**Link:** https://medium.com/@gaddamnaveen192/spring-boot-interview-prep-300-questions-and-real-world-challenges-explained-c1f5ab7bb4b1

# Basic Spring Boot Interview Questions:

Here are the answers in simple words with multiple examples:

**Basic Spring Boot Questions**

**1. What is Spring Boot, and how does it differ from the Spring framework?**

Spring Boot is a framework that makes it easier to build Spring applications with minimal configuration. It provides built-in tools and dependencies to speed up development.

**Difference:**

* **Spring Framework** requires manual setup (like configuring beans, XML, etc.).
* **Spring Boot** reduces this work by offering auto-configuration and an embedded server.

**Example:**  
With **Spring Framework**, you must configure a web server like Tomcat manually. In **Spring Boot**, it comes built-in.

**2. What are the advantages of using Spring Boot?**

Spring Boot offers many benefits:

* **Auto-configuration:** No need to configure everything manually.
* **Embedded server:** Runs without needing a separate web server.
* **Less boilerplate code:** Requires less coding effort.
* **Microservices-friendly:** Ideal for microservices applications.
* **Production-ready:** Includes monitoring tools like Actuator.

**Example:**  
Instead of writing complex configurations for database connections, Spring Boot allows you to simply add properties in application.properties.

**3. What is the purpose of the @SpringBootApplication annotation?**

@SpringBootApplication is a combination of three annotations:

* @Configuration (marks the class for configuration)
* @EnableAutoConfiguration (enables auto-configuration)
* @ComponentScan (scans for components like controllers and services)

**Example:**

@SpringBootApplication

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

Without it, you’d have to manually add multiple annotations.

**4. How does Spring Boot simplify the development of Spring applications?**

Spring Boot reduces setup time by:

* Removing the need for XML configurations.
* Providing a default setup for databases, servers, and security.
* Offering dependency management through starter dependencies.

**Example:**  
In Spring Boot, you just add spring-boot-starter-web, and it sets up everything needed for a web application.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**5. What is the role of the pom.xml file in a Spring Boot project?**

pom.xml (Project Object Model) is used in **Maven-based** projects to manage dependencies.

**Roles:**

* Defines project dependencies (e.g., Spring Boot, database, security).
* Handles build processes (compiling, testing, packaging).
* Manages dependency versions.

**Example:**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

</dependencies>

This automatically downloads everything needed for JPA (database handling).

**6. What is the default embedded server in Spring Boot, and can you change it?**

* **Default:** Tomcat.
* **Can you change it?** Yes, by adding a different server dependency and excluding Tomcat.

**Example (Changing to Jetty):**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

This replaces Tomcat with Jetty.

**7. How do you configure the port number for a Spring Boot application?**

By changing the server.port property in **application.properties**.

**Example (Changing port to 8081 in application.properties):**

server.port=8081

Or, in **application.yml**:

server:

port: 8081

**8. What is the purpose of the application.properties or application.yml file?**

This file is used for configuring the application without changing the code.

**Examples of what you can configure:**

* Database connection
* Logging levels
* Server port
* Security settings

**Example (Configuring MySQL in application.properties):**

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=1234

Same in **application.yml**:

spring:

datasource:

url: jdbc:mysql://localhost:3306/mydb

username: root

password: 1234

**9. How do you enable or disable specific auto-configurations in Spring Boot?**

By using the @EnableAutoConfiguration or @SpringBootApplication annotation.

**Example (Disabling a specific auto-configuration class):**

@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

Alternatively, in **application.properties**:

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

This prevents auto-configuration of a database.

**10. What is the difference between @RestController and @Controller?**

* @Controller is used for traditional web applications (returns HTML pages).
* @RestController is used for REST APIs (returns JSON data).

@RestController = @Controller + @ResponseBody

**Example (Using @Controller to return a view):**

@Controller

public class HomeController {

@GetMapping("/")

public String home() {

return "home"; // Returns an HTML page named home.html

}

}

**Example (Using @RestController to return JSON):**

@RestController

public class ApiController {

@GetMapping("/hello")

public String sayHello() {

return "Hello, World!"; // Returns plain text as a response

}

}

In a REST API, you should use @RestController to return JSON data.

# Queries related to above questions

Great questions! Let's go step by step with clear explanations and examples.

