Deloitte.

Spring Framework

Deloitte Technology Academy (DTA)

Agenda

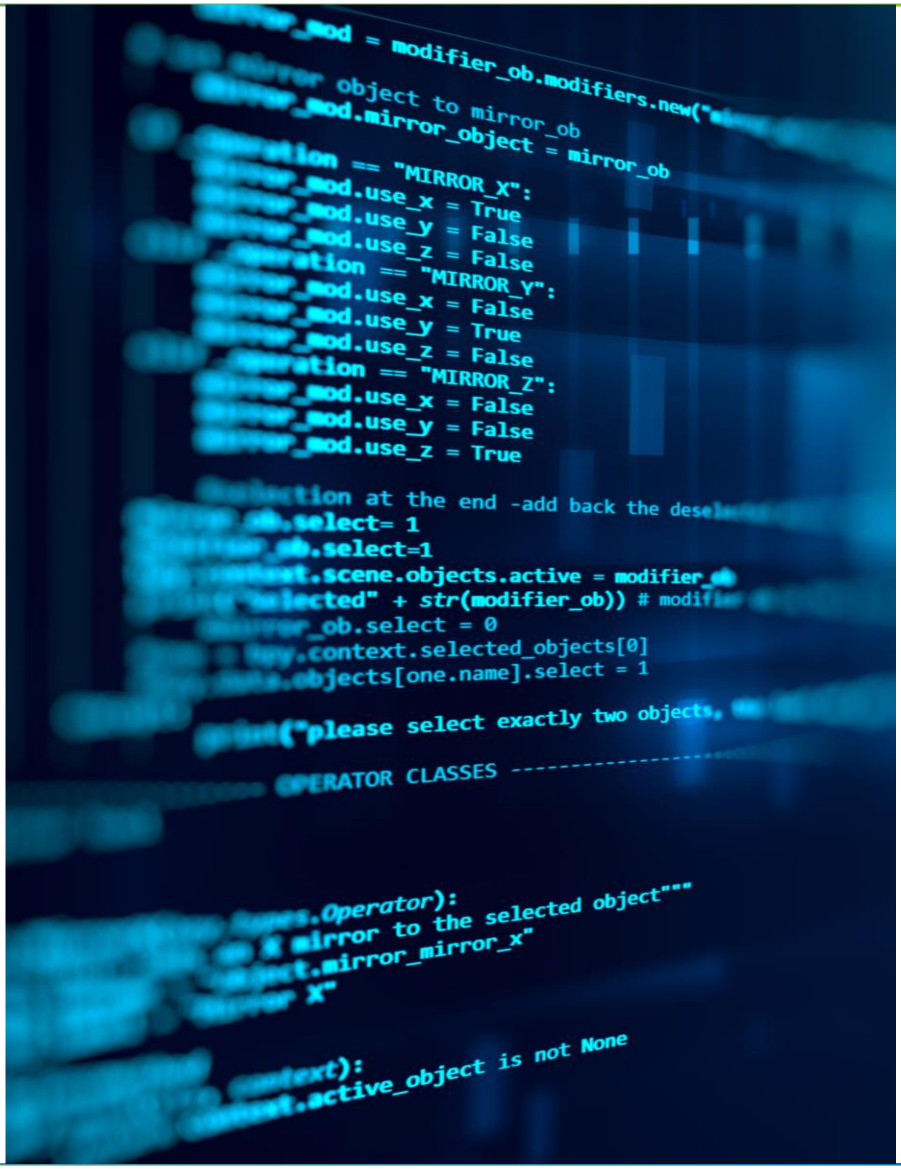
Topic	Descriptions	Duration
Introduction to Spring	Introduction to Spring, Model View Controller (MVC), and Data Access Object (DAO)	X hours XX minutes
Spring Versions	Spring Version Details	X hours XX minutes
IDE Setup	Setup Spring Tool Suite (STS) and Starting with Spring Framework	X hours XX minutes
Overview of IoC, DI, and Bean	Inversion of Control (IoC), Dependency Injection (DI)	X hours XX minutes
Spring Modules	Spring Modules, Bean Scope, and Lifecycle	X hours XX minutes
Autowiring	Spring Autowire, Autowire Using Annotations, and Autowire Using XML	X hours XX minutes
Introduction to AOP	Aspect Oriented Programming (AOP) Problem and Solution	X hours XX minutes
Spring: JDBC Template and ORM	Introduction, Features, and Benefits	X hours XX minutes

Learning Objectives

By the end of this session, you will be able to:

- Define Spring Framework
- Describe the purpose of Spring Framework
- Identify the features and eco-system of Spring Framework
- Explain cross-cutting concerns
- Identify gain implementation knowledge
- Explain a spring MVC application

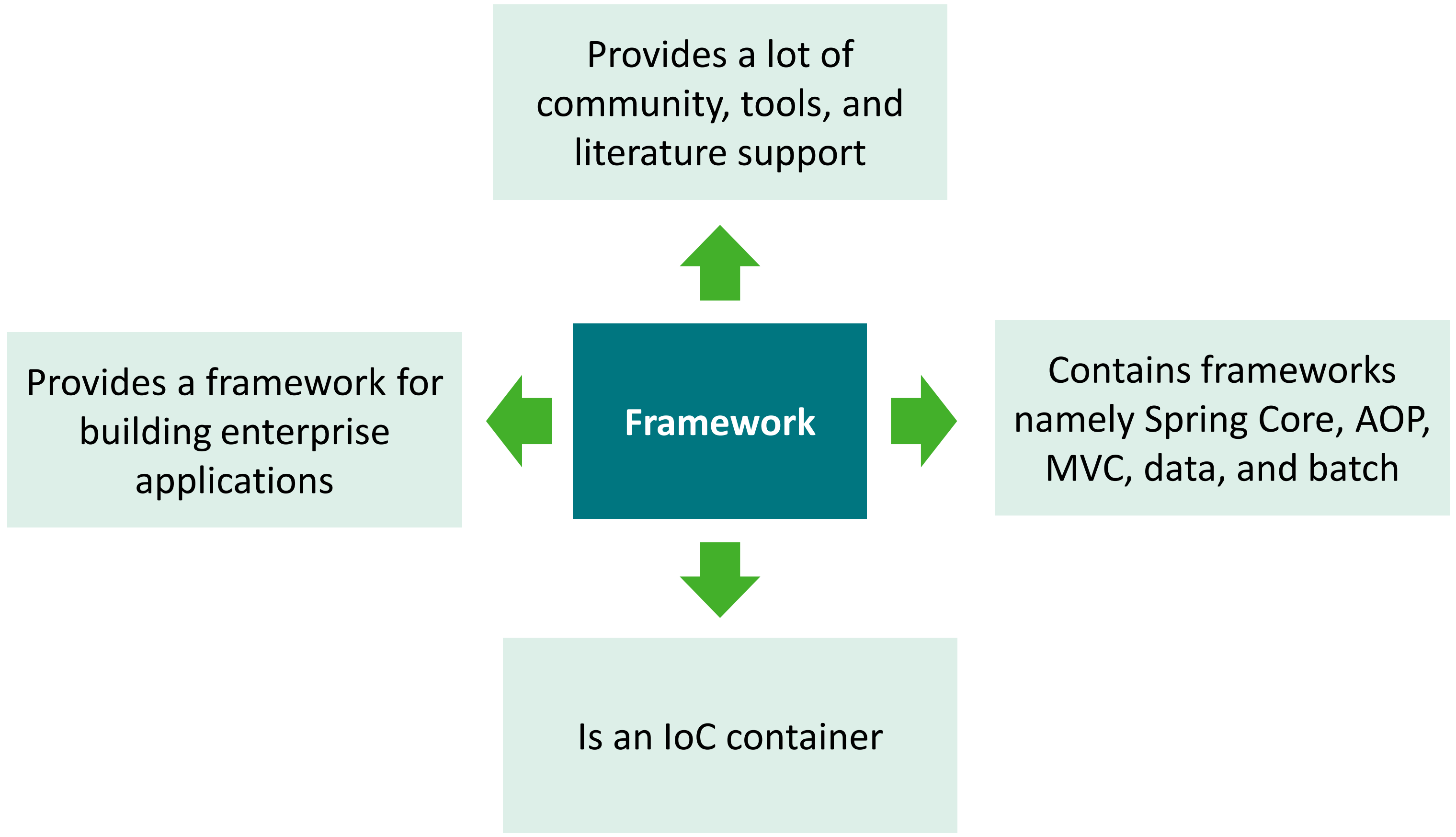
Introduction to Spring



Spring Introduction

Spring Framework	To develop Java applications comprehensive infrastructure support is provided by the Java Platform, Spring Framework. The latest version of Spring is Spring 5.2.
Inversion of Control (IoC) and Dependency Injection (DI)	The Dependency Injection (DI) is the most identified Spring technology a flavor of Inversion of Control. The Inversion of Control (IoC) can be expressed in many different ways. Dependency Injection is one concrete example of Inversion of Control.
Collection of Sub-Frameworks	Spring web MVC, Object Relational Mapping (ORM), Spring web flow, and Spring AOP are considered a collection of layers or sub-frameworks of the Spring Framework. The modules can be used separately or grouped together when constructing a web application.

Spring Framework


Provides a lot of community, tools, and literature support
Provides a framework for building enterprise applications
Framework
Contains frameworks namely Spring Core, AOP, MVC, data, and batch
Is an IoC container

Applications and Benefits

Applications

Plain Old Java Object (POJO) Based

- No interfaces
- Uses annotations and reflections

IoC Principle

- Dependency injection (better testability and reusability)
- Aspect weaving-cross cutting concerns

Cloud-Based Service

- Microservices for cloud

Plugins

- Maven
- Gradle

Benefits

Data Integration

- Java Database Connectivity (JDBC) data sources—MySQL, PostgreSQL, and Oracle
- JPA entity managers—Hibernate

Web Developments

- Traditional MVC applications-templates
- Representative State Transfer (REST) API JavaScript Object Notation (JSON)

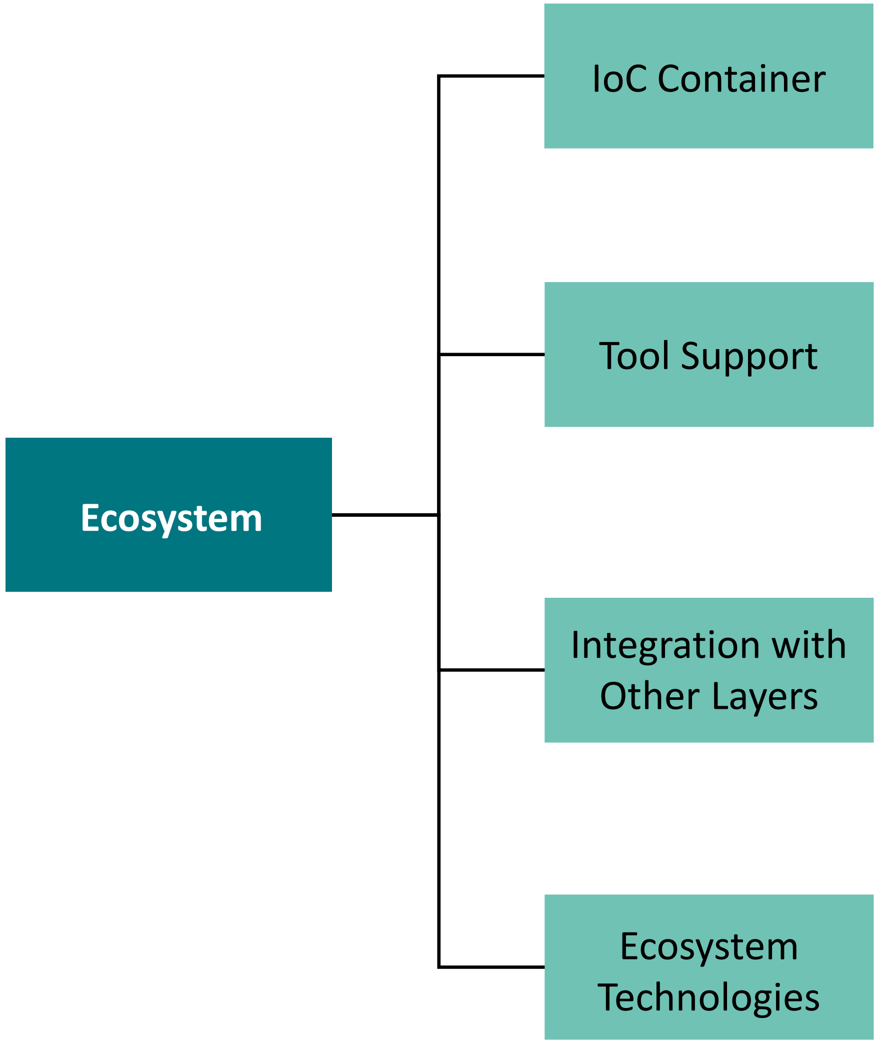
Modular in Nature

Only worry about the ones you need, even though the number of classes and package is substantial and ignore the rest.

