**Database Connectivity**

* Application properties- Add jdbc connectivity api to just connect to the database.
* spring.jpa.show-sql=true - The database updates are seen and printed in the console or logging
* spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect spring.jpa.hibernate.ddl-auto=update- They create one session for the database and update in the database tables,otherwise separate sessions will be created for each query inputs.

**ORM**

* They map the java objects with the database rows or tables.Each object in the model is a row in the database table.
* DAO is similar to JPA repository and DTO is the model class for the required fields of the table that is created so that the most frequently used fields or sub tables can be accessed easily.
* JPA and Hibernate provides the ORM.The difference between them is that JPA gives the structure and not the concrete methods. Hibernate implements the JPA with more advanced features like caching, lazy loading,etc. Along with the JPA methods.
* @Repository public interface UserRepository extends JpaRepository<User, Long> { User findByUsernameAndPassword(String username, String password); }

Here, the User is the entity and the Long is the datatype of the primary key.

* The code is safe from sql injection since we pass the fields in the parameters of the methods present in the JPA.

@Repository

public interface UserRepository extends JpaRepository<User, Long>{

User findByUsernameAndPassword(String username, String password); }

* Instead of loading related entities **immediately** (Eager Loading), Hibernate loads them **only when needed**.This is called Lazy Loading.
* POJO class is a simple model class with no getter and setter(DTO).Java Bean is a model class with getter and setter(JSP).JPA Entity or Model class is bean that connects to database tables.

**MVC**

* The controller will get the request from the user.
* It connects to the model and either it fetch the data or updates the database
* The result of getting the data or updates are seen in the UI(view).

**Controller**

They deal with the url request of the user.

### @RestController

* It is a Spring annotation that marks a class as a controller that returns data (JSON/XML) instead of a view.
* It is a combination of @Controller + @ResponseBody

### **@RequestMapping**

* Used to map **URL requests** to specific controller methods.
* Can be applied **at the class level** (for a base URL) and **method level** (for specific endpoints).

**Example**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/products") // Base URL: localhost:8080/products

public class ProductController {

@Autowired

private ProductService productService;

// 1️⃣ GET all products

@GetMapping

public List<Product> getAllProducts() {

return productService.getAllProducts();

}

// 2️⃣ GET a product by ID

@GetMapping("/{id}")

public Product getProductById(@PathVariable Long id) {

return productService.getProductById(id);

}

// 3️⃣ POST: Create a new product

@PostMapping

public Product createProduct(@RequestBody Product product) {

return productService.saveProduct(product);

}

// 4️⃣ PUT: Update an existing product

@PutMapping("/{id}")

public Product updateProduct(@PathVariable Long id, @RequestBody Product product) {

return productService.updateProduct(id, product);

}

// 5️⃣ DELETE: Remove a product

@DeleteMapping("/{id}")

public String deleteProduct(@PathVariable Long id) {

productService.deleteProduct(id);

return "Product deleted successfully!";

}

}

* @Autowired- This creates the objects for the class used without the new keyword Automatic dependency injection and bean creation for the particular class.

@Qualified(<Name>) and @Primary is used to resolve conflicts and create using the user-defined name.

**@Path variable**

This is used to extract parameters from the path.It can pass single value

GET /products/5/category/Electronics

@GetMapping("/{id}/category/{categoryName}")

public String getProductCategory(

@PathVariable Long id,

@PathVariable String categoryName) {

return "Product ID: " + id + ", Category: " + categoryName;

}

**@RequestParam-**  extract query parameters. They are not mandatory fields.

GET /products/10/details?includeReviews=true

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

public String getProductDetails(

@PathVariable Long id,

@RequestParam(required = false, defaultValue = "false") boolean includeReviews) {

return "Product ID: " + id + ", Include Reviews: " + includeReviews;

}

**@RequestBody-** It is used to pass json or xml objects. We have to create a DTO class for this to pass the parameter in the controller class.

**Json to Oject Conversion and Vice-versa**

@PostMapping("/add")

public ResponseEntity<Order> addOrder(@RequestBody Order order) throws Exception {

Order savedOrder = orderRepository.save(order);

System.out.println(savedOrder.toString());

ObjectMapper objectMapper=new ObjectMapper();

Order order1=objectMapper.readValue(savedOrder.toString(),Order.class);

String jsonString = objectMapper.writeValueAsString(order);

System.out.println("Json to Object"+order1.toString());

System.out.println("Object to Json"+jsonString);

return new ResponseEntity<>(savedOrder, HttpStatus.CREATED);

}

**Working of Spring Boot Application**

* **Dependency Injection or Inversion Of Control(IOC)**

The dependency jdbc driver in the porm.xml file is automatically configures the datasource.

