API AND MICROSERVICES
(Job Oriented Course)

Course Outcomes:

At the end of this course, the student will be able to

- Develop a Spring Data JPA application with Spring Boot
- Implement CRUD operations using Spring Data JPA
- Implement pagination and sorting mechanism using Spring Data JPA
- Implement query methods for querying the database using Spring Data JPA
- Implement a custom repository to customize a querying mechanism using Spring Data JPA
- Understand update operation using query approaches in Spring Data JPA
- Implement Spring Transaction using Spring Data JPA
- Develop RESTful endpoints using Spring REST Processing URI parameters
- Write RESTful services using Spring REST that consumes and produces data in different formats
- Handle exceptions and errors in Spring REST endpoints
- Write Spring based REST clients to consume RESTful services programmatically
- Create secure RESTful endpoints using Spring Security Document and version the Spring REST endpoints Implement CORS in a Spring REST application

UNIT I:

Spring 5 Basics : Why Spring, What is Spring Framework, Spring Framework - Modules, Configuring IoC container using Java-based configuration, Introduction To Dependency Injection, Constructor Injection, Setter Injection, What is AutoScanning

UNIT II:

Spring Boot: Creating a Spring Boot Application, Spring Boot Application Annotation, What is Autowiring , Scope of a bean, Logger, Introduction to Spring AOP, Implementing AOP advices, Best Practices : Spring Boot Application

UNIT III:

Spring Data JPA with Boot: Limitations of JDBC API, Why Spring Data JPA, Spring Data JPA with Spring Boot, Spring Data JPA Configuration, Pagination and Sorting, Query Approaches, Named Queries and Query, Why Spring Transaction, Spring Declarative Transaction, Update Operation in Spring Data JPA, Custom Repository Implementation, Best Practices - Spring Data JPA

UNIT IV:

Web Services: Why Web services, SOA - Service Oriented Architecture, What are Web Services, Types of Web Services, SOAP based Web Services, RESTful Web Services, How to create RESTful Services

UNIT V:

Spring REST: Spring REST - An Introduction, Creating a Spring REST Controller, @RequestBody and ResponseEntity, Parameter Injection, Usage of @PathVariable, @RequestParam and @MatrixVariable, Exception Handling, Data Validation, Creating a REST Client, Versioning a Spring REST endpoint, Enabling CORS in Spring REST, Securing Spring REST endpoints

Hardware and software configuration

- 4 or 8 GB RAM/126 GB ROM
- Swagger tool suite(opensource)
- OpenJDK 17 or Java 11,Maven 3.2 or above and MySQL 8.0 or above,Spring Tool suite, Postman

Text Books:

1. Spring in action, 5th Edition, Author: Craig Walls, Ryan Breidenbach, Manning books

Web Links [Courses mapped to Infosys Springboard platform]:

Infosys Springboard courses:

1. https://infyspringboard.onwingspan.com/en/app/toc/lex_auth_01296689056211763272_shared/overview [Spring 5 Basics with Spring Boot]
2. https://infyspringboard.onwingspan.com/en/app/toc/lex_4313461831752789500_shared/overview [Spring Data JPA with Boot]
3. https://infyspringboard.onwingspan.com/en/app/toc/lex_auth_01273190096390553619 0_shared/overview [Spring REST]

MEAN STACK DEVELOPMENT

UNIT I:

Spring 5 Basics : Why Spring, What is Spring Framework, Spring Framework - Modules, Configuring IoC container using Java-based configuration, Introduction To Dependency Injection, Constructor Injection, Setter Injection, What is AutoScanning

Spring

About Spring Basics

- Spring is a popular open source Java application development framework created by Rod Johnson. Spring supports developing any kind of Java applications 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 Spring umbrella to develop different kind of applications of today's need such as cloud based applications, mobile applications, batch application etc.
- Spring framework helps in developing a loosely coupled application which is simple, easily testable, reusable and maintainable.
- The basic Spring Framework is organized as loosely coupled modules, development team can choose modules based on the need. The fundamental module of framework is the core container providing core functionalities on which all other Spring modules/projects are based.

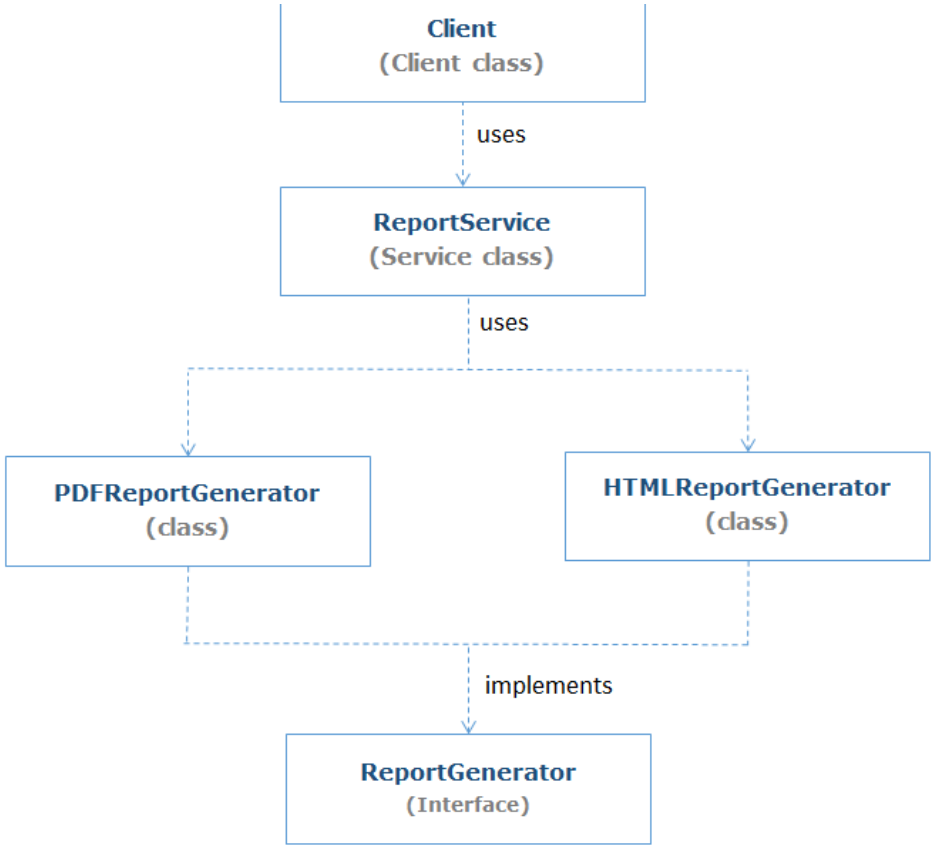
Spring

Report Generation Application

In this course, we will be using a simple Report Generation application scenario for learning the Spring core concepts.

Report Generation application is basically designed to generate reports in different formats such as PDF and HTML.

The application architecture is as shown below:



Application description

1. ReportGenerator : Interface with generateReport() method for report generation.
2. HTMLReportGenerator : Class for implementing ReportGenerator interface to generate report in Html format.
3. PDFReportGenerator: Class for implementing the ReportGenerator interface to generate the report in Pdf format.
4. ReportService: Class to provide report generation service to the client by using HTMLReportGenerator/PDFReportGenerator class.
5. Client: Class with client code to access report generation service to generate the report in PDF/HTML format.

Why Spring?

Spring

Let us consider an application scenario of generating a report in different formats such as PDF/HTML using Java.

Consider the following code:

ReportGenerator.java --> Interface

```java
package com.infosys.demo;

public interface ReportGenerator {
        public String generateReport(int recordsPerPage);
}

```

HTMLReportGenerator.java --> Implementation class for generating HTML reports

```java
package com.infosys.demo;
public class HTMLReportGenerator implements ReportGenerator{
        @Override
        public String generateReport(int recordsPerPage) {
                return "Generated HTML Report  with " + recordsPerPage + "records";
        }
}
```

Why Spring?

