**UNIT-2-** **Spring Boot Application Development: Overview of Dependency Injection and Inversion of Control , Spring Boot Configuration - Configuration properties and profiles, Externalizing configurations , Spring Boot Auto-Configuration- Understanding and Customizing auto-configuration.**

**Creating RESTful Web Services : REST principles and concepts, Building REST controllers with RestController , Request mapping with RequestMapping and GetMapping, PostMapping.**

**Consuming REST services-Consuming REST endpoints with RestTemplate( GETting resources, PUTting resources, DELETEing resources, POSTing resource data), Navigating REST APIs with Traverson.**

**Textbook 1: Chapter 2: 2.1 ,Chapter 5: 5.1 to 5.3**

**Textbook 1: Chapter 6: 6.1 to 6.3 ,Chapter 7: 7.1 to 7.2**

**Spring Boot Application Development**

**1. Dependency Injection and Inversion of Control**

**1.1 Inversion of Control (IoC)**

**Definition: IoC is a design principle in software engineering where the control of object creation and lifecycle management is transferred from the programmer to a container or framework.**

**Benefits:**

**Decouples object creation from its usage**

**Improves testability and maintainability**

**Promotes reusability and modularity**

**Implementation in Spring Boot: Spring implements IoC using the Spring IoC Container, which manages the lifecycle and dependencies of Spring beans.**

**1.2 Dependency Injection (DI)**

**Definition: Dependency Injection is a design pattern where dependencies (objects) are injected into a class instead of being created within the class.**

**Types of Dependency Injection:**

**Constructor Injection - Injecting dependencies via the class constructor.**

**This is the most recommended approach as it ensures the class is always in a valid state with all dependencies provided.**

**Example:**

**@Service**

**public class UserService {**

 **private final UserRepository userRepository;**

 **@Autowired**

 **public UserService(UserRepository userRepository) {**

 **this.userRepository = userRepository;**

 **}**

**}**

**Setter Injection - Injecting dependencies via setter methods.**

**This allows optional dependencies and greater flexibility.**

**Example:**

**@Service**

**public class UserService {**

 **private UserRepository userRepository;**

 **@Autowired**

 **public void setUserRepository(UserRepository userRepository) {**

 **this.userRepository = userRepository;**

 **}**

**}**

**Field Injection - Injecting dependencies directly into fields using @Autowired annotation.**

**This is the least recommended approach as it makes testing and dependency management harder.**

**Example:**

**@Service**

**public class UserService {**

 **@Autowired**

 **private UserRepository userRepository;**

**}**

**2. Spring Boot Configuration**

**2.1 Configuration Properties and Profiles**

**Spring Boot allows configuring application properties using application.properties or application.yml files.**

**Configuration Properties**

**Used to define and manage configurations in a structured way using @ConfigurationProperties annotation.**

**Example:**

**@ConfigurationProperties(prefix = "app")**

**@Component**

**public class AppConfig {**

 **private String name;**

 **private int timeout;**

 **// Getters and Setters**

**}**

**Profiles**

**Profiles allow defining different configurations for different environments like development, testing, and production.**

**Example of setting profiles in application.properties:**

**spring.profiles.active=dev**

**Or in application-dev.properties:**

**server.port=8081**

**3. Externalizing Configurations**

**Externalized configurations allow modifying the application behavior without changing the source code.**

**Ways to Externalize Configuration:**

1. **Application Properties/YAML files**
2. **Command-line arguments**
3. **Environment Variables**
4. **Java System Properties**
5. **Spring Cloud Config Server (for distributed systems)**

**Example of Command-line Arguments:**

**java -jar myapp.jar --server.port=9090**

**4. Spring Boot Auto-Configuration**

**4.1 Understanding Auto-Configuration**

**Spring Boot automatically configures components based on the dependencies present in the classpath.**

**Example: If spring-boot-starter-web is added, Spring Boot automatically configures:**

* **Embedded Tomcat**
* **DispatcherServlet**
* **Web MVC components**

**4.2 Customizing Auto-Configuration**

**Spring Boot provides mechanisms to customize auto-configuration:**

1. **Excluding Auto-Configuration Classes**

**@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)**

1. **Using @Conditional Annotations Spring Boot uses conditions to enable or disable beans dynamically:**

**@Bean**

**@ConditionalOnMissingBean(DataSource.class)**

**public DataSource dataSource() {**

 **return new HikariDataSource();**

**}**

1. **Spring Boot Configuration Properties**

**@ConfigurationProperties(prefix = "app")**

**public class AppConfig {**

 **private String name;**

**}**

**Conclusion**

* **IoC and DI simplify dependency management and improve modularity.**
* **Spring Boot configuration allows flexible management using properties and profiles.**
* **Externalization enables dynamic configurations without modifying source code.**
* **Auto-Configuration simplifies setup and can be customized as needed.**

**5. Creating RESTful Web Services**

**5.1 REST Principles and Concepts**

**REST (Representational State Transfer) is an architectural style that defines a set of constraints for building web services.**

**Key principles:**

* **Statelessness**
* **Client-Server Architecture**
* **Cacheability**
* **Layered System**
* **Uniform Interface**
* **Code on Demand (optional)**

**5.2 Building REST Controllers with @RestController**

**Spring Boot provides the @RestController annotation to simplify REST API development.**

**Example:**

**@RestController**

**@RequestMapping("/users")**

**public class UserController {**

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

 **public User getUser(@PathVariable Long id) {**

 **return new User(id, "John Doe");**

 **}**

**}**

**5.3 Request Mapping with @RequestMapping, @GetMapping, and @PostMapping**

* **@RequestMapping maps HTTP requests to controller methods.**
* **@GetMapping handles GET requests.**
* **@PostMapping handles POST requests.**

**Example:**

**@RestController**

**@RequestMapping("/products")**

**public class ProductController {**

 **@GetMapping**

 **public List<Product> getAllProducts() {**

 **return productService.getAllProducts();**

 **}**

 **@PostMapping**

 **public Product createProduct(@RequestBody Product product) {**

 **return productService.saveProduct(product);**

 **}**

**}**

**6. Consuming REST Services**

**6.1 Using RestTemplate to Consume REST APIs**

**GET Request Example:**

**RestTemplate restTemplate = new RestTemplate();**

**User user = restTemplate.getForObject("https://api.example.com/users/1", User.class);**

**POST Request Example:**

**User newUser = new User("Jane Doe");**

**User response = restTemplate.postForObject("https://api.example.com/users", newUser, User.class);**

**PUT Request Example:**

**restTemplate.put("https://api.example.com/users/1", newUser);**

**DELETE Request Example:**

**restTemplate.delete("https://api.example.com/users/1");**

**6.2 Navigating REST APIs with Traverson**

**Traverson helps navigate hypermedia-driven APIs.**

**Example:**

**Traverson traverson = new Traverson(URI.create("https://api.example.com"), MediaTypes.HAL\_JSON);**

**String userName = traverson.follow("users", "self").toObject(String.class);**

**Conclusion**

* **REST principles guide API design.**
* **Spring Boot simplifies REST API creation using @RestController.**
* **RestTemplate and Traverson help consume REST services effectively.**