A **DataSource** is an object that provides **database connections**

Traditionally, we use

@Configuration

public class CustomDataSourceConfig {

@Bean

public DataSource customDataSource() {

HikariDataSource dataSource = new HikariDataSource();

dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/customdb");

dataSource.setUsername("user");

dataSource.setPassword("password");

return dataSource;

}

}

This configuration class must be defined. But now we only include the credentials in the application properties. It automatically get the configuration of the database.

* **No bean creation**

No new keyword is used to create the class.

Once the application is launched all the beans will be created automatically.

**Benefits of Using Beans and Dependency Injection**

**Loose Coupling** → The controller doesn’t create the object; Spring provides it.  
 **Easier Testing** → You can easily replace MyService with a mock during testing.  
 **Centralized Management** → Spring **creates, manages, and injects** objects efficiently.  
 **Efficient Memory Usage** → Spring maintains a **single shared instance** of beans (Singleton Scope by default).

* **Autoconfiguration**

The process of setting up and managing the application's behavior, including defining beans, dependencies, and externalized settings is configuration.

@EnableAutoConfiguration-present inside @SpringBootApplication by default

### 

### 

### **How does Auto-Configuration Work?**

1. Spring Boot loads **spring.factories** file from spring-boot-autoconfigure.jar.
2. It checks which configurations are required based on dependencies.
3. It automatically registers required beans.

Example:  
 If spring-boot-starter-data-jpa is present, Spring Boot:

* Auto-configures **Hibernate EntityManager**
* Sets up a **DataSource**
* Enables **Spring Data JPA**

### **What Happens in SpringApplication.run()?**

1. **Creates an Application Context**
   * Spring Boot initializes the ApplicationContext (usually AnnotationConfigServletWebServerApplicationContext for web apps).
2. **Starts an Embedded Web Server** (if it's a web application)  
   * Default server is **Tomcat** (can be changed to Jetty, Undertow, etc.).
3. **Performs Auto-Configuration**
   * Uses @EnableAutoConfiguration to configure beans based on the classpath.
4. **Scans Components (@ComponentScan)**
   * Looks for @Component, @Service, @Repository, @Controller, and @RestController in the package.

Here the server is responsible for http request and response management

### **What is DispatcherServlet?**

* The **DispatcherServlet** is the core component that processes all HTTP requests in a Spring Boot application.
* It acts as a **front controller** (entry point for all incoming requests).

### **How Does It Work?**

* DispatcherServlet receives the request and **checks the URL pattern** (/api/message).
* It then **looks for a matching controller method**.

It also handles the responses and the return type is **json by default**, if the return type is of type object or the return type is not mentioned as string explicitly.

## **Spring Boot Shutdown Process**

* When the application stops (Ctrl+C or server shutdown), Spring Boot:  
  1. **Calls @PreDestroy methods** of beans.
  2. **Triggers DisposableBean.destroy()**.
  3. **Stops the Web Server** (Tomcat, Jetty, etc.).
  4. **Closes the ApplicationContext**.

**@Autowired**- it automatically injects the bean of the class without new keyword.it is also used to forcefully inject the bean.

Example:

@Autowired // Injecting UserService automatically

private UserService userService;

Forceful injection

@Autowired

@Qualifier("serviceB1") // Force Spring to inject "serviceB1"

private ServiceB serviceB;

**@Component**

It tells Spring that the class **should be a managed bean**.  
 Spring **automatically detects and registers** this class.  
 Specialized versions of @Component:

* + @Service → For **business logic classes**
  + @Repository → For **database layer classes**
  + @Controller / @RestController → For **handling web requests**

**@Configuration**

It is used to create a custom or user defined bean

**@postconstruct**

* It is executed after the autowired, that is after the bean is initialised.
* It runs automatically after all the dependencies and beans are injected.
* Initializing resources, loading configurations, setting up cache are the use cases.
* The scope is only for methods, not for class
* If many postconstruct methods are present in the main class, then they are executed with no order.

**@ApplicationContext**

It manages all the beans and dependencies

**Lifecycle of ApplicationContext**

### 1. Bean Instantiation

Spring creates objects (beans) for @Component, @Service, @Repository, etc.

### 2. Dependency Injection

It injects dependencies marked with @Autowired.

### 3. Bean Initialization

Calls @PostConstruct methods.

### 4. ApplicationContext is Ready

* The application is now ready to handle requests.

### 5. ApplicationContext Shutdown

* Calls @PreDestroy methods before closing.