**1. How does .class work? Does it work at the class level?**

.class is a special syntax in Java that gives a reference to the **Class object** of a specific type. It works at the **class level** and is used in reflection, annotations, and dependency injection.

**Example 1: Getting a Class Object**

Class<MyService> myServiceClass = MyService.class;

System.out.println(myServiceClass.getName()); // Output: com.example.MyService

Here, MyService.class returns the **Class object** of MyService.

**Example 2: Using .class in Annotations**

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

Here, DataSourceAutoConfiguration.class refers to a class that we are excluding from auto-configuration.

**Example 3: Using .class in Reflection**

Class<?> clazz = Class.forName("com.example.MyService");

Object obj = clazz.getDeclaredConstructor().newInstance();

This dynamically loads the MyService class and creates an object.

So, **.class works at the class level and provides metadata about the class.**

**2. How is a Repository Interface Object Created?**

In Spring Boot, we inject a Repository (which is an interface), but we never create an object manually. The object is automatically created by **Spring Data JPA or Spring Data MongoDB**.

**How does it work?**

1. You create an interface extending JpaRepository (for SQL) or MongoRepository (for MongoDB).
2. Spring automatically generates an implementation at runtime.
3. The repository object is injected using @Autowired.

**Example: Repository Interface**

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

User findByUsername(String username);

}

**Q: Who implements this interface?**  
**A: Spring Boot automatically generates an implementation at runtime.**

**Example: Injecting Repository into a Service**

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public User getUserByUsername(String username) {

return userRepository.findByUsername(username);

}

}

Since UserRepository extends JpaRepository, Spring Boot automatically provides an implementation, so we don’t need to write it manually.

**3. Does a Simple String Response Count as JSON in @RestController?**

No, a simple string is **not** JSON. When a @RestController returns a string, Spring Boot automatically wraps it in a response.

**Example 1: Returning a String**

@RestController

public class HelloController {

@GetMapping("/hello")

public String sayHello() {

return "Hello, World!";

}

}

**Response:**

Hello, World!

This is just a plain string, not JSON.

**Example 2: Returning JSON (Recommended)**

@RestController

public class UserController {

@GetMapping("/user")

public User getUser() {

return new User("John", 25);

}

}

**Response (Proper JSON):**

{

"name": "John",

"age": 25

}

So, **a simple string response is just text, not JSON.** JSON requires **key-value pairs** or an array.

**4. What Does @Configuration Do?**

@Configuration marks a class as a **configuration class**, meaning it contains Spring bean definitions.

**Example: Defining Beans in @Configuration**

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyService();

}

}

Now, MyService is available as a **Spring Bean**.

**5. What Does @EnableAutoConfiguration Do?**

@EnableAutoConfiguration allows Spring Boot to **automatically configure** beans based on the dependencies.

**Example 1: Without @EnableAutoConfiguration (Manual Configuration)**

@Configuration

public class ManualConfig {

@Bean

public DataSource dataSource() {

return new HikariDataSource();

}

}

Here, we **manually configure** the database.

**Example 2: With @EnableAutoConfiguration (Auto Configuration)**

@SpringBootApplication

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

Here, Spring Boot **automatically configures** the database without manual setup.

**6. Why Do We Need to Disable Auto-Configuration?**

Sometimes, auto-configuration loads unwanted components, so we exclude them.

**Example: Disabling DataSource Auto-Configuration**

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

public class MyApplication {

}

This prevents Spring Boot from **automatically configuring a database**, useful when using MongoDB (which does not need a SQL DataSource).

**7. Why Do We Need @ComponentScan?**

@ComponentScan tells Spring **where to look for components** (e.g., @Service, @Controller, @Repository).

**Example: Scanning a Specific Package**

@ComponentScan("com.example.service")

Now, Spring only looks for beans in **com.example.service**.

**Without @ComponentScan (Wrong Setup)**

@SpringBootApplication

public class MyApplication {

}

If your services are in a different package (com.other.service), Spring **won’t find them**.

**Final Summary**

| **Annotation** | **Purpose** | **Example** |
| --- | --- | --- |
| @Configuration | Defines beans manually | @Bean public MyService myService() { return new MyService(); } |
| @EnableAutoConfiguration | Enables auto-configuration | @SpringBootApplication |
| @ComponentScan | Scans for components | @ComponentScan("com.example.service") |
| @SpringBootApplication | Combines all three above | @SpringBootApplication |