Spring

Index

Spring AOP

SPRING DAO

Spring MVC

Spring Security

Spring

Spring is one of the most popular enterprise application frameworks in the Java landscape today. It is so popular that it's even considered the de-facto standard for building large scale applications in Java platform today.

The first version of Spring was written by **Rod Johnson**, who released the framework with the publication of his book Expert One-on-One J2EE Design and Development in October 2002. The framework was first released under the Apache 2.0 license in June 2003. The first milestone release, 1.0, was released in March 2004 with further milestone releases in September 2004 and March 2005.

Spring 2.0 was released in October 2006, Spring 2.5 in November 2007, Spring 3.0 in December 2009, Spring 3.1 in December 2011, and Spring 3.2.5 in November 2013.

Spring Framework 4.0 was released in December 2013. Notable improvements in Spring 4.0 included support for Java SE (Standard Edition) 8, Groovy 2, some aspects of Java EE 7, and Web Socket.

Spring Framework 4.2.0 was released on 31 July 2015 and was immediately upgraded to version 4.2.1, which was released on 01 Sept 2015. It is "compatible with Java 6, 7 and 8, with a focus on core refinements and modern web capabilities".

Spring Framework 4.3 has been released on 10 June 2016 and will be supported until 2020. It "will be the final generation within the general Spring 4 system requirements (Java 6+, Servlet 2.5+).

Spring 5 is announced to be built upon Reactive Streams Compatible Reactor Core.

Before jumping into Spring Framework first let's understand why people are choosing Spring framework to develop enterprise applications and what is specialty of the Spring Framework and why Spring having lot of demand in the industry.

Developing software applications is hard enough even with good tools and technologies. Implementing applications using platforms which promise everything but turn out to be heavy-weight, hard to control and not very efficient during the development cycle makes it even harder. To make application development, testing and maintenance easy we will develop applications using Layered Architecture pattern.

Layered Architecture

The most common architecture pattern is the layered architecture pattern and it is also known as the n-tier architecture pattern. This pattern is the de facto standard for most Java EE applications and therefore is widely known by most architects, designers, and developers. The layered architecture pattern closely matches the traditional IT communication and organizational structures found in most companies, making it a natural choice for most business application development efforts.

Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database. In some cases, the business layer and persistence layer are combined into a single business layer, particularly when the persistence logic (e.g., SQL or HSQL) is embedded within the business layer components. Thus, smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more layers.

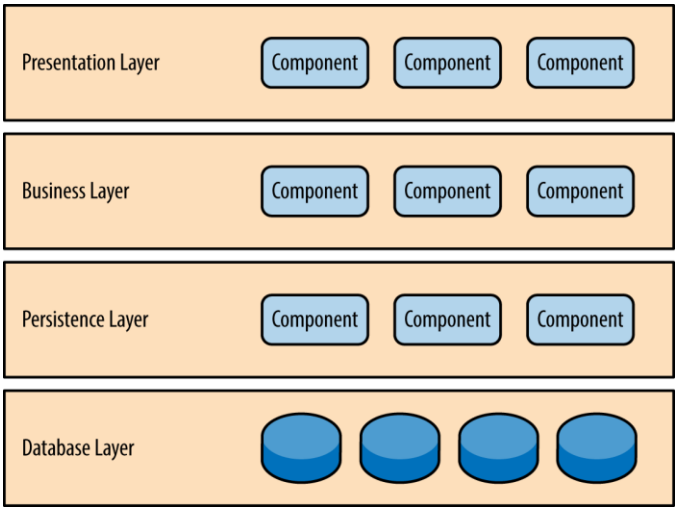
Each layer of the layered architecture pattern has a specific role and responsibility within the application.

Presentation Layer: A presentation layer would be responsible for handling all user interface and browser communication logic.

Spring

Business Layer: A business layer would be responsible for executing specific business rules associated with the request.

Persistence Layer: Persistence layers would be responsible for Database communication



Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request.

For example, the presentation layer doesn't need to know or worry about how to get customer data; it only needs to display that information on a screen in particular format.

Similarly, the business layer doesn't need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from; it only needs to get the data from the persistence layer, perform business logic against the data (e.g., calculate values or aggregate data), and pass that information up to the presentation layer.

One of the powerful features of the layered architecture pattern is the separation of concerns among components. Components within a specific layer deal only with logic that pertains to that layer. For example, components in the presentation layer deal only with presentation logic, whereas components residing in the business layer deal only with business logic. This type of component classification makes it easy to build effective roles and responsibility models into your architecture, and also makes it easy to develop, test, govern, and maintain applications using this architecture pattern due to well-defined component interfaces and limited component scope.

Using J2SE and J2EE we can develop applications using layered architecture pattern but these J2SE and J2EE are APIs. APIs are always partial and they contain so many interfaces. If we want to work with these APIs we should know all interfaces and relationships among those interfaces.

For example,

- To get Data from database using JDBC we should know ResultSet
- To get ResultSet we should know Statement or PreparedStatement interfaces
- To Create Statement or PreparedStatement we should know Connection
- To get Connection we should know DriverManager.

But in the other hand Spring is a framework which provides predefined Template classes to avoid boilerplate logic.

Spring

What is a Framework?

Framework is a semi developed software which provides some common logics which are required for several applications developments.

We can't install frameworks (These are Un-installable software's).

Frameworks will be released into market in the form of jar files. Jar files contains set of .class files (those class files contain several methods to provide common logics)

To develop a project using a framework then we have to add framework jars to project build path(then our project will contains commons logics which are provided by framework)

Types of frameworks

In the world of Java, we can classify the frameworks in below 3 categories

ORM Frameworks: These frameworks are used to develop persistence layer in the application.

Ex: Hibernate, JPA, IBatis, MyBatis and Toplink etc...

Web Frameworks: These frameworks are used to develop web components (Controllers, Filters, Listeners & Interceptors etc..) in web application.

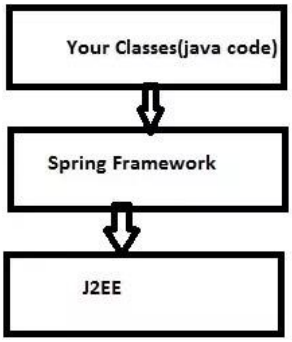
Ex: Struts 1.x Struts 2.x and JSF etc...

Application Frameworks: These frameworks will provide all components support to complete one application. They will provide web components and persistence components as well.

Ex: Spring

As discussed above using only Hibernate we can't complete one application because it helps only in Persistence layer. Similarly using only Struts framework, we can't complete one application because Struts provides support for only web components development. But using Spring we can complete one application development hence Spring is called as Application Development framework.

As spring is providing everything to complete one Application can we call Spring is replacement for J2EE?. We can't call Spring is replacement for J2EE because Spring internally uses J2EE API only (i.e. Spring is providing abstraction layer for J2EE API). Our Application classes will talk to Spring framework and internally Spring framework will talk to J2EE.



Spring

Java EE is the abstract specifications for developing enterprise applications. These specifications are implemented by the application server vendors like WebLogic, WebSphere etc.

For example, JSP and Servlet are speciation for developing web applications. Obviously, web applications deployed to the server. So, server vendors have to implement the specification.

Whereas spring follows some of the specifications from Java EE and override with better options. It is a standalone framework built on top of the Java EE technology.

Spring Introduction

Spring framework provides a light-weight solution for building enterprise-ready applications, while still supporting the possibility of using declarative transaction management, remote access to your logic using RMI or Webservices, mailing facilities and various options in persisting your data to a database. Spring provides an MVC framework, transparent ways of integrating AOP into your software and a well-structured exception hierarchy including automatic mapping from proprietary exception hierarchies.

Spring could potentially be a one-stop-shop for all your enterprise applications, however, spring is modular, allowing you to use parts of it, without having to bring in the rest. You can use the bean container, with Struts on top, but you could also choose to just use the Hibernate integration or the JDBC abstraction layer. Spring is non-intrusive, meaning dependencies on the framework are generally none or absolutely minimal, depending on the area of use.

Spring is a light weight and open source framework created by Rod Johnson in 2002. Spring is a complete and a modular framework, it means spring framework can be used for all layer implementations for a real time application or spring can be used for the development of particular support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

Spring framework is said to be a non-invasive means it doesn't force a programmer to extend or implement their class from any predefined class or interface given by Spring API, in struts we used to extend Action Class right that's why struts is said to be invasive.

- In case of struts framework, it will force the programmer that, the programmer class must extend from the base class provided by struts API
- Spring is light weight framework because of its POJO model
- Spring Framework made J2EE applications development little easier, by introducing POJO model

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1. Predefined Templates
2. Loose Coupling
3. Easy to test
4. Lightweight
5. Fast Development
6. Powerful abstraction

Spring

Predefined Templates: Spring system gives layouts to JDBC, Hibernate, JPA and so forth advances. So there is no compelling reason to compose a lot of code. It shrouds the essential strides of these advances.

Loose Coupling: The Spring applications are approximately coupled due to reliance infusion.

Easy to test: The Dependency Injection makes less demanding to test the application. The EJB or Struts application oblige server to run the application however spring system doesn't oblige server.

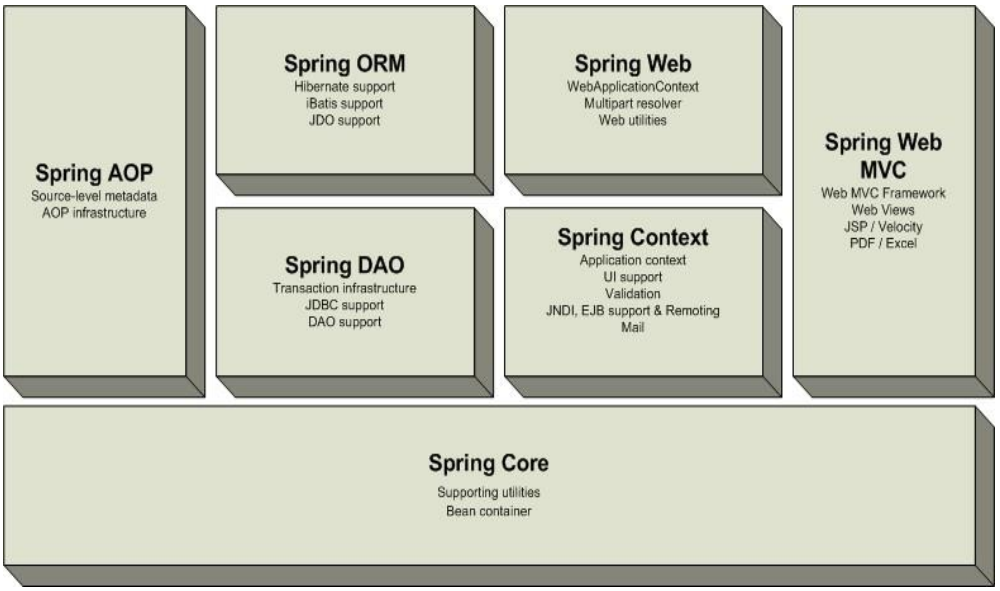
Lightweight: Spring system is lightweight as a result of its POJO usage. The Spring Framework doesn't compel the developer to acquire any class or actualize any interface. That is the reason it is said non-obtrusive.

Fast Development: The Dependency Injection highlight of Spring Framework and it backing to different systems makes the simple advancement of JavaEE application.

Powerful reflection: It gives intense deliberation to JavaEE determinations, for example, JMS, JDBC, JPA and JTA.

Spring Framework Architecture

Spring contains a lot of functionality and features, which are well-organized in seven modules shown in the diagram below.



The Core package is the most fundamental part of the framework and provides the Dependency Injection features allowing you to manage bean container functionality. The basic concept here is the BeanFactory, which provides a factory pattern removing the need for programmatic singletons and allowing you to decouple the configuration and specification of dependencies from your actual program logic.

On top of the Core package sits the Context package, providing a way to access beans in a framework-style manner, somewhat resembling a JNDI-registry. The context package inherits its features from the beans package and adds support for text messaging using e.g. resource bundles, event-propagation, resource-loading and transparent creation of contexts by, for example, a servlet container.

The DAO package provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes. Also, the JDBC package provides a way to do programmatic as well as declarative transaction management, not only for classes implementing special interfaces, but for all your POJOs (plain old java objects).

Spring

The ORM package provides integration layers for popular object-relational mapping APIs, including JDO, Hibernate and iBatis. Using the ORM package you can use all those O/R-mappers in combination with all the other features Springoffers, like simple declarative transaction management mentioned before.

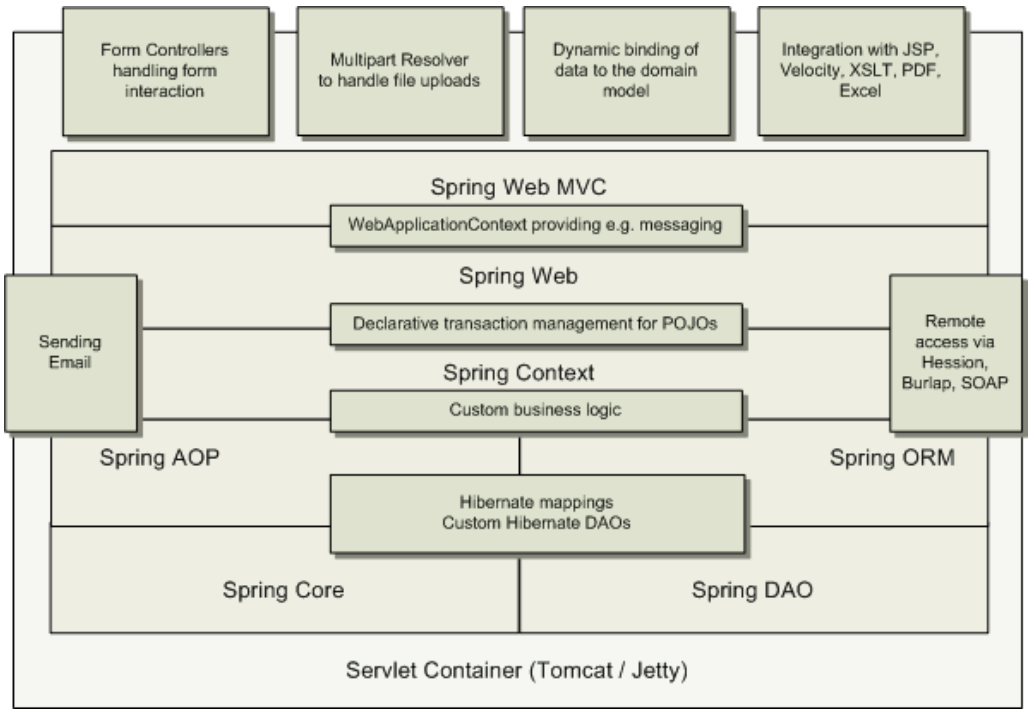
Spring's AOP package provides an AOP Alliance compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated. Using source-level metadata functionality you can incorporate all kinds of behavioral information into your code, a little like .NET attributes.

Spring's Web package provides basic web-oriented integration features, such as multipart functionality, initialization of contexts using servlet listeners and a web-oriented application context. When using Spring together with WebWork or Struts, this is the package to integrate with.

Spring's Web MVC package provides a Model-View-Controller implementation for web-applications. Spring's MVC implementation is not just any implementation, it provides a clean separation between domain model code and web forms and allows you to use all the other features of the Spring Framework like validation.

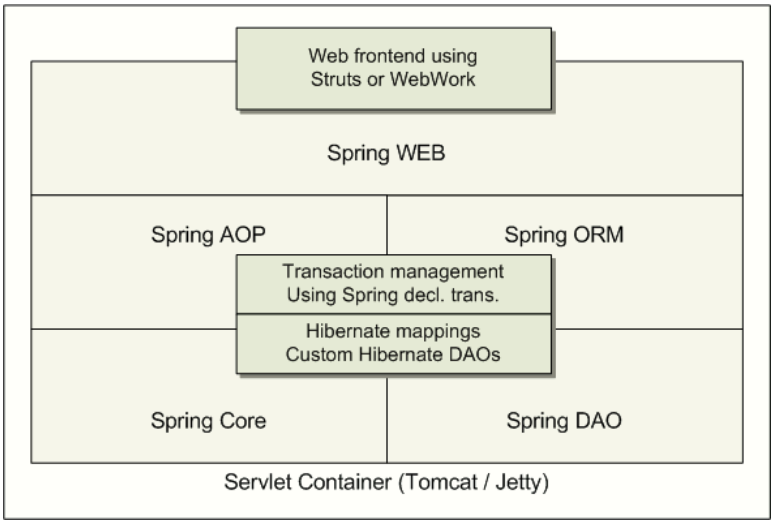
Usage scenarios

With the building blocks described above you can use spring in all sorts of scenarios, from applets up to fully-fledged enterprise applications using spring's transaction management functionality and Web framework.

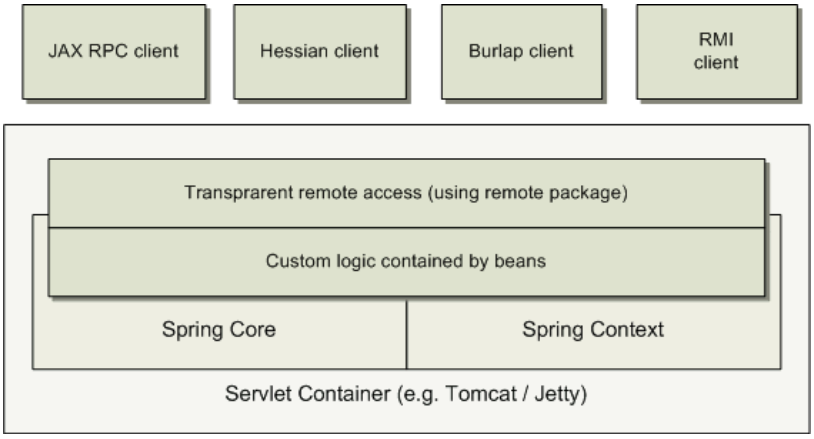


A typical web application using most of spring's features. Using TransactionProxyFactoryBeans the web application is fully transactional, just as it would be when using container managed transaction as provided by Enterprise JavaBeans. All your custom business logic can be implemented using simple POJOs, managed by spring's Dependency Injection container. Additional services such as sending email and validation, independent of the web layer enable you to choose where to execute validation rules. Spring's ORM support is integrated with Hibernate, JDO and iBatis. Using for example HibernateDaoSupport, you can re-use your existing Hibernate mappings. Form controllers seamlessly integrate the web-layer with the domain model, removing the need for ActionForms or other classes that transform HTTP parameters to values for your domain model.

