Spring Framework Assessment - 13

Section I [Multiple Choice Questions] : 1 Marks Each

1. What is the core concept behind the Spring Framework?*

a) Servlets

b) Inversion of Control (IoC)

c) JSP

d) JDBC

2. Which are the IoC containers in Spring?

A. BeanFactory, BeanContext, IocContextFactory

B. BeanFactory, ApplicationContext

C. BeanFactory, ApplicationContext, BeanContext

D. BeanFactory, ApplicationContext, IocContextFactory

3 Which are the different modes of autowiring?

A. byName, byContent, setter, autodetect

B. byName, byContent, constructor, autodetect

C. byName, byType, constructor, autodetect

D. byName, byType, constructor, autocorrect

4. What is Dependency Injection (DI) in Spring Framework?*

a) A design pattern

b) A way to achieve Inversion of Control

c) A module in Spring

d) A Spring annotation

5. Which of the following is NOT a module in the Spring Framework?*

a) Spring AOP

b) Spring MVC

c) Spring JPA

d) Spring JDBC

6. What does the Spring @Autowired annotation do?*

a) Creates a new instance of a bean

b) Performs dependency injection

c) Initialises the application context

d) Specifies a bean's lifecycle method

7. Which annotation is used to inject a dependency in Spring?*

a) @Inject

b) @Autowired

c) @Resource

d) All of the above

8. Can we inject value and ref both together in a bean?*

True

False

9. Which annotation is used to make a Java class serve RESTful requests?*

a) @RestController

b) @Service

c) @Controller

d) @Repository

10. Which of the following method can be used to used to instantiate a method?

A. static factory method

B. default-init method

C. destroy method

D. lazy-init method

11. What does the @Required annotation do?*

a) Makes a bean's property mandatory

b) Indicates that a field should be autowired

c) Specifies a bean's lifecycle method

d) Specifies a bean's scope

12. Class which declares a number of overloaded update() template methods to control the overall update process.

a) org.springframework.jdbc.core.JdbcTemplate

b) org.springframework.jdbc.core.*

c) org.springframework.jdbc.*

d) none of the mentioned

13. Spring HibernateTemplate can simplify your DAO implementation by managing sessions and transactions for you.

True False

14. DAO methods require access to the session factory, which can be injected:-

a) a setter method

b) constructor argument

c) none of the mentioned

d) all of the mentioned

15. Which of the following is NOT an advice type in Spring AOP?*

a) Before

b) After

c) Across

16. Design pattern implemented by Dispatcher Servlet.

a) jsp

b) tiles

c) front controller

d) none of the mentioned

17. To load root application context at the start up.

a) ContextListener

b) ContextLoader

c) ContextLoaderListener

d) ContextEventListener

18. Which annotation denotes a transaction boundary in Spring?*

a) @Boundary

b) @Transaction

c) @Transact

d) @Transactional

19. Which of the following is not a type of metadata provided by Spring?*

a) Annotation-based

b) Java-based

c) XML-based

d) YAML-based

20. Which of the following is true about the @Controller annotation in Spring?*

a) It is used to indicate a RESTful web service

b) It is used to mark classes as Spring beans

c) It is used to indicate a class as a Spring MVC controller

d) It is used to configure beans

Section II [Descriptive Questions]: 5 Marks Each

1. What is Spring Framework ? List the benefits of Using Spring Framework.

The Spring Framework is an open-source, comprehensive framework for building Java-based enterprise applications. It provides a programming and configuration model to simplify application development by handling the infrastructure and supporting modern development features. It supports Aspect-Oriented Programming (AOP) to modularize cross-cutting concerns such as logging, transaction management, and security.

Benefits of Using Spring Framework

1) Simplified Development: Spring reduces boilerplate code using DI and AOP, speeding up development and reducing errors.

2) Loose Coupling: Dependency Injection promotes loose coupling between components, making code easier to maintain and test.

3) Modular Architecture: Developers can pick and choose required modules without adopting the entire framework, improving flexibility and efficiency.

4) Integration Support: Spring supports integration with various technologies like JDBC, JMS, JPA, Hibernate, and more, easing enterprise integration.

5) Scalability: Its lightweight and modular nature makes Spring highly scalable for large applications.

6) Transaction Management: Provides consistent and declarative transaction management across different environments.

7) Aspect-Oriented Programming: Separates cross-cutting concerns for cleaner and more modular business logic.

8) Security and Testing Support: Includes subprojects like Spring Security and supports easy testing due to decoupled components.

2. Explain Dependency Injection. How Can We Inject Beans in Spring? Which Is the Best Way of Injecting Beans and Why? [With a Sample]

Dependency Injection (DI) is a design pattern central to the Spring Framework that implements Inversion of Control (IoC). Instead of classes creating their dependencies, Spring manages these dependencies and injects them as needed. This results in loose coupling, making code more modular, testable, and flexible.