**Note**

* The spring creates beans only for the classes that has @component,@Autowired and mandatory classes.others are created based on the requirements.
* The default database in spring boot is h2 database

**Event Listener**

### **How It Works**

1. Define a Custom Event (UserRegisteredEvent).
2. Publish the Event (UserService triggers UserRegisteredEvent).
3. Listen for the Event (WelcomeEmailListener and AsyncNotificationListener handle the event).
4. Execute Synchronous & Asynchronous Tasks (@EventListener for sync, @Async for async).
5. Test It on Application Startup (AppStartupRunner simulates user registration).

**Example**

**Send a Welcome Email after Registration to a website**

**Asynchronous**

Registration completes faster, even if email sending is slow.

**Traditional method or Tight coupling**

public void registerUser(String username, String email) {

System.out.println("✅ User registered: " + username);

// Directly calling multiple services

emailService.sendWelcomeEmail(email);

couponService.generateDiscountCoupon(username);

loggingService.logUserRegistration(username);

}

Here all the services are in one method

**Solution**

**The service class the event is published, that is all the related events such as email sending, coupon generation,etc are called which independent of all others(loose coupled)**

**1) Step 1: Create an Event (UserRegisteredEvent)**

import org.springframework.context.ApplicationEvent;

public class UserRegisteredEvent extends ApplicationEvent {

private final String username;

private final String email;

public UserRegisteredEvent(Object source, String username, String email) {

super(source);

this.username = username;

this.email = email;

}

public String getUsername() { return username; }

public String getEmail() { return email; }

}

**2) Step 2: Publish the Event in UserService**

import org.springframework.context.ApplicationEventPublisher;

import org.springframework.stereotype.Service;

@Service

public class UserService {

private final ApplicationEventPublisher eventPublisher;

public UserService(ApplicationEventPublisher eventPublisher) {

this.eventPublisher = eventPublisher;

}

public void registerUser(String username, String email) {

System.out.println("✅ User registered: " + username);

// 🔥 Publishing the event

UserRegisteredEvent event = new UserRegisteredEvent(this, username, email);

eventPublisher.publishEvent(event);

}

}

**3)Step 3: Create an Event Listener for Welcome Email**

import org.springframework.context.event.EventListener;

import org.springframework.stereotype.Component;

@Component

public class WelcomeEmailListener {

@EventListener

public void handleUserRegistered(UserRegisteredEvent event) {

System.out.println("Sending welcome email to: " + event.getEmail());

// Logic to send email

}

}

**4)Step 4: Create Another Event Listener for Coupons**

import org.springframework.context.event.EventListener;

import org.springframework.stereotype.Component;

@Component

public class CouponServiceListener {

@EventListener

public void handleUserRegistered(UserRegisteredEvent event) {

System.out.println("🎁 Generating discount coupon for: " + event.getUsername());

// Logic to generate and send coupon

}

}

**Spring Boot Actuators**

Spring Boot Actuator is an essential tool for **monitoring, managing, and troubleshooting** your Spring Boot application in real time. It provides **built-in endpoints** for health checks, metrics, environment details, **Loggers and Tracing** (Application logs, request tracing),**Thread dumps and HTTP traces** (Identify slow requests),**JVM statistics** (Garbage Collection, Heap usage).

**Spring Boot Profiles**

Spring Boot Profiles allow developers to define different configurations for different environments (e.g., development, testing, production). Instead of modifying the configuration manually in the application properties, you can use profiles to switch configurations automatically based on the environment.

## **Annotations**

| **Annotation** | **Used For** | **Example Usage** |
| --- | --- | --- |
| @Profile("env") | Enable a bean only for a specific environment | @Profile("dev") |
| @Value("${property}") | Inject environment-specific values | @Value("${server.port}") |
| @ConditionalOnProperty | Enable/Disable a bean based on properties | @ConditionalOnProperty(name = "cache.enabled", havingValue = "true") |
| @EnableAutoConfiguration | Auto-configure Actuator and other Spring components | @EnableAutoConfiguration |
| @Endpoint(id = "name") | Create a custom Actuator endpoint | @Endpoint(id = "custom") |
| @ReadOperation | Define a GET operation in an Actuator endpoint | @ReadOperation |
| @WriteOperation | Define a POST operation in an Actuator endpoint | @WriteOperation |
| @DeleteOperation | Define a DELETE operation in an Actuator endpoint | @DeleteOperation |
| @ConditionalOnEnabledHealthIndicator | Enable a health check only if configured | @ConditionalOnEnabledHealthIndicator("customHealth") |

