**WEEK 6 COUSE CONTENT: Introduction to Spring Boot and its features**

**TOPICS:**

- Provide an overview of Spring Boot and its advantages in Java application development.
- Explain the key features and benefits of using Spring Boot for building Java applications.
- Spring Boot configuration and auto-configuration
- Dependency Injection and Inversion of Control
- Spring Boot Starter Packs and Libraries
- Building RESTful APIs with Spring Boot

**LEARNING OUTCOMES:**

**By the end of this module, students will be able to:**

- Develop spring boot applications
- Building application using spring boot
- Configuration of spring boot application using YAML, application.properties and other files
- Injecting dependency and services
- Developing and running RESTful Web APIs
- Configuring spring application

**DAY 1**

**Introduction to Spring Boot and its Features**

**Overview of Spring Boot:**

Spring Boot is an open-source framework built on top of the Spring Framework that simplifies the development of Java applications. It provides a convention-over-configuration approach and aims to make it easy to create stand-alone, production-grade Spring-based applications with minimal effort. Spring Boot eliminates much of the boilerplate configuration required in traditional Spring applications, allowing developers to focus more on writing business logic.

**Key Features and Benefits of Spring Boot:**

- **Auto (Easy)-configuration:** Spring Boot uses a feature called auto-configuration, which automatically configures the application based on the dependencies present in the classpath. It analyzes the classpath and sets up sensible defaults, reducing the need for manual configuration. Developers can override or customize the auto-configuration as per their requirements.
- **Starter Dependencies:** Spring Boot provides a set of starter dependencies that encapsulate all the necessary libraries and configurations required to work with specific

technologies, such as web applications, databases, security, messaging, etc. These starters simplify dependency management and reduce the time spent on configuring dependencies manually.

- **Embedded (Built-in) Server:** Spring Boot includes an embedded server, such as Tomcat, Jetty, or Undertow, which allows you to create standalone, executable JAR files. This eliminates the need for deploying the application to an external server and simplifies the deployment process.
- **Production-Ready Features:** Spring Boot offers several production-ready features, including metrics, health checks, externalized configuration, logging, and more. These features make it easier to monitor and manage the application in a production environment.
- **Actuator:** Spring Boot Actuator provides endpoints to monitor and manage the application at runtime. It offers various out-of-the-box endpoints, such as health, metrics, info, and environment, allowing you to gather insights and perform administrative tasks easily.
- **Developer Tools:** Spring Boot provides a set of developer tools that enhance the developer experience. These tools include automatic application restart, live reload, and remote debugging support, improving productivity during the development process.
- **Rapid Prototyping:** Spring Boot's ease of use and rapid development capabilities make it ideal for rapid prototyping. It allows developers to quickly create a functional prototype with minimal effort and iterate on it to validate ideas or demonstrate concepts.

**Spring Boot Ecosystem and Integration with Other Spring Projects:**

Spring Boot is part of the broader Spring ecosystem and integrates seamlessly with other Spring projects. Some notable integrations include:

- **Spring Framework:** Spring Boot builds on top of the Spring Framework, leveraging its powerful features and providing additional conveniences. It simplifies the configuration and setup process for Spring-based applications.
- **Spring Data:** Spring Boot integrates with Spring Data, which simplifies database access and management. Spring Data provides a unified and consistent API for working with different databases, such as relational databases (Spring Data JPA), NoSQL databases (Spring Data MongoDB), and more.
- **Spring Security:** Spring Boot seamlessly integrates with Spring Security, which provides authentication, authorization, and security features for applications. Spring Security simplifies securing your application by providing default configurations and customizable options.
- **Spring Cloud:** Spring Boot plays a crucial role in the Spring Cloud ecosystem. Spring Cloud provides tools and libraries for building distributed systems and microservices. Spring Boot simplifies the development of microservices by providing a strong

foundation and compatibility with Spring Cloud components like service discovery, circuit breakers, distributed configuration, and more.

- **Spring Integration:** Spring Boot integrates with Spring Integration, which enables the development of messaging and integration solutions. It provides seamless integration with various messaging systems, such as Apache Kafka, RabbitMQ, and others.

By leveraging the Spring ecosystem, Spring Boot offers a comprehensive solution for building enterprise-grade Java applications. It provides a unified and cohesive environment, enabling developers to focus on business logic rather than infrastructure and configuration details.

