

Unit-1

Spring 5 Basics

About Spring core with Spring Boot

Spring is a popular open-source Java application development framework created by Rod Johnson. Spring supports developing any kind of Java application such as standalone applications, web applications, database-driven applications, and many more.

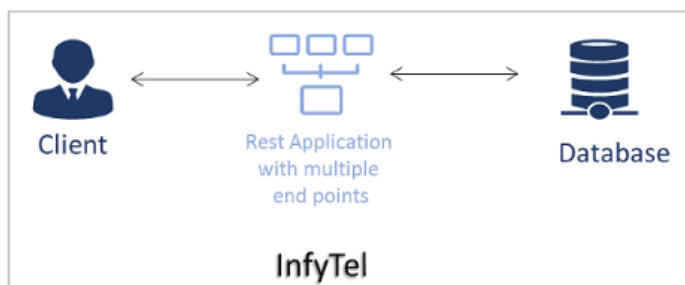
The basic objective of the framework was to reduce the complexity involved in the development of enterprise applications. But today Spring is not limited to enterprise application development, many projects are available under the Spring umbrella to develop different kinds of applications of today's need such as cloud-based applications, mobile applications, batch applications, etc. Spring framework helps in developing a loosely coupled application that is simple, easily testable, reusable, and maintainable.

However, configuring a Spring application is a tedious job. Even though it provides flexibility in bean configuration with multiple ways such as XML, annotation, and Java-based configurations, there is no escape from the configuration. Configuring the Spring features may need more time for developers and may distract the developers from solving business problems.

Spring Boot, one of the topmost innovations in all the existing Spring Framework. Spring Boot provides a new prototype for developing Spring applications with nominal effort.

Infytel Customer Application:

A telecom application called InfyTel application is a Rest based Application that provides functionality for adding, updating, and deleting a customer.

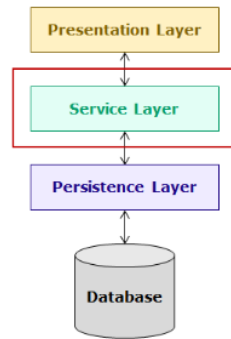


Service layer

InfyTel is built as a three-tier application which consists of

- Presentation Layer
- Service/Business Layer
- Persistence Layer

The service layer of an enterprise application is the layer in which the business logic is implemented. It interacts with the persistence layer and the presentation layer.



In this course Spring Core with Boot, we will see how to develop the Service/Business layer for this application.

Infytel Application Architecture

The InfyTel Application Architecture is as follows



Below are the classes used in the InfyTel application.

- CustomerService.java -> Interface to define service methods
- CustomerServiceImpl.java -> A service class which implements the CustomerService interface
- Client.java-> A class for the main method.
- CustomerRepository.java-> A class for the persistence layer where all CRUD operations are performed.

Why Spring?

Consider our InfyTel application here, we have a CustomerServiceImpl class which implements the CustomerService interface.

```
1. package com.infy.service;
2. public interface CustomerService {
3.     public String fetchCustomer();
4.     public String createCustomer(CustomerDto dto)
5. }
```

The business logic of InfyTel application is implemented in this class. This class is used to perform operations like creating a customer, deleting a customer, etc... CustomerServiceImpl class interacts with CustomerRepository to perform the database related operations.

CustomerServiceImpl.java

```
1. public class CustomerServiceImpl implements CustomerService{
2.     CustomerRepository customerRepository= new CustomerRepositoryImpl();
3.     public String createCustomer(CustomerDto dto) {
4.         return customerRepository.createCustomer(dto);
5.     }
6.     public String fetchCustomer() {
7.         return customerRepository.fetchCustomer();
8.     }
9. }
```

In this implementation, CustomerServiceImpl depends on CustomerRepository. It also instantiates CustomerRepositoryImpl class which makes it tightly coupled with CustomerRepositoryImpl class. This is a bad design because of the following reasons:

- If you want to unit test for createCustomer() method then you need a mock object of CustomerRepository. But you cannot use a mock object because there is no way to substitute the CustomerRepositoryImpl object that CustomerServiceImpl has. So testing CustomerServiceImpl becomes difficult.
- Also, you cannot use a different implementation of CustomerRepository other than CustomerRepositoryImpl because of tight coupling.

So we need a more flexible solution where dependencies can be provided externally rather than a dependent creating its own dependencies.

Need of Spring Framework

Imagine you are building a customer service application for a company. The application should handle customer inquiries, create support tickets, and store customer data in a database. Without using the Spring Framework, you might start with a straightforward implementation.

Customer Class (Plain Java Approach):

```
public class Customer {
    private int id;
    private String name;
    private String email;
    // Getters and setters
}
```

CustomerDAO Class (Plain Java Approach):

```
public class CustomerDAO {  
    public void saveCustomer(Customer customer) {  
        // Code to save the customer data to the database  
    }  
    public Customer getCustomerById(int id) {  
        // Code to retrieve the customer from the database using the provided ID  
        return null;  
    }  
    public List<Customer> getAllCustomers() {  
        // Code to retrieve all customers from the database  
        return new ArrayList<>();  
    }  
}
```

CustomerService Class (Plain Java Approach):

```
public class CustomerService {  
    private CustomerDAO customerDAO;  
  
    public CustomerService() {  
        customerDAO = new CustomerDAO();  
    }  
  
    public void addCustomer(Customer customer) {  
        // Perform validation or business logic if needed  
        customerDAO.saveCustomer(customer);  
    }  
  
    public Customer getCustomerById(int id) {  
        // Perform business logic if needed  
        return customerDAO.getCustomerById(id);  
    }  
  
    public List<Customer> getAllCustomers() {  
        // Perform business logic if needed  
        return customerDAO.getAllCustomers();  
    }  
}
```

As the application grows, you might have more components, such as logging, security, and transaction management, which can make your codebase complex and harder to maintain.

Now, let's see how the Spring Framework can simplify this customer service application:

Customer Class (Spring Approach):

```
public class Customer {  
    private int id;  
    private String name;  
    private String email;  
    // Getters and setters  
}
```

CustomerDAO Class (Spring Approach):

@Repository // Annotation to indicate that this class handles data access

```
public class CustomerDAO {  
    public void saveCustomer(Customer customer) {  
        // Code to save the customer data to the database  
    }  
  
    public Customer getCustomerById(int id) {  
        // Code to retrieve the customer from the database using the provided ID  
        return null;  
    }  
  
    public List<Customer> getAllCustomers() {  
        // Code to retrieve all customers from the database  
        return new ArrayList<>();  
    }  
}
```

CustomerService Class (Spring Approach):

@Service // Annotation to indicate that this class provides a service/business logic

```
public class CustomerService {  
    private final CustomerDAO customerDAO;  
  
    @Autowired // Annotation for automatic dependency injection  
    public CustomerService(CustomerDAO customerDAO) {  
        this.customerDAO = customerDAO;  
    }  
  
    public void addCustomer(Customer customer) {  
        // Perform validation or business logic if needed  
        customerDAO.saveCustomer(customer);  
    }  
  
    public Customer getCustomerById(int id) {  
        // Perform business logic if needed  
    }  
}
```

```
    return customerDAO.getCustomerById(id);
}

public List<Customer> getAllCustomers() {
    // Perform business logic if needed
    return customerDAO.getAllCustomers();
}
}
```

With the Spring Framework:

Dependency Injection (DI): The CustomerService class no longer creates the CustomerDAO instance manually. Instead, it relies on Spring to inject the appropriate CustomerDAO instance when the CustomerService is constructed. This makes your code more modular and allows easier unit testing.

Inversion of Control (IoC): The flow of control in managing objects and their lifecycles is shifted to Spring. You don't have to worry about managing the creation and initialization of objects anymore.

Data Access and Transactions: The CustomerDAO class can utilize Spring's data access and transaction management features, making it easier to interact with the database and ensure data integrity.

By using the Spring Framework, your customer service application becomes more maintainable, scalable, and easier to extend with additional functionalities, such as adding new services, applying cross-cutting concerns like logging, and managing transactions without much hassle. Spring enables you to focus on the core business logic of your application, while it takes care of many of the underlying infrastructure concerns.

What is Spring Framework? Features of Spring Framework:

Spring Framework is an open source Java application development framework that supports developing all types of Java applications such as enterprise applications, web applications, cloud based applications, and many more.

Java applications developed using Spring are simple, easily testable, reusable, and maintainable.

The Spring Framework is used for several reasons, as it provides a wide range of features and benefits that make it a popular choice for building enterprise-level applications. Some of the key reasons why Spring Framework is used are:

Dependency Injection (DI): One of the core principles of the Spring Framework is dependency injection. DI enables loose coupling between classes, making the code more maintainable and easier to test. By removing the responsibility of object creation and wiring from the application code, Spring allows components to be easily configured and managed, promoting better modularity and reusability.