Usage of Existing Frameworks

Spring makes use of the existing technologies like ORM frameworks, logging frameworks, Java Enterprise Edition (JEE), quartz, and Java Development Kit (JDK) timers

Ecosystem of Spring Framework



At the core, spring is just an IoC container

It provides tools support like:

- Spring tool suite-customized eclipse
- Git, maven, docker and more

It integrates with several other layers:

- Relational database management system (RDBMS) and Mongo
- Servlet containers, such as Tomcat
- Messaging systems, such as RabbitMQ
- Streaming systems, such as Kafka
- Cloud services, such as Netflix

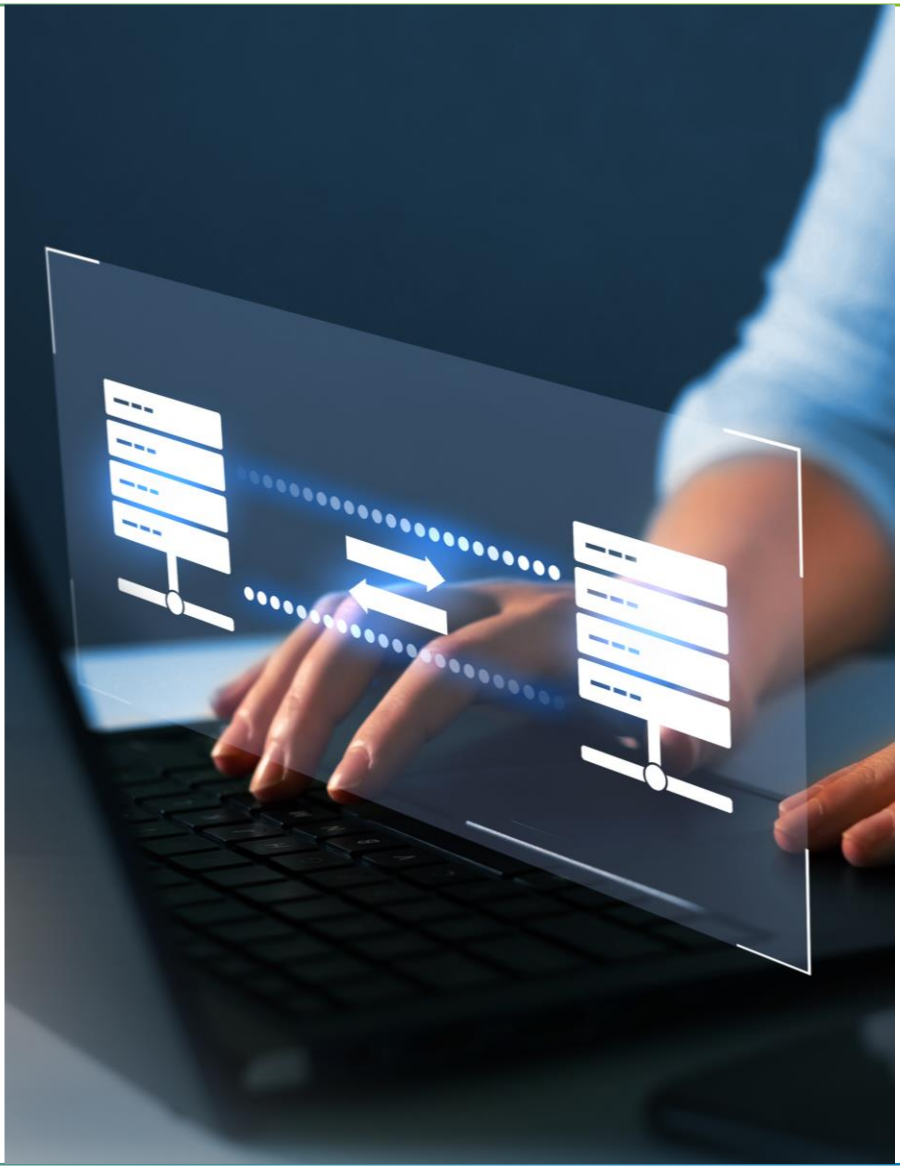
Spring demands expertise in ecosystem technologies

Spring Versions

Spring Version Details

- The first production release, Spring 1.0, was released in 2004.
- The second production release, Spring 2.0, was released in 2006.
- The third production release, Spring 3.0, was released in 2009.
- The fourth production release, Spring 4.0, was released in 2013.
- The fifth production release, Spring 5.0, was released in 2017.

IDE Setup



Setting Up DEV Environment

What do you need?

- Spring Tool Suite (STS)
- JDK 8 or later

Installing STS

- If STS isn't installed yet, visit the link to download a copy for your platform and to install it, simply unpack the downloaded archive
- When you are done, go ahead and launch STS

Basic Hello World Program

Once the installation is complete, you can go ahead and write and execute a basic hello world program and see how the Spring Framework functions

Steps to Run Hello World Program

Step 1: Specify the Spring version in POM.XML.

```
<properties>
       <java.version>1.8</java.version>
      <spring.version>5.2.0</spring.version>
</properties>
```

Step 2: Add the Spring dependencies to POM.XML.

```
<dependencies>
    </dependency>
        <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
        </dependency>
    <dependency>
   <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${spring.version}</version>
        </dependency>
        <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
        </dependency>
</dependencies>
```

Steps to Run Hello World Program (Cont.)

Step 3: Register the beans in beans.xml.

```
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:context = "http://www.springframework.org/schema
           /context"
   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-3.0
           .xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3
           .0.xsd">

   <bean id = "hello" class="java.lang.String">
   <constructer-arg>
   <value>Hello World</value>
   </constructer-arg>
   </bean>
</beans>
```

Step 4: Load the beans using application context.

```
package com.springpeople

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

public class SpringPeopleApplication{
        public static void main(String[] args)
        @SuppressWarnings("resource")
        ApplicationContext context=new
                ClassPathXmlApplicationContext("beans.xml");

        String hello = (String)context.getBean("Hello World");
        System.out.println(hello);
}
```

Overview of IoC, DI, and Bean

< / >

Inversion of Control (IoC) and Dependency Injection (DI)

What Is Inversion of Control (IoC)?

- Container assumes the responsibility of creating and controlling objects.
- Spring object container creates the object graph byDI.
- In contrast with traditional programming, IoC enables a framework that allows an opportunity to make calls to our custom code and to take control of the flow of a program.
- To enable this, frameworks use abstractions with additional behavior built in. If one wants to add his own behavior, he needs to extend the classes of the framework or plugin his own classes.
- IoC decouples the execution of a task from its implementation, making it easier to switch between different implementations.
- IoC helps in modularizing the program and eases out testing by isolating a component and allowing them to communicate through contracts.
- DI is a flavor of IoC.

What Is Spring Container?

- The core of the spring Framework is the spring container.
- The container will create the objects called spring Beans and manage their complete life cycle from creation to destruction using DI, it will wire them together and configure them.
- By reading the configuration metadata instructions the container knows what objects to instantiate, configure, and assemble.
- Java code, Java annotations, or XML can represent the configuration metadata
- To produce a fully configured and executable application or system the Spring IoC container uses the Java POJO classes and configuration metadata.
- The following two distinct types are provided by Spring; containers-spring bean factory container and spring application context container.

Code Demo

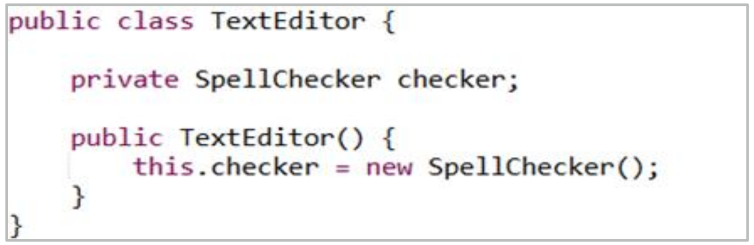
Use Case

Write a program that:

- Implements IoC
- Points out the difference between traditional programming and the IoC way of doing it

Example

The application has a text editor component, and one wants to provide spell checking. The standard code would look like this:

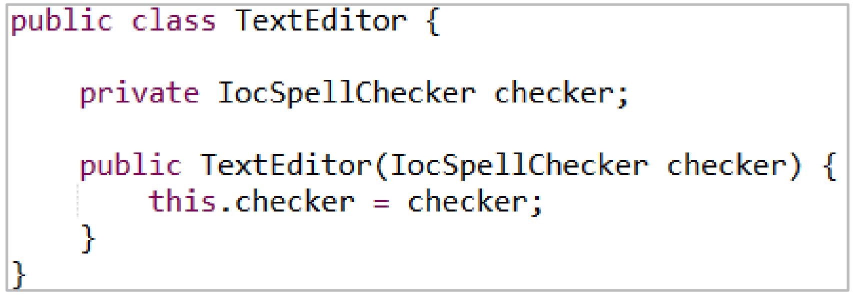


```
public class TextEditor {

    private SpellChecker checker;

    public TextEditor() {
        this.checker = new SpellChecker();
    }
}
```

As done in this example, it creates a dependency between the TextEditor and the SpellChecker

Example

In an IoC scenario, we would instead do like this:



```
public class TextEditor {

    private IocSpellChecker checker;

    public TextEditor(IocSpellChecker checker) {
        this.checker = checker;
    }
}
```

- In the first code example, we are instantiating `SpellChecker (this.checker = new SpellChecker();)`, which means the TextEditor class directly depends on the SpellChecker class.
- In the second code example, we are creating an abstraction by having the SpellChecker dependency class in the TextEditor's constructor signature (not initializing dependency in class). This allows us to call the dependency, and then pass it to the TextEditor class.

Overview of Dependency Injection (DI)

What is DI?

- DI is a pattern used to implement IoC, where the control being inverted is the setting of object's dependencies.
- The act of connecting objects with other objects, or "injecting" objects into other objects, is done by an assembler rather than by the objects themselves. Dependencies are resolved at runtime by the container, based on the configurations and using Java annotations and Java configuration.
- DI in spring can be done through constructors, setters, or fields.
- It is the flavor of IoC.
- Spring DI expects Java Beans and the usage of application context.

Need for DI

- Suppose Class 1 needs the object of Class 2 to instantiate or operate a method, then Class 1 is said to be dependent on Class 2. Though it might appear to be okay to make a module dependable on the other, but, in the real world, this could lead to a lot of problems, including system failure. Hence, such dependencies need to be avoided.
- Spring IoC resolves such dependencies with DI, which makes the code easier to test and reuse.
- DI in spring also ensures loose-coupling between the classes.

Types of DI

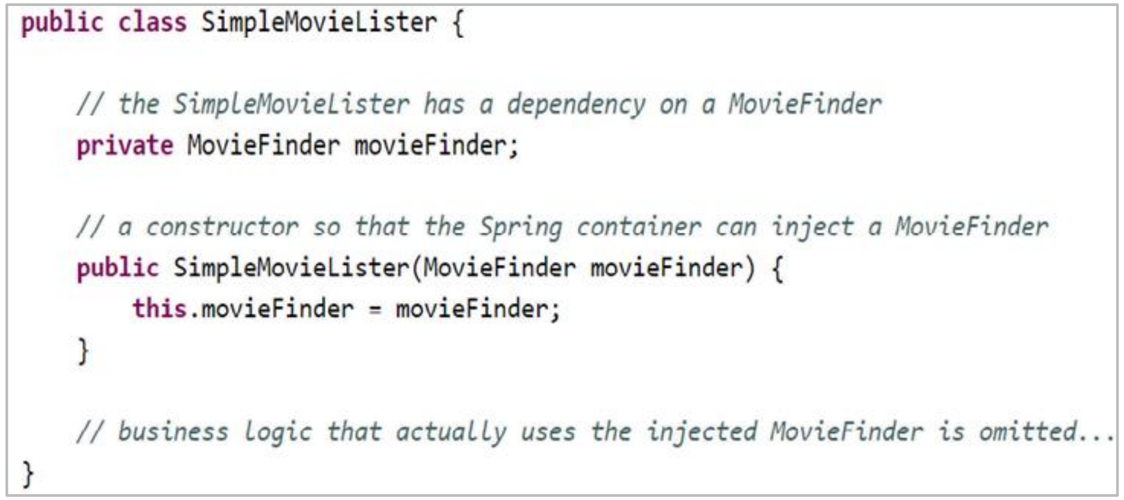
There are two types of DI-Constructor-Based DI and Setter-Based DI.

Types of DI

Constructor-Based DI

- When the container invokes a constructor with several arguments, and each represents a dependency then a constructor-based DI has been accomplished.
- If the bean is constructed by calling a static factory method with specific arguments, it is nearly equivalent. This discussion will also treat arguments to a constructor and to a static factory method.
- The example displayed shows a class that can only be dependency injected with the constructor injection.
- This class is a POJO that has no dependency on container-specific interfaces, base classes, or container-specific this class is nothing special.