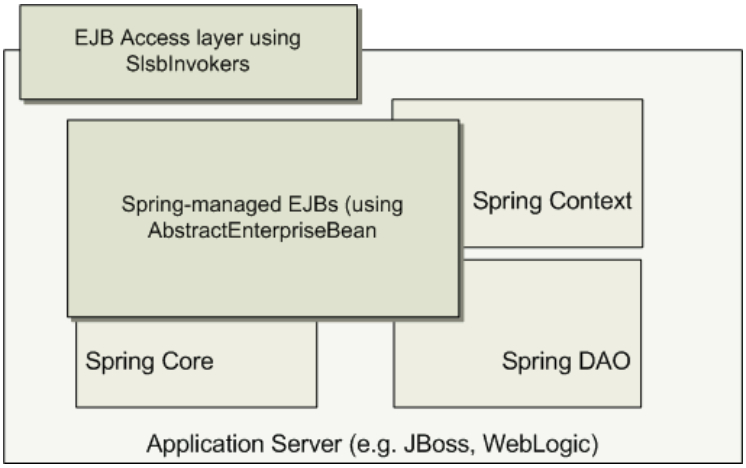
Spring



Sometimes the current circumstances do not allow you to completely switch to a different framework. Spring does not force you to use everything within it; it's not an all-or-nothing solution. Existing frontends using Web Work, Struts, Tapestry, or other UI frameworks can be integrated perfectly well with a Spring-based middle-tier, allowing you to use the transaction features that spring offers. The only thing you need to do is wire up your business logic using an ApplicationContext and integrate your Web UI layer using a WebApplicationContext.



When you need to access existing code via Webservices, you can use spring's Hessian-, Burlap Rmi- or JAXRpc ProxyFactory classes. Enabling remote access to existing application is all of a sudden not that hard anymore.

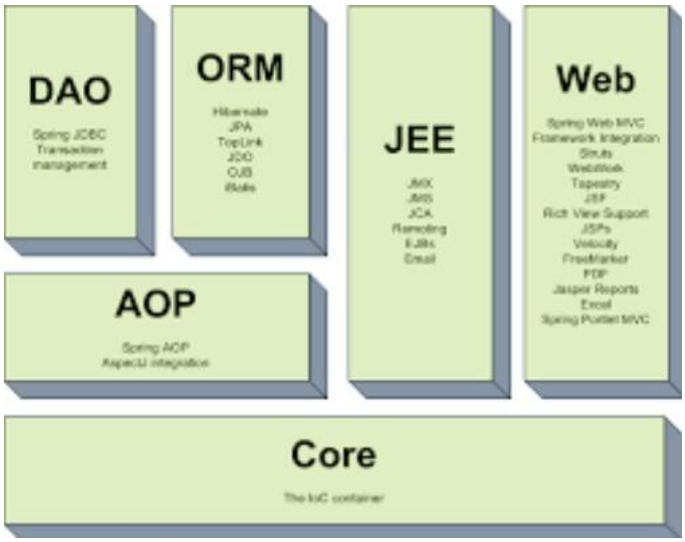


Spring also provides an access layer and abstraction layer for Enterprise JavaBeans, enabling you to reuse your existing POJOs and wrap them in Stateless Session Beans, for use in scalable failsafe web applications that might need declarative security.

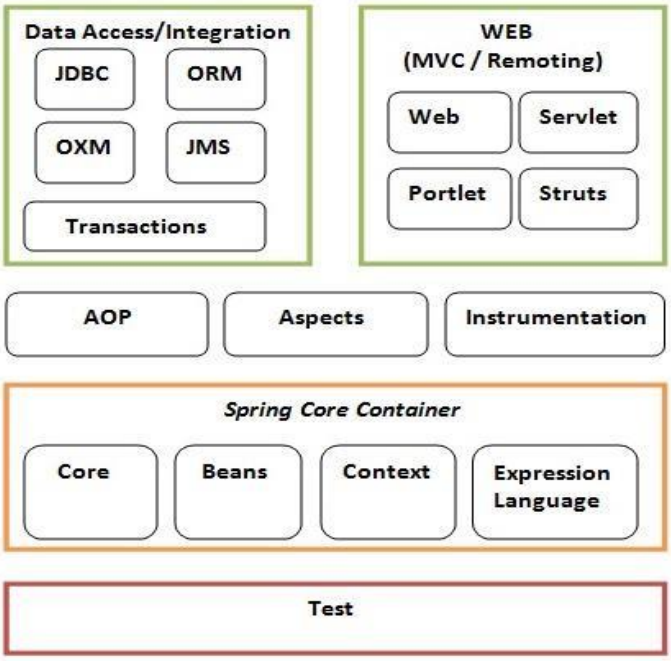
Spring

As discussed above, in Spring 1.x version we have total 7 modules.

Coming to Spring 2.x version we have total 6 modules (Spring Web and Web MVC combined as single module)



In Spring 3.x version main modules got divided into sub modules like below



Test

- This layer gives backing of testing JUnit and TestNG.
- Spring Core Container
- The Spring Core holder contains center, beans, setting and expression dialect (EL) modules.
- These modules give IOC and Dependency Injection highlights.

<u>Core Container: Core container includes below four modules-</u>

- **Spring Core-** As its name suggests, this module is the core of the spring framework and provides implementation of several features including Inversion of Control (IOC). IoC is also known as Dependency Injection and it allows

objects to define their dependencies and container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC)

- **Spring Beans-**module provides the implementation of BeanFactory
- **Spring Context** – Spring Core and Spring Beans modules are the foundation of the Context (spring-context) module. This module provides the implementation of Application Context which is a mean to access objects in a framework-style manner that is similar to a JNDI registry. Context module inherits its features from the Beans module and adds support for other features like internationalization events propagation etc
- **Spring Expression Language-** This module is an extension to expression language supported by Java server pages and also represented as SpEL. This modules helps in achieving dynamic behavior.

Aspect Oriented Programming (AOP)

Logic of any application consists of two parts-

- **Business Logic –** is the actual logic that is intended to achieve the functionality.
- **Cross Cutting Concerns** – is the utility code that can be applied to business logic and required at several parts of the application like logging, transaction management etc.

The basic principle of AOP is finding common tasks which is required at many places in the code and does not belong to the business logic .Aspect Oriented Programming (AOP) provides a way of modularising application logic, so that each module addresses a distinct concern.

For example if we want to write a log statement on entry of every method of our application then instead of having logging logic at several places in an application, AOP provides a means of modularising this logic, and applying it to various parts of the application at runtime. This provides a clear Separation of Concerns and need not to be tied with business logic.

Data Access and Integration

Data Access and Integration consists of below five modules-

- **Transaction-** The transaction module provides supports programmatic and declarative transaction management that provides benefits like Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO). SimplerAPI for programmatic transaction management than complex transaction APIs such as JTA.
- **OXM (Object/XML Mapping) -**This module provides an abstraction layer that supports Object/XML mapping implementations such as JAXB, XMLBeans and JiBX.
- **ORM (Object Relationship Mapping) -** this module supports the integration of application with other ORM frameworks like JPA, JDO and Hibernate.
- **JDBC (Java Database Connectivity) -** We all have used JDBC somewhere to connect to the database and repetitive code need to be written every time. The module is kind of wrapper on JDBC which eliminates the need to repetitive and unnecessary exception handling overhead
- **JMS (Java Messaging Service) -**The JMS module (Java Messaging Service) includes features for sending and receiving messages between many clients

Web

Web layer includes below modules

- **Web-** The web module provides support for features like file upload, web application-context implementation etc.
- **Servlet –**This module is also known as spring-web-mvc module and provides MVC implementation for web applications.
- **Portlet-** This module is also known as spring-webmvc-portlet module provides the support for Spring based Portlets

Spring

As Spring Core module is base module for all other module, let us start our discussion with Spring Core Module

Spring Core

In an application we will have multiple classes; all the classes will not play the same role. Few classes are called Java beans, few are called pojo and some other are called component or bean classes.

Java bean: If a class only contains attributes with accessor methods (setters & getters) those are called java beans. These classes generally will not contain any business logic rather those are used for holding some data in it.

Pojo: If a class can be executed with underlying Jdk, without any other external third-party libraries support then it is called Pojo.

Component/Bean: If class has attributes with some member methods. Those member methods may use the attributes of that class and will fulfill some business functionality then it is called Component classes.

So a project cannot be built by just one. Say with Java beans we cannot complete the project rather we need business logic to perform something, so component classes will also exist in a project.

So, in a project do we have one component class or multiple component classes? By just having one we cannot complete rather we will have multiple component classes. Every component class in our project will talk to other or will not? Every component class cannot be isolated; it has to talk to some other component classes in the project to fulfill the functionality.

For e.g.

```
class A {
   public void m1() {
   }
}

class B {
   public int m2() {
   }
}
```

In the above code we have class A and class B. class A ml() method may have to talk to m2() method of the class B to fulfill business functionality, so these two classes are called dependent on each other as, unless B is there A cannot be used. So how to make the B available to A or how to manage the dependencies between A and B is what spring core is all about.

Now the question here is can I give any two classes and ask it to manage or spring cannot manage any arbitrary classes. Spring can manage any two arbitrary classes but in-order it to be effective spring recommends us to design our classes following some design guidelines, so that those components will be loosely-coupled. The spring can effective manage them.

Spring recommends us to design our classes following a design pattern called Strategy design pattern. Strategy design pattern is not belonging to spring, it is the pattern that comes from "Gang of four design patterns". If we design our classes following strategy design patterns the spring can manage them easily.

Strategy Design Pattern

Strategy pattern lets you build software as loosely coupled collection of interchangeable parts, in contrast with tightly coupled system. This loosely coupling makes your software much more flexible, extensible, maintainable and reusable.

Strategy pattern recommends mainly three principles every application should follow to get the above benefits those are as follows

1. **Favor Composition over Inheritance**
2. **Always design to Interfaces never code to Implementation**
3. **Code should be open for extension and closed for modification**

Intent of Strategy Design Pattern

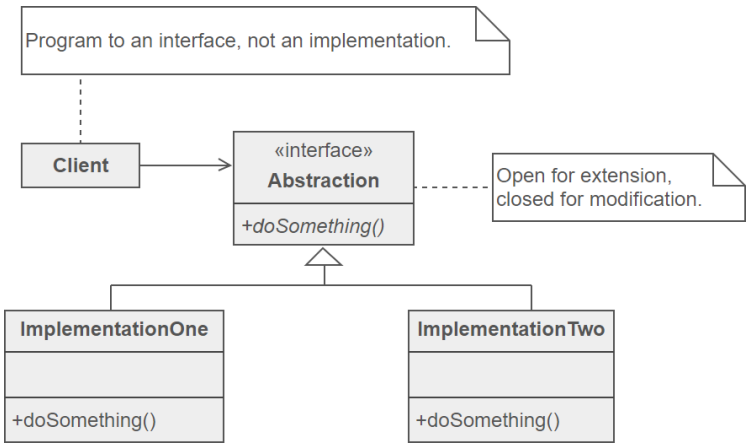
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Capture the abstraction in an interface, bury implementation details in derived classes.

Problem

One of the dominant strategies of object-oriented design is the "open-closed principle".

Figure demonstrates how this is routinely achieved - encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.



A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in figure is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . An object-oriented variant of the more generic exhortation "minimize coupling".

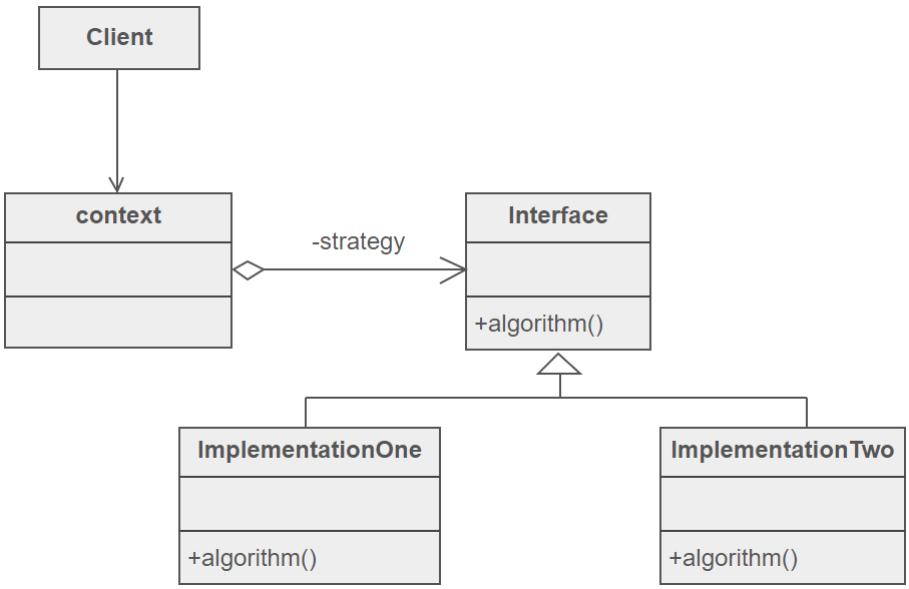
A more popular characterization of this "abstract coupling" principle is "Program to an interface, not an implementation".

Clients should prefer the "additional level of indirection" that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the "useful fiction") the client wants to exercise, and the implementations of that interface are effectively hidden.

Structure

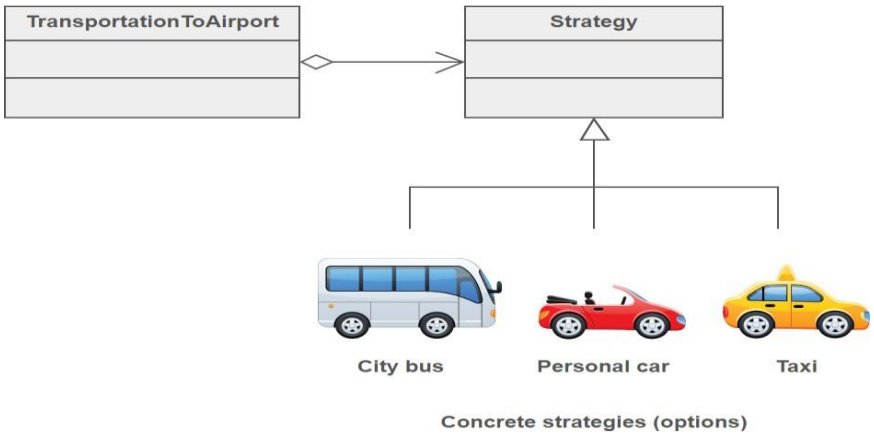
The Interface entity could represent either an abstract base class, or the method signature expectations by the client. In the former case, the inheritance hierarchy represents dynamic polymorphism. In the latter case, the Interface entity represents template code in the client and the inheritance hierarchy represents static polymorphism.

Spring



Example

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on trade-offs between cost, convenience, and time.



Check list

- Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
- Specify the signature for that algorithm in an interface.
- Bury the alternative implementation details in derived classes.
- Clients of the algorithm couple themselves to the interface.

IOC Container in spring

IOC is a principle, IOC means collaborating objects and managing the lifecycle of it. There are two ways using which we can collaborate objects

IOC is of two types : -Again, each type is classified into two more types as follows.

1. **Dependency - Lookup**
    i. **Dependency Pull**
    ii. **Contextual Dependency lookup**
2. **DependencyInjection**
    i. **SetterInjection**
    ii. **ConstructorInjection**

1) Dependency Lookup

In a nutshell, dependency lookup is when an object finds a way to create/instantiate/lookup its own dependencies (such as a private property)

i) Dependency Pull

If we take a J2EE Web application as example, we retrieve and store data from database for displaying the web pages. So, to display data your code requires connection object as dependent, generally the.connection object will be retrieved from a Connection Pool. Connection Pools are created and managed on an Application Server. Dependency pull is where class itself looks up for dependency and chooses appropriate implementation. Use of JNDI lookup for proper bean of datasource is a good example.

```
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jndiName");
```

ii) Contextual Dependency Lookup

In this technic your component and your environment/server will agree upon a contract/context, through which the dependent object will be injected into your code. For e.g If you see a Servlet API, if your servlet has to access environment specific information (init params or to get context) it needs ServletConfig object. But the ServletContainer will create the ServletConfig object. So in order access ServletConfig object in your servlet, you servlet has to implement Servlet interface and override init(ServletConfig) as parameter.

2) Dependency Injection

In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it.

Note: The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

The intent behind dependency injection is to decouple objects to the extent that no client code has to be changed simply because an object it depends on needs to be changed to a different one.

As with other forms of inversion of control, dependency injection supports the dependency inversion principle. The client delegates the responsibility of providing its dependencies to external code (the injector).

The client is not allowed to call the injector code. It is the injecting code that constructs the services and calls the client to inject them. This means the client code does not need to know about the injecting code. The client does not need to know how to construct the services. The client does not need to know which actual services it is using. The client only needs to know about the intrinsic interfaces of the services because these define how the client may use the services. This separates the responsibilities of use and construction.

There are three common means for a client to accept a dependency injection: setter-, interface- and constructor-based injection. Setter and constructor injection differ mainly by when they can be used. Interface injection differs in that the dependency is given a chance to control its own injection. All require that separate construction code (the injector) take responsibility for introducing a client and its dependencies to each other.

The Dependency Injection design pattern solves problems like:

1. **How can an application be independent of how its objects are created?**
2. **How can a class be independent of how the objects it requires are created?**
3. **How can the way objects are created be specified in separate configuration files?**
4. **How can an application support different configurations?**

Creating objects directly within the class that requires the objects is inflexible because it commits the class to particular objects and makes it impossible to change the instantiation later independently from (without having to change) the class. It stops the class from being reusable if other objects are required, and it makes the class hard to test because real objects can't be replaced with mock objects.