Benefits of Dependency Injection in Spring

1) Loose Coupling: Classes are independent of their dependencies, making the system more maintainable and scalable.
2) Easier Testing: Dependencies can be mocked or stubbed easily for unit tests.
3) Flexibility: Swap implementations without code changes, just by changing configuration.
4) Central Configuration: All dependencies are managed in one place (XML, annotations, or Java config).
5) Cleaner Code: Keeps object creation separate from business logic, making the code easier to read and manage.

How to Inject Beans in Spring

There are three primary ways to inject beans:

1. Constructor Injection - Dependencies are provided through the class constructor.

```
@Component
public class EmployeeService {
    private final EmployeeRepository repository;
    @Autowired
    public EmployeeService(EmployeeRepository repository) {
        this.repository = repository;
    }
}
```

Spring will automatically inject the bean EmployeeRepository into EmployeeService via the constructor.

2. Setter Injection - Dependencies are provided through public setter methods.

```
@Component
public class EmployeeService {
    private EmployeeRepository repository;
```

```
    @Autowired
    public void setRepository(EmployeeRepository repository) {
        this.repository = repository;
    }
}
```

3. Field Injection - Dependencies are injected directly into fields.

```
@Component
public class EmployeeService {
    @Autowired
    private EmployeeRepository repository;
}
```

Injection Methods:

Using annotations like @Autowired, @Resource, or @Inject

Via XML configuration

Using Java-based @Configuration and @Bean methods for explicit Java config

Which Is the Best Way?

A) Constructor Injection is recommended for most use cases:

1) Ensures all required dependencies are available at object creation (no partial objects)

2) Fields can be made final (improving immutability and thread safety)

3) Encourages immutable designs and better testability

4) Easier to write unit tests, as all dependencies are explicit in the constructor

```
@Component
public class EmailService {
    public void sendEmail(String message) { /* send email */ }
}

@Component
public class NotificationService {
    private final EmailService emailService;

    @Autowired
```

```
    public NotificationService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void notify(String message) {
        emailService.sendEmail(message);
    }
}
```

B) Setter injection is best when you have optional dependencies.

```
@Component
public class NotificationService {
    private EmailService emailService;

    @Autowired
    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }
}
```

C) Field injection is concise, but not recommended for production as it hinders testing and you lose immutability.

```
@Component
public class NotificationService {
    @Autowired
    private EmailService emailService;
}
```

3. Define Auto Wiring with a suitable sample. Explain Scopes of Bean.

Autowiring is a feature in the Spring Framework that enables automatic dependency injection. Instead of explicitly configuring bean dependencies (e.g., via XML or setter methods), Spring automatically detects and wires the collaborating beans by inspecting the Spring container. This reduces the need for explicit configuration and makes development easier.

Modes of Autowiring in Spring

1) no (default) - No autowiring is performed; dependencies must be explicitly wired.

2) byName - Spring injects a bean whose name matches the property name in the dependent bean.

3) byType - Spring injects a bean whose type matches the property type in the dependent bean. Fails if multiple beans of the same type exist.

4) constructor - Injects dependencies via constructor arguments by matching types.

5) autodetect - Attempts constructor autowiring first; if none found, falls back to byType.

Spring manages beans in different scopes which define the lifecycle and visibility of beans in the container:

a) Singleton - Only one shared instance of the bean is created for the entire Spring container.

b) Prototype - A new instance of the bean is created every time it is requested.

c) Request - One instance per HTTP request (used in web applications).

d) Session - One instance per HTTP session (used in web apps).

e) Application - One instance per ServletContext (application-wide scope).

f) Websocket - One instance per WebSocket lifecycle.

4. Explain AOP Module. Create a CurrencyConvertor Business bean containing Business Operations. Demonstrate the usage of AOP.

AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns—such as logging, security, transaction management, or caching—from business logic. Instead of scattering this common code across many classes, AOP enables you to define them separately and apply them declaratively to targeted parts of the program.

1). CurrencyConvertor Business Bean

```
package com.example.aopdemo;
public class CurrencyConvertor {
    public double dollarToINR(double amount) {
        return amount * 74.85; // Fixed conversion rate
    }
    public double inrToDollar(double amount) {
```

```
        return amount * 0.013; // Fixed conversion rate
    }
}
```

2) Logging Aspect

```
package com.example.aopdemo;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    // Advice executed before any method of CurrencyConvertor
    @Before("execution(* com.example.aopdemo.CurrencyConvertor.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("[AOP Before] Invoking method: " + joinPoint.getSignature().getName());
    }

    // Advice executed after method returns successfully
    @AfterReturning(pointcut = "execution(* com.example.aopdemo.CurrencyConvertor.*(..))",
returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        System.out.println("[AOP AfterReturning] Method " + joinPoint.getSignature().getName() +
" returned: " + result);
    }
}
```