Example



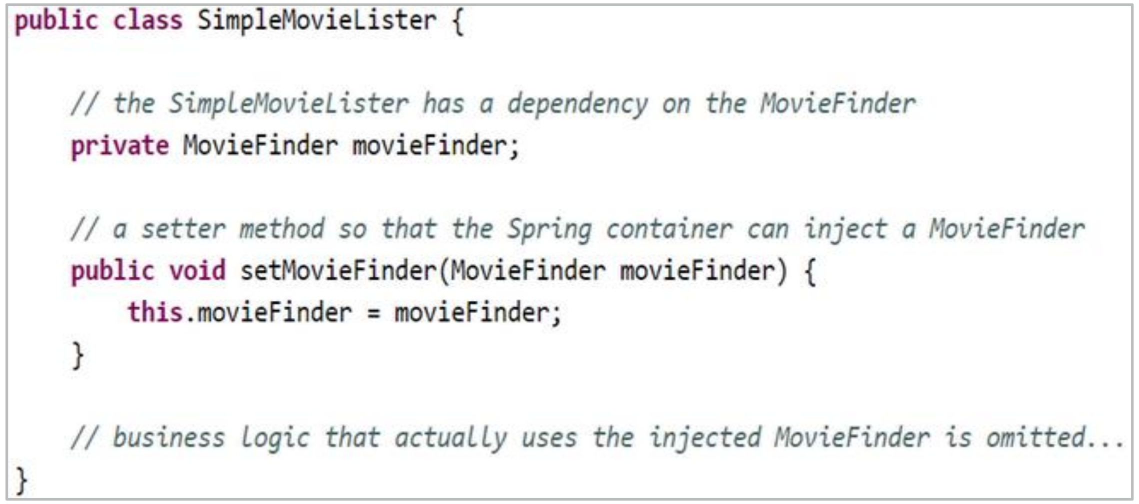
```java
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on a MovieFinder
    private MovieFinder movieFinder;

    // a constructor so that the Spring container can inject a MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Types of DI (Cont.)

Setter-Based DI

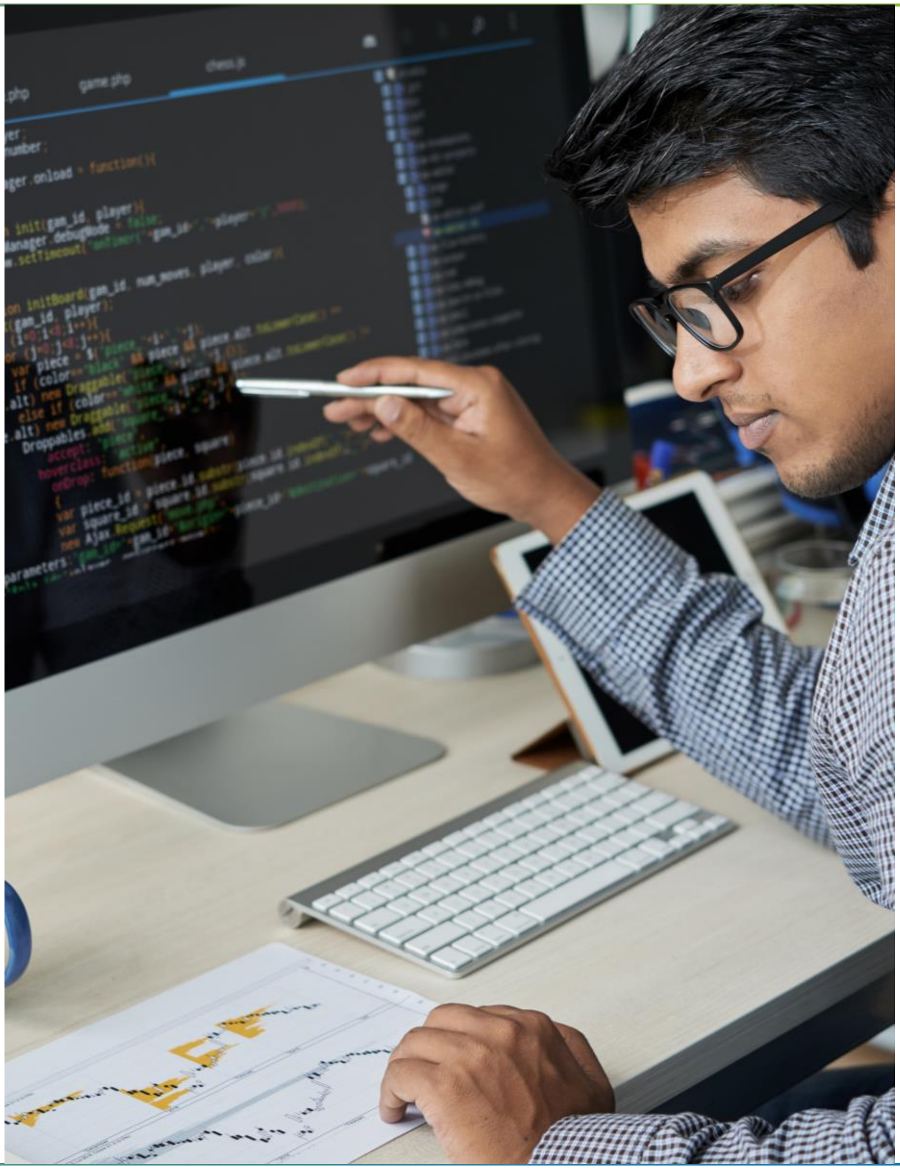
- When the container calls the setter methods on your bean after invoking a no-argument static factory or no-argument constructor method to instantiate your Bean then the setter-based DI is accomplished.
- The displayed example shows a class that can only be dependency injected using the pure setter injection.
- A conventional Java is this class. It has no dependencies on the container-specific interfaces, base classes, or annotations., so it is a POJO.
- You can mix both, constructor-based and setter-based DIs, but you should use constructor arguments for mandatory dependencies and setters for optional dependencies as a good rule of thumb.

Example



```java
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can inject a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Spring Modules

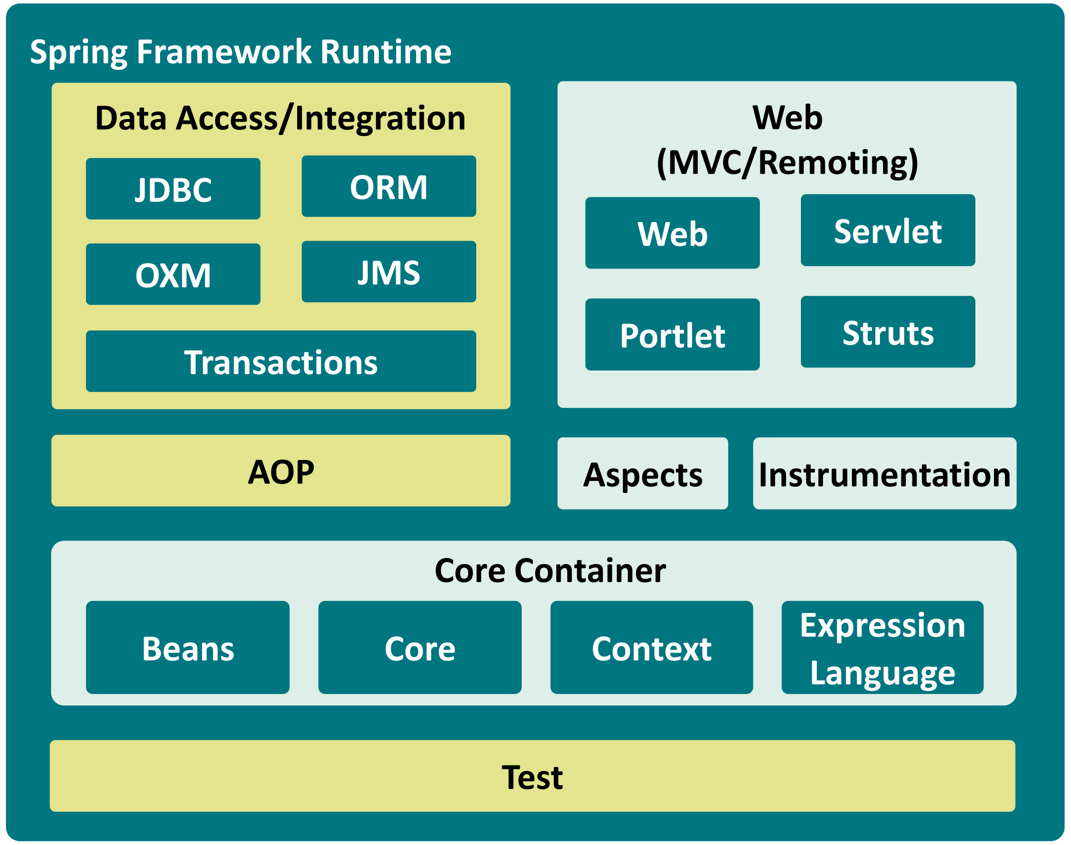


Spring Framework—Modules

Spring Framework

- Spring is modular, allowing one to pick and choose which modules are applicable to them, without having to bring in the rest.
- The Spring Framework provides about 20 modules which can be used based on an application requirement.
- The Spring Framework comprises of many modules such as core, beans, context, expression language, AOP, aspects, instrumentation, JDBC, ORM, OXM, JMS, transaction, web, servlet, struts, etc.
- These modules are grouped into core container, data access/integration, web, AOP (Aspect Oriented Programming), instrumentation, and test as shown in the adjacent diagram.

Architecture



Spring Framework—Overview

The core container consists of the core, context, beans, and expression language modules, the details of which are given below.

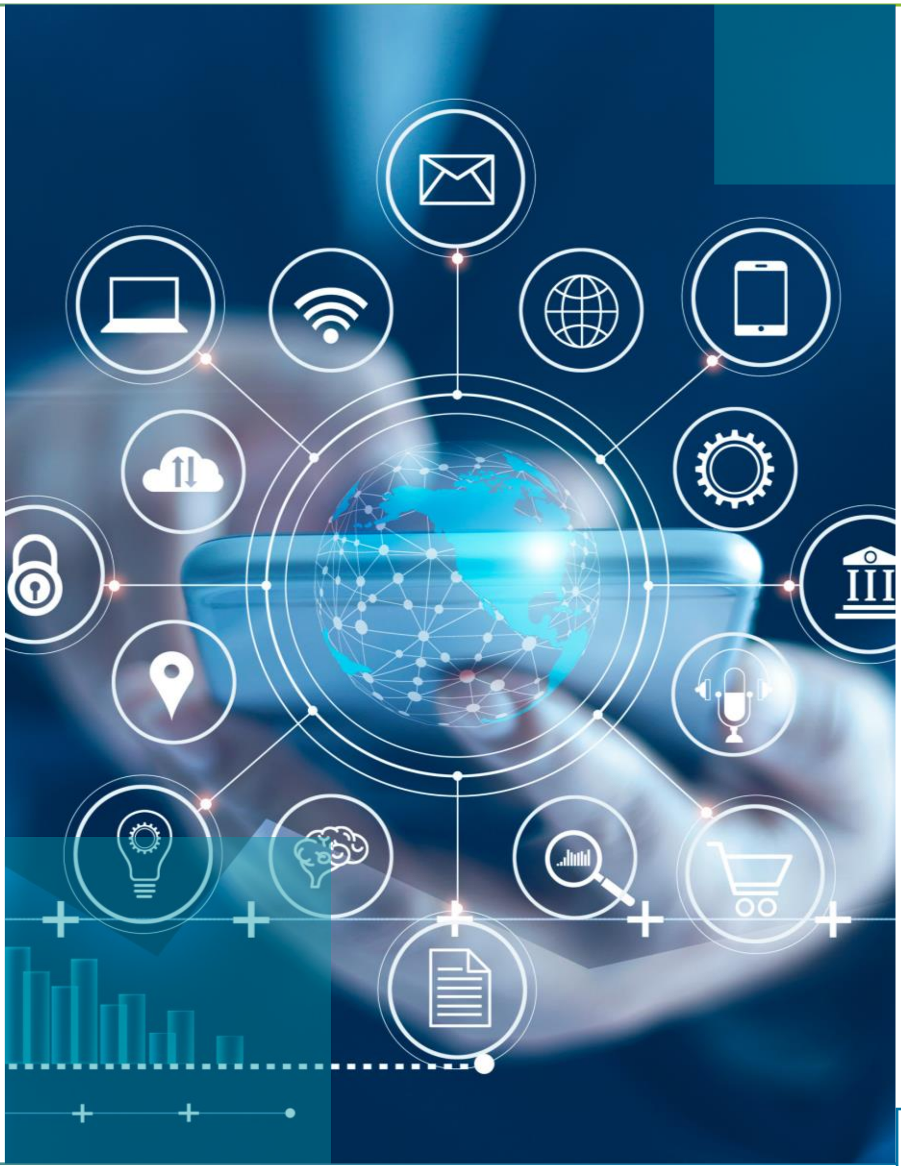
Core Container
The fundamental parts of the framework are provided in the core module, including the IoC and dependency injection features.
• An implementation of the factory pattern is provided by the bean module, BeanFactory. • It allows you to decouple the specification and configuration of dependencies from your actual program logic and removes the need for programmatic singletons.
• A context module is a medium to access defined and configured objects and builds on the solid base provided by the beans and core modules. • The focal point of the context module is the ApplicationContext interface.
• To query and manipulate an object graph at runtime a powerful expression language is provided by the expression language module. • It supports getting and setting property values, method invocation, property assignment, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from spring's IoC container.

Spring Framework—Overview (Cont.)

Data Access/Integration

The JMS, JDBC, OXM, ORM, and transaction modules consist in the data access/integration layer whose detail is given below.

- Removing the need for tedious JDBC related coding the JDBC module provides a JDBC-abstraction layer.
- Popular object-relational mapping APIS, including JDO, JPA, IBATIS, and hibernate are provided integration layers by ORM modules.
- object/xml mapping implementations for JIBX, JAXB, XMLBEANS, castor, and XSTREAM is provided by the OXM module as an abstraction layer that supports.
- Features for producing and consuming messages are contained in the JMS, Java messaging service
- Declarative and programmatic transaction management for classes are supported by the transaction module, that implement special interfaces and for all the POJOS.



Spring Framework—Overview (Cont.)

Web

The web, web-MVC, web-socket, and web-portlet modules consist in the web layer. The details are given below-

- Basic web-oriented integration features are provided by the Web model, such features include the multipart file-upload functionality and using a web-oriented application context and servlet listeners you can initialize the IoC container.
- The Web-MVC module contains spring's MVC, (Model-View-Controller) implementation for web applications.
- Support for webSocket-based, two-way communication between the server and the client in web applications is provided by the Web-Socket module.
- The MVC implementation can be used in a portlet environment and mirrors the functionality of Web-Servlet module provided by the Web-Portlet module.