Spring

PDFReportGenerator.java --> Implementation class for generating PDF reports

```java
package com.infosys.demo;
public class PDFReportGenerator implements ReportGenerator{
        @Override
        public String generateReport( int recordsPerPage) {
                return "Generated PDF Report with " + recordsPerPage + "records";
        }
}
```

ReportService .java --> Service class to provide report generation service

```java
package com.infosys.demo;

public class ReportService {
    ReportGenerator master = new HTMLReportGenerator();   // Line1
    int recordsPerPage = 100;

    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Client.java --> Client code to access the report service

```java
package com.infosys.demo;
public class Client {
        public static void main(String[] args) {
                ReportService service = new ReportService();
                service.generateReport();
        }
}

```

Why Spring?

Spring

In the above code,

ReportService is responsible for instantiating HTMLReportGenerator to generate the report in an appropriate format which makes it tight coupled. This is bad design because of the following reasons:

- You cannot use a different implementation of ReportGenerator other than HTMLReportGenerator because of tight coupling. If the pdf report needs to be generated then Line1 of ReportService has to be modified.
- Also, if you want to unit test generateReport() method then you need a mock object of ReportGenerator. But you cannot use mock object because there is no way to substitute the HTMLReportGenerator object that ReportService has. So testing ReportService becomes difficult.

Refer complete source code of Report Generation application implemented using Java (**demo1-report-java**) from downloaded demos.

Are we developing a loosely coupled application in the above scenario?

The answer is No because code change is required for any changes to the dependencies. In a traditional way, it is the application developer's responsibility to manage the application object life cycle along with its dependencies. Hence the application code is not completely loosely coupled which results in more complexity as the application grows bigger.

Spring

Why Spring?

Is it possible to make an application more loosely coupled?

Yes, this can be done by moving the application object life cycle management responsibility to the third party such as a framework. Inversion of control(IoC) is the approach used to achieve this, there are different implementations for IoC and Spring Framework provides IoC implementation using **Dependency Injection(DI)**.

Spring's Dependency Injection features make an application loosely coupled by moving object dependencies away from the application to the Spring container. Now the container manages objects and its dependencies allowing the developer to focus more on application code.

You will be learning Dependency Injection in detail through this course.

Note: Inversion of Control (IoC) represents the inversion of application responsibility of the object's creation, initialization, dependency, and destruction from the application to the third party.

Spring

Why Spring?

Getting Started

1. Introduction to Spring Framework
2. Spring Inversion of Control (IoC)
3. Dependency Injection (DI)
4. Autowiring
5. Bean Scope
6. Spring Annotation and Java based configuration
7. Spring AOP

Spring Framework

Spring

Spring Framework is an **open source Java application development framework** which 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**.

Spring modules does not have tight coupling on each other, developer can pick and choose the modules as per the need for building an enterprise application.

Spring Framework

Spring

Spring Framework Features

Following are the main features of the Spring Framework.

Spring Feature	Description
Light Weight	Spring JARs are relatively small. A basic Spring framework would be lesser than 10MB. It can be deployed in Tomcat and they do not require heavy-weight application servers.
Non-Invasive	The application is developed using POJOs. (Plain OLD Java Object) No need to extend /implement any pre-defined classes.
Loosely Coupled	Spring features like Dependency Injection and Aspect Oriented Programming help in loosely coupled code.
Inversion of Control(IoC)	IoC takes care of the application object's life cycle along with their dependencies.
Spring Container	Spring Container takes care of object creation, initialization, and managing object dependencies.
Aspect Oriented Programming(AOP)	Promotes separation of supporting functions(concerns) such as logging, transaction, and security from the core business logic of the application.

Spring

Spring Framework - Modules

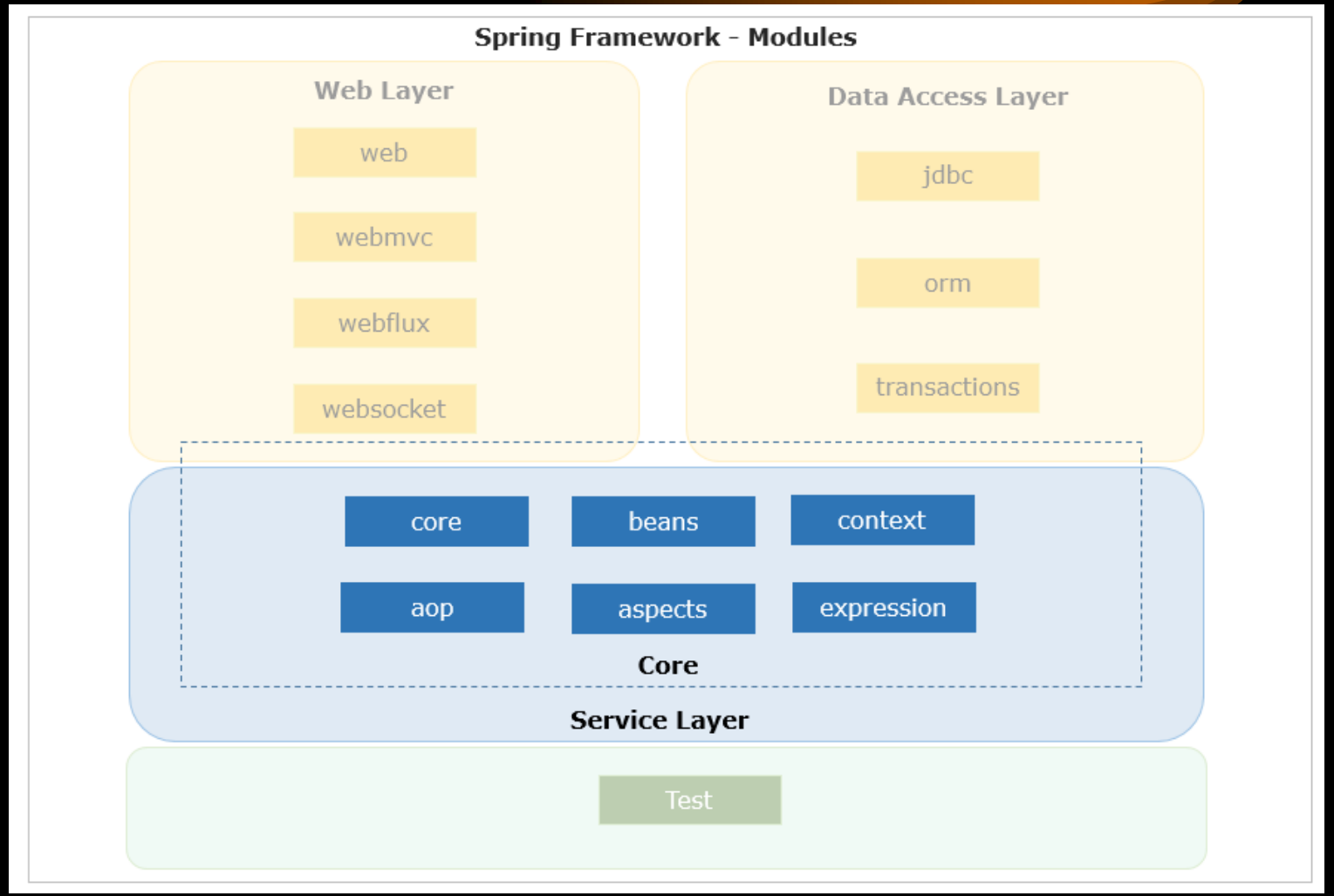
Spring Framework 5.x has the following key module groups:

- 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 Framework - Modules

Spring Modules - Core Container

Core Container of Spring framework provides the Spring IoC container and Dependency Injection features.



Spring Framework - Modules

Spring Modules - Core Container

Spring

Core container has the following modules:

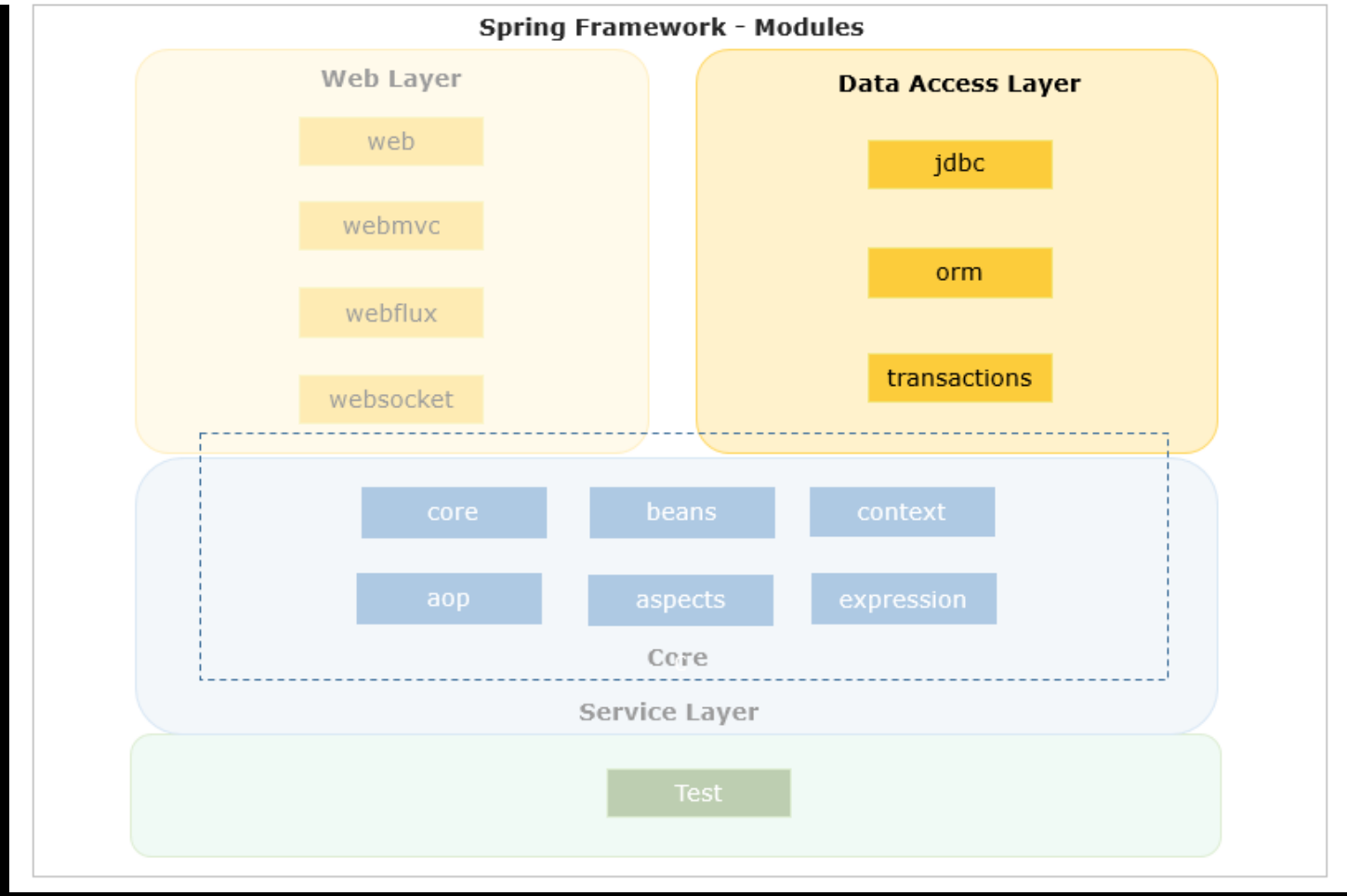