Overall, Spring Boot's key features, such as auto-configuration, starter dependencies, embedded server, and production-ready features, combined with its seamless integration with other Spring projects, make it a powerful framework for building robust, scalable, and maintainable Java applications.

**Spring Boot Configuration and Auto-configuration**

**Introduction:** Spring Boot is a popular framework for building Java applications that simplifies the configuration process and provides opinionated defaults to get started quickly. In this discussion, we will explore the configuration options available in Spring Boot, including property files, YAML files, and environment variables. We will also learn how to configure a Spring Boot application using these approaches and understand the concept of auto-configuration and its benefits.

**Configuration Options:** Spring Boot provides multiple options to configure your application:

- **Property Files:** Property files are a common way to configure Spring Boot applications. By default, Spring Boot looks for a file named **application.properties** in the classpath. You can define key-value pairs in this file to configure various aspects of your application, such as database connection settings, server port, logging levels, etc. Here's an example of configuring the server port in **application.properties**:

```
server.port=8080
```

- **YAML Files:** YAML files offer a more structured and readable configuration format compared to property files. Spring Boot supports YAML configuration out of the box. To use YAML files, create a file named **application.yml** in the classpath and define your configuration in a hierarchical structure. Here's an example of configuring the server port in **application.yml**:

```
server:
    port: 8080
```

- **Environment Variables:** Spring Boot also allows configuration through environment variables. You can set environment variables on your server or in your deployment environment and access them in your application. To configure the server port using an environment variable:

```
export SERVER_PORT=8080
```

**Configuring a Spring Boot Application**: To configure a Spring Boot application, you can use the **@Configuration** annotation to create a configuration class. In this class, you can define beans, configure third-party libraries, and customize various aspects of your application.
Here's an example of a configuration class defining a bean:

```
@Configuration
        public class MyConfiguration {

           @Bean
           public MyBean myBean() {
              // Bean initialization logic
              return new MyBean();
           }
        }
```

**Auto-configuration in Spring Boot**: Auto-configuration is a powerful feature of Spring Boot that automatically configures your application based on the classpath dependencies you include. It eliminates the need for manual configuration by analyzing the classpath and providing sensible defaults.

When you include a dependency in your project, Spring Boot scans it and automatically configures beans, services, and other components required by that dependency. This greatly simplifies application setup and dependency management, reducing the boilerplate code and manual configuration.

For example, if you include the Spring Data JPA dependency, Spring Boot will auto-configure the database connection, transaction management, and repositories, allowing you to start working with a database without writing extensive configuration code.

Auto-configuration in Spring Boot is based on condition annotations that check for the presence of certain classes, dependencies, or properties. You can also customize auto-configuration using **@ConditionalOn...** annotations.

Conclusion: In this discussion, we covered the configuration options available in Spring Boot, including property files, YAML files, and environment variables. We learned how to configure a

Spring Boot application using these approaches and explored the concept of auto-configuration, which simplifies application setup and dependency management. By leveraging Spring Boot's configuration capabilities, developers can focus more on writing business logic and less on tedious configuration tasks.

## DAY 2

### Dependency Injection and Inversion of Control (IoC)

**Introduction:** In the context of Spring Boot, understanding dependency injection and inversion of control (IoC) is essential. Dependency injection is a concept that helps manage object dependencies, while inversion of control refers to a design principle where the control over object creation and management is handed over to a framework or container. In this discussion, we will explore these concepts in detail, explain their benefits, and demonstrate how to use them in Spring Boot using annotations and XML configuration.

- Dependency Injection and its Benefits: Dependency injection is a powerful technique used in Spring Boot to manage object dependencies. It allows objects to receive their required dependencies from an external source, rather than creating them on their own. This has several benefits:
- Loose Coupling: Dependency injection promotes loose coupling between objects, which means they are not tightly bound to each other. It allows for more flexibility in changing and maintaining code because modifications to one object's dependencies do not affect the entire codebase.
- Testability: With dependency injection, it becomes easier to write unit tests for individual objects. By injecting mock or test-specific dependencies, we can isolate the object being tested and verify its behaviour without relying on real dependencies.
- Modular Development: Dependency injection encourages modular development by focusing on the specific responsibilities of each object. It helps break down complex systems into smaller, manageable components that can be easily developed, maintained, and reused.
- Scalability: By decoupling dependencies, it becomes simpler to add or replace components as the application grows or requirements change. Dependency injection provides a flexible way to extend the functionality of an application without modifying existing code extensively.
- Dependency Injection in Spring Boot: Spring Boot provides convenient ways to implement dependency injection:

**Annotation-based Dependency Injection:** In Spring Boot, you can use annotations like **@Autowired**, **@Inject**, or **@Resource** to indicate that a dependency should be injected into a class.