1. **The Dependency Injection design pattern describes how to solve such problems:**
2. **Define a separate (injector) object that creates and injects the objects a class requires.**
3. **A class accepts the objects it requires from an injector object instead of creating the objects directly.**
4. **This makes a class independent of how its objects are created (which concrete classes are instantiated).**
5. **A class is no longer responsible for creating the objects it requires, and it doesn't have to delegate instantiation to a factory object as in the Abstract Factory design pattern.**

This greatly simplifies a class and makes it easier to implement, change, test, and reuse.

If we want our class objects to be managed by spring then we should declare our classes as spring beans, any object that is managed by spring is called spring bean. Here the term manages refers to Object creation and Dependency injection (via setter, constructor etc...).

Basics - containers and beans

In spring, those objects that form the backbone of your application and that are managed by the Spring IoC container are referred to as beans. A bean is simply an object that is instantiated, assembled and otherwise managed by a Spring IoC container; other than that, there is nothing special about a bean (it is in all other respects one of probably many objects in your application). These beans, and the dependencies between them, are reflected in the configuration metadata used by a container.

Why... bean?

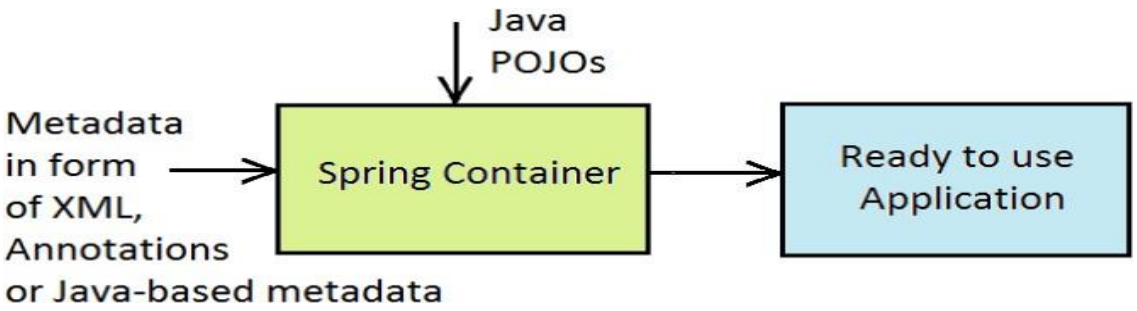
The motivation for using the name 'bean', as opposed to 'component' or 'object' is rooted in the origins of the Spring Framework itself (it arose partly as a response to the complexity of Enterprise JavaBeans).

The spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The spring container uses dependency injection (DI) to manage the components that make up an application. These objects are called Spring Beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code.

Spring

The following diagram is a high-level view of how spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



Configuration metadata

As can be seen in the above image, the Spring IoC container consumes some form of configuration metadata; this configuration metadata is nothing more than how you (as an application developer) inform the Spring container as to how to "instantiate, configure, and assemble [the objects in your application]". This configuration metadata is typically supplied in a simple and intuitive XML format. When using XML-based configuration metadata, you write bean definitions for those beans that you want the Spring IoC container to manage, and then let the container do it's stuff.

```
<?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="ccd" class="com.spapps.payment.beans.CrediCardPaymentImpl" />
       <bean id="debit" class="com.spapps.payment.beans.DebitCardPaymentImpl" />

       <bean id="paymentCtx" class="com.spapps.payment.beans.PaymentContext">
              <property name="paymentService" ref="ccd" />
       </bean>
</beans>
```

application-Context.xml

Responsibilities of Spring Containers

- **to instantiate the application class**
- **to configure the object**
- **to design the article**
- **to assemble the dependencies between the objects**

There are two ways of starting IOC container they are:

1. **BeanFactory**
2. **ApplicationContext**

BeanFactory: BeanFactory provides the basic support for DI and defined in org.springframework.beans.factory.BeanFactory interface. BeanFactory is based on factory design pattern which creates the beans with the only difference that it creates the bean of any type (as compared to traditional factory pattern which creates a beans of similar types).
BeanFactory Implementations**: There are several implementations of BeanFactory is available but out of all the most popular is XmlBeanFactory which loads the bean definitions from a XML file.**

Syntax to get BeanFactory

```
Resource resource=new ClassPathResource ("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory (resource);
```

The constructor of XmlBeanFactory class will take Resource object as parameter. So we need to pass the resource object to create the object of BeanFactory. Below are some of the Resource classes

- **ByteArrayResource**
- **ClassPathResource**
- **FileSystemResource**
- **InputStreamResource**
- **UrlResource**

ApplicationContext: **ApplicationContext is the advanced spring container and defined by org.springframework.context.ApplicationContext interface. Application supports the features supported by Bean Factory but also provides additional features like –**

- **Convenient MessageSource access (for i18n)**
- **ApplicationEvent publicationEvent handling**
- **Generic way to load resources**
- **Automatic BeanPostProcessor registration**
- **Automatic BeanFactoryPostProcessor registration**

ApplicationContext Implementations: There are several implementations of ApplicationContext is available but out of all the most commonly used are –

- **ClassPathXmlApplicationContext- Loads the file beans configuration from XML file available in class path.**
- **FileSystemXmlApplicationContext- Loads the file beans configuration from XML file available in file system.**
- **XmlWebApplicationContext- Applicable for web applications only and loads the beans configurations available in web application.**

Syntax to get ApplicationContext

```
ApplicationContext context =

        new ClassPathXmlApplicationContext("applicationContext.xml");
```

The constructor of ClasspathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

Users are sometimes unsure whether a BeanFactory or an ApplicationContext is best suited for use in a particular situation. A BeanFactory pretty much just instantiates and configures beans.

An ApplicationContext also does that, and it provides the supporting infrastructure to enable lots of enterprise-specific features such as transactions and AOP.

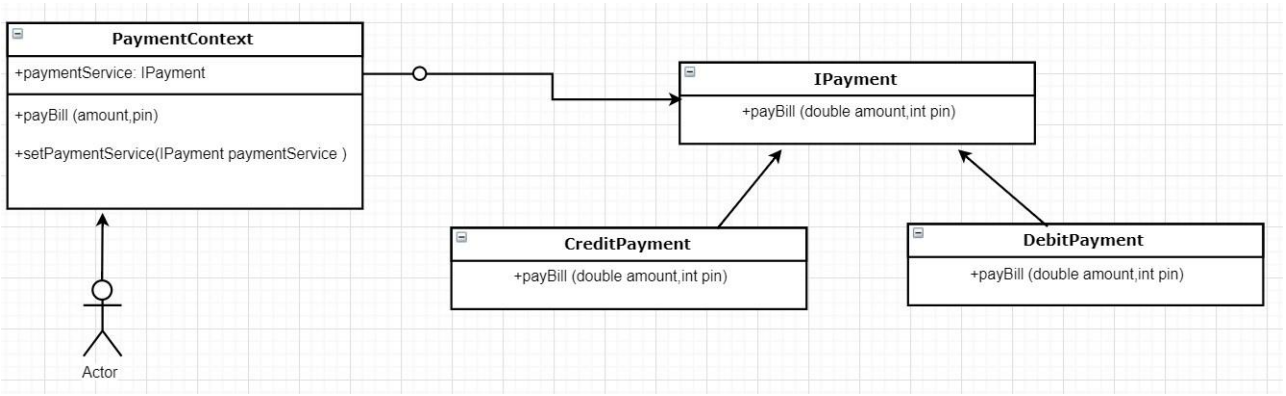
In short, favor the use of an ApplicationContext.

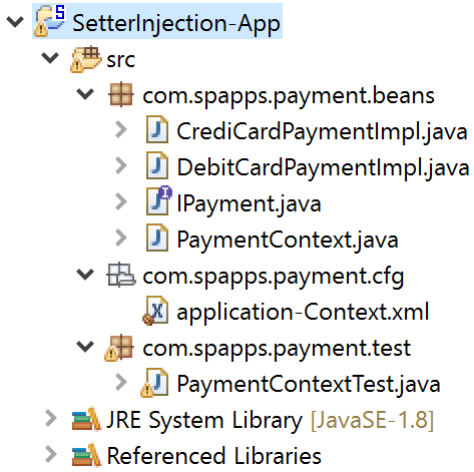
In short, the BeanFactory provides the configuration framework and basic functionality, while the ApplicationContext adds more enterprise-centric functionality to it. The ApplicationContext is a complete superset of the BeanFactory, and any description of BeanFactory capabilities and behavior is to be considered to apply to the ApplicationContext as well.

Spring

Setter Injection

In setter injection if an object depends on another object then the dependent object will be injected into the target class through the setter method that has been exposed on the target classes as shown below.





Below are the Bean classes which are implementing IPayment.java interface

```java
package com.spapps.payment.beans;

public interface IPayment {
	public boolean pay(double amount, int pin);
}
```

IPayment.java

```java
package com.spapps.payment.beans;

public class CrediCardPaymentImpl implements IPayment {

	public CrediCardPaymentImpl() {
		System.out.println("CCD Payment:0-param Const");
	}

	@Override
	public boolean pay(double amount, int pin) {
		System.out.println("Connecting to Paypal....");
		return true;
	}
}
```

CrediCardPaymentImpl.java

```java
package com.spapps.payment.beans;

public class DebitCardPaymentImpl implements IPayment {

	public DebitCardPaymentImpl() {
		System.out.println("Debit Card : 0-param Const");
	}

	@Override
	public boolean pay(double amount, int pin) {
		System.out.println("Connecting to Paypal....");
		return true;
	}
}
```

DebitCardPaymentImpl.java

```java
package com.spapps.payment.beans;

public class PaymentContext {

	private IPayment paymentService;

	public void setPaymentService(IPayment paymentService) {
		this.paymentService = paymentService;
	}

	public void payBill(Double amount, int pin) {
		boolean status = paymentService.pay(amount, pin);
		System.out.println("Payment Completed : " + status);
	}

}
```

PaymentContext.java

Below is the Spring Bean Configuration file which contians Bean definitions

```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="ccd" class="com.spapps.payment.beans.CrediCardPaymentImpl" />
       <bean id="debit" class="com.spapps.payment.beans.DebitCardPaymentImpl" />

       <bean id="paymentCtx" class="com.spapps.payment.beans.PaymentContext">
              <property name="paymentService" ref="ccd" />
       </bean>
</beans>
```

application-Context.xml

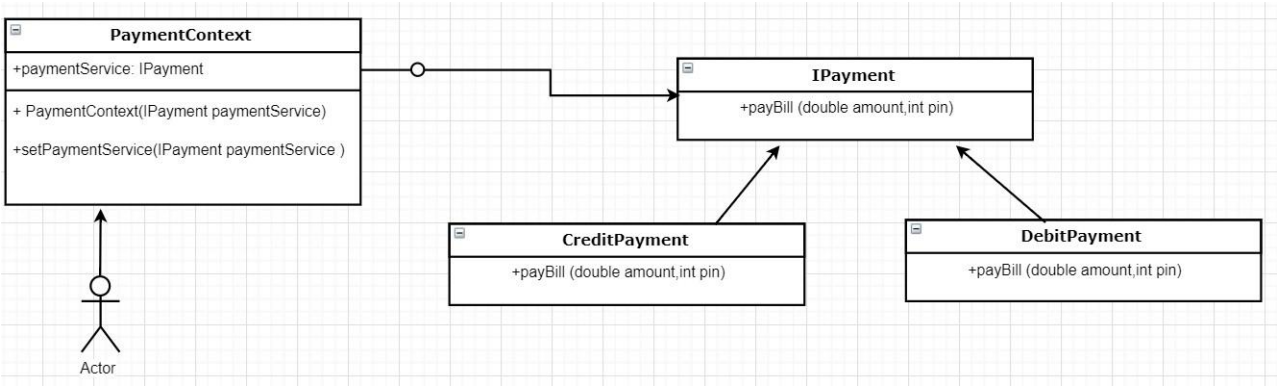
_Below is the Test class to start IOC contianer and Test our Application

```java
package com.spapps.payment.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import com.spapps.payment.beans.PaymentContext;

public class PaymentContextTest {

       public static void main(String[] args) {

              try {
                     Resource res = new ClassPathResource("com/sapps/payment/cfg/application-Context.xml");
                     BeanFactory factory = new XmlBeanFactory(res);

                     PaymentContext ctx = factory.getBean("paymentCtx", PaymentContext.class);
                     ctx.payBill(1000.00, 7979);
              } catch (Exception e) {
                     e.printStackTrace();
              }
       }
}
```

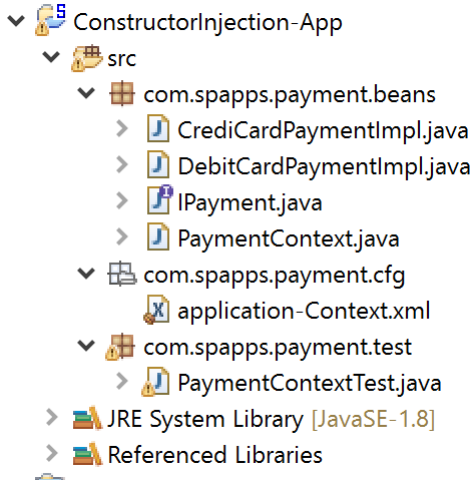
PaymentContextTest.java

Constructor Injection

In this technic instead of exposing a setter, your target class will expose a constructor, which will take the parameter of your dependent object. As your dependent object gets injected by calling the target class constructor, hence it is called constructor injection as shown below.



Constructor Injection – Example




```java
package com.spapps.payment.beans;

public interface IPayment {
	public boolean pay(double amount, int pin);
}
```

IPayment.java

```java
package com.spapps.payment.beans;

public class CrediCardPaymentImpl implements IPayment {

	public CrediCardPaymentImpl() {
		System.out.println("CCD Payment:0-param Const");
	}

	@Override
	public boolean pay(double amount, int pin) {
		System.out.println("Connecting to Paypal....");
		return true;
	}
}
```

CrediCardPaymentImpl.java

```java
package com.spapps.payment.beans;

public class PaymentContext {

	private IPayment paymentService;

	public PaymentContext(IPayment paymentService) {
		this.paymentService = paymentService;
	}

	public void payBill(Double amount, int pin) {
		boolean status = paymentService.pay(amount, pin);
		System.out.println("Payment Completed : " + status);
	}

}
```

PaymentContext.java

```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="ccd" class="com.spapps.payment.beans.CrediCardPaymentImpl" />
       <bean id="debit" class="com.spapps.payment.beans.DebitCardPaymentImpl" />

       <bean id="paymentCtx" class="com.spapps.payment.beans.PaymentContext">
              <constructor-arg name="paymentService" ref="ccd" />
       </bean>
</beans>
```

application-Context.xml

```java
package com.spapps.payment.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import com.spapps.payment.beans.PaymentContext;

public class PaymentContextTest {

       public static void main(String[] args) {

              try {
                     Resource res = new ClassPathResource("com/sapps/payment/cfg/application-Context.xml");
                     BeanFactory factory = new XmlBeanFactory(res);

                     PaymentContext ctx = factory.getBean("paymentCtx", PaymentContext.class);
                     ctx.payBill(1000.00, 7979);
              } catch (Exception e) {
                     e.printStackTrace();
              }
       }
}
```

PaymentContextTest.java

As we have seen above two types of Dependency Injection's, constructor injection and setter injection. Here are the points that make us understand when to use constructor injection and setter injection.

Differences between Setter and Constructor injection

Setter Injection (SI)	Constructor Injection (CI)
->SI will performed through target bean setter method	->CI will be performed throught target bean constructor
->Will be represent by using <property /> tag	-> Will be represented by using <constructor-arg /> tag
->Injecting all dependent beans are optional	-> Injecting all dependent beans are mandatory
-> Target bean obj will be created first	-> Dependent obj will be created first
-> Cyclic dependency injection is possible	-> Cyclic dependency injection is not possible

Injecting Collections in spring

We often need to define a properties of type collections like map, set, list etc in our java beans. Spring provides a convenient way to inject the collections in our beans.

Spring framework provides four type of collection (<list>, <set>, <map> and <props>) element to configure the collections.

<list>: This element is used to wire a list of values and is used for List and Arrays type of objects in java. Since list and arrays allows duplicates, so the duplicates are supported in <list> tag also. This tag can be used for both simple types and beans.

List of simple types can be defined as

```
<bean id="person" class="com.ci.beans.Person">
	<property name="personId" value="200" />
	<property name="personName" value="Mahesh" />
	<property name="visitedPlaces">
		<list value-type="java.lang.String">
			<value>Hyd</value>
			<value>Banglore</value>
			<value>101</value>
		</list>
	</property>
```

<set>: This element is used to wire a list of values and is used for Set type in java. Since set does allows duplicates, so the duplicates are not supported in <set> tag also. This tag can be used for both simple types and beans.

List of simple types can be defined as

```
<property name="courses">
	<set value-type="java.lang.String">
		<value>Java</value>
		<value>Java</value>
		<value>Oracle</value>
		<value>UI Technologies</value>
	</set>
</property>
```

<map> : This element is used to wire a key-value pair and used for java.util.Map type in java. In Maps, key and values can be of simple types and can be of object types so <map> tag also support both simple type and beans at both key and value level.