Inversion of Control (IoC): Spring follows the IoC principle, which means it manages the flow of the application and controls the creation and lifecycle of objects. This allows developers to focus on writing

business logic rather than managing object creation and relationships, making the code more readable and less error-prone.

Modularity and Component-Based Architecture: The Spring Framework encourages developers to build applications using a modular and component-based approach. Components can be easily integrated into the application and replaced when needed, which enhances maintainability and scalability.

AOP (Aspect-Oriented Programming): Spring supports Aspect-Oriented Programming, allowing developers to separate cross-cutting concerns (such as logging, security, and caching) from the core business logic. This promotes code modularity and reusability, as aspects can be applied to multiple parts of the application.

Data Access and Persistence: Spring provides powerful support for data access and persistence through its integration with popular ORM frameworks like Hibernate and JPA. This simplifies database interactions and reduces boilerplate code.

Transaction Management: Spring offers declarative transaction management, which simplifies handling database transactions. By using annotations or XML configuration, developers can define transaction boundaries easily, making it less error-prone and more maintainable.

Integration with other frameworks and libraries: Spring can be seamlessly integrated with other frameworks and libraries, such as Spring Security for authentication and authorization, Spring Web MVC for building web applications, and Spring Boot for rapid application development.

Testing Support: Spring provides robust support for unit testing and integration testing, allowing developers to write test cases and mock dependencies effectively. This makes testing easier and promotes test-driven development practices.

Security: Spring offers a range of security features through its Spring Security module, enabling developers to implement authentication, authorization, and other security measures in their applications.

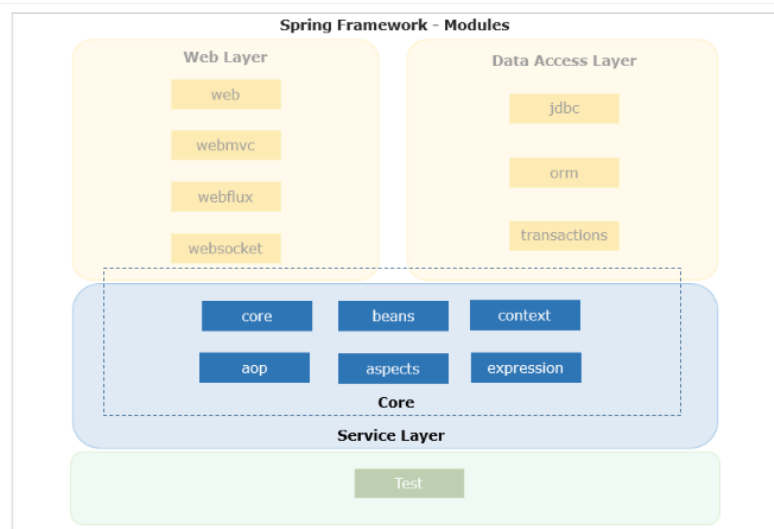
Community and Ecosystem: The Spring Framework has a large and active community, which means extensive documentation, tutorials, and resources are available to assist developers. Moreover, Spring's ecosystem includes various extensions and third-party libraries that can be leveraged to address specific application needs.

Spring Framework – Modules

Spring Framework 5.x has the following key module groups:

- **Core Container:** These are core modules that provide key features of the Spring framework.
- **Data Access/Integration:** These modules support JDBC and ORM data access approaches in Spring applications.
- **Web:** These modules provide support to implement web applications.
- **Others:** Spring also provides few other modules such as the Test for testing Spring applications.

Spring Modules - Core Container



In the Spring Framework, the core container provides the foundational modules that form the backbone of the entire Spring ecosystem. These modules are responsible for the fundamental features, including dependency injection (DI) and inversion of control (IoC). Here are the key modules of the Spring core container:

Spring Core: This module is the foundation of the Spring Framework and provides the core functionality for dependency injection and IoC. It includes the BeanFactory interface, which is responsible for managing and providing beans (objects) and their dependencies. The core module is found in the spring-core JAR.

Spring Beans: This module defines the concept of a "bean" in the Spring context. A bean is an object that is managed by the Spring container and is configured using XML, annotations, or Java-based configuration. The @Component, @Autowired, and other related annotations are part of this module. The Spring Beans module is found in the spring-beans JAR.

Spring Context: The Spring Context module builds on top of the core container and provides a more advanced application context. It includes features like internationalization, event propagation, resource loading, and environment profiles. The Spring Context module is found in the spring-context JAR.

Spring Expression Language (SpEL): SpEL is a powerful expression language that allows you to define expressions for evaluating values at runtime. It is widely used in annotations such as @Value and @Conditional expressions. The SpEL module is found in the spring-expression JAR.

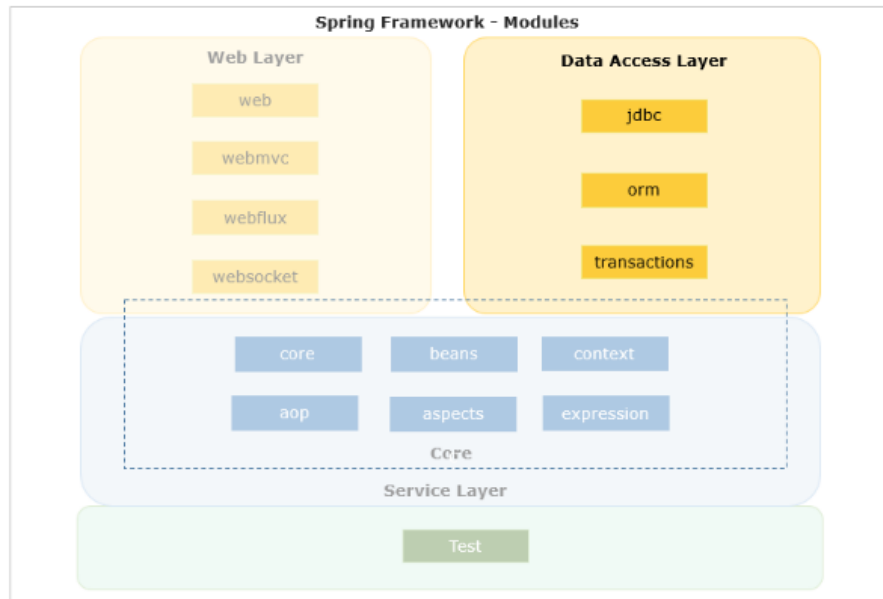
Spring AOP (Aspect-Oriented Programming): The AOP module enables aspect-oriented programming in Spring applications. AOP allows you to separate cross-cutting concerns, such as logging, security, and caching, from the core business logic. The Spring AOP module is found in the spring-aop JAR.

Spring Test: The Spring Test module provides support for testing Spring applications. It includes integration testing features for loading Spring contexts, managing transactional behavior, and more. The Spring Test module is found in the spring-test JAR.

These modules work together to create a robust and flexible foundation for building Spring applications. By utilizing the core container modules, developers can achieve loose coupling between components, easily

manage dependencies, and take advantage of various Spring features, making the development process more efficient and maintainable.

Spring Modules - Data Access/Integration



In the Spring Framework, data access is a critical aspect of building enterprise-level applications. These modules offer integration with various data access technologies, including JDBC, ORM (Object-Relational Mapping), and NoSQL databases. Here are the key data access modules in the Spring Framework:

Spring JDBC: The Spring JDBC module provides support for working with traditional JDBC (Java Database Connectivity) to interact with relational databases. It simplifies common JDBC operations like connection management, query execution, and exception handling. It offers features such as `JdbcTemplate` and `NamedParameterJdbcTemplate` to execute SQL queries and map results to Java objects efficiently.

Spring ORM (Object-Relational Mapping): The Spring ORM module integrates the Spring Framework with popular ORM frameworks like Hibernate, JPA (Java Persistence API), and JDO (Java Data Objects). It provides seamless integration and configuration options to work with ORM frameworks, allowing developers to focus on business logic rather than database interactions.