Other modules

AOP, aspects, instrumentation, web, and test modules are a few other important modules. The details are given below-

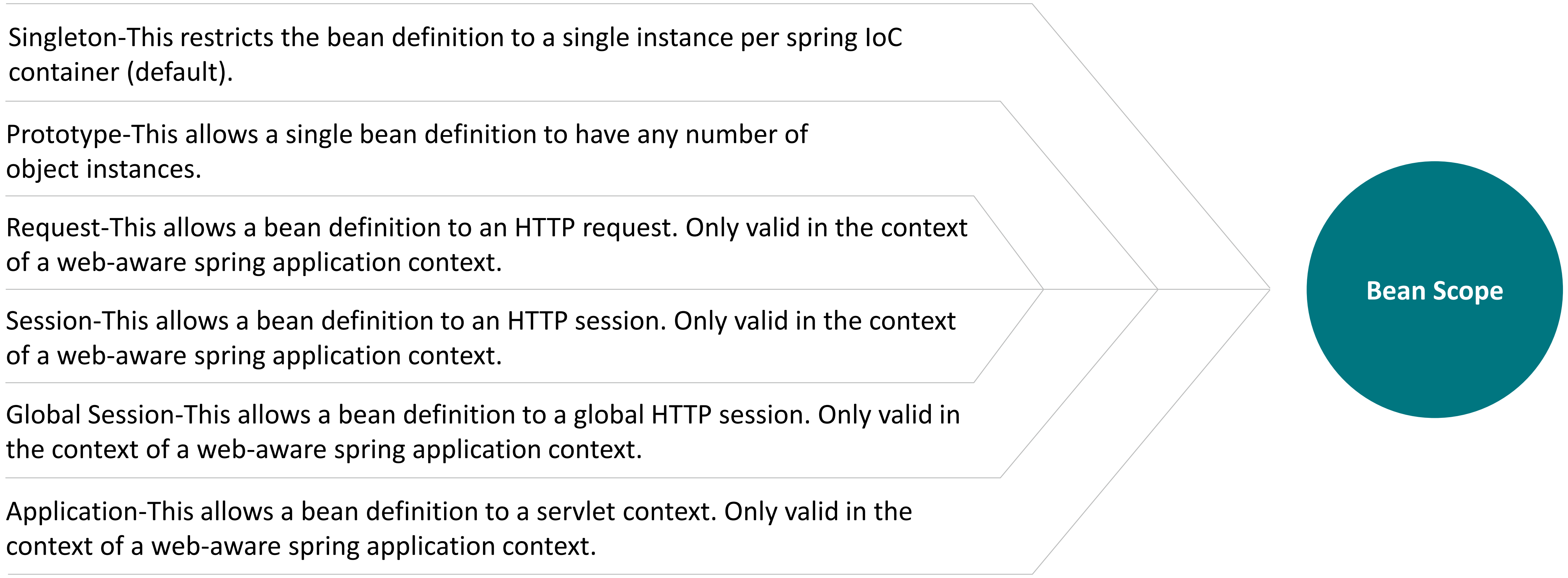
- An aspect-oriented programming implementation that allows you to define pointcuts and method-interceptors to decouple code cleanly to implement functionality that should be separated is provided by an AOP module.
- Integrated with AspectJ the aspects module provides a mature and powerful AOP framework.
- Used in certain application servers, instrumentation modules provide class loader implementation and class instrumentation support.
- Support for STOMP is provided by the messaging module as the WebSocket sub-protocol used in applications. STOMP messages from WebSocket clients can be processed and routed with support from an annotation programming model.
- The testing of spring components supported by the Test module with TestNG or JUnit frameworks.

Spring Bean—Scopes and Life Cycle

What is a Spring Bean?

- Beans are managed by the Spring IoC container and are objects that form the backbone of an application.
- Managed by the Spring IoC container, an object that is assembled and instantiated is a bean.
- The bean is created with the configuration metadata, that is supplied to the container.
- Bean definition contains the information called configuration metadata, which is needed for the container to know that how to create a bean, bean's lifecycle details and bean's dependencies.

Spring Bean—Scopes



Singleton-This restricts the bean definition to a single instance per spring IoC container (default).

Prototype-This allows a single bean definition to have any number of object instances.

Request-This allows a bean definition to an HTTP request. Only valid in the context of a web-aware spring application context.

Session-This allows a bean definition to an HTTP session. Only valid in the context of a web-aware spring application context.

Global Session-This allows a bean definition to a global HTTP session. Only valid in the context of a web-aware spring application context.

Application-This allows a bean definition to a servlet context. Only valid in the context of a web-aware spring application context.

Spring Bean—Scopes (Cont.)

The Singleton Scope

- Exactly one instance of the object is created by the spring IoC container defined by that bean definition if the scope is set to "singleton".
- Singleton beans are stored in a cache as a single instance where all subsequent references and requests to that named bean return the cached object. The default scope is always singleton.
- The scope property to singleton can be set if you need one and only instance of a bean in the bean configuration file shown below.

code snippet

```
<bean id = "..." class = "..." scope = "singleton">
</bean>
```

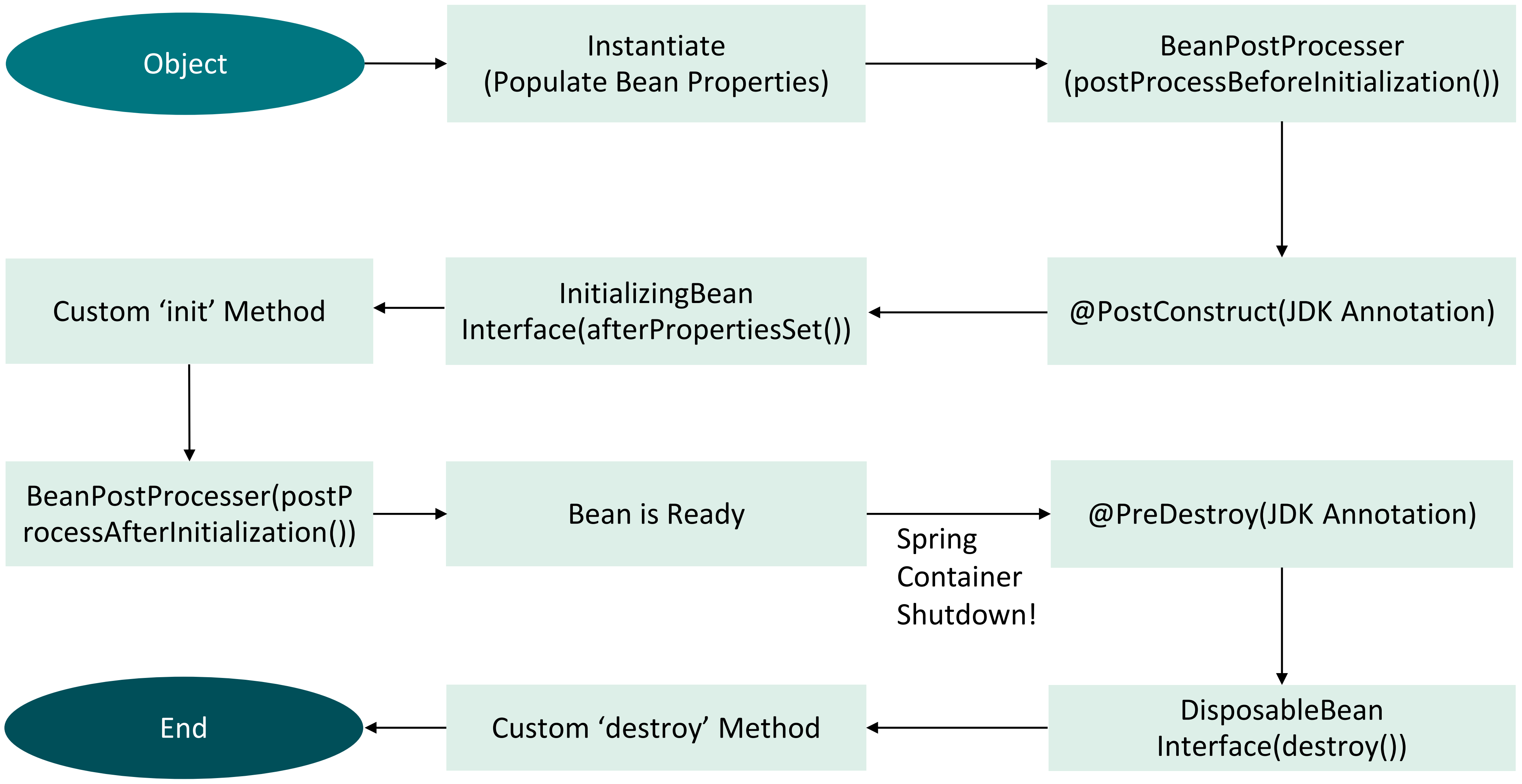
The Singleton Scope

- Every time a request is made for a specific bean the spring IoC container creates a new bean instance of the object if the scope is set to "prototype".
- By the rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.
- To define a prototype scope, one can set the scope property to prototype in the bean configuration file, as shown below.

code snippet

```
<bean id = "..." class = "..." scope = “prototype">
</bean>
```

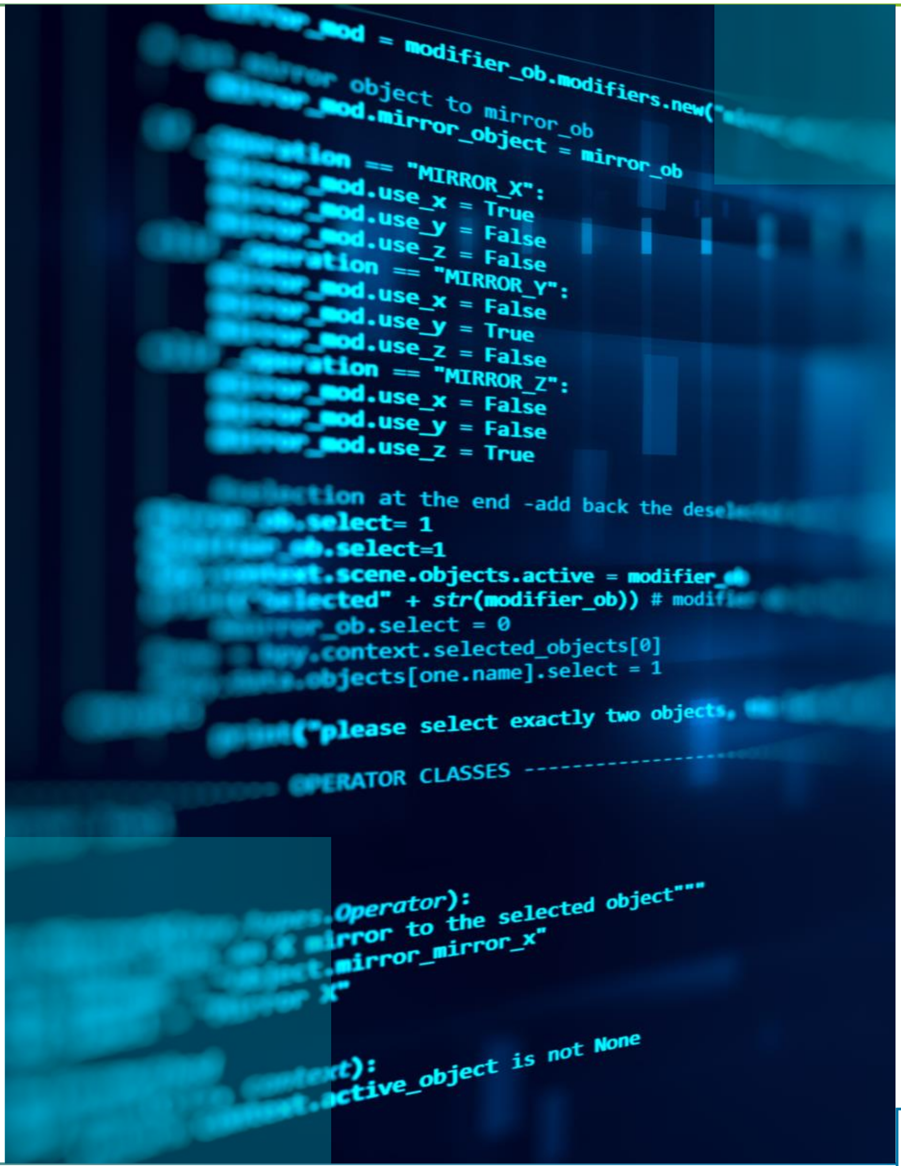
Spring Bean Lifecycle Diagram


Object
Instantiate (Populate Bean Properties)
BeanPostProcesser (postProcessBeforeInitialization())
@PostConstruct(JDK Annotation)
InitializingBean Interface(afterPropertiesSet())
Custom ‘init’ Method
BeanPostProcesser(postProcessAfterInitialization())
Bean is Ready
Spring Container Shutdown!
@PreDestroy(JDK Annotation)
DisposableBean Interface(destroy())
Custom ‘destroy’ Method
End

Bean Lifecycle

Lifecycle Callbacks

- By implementing the DisposableBean and InitializingBean interfaces you can interact with the container's management of the bean lifecycle.
- For the bean to be allowed to perform certain actions upon initialization and destruction of your bean the container calls AfterPropertiesSet() for the former and destroy() for the latter.
- One can also achieve the same integration with the container without coupling the classes to Spring interfaces using Init method and destroy method object definition metadata.
- The BeanPostProcessor implementations is used to process any callback interfaces it can find and call the appropriate methods by the Spring Framework internally.
- BeanPostProcessor can be implemented if you need a custom feature or other lifecycle behavior that isn't offered by spring out-of-the-box.
- Spring-managed objects can implement a lifecycle interface so they can participate in the startup and shutdown process in addition to the initialization and destruction callbacks as driven by the container's own lifecycle.



Bean Lifecycle (Cont.)

Initialization Callbacks

- A bean is allowed to perform initialization work after all necessary properties have been set by the container using the org.springframework.beans.factory.InitializingBean interface. The InitializingBean interface specifies a single method.

```
void afterPropertiesSet() throws Exception;
```

- It is recommended not to use the InitializingBean interface because it unnecessarily couples the code to spring. Alternatively, it specifies a POJO initialization method. The init method attribute should be used in the case of XML-based configuration metadata for methods that have a void no-argument signature to specify the name of the method.

Example

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

public class ExampleBean {

  public void init() {
      // do some initialization work
  }
}
```

The above example is same as the below snippet but does not couple the code to spring.

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

  public void afterPropertiesSet() {
      // do some initialization work
  }
}
```

Bean Lifecycle (Cont.)