Elements supported by <map> tag are

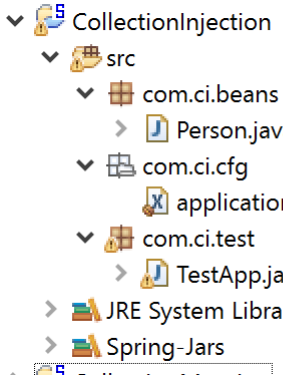
- **<key> - to define a key of simple type**
- **<key-ref> - to define a key referring to a bean**
- **<value> - to define simple type values**
- **<value-ref> - to define values referring to a bean**

```
<property name="parents">
	<map key-type="java.lang.String" value-type="java.lang.String">
		<entry key="father" value="Ramu" />
		<entry key="mother" value="Sita" />
	</map>
</property>
```

<props> : This element is used to define a java.util.Properties to wire a key-value pair similar to map with the only difference is that both key and value can be of type String only.

```
<property name="emails">
	<props>
		<prop key="office">ashok@oracle.com</prop>
		<prop key="personal">ashok@gmail.com</prop>
	</props>
</property>
```

Collection Injection – Example




```
package com.ci.beans;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class Person {

	private Integer personId;
	private String personName;
	private List<String> visitedPlaces;
	private Set<String> courses;
	private Map<String, String> parents;
	private Properties emails;

	//setters
}
```

Person.java

```
<bean id="person" class="com.ci.beans.Person">
        <property name="personId" value="200" />
        <property name="personName" value="Mahesh" />
        <property name="visitedPlaces">
                <list value-type="java.lang.String">
                        <value>Hyd</value>
                        <value>Banglore</value>
                        <value>101</value>
                </list>
        </property>

        <property name="courses">
                <set value-type="java.lang.String">
                        <value>Java</value>
                        <value>Java</value>
                        <value>Oracle</value>
                        <value>UI Technologies</value>
                </set>
        </property>
        <property name="parents">
                <map key-type="java.lang.String" value-type="java.lang.String">
                        <entry key="father" value="Ramu" />
                        <entry key="mother" value="Sita" />
                </map>
        </property>

        <property name="emails">
                <props>
                        <prop key="office">ashok@oracle.com</prop>
                        <prop key="personal">ashok@gmail.com</prop>
                </props>
        </property>
</bean>
```

```
public class TestApp {

        public static void main(String[] args) {
                ClassPathResource res = new ClassPathResource("com/ci/cfg/application-Context.xml");
                BeanFactory factory = new XmlBeanFactory(res);
                Person p = factory.getBean("person", Person.class);
                System.out.println(p);
        }
}
```

TestApp.java

Injecting Inner Beans in spring

Similar to the concept of inner classes in java, it is possible to define a bean inside another bean.e.g. In an ATM (Automated Teller Machine) system, it is possible for the Printer bean to be defined as an inner bean of ATM class.

The following are the typical characteristics of Spring inner beans:

a) When a bean does not need to be shared with other beans, then it can optionally be declared as an inner bean
b) Inner bean is defined within the context of its enclosing bean
c) Inner bean typically does not have an id associated with it although it is perfectly legal to provide an id. This is because, by its very definition, inner bean will not be shared outside of its enclosing bean. The value of the id attribute of an inner bean is ignored by spring.
d) Defining an inner bean does not imply that it should be defined as an inner class

E.g. Printer bean is defined as inner bean but Printer class is not an inner class.

e) The scope of an inner bean is always prototype. Spring will ignore the value of the scope attribute of an inner bean

The following are some of the limitations of using inner beans:

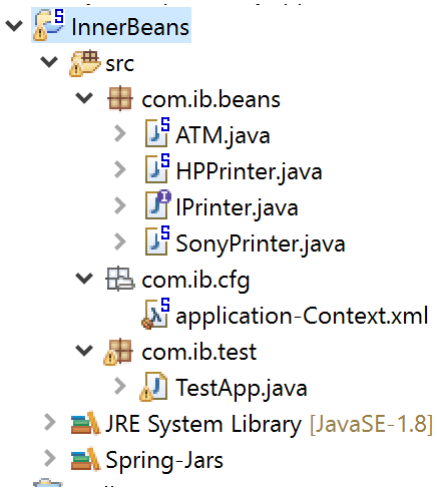
- Inner beans cannot be reused/shared with other beans
- In practice, defining inner beans affects the readability of Spring configuration XML

Injecting Inner Bean - Example

We will create an ATM bean and Printer bean. As the Printer bean is not referenced anywhere outside of the ATM class we will declare Printer bean as an inner bean (inside the enclosing ATM bean). The ATM.withdraw(double amount) method delegates the call to Printer.print() method.

Also, the Printer bean will be declared as an inner bean and will be injected into ATM bean using setter injection.

Note: Although in this program we are using setter injection, it is also possible to declare inner beans using constructor injection.



ATM.java class is our Target class (Need to inject Printer into ATM)

```
package com.ib.beans;

public class ATM {

	private double availableBal = 20000.00;

	private IPrinter printer;

	// To inject printer thorugh setter injection
	public void setPrinter(IPrinter printer) {
		this.printer = printer;
	}

	public void withdraw(double amount) {
		if (amount <= availableBal) {
			availableBal = availableBal - amount;
			printer.print(availableBal, amount);
		} else {
			printer.print("*****Insufficient Funds*****");
		}
	}
}
```

ATM.java

Below are the Dependent classes which are implementing IPrinter.java interface

```java
package com.ib.beans;

public interface IPrinter {

	public void print(double availableBal, double withdrawnAmt);

	public void print(String msg);

}
```

IPrinter.java

```java
package com.ib.beans;

public class SonyPrinter implements IPrinter {

	public SonyPrinter() {
		System.out.println("SonyPrinter : Injected");
	}

	@Override
	public void print(double availableBal, double withdrawnAmt) {
		System.out.println("Amount with drawn succesfully : " + withdrawnAmt);
		System.out.println("Available Balance : " + availableBal);
		System.out.println("Thankyou...!!");

	}

	@Override
	public void print(String msg) {
		System.out.println(msg);
	}

}
```

SonyPrinter.java

```java
package com.ib.beans;

public class HPPrinter implements IPrinter {

	public HPPrinter() {
		System.out.println("HPPrinter : Injected");
	}

	@Override
	public void print(double availableBal, double withdrawnAmt) {
		System.out.println("Amount with drawn succesfully : " + withdrawnAmt);
		System.out.println("Available Balance : " + availableBal);
		System.out.println("Thankyou...!!");
	}

	@Override
	public void print(String msg) {
		System.out.println(msg);
	}

}
```

HPPrinter.java

Spring Bean Configuration file

```xml
<bean id="atm" class="com.ib.beans.ATM">
		<property name="printer">
			<bean id="sony" class="com.ib.beans.SonyPrinter" />
			<!-- <bean id="hp" class="com.ib.beans.HPPrinter" /> -->
		</property>
</bean>
```

application-Context.xml

Test class to Start IOC container and get Bean object

```java
package com.ib.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import com.ib.beans.ATM;
import com.ib.beans.SonyPrinter;

public class TestApp {

	public static void main(String[] args) {

		Resource resource = new ClassPathResource("com/ib/cfg/application-Context.xml");
		BeanFactory factory = new XmlBeanFactory(resource);

		ATM atm = factory.getBean("atm", ATM.class);
		atm.withdraw(28888000.00);

	}

}
```

TestApp.java

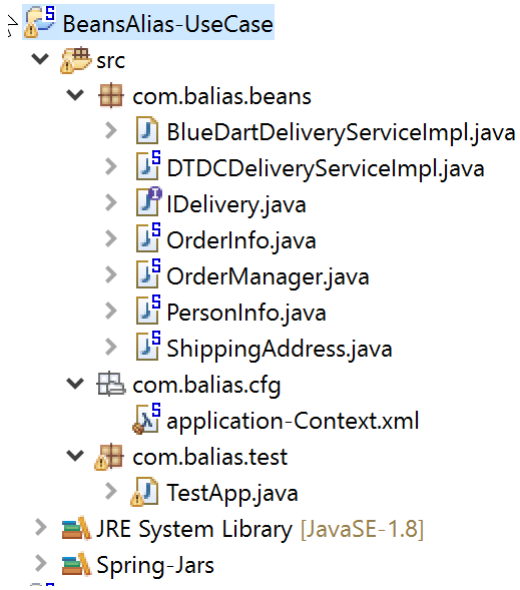
Beans Aliasing

In spring bean we can define alias of bean itself. We can keep more than one name of the same bean. Usability of alias is in the case where application is too large and we want loose coupling of even bean names among each other. We use alias as below.

Bean aliasing allows us to override already configured beans and to substitute them with a different object definition. This is most useful when the bean definitions are inherited from an external resource, which is out of our control.

```xml
<alias name="ent" alias="alias_ent"/>
```

Beal Alias – Example



OrderManager.java (Target class)

IDelivery.java (Interface – Delivery service class will implement)

BlueDartDeliveryServiceImpl.java & DTDCDeliveryServiceImpl.java (Impl classes-for delivery logic)

OrderInfo.java , PersonInfo.java & ShippingAddress.java (Model classes to hold the data)

TestApp.java (Test class to start IOC container & get Bean object)

Application-Context.xml (Spring Bean Configuration file)

```java
package com.balias.beans;

public interface IDelivery {

	public void deliverProduct(OrderInfo oInfo,
			PersonInfo pInfo, ShippingAddress sAddr);
}
```

IDelivery.java

```java
public class OrderInfo {

	private String orderId;
	private Set<String> items;
	private Double orderPrice;

	//setters  getters
}
```

OrderInfo.java

```java
package com.balias.beans;

public class PersonInfo {

	private String personName;
	private String email;
	private long mobileNo;

	//setters

}
```

PersonInfo.java

```java
package com.balias.beans;

public class ShippingAddress {

	private String area;
	private String city;
	private String state;
	private String country;
	private long zipcode;

	//setters & Getters

}
```

ShippingAddress.java

```java
package com.balias.beans;

public class OrderManager {

	private IDelivery blueDartDelivery, dtdcDelivery;
	private PersonInfo person;
	private OrderInfo order;
	private ShippingAddress addr;

	//setters & getters

	public void processOrder() {
		boolean isPaymentSuccs = processPayment(order.getOrderPrice());
		if (isPaymentSuccs) {
			if (addr.getZipcode() >= 5000) {
				// Deliver through blue dart
			blueDartDelivery.deliverProduct(order, person, addr);
			} else {
				// Deliver through DTDC
				dtdcDelivery.deliverProduct(order, person, addr);
			}
		}

	}

	public boolean processPayment(double amount) {
		System.out.println("Connecting to Paypal....");
		return true;
	}
}
```

OrderManager.java

```java
package com.balias.beans;

public class BlueDartDeliveryServiceImpl implements IDelivery {

	@Override
	public void deliverProduct(OrderInfo oInfo, PersonInfo pInfo,
ShippingAddress sAddr) {
		System.out.println("Delivery hand over to BLUE DART..");
		System.out.println("Order Delivered : " +
oInfo.getOrderId());
	}
}
```

BlueDartDeliveryServiceImpl.java

```java
package com.balias.beans;

public class DTDCDeliveryServiceImpl implements IDelivery {

	@Override
	public void deliverProduct(OrderInfo oInfo, PersonInfo pInfo, ShippingAddress sAddr) {
		System.out.println("Delivery handover to DTDC..");
		System.out.println("Order Delivered : " + oInfo.getOrderId());

	}

}
```

DTDCDeliveryServiceImpl.java

```xml
<bean id="pInfo" class="com.balias.beans.PersonInfo">
	<property name="personName" value="Rajesh" />
	<property name="mobileNo" value="97979797" />
	<property name="email" value="rajesh@gmail.com" />
</bean>

<bean id="orderInfo" class="com.balias.beans.OrderInfo">
	<property name="orderId" value="OD8686868" />
	<property name="orderPrice" value="10000.00" />
	<property name="items">
		<set value-type="java.lang.String">
			<value>Apple IPhone 6s</value>
			<value>Fastrack Wrist Watch</value>
		</set>
	</property>
</bean>
```

```xml
<bean id="addr" class="com.balias.beans.ShippingAddress">
		<property name="area" value="S.R.Nagrr" />
		<property name="city" value="HYD" />
		<property name="state" value="TG" />
		<property name="country" value="India"></property>
		<property name="zipcode" value="7500369" />
	</bean>

<!--
	<bean id="bluedart" class="com.balias.beans.BlueDartDeliveryServiceImpl"
/> -->
<bean id="dtdc" class="com.balias.beans.DTDCDeliveryServiceImpl" />
<alias name="dtdc" alias="bluedart" />
```

application-Context.xml

```java
package com.balias.test;

import com.balias.beans.OrderManager;

public class TestApp {

	public static void main(String[] args) {
		Resource resource = new ClassPathResource("com/balias/cfg/application-Context.xml");
		BeanFactory factory = new XmlBeanFactory(resource);
		OrderManager om = factory.getBean("orderManager", OrderManager.class);
		om.processOrder();
	}
}
```

TestApp.java

Bean Inheritance

Inheritance is the concept of reusing the existing functionality. In case of java you can inherit a class from an Interface or another class. When you inherit a class from another class, your child or derived class can use all of the functionalities of your base class.

A bean in spring is a class configured with some configuration around it using which spring creates the object for it. Bean inheritance is the concept of re-using the configuration property values of one bean inside another bean is called bean inheritance.

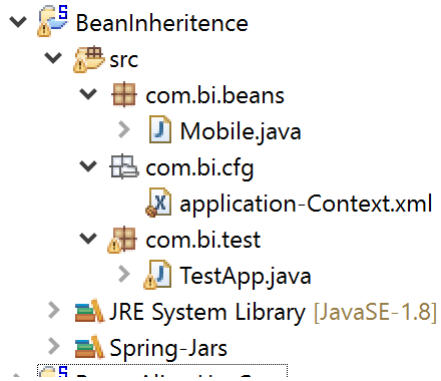
I have a class and I configured it as a bean with some values that has to be injected while creating. For the same class I want 10 beans of such type, so how many times I need to configure it as bean, with the same configuration changing the bean id I need to configure 10 beans.

This makes your configuration bulgier (more) and any change for one of the value of the property need to be changed across all the 10 beans, duplication of configuration results in more maintenance cost.

Is there any alternate to copy the values of one bean into another bean rather than re• declaring the entire configuration? That's where bean inheritance comes into picture.

For e.g. we own a mobile showroom and it has some mobile, moble is represented as an object of class Mobile. A showroom contains several mobiles to represent. Let's see how we can configure several beans using bean inheritance.

Bean Inheritance - Example




```java
package com.bi.beans;

public class Mobile {

	private Integer mobileId;
	private String name;
	private String color;
	private Double price;
	private String imeiNo;

	//setters & toString()
}
```

Mobile.java

```xml
<bean id="baseMobile" class="com.bi.beans.Mobile" abstract="true">
	<property name="mobileId" value="101" />
	<property name="imeiNo" value="79797979" />
	<property name="name" value="Iphone-6s" />
	<property name="color" value="Grey" />
	<property name="price" value="30000.00" />
</bean>

<bean id="m2" class="com.bi.beans.Mobile" parent="baseMobile">
	<property name="mobileId" value="102" />
	<property name="imeiNo" value="46446868" />
</bean>

<bean id="m3" class="com.bi.beans.Mobile" parent="baseMobile">
	<property name="mobileId" value="103" />
	<property name="imeiNo" value="68688868" />
</bean>
```

application-Context.xml

```java
package com.bi.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.bi.beans.Mobile;

public class TestApp {

	public static void main(String[] args) {
		ClassPathResource res = new ClassPathResource("com/bi/cfg/application-Context.xml");
		BeanFactory factory = new XmlBeanFactory(res);

		Mobile m1 = factory.getBean("m2", Mobile.class);
		System.out.println(m1);

		Mobile m2 = factory.getBean("m3", Mobile.class);
		System.out.println(m2);
	}
}
```

TestApp.java

Few points to remember

- When we use bean inheritance, the parent bean and the child bean class types are not necessary to be same. All the properties of the parent bean must and should be present in child bean only
- When we inherit one bean from another bean, only the parent bean property values will be copied into child bean property values but the physical classes will never get extended.
- It is recommended to declare one of the beans as abstract bean which contains all the common values that should be inherited to the child. As it is an abstract bean we never modify any property unless it has to be affected to all.

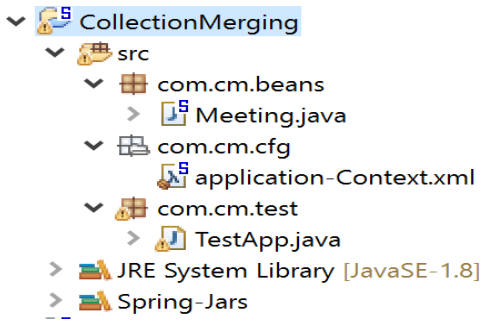
- When we declare a bean as abstract, the class will not become abstract only the current bean definition will become abstract, which means spring wilt not create the object of that bean. Any call to factory.getBean("abstractbean") to an abstract bean will results out in an error.

Collection Merging

The merging of one bean property collection that's dependent on another bean property collection dependent is called a collection merge.

In spring, java collections can be injected in bean using application xml. According to the need, we can merge collection data while creating bean. There will be a parent and child bean. Child will inherit the parent data. It is done by the attribute merge="true". This attribute is available in all collection tags like <list/>, <set/>, <map/> and <props/> . The child bean will use parent attribute to inherit parent bean data. The child bean collection tag will use merge="true" attribute to inherit parent collection elements. When we access child bean, we will get elements of child as well as parent bean collection. This is collection merging of beans in spring. To understand about spring collection injection.

Collection Merging – Example




```java
package com.cm.beans;

import java.util.Set;

public class Meeting {

	private String meetingType;
	private Set<String> participants;

	public void setMeetingType(String meetingType) {
		this.meetingType = meetingType;
	}

	public void setParticipants(Set<String> participants) {
		this.participants = participants;
	}

	@Override
	public String toString() {
		return "Meeting [meetingType=" + meetingType + ", participants=" + participants + "]";
	}

}
```

Meeting.java

```
<bean id="standUpcall" class="com.cm.beans.Meeting"  >
        <property name="meetingType" value="Standup Meeting" />
        <property name="participants">
                <set value-type="java.lang.String" merge="true">
                        <value>Sita</value>
                        <value>Gita</value>
                        <value>Ramesh</value>
                        <value>Suresh</value>
                </set>
        </property>
</bean>

<bean id="weeklyMeeting" class="com.cm.beans.Meeting" parent="standUpcall">
        <property name="meetingType" value="Weekly Status Meeting" />
        <property name="participants">
                <set value-type="java.lang.String" merge="true">
                        <value>Mohan</value>
                        <value>Aushtosh</value>
                </set>
        </property>
</bean>
```

```
package com.cm.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import com.cm.beans.Meeting;

public class TestApp {

        public static void main(String[] args) {
                Resource resource = new ClassPathResource("com/cm/cfg/application-Context.xml");
                BeanFactory factory = new XmlBeanFactory(resource);
                Meeting weeklyMeeting = factory.getBean("weeklyMeeting", Meeting.class);
                System.out.println(weeklyMeeting);
        }
}
```

TestApp.java

Few points to remember about collection merging:

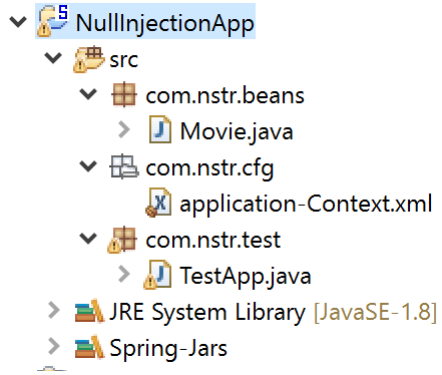
- **Collection merging always comes into picture in-case of bean inheritance only.**
- **The parent property collection type and child property collection type we are merging should be of same type.**
- **The parent property collection generic type and the chil property collection generic type should always be same**

Null Injection

The concept of Null string is how to pass Null Value for a Bean property. Let's consider a case where a class has attribute as String or other Object. We are trying to inject the value of this attribute using Constructor Injection.