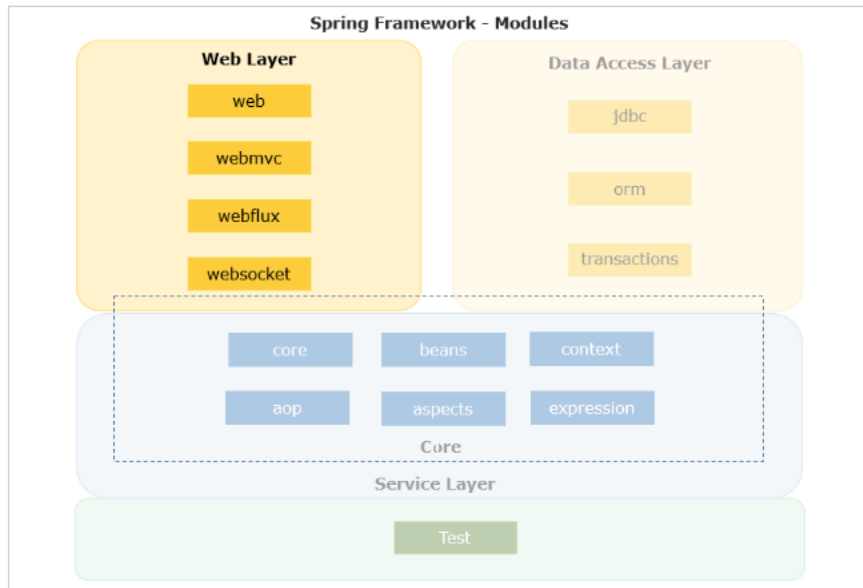
Spring Transaction Management: The Spring Transaction Management module offers declarative and programmatic transaction management capabilities. It supports both local and distributed transactions. By using annotations or XML-based configuration, developers can define transaction boundaries, and the Spring container handles transaction management transparently.

Spring Data JPA: Spring Data JPA is part of the larger Spring Data project and simplifies working with JPA-based data repositories. It provides powerful repository abstractions, reducing the amount of boilerplate code required for database interactions.

Spring Data MongoDB: For NoSQL databases like MongoDB, Spring Data MongoDB provides integration and a convenient way to work with MongoDB collections. It offers repository abstractions and query methods tailored to MongoDB's document-based nature.

These data access modules in the Spring Framework provide a unified and consistent approach to working with various data sources, making it easier for developers to manage data interactions and switch between different data technologies seamlessly.

Spring Modules – Web



In the Spring Framework, web modules provide powerful capabilities for building web applications and RESTful web services. These modules facilitate the development of web-related components, handle HTTP requests and responses, and support various web-related features. Here are the key web modules in the Spring Framework:

Spring Web (spring-web): The Spring Web module forms the foundation for web applications. It provides basic web-related functionalities, such as handling HTTP requests and responses, serving static resources, and managing servlet and filter lifecycles. This module includes the `DispatcherServlet`, which acts as the front controller in Spring web applications and handles incoming requests.

Spring Web MVC (spring-webmvc): Spring Web MVC is an extension of the Spring Web module and focuses on building Model-View-Controller (MVC) web applications. It provides a powerful and flexible MVC architecture that separates the application into model (data), view (UI), and controller (request processing). Developers can use annotations or XML configuration to define mappings between URLs and controller methods, enabling easy handling of HTTP requests and rendering responses.

Spring WebFlux (spring-webflux): Spring WebFlux is a reactive web framework introduced in Spring 5 that supports building reactive web applications. It is designed to handle high concurrency with a non-blocking, event-driven approach. Developers can choose between the traditional annotation-based

programming model (with `@Controller` and `@RestController`) or the functional programming model using `RouterFunctions`.

Spring Web Services (spring-ws): Spring Web Services provides support for building SOAP-based web services. It simplifies the development of contract-first web services using XML schemas (XSD) to define the message formats. The module includes classes for handling XML marshaling and unmarshaling, as well as endpoints for processing SOAP requests.

Spring Security (spring-security): Spring Security is a powerful module that provides comprehensive security features for web applications and RESTful services. It supports various authentication mechanisms, authorization, secure access control, and integration with various authentication providers like LDAP, OAuth, and more.

Spring WebSocket (spring-websocket): The Spring WebSocket module enables real-time, bidirectional communication between web clients and the server. It builds on top of the WebSocket protocol and provides support for handling WebSocket connections, sending and receiving messages, and managing WebSocket sessions.

Spring Web Services Client (spring-ws-support): This module offers support for building client-side components to interact with SOAP-based web services. It provides classes for marshaling and unmarshaling SOAP messages and creating client-side proxies for web service endpoints.

These web modules in the Spring Framework offer a wide range of features to build robust and scalable web applications and services. They follow the principles of dependency injection and inversion of control, allowing developers to create highly maintainable and testable web applications. With Spring's modular architecture, developers can pick and choose the necessary modules that suit their specific requirements, making it a popular choice for web development in the Java ecosystem.

Spring Modules – Testy modules



The Spring Test module, also known as the Spring TestContext Framework, is a crucial component of the Spring Framework. It provides powerful support for testing Spring applications, including integration testing, unit testing. The Spring Test module is included in the `spring-test` JAR.

Key features and components of the Spring Test module include:

Spring TestContext Framework: The core of the Spring Test module is the TestContext framework. It allows you to load and manage the Spring application context for testing. With this framework, you can create and configure application contexts, set up test data, and perform various assertions on the context and its components.

Spring Test Annotations: The module provides a set of annotations that simplify the configuration and execution of tests:

@RunWith(SpringRunner.class): This annotation is used at the class level to specify the test runner that integrates JUnit with the Spring TestContext Framework.

@ContextConfiguration: This annotation is used to specify the location of the Spring configuration files or Java configuration classes that define the test context.

@Test: Standard JUnit annotation to mark test methods.

@Autowired: This annotation is used to inject Spring beans into test classes, allowing you to use Spring's dependency injection in your tests.

@DirtiesContext: This annotation indicates that the Spring context should be dirtied (i.e., closed and reloaded) after the test, which is useful when the test modifies the context.

Mock Objects Support: Spring Test provides support for creating and working with mock objects using the @MockBean annotation. It allows you to easily mock dependencies during testing.

Transactional Support: With the @Transactional annotation, you can manage transactions during tests. This annotation ensures that the test methods run within a transaction, and any changes made during the test are rolled back at the end.

Testing Support for Web Applications: Spring Test offers support for testing web applications with the MockMvc and TestRestTemplate classes. They allow you to send HTTP requests to controllers and REST endpoints and perform assertions on the responses.

The Spring Test module is widely used to ensure the correctness and reliability of Spring applications. It enables developers to perform comprehensive testing of Spring components, including controllers, services, repositories, and more. By providing support for various testing scenarios and the ability to set up isolated and repeatable test environments, the Spring Test module promotes best practices for testing in Spring-based projects.

Scope - Spring Framework Modules

The Spring Framework is a comprehensive and modular framework that provides a wide range of features to build robust and scalable Java applications. Each module in the Spring Framework addresses specific concerns and can be used independently or in combination with other modules to meet the requirements of diverse application types. Here's an overview of the scope and common use cases of some key Spring Framework modules:

Core Container Modules:

Scope: The core container modules, including Spring Core and Spring Beans, provide the foundation for the Spring Framework. They offer dependency injection (DI) and inversion of control (IoC) capabilities, which help in building loosely coupled and maintainable components.

Use Cases: These modules are fundamental to almost all Spring applications, as they enable the management of beans, their lifecycles, and their dependencies.

Data Access Modules:

Scope: Spring's data access modules facilitate interaction with various data sources, such as relational databases, NoSQL databases, and message queues. They provide seamless integration with popular data access technologies and simplify database operations.

Use Cases: These modules are essential for applications that require data storage and retrieval. Developers can choose the appropriate module based on the data source and preferred data access technology.

Web Modules:

Scope: The web modules of Spring Framework cater to web application development, supporting both traditional MVC applications (Spring Web MVC) and reactive applications (Spring WebFlux).

Use Cases: These modules are used for building web applications, RESTful web services, and real-time communication through WebSocket.

Security Module:

Scope: Spring Security provides comprehensive security features to secure web applications and RESTful services. It offers authentication, authorization, and protection against common security vulnerabilities.

Use Cases: Any application that requires user authentication, authorization, and protection against security threats can benefit from using Spring Security.

Integration Module:

Scope: Spring Integration supports building message-driven applications by providing lightweight messaging and integration with various messaging systems.

Use Cases: These modules are suitable for applications that require asynchronous communication between components or systems using messaging systems like JMS or Apache Kafka.

Testing Module:

Scope: The Spring Test module offers powerful testing support, including integration testing and unit testing with Spring applications. It allows developers to set up isolated test environments and easily perform assertions on Spring components.

Use Cases: These modules are used to write comprehensive test suites to ensure the correctness and reliability of Spring applications.

Batch Module:

Scope: Spring Batch is used for building batch processing applications that deal with large volumes of data in a batch-oriented manner.

Use Cases: Batch processing applications like data imports, exports, or periodic data processing tasks can benefit from using Spring Batch.