Destruction Callbacks

- When a container containing a bean is destroyed the bean can get a callback by implementing the org.springframework.beans.factory.DisposableBean interface. A single method is specified by the DisposableBean interface.

```
void destroy() throws Exception;
```

- The code to spring is unnecessarily coupled when using the DisposableBean callback interface so it is recommended that you do not use it. Instead specify a supported bean definition generic method. Use the destroy-method attribute on the <bean/> with XML-based configuration metadata

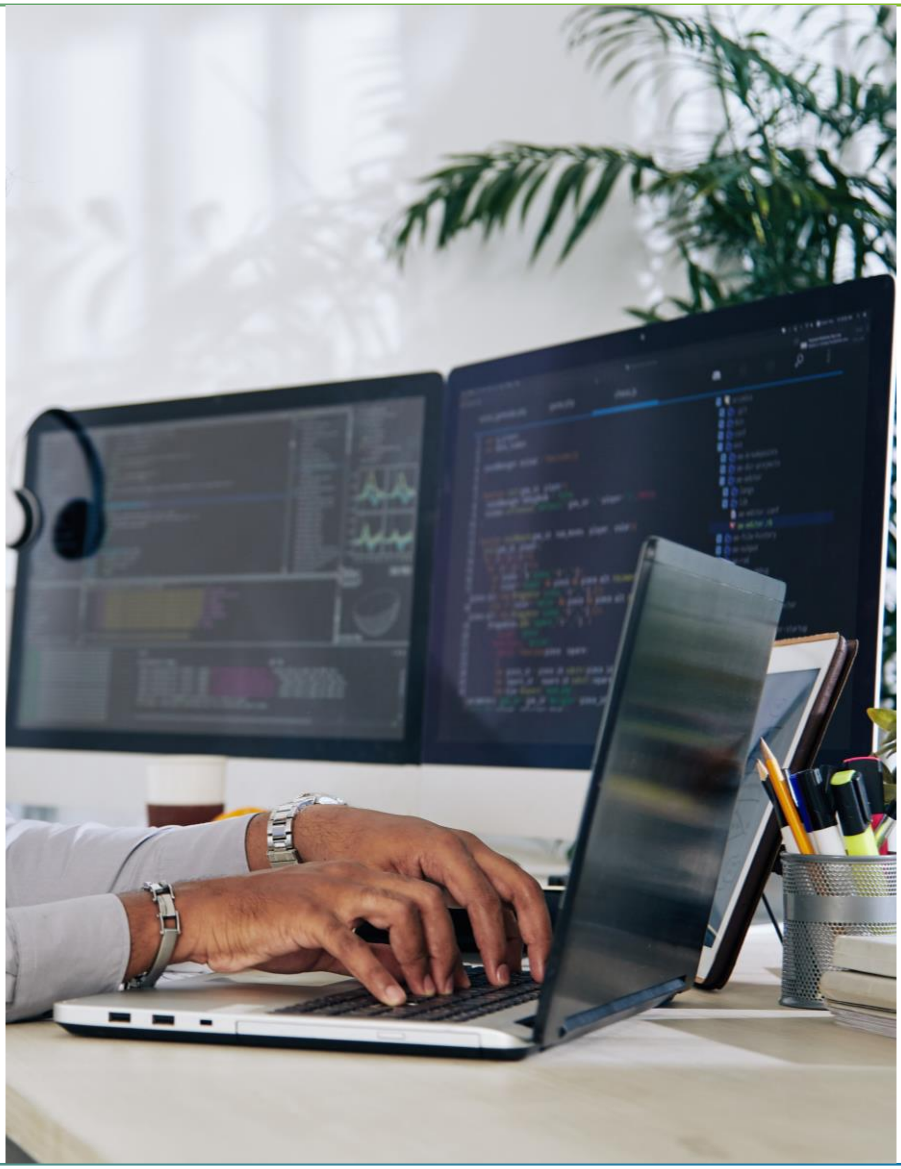
Example

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>


public class ExampleBean {

  public void cleanup() {
      // do some destruction work (like releasing pooled connections)
  }
}
```

The above example is same as the below snippet but does not couple the code to spring.

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements DisposableBean {

  public void destroy() {
      // do some destruction work (like releasing pooled connections)
  }
}
```

Autowiring in Spring



Autowiring in Spring

You can inject the object dependency implicity by enabling the autowiring feature of Spring Framework.

It internally uses setter or constructor injection.

Autowiring cannot be used to inject primitive and string values. It works with reference only.

Autowiring allows the spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.

Modes of Autowiring

no	No autowiring can be utilized in this default setting and you should use explicit bean reference for the wiring.
byName	• The name of the property is the basis of autowiring, meaning spring will use the property name to look for a bean that needs to be set. • It internally calls the setter method.
byType	• The type of the property is the basis of byName autowiring. This means spring will search for a bean to set to with the same type of the property. The framework throws an exception if there is more than one bean of that type. • One can use byType or constructor autowiring mode to wire arrays and other typed-collections. • It internally calls setter method.
constructor	• Autowiring is done based on the constructor arguments, means spring will look for beans with the same type as the constructor arguments. • Wire is first tried by spring through constructor, but if that doesn't work it will autowire by byType.

Autowiring in Spring—Pros and Cons

Pros/Cons	Reason
Pros	Code doesn't need to be written to inject the dependency explicitly, so less coding is required.
Cons	Can't be used for string values and primitive
Cons	• Dependencies can still be specified using <constructor-arg> and <property> settings which will always override autowiring • No control of programmer
Cons	Autowiring is less exact than explicit wiring and so, if possible, using explict wiring should be preferred

Demo

Example

- Let’s autowire the item1 bean by type into the store bean

```
@Bean(autowire = Autowire.BY_TYPE)
public class Store {

    private Item item;

    public setItem(Item item){
        this.item = item;
    }
}
```

- We can also inject beans using the @Autowired annotation for autowiring by type

```
public class Store {

    @Autowired
    private Item item
}
```

Example

- If there's more than one bean of the same type, we can use the @Qualifier annotation to reference a bean by name

```
public class Store {

    @Autowired
    @Qualifier("item1")
    private Item item;
}
```

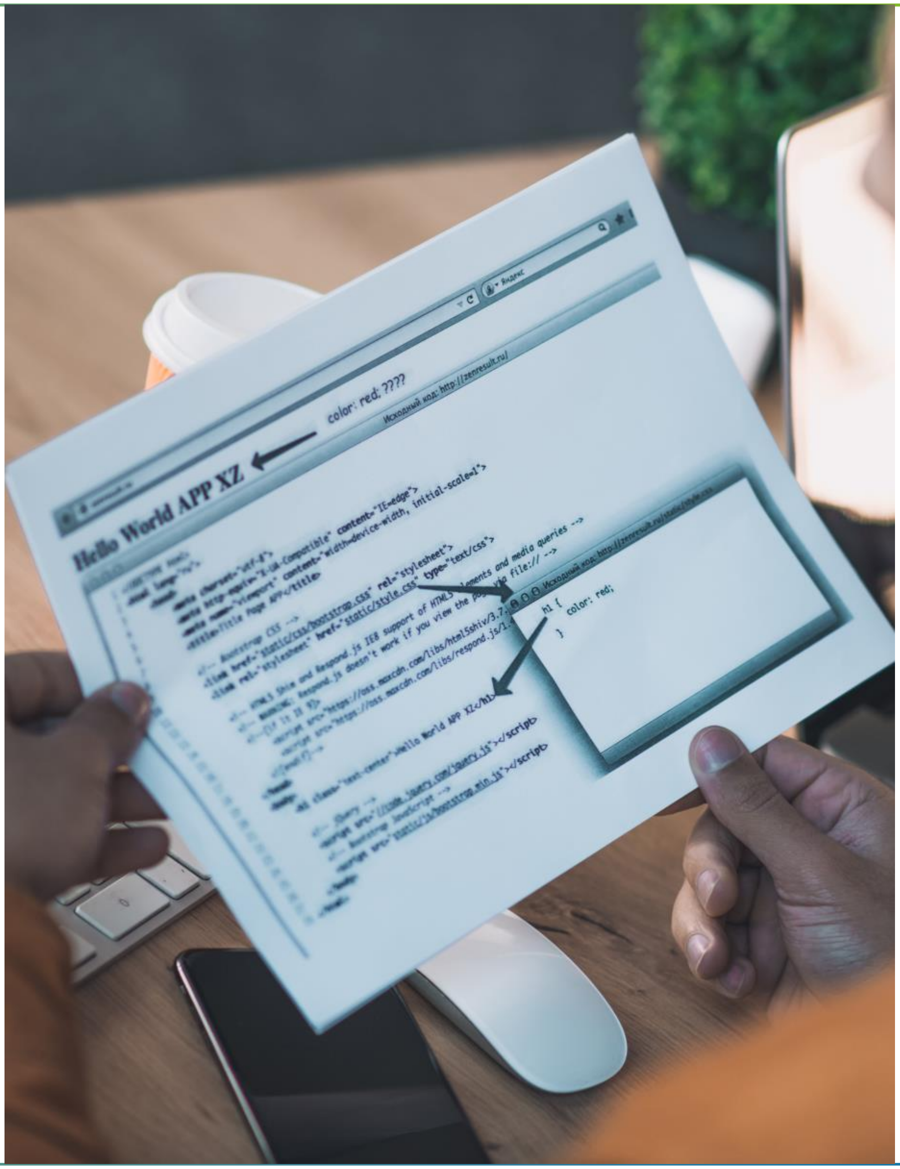
- Now, let's autowire beans by type through XML configuration

```
<bean id="store" class="org.baeldung.store.Store" autowire="byType"> </bean>
```

- Next, let's inject a bean named item into the item property of store bean by name through XML

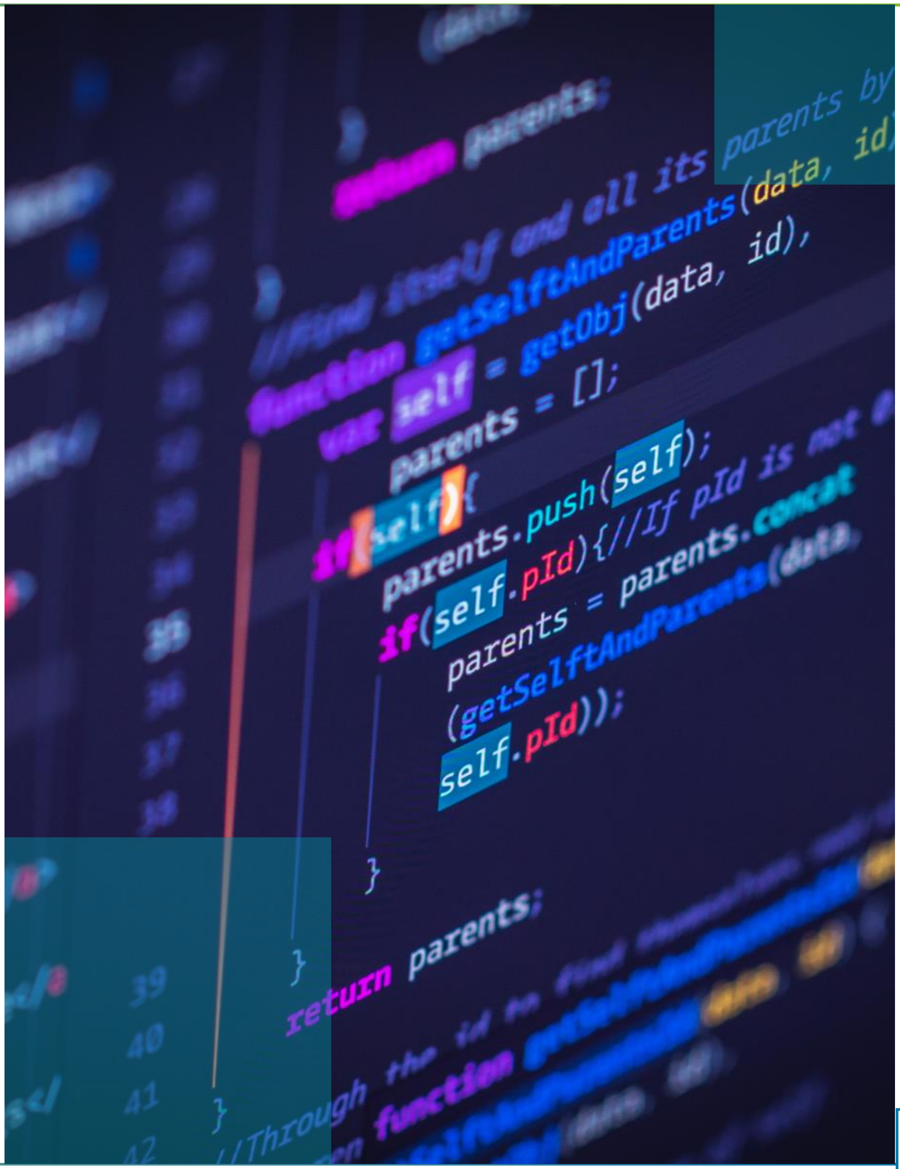
```
<bean id="item" class="org.baeldung.store.ItemImpl1" />

<bean id="store" class="org.baeldung.store.Store" autowire="byName"
</bean>
```

Working with Properties Files



Overview

- When working with properties, spring has always tried to be as transparent as possible.
- Two property sources are now contained in the default spring environment, namely the system properties and the JVM properties, with the system properties having precedence.
- It became possible, starting from spring 2.5, to configure the dependency injection using annotations.
- Properties can be registered in two ways, namely JavaAnnotations and XML.



Properties with Spring

Registering properties files using annotations

- It became possible, starting from spring 2.5, to use annotations to configure the dependency injection.
- The bean can move the configuration into the component class itself by using annotations on the method, relevant class, or field declaration instead of using XML to describe a bean wiring.
- XML injection is performed after annotation injection and it will override the former properties wired through all approaches.
- The spring container does not turn on annotation wiring by default. The spring configuration file will need to be enabled before annotation-based wiring can be used.
- The following configuration file should be considered in case it is desired to use any annotation in the spring application.
- You can start annotating once the <context:annotation-config/> is configured code to indicate that spring should automatically wire values into properties, methods, and constructors.

Example

```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-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>
   <!-- bean definitions go here -->

</beans>
```

Properties with Spring (Cont.)

