Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Spring Boot

Study

ARNIKA PATEL
CSE, PIT
Parul University

6.1 Introduction to Spring Boot

- Spring Boot is a Java framework that makes it easier to create and run Java applications. It simplifies the configuration and setup process, allowing developers to focus more on writing code for their applications.
- **Spring Boot** is an open-source Java framework used to create a Micro Service.
- Spring boot is developed by Pivotal Team, and it provides a faster way to set up and an easier, configure, and run both simple and web-based applications.
- It is a combination of Spring Framework and Embedded Servers.
- The main goal of Spring Boot is to reduce development, unit test, and integration test time and in Spring Boot, there is no requirement for XML configuration.

- **Features of Spring Boot:** Spring Boot is built on top of the conventional Spring framework, providing all the features of Spring while being significantly easier to use. Here are its key features:

  1. Auto-Configuration: Spring Boot **eliminates the need for heavy XML configuration**, which is common in traditional Spring MVC projects. Instead, everything is auto-configured. Developers only need to add the appropriate configuration to utilize specific functionalities. For example, if we want to use Hibernate (ORM), we can simply add the **@Table** annotation to our model/entity class and the **@Column** annotation to map it to database tables and columns. By 2025, Spring Boot's auto-configuration has become even smarter, utilizing AI-driven optimizations to further reduce manual setup.

  2. Easy Maintenance and Creation of REST Endpoints: Creating REST APIs is incredibly easy in Spring Boot. With annotations like **@RestController** and **@RequestMapping**, developers can quickly define endpoints. By 2025, Spring Boot has introduced even more advanced annotations like **@GetMapping, @PostMapping, @PutMapping,** and **@DeleteMapping**, making REST API development more intuitive and efficient.

  3. Embedded Tomcat Server: Unlike traditional Spring MVC projects, where you need to manually install and configure a Tomcat server, Spring Boot comes with an embedded Tomcat server. This allows applications to be hosted directly without additional setup. By 2025, Spring Boot also supports other embedded servers like Jetty and Undertow, giving developers more flexibility based on their application requirements.

  4. Easy Deployment: Spring Boot simplifies deployment by allowing applications to be packaged as JAR or WAR files. These files can be directly deployed to a Tomcat server or cloud environment. By 2025, Spring Boot has enhanced its deployment capabilities with

seamless integration into Kubernetes and Docker, making it easier to deploy and scale applications in cloud-native environments.

5. Microservice-Based Architecture: Spring Boot is designed for microservices, which are small, independent modules that focus on a single functionality. For example, in a hospital management system, you might have separate services for patient registration, billing, and database management. By 2025, Spring Boot has further optimized its microservice support with features. In a monolithic system, all features are bundled into a single codebase, making it difficult to maintain and scale. In a microservice-based system, each feature is divided into smaller, independent services. This modular approach makes the system easier to maintain, debug, and deploy. Each service can be built using different technologies suited to its specific requirements.

- **Reactive Programming:** For building non-blocking, scalable applications.
- **Distributed Tracing**: For monitoring and debugging microservices.
- **Service Mesh Integration:** For better communication between microservices.

Spring Boot Architecture

To understand the architecture of Spring Boot, let's examine its different layers and components.

Layers in Spring Boot

Spring Boot follows a layered architecture with the following key layers:

- **Client Layer:**
  - This represents the external system or user that interacts with the application by sending HTTPS requests.
- **Controller Layer (Presentation Layer):**
  - Handles incoming HTTP requests from the client.
  - Processes the request and sends a response.
  - Delegates business logic processing to the Service Layer.
- **Service Layer (Business Logic Layer)**:
  - Contains business logic and service classes.
  - Communicates with the Repository Layer to fetch or update data.
  - Uses Dependency Injection to get required repository services.
- **Repository Layer (Data Access Layer)**:
  - Handles CRUD (Create, Read, Update, Delete) operations on the database.

- Extends Spring Data JPA or other persistence mechanisms.

- **Model Layer (Entity Layer):**
  - Represents database entities and domain models.
  - Maps to tables in the database using JPA/Spring Data.
- **Database Layer:**
  - The actual database that stores application data.
  - Spring Boot interacts with it through JPA/Spring Data.