## **Order of Execution**

1️⃣ Command-line arguments (--server.port=9090)  
 2️⃣ Java System Properties (-Dserver.port=9090)  
 3️⃣ OS Environment Variables (export SERVER\_PORT=9090)  
 4️⃣ JNDI Attributes (java:comp/env)  
 5️⃣ Random Values (random.\*)  
 6️⃣ @PropertySource in @Configuration classes  
 7️⃣ Application Properties (External application.properties)  
 8️⃣ Application Properties (Inside JAR)  
 9️⃣ Default Properties (SpringApplication.setDefaultProperties)  
 🔟 Dependency Injection & Auto-Configuration(Application Context)  
 1️⃣1️⃣ Database Initialization (DataSource, Hibernate, Migrations) and connections  
 1️⃣2️⃣ Run @PostConstruct, CommandLineRunner, ApplicationRunner  
 1️⃣3️⃣ Application Ready Event (ApplicationReadyEvent)  
 1️⃣4️⃣ Handles Web Requests & Actuators  
 1️⃣5️⃣ Shutdown Hooks & Cleanup (@PreDestroy)

**Questions**

1. What is Spring Boot, and how does it differ from the traditional Spring Framework?

2. What is the purpose of @springbootapplication annotation?

3. Explain the concept of auto-configuration in Spring Boot.

4. What is the role of [application.properties](https://www.linkedin.com/redir/redirect?url=http%3A%2F%2Fapplication%2Eproperties&urlhash=zpBF&trk=public_post-text) or [application.yml](http://application.yml/?trk=public_post-text) in Spring Boot?

5. What are the advantages of using Spring Boot over traditional Spring applications?

6. What is the difference between @̲component,̲ ̲@̲service,̲ ̲@repository,̲ and @controller in Spring Boot?

7. How do you create a Spring Boot RESTful web service?

8. What is Spring Boot Actuator, and what are its uses?

9. What is the significance of @̲restcontroller and how is it different from @controller?

10. How do you handle exceptions in Spring Boot applications?

11. What is Spring Boot DevTools, and what benefits does it provide during development?

12. How do you use Spring Boot to connect to a database (e.g., MySQL or PostgreSQL)?

13. Explain the difference between @bean and @component in Spring Boot.

14. How do you configure logging in Spring Boot?

15. What are Spring Boot profiles, and how do you manage them for different environments (dev, prod)?

16. How do you implement security in a Spring Boot application?

17. What is Spring Boot’s embedded server, and how does it work?

18. What is the use of @enableautoconfiguration in Spring Boot?

19. What is the difference between Spring Boot’s [application.properties](https://www.linkedin.com/redir/redirect?url=http%3A%2F%2Fapplication%2Eproperties&urlhash=zpBF&trk=public_post-text) and [application.yml](http://application.yml/?trk=public_post-text) files?

20. What is Spring Boot's support for creating microservices?

21. How does Spring Boot handle dependency injection?

22. How do you test a Spring Boot application using @springboottest?

23. Explain the difference between @̲requestmapping,̲ ̲@getmapping,̲ ̲@̲postmapping and other HTTP method annotations.

24. What are Spring Boot’s default error handling mechanisms, and how can they be customized?

25. How do you perform batch processing in Spring Boot?

26. What is Spring Data JPA, and how is it used in Spring Boot?

27. How do you manage transaction handling in Spring Boot applications?

28. What are the different ways to run a Spring Boot application?

29. Explain the Spring Boot logging mechanism with default loggers (e.g., Logback).

30. How do you configure Spring Boot to send an email (e.g., using JavaMailSender)?

1. Suppose you have different types of users (AdminUser, RegularUser, GuestUser). How to implement this using inheritance?

2. What are the advantages and disadvantages of using inheritance for user roles instead of using an enum?

3. Can you explain how you would handle multiple authentication mechanisms (OAuth, database authentication) using inheritance?

4. How to design a LoginService interface that supports both JWT-based authentication and OAuth authentication?

5. Can you implement a UserService where the authenticateUser() method behaves differently for Admin and Regular users?

6. How does polymorphism help in implementing different authentication strategies in Spring Security?

7. Why is it a good practice to define an AuthenticationProvider abstract class instead of writing authentication logic directly in a controller?

8. How does Spring Security use abstraction to handle different types of authentication mechanisms?

9. Can you create an abstract class for UserValidation that enforces validation rules for all user types?

### 

| **Annotation** | **Purpose** | **Use Case** |
| --- | --- | --- |
| @Component | Generic Spring Bean | Used for general-purpose Spring-managed classes |
| @Service | Business Logic Layer | Service classes handling business logic |
| @Repository | Data Access Layer | DAO classes interacting with the database |
| @Controller | Web Layer | Handles HTTP requests in MVC architecture |

**@Bean and @Component**

@Bean → Used for manual bean creation inside @Configuration.

@Component → Used for automatic component scanning in the classpath.

@Bean is useful for third-party classes, while @Component is used for custom classes in the application.

**Exception Handling**

@ExceptionHandler(UserNotFoundException.class)\

@ControllerAdvice