Spring Framework

What is Spring Framework ?

- popular java framework for creating web applications
- popular for implementing microservice architecture

Spring Framework
- provides comprehensive infrastructure support for building Java applications.
- Initially, it was focused on dependency injection and lightweight application development.
- Over time, it evolved into a powerful ecosystem with modules supporting various needs,
- such as web applications (Spring MVC), data access (Spring Data), and integration with other systems.

Spring Boot builds on top of the Spring framework,
- providing a streamlined approach to application development with less configuration.
- It offers pre-configured templates and embedded servers,
- which significantly reduce boilerplate code and setup time.
- Spring Boot's auto-configuration and embedded server features make it highly suited for microservices and standalone applications.

Difference b/w Microservice Architecture vs Monolithic Architecture

Modules of Spring Framework

Spring is divided into modules
- Core - IoC, Dependency Injection, Internationalisation, AOP (Aspect Oriented Programming)
- Data Access - JDBC, JTA (Java Transaction API), JPA(Java Persistance API)
- Web - Spring MVC, Servlet API, Reactive API
- Testing - Unit testing, Mock, Test fixtures, Caching
- Integration - JMX(Java Management Extension), RMI(Remote Method Invocation), JMS(Java Messaging Service)

Spring Projects
- Boot
- Mobile
- Cloud

Getting Started with Maven to implement first Spring Boot Project

Maven is a powerful build automation and dependency management tool primarily used for Java projects.
It simplifies project setup and management by
- handling the compilation,
- testing, and
- packaging processes,
- as well as managing dependencies with external libraries.

Key Concepts of Maven

Project Object Model (POM):
The POM (pom.xml) is the core of a Maven project. It is an XML file that contains information about the project, such as dependencies, plugins, goals, and project-specific configurations.
POMs make it easy to manage and maintain project dependencies and build settings.

Dependencies:
Maven manages project dependencies automatically. Developers list required libraries in the POM, and Maven downloads them from a central repository.
This helps avoid "JAR Hell" by automatically handling version conflicts and updating libraries as needed.
Build Lifecycle:

Maven uses a predefined build lifecycle with phases like clean, compile, test, package, and install.
Each phase represents a step in the build process and can be customized with plugins to perform tasks (e.g., compiling code, running tests, packaging the application).
Repositories:

Central Repository: Maven Central is a public repository where commonly used libraries and plugins are hosted.
Local Repository: Each machine has a local repository where downloaded dependencies are cached for reuse.
Remote Repositories: Organizations can host private repositories to manage their internal libraries or proprietary dependencies.
Plugins:

Maven's functionality is extended through plugins, which are tools or code modules that handle specific tasks in the build lifecycle.
Popular plugins include the Compiler Plugin for compiling Java code, the Surefire Plugin for running tests, and the Assembly Plugin for creating JAR/WAR packages.
Archetypes:

Maven provides archetypes, which are templates for creating new projects with predefined structure and dependencies, such as a web application archetype or a simple Java project archetype.

Example of a Basic POM File:

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.example</groupId>
<artifactId>my-app</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencies>
   <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-context</artifactId>
         <version>5.3.13</version>
   </dependency>
```

```xml
    <dependency>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
    </dependency>

</dependencies>
```

Advantages of Using Maven
- Dependency Management: Automates the process of resolving and updating dependencies.
- Consistent Builds: Standardized build lifecycle helps maintain consistency across environments.
- Reusable Builds: POM files can be shared, making it easy to replicate and configure builds across teams.
- Extensibility: Plugins extend Maven's capabilities, from code analysis to packaging.

Maven Build Life cycle

Maven Commands
- clean
- deploy
- package
- validate
- test
- compile

```
mvn clean install
mvn compile
mvn package
mvn test
mvn spring-boot:run
```

Install in Mac - brew install maven
Install in Windows - https://phoenixnap.com/kb/install-maven-windows

Fundamental Concepts of Spring Core
1. Inversion of Control (IoC)
- IoC is a design principle
- where the control of creating and managing dependencies is shifted from the application code to a container or framework (like Spring).
- In Spring, the IoC container instantiates and manages beans, handling their dependencies as per configuration.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code.

To enable this, frameworks use abstractions with additional behavior built in.
If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.

The advantages of this architecture are:
- decoupling the execution of a task from its implementation
- making it easier to switch between different implementations
- greater modularity of a program
- greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as:
Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

2. Dependency Injection (DI)
- Dependency Injection is a technique where an object's dependencies are injected rather than instantiated within the object itself.

Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.
Connecting objects with other objects, or "injecting" objects into other objects, is done by an assembler rather than by the objects themselves.

Here's how we would create an object dependency in traditional programming:
```
public class Store {
        private Item item;
        public Store() {
                item = new ItemImplentation1();
        }
}
```

In the example above, we need to instantiate an implementation of the Item interface within the Store class itself.
By using DI, we can rewrite the example without specifying the implementation of the Item that we want:

```
public class Store {
   private Item item;
   public Store(Item item) {
      this.item = item;
   }
}

Store obj = new Store(new ItemImpl1());
Store obj = new Store(new ItemImpl2());
Store obj = new Store(new ItemImpl3());
Store obj = new Store(new ItemImpl4());
```

Spring supports multiple types of DI:
- Constructor Injection: Dependencies are provided through a class constructor.
- Setter Injection: Dependencies are provided through setter methods.
- Field Injection: Dependencies are injected directly into fields without using constructors or setters (not recommended due to testing limitations).

3. Annotations for Java-Based Configuration
- Annotations simplify bean configuration in Spring by reducing the need for XML files.

@Component: Marks a class as a Spring-managed bean, making it available for DI. Spring automatically detects it during classpath scanning.
@Autowired: Enables automatic injection of dependencies into a Spring-managed bean. Spring will look for a matching bean in the IoC container and inject it.
@ComponentScan: Scans the specified package for classes annotated with @Component, @Service, @Repository, etc., and registers them as beans in the application context.

4. Automated Java-Based Configuration
- Constructor Injection:
  - Involves defining dependencies in a constructor. It is the preferred method, as it makes dependencies required at object creation.

- Field Injection:
  - Injects dependencies directly into a field, typically with @Autowired on the field itself.
  - Though convenient, it's discouraged in favor of constructor injection, as it complicates testing.

- Setter Injection:
  - Dependencies are provided via setter methods. This allows optional dependencies.

Each approach to dependency injection has its pros and cons, but constructor injection is generally preferred due to its alignment with immutability and testability.

Project - Ticket Booking Application
- Spring Boot - Dependency Injection
- MySQL as database
- JPA - Java Persistance API
- ORM - Object Relational Mapping
- Maven - Build Tool

CRUD Operations
Use Postman for Testing