Spring Framework & Spring

1. Inversion of Control (IoC)

Definition:

IoC is a design principle where the control of object creation and lifecycle is transferred from the application to a framework (like Spring). Instead of manually creating dependencies, the framework manages them.

Example:

Without IoC:

```
public class UserService {
    private UserRepository userRepository = new UserRepository(); // Tight coupling
}
```

With IoC (Spring manages `UserRepository`):

```
public class UserService {
    private UserRepository userRepository; // Injected by Spring
}
```

- **Spring IoC Container:** Manages beans (objects) defined in the configuration.

2. Dependency Injection (DI)

Definition:

DI is a technique where dependencies are provided to a class rather than the class creating them. It helps achieve **loose coupling**.

Types of DI:

1. **Constructor Injection** (Recommended)

```
@Service
public class UserService {
    private final UserRepository userRepository;

    @Autowired // Optional in Spring 4.3+
```

```
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

2. **Setter Injection**

```
public class UserService {
    private UserRepository userRepository;

    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

3. **Field Injection** (Not recommended)

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
}
```

3. Spring Basics

- **Spring Framework:** A lightweight, modular framework for building enterprise Java applications.
- **Core Features:**
  - Dependency Injection (DI)
  - Aspect-Oriented Programming (AOP)
  - Spring MVC (Web Framework)
  - Data Access (JDBC, JPA, etc.)
- **Spring Modules:**
  - **Core Container (IoC, DI)**
  - **Spring MVC (Web)**
  - **Spring Data (JPA, JDBC)**
  - **Spring Security**
  - **Spring Boot (Auto-configuration)**

4. Spring Boot Introduction

Philosophy:

- **Convention over Configuration:** Reduces boilerplate code.
- **Standalone Applications:** Embedded servers (Tomcat, Jetty).
- **Auto-configuration:** Automatically configures beans based on dependencies.
- **Starter Dependencies:** Simplifies dependency management.

Example:

A traditional Spring app requires XML configuration, whereas Spring Boot auto-configures most things.

5. Dependency Management with Maven

- **Maven:** A build tool that manages dependencies.
- `pom.xml` **(Project Object Model):** Defines dependencies and plugins.

Example:

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

- **Spring Boot Starters:**
  - `spring-boot-starter-web` (Web apps)
  - `spring-boot-starter-data-jpa` (JPA & Hibernate)
  - `spring-boot-starter-test` (Testing)

6. Creating Your First Spring Boot Application

Using Spring Initializr

1. Go to https://start.spring.io

2. Select:

- **Project:** Maven
- **Language:** Java
- **Spring Boot Version:** Latest
- **Dependencies:** Spring Web

3. Generate & import into IDE.

Main Class:

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

7. Bean Scope

- Defines the lifecycle and visibility of a bean.
- **Common Scopes:**
  - **Singleton (Default):** One instance per Spring container.

```
@Bean
@Scope("singleton")
public MyBean myBean() { return new MyBean(); }
```

  - **Prototype:** New instance every time.

```
@Bean
@Scope("prototype")
public MyBean myBean() { return new MyBean(); }
```

  - **Request, Session, Application (Web-aware scopes)**

8. Loose Coupling & Tight Coupling

- **Tight Coupling:** Classes are highly dependent on each other.

```
class UserService {
    private UserRepository repo = new UserRepository(); // Hard dependency
}
```

- **Loose Coupling:** Classes depend on abstractions (interfaces).

```
class UserService {
    private final UserRepository repo;
    public UserService(UserRepository repo) { this.repo = repo; }
}
```

9. Spring Boot Auto-configuration

- Automatically configures beans based on:
  - Classpath dependencies.
  - Existing beans.
  - Property settings (`application.properties`).
- **Example:** If `spring-boot-starter-data-jpa` is present, Spring Boot auto-configures `DataSource`, `EntityManager`, etc.

10. Configuration with Annotations

- `@SpringBootApplication`: Combines:
  - `@Configuration` (Defines beans)
  - `@EnableAutoConfiguration` (Auto-configures beans)
  - `@ComponentScan` (Scans for components)
- `@Configuration`: Marks a class as a source of bean definitions.

java

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() { return new MyService(); }
}
```

- `@Bean`: Indicates that a method produces a Spring-managed bean.