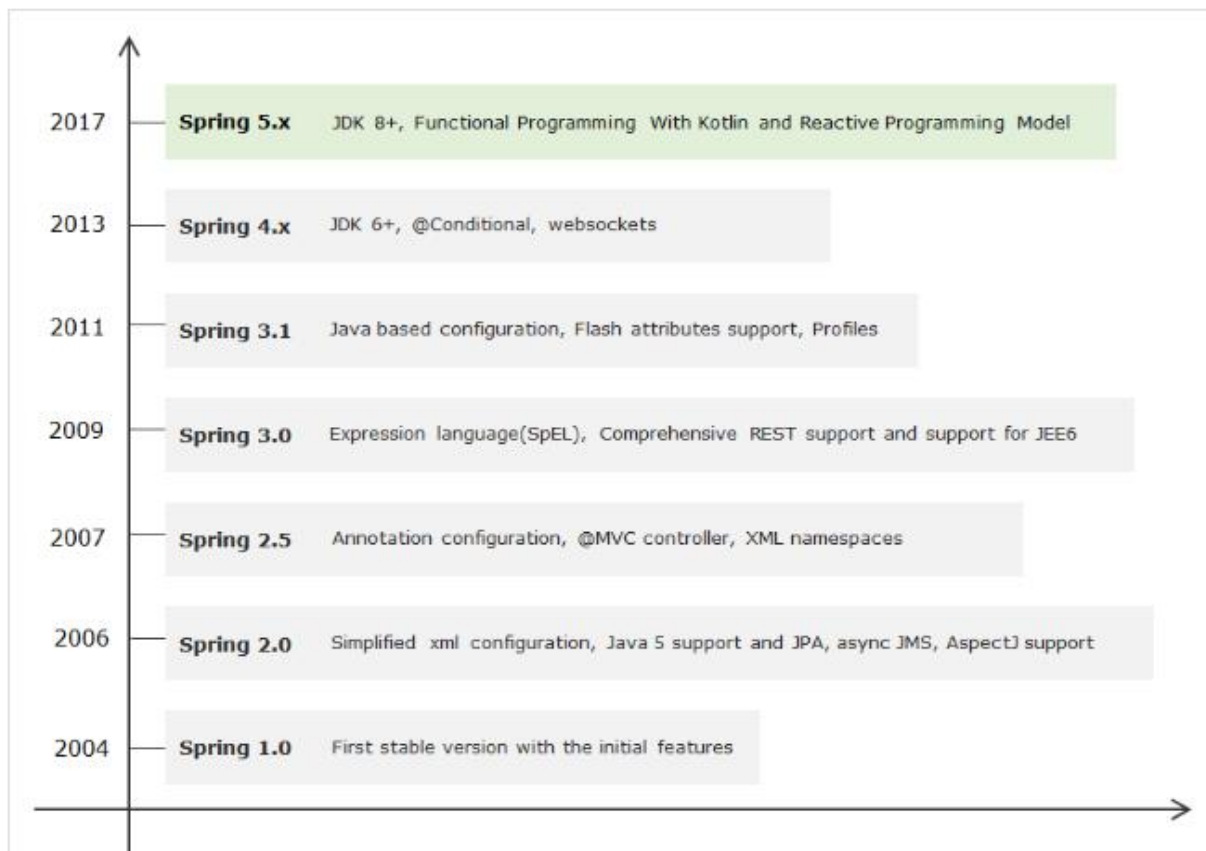
Cloud and Reactive Modules (optional):

Scope: Spring Cloud and Spring WebFlux (reactive) modules are optional but beneficial for cloud-native and reactive application development, respectively.

Use Cases: These modules are relevant for building cloud-native applications, microservices, and applications that require high concurrency and responsiveness.

The Spring Framework's modular design allows developers to select and use the modules that fit their application's needs, making it versatile and suitable for various application domains and architectural styles.

Spring Version History



The current version of Spring Framework is 5.x, the framework has been enhanced with new features keeping core concepts the same as Spring 4.x.

At a high level, the new features of Spring Framework 5.x are:

- JDK baseline update
- Core framework revision
- Reactive Programming Model: Introduces a new non-blocking web framework called Spring WebFlux
- Functional programming using Kotlin language support
- Testing improvements by supporting integration with JUnit5

Let us look at Spring core relevant changes in detail:

JDK baseline update

The entire Spring framework 5.x codebase runs on Java 8 and designed to work with Java 9. Therefore, Java 8 is the minimum requirement to work on Spring Framework 5.x

Core framework revision

The core Spring Framework 5.x has been revised, one of the main changes is Spring comes with its own commons-logging through spring-jcl jar instead of standard Commons Logging.

Spring IoC(Inversion of Control)

As discussed earlier, usually it is the developer's responsibility to create the dependent application object using the new operator in an application. Hence any change in the application dependency requires code change and this results in more complexity as the application grows bigger.

Inversion of Control (IoC) helps in creating a more loosely coupled application. IoC represents the inversion of the responsibility of the application object's creation, initialization, and destruction from the application to the third party such as the framework. Now the third party takes care of application object management and dependencies thereby making an application easy to maintain, test, and reuse.

There are many approaches to implement IoC, Spring Framework provides IoC implementation using Dependency Injection(DI).

Introduction to Spring Inversion of Control(IoC)

Spring Container managed application objects are called beans in Spring.

We need not create objects in dependency injection instead describe how objects should be created through configuration.

DI is a software design pattern that provides better software design to facilitate loose coupling, reuse, and ease of testing.

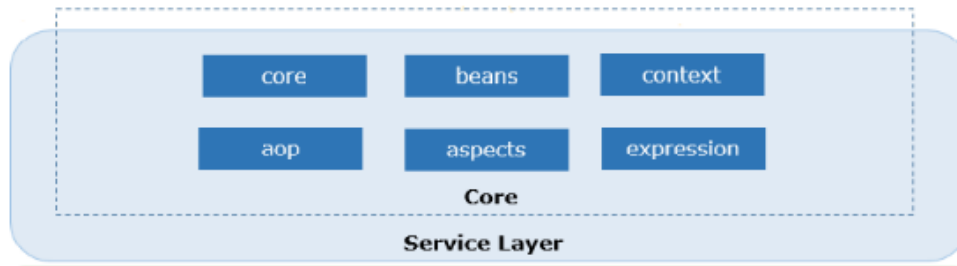
Benefits of Dependency Injection(DI):

- Helps to create loosely coupled application architecture facilitating re-usability and easy testing.

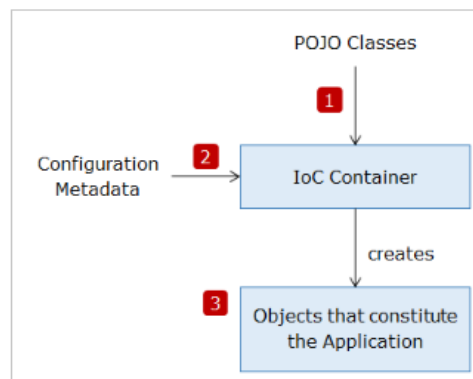
- Separation of responsibility by keeping code and configuration separately. Hence dependencies can be easily modified using configuration without changing the code.
- Allows to replace actual objects with mock objects for testing, this improves testability by writing simple JUnit tests that use mock objects.

Spring IoC

The core container module of the Spring Framework provides IoC using Dependency Injection.



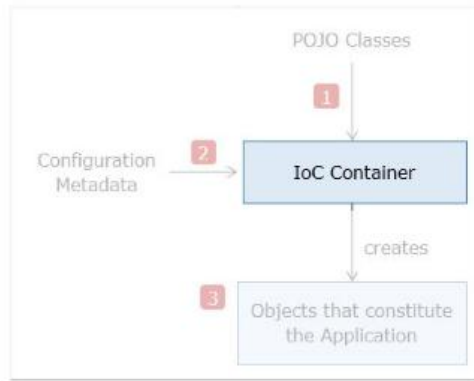
The Spring container knows which objects to create and when to create through the additional details that we provide in our application called **Configuration Metadata**.



1. Application logic is provided through POJO classes.
2. Configuration metadata consists of bean definitions that the container must manage.
3. IoC container produces objects required by the application using POJO classes and configuration metadata. IoC container is of two types – **BeanFactory** and **ApplicationContext**.

Spring IoC - Containers

Spring provides two types of containers

**BeanFactory:**

- It is the basic Spring container with features to instantiate, configure and manage the beans.
- **org.springframework.beans.factory.BeanFactory** is the main interface representing a BeanFactory container.

ApplicationContext:

- ApplicationContext is another Spring container that is more commonly used in Spring applications.
- **org.springframework.context.ApplicationContext** is the main Interface representing an ApplicationContext container.
- It inherits the BeanFactory features and provides added features to support enterprise services such as internationalization, validation, etc.

ApplicationContext is the preferred container for Spring application development. Let us look at more details on the ApplicationContext container.