6.2 Spring Boot- Dependency Injection

- Dependency injection(DI) is a design pattern that helps eliminate the dependencies within a class, making it independent of the specific objects it relies on.
- Dependency injection promotes loose coupling and easier testing. By implementing dependency injection, we can adhere to the principles of SOLID, particularly dependency inversion.
- In the context of software development and the Java programming language, there are several options for implementing dependency injection (DI) utilizing the Spring framework.
- **Constructor Injection**
  - In constructor injection, dependencies are provided through a class constructor.
  - The dependencies are explicitly declared as constructor parameters.
  - Once instantiated, the class holds references to these dependencies.
  - Constructor injection ensures that the required dependencies are provided at the time of object creation.
  - It promotes immutability, as dependencies can be declared as final.

    @Service
    public class RoleService {
    private final RoleRepository roleRepository;
    public RoleService(RoleRepository roleRepository) {
    this.roleRepository = roleRepository;
    }
    // ...
    }
  - In this example, the class **RoleService** has a constructor that takes a **RoleRepository** dependency. The dependency is declared as a constructor

parameter, and Spring Boot automatically resolves and injects the **RoleRepository** bean when creating an instance of **RoleService**.

- **Constructor-based Dependency Injection (using Lombok):** Lombok is a library that can help reduce boilerplate code. It provides annotations like @**RequiredArgsConstructor**, which automatically generates a constructor with the required dependencies.

```
@Service
@RequiredArgsConstructor
public class RoleService {
private final RoleRepository roleRepository;
// ...
}
```

- **Setter Injection**
  - Setter injection involves providing dependencies using setter methods.
  - Dependencies are declared as private instance variables and corresponding setter methods are defined.
  - These setters are used to inject the dependencies into the class.
  - Setter injection allows for optional dependencies and the ability to change dependencies at runtime.
- **I**n this example, the **RoleService** class has a setter method annotated with @**Autowired**. When the Spring Boot application context initializes, it automatically identifies the **RoleRepository** bean and injects it into the setRoleRepository method.

```
@Service
public class RoleService {
private RoleRepository roleRepository;
@Autowired
public void setRoleRepository(RoleRepository roleRepository) {
this.roleRepository = roleRepository;
}
// ...
}
```

- **Field Injection**

- Field injection involves injecting dependencies directly into class fields or properties.
- Dependencies are declared as private instance variables with the @**Autowired** annotation.
- The Spring framework uses reflection to directly set the field values.
- Field injection can simplify code and reduce verbosity, but it may hinder testability and make it harder to detect missing dependencies.

```
@Service
public class RoleService {
@Autowired
private RoleRepository roleRepository;
// ...
}
```

- In this example, the **RoleService** class has a field annotated with @**Autowired**. Spring Boot identifies the **RoleRepository** bean and directly injects it into the **roleRepository** field when creating an instance of **RoleService**. However, it's generally recommended to avoid field injection as it can make testing and mocking more challenging.

- **Interface Injection**
  - Interface injection is not directly supported by the Spring framework.
  - It involves implementing an interface that defines setter methods for the dependencies.
  - The implementing class is responsible for providing the necessary dependency injection logic.
  - While it offers flexibility, it can introduce complexity and increase coupling.
  - In a Spring Boot application, we can enable dependency injection by using appropriate annotations such as @**Autowired** and @**Service** to mark the classes where dependencies need to be injected.
  - Additionally, we need to configure the application with annotationslike @**ComponentScan** or @**SpringBootApplication** to allow Spring Boot to scan and discover the beans.

