**Spring Boot Interview Questions**

Contents

[1. Is Spring Boot a library or a framework? What is the difference between the two? 1](#_Toc1415827861)

[Difference Between Library and Framework 1](#_Toc1424018095)

[Inversion of Control 1](#_Toc1914342570)

[Summary 2](#_Toc315264068)

[2. How do Api requests map to the Controller? 2](#_Toc159561740)

[Lifecycle Overview 2](#_Toc1514858177)

[Key Annotations Used: 2](#_Toc1352753986)

[3. Give a description of each layer in a Spring Boot RESTful Web Services application. 3](#_Toc624103871)

[1. Controller Layer (API layer) 3](#_Toc1853256004)

[2. Service Layer (Business Logic) 3](#_Toc1465355154)

[3. Repository Layer (Data Access) 4](#_Toc1388508761)

[4. Model Layer (Entity/DTO) 4](#_Toc506468807)

[4. Explain Spring Boot's REST mappings — @PostMapping, @GetMapping, and @PutMapping 5](#_Toc1544226653)

# Is Spring Boot a library or a framework? What is the difference between the two?

**Spring Boot is a *framework***.

It is an extension of the Spring framework that simplifies the setup and development of Spring applications by offering:

* Auto-configuration
* Embedded server support
* Production-ready features (Actuator, external config)
* Minimal boilerplate

## **Difference Between Library and Framework**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Library** | **Framework** |
| **Control Flow** | You call the library code | Framework calls your code (Inversion of Control) |
| **Usage** | Use when and where needed | Must follow the framework’s structure |
| **Flexibility** | More flexible, less opinionated | More structured and opinionated |
| **Learning Curve** | Generally easier | Steeper due to conventions and life cycle |
| **Examples** | Jackson, Apache Commons, JUnit | Spring Boot, Angular, Django, React |

## **Inversion of Control**

**Library**:  
 You are in charge. You decide when and how to use it.

List<String> names = new ArrayList<>();  
Collections.sort(names); // You control the flow

**Framework**:  
 It is in charge. It calls your components based on annotations and lifecycle.

@RestController  
public class HelloController {  
 @GetMapping("/hello")  
 public String sayHello() {  
 return "Hi!";  
 }  
}  
// Spring Boot handles routing and lifecycle

## **Summary**

**Spring Boot is a framework** because it controls the application flow, provides a structured lifecycle, and calls your components (controllers, services, etc.) via annotations and IoC.  
 A **library** is something you call manually when you need it.

# How do Api requests map to the Controller?

### **Lifecycle Overview**

1. **Client** sends an HTTP request (e.g. GET /employees).
2. **Spring Boot** uses **DispatcherServlet** (front controller) to intercept the request.
3. It looks up the correct **@RestController** and **@RequestMapping** or **@GetMapping**, etc.
4. The corresponding method is invoked with parsed parameters.
5. Response (usually JSON) is returned via **HttpMessageConverter**.

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@GetMapping("/{id}")

public Employee getEmployee(@PathVariable Long id) {

return employeeService.getEmployeeById(id);

}

}

**URL:** GET /employees/10  
**Mapped to:** getEmployee(10)

### **Key Annotations Used:**

|  |  |
| --- | --- |
| **Annotation** | **Purpose** |
| @RestController | Declares the class as a REST API |
| @RequestMapping | Base URL for all methods in class |
| @GetMapping | Handles GET requests |
| @PostMapping | Handles POST requests |
| @PathVariable | Extracts value from URL path |
| @RequestParam | Extracts query parameter |
| @RequestBody | Binds request JSON to Java object |

# Give a description of each layer in a Spring Boot RESTful Web Services application.

### **1. Controller Layer (API layer)**

* Receives HTTP requests.
* Maps to endpoints using annotations.
* Passes control to service layer.

@RestController

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping("/employees/{id}")

public Employee getById(@PathVariable Long id) {

return employeeService.getEmployeeById(id);

}

}

### **2. Service Layer (Business Logic)**

* Contains core business rules.
* Orchestrates calls to repository, other services, validation, etc.

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public Employee getEmployeeById(Long id) {

return employeeRepository.findById(id).orElseThrow();

}

}

### **3. Repository Layer (Data Access)**

* Talks to the database using **Spring Data JPA** or similar.
* No SQL needed for basic queries — Spring auto-implements it.

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

### **4. Model Layer (Entity/DTO)**

* Java objects (POJOs) representing DB tables or data structure.

@Entity

public class Employee {

@Id

private Long id;

private String name;

}

**Summary Flow (End-to-End)**

**[Client]**

**↓**

**[DispatcherServlet]**

**↓**

**[Controller] → [Service] → [Repository] → [Database]**

**↑**

**[Entity]**

**Typical Package Structure**

com.example.myapp

│

├── controller // REST APIs

├── service // Business logic

├── repository // DB access

├── entity/model // Entity classes

└── dto // (optional) Request/Response models

# Explain Spring Boot's REST mappings — @PostMapping, @GetMapping, and @PutMapping

In Spring Boot, we use specific annotations to map HTTP requests to controller methods. These include @PostMapping for creating data, @GetMapping for retrieving data, and @PutMapping for updating data.

### **Example: Employee API**

Let’s say I have a REST controller for managing employees. I’ll break it into three parts:

#### **@PostMapping – Create an Employee**

I use @PostMapping to map a POST request to a method that creates a new employee.  
 The method takes an @RequestBody object, saves it using the repository, and returns the created entity.

@PostMapping  
public Employee createEmployee(@RequestBody Employee emp)

This takes the employee JSON from the request body and stores it in the database.

#### **@GetMapping – Fetch an Employee by ID**

For reading data, I use @GetMapping("/{id}").  
 It takes a path variable, looks up the employee by ID, and returns it if found.

@GetMapping("/{id}")  
public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id)

This safely returns the employee or a 404 if not found.

#### **@PutMapping – Update an Existing Employee**

To update an employee, I use @PutMapping("/{id}").  
 The method fetches the existing record, updates its fields with new data from the request body, and saves it back.

@PutMapping("/{id}")  
public ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody Employee updated)

This lets me replace or update the employee’s name, role, or other fields.

I also wrap responses in ResponseEntity<> to handle 200 OK or 404 Not Found status codes, which is good REST practice.

### **Summary**

So, in short, @PostMapping is for create, @GetMapping is for read, and @PutMapping is for update.  
 Each maps directly to its respective HTTP method and helps build clean, RESTful APIs in Spring Boot.