In case of constructor injection the dependent object is mandatory to be injected in configuration. In case if the dependent object is not available, you can pass null for the dependent object in the target class constructor as shown below.

Null Injection – Example




```java
package com.nstr.beans;

public class Movie {

	private Integer movieId;
	private String movieName;
	private String hero;
	private String heroine;

	public Movie(Integer movieId, String movieName, String hero, String heroine) {
		this.movieId = movieId;
		this.movieName = movieName;
		this.hero = hero;
		this.heroine = heroine;
	}

	@Override
	public String toString() {
		return "Movie [movieId=" + movieId + ", movieName=" + movieName + ", hero=" +
hero + ", heroine=" + heroine
				+ "]";
	}

}
```

Movie.java

```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="movie" class="com.nstr.beans.Movie">
		<constructor-arg name="movieId" value="101" />
		<constructor-arg name="hero" value="Prabhas" />
		<constructor-arg name="heroine" value="Kajal" />
		<constructor-arg name="movieName">
			<null />
		</constructor-arg>
	</bean>
</beans>
```

application-Context.xml

```
package com.nstr.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.nstr.beans.Movie;

public class TestApp {

	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource("com/nstr/cfg/application-Context.xml"));
		Movie m = factory.getBean("movie", Movie.class);
		System.out.println(m);
	}
}
```

TestApp.java

Spring Bean Scopes

A scope can be considered as a logical boundary within which a bean is deployed and known to the application. There are multiple types of scopes available in spring.

Before moving on to the scopes defined by spring, let us understand how a scope in general can be enforced. Consider a scenario where I have a web application, I want a user's information to be available only to him/her during his session. In such a scenario, I have to define the scope of the user object with the following considerations:

* **It must get created with the login**
* **It should be destroyed with logout or may be inactivity for a pre-configured period.**

In traditional applications, all this was done using pure Java code and everything was hand coded. A developer had to write the code carefully such that it will destroy the user object

* **after detecting a log out or**
* **Certain period of inactivity.**

There are many such scenarios, not just limited to the scope of the session of course.

Consider another scenario where a Database Connection Pool has to be created. Now in a typical application, such a connection pool exists one per data source. This is typically implemented as a singleton pattern which guarantees only one connection pool object per data source.

Spring framework supports five type of scopes and for bean instantiation and

- **Singleton- This is the default scope of the bean in Spring and it means that container will create a single instance per Spring IoC container.**
- **Prototype- This scope allows the bean to be instantiated whenever it is requested which means multiple instance of a bean can be available.**
- **Request- Spring bean configured as request scope instantiates the bean for a single HTTP request. Since HTTP requests are available only in web applications this scope is valid only for web-aware spring application context.**
- **Session – Spring Beans configured as session scope lives through the HTTP session. Similar to request scope, it is applicable only for web aware spring application contexts.**
- **global_session- This scope is equal to the session scope on portlet applications. This scope is also applicable only for web aware spring application contexts. If this is global_session is used in normal web application (not in portlet), then it will behave as session scope and there will not be any error. (The global-session scope has been removed from Spring 3.0)**

Define Bean Scopes: We can define a scope of bean in bean definition itself using scope attribute of bean tag

<bean id="bean_id" class="class_name" scope=" ">

Singleton Scope: The most important and widely used is the singleton scope. In fact this is the default scope for a Spring bean. If no scope information is provided then by default the bean is created to be deployed in a singleton scope.

However, this singleton is not exactly similar to the objects created using the Singleton Design Pattern. The Singleton Design Pattern dictates hard coding the scope of the object. This enforces creation of only one instance of the particular class per class loader. Spring guarantees exactly one shared bean instance for the given bean definition per IoC container.

How does spring guarantees a single instance per container?

Once a singleton bean is created by the container, it is kept in a container specific singleton cache. The next time if a request is made with either the id, name or alias of the bean it is returned from the cache. Due to the nature of the beans having singleton scope, it is pretty obvious that these beans can be easily used as a stateless bean.

For e.g.: If you want to create a DAO object, which saves customer data in the database, then the DAO class is created as a singleton. This works because the DAO class need not care about the state. It should behave the same way irrespective of the Customer object we are trying to persist.

Below two statements are equivalent

```
<bean id="singletonBean" class="SingletonBean"/>
<bean id="singletonBean" scope="singleton" class="SingletonBean"/>
```

Example to demonstrate the singleton scope

a) Define a bean (SingletonBean)

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

b) Create a beans.xml file in src directory to define the SingletonBean

```
<?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-3.0.xsd">

<bean  id="singletonBean" scope="singleton" class="SingletonBean" />

</beans>
```

Prototype Scope: Another important scope is the prototype scope. Unlike the singleton scoped beans, the prototype scoped bean deployment results in creation of a new instance of the bean upon request for bean. From our example above, a good candidate for the prototype scope is the Customer bean. Hence, all the stateful objects can be scoped as prototype. To scope a bean as prototype, you need to explicitly mention the scope in the bean metadata like below

```
<bean id="prototypeBean" scope="prototype" class="PrototypeBean"/>
<bean id="prototypeBean" singleton="false" class="PrototypeBean"/>
```

Whenever a bean configured with prototype scope is requested , container creates a new instance of bean.
Stateful beans which hold the conversational state should be declared as prototype

To create a bean in prototype scope, use scope=”prototype” or singleton=”false”

Spring Beans Autowiring

In spring when you want to inject one bean into another bean, we need to declare the dependencies between the beans using <property> or <constructor-arg> tag in the configuration file. This indicates we need to specify the dependencies between the beans and spring will reads the declarations and performs injection.

In spring, beans can be wired together in two ways: Manually and Autowiring

Manual Wiring: Using <ref> attribute in <property> or <constructor-arg> tag
Auto wiring: In Auto-wiring instead of we declaring the dependencies we will instruct spring to automatically detect the dependencies and perform injection between· them. This we can achieve using autowire attribute in <bean> tag.
Syntax: <bean id="bean_id" class="bean_class" autowire=" mode " />

Advantage of Autowiring

It requires the less code because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

- **No control for programmer.**
- **It can't be used for primitive and string values.**

Following are the different autowiring modes supported by spring.

No.	Mode	Description
1-	no	It is the default autowiring mode. It means no autowiring bydefault.
2-	byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
3-	byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
4-	constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
5-	autodetect	It is deprecated since Spring 3.

Note: Explicit dependencies using < property> and <constructor-arg> settings always override the auto wiring.

no : This is default setting and there will be no auto wiring. We need to explicitly wires the bean using ref attribute and this is what we have done in earlier chapters.

Default value of autowire attribute of bean is “default” which means no autowiring will be performed.
Setting autowire attribute with value “no” will have the same behavior.

Below three statements are equivalent

```
<bean id="bean_id" class="bean_class" autowire="default" />
<bean id="bean_id" class="bean_class" autowire="no " />
```

```
<bean id="bean_id" class="bean_class" />
```

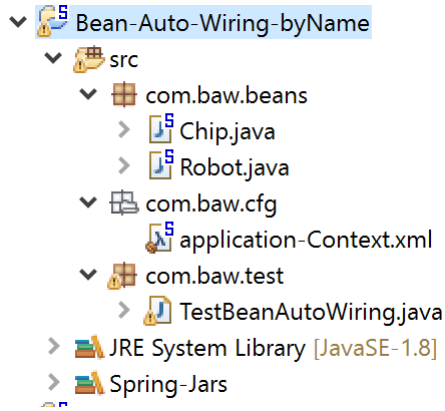
byName : In this mode autowiring performs based on the name of property. Container inspects the spring configuration file and looks for the bean having name or id attribute same as the property name. This mode is applied on Setter methods.

Since the we cannot have a multiple beans in configuration files having same value of id attribute, thus there is no chance having a multiple match for a property. In case bean with given name or id does not found , that property will remain unset.

Refer below snippet on how to configure byName auto wiring mode

```
<bean id="bean_id" class="bean_class" autowire="byName"/>
```

byName-Example



Robot.java (Target class)
Chip.java (Dependent – Will be injected into Robot.java)

```java
package com.baw.beans;

public class Chip {

	private Integer chipId;
	private String chipType;

	//setters &getters toString()

}
```

Chip.java

```java
package com.baw.beans;

public class Robot {

	private Integer robotId;
	private String robotType;

	private Chip chip;

	//setters & toString()
}
```

Robot.java

```
<bean id="robot" class="com.baw.beans.Robot" autowire="byName">
		<property name="robotId" value="5001" />
		<property name="robotType" value="Military Robots" />
	</bean>

	<bean id="chip" class="com.baw.beans.Chip" autowire-candidate="false">
		<property name="chipId" value="1002" />
		<property name="chipType" value="Chip-64-Bit" />
	</bean>
```

application-Context.xml

```
public class TestBeanAutoWiring {

	public static void main(String[] args) {
		Resource res = new ClassPathResource("com/baw/cfg/application-Context.xml");
		BeanFactory factory = new XmlBeanFactory(res);
		Robot r = factory.getBean("robot", Robot.class);
		System.out.println(r);
	}
}
```

TestBeanAutoWiring.java

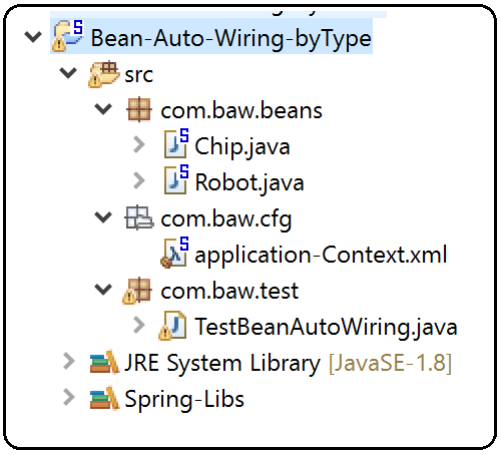
byType : **In this mode autowiring performs based on the class type of property. Container inspects the spring configuration file and looks for the bean of same class type. This mode is applied on Setter methods.**

* **If no bean with class type found than property will remain unset**
* **If one bean with class type found then property will be set**
* **If there are multiple beans found with class type in beans configuration found, then exception will be thrown.**

Refer below snippet on how to configure byType auto wiring mode

```
<bean id="bean_id" class="bean_class" autowire="byType"/>
```

byType-Example




```
package com.baw.beans;

public class Chip {

	private Integer chipId;
	private String chipType;

	//setters &getters toString()

}
```

Chip.java

```
package com.baw.beans;

public class Robot {

	private Integer robotId;
	private String robotType;

	private Chip chip;

	//setters & toString()
}
```

Robot.java

```xml
<bean id="robot" class="com.baw.beans.Robot" autowire="byType">
	<property name="robotId" value="5001" />
	<property name="robotType" value="Military Robots" />
</bean>

<bean id="chip32Bit" class="com.baw.beans.Chip">
	<property name="chipId" value="1001" />
	<property name="chipType" value="Chip-32-Bit" />
</bean>
```

application-Context.xml

```java
public class TestBeanAutoWiring {

	public static void main(String[] args) {

	Resource res = new ClassPathResource("com/baw/cfg/application-Context.xml");
		BeanFactory factory = new XmlBeanFactory(res);
		Robot r = factory.getBean("robot", Robot.class);
		System.out.println(r);
	}
}
```

TestBeanAutoWiring.java

Note : if we configure Chip bean for more than one time with different ids then IOC will throw an Exception. To overcome this duplicate beans problem, we can use autowire-candidate="false" attribute like below

```xml
<bean id="robot" class="com.baw.beans.Robot" autowire="byName">
	<property name="robotId" value="5001" />
	<property name="robotType" value="Military Robot" />
</bean>

<bean id="chip32Bit" class="com.baw.beans.Chip" autowire-candidate="false">
	<property name="chipId" value="1001" />
	<property name="chipType" value="Chip-32-Bit" />
</bean>


<bean id="chip64Bit" class="com.baw.beans.Chip">
	<property name="chipId" value="1002" />
	<property name="chipType" value="Chip-64-Bit" />
</bean>
```

application-Context.xml

As per above configuration, chip32Bit bean will be ignored and chip64Bit bean will be injected.

Constructor: This **auto wiring mode is similar to byType with the difference is that this mode is applied on constructor arguments where as byType applies on setter methods of properties.**

* If no bean with class type found in beans configuration found, then exception will be thrown.
* If one bean with class type found then property will be set
* If there are multiple beans found with class type in beans configuration found, then exception will be thrown if default constructor is not defined else default constructor will be used.

Refer below snippet on how to configure constructor auto wiring mode

```xml
<bean id="bean_id" class="bean_class" autowire="constructor"/>
```

```java
package com.baw.beans;

public class Robot {

	private Integer robotId;
	private String robotType;

	private Chip chip;

	public Robot(Chip chip) {
		this.chip = chip;
	}
	//setters & toString()
}
```

Robot.java

```java
public class Chip {

	private Integer chipId;
	private String chipType;
	//setters & toString( )
}
```

Chip.java

```xml
<bean id="robot" class="com.baw.beans.Robot" autowire="constructor">
	<property name="robotId" value="5001" />
	<property name="robotType" value="Military Robots" />
</bean>

<bean id="chip32Bit" class="com.baw.beans.Chip">
	<property name="chipId" value="1002" />
	<property name="chipType" value="Chip-64-Bit" />
</bean>
```

application-Context.xml

Spring Bean Life Cycle

Life of traditional java objects starts on calling new operator which instantiates the object and finalize () method is getting called when the object is eligible for garbage collection. Life cycle of spring beans are different as compared to traditional java objects.

If you consider a Servlet, J2EE containers will call the init () and destroy() methods after creating the servlet object and before removing it from the web container respectively. In the Servlet init method, developer has to write the initialization logic to initialize.

In Spring framework, IoC container is responsible for managing the Spring Bean Life Cycle, the life cycle of beans consist of call back methods such as Post initialization call back method and Pre destruction call back method. Below steps are followed by Spring IoC Container to manage bean life cycle.

1. **Creation of bean instance by a factory method.**
2. **Set the values and bean references to the bean properties.**
3. **Call the initialization call back method.**
4. **Bean is ready for use.**
5. **Call the destruction call back method.**

Spring framework provides the following ways which can be used to control the lifecycle of bean:

- Declarative approach
- Programmatic approach
- Annotation based approach

Declarative Approach

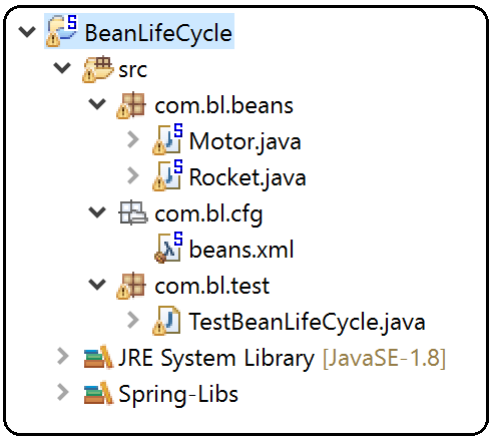
In the declarative approach you will declare the init and destroy methods of a bean in spring beans configuration file. In the bean class declare the methods whose signature must be public and should have return type as void and should not take any parameters, any method which follows this signature can be used as lifecycle methods

After writing the methods in your bean, you need to declare those methods as lifecycle methods in spring configuration file, at the <bean> tag level, you need to declare two attributes init-method = "methodname" and destroy-method ="destroymethod"

Core container after creating the bean (this includes after all injections); it will automatically calls the init method to perform the initialization. But spring roe container cannot automatically calls the destroy-method of the bean class, because in spring core application roe container will not be able to judge when the bean is available for garbage collection or how many references of this bean is held by other beans.

In order to invoke the destroy-method on the bean class you need to explicitly call on the eonfigurablelistableBeanFactory, destroySingleton or destroyScopedBean("beanscope") method, so that roe container delegates the call to all the beans destroy-methods in that roe container.

Spring Bean Lifecycle – Example




```
package com.bl.beans;

public class Motor {

	private String name;

	public void setName(String name) {
		this.name = name;
	}

	public void doingWork() {
		System.out.println(name + ": Motor Running.....");
	}

	public void startup() {
		System.out.println("startup() : Starting....");
	}

	public void stop() {
		System.out.println("stop() : Releasing resource");
	}
}
```

Motor.java

```
<bean id="motor" class="com.bl.beans.Motor" init-method="startup"  destroy-method="stop">
	<property name="name" value="Diesel Motor" />
</bean>
```

beans.xml

```
public class MyApp {
	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
		Motor m = (Motor) context.getBean("motor");
		System.out.println(m);
	}
}
```

MyApp.java

Programmatic Approach : The problem with declarative approach is developer has provide the init and destroy method declarations for each bean of that class in configuration, if he misses for atleast one bean, then the initialization or destruction will not happen. It would be tough to maintain the configuration, lets say if the init or destroy method name changed in class, all the references to those methods in the configuration has to be modified to match with class method names.

In order to avoid the problems with this spring has provided programmatic approach.In this the bean class has to implement from two interfaces

InitalizingBean interface to handle initialization process and DisposableBean interface to handle destruction process and should override afterPropertiesSet and destroy methods respectively.

