Combines | @Controller + @ResponseBody | Just @Controller |
| Common use case | APIs for mobile, frontend apps, microservices | MVC apps with UI rendering |

## 1. Change Maven Local Repository via settings.xml (Recommended)

1. Open your Maven settings.xml file:
   * For most systems:  
     ~/.m2/settings.xml (Linux/macOS)  
     C:\Users\<your-user>\.m2\settings.xml (Windows)
2. Add or modify the <localRepository> tag:

xml

CopyEdit

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0

http://maven.apache.org/xsd/settings-1.0.0.xsd">

<!-- Set custom local repo location -->

<localRepository>D:/custom-m2-repo</localRepository>

</settings>

🔄 After this, Maven will start downloading and storing dependencies in D:/custom-m2-repo instead of the default .m2/repository.

## All CascadeType Options

| **CascadeType** | **Description** |
| --- | --- |
| PERSIST | When the parent is persisted, the child is also persisted |
| MERGE | When the parent is merged (updated), the child is also merged |
| REMOVE | When the parent is deleted, the child is also deleted |
| REFRESH | Refreshes child entities when the parent is refreshed from the database |
| DETACH | Detaches child entities when the parent is detached from the persistence context |
| ALL | Shortcut for applying all above types |

………………………………………………………………………………………………………………………………………………………..

## 🧱 1. One-to-Many and Many-to-One

### 🔹 School.java (One-to-Many)

@Entity

public class School {

@Id @GeneratedValue

private Long id;

private String name;

@OneToMany(mappedBy = "school", cascade = CascadeType.ALL)

private List<Student> students = new ArrayList<>();

// Getters, setters

}

### 🔹 Student.java (Many-to-One)

@Entity

public class Student {

@Id @GeneratedValue

private Long id;

private String name;

@ManyToOne

@JoinColumn(name = "school\_id") // ← creates school\_id FK in student table

private School school;

// Getters, setters

}

| **Concept** | **Explanation** |
| --- | --- |
| @JoinColumn | Tells JPA where the **foreign key column** is stored |
| mappedBy | Tells JPA the **owning side is the other entity** |
| school\_id column | Created in the student table to link to school.id |
| Not in School.java | Because OneToMany doesn't hold the foreign key itself |

## 🧱 3. Many-to-Many

### 🔹 Student.java

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id")

)

private List<Course> courses = new ArrayList<>();

### 🔹 Course.java

@Entity

public class Course {

@Id @GeneratedValue

private Long id;

private String title;

@ManyToMany(mappedBy = "courses")

private List<Student> students = new ArrayList<>();

}

## 2. One-to-One

### 🔹 Student.java (add address)

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "address\_id")

private Address address;

### 🔹 Address.java

@Entity

public class Address {

@Id @GeneratedValue

private Long id;

private String city;

private String postalCode;

// Optional reverse mapping:

@OneToOne(mappedBy = "address")

private Student student;

}

…………………………………………………………………………..

@Query("SELECT s FROM Student s WHERE s.department = ?1")

List<Student> findByDepartment(String department);

SpringBoot request flow?

## Where It Fits in the Flow:

Here's a simplified flow of how Spring handles a request:

1. **Request → DispatcherServlet**
2. **DispatcherServlet → HandlerMapping** → Finds the correct controller method
3. **DispatcherServlet → HandlerAdapter** → Calls the method
4. **Return value → ViewResolver or HttpMessageConverter**
5. **Response sent to client**

…………………………………………………………………………………………………………………………

wh we need to use DTOS. why dont we directly send entity only?

###### ChatGPT said:

Great question! You **can** send entities directly from your Spring Boot backend—but using **DTOs (Data Transfer Objects)** is considered **a better, safer, and cleaner practice** in most real-world applications.

Let’s break it down:

## ✅ What is a DTO?

A **DTO (Data Transfer Object)** is a plain Java object (POJO) that **carries data between layers**, such as from:

* Controller → Client (in API response)
* Client → Controller (in request body)

Example:

public class UserDTO {

private String name;

private String email;

}

## ❌ Why Not Send Entities Directly?

### 1. ****Entities expose sensitive/internal data****

Entities often contain fields you **don’t want to send to the client**:

java

CopyEdit

@Entity

public class User {

private Long id;

private String name;

private String email;

private String password; // ⚠️ should NOT be sent in API response

}

If you send this User entity directly, you'll accidentally leak the password.