Registering properties files using annotations

- Java-based configuration option enables to write most of your spring configuration without XML but with the help of few Java-based annotations.
- Indicating a class can be used by the spring IoC container as a source of bean definition after annotating a class with the @Configuration.
- An object will be returned that should have been registered as a bean in the spring application context when @Bean annotation tells Spring that a method annotated with @Bean.
- The following @Configuration class would be the simplest possible:

```
package com.example;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {
   @Bean
   public HelloWorld helloWorld(){
      return new HelloWorld();
   }
}
```

Example

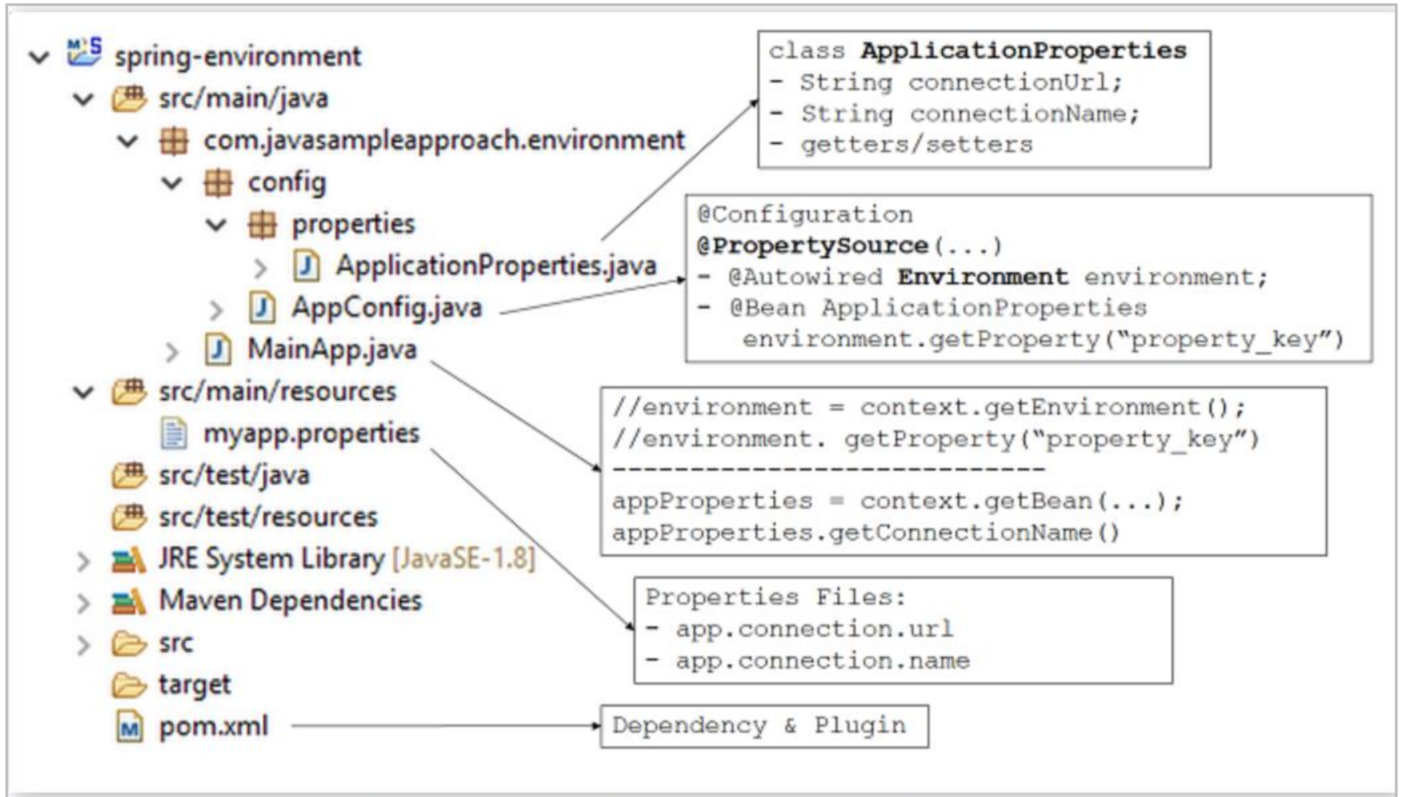
- The following XML configuration will be equivalent with the code above:

```
<beans>
   <bean id = "helloworld" class = "com.test" />
</beans>
```

- The actual bean will be created and returned with the method name annotated with @Bean works as bean ID.
- More than one @Bean can have a declaration in your configuration class. You can load and provide your defined configuration classes to the Spring container using AnnotationConfigApplicationContext as follows:

```
public static void main(String[] args) {
   ApplicationContext ctx = new AnnotationConfigApplicationContext
      (HelloWorldConfig.class);

   HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
   helloWorld.setMessage("Hello World!");
   helloWorld.getMessage();
}
```

Injecting Properties—Demo



```
class ApplicationProperties
- String connectionUrl;
- String connectionName;
- getters/setters
```

```
@Configuration
@PropertySource(...)
- @Autowired Environment environment;
- @Bean ApplicationProperties
    environment.getProperty("property_key")
```

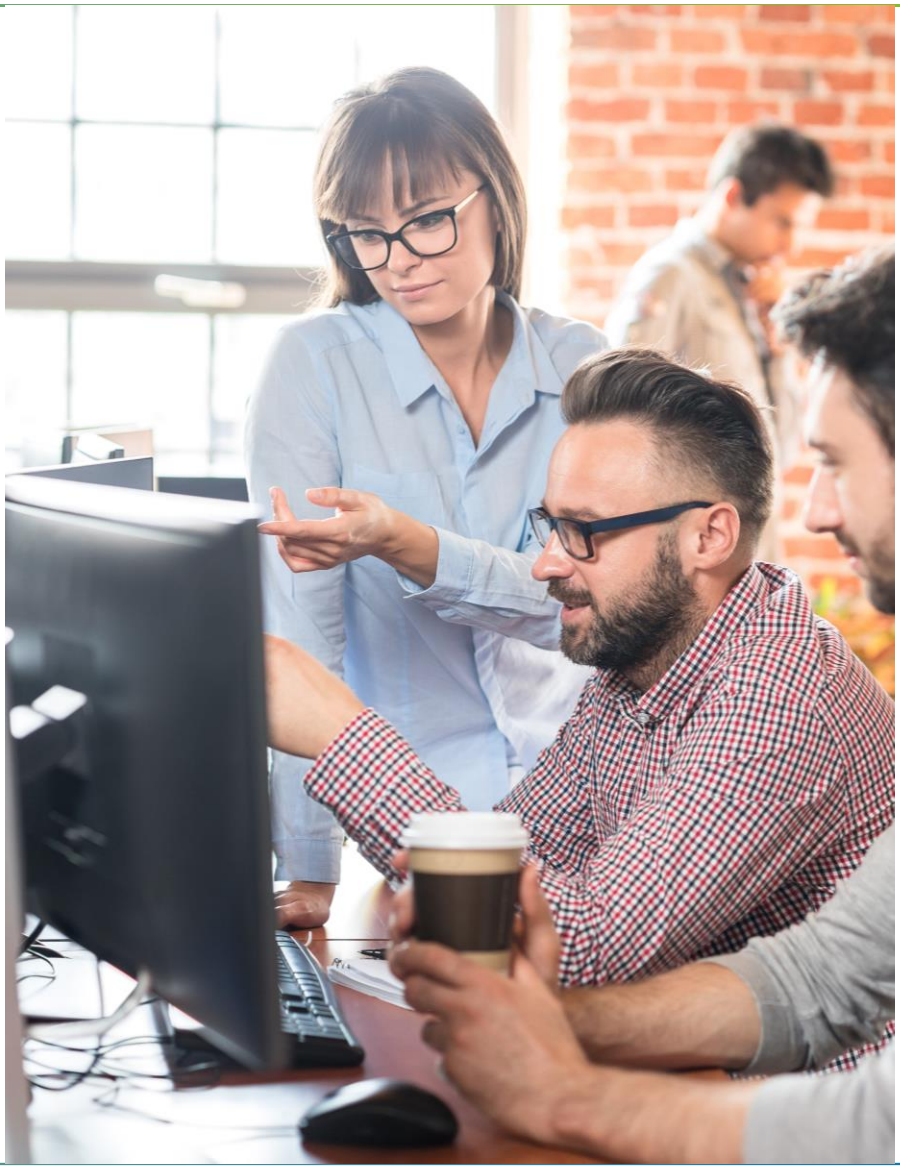
```
//environment = context.getEnvironment();
//environment. getProperty("property_key")
-----------------------------
appProperties = context.getBean(...);
appProperties.getConnectionName()
```

```
Properties Files:
- app.connection.url
- app.connection.name
```

```
Dependency & Plugin
```

In this example, we have 2 ways to access Properties from **Environment**:
– get **Environment** from **Application Context** wherever we wanna use it.
– use a seperate properties bean object to get **Environment** in the configuration class.

Spring Factory Method



Spring Factory Method

Factory Method Pattern

- Factory Method can be a useful technique for hiding complex creation logic within a single method call.
- While one can commonly create beans in Spring using constructor or field injection, one can also create spring beans using factory methods.
- Factory method: It represents the factory method that will be invoked to inject the bean.
- Factory bean: It represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

Factory Method—Use Case

Use Case

Write a program that:

- will help in understanding the spring factory method.
- will show the different ways in which static and non static method of the factory class can be used.
- will show the use of application context files in both scenarios.

Approach

Step 1: Create a POJO bean `class.Samplebean.Java`.

Step 2: Two factory classes with static and non-static factory methods namely `Staticmethodfactory.Java` and `nonstaticmethodfactory.Java`.

Step 3: Configure sample beans from `staticmethodfactory`.

Step 4: Configure sample beans from `nonstaticmethodfactory`.

Step 5: Configure application context file for each of the types with java config and with xml config.

Example

```java
package com.codesample.factory;
public class SampleBean {
  private String message;

  public SampleBean(String message){
    this.message = message;
  }

  public void hello() {
        System.out.println("Message = " + message);
    }

  public String getMessage() {
    return message;
  }
  public void setMessage(String message) {
    this.message = message;
  }
}
```

Static and Non-Static Ways

</>

StaticMethodFactory.java

```java
package com.codesample.factory;
public class StaticMethodFactory {
    public static SampleBean createBean(String message) {
        return new SampleBean(message);
    }
}
```

NonStaticMethodFactory.java

```java
package com.codesample.factory;
public class NonStaticMethodFactory {
    public SampleBean createBean(String message) {
        return new SampleBean(message);
    }
}
```

With Java Config

```java
package com.codesample.javaconfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.javasampleapproach.factory.NonStaticMethodFactory;
import com.javasampleapproach.factory.SampleBean;
import com.javasampleapproach.factory.StaticMethodFactory;

@Configuration
public class JavaConfig {

  @Bean(name = "bean1")
  public SampleBean createBeanFromStaticMethodFactory() {
    return StaticMethodFactory.createBean("This is a Bean created from
        StaticMethodClass!");
  }

  @Bean(name = "bean2")
  public SampleBean createBeanFromNonStaticMethodFactory() {
    NonStaticMethodFactory beanFactory = new NonStaticMethodFactory();
    return beanFactory.createBean("This is a Bean created from
        NonStaticMethodFactory!");
  }
}
```

Using XML Config / Java Config

Application Context file

```java
package com.javasampleapproach.javaconfig;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import com.javasampleapproach.factory.SampleBean;

public class SpringFactoryMethod {

  public static void main(String[] args){
    ApplicationContext ctx = new AnnotationConfigApplicationContext
    (JavaConfig.class);

    SampleBean bean1 = (SampleBean) ctx.getBean("bean1");
    SampleBean bean2 = (SampleBean) ctx.getBean("bean2");

    bean1.hello();
    bean2.hello();
  }
}
```

Use Spring Factory Method with XMLconfig

Case 1:

With Static Method Factory like `StaticMethodFactory` class

Use:

- Factory method is used to indicate the method name of the Factory class
- Constructor-arg is used to specify arguments of the factory method

```xml
<bean id="bean1"
      class="com.javasampleapproach.factory.StaticMethodFactory" factory-method
              ="createBean">
      <constructor-arg name="message" value="This is Bean created from
              StaticMethodClass!"/>
</bean>
```

Using XML Config / Java Config (Cont.)

Application Context file

Create a simple application context file

```java
package com.codesample.xmlconfig;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.javasampleapproach.factory.SampleBean;

public class SpringFactoryMethod {

  public static void main(String[] args){
    ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

    SampleBean bean1 = (SampleBean) ctx.getBean("bean1");
    SampleBean bean2 = (SampleBean) ctx.getBean("bean2");

    bean1.hello();
    bean2.hello();
  }
}
```

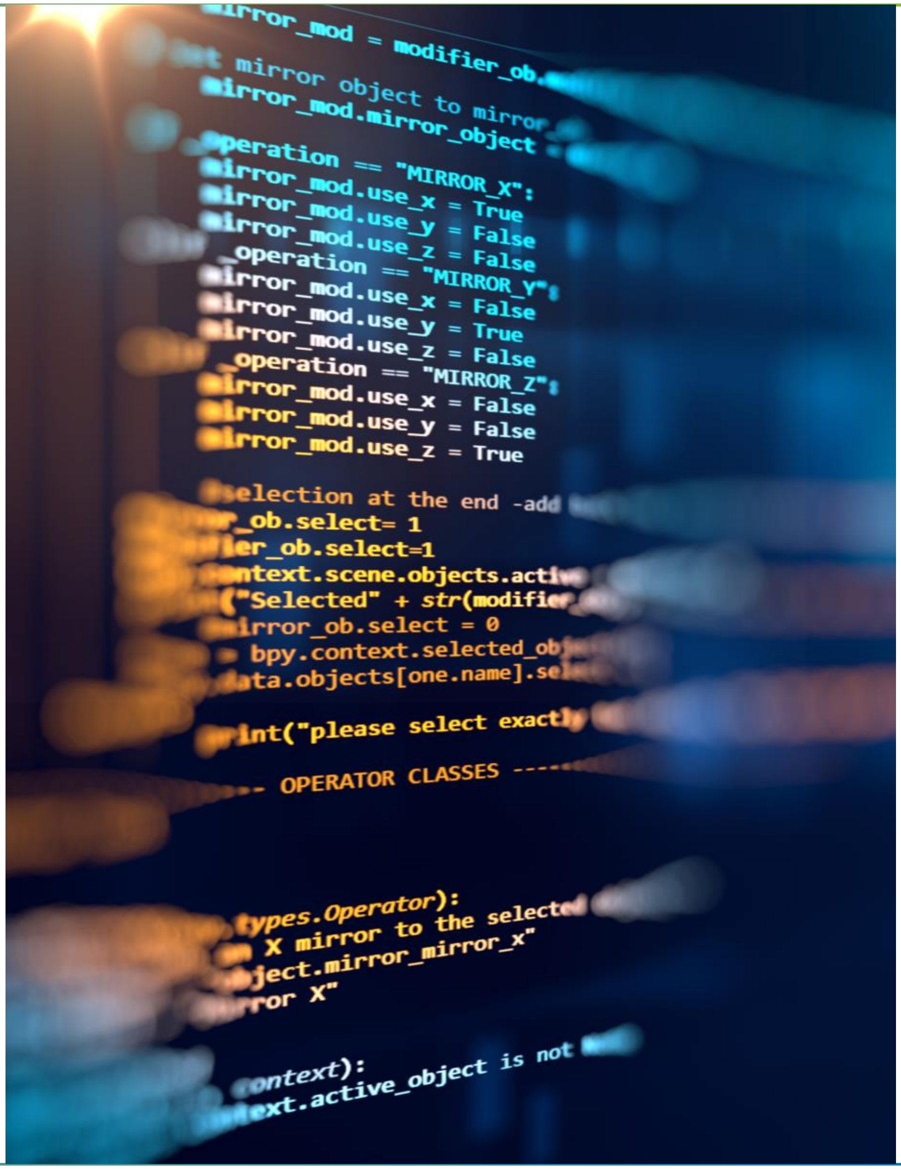
Use Spring Factory Method with XMLconfig