Now the developer don't need to declare these methods as init-method or destroy-method rather the core container while creating these beans will detects

These beans as InitializingBean type or DisposableBean type and calls the afterPropertiesSet and destroy method automatically.

InitializingBean and DisposbleBean callback interfaces

- InitalizingBean interface is defined under org.springframework.beans.factory package and declares a single method where we can be used to add any initialization related code. Any bean implementing InitalizingBean needs to provide an implementation of afterPropertiesSet() method. Signature of method is:
  void afterPropertiesSet() throws Exception;
- Similarly DisposableBean interface is defined under the org.springframework.beans.factory and declares a single method which gets executed when bean is destroyed and can be used to add any cleanup related code. Any bean implementing DisposableBean needs to provide an implementation of destroy() method. Signature of method is :
  void destroy() throws Exception;

Bean class which is implementing Callback interfaces to perform Bean Life cycle

```
package com.bl.beans;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Motor implements InitializingBean, DisposableBean {
        private String name;

        public void setName(String name) {
                this.name = name;
        }

        public void doingWork() {
                System.out.println(name + ": Motor Running.....");
        }
        @Override
        public void afterPropertiesSet() throws Exception {
                System.out.println("afterPropertiesSet called:");
                System.out.println("Motor starting.....");
        }
        @Override
        public void destroy() throws Exception {
                System.out.println("destory() : releasing resource");
        }
}
```

Motor.java

```
public class MyApp {
        public static void main(String[] args) {
                ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
                Motor m = (Motor) context.getBean("motor");
                System.out.println(m);
                m.doWork();
                ((AbstractApplicationContext) context).registerShutdownHook();
        }
}
```

MyApp.java

This approach is simple to use but it's not recommended because it will create tight coupling with the Spring framework in our bean implementations.

@PostConstruct and @PreDestroy annotations

Spring 2.5 onwards we can use annotations to specify life cycle methods using @PostConstruct and @PreDestroy annotations. Details on the annotation will be covered in upcoming chapters.

P-Namespace

Spring's P namespace is an alternative to using the property tag. By using p namespace, we can perform dependency injection by directly using the attribute of bean tag instead of using the property tag.

The benefits of using p namespace are:

- p namespace is more compact than property tag
- Using p namespace, we can reduce the amount of XML required in Spring configuration.

P-Namespace Example

Person.java (Target class)

Address.java (Dependent class and will be injected into Person.java)

```
public class Person {

   private String firstName;
   private String lastName;
   private Integer age;

   private Address address;

   //setter methods
   //toString ( )

}
```

Person.java

```
public class Address {

    private Integer houseNo;
    private String street;
    private String city;
    private String zipCode;

     //setter methods
    // toString method
}
```

Address.java

Configuring above classes as Spring Beans using traditional approach (without P-Namespace) - <property /> tags we are writing

```
<?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="addr1" class="com.ashok.cpns.Address">
    <property name="houseNo" value="2106"/>
    <property name="street" value="Newbury St"/>
    <property name="city" value="Boston"/>
    <property name="zipCode" value="02115-2703"/>
  </bean>

  <bean id="constArgAnotherBean" class="com.ashok.cpns.Person">
    <property name="firstName" value="Joe"/>
    <property name="lastName" value="John"/>
    <property name="age" value="35"/>
    <property name="address" ref="addr1"/>
  </bean>

</beans>
```

Configuring above class as Spring Beans using P-Namespace (No need to <property /> tags)

```
<?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:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean name="addr1" class="com.ashok.cpns.Address"
    p:houseNo="2106" p:street="Newbury St"
    p:city="Boston" p:zipCode="02115-2703"/>

  <bean name="person1" class="com.ashok.cpns.Person"
    p:firstName="Joe" p:lastName="John"
    p:age="35" p:address-ref="addr1"/>

</beans>
```

It is simple! All we need to do is:

- Add the p namespace URI (xmlns:p="http://www.springframework.org/schema/p") at the configuration xml with namespace prefix p. Please note that it is not mandatory that you should use the prefix p, you can use your own sweet namespace prefix
- For each nested property tag, add an attribute at the bean with name having a format of p:<property-name> and set attribute value to the value specified in the nested property element.
- If one of your property is expecting another bean, then you should suffix -ref to your property name while constructing the attribute name to use in bean definition. For eg, if your property name is address then you should use attribute name as address-ref.

c-namespace (Constructor namespace)

c-namespace is similar to p-namespace but used with constructor-based dependency injection code. It simplifies the constructor-based injection code by enabling you to use the constructor arguments as attributes of bean definition rather than the nested <constructor-arg> elements.

C-Namespace example

Configuring Constructor-arguments without using C-Namespace

```
<bean id="addr1" class="com.ashok.cpns.Address">
 <constructor-arg name="houseNo" value="2106"/>
 <constructor-arg name="street" value="Newbury St"/>
 <constructor-arg name="city" value="Boston"/>
 <constructor-arg name="zipCode" value="02115-2703"/>
</bean>

<bean id="constArgAnotherBean" class="com.ashok.cpns.Person">
 <constructor-arg name="firstName" value="Joe"/>
 <constructor-arg name="lastName" value="John"/>
 <constructor-arg name="age" value="35"/>
 <constructor-arg name="address" ref="addr1"/>
</bean>
```

Configuring Constructor-arguments using C-Namespace

```
<?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:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean name="addr1" class="com.ashok.cpns.Address"
    c:houseNo="2106" c:street="Newbury St"
    c:city="Boston" c:zipCode="02115-2703"/>

  <bean name="person1" class="com.ashok.cpns.Person"
    c:firstName="Joe" c:lastName="John"
    c:age="35" c:address-ref="addr1"/>

</beans>
```

Factory Methods

When you configure a class as spring bean, IOC container will instantiate the bean by calling the new operator on it. But in java not all the classes can be instantiated using new operator. For example, Singleton classes cannot be instantiated using new operator, rather you need to call the factory method that has been exposed by the class to create object of that class.

So, for the classes which cannot be created out of new operator, those which has to be created by calling factory methods of the Bean.

What is Factory Method?

The Method which is capable of creating either same class object or different class object is called as Factory method.

Two types of factory methods

1) Static Factory Method

E.g. 1: Class c = Class.forName ("java.util.Date") (return same class Object)

E.g. 2: Console c = System. Console ( ) (Return different class object)

2) Instance Factory method

E.g. 1: String s1 = new String ("hello");
String str = s1.substring (3); (returning same class object)

E.g. 2: StringBuffer sb = new StringBuffer ("hi ");
String s = sb.substring (0, 3); (returning different class object)

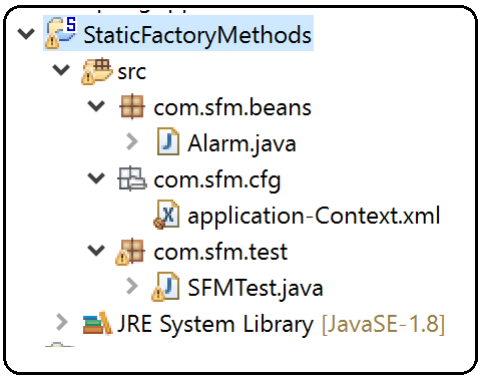
In spring, we can make IOC container to create Spring Bean object in 4 ways.

a. **Using 0-parameterized constructor**
b. **Using Parameterized constructor**
c. **Using Static Factory method**
d. **Using instance Factory method**

Static Factory Method

For example an alarm class needs an Calendar as dependent object, but java.util.Calendar class object cannot instantiated using new operation we need to call the getInstance() static method that is exposed in that class to get object of that class. Following example depicts the same.

Static Factory Method Example



Alaram.java (Target class which requires Calendar obj)

```
package com.sfm.beans;

import java.util.Calendar;

public class Alarm {

	private Calendar time;

	public void setTime(Calendar time) {
		this.time = time;
	}

	public void ring() {
		System.out.println("Ringing : " + time.getTime());
	}
}
```

Alarm.java

Bean Configuration file

```
<bean class="java.util.Calendar" factory-method="getInstance" id="calendar"></bean>

<bean class="com.sfm.beans.Alarm" name="alarm">
	<property name="time" ref="calendar"></property>
</bean>
```

```
public class SFMTest {

	public static void main(String[] args) {
		Resource resource = new ClassPathResource("com/sfm/cfg/application-Context.xml");
		BeanFactory factory = new XmlBeanFactory(resource);

		Alarm a = factory.getBean("a", Alarm.class);
		a.ring();
	}
}
```

SFMTest.java

Instance Factory Method

As said above not all class objects cannot be created out of new operator, few can be constructed by calling factory method on the class, but few objects can be constructed by calling methods on other classes.

If you consider EJB's are any other distributed objects, those cannot be constructed out of new operator or a static method. In order to create an EJB object you need to look up the reference of home object in the JNDI and then you need to call the create () method on it to get the EJB Remote object. This indicates that creation of object involves some sequence of steps or operations

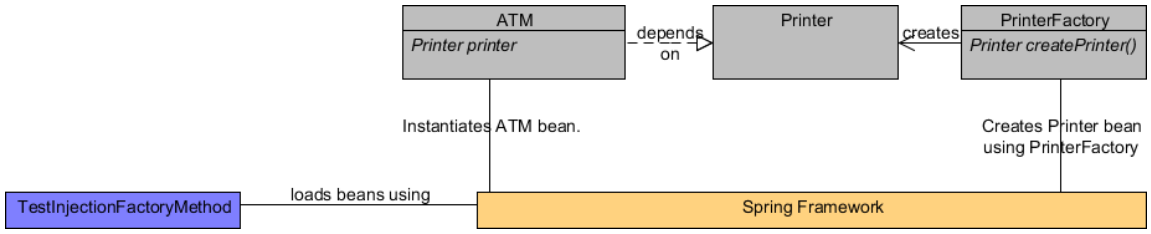
to be performed, so these steps are coded in a separate class methods and generally those classes are called Service locator's, these are the classes who knows how to instantiate the objects of other classes.

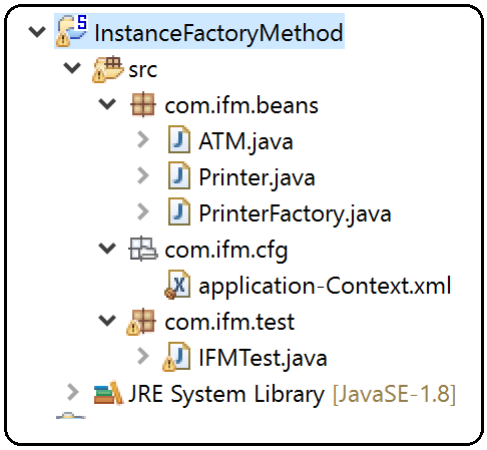
So in order to create such type of objects you need to instantiate the ServiceLocator or helper class on which in-turn you need to call the factory method to create our class object. Instance Factory method injection allows you to instantiate classes object by calling method on other (Service Locator) class. For this you need to declare the (Service Locator) class on which you call the factory method as spring bean, along with this configure the (Target) bean which has to be created by the Service Locator and declare on the <bean> tag factory-bean is your service locator bean and factory• method is your method in service locator class which will creates your target bean, which is shown below.

The sample program is modelled on the ATM (Automated Teller Machine) system. We will create an ATM class and Printer class. The ATM class depends on the Printer class to perform printing related responsibilities like printing the balance information for an account. In this sample, we will assume that creating the instance of Printer involves complex operations and it would be better to use a PrinterFactory class to instantiate the Printer class.

So we will create a PrinterFactory class with static createPrinter() method and returns the Printer type.
We will create Beans.xml and use the factory-method attribute to instantiate Printer class using PrinterFactory.createPrinter() method.

Finally, we will test our setup using TestInjectionFactoryMethod class which will load Spring context and get a reference to ATM class. We will print the balance information for an account using the ATM class and verify that the Printer class has been successfully injected into ATM class using *factory-method.*






```
public class ATM {
        private Printer printer;

        public void setPrinter(Printer printer) {
                this.printer = printer;
        }

        public void printBalanceInformation(String accNo) {
                printer.printBalInfo(accNo);
        }
}
```

```
package com.ifm.beans;

public class Printer {

        public void printBalInfo(String accNo) {
                S.o.p("Printing Bal Info for:" + accNo);
        }
}
```

```
package info.inetsolv.beans;

public class PrinterFactory {

        public Printer createPrinter() {
                return new Printer();
        }
}
```

```
<bean class=" com.ifm.beans.PrinterFactory" id="printerFactory" />

<bean id="printerBean"  factory-method="createPrinter" factory-bean="printerFactory" />

<bean class=" com.ifm.beans.ATM" name="atm">
        <property name="printer" ref=" printerBean"></property>
</bean>
```

```
public class IFMTest {

        public static void main(String[] args) {
                Resource resource = new ClassPathResource("com/ifm/cfg/application-Context.xml");
                BeanFactory factory = new XmlBeanFactory(resource);
                ATM atm = factory.getBean("atm", ATM.class);
                System.out.println(atm);
        }
}
```

IFMTest.java

Method Injection in spring

Method Injection is one of the cool features provided by spring where you can change the behavior of methods in a managed bean. Method Injection would help us to resolve many issues during the collaboration of different scoped managed beans and also reduce the coding effort (especially with bean lookup code)

Spring provides two ways for Method Injection –

- **Lookup method Injection**
- **Arbitrary method replacement**

Lookup method Injection

Method Injection should be used is when a Singleton bean has a dependency on Prototype bean.

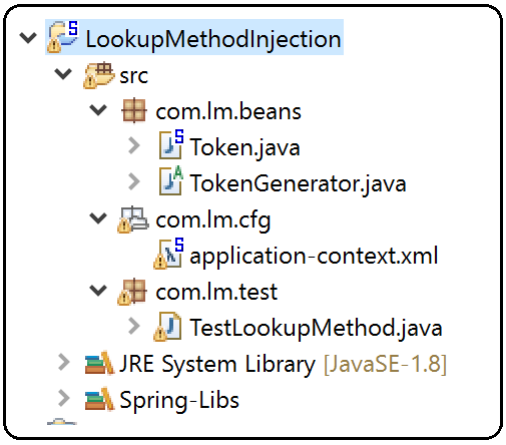
We know that the spring beans can be created with singleton or prototype scope

Singleton: Instantiate only one object

Prototype: Instantiate a new object every time.

Note: In spring when a non-singleton bean is getting injected into a singleton bean the result will be always singleton only.

Let's understand this with below example



TokenGenerator.java (Target class which is Singleton)
Token.java (Dependent which is Prototype)

```
package com.lm.beans;

public class Token {

}
```
Token.java

```
package com.lm.beans;

public abstract class TokenGenerator {
        public abstract Token getToken();
}
```
TokenGenerator.java

```
<bean id="tg" class="com.lm.beans.TokenGenerator">
        <lookup-method name="getToken" bean="tkn" />
</bean>

<bean id="tkn" class="com.lm.beans.Token" scope="prototype" />
```

```
public class TestLookupMethod {

        public static void main(String[] args) {
                Resource resource = new ClassPathResource("com/lm/cfg/application-context.xml");
                BeanFactory factory = new XmlBeanFactory(resource);

                TokenGenerator tg = factory.getBean("tg", TokenGenerator.class);
                System.out.println(tg.getToken().hashCode());
                System.out.println(tg.getToken().hashCode());
        }
}
```
TestLookupMethod.java

Run the program and we can see that getPrototypeBean () returned same object both times.

If we need to have a different prototype bean instance every time than we have to use method injection

How to use Method Injection ?: We need to create an abstract method in Singleton class with return type as PrototypeBean and Prototype bean class will declare one method which returns the current instance (this).

Spring provides below tag that can be used for method injection.

"<lookup method name="prototype_bean" value="method name"/>

This tag specifies that dependency will be resolved by calling method of a bean and since the bean is of type prototype, we will get new instance on every call.

Execute the above program, we can see new instance of PrototypeBean on every call.

Method Injection is the approach in which method name is configured which will be used to resolve the dependency. This approach is primarily used when a singleton bean has a dependency on a prototype bean and a new instance of prototype bean is required from a singleton bean every time

Method Replacement

There are many cases in an application we want to replace the logic inside a method with new logic. To replace the logic we can use multiple ways
Comment the logic inside the existing method and write the new logic as part of it
Write one more class extending the existing class and override the method and write the new logic

If you observe in the above two mechanisms we need to modify the existing source of the application. This pulls lot of questions

l) What if the new logic we are implementing has not certainty (not guaranteed to work)?
2) Sometimes we want to implement the new logic in experimental way.

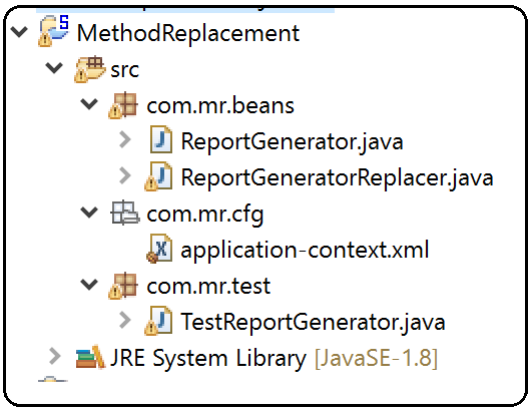
If the new logic we are writing has uncertainty or experimental after writing the logic by modifying the existing source, it needs to be tested and should be deployed. But if after moving into production if the logic is not working then to revert the logic back to original again it needs to modify as we modified it has to be tested and should be redeployed. This incurs the cost equal to writing a new logic, just to revert back to the earlier logic which is a severe loss to the client.

So, if we manage to replace the logic inside the application without modifying the existing code inside the application, then if new logic is failing reverting to the old will not incurs cost as we haven't touched the existing code.

This can be done through the Method replacement. For replacing a method on a bea11 the method should not be final and should not be static. n order to replace the method logic, you need to write new class which implement from MethodReplacer interface and should override reimplement method. The reimplement method takes Object target, java.lang.Method method and Object[] arguments as parameters. These parameters are the values with which you called the original method.