BeanFactory Vs ApplicationContext

BeanFactory	ApplicationContext
It does not support annotation based Dependency Injection.	Support annotation based Dependency Injection.
It does not support enterprise services.	Support enterprise services such as validations, internationalization, etc.
By default, it supports Lazy Loading.	By default, it supports Eager Loading. Beans are instantiated during load time.
//Loading BeanFactory BeanFactory factory = new AnnotationConfigApplicationContext(SpringConfigur ation.class); // Instantiating bean during first access using getBean() CustomerServiceImpl service =	// Loading ApplicationContext and instantiating bean ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfigu ration.class); // Instantiating bean during first access using getBean()

```
(CustomerServiceImpl)
factory.getBean("customerService");
```

```
CustomerServiceImpl service =
(CustomerServiceImpl)
context.getBean("customerService");
```

ApplicationContext and AbstractApplicationContext


org.springframework.context.annotation.AnnotationConfigApplicationContext is one of the most commonly used implementation of ApplicationContext.

Example: ApplicationContext container instantiation.

```
1.ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);
2.Object obj = context.getBean("customerService");
```

1. ApplicationContext container is instantiated by loading the configuration from the SpringConfiguration.class which is available in the utility package of the application.
2. Accessing the bean with id "customerService" from the container.

AbstractApplicationContext

 Resource leak: 'context' is never closed ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);

You can see a warning message as **Resource leak: 'context' is never closed** while using the ApplicationContext type. This is for the reason that you don't have a close method with BeanFactory or even ApplicationContext. AbstractApplicationContext is an abstract implementation of the ApplicationContext interface and it implements Closeable and AutoCloseable interfaces. To close the application context and destroy all beans in its abstractApplicationContext has a close method that closes this application context.

Access bean in Spring

There are different ways to access bean in Spring

1. The traditional way of accessing bean based on bean id with explicit typecast

```
1. CustomerServiceImpl service = (CustomerServiceImpl) context.getBean("customerService");
```

2. Accessing bean based on class type to avoid typecast if there is a unique bean of type in the container

```
2. CustomerServiceImpl service = context.getBean(CustomerServiceImpl.class);
```

3. Accessing bean through bean id and also type to avoid explicit typecast

```
CustomerServiceImpl service = context.getBean("customerService", CustomerServiceImpl.class);
```

Configuration Meta data in spring/ Configuring the IoC Container:

In Spring Framework, configuration metadata is essential for defining how the various components in your application should be wired together. Spring provides multiple ways to configure the application, and each method uses configuration metadata in different formats.

Here are some common ways to provide configuration metadata in Spring:

1. **XML Configuration:** XML-based configuration is one of the earliest and most widely used ways to configure a Spring application. In this approach, you define the beans, their dependencies, and other configuration details in XML files.

Example of XML configuration:

```
<!-- beans.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="myService" class="com.example.MyService"/>
    <bean id="myRepository" class="com.example.MyRepository">
        <property name="dataSource" ref="myDataSource"/>
    </bean>
    <bean id="myDataSource" class="com.example.MyDataSource"/>
</beans>
```

2. **Java-based Configuration (Annotations):** Instead of using XML, Spring also allows you to use Java-based configuration with annotations. Here, you use special annotations to define beans and their dependencies directly in Java classes.

Example of Java-based configuration:

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyService();
    }

    @Bean
    public MyRepository myRepository() {
        return new MyRepository(myDataSource());
    }

    @Bean
```

```
public MyDataSource myDataSource() {  
    return new MyDataSource();  
}  
}
```

- 3. Java-based Configuration (Java DSL):** In addition to annotations, Spring provides a Java DSL (Domain-Specific Language) to configure the application programmatically. This gives you more flexibility and control over the configuration process.

Example of Java-based configuration using Java DSL:

@Configuration

```
public class AppConfig {  
    @Bean  
    public MyService myService() {  
        return new MyService();  
    }  
    @Bean  
    public MyRepository myRepository() {  
        return new MyRepository(myDataSource());  
    }  
    @Bean  
    public MyDataSource myDataSource() {  
        return new MyDataSource();  
    }  
}
```

- 4. Property File Configuration:** Spring allows you to externalize configuration properties in property files. These properties can be loaded into the application context and used to configure beans or other parts of the application.

Example of property file configuration:

```
# application.properties  
db.username=myuser  
db.password=mypassword
```

AppConfig.java file:

@Configuration

@PropertySource("classpath:application.properties")