- Core: This is the key module of Spring Framework which provides fundamental support on which all other modules of the framework are dependent.
- Bean: This module provides a basic Spring container called BeanFactory.
- Context: This module provides one more Spring container called ApplicationContext which inherits the basic features of the BeanFactory container and also provides additional features to support enterprise application development.
- Spring Expression Language (SpEL): This module is used for querying/manipulating object values.
- AOP (Aspect Oriented Programming) and aspects: These modules help in isolating cross-cutting functionality from business logic.

Spring

Spring Framework - Modules

Spring Modules - Data Access/Integration

The Data Access/Integration module of Spring provides different data access approaches.



Spring

Spring Framework - Modules

Spring Modules - Data Access/Integration

The following modules support Data Access/Integration:

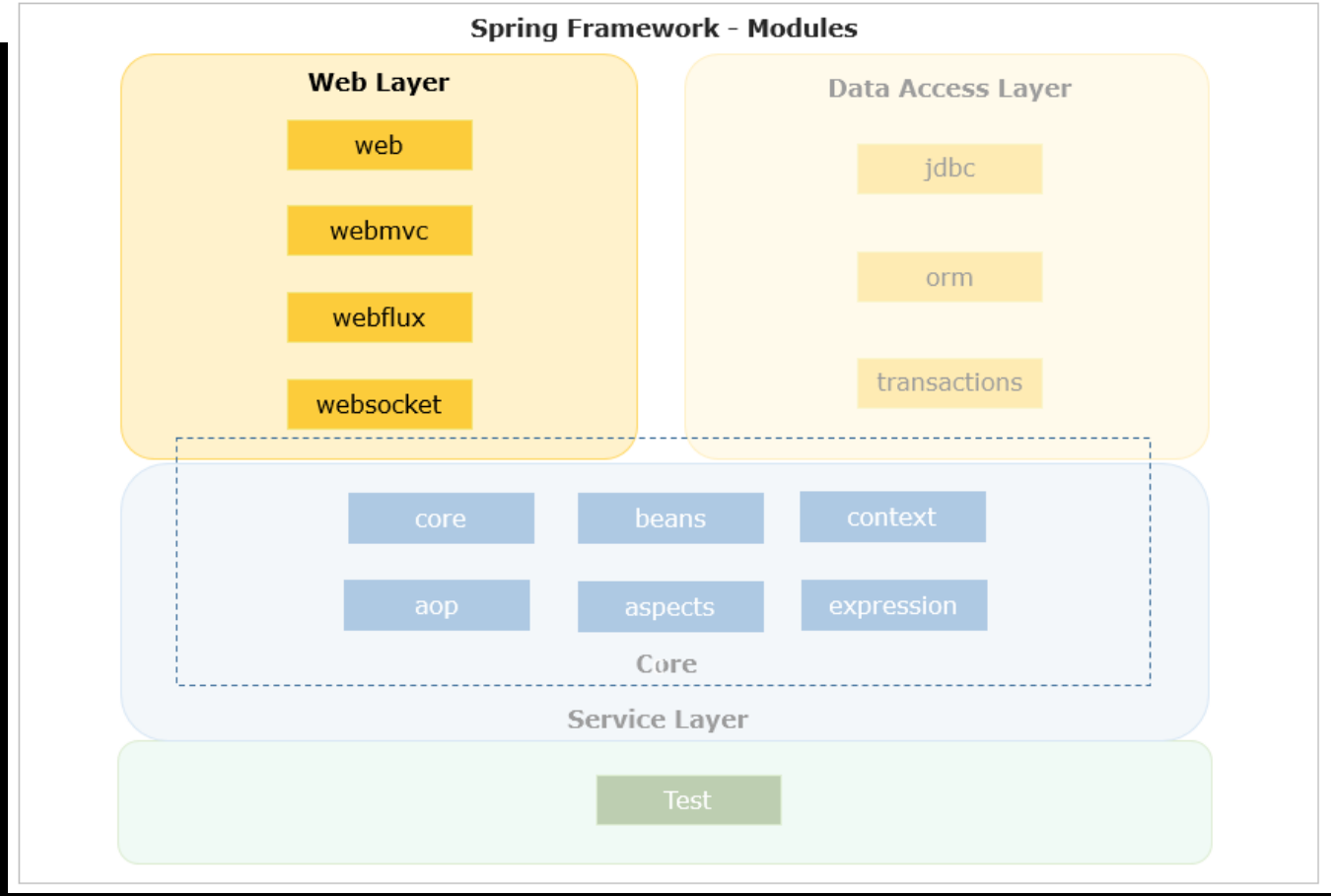
- Java Database Connectivity (JDBC): It provides an abstract layer to support JDBC calls to relational databases.
- Object Relational Mapping (ORM): It provides integration support for popular ORM(Object-Relational Mapping) solutions such as Hibernate, JPA, etc.
- Transactions: It provides a simple transaction API which abstracts the complexity of underlying repository specific transaction API's from the application.

Spring

Spring Framework - Modules

Spring Modules - Web

Spring Framework Web module provides basic support for web application development. The Web module has a web application context that is built on the application context of the core container. Web module provides complete Model-View-Controller(MVC) implementation to develop a presentation tier of the application and also supports a simpler way to implement RESTful web services.



Spring

Spring Framework - Modules

Spring Modules - Web

Spring Framework provides the following modules to support web application development:

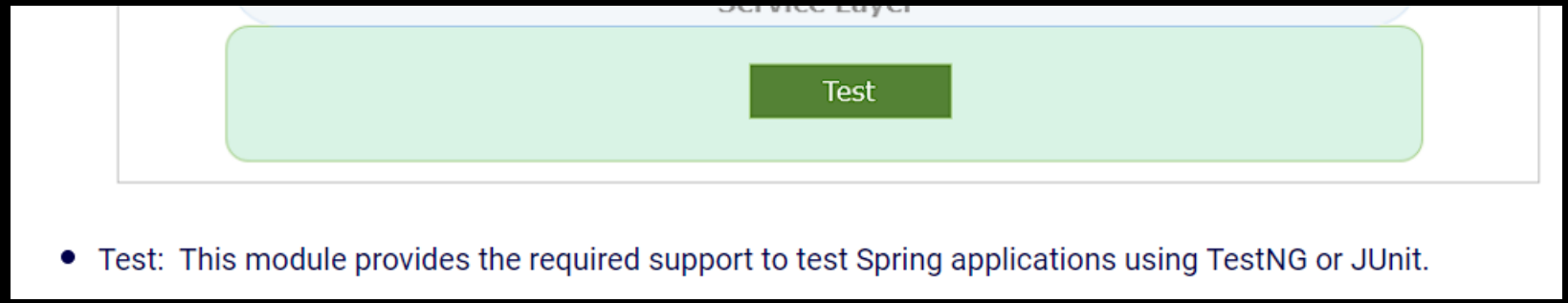
- Web: This module has a container called web application context which inherits basic features from ApplicationContext container and adds features to develop web based applications.
- Webmvc: It provides the implementation of the MVC(model-view-controller) pattern to implement the serverside presentation layer and also supports features to implement RESTful Web Services.
- WebFlux: Spring 5.0 introduced a reactive stack with a web framework called Spring WebFlux to support Reactive programming in Spring's web layer and runs on containers such as Netty, Undertow, and Servlet 3.1+.
- WebSocket: It is used for 2 way communication between client and server in WebSocket based web applications.

Spring

Spring Framework - Modules

Spring Modules – Other modules

Spring Framework has few additional modules, test module is one of the most commonly used ones for testing Spring applications.

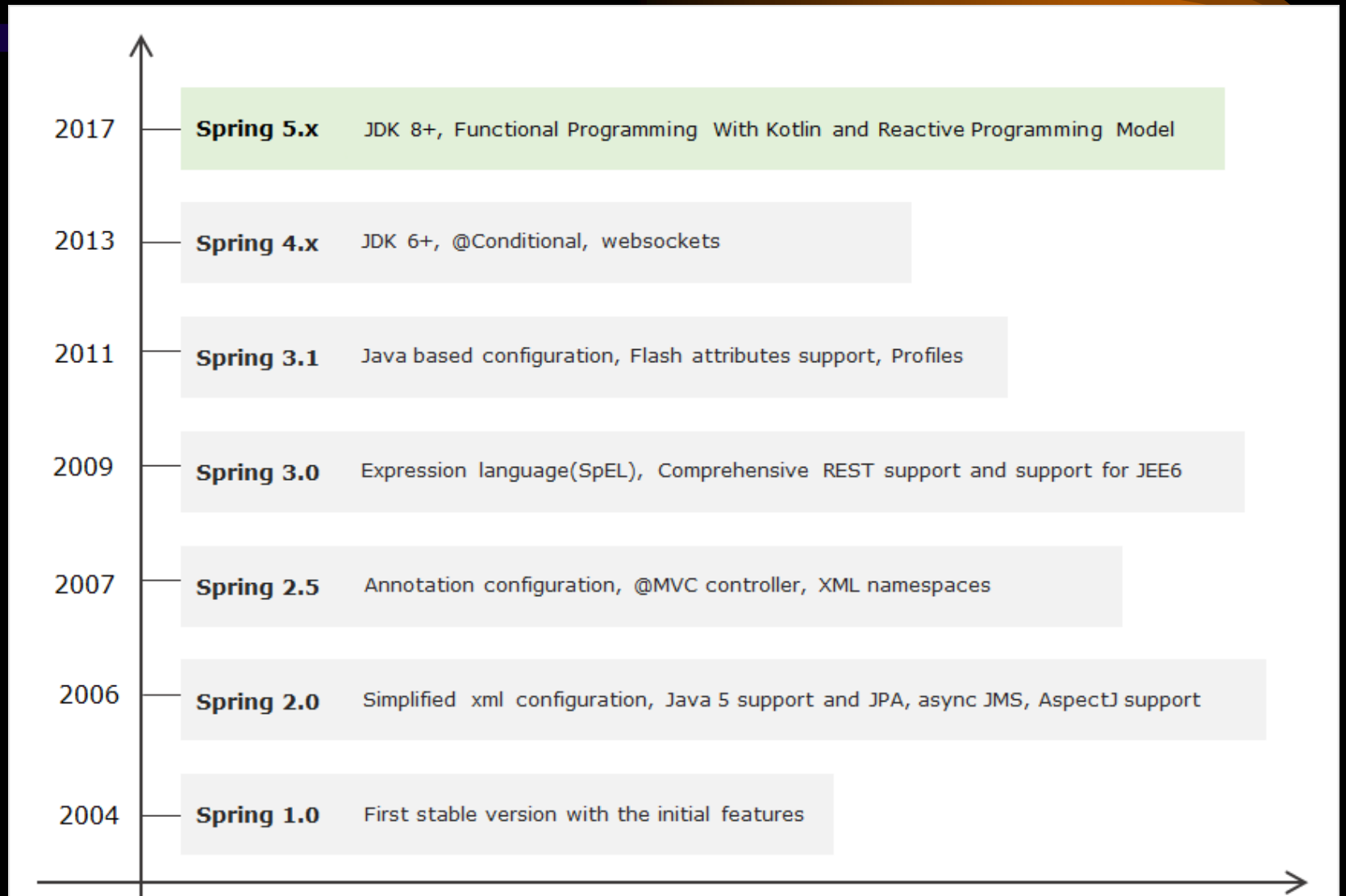


- Test: This module provides the required support to test Spring applications using TestNG or JUnit.

Spring

Spring Framework - Modules

Spring Version History



Spring Framework - Modules

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.

There are few more changes in Spring 5.x with respect to library support and discontinued support, you can refer to **the Spring documentation** for additional details.

Spring Inversion of Control(IoC)

As discussed earlier, Usually it is the developer responsibility to create the dependent application object using 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 application object's creation, initialization and destruction from the application to the third party such as framework. Now the third party takes care of application object management and dependencies there by 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)**.

Spring Inversion of Control(IoC)

Introduction to Spring Inversion of Control(IoC)

Spring

Spring Inversion of Control(IoC)

Introduction to Spring Inversion of Control(IoC)

Spring container managed application objects are called **beans** in Spring.

We need not create objects in depenedency 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):

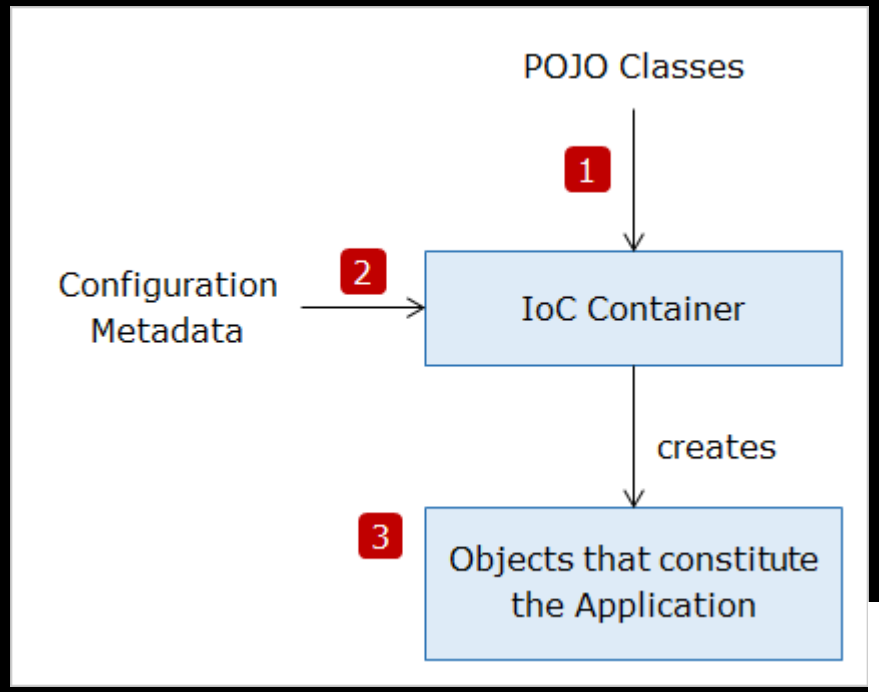
- Helps to create loosely coupled application architecture facilitating re-usability and easy testing.
- Separation of responsibility by keeping code and configuration separately. Hence dependencies can be easily modified using configuration without changing the code.
- Allows to replace actual objects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

Spring Inversion of Control(IoC)

Spring

The core container module of Spring framework provide 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

Spring Inversion of Control(IoC)

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 one more 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.

Spring

Spring Inversion of Control(IoC)

Spring IoC - Containers

org.springframework.context.support.ClassPathXmlApplicationContext is one of the most commonly used implementation of ApplicationContext.

Example: ApplicationContext container instantiation.