Case 2:

- With Non-Static Method Factory like NonStaticMethodFactory class
- Need to declare an additional factory bean: NonStaticMethodFactory, then use factory-bean to indicate it

```xml
<!-- Create Bean from NonStaticMethodFactory -->
<bean id="factoryBean"
      class="com.javasampleapproach.factory.NonStaticMethodFactory"/>

<bean id="bean2" class="com.javasampleapproach.factory.NonStaticMethodFactory"
    factory-method="createBean"  factory-bean="factoryBean">
    <constructor-arg name="message" value="This is Bean created from
            NonStaticMethodClass!"/>
</bean>
```

Spring AOP

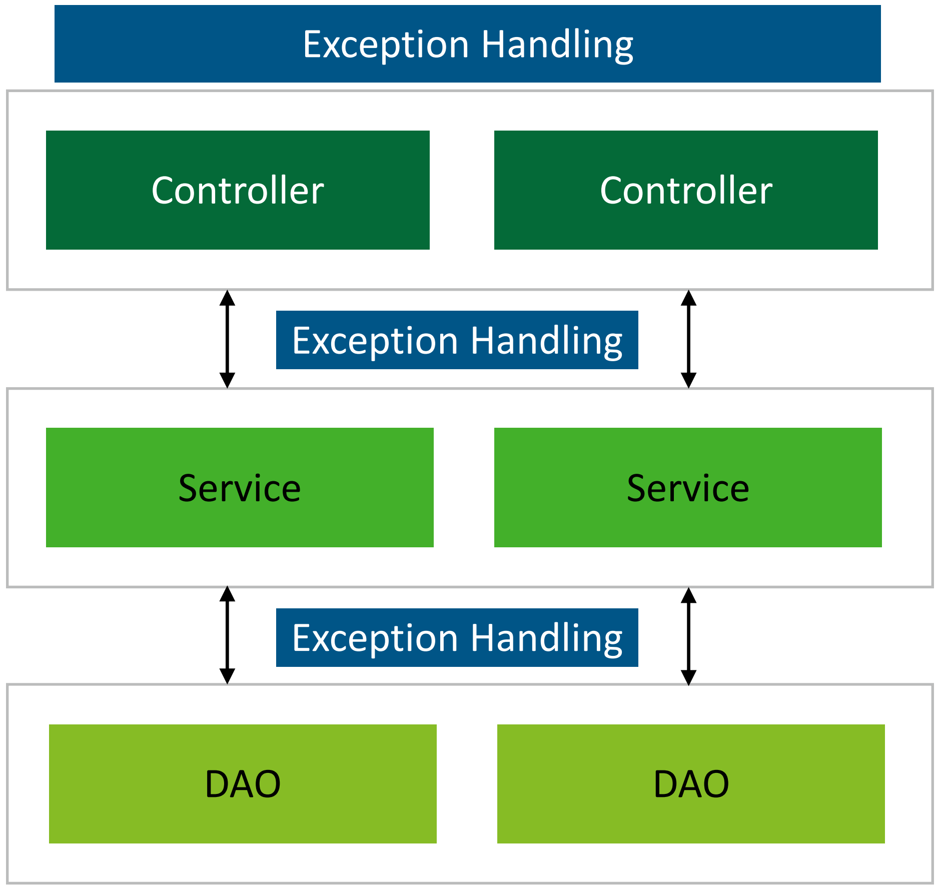


Spring AOP

Problem without Spring AOP:

- If you have a system that contains multiple classes and packages, such as tracing, transactions, and exception handling, we must implement them in every class and every method.
- This results in two problems:
  - Code tangling-each class and method contains tracing, transactions, and exception handling even business logic. It is often difficult to see what is happening in a method in tangled code.
  - Code scattering-aspects are not just implemented in one specific part of the system but are scattered throughout the code, such as transactions.
- These problems can be solved by using AOP. AOP places all transaction code into a transaction aspect. Then, another aspect is created to place all of the tracing code inside. Finally, exception handling is also put into a separate aspect.
- After using AOP, there will be a clean separation between the business logic and all additional aspects.
- AOP provides the pluggable way to dynamically add the additional concern before, after, or around the actual logic.

Exception Handling AOP Applied

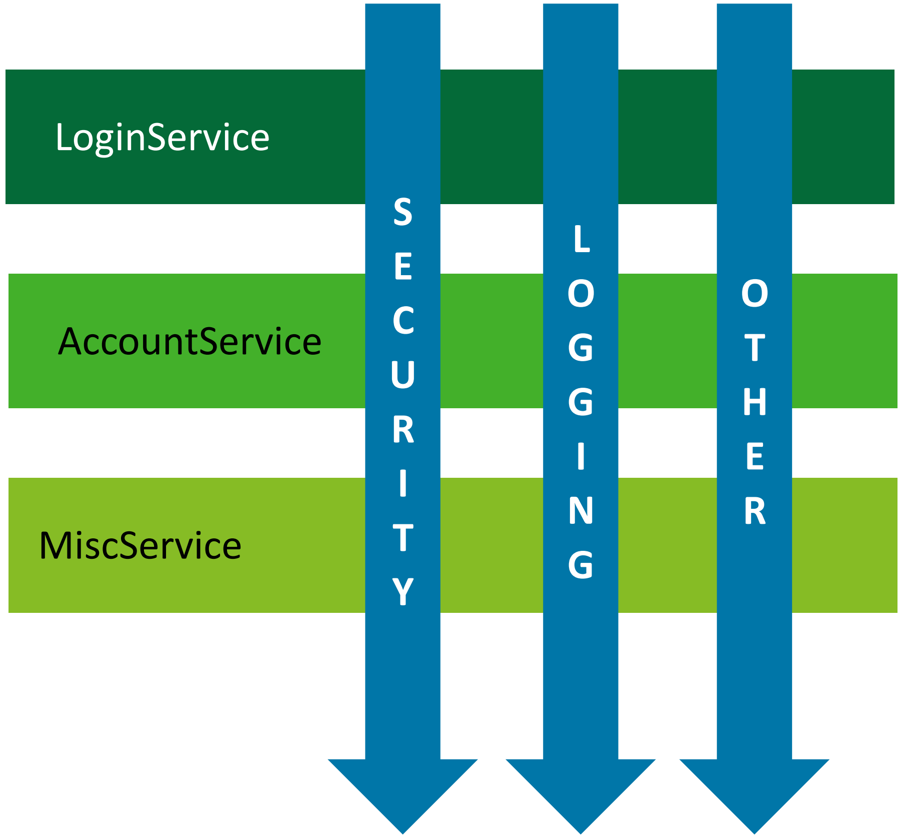


</>

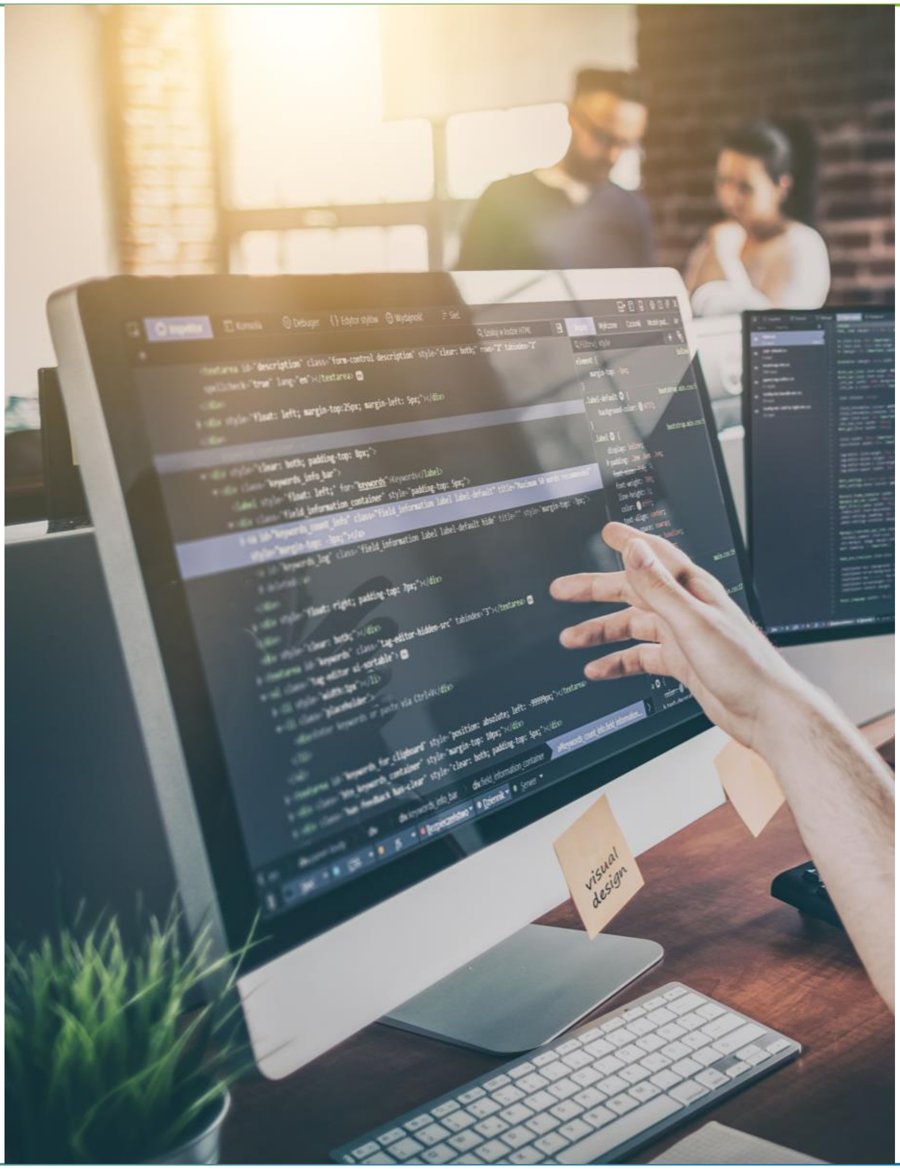
AOP Solution and Use Cases

AOP Solution

- Aspect oriented programming entails breaking down program logic into distinct parts called “concerns”. The functions that span multiple points of an application are called “cross-cutting concerns”. These cross-cutting concerns are used to increase modularity and are conceptually separate from the application's business logic.
- In OOP The key unit of modularity is the class, in comparison in AOP the unit of modularity is the aspect.
- It can be used in cross cutting layer functionality like Security, Transactional, Logging, Monitoring and Cache.
- To provide a more controlled and easy implementation the Spring Framework recommends you the usage of spring ASPECTJ AOP implementation. The spring AOP ASPECTJ implementation can be utilized in two ways i.e. by annotation and by xml configuration.



Spring—JDBC Template and ORM



Spring—JDBC Template—Introduction

Problems of JDBC API:

- Lots of boiler plate code is produced by JDBC, such as opening/closing a connection to a database, handling sql exceptions etc.
- One needs to perform exception handling code on the database logic.
- Repetition of all these codes from one to another database logic is a time-consuming task.
- One needs to handle transaction.

Spring JDBC to the rescue:

- Spring JDBC eliminates all the above-mentioned problems of JDBC API.
- It provides the methods to write the queries directly, so it saves a lot of work and time.
- Spring framework provides approaches for JDBC database access such as JdbcTemplate, NamedParameterJdbcTemplate, SimpleJdbcTemplate, SimpleJdbcInsert and SimpleJdbcCall.

1. JdbcTemplate
2. NamedParameterJdbcTemplate
3. SimpleJdbcTemplate
4. SimpleJdbcInsert
5. SimpleJdbcCall

JDBC Template Types

JDBCTemplate class

- In the spring JDBC support classes it is the central class.
- If you forget to close the connection when creating or closing of connection objects it will not cause any problems since it is taking care of creation and release of resources.
- By the assistance of exception classes defined in the org.Springframework.Dao package it handles the exception and the information exception message is provided.
- CRUD is a data-oriented and the standardized use of HTTP action verbs. HTTP has a few important verbs.
- Within a database, each of these operations maps directly to a series of commands. However, their relationship with a REST API is slightly more complex.

JDBC Template Types

Methods of JdbcTemplate class	Description
public int update(String query)	To create a new record the INSERT statement is performed.
public int update(String query,Object... args)	Using PreparedStatement it can update, insert, and delete records with given arguments is used to execute DDL query.
public void execute(String query)	It executes the query by using PreparedStatement callback.
public T execute(String sql, PreparedStatementCallback action)	ResultSetExtractor and RowMapper is used to fetch records.
public T query(String sql, ResultSetExtractor rse)	
public List query(String sql, RowMapper rm)	

JDBC Template Types (Cont.)