11. Spring JDBC

- Simplifies JDBC operations.
- `JdbcTemplate`: Executes SQL queries.

```
@Repository
```

```
public class UserRepository {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public List<User> findAll() {
        return jdbcTemplate.query("SELECT * FROM users", new UserRowMapper());
    }
}
```

12. Spring Boot Logging

- **Default:** Uses Logback with SLF4J.
- **Log Levels:** `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`
- **Configuration (`application.properties`):**
  properties

```
logging.level.root=WARN
logging.level.com.example=DEBUG
logging.file=logs/app.log
```

- **Custom Logging:**

```
@RestController
public class MyController {
    private static final Logger log = LoggerFactory.getLogger(MyController.class);

    @GetMapping("/")
    public String home() {
        log.info("Request received");
        return "Hello";
    }
}
```

Summary

Concept	Key Takeaway
IoC	Framework manages object lifecycle.
DI	Dependencies are injected rather than created.

Concept	Key Takeaway
Spring Boot	Auto-configuration, embedded server, starters.
Bean Scope	Singleton (default), prototype, request, session.
Loose Coupling	Depend on interfaces, not concrete classes.
`@SpringBootApplication`	Combines config, auto-config, component scan.
Spring JDBC	`JdbcTemplate` simplifies database operations.
Logging	SLF4J with Logback, configurable via properties.

`@Service`, `@Component`, `@Repository`, `@Autowired`

1. `@Component`

Definition

- A generic stereotype annotation indicating that a class is a **Spring-managed component** (Bean).
- Used for **auto-detection** and **dependency injection** via classpath scanning.

When to Use?

- For general-purpose Spring beans (not specifically a service, repository, or controller).

Example

```
@Component
public class EmailService {
    public void sendEmail(String message) {
        System.out.println("Email sent: " + message);
```

```
    }
}
```

- Spring will detect `EmailService` and register it as a bean.

2. @Service

Definition

- A specialization of `@Component` used for **business logic** (service layer).
- Improves code readability (makes it clear that the class is a service).

When to Use?

- For classes that contain **business logic** (e.g., `UserService`, `PaymentService`).

Example

```
@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserById(Long id) {
        return userRepository.findById(id);
    }
}
```

3. @Repository

Definition

- A specialization of `@Component` used for **data access** (DAO layer).
- Provides **database exception translation** (converts JDBC exceptions into Spring's `DataAccessException`).

When to Use?

- For classes that interact with databases (e.g., `UserRepository`, `OrderRepository`).

Example

```
@Repository
public class UserRepository {
    private final JdbcTemplate jdbcTemplate;

    public UserRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public User findById(Long id) {
        return jdbcTemplate.queryForObject(
            "SELECT * FROM users WHERE id = ?",
            new Object[]{id},
            (rs, rowNum) -> new User(rs.getLong("id"), rs.getString("name"))
        );
    }
}
```

4. `@Autowired`

Definition

- Used for **automatic dependency injection** (Spring injects the required bean).
- Can be applied to:
  - **Constructor (Recommended)**
  - **Setter method**
  - **Field (Not recommended)**

When to Use?

- Whenever a Spring bean needs another bean as a dependency.

Examples

1. Constructor Injection (Recommended)

```
@Service
```

```
public class OrderService {
    private final PaymentService paymentService;

    @Autowired // Optional in Spring 4.3+ if only one constructor
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

2. Setter Injection

```
@Service
public class OrderService {
    @Autowired
    private PaymentService paymentService;
```