```
1 ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
2 Object obj = context.getBean("reportService");
```

1. ApplicationContext container is instantiated by loading the configuration from the config.xml file available in CLASSPATH of the application.
2. Accessing the bean with id "reportService" from the container.

Spring

Spring Inversion of Control(IoC)

BeanFactory Vs ApplicationContext

Let us now understand the differences between **BeanFactory** and **ApplicationContext** containers.

BeanFactory	ApplicationContext
Does not support annotation based Dependency Injection.	Support annotation based Dependency Injection.
Does not support enterprise services.	Support enterprise services such as validations, internationalization, etc.
By default it support Lazy Loading. **// Loading BeanFactory** BeanFactory factory = new ClassPathXmlApplicationContext ("config.xml"); **// Instantiating bean during first access using getBean()** ReportService reportService = factory.getBean("reportService");	By default it support Eager Loading.Beans are instantiated during load time. **// Loading ApplicationContext and instantiating bean** ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");

Spring

Spring Inversion of Control(IoC)

Configuration Metadata

The Spring configuration metadata consists of definitions of the beans that the container must manage.

Spring allows providing the configuration metadata using :

- XML configuration
- Annotation based configuration
- Java based configuration

Spring Inversion of Control(IoC) - XML based Configuration Metadata

In XML configuration, the configuration metadata is provided as a simple XML file describing bean definitions that the Spring container has to manage.

Basic XML based configuration structure is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<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 = "..." class = "..." >
        <!-- collaborators and configuration for this bean go here -->
    </bean>
    <bean id = "..." class = "..." >
        <!-- collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions go here -->
</beans>
```