PreparedStatement in JdbcTemplate

- One can execute parameterized query using spring JDBCTemplate by the help of execute() method of JDBCTemplate class
- To use parameterized query, pass the instance of PreparedStatementCallback in the execute method
- Syntax of execute method to use parameterized query is:

```
public T execute(String sql,PreparedStatement
Callback<T>);
```

Method of PreparedStatementCallback interface. It has only one method "doInPreparedStatement".

```
public T doInPreparedStatement(PreparedStatement ps)
throws SQLException, DataAccessException
```

Example:

```
String query = "insert into employee
values(?,?,?)";
public Boolean
doInPreparedStatement(Prepared
Statement ps)
throws SQLException, DataAccessException
{
ps.setInt(1,Id);
ps.setString(2,Name);
ps.setFloat(3,Salary);
return ps.execute();
}
dao.saveEmployeeByPreparedStatement(new
Employee(111,"User",35000));
```

JDBC Template Types (Cont.)

ResultSetExtractor(Fetching records by Spring JdbcTemplate)

- A callback interface org.springframework.jdbc.core.ResultSetExtractor interface is used by JdbcTemplate's query methods and the actual work of extracting results from a ResultSet is implemented using this interface perform, exception handling does not need to be worried about.
- The JdbcTemplate calling will catch and handle SQLExceptions and this interface is mainly used within the JDBC framework itself.
- One can easily fetch the records from the database using query() method of JdbcTemplate class where one needs to pass the instance of ResultSetExtractor
- Syntax of query method using ResultSetExtracto is: `public T query(String sql,ResultSetExtractor<T> rse)`
- It defines only one method extractData that accepts ResultSet instance as a parameter: `public T extractData(ResultSet rs)throws SQLException,DataAccessException`

JDBC Template Types (Cont.)

- SQL–Select query to read students
- jdbcTemplateObject-StudentJDBCTemplate object to read student object from database
- ResultSetExtractor–ResultSetExtractor object to parse resultset object

Example:

```
public List<Student> listStudents() {
String SQL = "select * from Student";
List <Student> students = jdbcTemplateObject.query(SQL,
new ResultSetExtractor<List<Student>>(){
public List<Student> extractData(
ResultSet rs) throws SQLException, DataAccessException {
List<Student> list = new ArrayList<Student>();
while(rs.next()){
Student student = new Student();
student.setId(rs.getInt("id"));
student.setName(rs.getString("name"));
student.setAge(rs.getInt("age"));
student.setDescription(rs.getString("description"));
student.setImage(rs.getBytes("image"));
list.add(student);
}
return list;
}});
return students;
}
```

JDBC Template Types (Cont.)

1. RowMapper (Fetching records by Spring JdbcTemplate)

- The JdbcTemplate uses The org.springframework.jdbc.core.RowMapper<T> to map rows of a ResultSet on a per-row basis and the work of mapping each row to a result object is actually performed by implementation of this interface
- By the calling JdbcTemplate any SQLExceptions thrown will be caught and handled.
- Syntax of query method using RowMapper is-
  `public T query(String sql, RowMapper<T> rm)`
- It defines only one method mapRow that accepts ResultSet instance as a parameter-

  ```
  public T (ResultSet rs)throws SQLException,
  DataAccessException
  ```
- Advantage of RowMapper over ResultSetExtractor- RowMapper interface allows to map a row of the relations with the instance of user-defined class and it iterates the ResultSet internally and adds it into the collection, so, one doesn't need to write a lot of code to fetch the records as ResultSetExtractor

Example:

```
String SQL = "select * from Student";
List <Student> students=jdbcTemplateObject
.querySQL, new StudentMapper());
```

- SQL – Read query to read all student records
- jdbcTemplateObject – StudentJDBCTemplate object to read student records from database
- StudentMapper – StudentMapper object to map student records to student objects

JDBC Template Types (Cont.)

2. NamedParameterJdbcTemplate

The org.springframework.jdbc.core.NamedParameterJdbcTemplate class-

- Instead of the traditional "?" placeholder the use of named parameters are aloud in this template class with a basic set of JDBC operations.
- This templates allows the expansion of a list of values to the appropriate number of placeholders
- Interface Declaration:

```
public class NamedParameterJdbcTemplate
extends Object implements
NamedParameterJdbcOperations
```

- Is fast with comparison to SOAP because there is no strict specification like SOAP
- Is reusable and language neutral
- in–SqlParameterSource object to pass a parameter to update a query
- SqlLobValue–Object to represent an SQL BLOB/CLOB value parameter
- jdbcTemplateObject–NamedParameterJdbcTemplate object to update student object in the database

Example:

```
MapSqlParameterSource in = new
MapSqlParameterSource();
in.addValue("id", id);

in.addValue("description", new
SqlLobValue(description, new
DefaultLobHandler()), Types.CLOB);

String SQL = "update Student set
description = :description where id =
:id"; NamedParameterJdbcTemplate
jdbcTemplateObject = new
NamedParameterJdbcTemplate(dataSource);
jdbcTemplateObject.update(SQL, in);
```

JDBC Template Types (Cont.)

3. SimpleJdbcTemplate—Update

- Spring JDBC supports the Java 5+ feature var-args (variable argument) and autoboxing by the help of SimpleJdbcTemplate class
- SimpleJdbcTemplate class wraps the JdbcTemplate class and provides the update method where one can pass arbitrary number of arguments
- Syntax of update method of SimpleJdbcTemplate class:
  `int update(String sql,Object... parameters)`
- The parameter values should be passed in the update method in the order they are defined in the parameterized query
- SQL-Update query to update student records
- jdbcTemplateObject−StudentJDBCTemplate object to read student records the from database
- StudentMapper−SudentMapper object to map student records to student objects
- sqlUpdate−SqlUpdate object to update student records

Example:

```
String SQL = "update Student set age = ?
where id = ?";
SqlUpdate sqlUpdate = new
SqlUpdate(dataSource,SQL);
sqlUpdate.declareParameter(new
SqlParameter("age", Types.INTEGER));
sqlUpdate.declareParameter(new
SqlParameter("id", Types.INTEGER));
sqlUpdate.compile();
sqlUpdate.update(age.intValue(),id.intValue
());
```

JDBC Template Types (Cont.)

4. SimpleJdbcInsert

- The org.springframework.jdbc.core.SimpleJdbcInsert class is a multi-threaded, reusable object providing easy insert capabilities for a table
- It provides meta data processing to simplify the code needed to construct a basic insert statement
- The actual insert is being handled using Spring's JdbcTemplate
- Class Declaration :

```
public class SimpleJdbcInsert
extends AbstractJdbcInsert
implements SimpleJdbcInsertOperations
```

- jdbcInsert−SimpleJdbcInsert object to insert record in student table
- jdbcTemplateObject−StudentJDBCTemplate object to read student object in database

Example:

```
jdbcInsert = new
SimpleJdbcInsert(dataSource).withTable
Name("Student");
Map<String,Object> parameters = new
HashMap<String,Object>();
parameters.put("name", name);
parameters.put("age", age);
jdbcInsert.execute(parameters);
```

JDBC Template Types (Cont.)

5. SimpleJdbcCall

- Representing a call to a stored function or a stored procedureThe org.springframework.jdbc.core.SimpleJdbcCall class is a multi-threaded, reusable object.
- Provides meta data processing to simplify the code needed to access basic stored procedures/functions
- Passing the name of the procedure/function and a map containing the parameters when a call is executed
- The names of the supplied parameters will then be matched up with in and out parameters declared when the stored procedure was created
- Class Declaration:

```
public class SimpleJdbcCall
extends AbstractJdbcCall
implements SimpleJdbcCallOperations
```

- jdbcCall–SimpleJdbcCall object to represent a stored procedure
- in–SqlParameterSource object to pass a parameter to a stored procedure

Example:

```
SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withProcedureName("
getRecord"); SqlParameterSource in = new
MapSqlParameterSource().addValue("in_id", id);

Map<String, Object> out =
jdbcCall.execute(in);

Student student = new Student();

student.setId(id);

student.setName((String) out.get("out_name"));

student.setAge((Integer) out.get("out_age"));
```

- student–Student object
- out–Map object to represent output of stored procedure call result

Spring: ORM

Advantages of ORM frameworks with Spring

Less Coding is Required
The help of Spring Framework removes the need to write extra codes before and after the actual database logic. For example, getting and closing the connection, starting transaction, committing transaction, etc.

Easy to Test
It is easy to test the application using the Spring's IoC approach.

Better Exception Handling
Its own API for exception handling with ORM framework is provided by Spring framework.

Integrated Transaction Management
An AOP stylemethod interceptor or and explicit template wrapper class can be used by the help of Spring Framework to wrap our mapping code.

Spring—JDBC Template and ORM

- To integrate JPA with spring application, Spring Data JPA API provides JpaTemplate class.
- Hibernate, Ibatis, OpenJPA etc. provide the implementation of JPA specifications.
- The advantages include not writing the before and after code for persisting, updating, deleting, or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, commiting EntityTransaction instance, and closing EntityManager.

Example of Spring and JPA Integration	
Step 1	create Account.java file
Step 2	create Account.xml file
Step 3	create AccountDao.java file
Step 4	create persistence.xml file
Step 5	create applicationContext.xml file
Step 6	create AccountsClient.java file

Spring—JDBC Template and ORM (Cont.)

- The Spring framework provides HibernateTemplate class, so one need not follow so many steps like create Configuration, BuildSessionFactory, Session, beginning, committing transaction etc.
- This saves lot of code.
- If one integrates the hibernate application with spring, it is not needed to create the hibernate.cfg.xml file. One can provide all the information in the applicationContext.xml file.

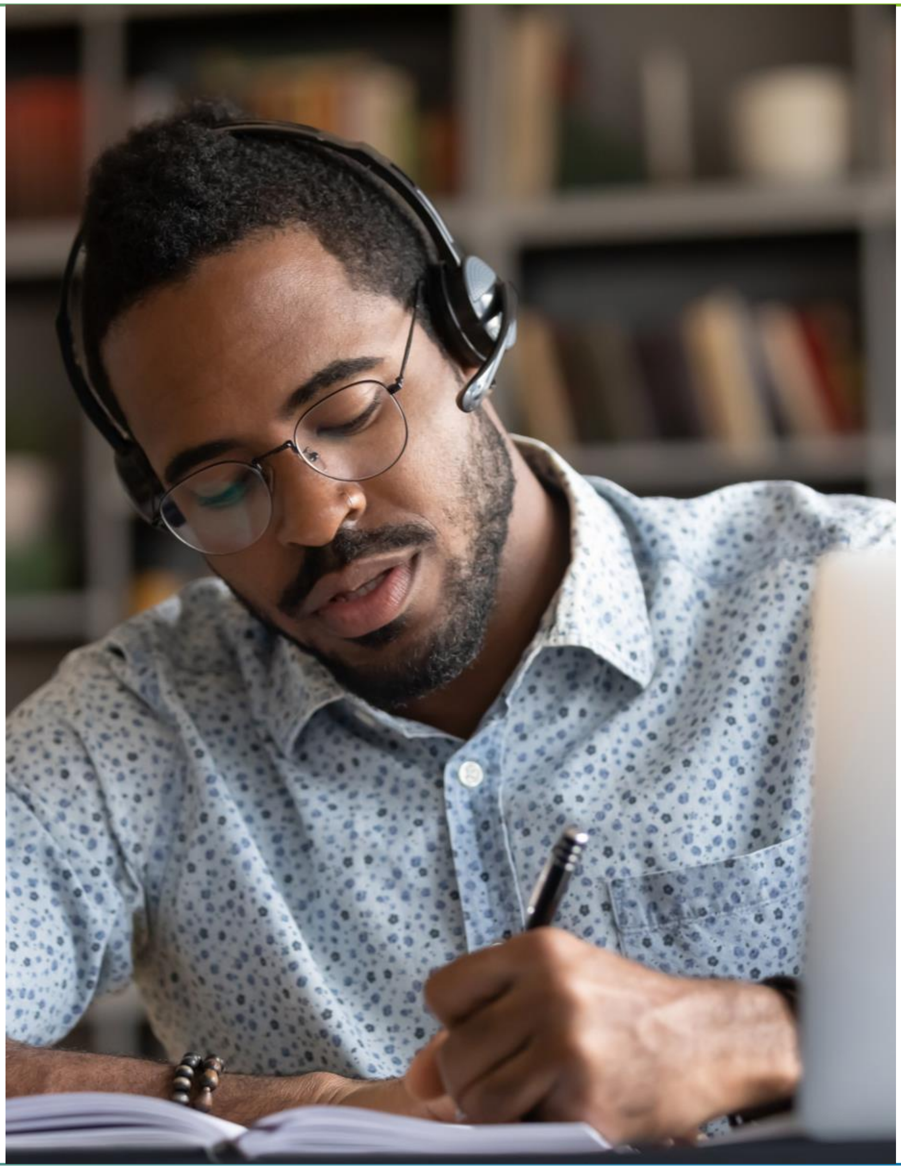
Steps for Hibernate and Spring Integration	
Step 1	• Create table in the database – Optional
Step 2	• Create applicationContext.xml – It contains information of DataSource, SessionFactory and more
Step 3	• Create Employee.java file – It is the persistent class
Step 4	• Create employee.hbm.xml file – It is the mapping file
Step 5	• Create EmployeeDao.java file – It is the dao class that uses HibernateTemplate
Step 6	• Create InsertTest.java file – It calls methods of EmployeeDao class

Summary

Here are the key learning points of the module.

- Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. The latest version of Spring is Spring 5.2.
- The Spring Framework comprises of many modules such as core, beans, context, expression language, AOP, aspects, instrumentation, JDBC, ORM, OXM, JMS, transaction, web, servlet, struts, etc.
- Spring is an open-source framework created to address the complexity of an enterprise application development.
- Spring MVC is a java framework to build web applications.
- The core of the spring Framework is the spring container. The container will create and manage the life cycle of the objects called spring Beans, wire them together, configure them, and manage them from creation till destruction using DI.
- Spring is modular, allowing one to pick and choose which modules are applicable to them, without having to bring in the rest.

Hands-On Labs



Hands-On Activity 1

Activity Details:	
Problem Statement	Create a Spring JdbcTemplate example program that will fetch data from a Database and display it to user.

Hands-On Activity 2

Activity Details:	
Problem Statement	Declare a Student class having rollNo,name, array of marks and address(instance of Address) as data member having default constructor. configure a student bean in students.xml file in class path having id as 'stu'

Thank You