6.3 Spring Boot- AOP

- AOP(Aspect Oriented Programming) breaks the full program into different smaller units. In numerous situations, we need to log, and audit the details as well as need to pay

importance to declarative transactions, security, caching, etc., Let us see the key terminologies of AOP

- **Aspect:** It has a set of APIs for cross-cutting requirements. Logging module is an example of the AOP aspect of logging.
- **Joint Point:** AOP aspect plug in place
- **Advice:** Via this, the actual implementation of code is taken care for the AOP approach. It can be either before/after/after returning/after throwing. In this article let us see the examples related to this.
- **Pointcut:** Set of one or more join points where an advice need to be executed.
- **Introduction:** This is the place where we can add new methods or attributes to the existing classes.
- **Target Object:** One or more aspects will be there to provide advice for the target object.
- **Weaving:** Linking aspects with other application types or objects. It can be done at compile-time/runtime/loadtime.
- In web applications, each of the layers(web, business, and data layer) are responsible for different tasks and they perform these tasks individually. But there are a few **common aspects** that are applied to each layer such as security, caching, validation, etc. These common aspects are known as **Cross-Cutting Concerns**.
- In web applications, each of the layers(web, business, and data layer) are responsible for different tasks and they perform these tasks individually. But there are a few **common aspects** that are applied to each layer such as security, caching, validation, etc. These common aspects are known as **Cross-Cutting Concerns**.

6.4 Spring Boot- Database

- In modern application development, integrating a database is crucial for persisting and managing data.
- Spring Boot simplifies this process by providing seamless integration with various databases through JPA (Java Persistence API) and Hibernate ORM (Object-Relational Mapping).
- **Database Integrations (MySQL and H2)**
- **MySQL:** A widely used relational database management system known for its reliability, performance, and ease of use. It is ideal for production environments where data needs to be persistently stored.
- **H2 Database:** An in-memory database that is extremely fast and useful for development and testing. As it is in-memory, data is lost once the application is stopped. H2 provides

excellent support and allows you to switch between databases with minimal configuration changes.

- **Steps to Integrate MySQL and H2 in a Spring Boot Application**

  Step 1: MySQL Integration

  1. **Add the Dependencies:** Include spring-boot-starter-data-jpa and mysql-connector-java inyour pom.xml.
  2. **Configure MySQL Connection:** Update application.properties with MySQL database configuration.
  3. **Database Initialization (Optional):** Use schema.sql and data.sql files to initialize the database schema and data.

  Step 2: H2 Database Integration

  1. **Add the Dependencies:** Include spring-boot-starter-data-jpa and h2 in your pom.xml.
  2. **Configure H2 Connection:** Update application-dev.properties with H2 database configuration. Enable the H2 console for easy access to the in-memory database via the web interface.
  3. **Auto Schema Generation:** H2 will automatically generate the schema based on JPA entities.

- **Switching Between Databases**
  - Spring Boot makes it easy to switch between MySQL and H2 or other databases by changing the configuration in the application.properties file. This flexibility allows developers to use H2 for local development and MySQL in production without altering the codebase.

6.5 Spring Boot- REST

- **Representational State Transfer (REST)** is a software architectural style that defines a set of constraints for creating web services. **RESTful web services** allow systems to access and manipulate web resources through a uniform and predefined set of stateless operations.
- Unlike **SOAP**, which exposes its own set of operations, RESTful web services rely on simple **HTTP methods** such as GET, POST, PUT, and DELETE.
- **Why Spring Boot?**
- Spring Boot is built on top of Spring Framework, simplifying project setup and configuration. It provides default configurations to avoid boilerplate code and is ideal for beginners looking to work with Spring.

- **Steps to Create a REST API**

  Step 1: Define the Employee Entity

  Step 2: Create a Storage Class

  Step 3: Create the DAO Class

  Step 4: Create the Controller

  Step 5: Run the Application

References:

1. **Book Reference**

   Jim Farley, William Crawford, David Flanagan. Java Enterprise in a Nutshell, O'Reilly

2. **Book Reference**

   Rocha, R., Purificação, J. (2018). Java EE 8 Design Patterns and Best Practices: Build Enterprise-ready Scalable Applications with Architectural Design Patterns. Germany: Packt Publishing..

3. **Website Reference**

   https://www.scribd.com/document/268349254/Java-8-Programming-Black-Book .

4. **Sources**

   https://developers.redhat.com/topics/enterprise-java

5. **Article**

   https://www.researchgate.net/publication/276412369_Advanced_Java_Programming

Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

NAAC GRADE A++

https://paruluniversity.ac.in/