1. <beans> is the root element & also includes namespace declarations
2. Bean definition
3. id attribute represents a unique bean identifier
4. class attribute represents a fully qualified class name

```
Id = "..." class = "..."
```

Spring

Spring Inversion of Control(IoC)

Demo : Spring IoC

Objective

To understand how to instantiate ApplicationContext, define a bean in the configuration file and also learn how to access the bean in Spring application.

Required Files

- ReportService.java
- config.xml
- Client.java

Required Jars

- spring-core-5.0.5.RELEASE.jar
- spring-beans-5.0.5.RELEASE.jar
- spring-context-5.0.5.RELEASE.jar
- spring-expression-5.0.5.RELEASE.jar
- spring-jcl-5.0.5.RELEASE.jar

Spring

Spring Inversion of Control(IoC)

Demo : Spring IoC

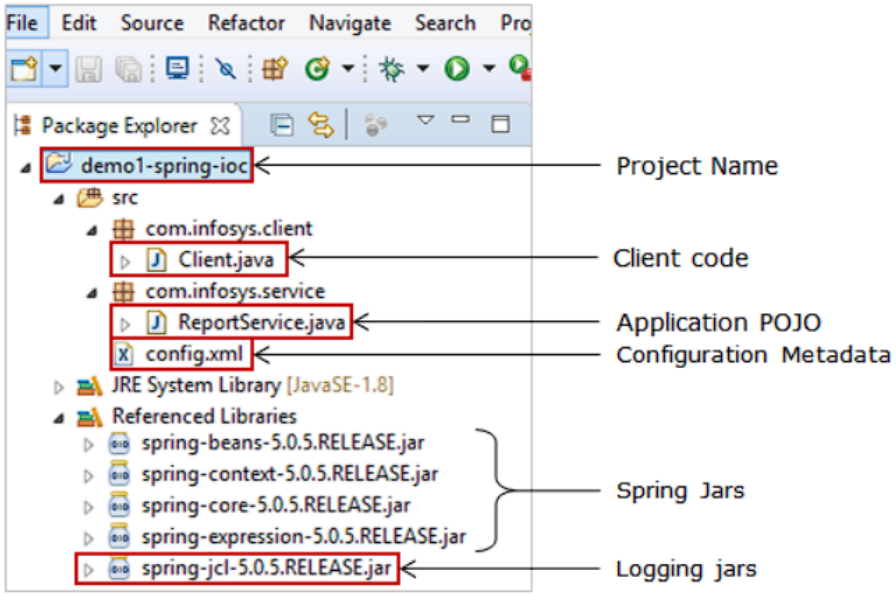
Demo Steps

Step 1: Create a project as follows

- File -> New -> Other -> Java Project -> Provide name of the project(demo2-spring-ioc) and click Finish

Step 2: Create a package "com.infosys.service" as below

- Right click on src folder -> New -> Package -> Provide package name(com.infosys.service) and click Finish



Spring

Spring Inversion of Control(IoC)

Demo : Spring IoC

Step 3: Create a class "ReportService.java" as below

- Right click on com.infosys.service package -> New -> Class -> Provide class name (ReportService.java) and click Finish. Write below code.

```
1. package com.infosys.service;
2.
3. public class ReportService {
4.
5.     public void display() {
6.         System.out.println("Hi, Welcome to Report Generation application");
7.     }
8. }
```

Spring Inversion of Control(IoC)

Demo : Spring IoC

Step 4: Create a configuration file "**config.xml**" as below with required bean definitions