```
public class AppConfig {  
    @Value("${db.username}")  
    private String dbUsername;  
    @Value("${db.password}")  
    private String dbPassword;  
    // ... other bean definitions using the dbUsername and dbPassword  
}
```

These are some of the common ways to provide configuration metadata in Spring. You can choose the approach that best fits your project's requirements and coding preferences. Additionally, Spring Boot offers even more streamlined and convenient ways to configure applications, building on top of these Spring configuration mechanisms.

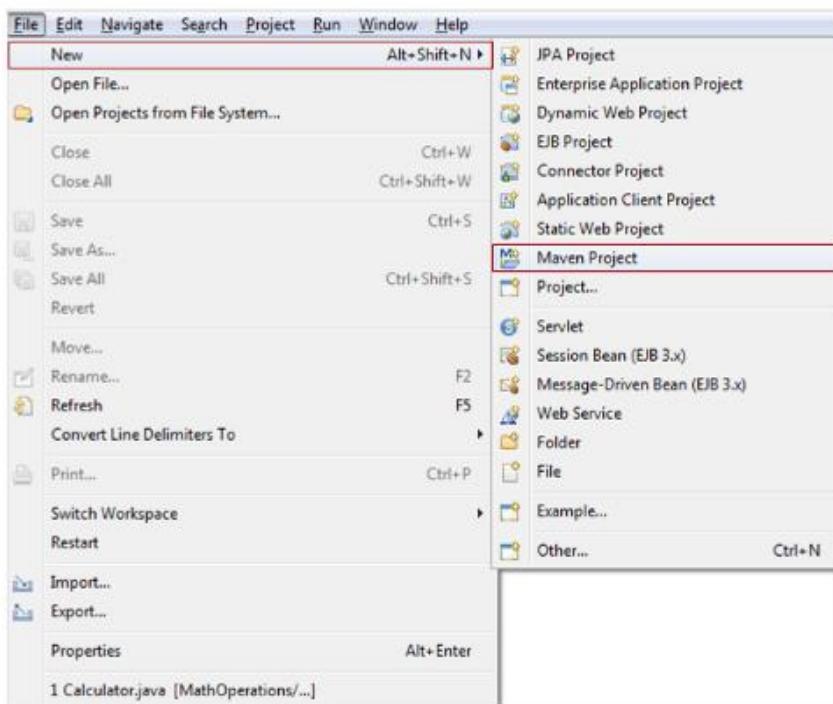
Demo Steps:

Let us understand the steps required to create a Maven project for Spring basic application using Spring Tool Suite(STS)/Eclipse IDE.

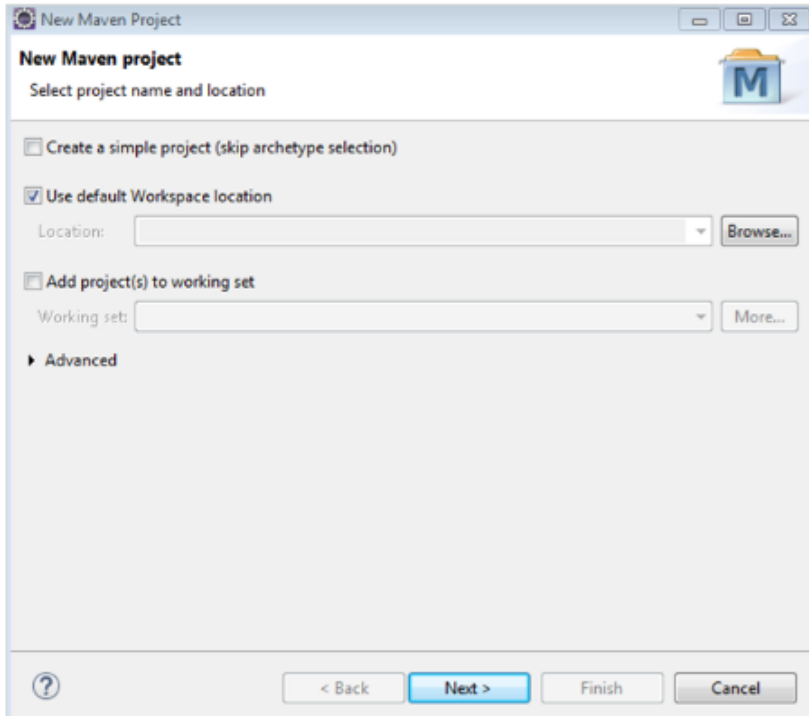
Note: Screenshots of this demo have been taken from Eclipse IDE, however, the steps remain similar even for the Spring Tool Suite IDE.

Step 1: Create a Maven project in Eclipse as shown below

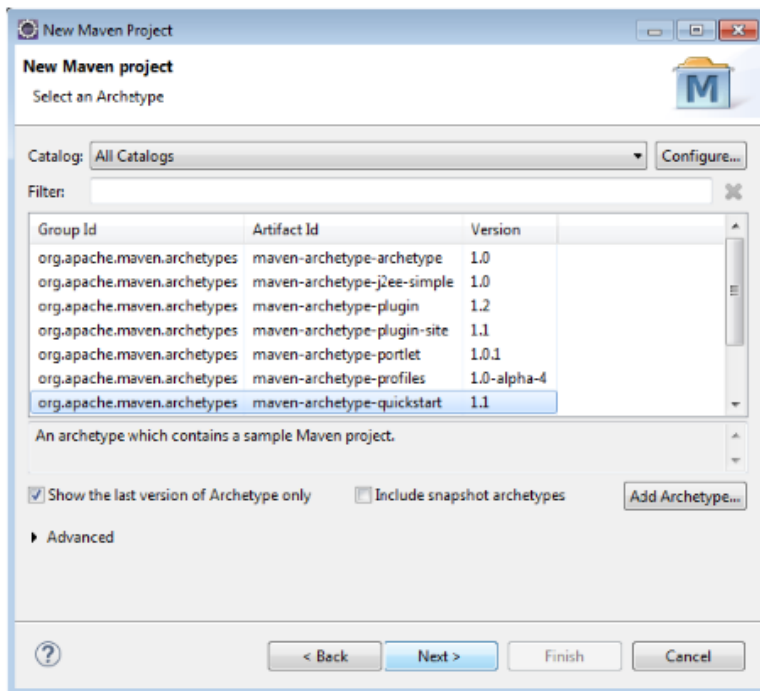
Goto File > new > Maven Project, you would get the below screen.



Step 2: Let the current workspace be chosen as the default location for the Maven project. Click on next.



Step 3: Choose maven-archetype-quickstart



Step 4: Provide groupId as com.infy, artifactId as demo1, and retain the default version which is 0.0.1-SNAPSHOT.

We can customize the package names as per our needs. In this demo, the package name is provided as **com.infy**. Click on the finish button to complete the project creation.

New Maven Project
Specify Archetype parameters

Group Id:

Artifact Id:

Version:

Package:

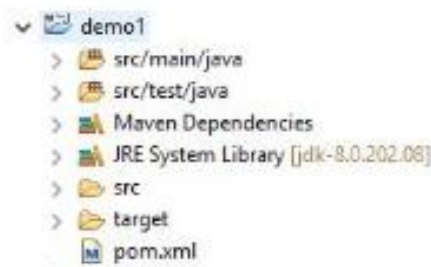
Properties available from archetype:

Name	Value

► Advanced

< Back Next > **Finish** Cancel

Step 5: New project along with a POM.xml file is created as shown below

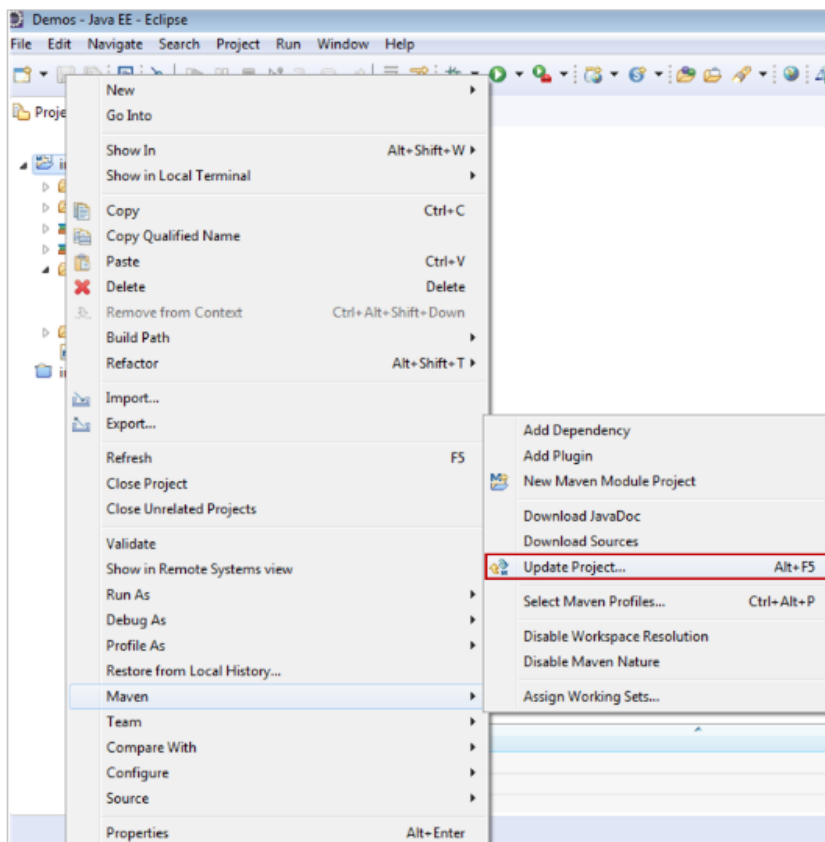


Step 6: Add the below dependency in the POM.xml file.

1. `<project xmlns="http://maven.apache.org/POM/4.0.0"`
`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
2. `xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-`
`4.0.0.xsd">`
3. `<modelVersion>4.0.0</modelVersion>`
4. `<groupId>com.infosys</groupId>`
5. `<artifactId>demo1-spring-ioc</artifactId>`
6. `<version>0.0.1-SNAPSHOT</version>`
7. `<packaging>jar</packaging>`
8. `<name>demo1-spring-ioc</name>`
9. `<url>http://maven.apache.org</url>`
10. `<properties>`
11. `<spring.version>5.0.5.RELEASE</spring.version>`

```
12. <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13. </properties>
14. <dependencies>
15. <!-- Spring Dependency -->
16. <dependency>
17. <groupId>org.springframework</groupId>
18. <artifactId>spring-context</artifactId>
19. <version>${spring.version}</version>
20. </dependency>
21. <dependency>
22. <groupId>junit</groupId>
23. <artifactId>junit</artifactId>
24. <version>3.8.1</version>
25. <scope>test</scope>
26. </dependency>
27. </dependencies>
28. </project>
```

Step 7: If the dependencies are not getting downloaded automatically update the Maven project as shown below.



Step 8: Add the required project files in the respective packages

groupid and artifactid in maven project

In a Maven project, the groupId and artifactId are two important identifiers used to uniquely identify the project and its artifacts.

groupId: The groupId represents the unique identifier for your project's group or organization. It is typically based on your company's domain name in reverse order, or it can be any other unique identifier you choose. The groupId is used to group related projects together under the same organization.

For example:

If your company's domain name is "com.example," your groupId could be com.example.

If you don't have a domain or are working on a personal project, you can use a different identifier, such as org.myproject.

The groupId is specified in the Maven pom.xml file as follows:

```
<groupId>com.example</groupId>
```

artifactId: The artifactId represents the unique identifier for a specific artifact, typically a single project or module within the larger organization or group. It is the name of the jar/war/ear file that will be generated when you build the project.

For example:

If your project is a web application, the artifactId might be **my-webapp**.

If your project is a library, the artifactId might be **my-library**.

The artifactId is also specified in the Maven pom.xml file as follows:

```
<artifactId>my-webapp</artifactId>
```

Together, the combination of groupId and artifactId uniquely identifies your project and its artifacts in the Maven ecosystem. This is especially useful when publishing artifacts to repositories or when resolving dependencies from repositories during the build process.

A typical Maven project structure may look like this:

```
my-project/
```

```
|- pom.xml
```

```
|- src/
```

```
    |- main/
```

```
        |- java/
```

```
|- resources/  
  
|- test/  
  
|- java/  
  
|- resources/
```

In the **pom.xml** file, you define the groupId and artifactId along with other project information and dependencies. For example:

```
<project>  
  <groupId>com.example</groupId>  
  <artifactId>my-webapp</artifactId>  
  <version>1.0.0</version>  
  <!-- Other configuration details and dependencies -->  
</project>
```

When building the project with Maven, it will use this information to generate the appropriate output and handle dependencies accordingly.

Demo Spring IOC

CustomerService.java

```
1. package com.infy.service;  
2. public interface CustomerService {  
3.     public String createCustomer();  
4. }
```

CustomerServiceImpl.java

```
1. package com.infy.service;  
2. public class CustomerServiceImpl implements CustomerService {  
3.     public String createCustomer() {  
4.         return "Customer is successfully created";  
5.     }  
6. }
```

SpringConfiguration.java

```
1. package com.infy.util;  
2. import org.springframework.context.annotation.Bean;  
3. import org.springframework.context.annotation.Configuration;  
4. import com.infy.service.CustomerServiceImpl;  
5. @Configuration  
6. public class SpringConfiguration {  
7.     @Bean(name = "customerService")  
8.     public CustomerServiceImpl customerServiceImpl() {
```

```
9. return new CustomerServiceImpl();
10. }
11. }
```

Client.java

```
1. package com.infy;
2. import org.springframework.context.annotation.AnnotationConfigApplicationContext;
3. import org.springframework.context.support.AbstractApplicationContext;
4. import com.infy.service.CustomerServiceImpl;
5. import com.infy.util.SpringConfiguration;
6. public class Client {
7.     public static void main(String[] args) {
8.         CustomerServiceImpl service = null;
9.         AbstractApplicationContext context = new
            AnnotationConfigApplicationContext(SpringConfiguration.class);
10.        service = (CustomerServiceImpl) context.getBean("customerService");
11.        System.out.println(service.createCustomer());
12.        context.close();
13.    }
14. }
```

Output:

Customer is successfully created

Dependency Injection

For the previously discussed InfyTel Customer application, we defined CustomerService bean as shown below.