For example, suppose we have a service class that requires a repository dependency:

```
@Service
    public class MyService {
      private final MyRepository myRepository;

      @Autowired
      public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
      }
      // ...
    }
```

In this example, the **@Autowired** annotation is used to inject the **MyRepository** dependency into the **MyService** class through the constructor.

**XML-based Dependency Injection:** Spring Boot also supports XML-based configuration for dependency injection. You can define beans and their dependencies in an XML file, typically named **applicationContext.xml**.

Here's an example of XML configuration for dependency injection:

```
    <beans>
      <bean id="myService" class="com.example.MyService">
        <constructor-arg ref="myRepository"/>
      </bean>

      <bean id="myRepository" class="com.example.MyRepository"/>
    </beans>
```

**Inversion of Control (IoC) in Spring Boot:** Inversion of Control is a principle closely related to dependency injection. In Spring Boot, it means that the control over object creation and management is handed over to the framework.
The IoC container in Spring Boot, known as the ApplicationContext, takes care of creating and managing beans based on the configuration provided. It automatically resolves and injects dependencies, relieving developers from the burden of manually creating and managing objects.

By using inversion of control, Spring Boot promotes loose coupling, modular development, and simplifies application configuration.

**Conclusion:** Dependency injection and inversion of control are fundamental concepts in Spring Boot. Dependency injection helps manage object dependencies, promotes loose coupling, testability, and modular development. Spring Boot provides convenient ways to implement dependency injection using annotations and XML configuration. Additionally, by embracing inversion of control, Spring Boot's IoC container takes care of object creation and management, reducing the amount of boilerplate code. Understanding and applying these concepts in Spring Boot empowers beginners to build well-structured, maintainable, and scalable applications.

**Spring Boot Starter Packs and Libraries**

Introduction: Spring Boot starter packs and libraries are essential components of the Spring Boot ecosystem. They simplify the setup and configuration of Spring Boot applications by providing pre-configured dependencies and opinionated defaults. In this discussion, we will introduce Spring Boot starter packs, explore commonly used libraries, and demonstrate how to include and configure them in a Spring Boot application.

**Spring Boot Starter Packs:** Spring Boot starter packs are a set of dependencies bundled together to provide a specific functionality or support for a particular technology. They simplify the process of setting up a Spring Boot application by including all the necessary dependencies and providing sensible defaults and configurations.

The purpose of starter packs is to reduce the amount of boilerplate code and configuration required to start a new project. They eliminate the need to manually search for and add individual dependencies, as well as configure various components. Starter packs help developers quickly get started with a specific feature or technology in Spring Boot.

**Commonly Used Starter Packs and Libraries:** There are numerous starter packs and libraries available in the Spring Boot ecosystem, catering to different needs and technologies. Here are some commonly used ones:

- **Spring Boot Starter Web:** The **spring-boot-starter-web** pack is used for developing web applications. It includes dependencies for building RESTful APIs, handling HTTP requests, and serving web content. It integrates with libraries like Spring MVC and embeds a servlet container for deployment.

- **Spring Boot Starter Data JPA:** The **spring-boot-starter-data-jpa** pack simplifies the integration of Spring Data JPA into your application. It provides dependencies for working with databases using the Java Persistence API (JPA), including automatic configuration for data source, transaction management, and entity management.
- **Spring Boot Starter Security:** The **spring-boot-starter-security** pack enables security features in your application. It includes dependencies for authentication, authorization, and securing endpoints. It provides an easy way to configure user authentication, role-based access control, and integration with various authentication providers.
- **Spring Boot Starter Test:** The **spring-boot-starter-test** pack includes dependencies for testing Spring Boot applications. It provides tools for unit testing, integration testing, and writing mock objects. It also includes libraries like JUnit and Mockito for writing test cases.

**Including and Configuring Starter Packs and Libraries:** To include a starter pack or library in your Spring Boot application, you need to add the corresponding dependency to your project's build configuration, such as Maven or Gradle.

For example, to include the Spring Boot Starter Web pack using Maven, add the following dependency in your **pom.xml** file:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

Once the dependency is added, Spring Boot automatically detects and configures the required components based on the starter pack. You can further customize the configuration by modifying properties in the **application.properties** or **application.yml** file.