- Right click on src folder New -> Other -> XML -> XML file -> click on Next -> Provide xml file name(config.xml) and click Finish. Write below code in "config.xml" file

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://www.springframework.org/schema/beans
5.     http://www.springframework.org/schema/beans/spring-beans.xsd">
6.
7.     <bean id="reportService" class="com.infosys.service.ReportService"/>
8.
9. </beans>
```

In the above given bean definition :

- reportService --> bean id
- com.infosys.service.ReportService --> Fully qualified class name

Best Practices

- It is a recommended good practice to have bean id same as class name lowering first letter.
- From Spring 4.x onwards, it is NOT recommended to specify the version details explicitly for the referenced xsd's in the declaration section. Spring takes care of version details from the application classpath.
- Spring configuration xml file name is user defined and hence you can give any meaningful name of your choice.

Spring Inversion of Control(IoC)

Demo : Spring IoC

Step 5: Follow the below step to add **required jars** to the classpath

- Right click on project -> Properties -> Java Build Path -> Choose "Libraries" option -> Click on "Add External JARs" -> Browse and add required jars and click OK

Step 6: Create a package "com.infosys.client" and then create a class "**Client.java**" in this package and below code.

```java
package com.infosys.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.infosys.service.ReportService;
/**
 * The Class Client.
 */
public class Client{

    public static void main(String[] args) {

        //ApplicationContext container is instantiated by loading the configuration from config.xml available in application classpath
        ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");

        // Accessing bean with id "reportService" and typecast from Object type to ReportService type
        ReportService reportService = (ReportService) context.getBean("reportService");
         // Invoking display method of ReportService to display greeting on console
       reportService.display();
    }

}
```

Spring Inversion of Control(IoC)

Demo : Spring IoC

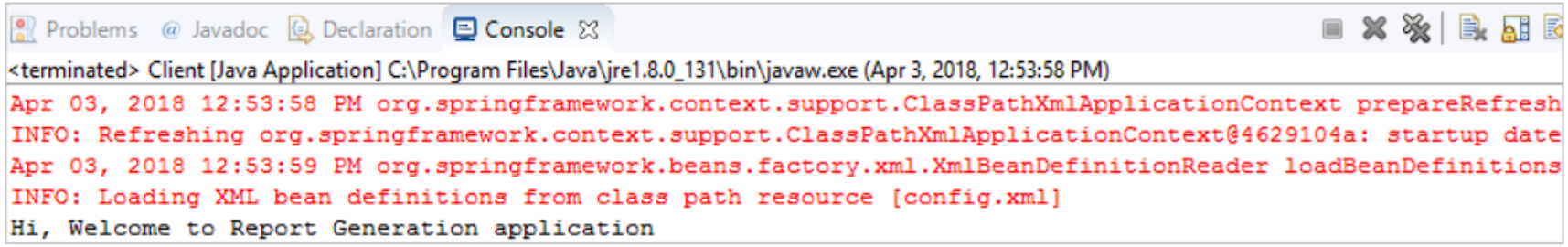
Spring

Step 7: Follow the below step to run the application

- Right click on Client file(which has main method) -> Run As -> Java Application

Output

- You can observe below response on the console.



Note:

Since we are not explicitly closing the context in the client code, there will be a warning for resource leakage, you can use one more interface from Spring "AbstractApplicationContext" as shown below which has a method to close the context explicitly.

```
1. public class Client{
2.     public static void main(String[] args) {
3.         AbstractApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
4.         ------------------
5.         context.close();
6.     }
7.
8. }
```

Dependency Injection

Spring

1. Introduction to Spring Framework
2. Spring Inversion of Control (IoC)
3. Dependency Injection (DI)
   - 3.1 Understand Dependency Injection (DI)
   - 3.2 Implement DI using
     - Constructor Injection
     - Setter Injection
4. Autowiring
5. Bean Scope
6. Spring Annotation and Java based configuration
7. Spring AOP

Dependency Injection

Spring

Let us now understand in detail about how to implement **Dependency Injection** in Spring.

For the previously discussed report generation application, we defined ReportService bean as shown below

```
1. <bean id = "reportService" class="com.infosys.demo.ReportService" >
2.
```

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

```
1. ReportService reportService = new ReportService();
2.
```

Dependency Injection

Spring

How do we initialize bean 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, 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 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 construct

Spring

Dependency Injection

Constructor Injection - Primitive values

Let us consider the ReportService class of Report Generation application to understand constructor injection.

ReportService class has a **recordsPerPage** property, let us now modify this class to initialize recordsPerPage property during bean instantiation using constructor injection approach.

```
1. package com.infosys.service;
2. public class ReportService {
3.     private int recordsPerPage;
4.     public ReportService(int recordsPerPage) {
5.                 this.recordsPerPage = recordsPerPage;
6.         }
7.     --------------------
8. }
```

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

```
1. <bean id = "reportService" class = "com.infosys.service.ReportService" >
2.       <constructor-arg value = "150"/>
3. </bean>
4.
```

As shown above, **constructor-arg** tag has to be used to inject the property value in the bean definition.

What is mandatory for constructor injection?

- Parameterized constructor is required in the ReportService class as we are injecting the values through the constructor argument
- constructor-arg tag in the bean definition

Spring

Dependency Injection

Constructor Injection - Primitive values

Can we use constructor injection to initialize multiple properties ?

Yes, we can initialize more than one property. Let us take an example to understand better.

Consider the ReportService class with two properties and parametarized constructor as shown below:

```
1.  package com.infosys.service;
2.
3.  public class ReportService {
4.
5.      /** The properties. */
6.      private int recordsPerPage ;
7.      private int totalRecords;
8.
9.     public ReportService(int recordsPerPage,  int totalRecords) {
10.           this.recordsPerPage = recordsPerPage;
11.           this.totalRecords = totalRecords;
12.     }
13.
```

Spring

Dependency Injection

Constructor Injection - Primitive values

Define a bean in the configuration using multiple constructor-arg tags to initialize multiple values through parameterized constructor.

```
1. <bean id="reportService" class="com.infosys.service.ReportService" >
2.       <constructor-arg value="100" />
3.       <constructor-arg value="500" />
4. </bean>
5.
```

Here first value "100" is passed to **recordsPerPage** and second value "500" to **totalRecords** of the parameterized constructor. By default, values are supplied based on the sequence in which the constructor-arg's are defined in a bean definition.

By default, constructor-arg value is of type String specified within double quotes(" "). In case of different constructor argument type, Spring takes care of converting assigned String value to appropriate constructor arguments type if it is Java compatible conversion otherwise throws an exception.

Spring

Dependency Injection

Constructor Injection - Primitive values

How does Spring resolve ambiguities if any in terms of passing values to parameterized constructor?

From Spring 3.0 onwards, constructor parameter **name** can be used for providing value as shown below. Here the values are passed to appropriate constructor arguments based on argument name instead of in the order of sequence mentioned in the bean definition.

constructor definition

```
1.    public ReportService(int recordsPerPage,  int totalRecords) {
2.            this.recordsPerPage = recordsPerPage;
3.            this.totalRecords = totalRecords;
4.     }
```

Bean Definition

```
1. <bean id="reportService" class="com.infosys.service.ReportService">
2.            <constructor-arg name="totalRecords" value="500" />
3.            <constructor-arg name="recordsPerPage" value ="100" />
4. </bean>
5.
```

However Spring also provides two more attributes "type" and "index" which can also be used to provide values with clarity.

Spring

Dependency Injection

Constructor Injection - Primitive values

Let us understand attribute "type". The <constructor-arg> tag provides attribute namely type, which can be used to indicate the type of argument explicitly.

Consider this example wherein ReportService has properties of different type as shown below:

```
1.  package com.infosys.service;
2.
3.  public class ReportService {
4.
5.      /** The properties. */
6.      private int recordsPerPage ;
7.      private double totalRecords;
8.
9.      public ReportService(int recordsPerPage,  double totalRecords) {
10.           this.recordsPerPage = recordsPerPage;
11.           this.totalRecords = totalRecords;
12.     }
13.
14.     ------------------------
```

Spring

Dependency Injection

Constructor Injection - Primitive values

Now, Spring passes values to constructor arguments based on matching type of argument instead of default sequence. Hence value "200.00" is assigned to 2nd argument of constructor and value "50" to the first argument considering the type of values.