We can replace the logic of a method with new, only when we have information about the original method and its inputs with which we called it. So if we look at the signature of the re-implement, it has the parameters as Object pointing to the actual object with which we called, Method points to the method we are replacing and Object[] args contains the inputs we passed to actual method. Using the inputs now we can implement the logic to replace the actual as shown below.

Method Replacement - Example



ReportGenerator.java (This class method we are trying to replace)

```java
package com.mr.beans;

public class ReportGenerator {

	public void generate(int zipcode) {
		System.out.println("Using Poi api...");
		System.out.println("Report generated...");
	}
}
```

ReportGeneratorReplacer.java (This class is used to Replace target class method)

```java
package com.mr.beans;

import java.lang.reflect.Method;

import org.springframework.beans.factory.support.MethodReplacer;

public class ReportGeneratorReplacer implements MethodReplacer {

	@Override
	public Object reimplement(Object obj, Method m, Object[] args) throws Throwable {
		System.out.println(obj);
		if (m.getName().equals("generate")) {
			System.out.println("Using aspose api");
			System.out.println("Report generated...");
		}
		return obj;
	}
}
```

ReportGeneratorReplacer.java

Spring Bean Configuration (we have configured replace-method)

```
<bean id="rg" class="com.mr.beans.ReportGenerator">
        <replaced-method name="generate" replacer="rgReplacer">
                <arg-type>int</arg-type>
        </replaced-method>
</bean>

<bean id="rgReplacer" class="com.mr.beans.ReportGeneratorReplacer" />
```

```
public class TestReportGenerator {

        public static void main(String[] args) {
                Resource resource = new ClassPathResource("com/mr/cfg/application-context.xml");
                BeanFactory factory = new XmlBeanFactory(resource);
                ReportGenerator rg = factory.getBean("rg", ReportGenerator.class);
                rg.generate(500068);
        }
}
```

TestReportGenerator.java

We need to remember some best practices while working with Method Replacement

- It is good to have if condition to check whether the method we want to replace is right one or not. This makes code more clear and avoids configuration issues.
- For every method we want to replace we need to use a separate replacer class, we should not replace multiple methods with the same replacer even it is technically possible also.

Bean Postprocessor

In spring, we use Bean Lifecycle to perform initialization on a bean. But using init• method, destroy-method or InitalizingBean, DisposableBean we do initialization or destruction process for a specific bean which implement these.
If we have a common initialization process which has to be applied across all the beans in the configuration, Bean lifecycle cannot handle this. We need to use BeanPostProcessor.

The BeanPostProcessor interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic, etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the order property provided the BeanPostProcessor implements the Ordered interface.

The BeanPostProcessors operate on bean (or object) instances, which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An ApplicationContext automatically detects any beans that are defined with the implementation of the BeanPostProcessor interface and registers these beans as postprocessors, to be then called appropriately by the container upon bean creation.

PrintMsg.java (This is our target class and for this class we will perform Post Processing Logic)

```
package info.beans;

import java.util.Date;

public class PrintMsg {

		private String msg;
		private Date lastModifiedDt;

		public void writeMsg() {
		    System.out.println(msg);
		}

		public void init() {
		    System.out.println("PrintMsg : init() ");
		}

		public void destroy() {
		     System.out.println("PrintMsg : destory() ");
		}
}
```

PrintMsg.java

MyBeanPostProcessor.java (This class contains Post Processing logic)

```
import java.util.Date;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class MyBeanProcessor implements BeanPostProcessor {
	@Override
	public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
		System.out.println("MyBeanProcessor : postProcessAfterInitialization() ");
		if (bean instanceof PrintMsg) {
			((PrintMsg) bean).setLastModifiedDt(new Date());
		}
		return bean;
	}
	@Override
	public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
		System.out.println("MyBeanProcessor : postProcessBeforeInitialization() ");
		System.out.println(bean);
		return bean;
	}
}
```

MyBeanProcessor.java

```
<bean class="info.beans.PrintMsg" name="pm"
		init-method="init"
		destroy-method="destroy">
    <property name="msg" value="Welcome to Ashok IT School..!!"></property>
</bean>

<bean name="myProcessor" class="info.inetsolv.beans.MyBeanProcessor"></bean>
```

```
public class MyApp {
	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
		PrintMsg pm = context.getBean("pm", PrintMsg.class);
		System.out.println(pm);
		pm.writeMsg();
	}
}
```

BeanFactoryPostProcessor

If we want to perform some post processing on the BeanFactory after it has been created and before it instantiates the beans then we need to go for BeanFactoryPostProcessor.

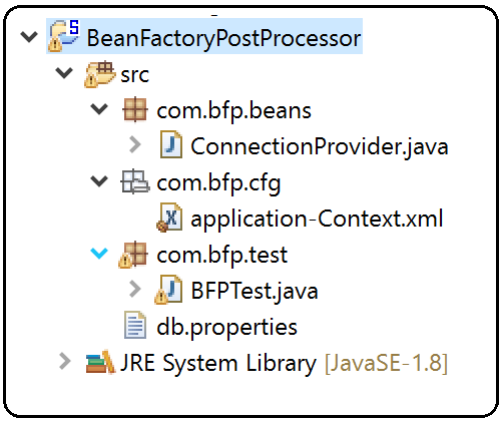
If we want to change the configuration, we can always change it by modifying the physical configuration file. Instead if want to modify the configuration dynamically at runtime within the IOC container metadata then we can use BeanFactoryPostProcessor.

There are many benefits of using BeanFactoryPostProcessor. We can modify the configuration during the deployment time by using some build tools like ant. But if we want to modify the configuration again it demands the rebuild and redeployment.
Instead if we go for BeanFactoryPostProcessor as the configuration is modified in the IOC container metada at runtime.

If we want to perform post initialization after creating the BeanFactory and before creating the beans, we need to use BeanFactoryPostProcessor. It's a one of the kind of extension hooks that spring has provided to the developer to perform customizations on the configuration that is loaded by the Factory.

Spring has provided built-in BeanFactoryPostProcessor' s to perform post processing, based on the type of post processor you use, you will get the respective behavior in your application. One of it is PropertyPlaceHolderConfigurer.
PropertyPlaceHolderConfigurer is a post processor which will reads the messages from the properties file and replaces the ${} place holder's of a bean configuration after the BeanFactory has loaded it.

Example




```
package info.beans;

public class ConnectionProvider {
        private String url;
        private String username;
        private String pwd;
        private String driverClass;

        //setters & toString()
}
```

ConnectionProvider.java

```
url=jdbc:oracle:thin:@localhost:1521/XE
username=IE_APP_DEV
pwd=ieapp_dev
driverClass=oracle.jdbc.driver.OracleDriver
```

dbconfig.properties

```
<bean name="cp" class="info.beans.ConnectionProvider">
        <property name="url" value="${url}"></property>
        <property name="username" value="${username}"></property>
        <property name="pwd" value="${pwd}"></property>
        <property name="driverClass" value="${driverClass}"></property>
</bean>
<bean name="ppch"
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="classpath:dbConfig.properties"></property>
</bean>
```

```java
public class BFPTest {

	public static void main(String[] args) {

		ApplicationContext context =
			new ClassPathXmlApplicationContext("com/bfp/cfg/application-Context.xml");

		ConnectionProvider cp = context.getBean("conProvider", ConnectionProvider.class);
		System.out.println(cp);
	}
}
```

BFPTest.java

Internationalization (I18N)

Internationalization (abbreviated "i18n") is the process of designing software so that it can be adapted (localized) to various languages and regions easily to display the content specific to the locale from which the user is accessing from.

Most commercial websites are targeted for global users. This means that you need to make your application to support different languages. Spring provides built in support for localizing strings used across application.

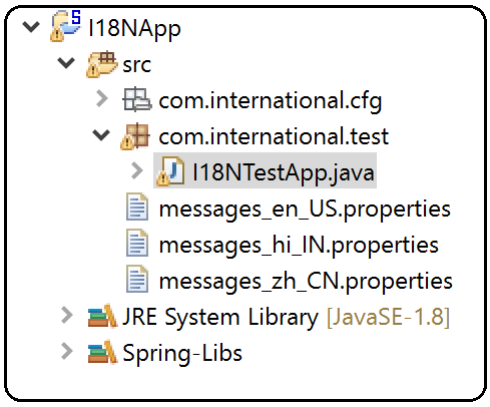
The string message for different locale are stored in separate properties files. These properties files are also called as resource bundles. Spring application context can resolve text messages for target locales by keys. By convention the Java resource bundle files are named as **<filename>_<language_code>_<country_code>.properties .**

Ex : message_en_US.properties

Spring uses the `MessageSource` interface to resolve the resource bundles. The ApplicationContext interface extends MessageSource interface so that all application contexts are able to resolve text messages. An application context delegates the message resolution to a bean with the name `messageSource`. `ResourceBundleMessageSource` is the most common

MessageSource implementation that resolves messages from resource bundles for different locales.

Let's take an example to demonstrate the same



In this example, we will create the Chinese translation of the strings. Create a new file named `message_zh_CN.properties` like below and keep this file in classpath of the project (inside src folder).

```
hello = 你好 {0}
```

messages_zh_CN.properties

```
hello = Hello {0}
```

messages_en_US.properties

```xml
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
      <property name="basename" value="messages" />
</bean>
```

application-Context.xml

Note : This bean id should be “messageSource” only

```java
public class I18NTestApp {

        public static void main(String[] args) {
                ApplicationContext context = new ClassPathXmlApplicationContext(
                                "com/international/cfg/application-Context.xml");
                String msg = context.getMessage("abc", new Object[] { "Ashok" }, Locale.CHINA);
                System.out.println(msg);
        }
}
```

I18NTestApp.java

Spring Annotations Support

Annotation is the code about the code that is metadata about the program itself. In other words information about the code is provided at the source code. Annotations are parsed/processed by compilers, annotation-processing tools and also executed at runtime.

Annotations have been introduced in JDK 1.5. It allows the programmers to specify the information about the code at the source code level itself there are several ways we can use annotations. Few helps in understanding the source code (like documentation assistance) @override is the annotation that will be used when we override a method from base class. This annotation doesn't have any impact on run-time behavior, rather it helps javadoc complier to generate documentation based on it.
Apart from these annotations we have another way in...using annotations'which assi!?,t source code generators to generate so. rce code·using them. If we take example as Web Servites, in order to expose a class as web service, we need to mark the class as @WebService. This annotation would be read by a tool and generates class to expose that class as web service.

Along with this there is another way, where when you mark a class with annotation, the run-time engine will executes the class based on that annotation with which it marked. For example, when you mark a class with @EJB, the class would be exposed as Enterprise Java Bean by the container rather than a simp.le pojo.

By the above we can understand that using annotations, a programmer can specify the various behavioral aspects of your code like documentation, code generation and run• time behavior. This helps in RAPID application development where in instead of specifying the information about your code in xml configuration file, the same can be specified by marking your classes with annotations.
Not,e: - Always your annotation configuration will be overwritten with the xml configuration if provided.

Spring support to annotations is an incremental development effort. Spring 2.0 has a very little support to annotation, when it comes to spring 2.5; it has extended its framework to support various aspects of spring development using annotations. In spring 3.0 it started supporting java config project annotations as well.

The journey to spring annotations has been started in spring 2.0; it has added
@Configuration, @Repository and @Required annotations. Along with this it added support to declarative and annotation based aspectj AOP.

In spring 2.5 it has added few more annotations @Autowired, @Qualifier and @Scope. In addition in introduced stereotype annotations @Component, @Controller and
@Service.

In spring 3.0 few more annotations has been added like @Lazy, @Bean, @DependsOn etc. n addition to spring based metadata annotation support, it has started adoption JSR - 250 Java Config project annotations like @PostConstruct, @PreDestory, @Resource, @Inject and @Named etc.

In all our previous examples, we have injected dependency by configuring XML file but instead of doing this,we can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

You might think what if you have done both i.e.used annotations and XML both.In that case,XML configuration will override annotations because XML configuration will be injected after annotations.

Version	Annotation
Spring 2.0	@Configuration @Required @Repository
Spring 2.5	@Autowired @Qualifier @Scope @Component @Service @Controller
Spring 3.0	@Bean @DependsOn @Lazy @PostConstruct @PreDestory @Resource @Inject @Named

Now annotations based configuration is turned off by default so you have to turn it on by entering <context: annotation-config/> into spring XML file.

==========Sample Beans.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/>

      <!-- beans declaration goes here -->

</beans>
```

==========0==========

Note : (notice the inclusion of the '`context`' namespace in beans.xml)

Spring AOP

Spring

Aspect-Oriented Programming (AOP) complements OOP by providing another way of thinking about program structure. While OO decomposes applications into a hierarchy of objects, AOP decomposes programs into aspects or concerns. This enables modularization of concerns such as transaction management that wouldotherwise cut across multiple objects. (Such concerns are often termed crosscutting concerns.)

One of the key components of Spring is the AOP framework. While the Spring IoC containers (BeanFactory and ApplicationContext) do not depend on AOP, meaning you don't need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

AOP is used in spring:

- To provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative transaction management, which builds on Spring's transaction abstraction.
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

Thus you can view Spring AOP as either an enabling technology that allows spring to provide declarative transaction management without EJB; or use the full power of the Spring AOP framework to implement custom aspects.

In every project there will be two types of logic, one is called primary business logic and other one is helper logic which makes your primary business logic work better. For example calculating the Net salary amount will be primarybusiness logic, but in addition we may write some logging, here logging is called secondary logic because without logging our functionality will be fulfilled but without Net Salary calculation logic we cannot say our application is complete.

Logging will be written in one place of the application or will be written across several?

As logging will be done across various components of the application so it is called cross-cutting logic.

Generalized examples of cross-cutting concerns are Auditing, Security, Logging, Transactions, Profiling and Caching etc.

In a traditional OOPS application if you want to apply a piece of code across various classes, you need to copy paste the code or wrap the code in a method and call at various places. The problem with this approach is your primary business logic is mixed with the cross-cutting logic and at any point of time if you want to remove your cross• cutting concern; you need to modify the source code. So, in order to overcome this, AOP helps you in separating the business logic from cross-cutting concerns.

By the above, we can understand that AOP compliments OOP but never AOP replaces OOP. The key unit of modularity in OOP is class, where as in AOP it is Aspect. As how we have various principles of OOP like abstraction, encapsulation etc., AOP also has principles describing the nature of its .use. Following are the AOP principles.

AOP concepts

Let us begin by defining some central AOP concepts. These terms are not Spring-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring used its own terminology.

- ***Aspect*: A modularization of a concern for which the implementation might otherwise cut across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. Aspects are implemented using Spring as Advisors or interceptors.**
- ***Joinpoint*: Point during the execution of a program, such as a method invocation or a particular exception being thrown. In Spring AOP, a joinpoint is always method invocation. Spring does not use the term joinpoint prominently; joinpoint information is accessible through methods on the MethodInvocation argument passed to interceptors, and is evaluated by implementations ofthe org.springframework.aop.Pointcut interface.**
- ***Advice*: Action taken by the AOP framework at a particular joinpoint. Different types of advice include "around," "before" and "throws" advice. Advice types are discussed below. Many AOP frameworks, including spring, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the joinpoint.**

- ***Pointcut*: A set of joinpoints specifying when an advice should fire. An AOP framework must allow developers to specify pointcuts: for example, using regular expressions.**
- ***Target object*: Object containing the joinpoint. Also referred to as *advised* or *proxied* object.**
- ***AOP proxy*: Object created by the AOP framework, including advice. In Spring, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.**
- ***Weaving*: Assembling aspects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), or at runtime. Spring, like other pure Java AOP frameworks, performs weaving at runtime.**

Different advice types include:

- ***Around advice*: Advice that surrounds a joinpoint such as a method invocation. This is the most powerful kind of advice. Around advices will perform custom behavior before and after the method invocation. They are responsible for choosing whether to proceed to the joinpoint or to shortcut executing by returning their own return value or throwing an exception.**
- ***Before advice*: Advice that executes before a joinpoint, but which does not have the ability to prevent execution flow proceeding to the joinpoint (unless it throws an exception).**
- ***Throws advice*: Advice to be executed if a method throws an exception. Spring provides strongly typed throws advice, so you can write code that catches the exception (and subclasses) you're interested in, without needing to cast from Throwable or Exception.**
- ***After returning advice*: Advice to be executed after a joinpoint completes normally: for example, if a method returns without throwing an exception.**

Around advice is the most general kind of advice. Most interception-based AOP frameworks, such as Nanning Aspects, provide only around advice.

As spring, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you don't need to invoke the proceed() method on the MethodInvocationused for around advice, and hence can't fail to invoke it.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

Spring AOP capabilities and goals

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.

Thus, for example, the Spring Framework's AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring **AOP, such as**

advise very fine-grained objects (such as domain objects typically): AspectJ is the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP.

Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Spring seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible. See the following chapter for a discussion of the Spring AOP APIs.

AOP is not something specific to spring; rather it is a programming technic similar to OOP, so the above principles are generalized principles which can be applied to any framework, supporting AOP programming style.

There are many frameworks in the market which allows you to work with AOP programming few of them are Spring AOP, AspectJ, JAC (Java aspect components), JBossAOP etc. Among which the most popular once are AspectJ and Spring AOP.

At the time spring released AOP, already aspectj AOP has acceptance in the market, and seems to be more powerful than spring AOP. Spring developers instead of comparing the strengths of each framework, they have provided integrations to AspectJ to take the advantage of it, so spring 2.x not only supports AOP it allows us to work with AspectJ AOP as well.

In spring 2.x it provided two ways of working with AspectJ integrations, 1) Declarative pro_gramming model 2) Aspect] Annotation model. With this we are left with three ways of working with spring AOP as follows.

1) Spring AOP API (alliance API) - Programmatic approach
2) Spring Aspect] declarative approach
3) Aspect] Annotation approach

Note: One of the central tenets of the Spring Framework is that of non-invasiveness; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario.
One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ and/or Spring AOP, and you also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style.

Programmatic AOP
We use Spring AOP API's to work with programmatic AOP. As described programmatic aop supports all the types of advices described above. Let us try to work with each advice type in the following section.

Around Advice
Aspect is the cross-cutting concern which has to be applied on a target class; advice represents the action indicating when to execute the aspect on a target class. Based on the advice type we use; the aspect will gets attached to a target class. If it is an around advice, the aspect will be applied before and after, which means around the target class joinpoint execution.

Do You Have Any One Of The Below Question In Your Mind?
1. How to inject your own code before and after a method execution?
2. How to modify a method parameters before the method execution starts?
3. How you modify (or) send your own return value?
4. How to validate method parameters before the method executions?

The answer for all the above questions is "Around advice". The main thing here is the calling method or the target method will not aware of any of these activities.

1) We have control over arguments; it indicates we can modify the arguments before executing the target class method.
2) We can control the target class method execution in the advice. This means we can even skip the target class method execution.
3) We can even modify the return value being returned by the target class method.

In order to work with any advice, first we need to have a target class, let us build the target class first.