```
1. @Bean
2. public CustomerService customerService() {
3.     return new CustomerService();
4. }
```

This is the same as the below Java code wherein an instance is created and initialized with default values using the default constructor.

```
CustomerService customerService = new CustomerService();
```

How do we initialize beans with some specific values in Spring?

This can be achieved through **Dependency Injection** in Spring.

Inversion of Control pattern is achieved through Dependency Injection (DI) in Spring. In Dependency Injection, the developer need not create the objects but specify how they should be created through configuration.

Spring container uses one of these two ways to initialize the properties:

- **Constructor Injection:** This is achieved when the container invokes a parameterized constructor to initialize the properties of a class

- **Setter Injection:** This is achieved when the container invokes setter methods of a class to initialize the properties after invoking a default constructor.

Constructor Injection - Primitive values

Let us consider the CustomerService class of InfyTel Customer application to understand constructor injection.

CustomerService class has a count property, let us now modify this class to initialize count property during bean instantiation using the constructor injection approach.

```
1. package com.infy.service;
2. public class CustomerServiceImpl implements CustomerService {
3.     private int count;
4.     public CustomerServiceImpl(int count) {
5.         this.count = count;
6.     }
7. }
```

How do we define bean in the configuration to initialize values?

```
1. @Configuration
2. public class SpringConfiguration {
3.     @Bean // CustomerService bean definition with bean dependencies through constructor injection
4.     public CustomerServiceImpl customerService() {
5.         return new CustomerServiceImpl(20);
6.     }
```

What is mandatory for constructor injection?

A parameterized constructor is required in the CustomerService class as we are injecting the values through the constructor argument.

Can we use constructor injection to initialize multiple properties?

Yes, we can initialize more than one property.

Constructor Injection - Non primitive values

So far, we learned how to inject primitive values using constructor injection in Spring. Now we will look into inject object dependencies using Constructor injection. Consider our InfyTel Customer application. **CustomerServiceImpl.java** class which is dependent on CustomerRepository(class used to in persistence layer to perform CRUD operations) object type to call fetchCustomer() method.

```
1. package com.infy.service;
2. public class CustomerServiceImpl implements CustomerService {
3.     // CustomerServiceImpl needs to contact CustomerRepository, hence injecting the
    customerRepository dependency
4.     private CustomerRepository customerRepository;
5.     private int count;
6.     public CustomerServiceImpl() {
7. }
```

```
8. public CustomerServiceImpl(CustomerRepository customerRepository, int count) {
9.   this.customerRepository = customerRepository;
10.  this.count=count;
11. }
12. public String fetchCustomer() {
13.  return customerRepository.fetchCustomer(count);
14. }
15. public String createCustomer() {
16.  return customerRepository.createCustomer();
17. }
18. }
```

Observe in the above code, CustomerRepository property of CustomerService class has not been initialized with any value in the code. This is because Spring dependency is going to be taken care of in the configuration.

How do we inject object dependencies through configuration using constructor injection?

```
1. package com.infy.util;
2. @Configuration
3. public class SpringConfiguration {
4.   @Bean// customerRepository bean definition
5.   public CustomerRepository customerRepository() {
6.     return new CustomerRepository();
7.   }
8.   @Bean // CustomerService bean definition with bean dependencies through constructor injection
9.   public CustomerServiceImpl customerService() {
10.    return new CustomerServiceImpl(customerRepository(),20);
11.  }
12. }
```

Demo: Constructor Injection: SpringConfiguration .java

```
1. package com.infy.util;
2. import org.springframework.context.annotation.Bean;
3. import org.springframework.context.annotation.Configuration;
4. import com.infy.repository.CustomerRepository;
5. import com.infy.service.CustomerServiceImpl;
6. @Configuration
7. public class SpringConfiguration {
8.   @Bean // Constructor Injection
9.   public CustomerServiceImpl customerService() {
10.    return new CustomerServiceImpl(customerRepository(), 20);
11.  }
12. @Bean // Constructor Injection
```

```
13. public CustomerRepository customerRepository() {  
14. return new CustomerRepository();  
15. }  
16. }
```

CustomerService.java

```
1. package com.infy.service;  
2. public interface CustomerService {  
3. public String fetchCustomer();  
4. public String createCustomer();  
5. }
```

CustomerServiceImpl.java

```
1. package com.infy.service;  
2. import com.infy.repository.CustomerRepository;  
3. public class CustomerServiceImpl implements CustomerService {  
4. private int count;  
5. private CustomerRepository repository;  
6. public CustomerServiceImpl(CustomerRepository repository, int count) {  
7. this.count = count;  
8. this.repository = repository;  
9. }  
10. public String fetchCustomer() {  
11. return repository.fetchCustomer(count);  
12. }  
13. public String createCustomer() {  
14. return repository.createCustomer();  
15. }  
16. }
```

CustomerRepository.java

```
1. package com.infy.repository;  
2. public class CustomerRepository {  
3. public String fetchCustomer(int count) {  
4. return " The no of customers fetched are : " + count;  
5. }  
6. public String createCustomer() {  
7. return "Customer is successfully created";  
8. }  
9. }
```

Client.java

```
1. package com.infy;  
2. import org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
3. import org.springframework.context.support.AbstractApplicationContext;
4. import com.infy.service.CustomerServiceImpl;
5. import com.infy.util.SpringConfiguration;
6. public class Client {
7.     public static void main(String[] args) {
8.         CustomerServiceImpl service = null;
9.         AbstractApplicationContext context = new
            AnnotationConfigApplicationContext(SpringConfiguration.class);
10.        service = (CustomerServiceImpl) context.getBean("customerService");
11.        System.out.println(service.fetchCustomer());
12.        context.close();
13.    }
14. }
```

Output:

The no of customers fetched are : 20

Setter Injection-primitive

Let us now understand Setter Injection in Spring.

In Setter Injection, Spring invokes setter methods of a class to initialize the properties after invoking a default constructor.

How can we use setter injection to inject values for the primitive type of properties?

Consider the below example to understand setter injection for primitive types.

Following the CustomerServiceImpl class has a count property, let us see how to initialize this property during bean instantiation using the setter injection approach.

```
1. package com.infy.service;
2. public class CustomerServiceImpl implements CustomerService {
3.     private int count;
4.     public int getCount() {
5.         return count;
6.     }
7.     public void setCount(int count) {
8.         this.count = count;
9.     }
10.    public CustomerServiceImpl(){
11.    }
12. }
```

How do we define bean in the configuration to initialize values?

```
1. package com.infy.util;
2. @Configuration
3. public class SpringConfiguration {
4.     @Bean // CustomerService bean definition using Setter Injection
```

```
5. public CustomerServiceImpl customerService() {  
6.     CustomerServiceImpl customerService = new CustomerServiceImpl();  
7.     customerService.setCount(10);  
8.     return customerService;  
9. }  
10. }
```

What is mandatory to implement setter injection?

Default constructor and setter methods of respective dependent properties are required in the CustomerServiceImpl class. For setter injection, Spring internally uses the default constructor to create a bean and then invokes a setter method of the respective property based on the name attribute in order to initialize the values.

Setter Injection-Reference datatype

So far, we learned how to inject primitive values using setter injection in Spring.

How do we inject object dependencies using setter injection?

Consider the CustomerServiceImpl class of InfyTel Customer application.

```
1. package com.infy.service;  
2. public class CustomerServiceImpl implements CustomerService {  
3.     private CustomerRepository customerRepository;  
4.     private int count;  
5.     public CustomerRepository getCustomerRepository() {  
6.         return customerRepository;  
7.     }  
8.     public void setCustomerRepository(CustomerRepository customerRepository) {  
9.         this.customerRepository = customerRepository;  
10.    }  
11.    public int getCount() {  
12.        return count;  
13.    }  
14.    public void setCount(int count) {  
15.        this.count = count;  
16.    }  
17. }
```

How do we inject object dependencies through configuration using setter injection?