```
1. <bean id="reportService" class="com.infosys.service.ReportService" >
2.          <constructor-arg value="200.00" type = "double"/>
3.          <constructor-arg value="50" type = "int" />
4. </bean>
5.
```

Spring

Dependency Injection

Constructor Injection - Primitive values

How does Spring resolves ambiguity among multiple arguments of same type?

To be more specific in passing the values, the <constructor-arg> tag provides "index" attribute to specify the index values of constructor arguments.

Consider this example:

```
1.  package com.infosys.service;
2.
3.  public class ReportService {
4.
5.      /** The properties. */
6.      private int recordsPerPage ;
7.      private int totalRecords;
8.
9.      public ReportService(int recordsPerPage,  int totalRecords) {
10.           this.recordsPerPage = recordsPerPage;
11.           this.totalRecords = totalRecords;
12.     }
13.
14.
```

Spring

Dependency Injection

Constructor Injection - Primitive values

Now, Spring pass values based on index of constructor arguments. Here, index is 0 based.

```
1. <bean id="reportService" class="com.infosys.service.ReportService" >
2.       <constructor-arg value="500" index="1" />
3.       <constructor-arg value="100" index="0" />
4. </bean>
5.
```

Here the value with index="0" is passed to first argument recordsPerPage and second value "500" to totalRecords of the parametarized constructor.

Dependency Injection

Constructor Injection - Primitive values

Spring

Consider the Report generation application discussed in why Spring section of this course.

ReportService .java class which is dependent on **ReportGenerator** object type to provide report generation in either HTML or PDF format.

```
1.  package com.infosys.demo;
2.  public class ReportService {
3.
4.      private ReportGenerator master;
5.      private int recordsPerPage;
6.
7.      public ReportService(ReportGenerator master, int recordsPerPage) {
8.          this.master = master;
9.          this.recordsPerPage = recordsPerPage;
10.     }
11.     public void generateReport() {
12.        System.out.println(master.generateReport(recordsPerPage));
13.      }
14. }
```

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

Spring

Dependency Injection

Constructor Injection - Primitive values

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

We can inject dependent object using **constructor-arg** tag with "**ref**" attribute. Following configuration is required for Spring to inject dependency of ReportGenerator through parameterized constructor.

```
1.  <bean id="reportService" class="com.infosys.demo.ReportService">
2.          <constructor-arg name="master" ref="pdfReportGenerator" />
3.          <constructor-arg name="recordsPerPage" value="150" />
4.
5.  </bean>
6.
7.  <!-- Bean of HTMLReportGenerator type -->
8.  <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
9.
10. <!-- Bean of PDFReportGenerator type -->
11. <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
12.
```

For object dependency, **constructor-arg** tag has to be used to inject the dependency by referring to the required bean present in the container using **ref** attribute. Here, pdfReportGenerator bean has been injected to reportService bean using **ref** attribute of constructor-arg tag in reportService bean definition.

Spring

Dependency Injection

Demo : Constructor Injection

Highlights:

- Objective :To understand the constructor based dependency injection in Spring
- Required Jars: Same as Demo1

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Spring

Dependency Injection

Demo : Constructor Injection

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
        @Override
        public String generateReport(int recordsPerPage) {
                return "Generated HTML Report  with " + recordsPerPage + " records";
        }
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
        @Override
        public String generateReport(int recordsPerPage) {
                return "Generated PDF Report with " + recordsPerPage + " records";
        }

}
```

Dependency Injection

Demo : Constructor Injection

ReportService.java --> Service class

```java
package com.infosys.demo;
public class ReportService {
    private ReportGenerator master;
    private int recordsPerPage;
    public ReportService(ReportGenerator master, int recordsPerPage) {
        System.out.println("Parameterized Constructor");
        this.master = master;
        this.recordsPerPage = recordsPerPage;
    }

    public ReportService() {
        System.out.println("Default Constructor");
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    public void setMaster(ReportGenerator master) {
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Spring

Dependency Injection

Demo : Constructor Injection

config.xml--> Spring configuration in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<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="reportService" class="com.infosys.demo.ReportService">
        <constructor-arg name="master" ref="pdfReportGenerator" />
        <constructor-arg name="recordsPerPage" value="150" />
    </bean>
    <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />

    <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
</beans>
```

Spring

Dependency Injection

Demo : Constructor Injection

Client.java --> Client Code

```java
package com.infosys.demo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Client {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
        ReportService srv = (ReportService)context.getBean("reportService");
        srv.generateReport();
    }
}
```

OUTPUT

```
1. Parameterized Constructor
2. Generated PDF Report with 150 records
3.
```

Spring

Dependency Injection

Setter Injection

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?

Bean definition with **property** tag is used for setter injection.

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

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

```
package com.infosys.demo;
public class ReportService {
    private int recordsPerPage;
     public int getRecordsPerPage() {
         return recordsPerPage;
     }
     public void setRecordsPerPage(int recordsPerPage) {
         this.recordsPerPage = recordsPerPage;
     }
    ------------------------
}
```

Spring

Dependency Injection

Setter Injection

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

```
1. <bean id="reportService" class="com.infosys.demo.ReportService" >
2.         <property name="recordsPerPage" value="500" />
3. </bean>
4.
```

As shown above, the **property** tag has to be used along with attribute **name** specifying the respective property name of the class to initialize the value.

What is mandatory to implement setter injection?

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

Spring

Dependency Injection

Setter Injection

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

How do we inject object dependencies using setter injection?

We can inject dependent object using **property** tag with "**ref**" attribute as shown in the below example.

Consider the ReportService class of Report generation application.

```
package com.infosys.demo;
public class ReportService {
    private ReportGenerator master;

    public void setMaster(ReportGenerator master) {
        this.master = master;
    }
    public ReportGenerator getMaster() {
        return master;
    }
   ----------------------------------
}
```

Dependency Injection

Setter Injection

Spring

Following configuration is required for Spring to inject dependency of ReportService through setter injection.

Let us assume HTMLReportGenerator and PDFReportGenerator classes are present.

```
1. <bean id="reportService" class="com.infosys.demo.ReportService">
2.         <property name="master" ref="htmlReportGenerator" />
3. </bean>
4.
5. <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
6.
7. <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
8.
```

For object dependency, **property** tag has to be used along with the name attribute to inject the dependency by referring to the required bean present in the container using **ref** attribute.

Here, Spring creates ReportService bean with required dependency by invoking default constructor and then setMaster() method of ReportService class to initialize the dependent bean.

Dependency Injection Setter Injection in Collections

Spring

So far you learned how to configure primitive data type using **value** attribute and object references using **ref** attribute of the <property> or <constructor-arg> tag in Spring configuration file.

Now let us look at how to Spring supports passing multiple values like Java Collection types such as List, Set, Map and Properties.

Spring provides four types of collection configuration elements as given below:

- <list> helps in injecting a list of values allowing duplicates
- <set> helps in injecting a set of values without any duplicates
- <map> helps in injecting a collection of key-value pairs where key and value can be of any type
- <props> helps in injecting a collection of key-value pairs where key and value are both Strings

Let us look at the below example with the list.

```
package com.infosys.demo;
public class ReportService {
    private List<ReportGenerator> reports;

    public void setReports(List<ReportGenerator> reports) {
        this.reports= reports;
    }
    // getter and setters
}
```

Spring

Dependency Injection

Setter Injection in Collections

Spring configuration is as shown below

```
1.  <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
2.
3.  <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
4.
5.  <bean id = "reportService" class = "com.infosys.demo.ReportService">
6.       <property name = "reports">
7.           <list>
8.              <ref bean = "PDFReportGenerator />
9.              <ref bean = "HTMLReportGenerator />
10.          </list>
11.       </property>
12. </bean>
13.
```

Spring

Dependency Injection

Setter Injection in Collections

Let us re-look at the Report Generation application implemented using Java without Spring. You can download code here.