For more advanced configurations, you can create additional classes and annotations specific to the starter pack or library. These classes can be annotated with various Spring annotations to further customize the behavior of your application.

Conclusion: Spring Boot starter packs and libraries simplify the setup and configuration of Spring Boot applications by providing pre-configured dependencies and opinionated defaults. They reduce the effort required to integrate specific features and technologies, allowing developers to focus on building their application logic. By including the desired starter packs or libraries in your project's build configuration and customizing the configuration as needed, you can leverage the power of Spring Boot's ecosystem to develop robust and feature-rich applications.

**DAY 3**

**Building RESTful APIs with Spring Boot**

**Introduction**: Building RESTful APIs has become a common practice in modern application development. REST (Representational State Transfer) is an architectural style that provides a set of principles for designing networked applications. In this discussion, we will provide an overview of RESTful principles, explain how to design and develop RESTful APIs using Spring Boot, and demonstrate key aspects such as request mapping, serialization, error handling, and more.

**Overview of RESTful Principles:** RESTful principles are based on the following key concepts:

- **Resources:** RESTful APIs are centered around resources, which are entities or objects that can be identified by a unique identifier (URI). Resources can be accessed, manipulated, and represented using standard HTTP methods.
- **Uniform Interface:** RESTful APIs adhere to a uniform interface that follows standard HTTP methods (GET, POST, PUT, DELETE) for different operations on resources. The API should have well-defined endpoints for each resource and use appropriate HTTP status codes to indicate the outcome of requests.
- **Statelessness:** RESTful APIs are stateless, meaning that each request from a client must contain all the necessary information for the server to understand and process it. The server does not store any session or client-specific data between requests.
- **Representation:** Resources in RESTful APIs are represented using a standardized format, such as JSON or XML. The client and server can negotiate the format using the **Accept** and **Content-Type** headers.

**Designing and Developing RESTful APIs with Spring Boot:** Spring Boot provides a powerful framework for building RESTful APIs. Here's an overview of the steps involved:

- **Define Resources:** Identify the resources that your API will expose and determine their unique URIs. Map each resource to a corresponding Java class.
- **Implement Controllers:** Create controller classes in Spring Boot that handle HTTP requests and define the endpoints for each resource. Use annotations like **@RestController** and **@RequestMapping** to map requests to specific methods in the controller.

- **Handle Requests and Responses:** In the controller methods, use annotations such as **@GetMapping**, **@PostMapping**, **@PutMapping**, and **@DeleteMapping** to handle specific HTTP methods. Use method parameters and annotations like **@PathVariable** and **@RequestBody** to access request parameters and payload. Return appropriate responses using the **ResponseEntity** class.
- **Serialize/Deserialize Data:** Use libraries like Jackson or Gson to serialize Java objects to JSON or XML, and deserialize incoming requests into Java objects. Spring Boot's default configuration automatically handles serialization and deserialization for JSON payloads.
- **Error Handling:** Implement error handling by creating exception classes and using the **@ExceptionHandler** annotation to define methods that handle specific exceptions. Return appropriate error responses with suitable HTTP status codes.

Coding Examples and Exercises: To practice building RESTful APIs with Spring Boot, here are some coding examples and exercises:
a. Creating a Simple GET Endpoint:

```
@RestController
@RequestMapping("/api")
public class MyController {

   @GetMapping("/resource")
   public ResponseEntity<String> getResource() {
      String resource = "This is the resource content.";
      return ResponseEntity.ok(resource);
   }
}
```

Handling POST Requests:

```
@PostMapping("/resource")
public ResponseEntity<String> createResource(@RequestBody ResourceData resourceData) {
   // Process the resourceData and save it
   return ResponseEntity.created(URI.create("/api/resource/" + resourceData.getId())).build();
}
```

Error Handling:

```
@ControllerAdvice
public class GlobalExceptionHandler {
```

```
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleResourceNotFound(ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGenericException(Exception ex) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("An error occurred.");
    }
}
```

These examples demonstrate the basic concepts of request mapping, serialization, and error handling in building RESTful APIs with Spring Boot.

Conclusion: Building RESTful APIs with Spring Boot is a fundamental aspect of modern application development. By following RESTful principles and leveraging the capabilities of Spring Boot, developers can design and develop robust APIs. The provided code samples and exercises serve as a starting point to practice and gain hands-on experience in building RESTful APIs using Spring Boot. Remember to refer to the official Spring Boot documentation and explore additional features and techniques to enhance your API development skills.