```
1. package com.infy.util;  
2. @Configuration  
3. public class SpringConfiguration {  
4.     @Bean  
5.     public CustomerRepository customerRepository() {  
6.         return new CustomerRepository();  
7.     }  
}
```



```
8. @Bean // Setter Injection
9. public CustomerServiceImpl customerService() {
10. CustomerServiceImpl customerService = new CustomerServiceImpl();
11. customerService.setCount(10);
12. customerService.setCustomerRepository(customerRepository());
13. return customerService;
14. }
15. }
```

Setter Injection Demo: SpringConfiguration .java

```
1. package com.infy.util;
2. import org.springframework.context.annotation.Bean;
3. import org.springframework.context.annotation.Configuration;
4. import com.infy.repository.CustomerRepository;
5. import com.infy.service.CustomerServiceImpl;
6. @Configuration
7. public class SpringConfiguration {
8. @Bean // Setter Injection
9. public CustomerRepository customerRepository() {
10. CustomerRepository customerRepository = new CustomerRepository();
11. return customerRepository;
12. }
13. @Bean // Setter Injection
14. public CustomerServiceImpl customerService() {
15. CustomerServiceImpl customerService = new CustomerServiceImpl();
16. customerService.setCount(10);
17. customerService.setRepository(customerRepository());
18. return customerService;
19. }
20. }
```

CustomerService.java

```
1. package com.infy.service;
2. public interface CustomerService {
3. public String fetchCustomer();
4. public String createCustomer();
5. }
```

CustomerServiceImpl.java

```
1. package com.infy.service;
2. import com.infy.repository.CustomerRepository;
3. public class CustomerServiceImpl implements CustomerService {
4. private int count;
```

```
5. private CustomerRepository repository;
6. public CustomerServiceImpl() {
7. }
8. public void setCount(int count) {
9. this.count = count;
10. }
11. public void setRepository(CustomerRepository repository) {
12. this.repository = repository;
13. }
14. public String fetchCustomer() {
15. return repository.fetchCustomer(count);
16. }
17. public String createCustomer() {
18. return repository.createCustomer();
19. }
20. }
```

CustomerRepository.java

```
1. package com.infy.repository;
2. public class CustomerRepository {
3. public String fetchCustomer(int count) {
4. return " The no of customers fetched are : " + count;
5. }
6. public String createCustomer() {
7. return "Customer is successfully created";
8. }
9. }
```

Client.java

```
1. package com.infy;
2. import org.springframework.context.annotation.AnnotationConfigApplicationContext;
3. import org.springframework.context.support.AbstractApplicationContext;
4. import com.infy.service.CustomerServiceImpl;
5. import com.infy.util.SpringConfiguration;
6. public class Client {
7. public static void main(String[] args) {
8. CustomerServiceImpl service = null;
9. AbstractApplicationContext context = new
   AnnotationConfigApplicationContext(SpringConfiguration.class);
10. service = (CustomerServiceImpl) context.getBean("customerService");
11. System.out.println(service.fetchCustomer());
12. context.close();
13. }
```

```
14. }
```

Output:

```
1. The no of customers fetched are : 10
```

Why Auto Scanning is needed?

As discussed in the previous example as a developer you have to declare all the bean definition in SpringConfiguration class so that Spring container can detect and register your beans as below :

```
1. @Configuration
2. public class SpringConfiguration {
3.     @Bean
4.     public CustomerRepository customerRepository() {
5.         return new CustomerRepository();
6.     }
7.     @Bean
8.     public CustomerServiceImpl customerService() {
9.         return new CustomerServiceImpl();
10.    }
11. }
```

Is any other way to eliminate this tedious beans declaration?

Yes, Spring provides a way to automatically detect the beans to be injected and avoid even the bean definitions within the Spring configuration file through Auto Scanning. In Auto Scanning, Spring Framework automatically scans, detects, and instantiates the beans from the specified base package, if there is no declaration for the beans in the SpringConfiguration class.

So your SpringConfiguration class will be looking as below:

```
1. @Configuration
2. @ComponentScan(basePackages="com.infy")
3. public class SpringConfiguration {
4. }
```

Component scanning

Component scanning isn't turned on by default, however. You have to annotate the configuration class with **@ComponentScan** annotation to enable component scanning as follows:

```
1. @Configuration
2. @ComponentScan
3. public class SpringConfiguration {
4. }
```

In the above code, Spring will scan the package that contains SpringConfig class and its subpackages for beans. But if you want to scan a different package or multiple packages then you can specify this with the basePackages attribute as follows:

```
1. @Configuration
```

```

2. @ComponentScan(basePackages = "com.infy.service,com.infy.repository")
3. public class SpringConfiguration {
4. }

```

Spring uses `@ComponentScan` annotation for the auto scan feature. It looks for classes with the stereotype annotations and creates beans for such classes automatically.

Spring bean stereotype annotations

- Stereotype annotations denote the roles of types or methods at the conceptual level.
- Stereotype annotations are `@Component`, `@Service`, `@Repository`, and `@Controller` annotations.
- These annotations are used for auto-detection of beans using `@ComponentScan`.
- The Spring stereotype `@Component` is the parent stereotype.
- The other stereotypes are the specialization of `@Component` annotation.

Annotation	Usage
<code>@Component</code>	It indicates the class(POJO class) as a Spring component.
<code>@Service</code>	It indicates the Service class(POJO class) in the business layer.
<code>@Repository</code>	It indicates the Repository class(POJO class in Spring DATA) in the persistence layer.
<code>@Controller</code>	It indicates the Controller class(POJO class in Spring MVC) in the presentation layer.

- `@Component` is a generic stereotype for any Spring-managed component.
- `@Repository`, `@Service`, and `@Controller` are specializations of `@Component` for more specific use cases.
- `@Component` should be used when your class does not fall into either of three categories i.e. Controllers, Services, and DAOs.

Configuring IoC container using Java Annotation-based configuration

`@Component`: It is a general purpose annotation to mark a class as a Spring-managed bean.

```

1. @Component
2. public class CustomerLogging{
3. //rest of the code
4. }

```

`@Service` - It is used to define a service layer Spring bean. It is a specialization of the `@Component` annotation for the service layer.

```

1. @Service
2. public class CustomerServiceImpl implements CustomerService {
3. //rest of the code
4. }

```

`@Repository` - It is used to define a persistence layer Spring bean. It is a specialization of the `@Component` annotation for the persistence layer.

```

1. @Repository

```

```
2. public class CustomerRepositoryImpl implements CustomerRepository {
3. //rest of the code
4. }
```

@Controller - It is used to define a web component. It is a specialization of the @Component annotation for the presentation layer.

```
1. @Controller
2. public class CustomerController {
3. //rest of the code
4. }
```

By default, the bean names are derived from class names with a lowercase initial character. Therefore, your above defined beans have the names customerController, customerServiceImpl, and customerRepositoryImpl. It is also possible to give a specific name with a value attribute in those annotations as follows:

```
1. @Repository(value="customerRepository")
2. public class CustomerRepositoryImpl implements CustomerRepository {
3. //rest of the code
4. }
```

Note: As a best practice, use @Service for the service layer, @Controller for the Presentation layer, and @Repository for the Persistence layer.

Demop: AutoScanning

Highlights:

Objective: To understand the AutoScanning feature in Spring

Demo Steps:

SpringConfiguration .java

```
1. package com.infy.util;
2. import org.springframework.context.annotation.ComponentScan;
3. import org.springframework.context.annotation.Configuration;
4. @Configuration
5. @ComponentScan(basePackages="com.infy")
6. public class SpringConfiguration {
7. }
```

CustomerService.java

```
1. package com.infy.service;
2. public interface CustomerService {
3. public String fetchCustomer(int count);
4. public String createCustomer();
5. }
```

CustomerServiceImpl.java

```
1. package com.infy.service;
2. import org.springframework.beans.factory.annotation.Value;
3. import org.springframework.stereotype.Service;
4. @Service("customerService")
5. public class CustomerServiceImpl implements CustomerService {
6.     @Value("10")
7.     private int count;
8.     public String fetchCustomer(int count) {
9.         return " The no of customers fetched are : " + this.count;
10.    }
11.    public String createCustomer() {
12.        return "Customer is successfully created";
13.    }
14. }
```

Client.java

```
1. package com.infy;
2. import org.springframework.context.annotation.AnnotationConfigApplicationContext;
3. import org.springframework.context.support.AbstractApplicationContext;
4. import com.infy.service.CustomerServiceImpl;
5. import com.infy.util.SpringConfiguration;
6. public class Client {
7.     public static void main(String[] args) {
8.         CustomerServiceImpl service = null;
9.         AbstractApplicationContext context = new
            AnnotationConfigApplicationContext(SpringConfiguration.class);
10.        service = (CustomerServiceImpl) context.getBean("customerService");
11.        System.out.println(service.createCustomer());
12.        System.out.println(service.fetchCustomer(20));
13.        context.close();
14.    }
15. }
```

Output:

```
Customer is successfully created
The no of customers fetched are: 10
```