### 2. ****Tightly couples API with database structure****

Entities represent your **database schema**. If you send them directly:

* A small DB change (e.g., renaming a column) **breaks your frontend**
* You lose flexibility to evolve the database separately from the API

DTOs **decouple** the internal model from the external contract.

### 3. ****You may want to send partial/custom data****

You often want to send only a subset or a combination of fields:

java

CopyEdit

// DTO combining fields from multiple entities

public class OrderSummaryDTO {

private Long orderId;

private String customerName;

private String productTitle;

}

This avoids exposing full Order, Customer, and Product entities.

### 4. ****Entities may have relationships that cause infinite recursion****

For example, with bidirectional mappings:

java

CopyEdit

class Department {

List<Employee> employees;

}

class Employee {

Department department;

}

Serializing this to JSON directly can cause **stack overflow** or **circular reference errors**.

DTOs avoid this by **breaking the loop**.

### 5. ****DTOs make validation cleaner****

You can add input validation directly to a request DTO:

java

CopyEdit

public class CreateUserRequest {

@NotBlank private String name;

@Email private String email;

}

This is cleaner than adding validation to your database entity.

## ✅ Summary: When to Use DTOs

| **Reason** | **Description** |
| --- | --- |
| ✅ Security | Prevent leaking sensitive fields (e.g., password, tokens) |
| ✅ Flexibility | Allows API to evolve separately from DB |
| ✅ Custom Response | Create lightweight or combined views |
| ✅ Validation | Use @Valid with DTOs for input checks |
| ✅ Avoid recursion | No more circular relationship issues |

## 🚫 When It's (Maybe) Okay to Send Entities

In small, internal-only apps or prototypes—when:

* There's **no sensitive data**
* You're not exposing the API publicly
* You control both backend & frontend
* You're okay with tight coupling

Even then, DTOs add clarity and are worth the minimal extra effort.

………………………………………………………………………………….

Bean scopes

scopes of spring beans

**ChatGPT said:**

In **Spring Framework**, **bean scope** defines **how and when** the Spring container creates new bean instances. There are **five main scopes** (two for web environments and three standard ones). Here's a complete breakdown:

**✅ 1. Singleton (Default)**

* **Only one instance per Spring container**
* Created at container startup (eager by default)
* Shared across the application

java

CopyEdit

@Bean

@Scope("singleton") // or omit since it's the default

public MyBean myBean() {

return new MyBean();

}

📌 Use Case: Stateless services, config classes

**✅ 2. Prototype**

* **New instance created every time** the bean is requested
* Spring **doesn't manage** complete lifecycle (no destroy method)

java

CopyEdit

@Bean

@Scope("prototype")

public MyBean myBean() {

return new MyBean();

}

📌 Use Case: Stateful beans, like form backing objects

**✅ 3. Request *(Web only)***

* One bean instance **per HTTP request**
* Scoped to a single HTTP request

java

CopyEdit

@Component

@Scope(value = WebApplicationContext.SCOPE\_REQUEST, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class RequestScopedBean {

// ...

}

📌 Use Case: Per-request data (e.g. user info)

**✅ 4. Session *(Web only)***

* One bean instance **per HTTP session**
* Lives until session expires

java

CopyEdit

@Component

@Scope(value = WebApplicationContext.SCOPE\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class SessionScopedBean {

// ...

}

📌 Use Case: Shopping cart, user session storage

**✅ 5. Application**

* One bean instance **per ServletContext**
* Shared across all sessions and requests

java

CopyEdit

@Component

@Scope(value = WebApplicationContext.SCOPE\_APPLICATION)

public class AppScopedBean {

// ...

}

📌 Use Case: App-wide caches or configurations

**🧪 Extra: Thread Scope (Custom Scope)**

* Not provided out-of-the-box
* You can define a bean per-thread (e.g., via Spring’s SimpleThreadScope)

**🔍 Summary Table**

| **Scope** | **Bean Lifecycle** | **Web Only** | **Typical Use** |
| --- | --- | --- | --- |
| Singleton | 1 instance per container | ❌ | Stateless services |
| Prototype | New instance per request | ❌ | Stateful beans |
| Request | 1 instance per HTTP request | ✅ | Request-specific logic |
| Session | 1 instance per session | ✅ | Session data (e.g. cart) |
| Application | 1 instance per ServletContext | ✅ | Application-wide configs |

…………………………………………………………………………………………………..