**Using External Configurations in Spring Boot Applications**

Introduction: In Spring Boot applications, it is essential to externalize application properties and manage configurations separately from the codebase. This allows for flexibility, ease of configuration management, and the ability to customize the application's behavior in different environments. In this discussion, we will explore the need for external configurations, explain how to use them in Spring Boot using property files, YAML files, and environment-specific profiles, and demonstrate their implementation in a Spring Boot application.

- Need for Externalizing Application Properties and Managing Configurations: Externalizing application properties offers several benefits:
- Configuration Flexibility: By externalizing properties, you can modify the application's behavior without changing the source code. This is useful for different environments, such as development, testing, and production, where configurations may vary.

- Separation of Concerns: External configurations separate the concerns of application logic and configuration. Developers can focus on writing code while operations teams can handle configuration changes independently.
- Security and Confidentiality: Sensitive information such as database credentials or API keys can be stored securely in external configurations, reducing the risk of accidental exposure.
- Easy Maintenance: External configurations allow for easy and centralized management of application properties. Changes can be made without rebuilding and redeploying the application.
- Using External Configurations in Spring Boot: Spring Boot supports various external configuration options:

**Property Files:** Property files, typically in **.properties** format, allow you to define key-value pairs of configuration properties. They can be placed in the classpath or at specific locations specified by the Spring Boot convention.

For example, create a **application.properties** file:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=dbuser
spring.datasource.password=dbpass
```

**YAML Files:** YAML files offer a more expressive and hierarchical configuration format. They are often preferred for complex configurations. Create a **application.yaml** file:

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: dbuser
    password: dbpass
```

**Environment-Specific Profiles:** Spring Boot allows defining environment-specific configurations by using profiles. Profiles can be activated based on the environment, system properties, or command-line arguments.

Create a **application-{profile}.properties** or **application-{profile}.yaml** file for each profile. For example, create **application-dev.properties** for the "dev" profile.

- Loading and Using External Configurations in a Spring Boot Application: Spring Boot automatically loads external configurations based on the configured locations and

profiles. To access these configurations in your application, you can use the **@Value** annotation or the **@ConfigurationProperties** annotation.
Using **@Value** annotation:

```
@RestController
public class MyController {

   @Value("${spring.datasource.url}")
   private String databaseUrl;

   @GetMapping("/database-url")
   public String getDatabaseUrl() {
      return databaseUrl;
   }
}
```

In this example, the **@Value** annotation is used to inject the value of **spring.datasource.url** from the external configuration file into the **databaseUrl** variable.

b. Using **@ConfigurationProperties** annotation:

```
@Configuration
@ConfigurationProperties("spring.datasource")
public class DataSourceProperties {
   private String url;
   private String username;
   private String password;

   // ... getters and setters

   @Bean
   public DataSource dataSource() {
      // Use the configured properties to create and return the DataSource
   }
}
```

In this example, the **@ConfigurationProperties** annotation is used to bind the properties under the **spring.datasource** prefix from the external configuration file to the **DataSourceProperties** class. The **dataSource()** method creates and returns a **DataSource** bean using the configured properties.

- Coding Examples and Exercises: To practice working with external configurations, consider the following examples and exercises:
- Define a property in an external configuration file and inject it into a controller using the **@Value** annotation.

```
@Value("${myapp.api-key}")
private String apiKey;
```

Create multiple profiles (e.g., "dev" and "prod") and configure different properties for each profile.

```
# application-dev.properties
myapp.api-key=dev-api-key

# application-prod.properties
myapp.api-key=prod-api-key
```

Use the **@ConfigurationProperties** annotation to bind an external configuration to a Java class and access the properties.

```
@Configuration
@ConfigurationProperties("myapp")
public class MyAppProperties {
   private String apiKey;
   // ... getters and setters
}
```

**Conclusion**: Using external configurations in Spring Boot applications provides flexibility, separation of concerns, and easy management of application properties. By leveraging property files, YAML files, and environment-specific profiles, developers can easily customize their applications' behavior in different environments. The provided code samples serve as starting points to gain hands-on experience with external configurations, and by exploring further, you can unlock the extensive features and possibilities offered by Spring Boot.

DAY 4

- **Revision of Days 1 - 3**
- **Practical Session on RESTful APIs and other concepts**