3. Spring Configuration to Enable AOP

java

```
package com.example.aopdemo;

import org.springframework.context.annotation.ComponentScan;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan("com.example.aopdemo")
@EnableAspectJAutoProxy  // Enables Spring AOP proxy support
public class AppConfig {
}
```

4. Main Application to Test AOP

java

```
package com.example.aopdemo;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class MainApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        CurrencyConvertor convertor = context.getBean(CurrencyConvertor.class);
        double usdAmount = 100.0;
        double inrAmount = convertor.dollarToINR(usdAmount);
        System.out.println("USD " + usdAmount + " = INR " + inrAmount);
        double inrAmountInput = 15000.0;
        double usdAmountConverted = convertor.inrToDollar(inrAmountInput);
        System.out.println("INR " + inrAmountInput + " = USD " + usdAmountConverted);
        context.close();
    }
}
```

What this demonstrates:

1) The LoggingAspect automatically logs before and after executions of any method in CurrencyConvertor.

2) You do not need to put logging code inside CurrencyConvertor methods; cross-cutting concerns are handled separately.

@Aspect defines the aspect class.

@Before and @AfterReturning define when advice runs relative to method execution.

@EnableAspectJAutoProxy activates Spring's proxy-based AOP.

5. Illustrate a sample to integrate Spring with ORM

1. Define an Entity (e.g., Product.java)

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "products")
public class Product {
    @Id
    private int id;
    private String name;

    // Constructors
    public Product() {}
    public Product(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters/setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

2. Create DAO Layer (ProductDao.java)

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
```

```
import jakarta.transaction.Transactional;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public class ProductDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void save(Product product) {
        entityManager.persist(product);
    }

    public Product findById(int id) {
        return entityManager.find(Product.class, id);
    }

    @Transactional
    public void update(Product product) {
        entityManager.merge(product);
    }

    @Transactional
    public void delete(Product product) {
        entityManager.remove(product);
    }

    public List<Product> findAll() {
        return entityManager.createQuery("FROM Product", Product.class).getResultList();
    }
}
```

3. Configure Spring Context (Java-based configuration example)

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = "com.example")  // your base package here
public class PersistenceJPAConfig {

   @Bean
   public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
      LocalContainerEntityManagerFactoryBean em
        = new LocalContainerEntityManagerFactoryBean();
      em.setDataSource(dataSource());
      em.setPackagesToScan("com.example.model"); // Package where Entity classes are

      HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
      em.setJpaVendorAdapter(vendorAdapter);
      em.setJpaProperties(additionalProperties());

      return em;
   }
```

```
    @Bean
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver"); // example H2 DB driver
        dataSource.setUrl("jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager transactionManager
          = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(
          entityManagerFactory().getObject() );
        return transactionManager;
    }

    @Bean
    public PersistenceExceptionTranslationPostProcessor exceptionTranslation(){
        return new PersistenceExceptionTranslationPostProcessor();
    }

    Properties additionalProperties() {
        Properties properties = new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "update");  // for auto schema update
        properties.setProperty("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
        properties.setProperty("hibernate.show_sql", "true");
        return properties;
    }
}
```

4. Service Layer (optional, cleaner approach)

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
public class ProductService {

    @Autowired
    private ProductDao productDao;

    @Transactional
    public void save(Product product) {
        productDao.save(product);
    }

    public Product findById(int id) {
        return productDao.findById(id);
    }

    @Transactional
    public void update(Product product) {
        productDao.update(product);
    }

    @Transactional
    public void delete(Product product) {
        productDao.delete(product);
    }

    public List<Product> findAll() {
        return productDao.findAll();
    }
```

```
}
```

5. Usage in Main (Spring Boot or Manual Context)

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class MainApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(PersistenceJPAConfig.class);

        ProductService service = context.getBean(ProductService.class);

        Product p1 = new Product(1, "Mobile Phone");
        service.save(p1);
        System.out.println("All products:");
        service.findAll().forEach(p -> System.out.println(p.getId() + ": " + p.getName()));
        context.close();
    }
}
```

6. Illustrate a Sample to Integrate Spring With Web

1) Spring MVC Configuration (Java-based)

```
java
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.example.webapp")
public class WebConfig implements WebMvcConfigurer {
```

```
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // Using JSP views located under /WEB-INF/views/
        registry.jsp("/WEB-INF/views/", ".jsp");

        // Alternatively, to use Thymeleaf or other view resolvers, configure here
    }
}
```

2) Controller Class

java

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

    @GetMapping("/greeting")
    public String greeting(@RequestParam(name="name", required=false, defaultValue="World")
String name, Model model) {
        model.addAttribute("name", name);
        return "greeting"; // returns the logical view name "greeting"
    }
}
```

3) JSP View (/WEB-INF/views/greeting.jsp)

text

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
    <title>Spring MVC Greeting</title>
```

```
</head>
<body>
    <h1>Hello, ${name}!</h1>
</body>
</html>
```

4) web.xml Configuration

xml

```
<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```