## ✅ Spring Bean Lifecycle Phases

### ****1. Instantiation****

* Spring creates the **object** of the bean using its constructor or factory method.
* No dependencies are injected yet.

java

CopyEdit

public MyBean() {

System.out.println("Bean Constructor called");

}

### ****2. Populate Properties (Dependency Injection)****

* Spring performs **DI (Dependency Injection)** to set the required properties.

### ****3. BeanNameAware (Optional)****

* If bean implements BeanNameAware, Spring calls setBeanName(String name).

java

CopyEdit

public class MyBean implements BeanNameAware {

public void setBeanName(String name) {

System.out.println("Bean name set: " + name);

}

}

### ****4. BeanFactoryAware / ApplicationContextAware (Optional)****

* Spring injects the container reference using:
  + setBeanFactory(BeanFactory) – BeanFactoryAware
  + setApplicationContext(ApplicationContext) – ApplicationContextAware

### ****5. Pre-Initialization (BeanPostProcessor before init)****

* Spring calls all BeanPostProcessor.postProcessBeforeInitialization() methods.

### ****6. InitializingBean / Custom init method****

* If the bean implements InitializingBean, Spring calls:
  + afterPropertiesSet()
* OR if defined via @PostConstruct or XML init-method, it gets called here.

java

CopyEdit

@PostConstruct

public void init() {

System.out.println("Custom init method called");

}

### ****7. Post-Initialization (BeanPostProcessor after init)****

* Spring calls all BeanPostProcessor.postProcessAfterInitialization() methods.

### ****8. Ready to Use****

* Now the bean is ready for use in the application.

### ****9. Destroy / Shutdown****

When the Spring context is closing:

* If the bean implements DisposableBean → destroy() is called
* OR custom destroy method (e.g. via @PreDestroy or XML destroy-method)

java

CopyEdit

@PreDestroy

public void cleanup() {

System.out.println("Custom destroy method called");

}

## 🔁 Summary Flow Diagram

less

CopyEdit

Constructor ➝ Dependency Injection ➝ Aware Interfaces ➝

BeanPostProcessor (before) ➝ Initialization (@PostConstruct / afterPropertiesSet) ➝

BeanPostProcessor (after) ➝ Ready to Use ➝

Destroy (@PreDestroy / destroy())

…………………………………………………………………….

1. **What is @SpringBootApplication?**
   * A combination of:
     + @Configuration
     + @EnableAutoConfiguration
     + @ComponentScan

……………………………………………………………………………………………….

**Spring boot security**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/public").permitAll()

.anyRequest().authenticated()

)

.httpBasic(withDefaults()); // Enables basic auth

return http.build();

}

}

………………………………………………………………………………..

### 11. ****How to write a custom JPQL query?****

@Query("SELECT e FROM Employee e WHERE e.name = :name")

List<Employee> findByName(@Param("name") String name);

……………………………………………………………………………….

@GetMapping

public Page<Employee> getEmployees(@RequestParam(defaultValue = "0") int page,

@RequestParam(defaultValue = "5") int size) {

Pageable pageable = PageRequest.of(page, size);

return employeeRepository.findAll(pageable) }

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeRepository employeeRepository;

// Endpoint: /employees?page=0&size=5&sort=name,desc

@GetMapping

public Page<Employee> getEmployees(@RequestParam(defaultValue = "0") int page,

@RequestParam(defaultValue = "5") int size,

@RequestParam(defaultValue = "id,asc") String[] sort) {

// Parse sort parameters

String sortBy = sort[0];

String direction = sort.length > 1 ? sort[1] : "asc";

Sort sortOrder = direction.equalsIgnoreCase("desc") ?

Sort.by(sortBy).descending() :

Sort.by(sortBy).ascending();

Pageable pageable = PageRequest.of(page, size, sortOrder);

return employeeRepository.findAll(pageable);

}

}

When to use @Pathvariable and @RequestParam