Observation:

- The dependency of ReportService class on ReportGenerator type of object has been resolved using the new operator in the ReportService class

As discussed earlier, the use of new operator makes application tightly coupled and hence any changes to the dependencies require appropriate code change and becomes tedious as the application grows.

Now, let us re-look at the same application implemented using Spring. You can refer to the demo implemented as part of the constructor/setter injection.

Observation:

- The dependency of ReportService class on ReportGenerator type of object has been moved to Spring configuration

Now, the Spring container is responsible for instantiating and injecting required dependencies based on bean definitions provided through configuration. Hence the application is relieved from looking for dependent objects and also it is very easy to change the dependencies based on the need without doing much changes to the code.

Dependency Injection helped in creating a more loosely coupled application.

Spring

Dependency Injection

Comparison Constructor Injection and Setter Injection

Following are the comparison between Constructor Injection and Setter Injection

Constructor Injection	Setter Injection
Dependency Injection is through parameterized constructor	Dependency Injection is through setter methods after invoking the default constructor
Need parameterized constructor in the POJO class	Need default constructor and setter methods in the POJO class
<constructor-arg> tag is used in configuration file	<property> tag is used in configuration file
<constructor-arg> tag "ref" attribute is used to provide dependency for Object type	<property> tag "ref" attribute is used to provide dependency for Object type
Preferred for • mandatory dependencies • Immutable dependencies • concise(pass several parameters once)	• optional / changeable dependencies • circular dependencies

Spring

Introduction to Auto Scanning

Annotation based configuration using Autowiring reduces the size of XML configuration file by replacing the tags like <constructor-arg> and <property> using @Autowired annotation.

But still, the bean definitions using <bean> tag need to be defined with Spring configuration file.

Auto scanning helps to remove explicit bean definition using <bean> tag from the XML file.

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**.

Spring

Introduction to 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 XML file.

Auto Scanning can be switched on by using **<context:component-scan>**.

<context:component-scan> can also do whatever <context:annotation-config> does in addition to auto scanning.

Automatic discovery of beans can be achieved by doing the following:

- Use @Component annotation at POJO class level
- Include the following statement in the configuration file

```
1. <context:component-scan base-package = "packageName"/>
2.
```

Example:

```
1. <context:component-scan base-package = "com.mypack"/>
2.
```

In the above example, Spring scans specified "**com.mypack**" package and all its sub-packages to detect **@Component** annotated classes and create beans of these classes with default bean name same as the class name after making its first letter lowercase.

It is a best practice to mention the specific package name to scan in order to avoid unnecessary scanning.

Spring

Introduction to Auto Scanning

Auto Scanning - Annotations

<context:component-scan /> tag looks for classes with the following annotations and creates beans for such classes automatically.

- @Component: It indicates the class(POJO class) as a Spring component.
- @Controller: It indicates Controller class(POJO class in Spring MVC) in the presentation layer.
- @Repository: It indicates Repository class(POJO class in Spring DATA) in the persistence layer.
- @Service: It indicates Service class(POJO class) in the business layer.

Please refer to Spring MVC or Spring Data Access courses to know more details on Controller class and Repository class. Service classes refer to objects which capture all the domain logic of your application.

Introduction to Auto Scanning

Auto Scanning – Annotations Example

Spring

@Component is a generic annotation and other three are more specific annotations. For classes annotated with any of these annotations, Spring creates a bean with a default name which is same as the class name after making its first letter lowercase.

```
@Component("empUtil")
public class EmployeeUtil
{
        .....
        .....
}

@Controller
public class EmployeeController
{
        .....
        .....
}
```

```
@Repository
public class EmployeeRepository
{
        .....
        .....
}

@Service
public class EmployeeService
{
        .....
        .....
}
```

We can provide an explicit bean name "**empUtil**"

This overrides the default bean name "**employeeUtil**"

Spring

Introduction to Auto Scanning

To understand Spring Auto Scanning feature

In the Report Generation application, following is the configuration to provide required bean definitions.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.      xsi:schemaLocation="http://www.springframework.org/schema/beans
5.      http://www.springframework.org/schema/beans/spring-beans.xsd">
6.
7.     <bean id="reportService" class="com.infosys.demo.ReportService" >
8.              <property name="master" ref= "htmlGenerator"/>
9.              <property name="recordsPerPage" value= "500"/>
10.    </bean>
11.    <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
12.    <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
13.
14. </beans>
15.
```

Introduction to Auto Scanning

Spring

To understand Spring Auto Scanning feature

Let us now understand how **Auto Scanning feature** of Spring helps to remove explicit bean definition(using <bean> tag) from the XML file using simple configuration as shown below:

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xmlns:context="http://www.springframework.org/schema/context"
5.         xsi:schemaLocation="http://www.springframework.org/schema/beans
6.         http://www.springframework.org/schema/beans/spring-beans.xsd
7.         http://www.springframework.org/schema/context
8.         http://www.springframework.org/schema/context/spring-context.xsd">
9.
10.        <!--  Enable auto scanning feature in Spring -->
11.        <context:component-scan base-package="com.infosys" />
12.
13. </beans>
```

Use of **component-scan** tag from context namespace enable auto scanning feature

- Allows Spring to scan package "com.infosys" for the classes annotatated with any of these annotations @Component/@Service/@Repository/@Controller
- Helps in auto creation of beans for such classes with bean name same as class name lowering the first letter

Auto scanning helps programmer to provide simple configuration for the applications by eliminating lot of coding effort.

Introduction to Auto Scanning

Spring

Demo : Auto Scanning Feature of Spring

Highlights:

- To understand how Spring supports auto scan feature
- Required Jars: Same as previous demo

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
```

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```
1. package com.infosys.demo;
2.
3. @Component(value="htmlGenerator")
4. public class HTMLReportGenerator implements ReportGenerator{
5.         @Override
6.         public String generateReport(int recordsPerPage) {
7.                 return "Generated HTML Report with " + recordsPerPage + " records";
8.         }
9. }
```

Spring

Introduction to Auto Scanning

Demo : Auto Scanning Feature of Spring

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

@Component(value="pdfGenerator")
public class PDFReportGenerator implements ReportGenerator {
        @Override
        public String generateReport(int recordsPerPage) {
                return "Generated PDF Report with " + recordsPerPage + " records";
        }

}

```

Introduction to Auto Scanning

Demo : Auto Scanning Feature of Spring

ReportService.java --> Service class

```java
package com.infosys.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@Component(value="reportSrv")
public class ReportService {
    @Autowired
    @Qualifier("pdfGenerator")
    private ReportGenerator master;

    @Value("100")
    private int recordsPerPage;
    public ReportService() {
        System.out.println("default constructor");
    }
    public ReportService(ReportGenerator master) {
        System.out.println("constructor");
        this.master = master;
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
```

Spring

Introduction to Auto Scanning

Demo : Auto Scanning Feature of Spring

ReportService.java --> Service class

```
    public void setMaster(ReportGenerator master) {
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

config.xml--> Spring configuration in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.infosys.demo" />
</beans>
```

Spring

Introduction to Auto Scanning

Demo : Auto Scanning Feature of Spring

Client.java --> Client Code

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6.  public class Client {
7.      public static void main(String[] args) {
8.          ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
9.          ReportService srv = (ReportService)context.getBean("reportSrv");
10.         srv.generateReport();
11.     }
12. }
```

OUTPUT

```
1.  default constructor
2.  Generated PDF Report with 100 records
3.
```

Thank you