```
public class Calculator {

	public int add(int no1, int no2) {
		return no1 + no2;
	}
}
```

calculator.java

Aspect class:

To get the method parameters ProceedingJoinPoint has a method 'getArgs()'. This will give you the method parameters as an array of Object. You can modify these values if you want.jp.proceed() is used to continue the normalmethod execution. You can pass the modified arguments to this method,it will be reflected in the business class. The return value of thisjp.proceed() will be given back to the calling point. You can modify the return value also.

I this example I am passing one parameter to the business method. I am modifying it in my around advice and sending the modified value to the business class. I am getting the return value in my advice and modifying the return value before sending it back to the calling point.

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class CalculatorAdvice implements MethodInterceptor {

	public Object invoke(MethodInvocation mi) throws Throwable {
		// Getting Target class method args
		Object[] args = mi.getArguments();
		Integer a = (Integer) args[0];
		Integer b = (Integer) args[1];

		// Modifying arguments
		args[0] = a + 1;
		args[1] = b + 1;
		// Calling Target class method
		Object returnVal = mi.proceed();

		// Modifying Target class Return value
		returnVal = (Integer) returnVal + 100;
		return returnVal;
	}
}
```

CalculatorAdvice.java

Testing the application

```
import org.springframework.aop.framework.ProxyFactory;

public class Test {
      public static void main(String[] args) {
            ProxyFactory pf = new ProxyFactory();
            pf.setTarget(new Calculator());
            pf.addAdvice(new CalculatorAdvice());
            Calculator proxy = (Calculator) pf.getProxy();
            int result = proxy.add(10, 20);
            System.out.println(result);
      }
}
```

Test.java

Before Advice

In a Before advice always the advice method executes before your target class joinpoint executes. Once the advice method finishes execution the control will be automatically transferred to the target class method and will not be returned back to the advice. So, in a Before Advice we have only one control point, which is only we can access arguments and can modify them. Below .list describes the control points in a Before Advice.

1) **Can access and modify the arguments of the original method.**
2) **Cannot control the target method execution, but we can abort the execution by throwing an exception in the advice method.**
3) **Capturing and modifying the return value is not applicable as the control doesn't return back to the advice method after finishing you target method.**

In order to create a Before Advice, we need to first build a target class. Then we need to create an Aspect class which implements from MethodBeforeAdvice and should override before method. This method has three arguments java.reflect.Method , Object[] args and Object target Object.

Target Class

```
public class SalaryCalculator {
      public void computeSalary(String empId, Double basicPay) {
            double da = basicPay * 20 / 100;
            double hra = basicPay * 15 / 100;
            double pf = basicPay * 12 / 100;
            double incomeTax = basicPay * 10 / 100;
            double gross = basicPay + da + hra;
            double deductions = pf + incomeTax;
            double netSalary = gross - deductions;
            System.out.println("Below are salary Details for Emp : "+empId);
            System.out.print("Net Salary : "+netSalary);
      }
}
```

SalaryCalculator.java

Aspect Class

```
package com.nit.apps;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingAspect implements MethodBeforeAdvice {
      public void before(Method m, Object[] args, Object target) throws Throwable {
            System.out.println("Method Executing : " + m.getName());
            System.out.print("Args : ");
            // Printing Target class method args
            for (Object obj : args) {
                  System.out.print(obj + " ");
            }
            System.out.println("");
            // Modifying target class method args
            String empId = (String) args[0];
            args[0] = empId + "A";
      }
}
```

LoggingAspect.java

Test class for Method Before Advice

```
import org.springframework.aop.framework.ProxyFactory;

public class TestMethodBeforeAdvice {
      public static void main(String[] args) {
            ProxyFactory pFactory = new ProxyFactory();
            pFactory.addAdvice(new Logging());
            pFactory.setTarget(new SalaryCalculator());
            SalaryCalculator sc = (SalaryCalculator) pFactory.getProxy();
            sc.computeSalary("EMP1234", 26000.00);
      }
}
```

TestMethodBeforeAdvice.java

After Returning Advice

In this the advice method will be executed only after the target method finishes execution and before it returns the value to the caller. This indicates that the target method has almost returned the value, but just before returning the value it allows the advice method to see the return value but doesn't allows to modify it. So, in a After returning advice we have the below control points.

1) We can see parameters of the target method, even we modify there is no use, because by the time the advice method is called the target method finished execution, so there is no effect of changing the parameter values.
2) You cannot control the target method execution as the control will enters into advice method only after the target method completes.
3) You can see the return value being returned by the target method, but you cannot modify it, as the target method has almost returned the value. But you can stop/abort the return value by throwing exception in the advice method.

In order to create an after returning advice, after building the target class, you need to write the aspect class implementing the AfterReturningAdvice and should override the method afterReturning, the parameters to this method are Object returnValue (Returned by actual method), java. reflect.Method method (original method), Object[] args (target method arguments), Object targetObject (with which the target method has been called).

```
public class SalaryCalculator {

      public double computeSalary(String empId) {
            return 45000.00;
      }
}
```

SalaryCalculator.java

```
public class LoggingAspect implements AfterReturningAdvice {

      public void afterReturning(Object returnValue, Method method,
                  Object[] args, Object target) throws Throwable {
            String methodName = method.getName();
            System.out.println("Execution completed for::" + methodName);
            System.out.println("returned value from method:" + methodName     + " is:"+ returnValue);
      }
}
```

LoggingAspect.java

```
import org.springframework.aop.framework.ProxyFactory;

public class TestAfterReturningAdvice {
      public static void main(String[] args) {
            ProxyFactory pFactory = new ProxyFactory();
            pFactory.addAdvice(new Logging());
            pFactory.setTarget(new SalaryCalculator());
            SalaryCalculator sc = (SalaryCalculator) pFactory.getProxy();
            sc.computeSalary("EMP1234", 26000.00);
      }
}
```

TestAfterReturningAdvice.java

Throws Advice

Throws advice will be invoked only when the target class method throws exception. The idea behind having a throws advice is to have a centralized error handling mechanism or some kind of central processing whenever a class throws exception.

As said unlike other advices, throws advice will be called only when the target throws exception. The throws advice will not catch the exception rather it has a chance of seeing the exception and do further processing based on the exception, once the throws advice method finishes execution the control will flow through the normal exception handling hierarchy till a catch handler is available to catch it.
Following are the control points a throws advice has.

1) It can see the parameters that are passed to the original method
2) There is no point in controlling the target method execution, as it would be invoked when the target method rises as exception.
3) When a method throws exception, it cannot return a value, so there is nothing like seeing the return value or modifying the return value.

In order to work with throws advice, after building the target class, you need to write a class implementing the ThrowsAdvice interface. The ThrowsAdvice is a marker interface this means; it doesn't define any method in it. You need to write method to monitor the exception raised by the target class, the method signature should public and void with name afterThrowing, taking the argument as exception class type indicating which type of exception you are interested in monitoring (the word monitoring is used· here because we are not catching the exception, but we are just seeing and propagating it to the top hierarchies).

When a target class throws exception spring IOC container will tries to find a method with name afterThrowing in the advice class, having the appropriate argument representing the type of exception class. We can have afterThrowing method with two signatures shown below.

afterThrowing([Method, args, target], sut>classOfThrowable)

In the above signature, Method, args and target is optional and only mandatory parameter is subclassOfThrowable. If a advice class contains afterThrowing method with both the signatures handling the same subclassOfThrowable, the max parameter method will be executed upon throwing the exception by target class.

```java
package com.ta.beans;

public class Thrower {
	public void alwaysThrow() throws NullPointerException {
		throw new NullPointerException("Throwing exception simply");
	}
}
```

Thrower.java

```java
package com.ta.beans;

import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;

public class ExceptionLogger implements ThrowsAdvice {

	public void afterThrowing(Exception npe) {
		System.out.println("Printing Stack Trace from exception : ");
		npe.printStackTrace();
	}
	public void afterThrowing(NullPointerException npe) {
		System.out.println("Printing Stack Trace: ");
		npe.printStackTrace();
	}
	public void afterThrowing(Method method, Object args[], Object target,NullPointerException npe) {
		System.out.println("Exception has been throwed by method : " + method.getName());
		npe.printStackTrace();
	}
}
```

ExceptionLogger.java

```java
import com.ta.beans.ExceptionLogger;
import com.ta.beans.Thrower;

public class ThrowsTest {
	public static void main(String args[]) {
		ProxyFactory factory = new ProxyFactory();
		factory.setTarget(new Thrower());
		factory.addAdvice(new ExceptionLogger());
		Thrower proxy = (Thrower) factory.getProxy();
		proxy.alwaysThrow();
	}
}
```

ThrowsTest.java

Pointcut

In all the above examples, we haven't specified any pointcut while advising the aspect on a target class; this means the advice will be applied on all the joinpoints of the target class. With this all the methods of the target class will be advised, so if you want to skip execution of the advice logic on a specific method of a target class, in the advice logic we can check for the method name on which the advice is being called and based on it we can execute the logic.

The problem with the above approach is even you don't want to execute the advice logic for some methods, still the call to those methods will delegate to advice, this has a performance impact.

In order to overcome this you need to attach a pointcut while performing the weaving, so· that the proxy would be generated based on the pointcut specified and will do some optimizations in generating proxies.

With this if you call a method that is not specified in the pointcut, spring ioc container will makes a direct call to the method rather than calling the advice for it.

Spring

Spring AOP API supports two types of pointcuts as discussed earlier. Those are static and dynamic pointcuts. All the pointcut implementations are derived from org.springframework.aop. Pointcut interface. Spring has provided in-built implementation classes from this interface. Few of them are described below.

Static Pointcut

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient - and best - for most usages. It's possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

1) StaticMethodMatcherPointcut
2) NameMatchMethodPointcut
3) JdkRegexpMethodPointcut

In order to use a StaticMethodMatcherPointcut, we need to write a class extending from StaticMethodMatcherPointcut and needs to override the method matches. The matches method takes arguments as Class and Method as arguments.

Spring while performing the weaving process, it will determines whether to attach an advice on a method of a target class by calling matches method on the pointcut class we supplied, while calling the method it will passes the current class and method it is checking for, if the matches method returns true it will advise that method, otherwise will skip advicing.

```
package com.pointcut.bean;

public class WeatherStation {

        public float getTemparature(int zipCode) {
                return 35.34f;
        }

        public boolean willRaininAnHour(int zipCode) {
                return false;
        }
}
```

WeatherStation.java

```
import org.aopalliance.intercept.MethodInvocation;

public class WeatherStationAdvice implements MethodInterceptor {
      public Object invoke(MethodInvocation invocation) throws Throwable {
            System.out.println("Called invoke");
            Object ret = invocation.proceed();
            ret = ((Float) ret) + 0.5f;
            return ret;
      }
}
```

WeatherStationAdvice.java

```
public class WeatherStationPointcut extends StaticMethodMatcherPointcut {

        public boolean matches(Method method, Class<?> objectClass) {
                if (method.getName().equals("getTemparature") && objectClass == WeatherStation.class) {
                        return true;
                }
                return false;
        }
}
```

WeatherStationPointcut.java

Dynamic Pointcut

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method arguments, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

1) DynamicMethodMatcherPointcut

```
package com.pointcut.bean;

import java.lang.reflect.Method;
import org.springframework.aop.support.DynamicMethodMatcherPointcut;

public class WeatherStationDynamicPointcut extends DynamicMethodMatcherPointcut {

        public boolean matches(Method method, Class<?> objectClass, Object[] args) {
                Integer zipCode = 0;
                if (method.getName().equals("getTemparature") && objectClass == WeatherStation.class) {
                        zipCode = (Integer) args[0];
                        if (zipCode >= 10 && zipCode <= 100) {
                                return true;
                        }
                }
                return false;
        }
}
```

WeatherStationDynamicPointcut.java

Pointcut Tester class

```
public class PointCutTest {
        public static void main(String[] args) {
                ProxyFactory pf = new ProxyFactory();
                pf.setTarget(new WeatherStation());
                /*
                 * pf.addAdvisor(new DefaultPointcutAdvisor(new
                 * WeatherStationPointcut(), new WeatherStationAdvice()));
                 */
                /*
                 * pf.addAdvisor(new DefaultPointcutAdvisor(new
                 * WeatherStationDynamicPointcut(), new WeatherStationAdvice()));
                 */
                NameMatchMethodPointcut nmp = new NameMatchMethodPointcut();
                nmp.addMethodName("getTemparature");
                pf.addAdvisor(new DefaultPointcutAdvisor(nmp,
                                new WeatherStationAdvice()));
                WeatherStation proxy = (WeatherStation) pf.getProxy();
                System.out.println("Tempature : " + proxy.getTemparature(353));
                System.out.println("Will rain : " + proxy.willRaininAnHour(353));
        }
}
```

PointCutTest.java

Schema-based AOP support

Spring 2.x has added support to declarative AspectJ AOP using the new "aop" namespace tags. The main problem with programmatic approach is your application code will tightly couple with spring, so that you cannot detach from spring If you use declarative approach, your advice or aspect classes are still pojo' s, you don't need to implement or extend from any spring specific interface or class. Your advice classes uses AspectJ AOP API classes, in order to declare the pojo as aspect you need to declare it in the configuration file rather than implementing pre-defined interfaces using programmatic approach. The declarative approach also supports all four types of advices and static pointcut.

```
<bean id="calc" class="com.ba.beans.Calculator" />

<bean id="logger" class="com.ba.beans.LogAdvice" />

<aop:config>
        <aop:pointcut expression="execution(* com.ba.beans.*.*(..))"    id="pc" />
        <aop:aspect ref="logger">
                <aop:before method="log" pointcut-ref="pc" />
        </aop:aspect>
</aop:config>
```

As per above configuration,

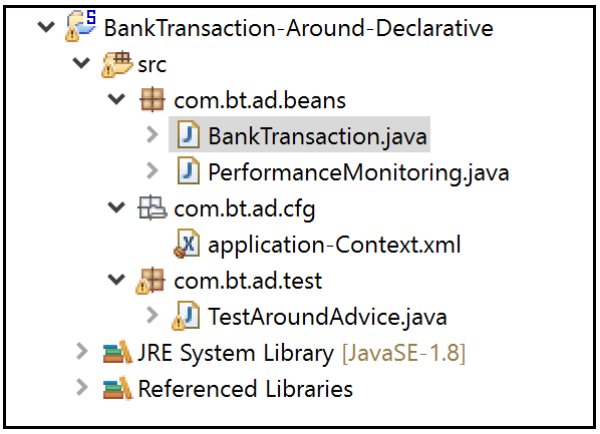
Calculator is a target class

LogAdvice is an aspect class

pc is id of pointcut expression

<aop:before … /> represents it is before advice

Around Advice-Declarative Approach




```java
package com.bt.ad.beans;

public class BankTransaction {
        public void deposit(double amount, long accNo) {
                System.out.println("Amount credited...");
        }

        public void withdraw(double amount, long accno) {
                System.out.println("Amount debited...");
        }
}
```

BankTransaction.java

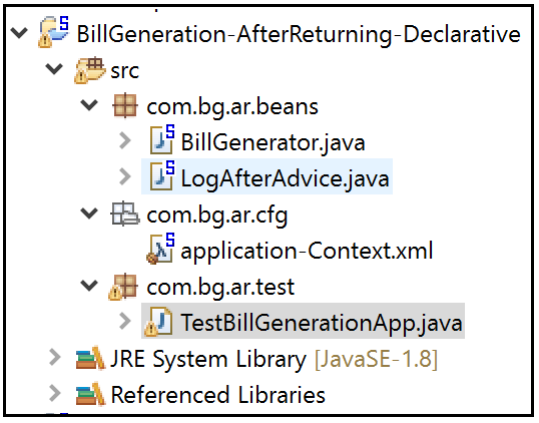
```java
package com.bt.ad.beans;

import org.aspectj.lang.ProceedingJoinPoint;

public class PerformanceMonitoring {
	public Object monitor(ProceedingJoinPoint pjp) {
		long start = System.currentTimeMillis();
		Object retVal = null;
		try {
			retVal = pjp.proceed();
		} catch (Throwable e) {
			e.printStackTrace();
		}
		long end = System.currentTimeMillis();
		String methodName = pjp.getSignature().getName();
		Sop("Time taken for : " + methodName + " : " + (end - start));
		return retVal;
	}

}
```

PerformanceMonitoring.java

```xml
<bean id="bt" class="com.bt.ad.beans.BankTransaction" />
<bean id="pmonitor" class="com.bt.ad.beans.PerformanceMonitoring" />

<aop:config>
	<aop:pointcut expression="execution(* com.bt.ad.beans.*.*(..))" id="pc1" />
	<aop:pointcut expression="execution(* com.bt.ad.beans.*.*(..))" id="pc2" />
	<aop:aspect ref="pmonitor">
		<aop:around method="monitor" pointcut-ref="pc1" />
	</aop:aspect>
</aop:config>
```

```java
public class TestAroundAdvice {

	public static void main(String[] args) {
		ApplicationContext context = new
ClassPathXmlApplicationContext("com/bt/ad/cfg/application-Context.xml");
		BankTransaction bt = context.getBean("bt", BankTransaction.class);
		bt.deposit(50000, 8686868);
		bt.withdraw(6700, 5757586);
	}
}
```

AfterReturning-Declarative:



Spring

```java
package com.bg.ar.beans;

public class BillGenerator {

	public double generateBill(long phno) {
		// business logic
		return 15000.90;
	}
}
```

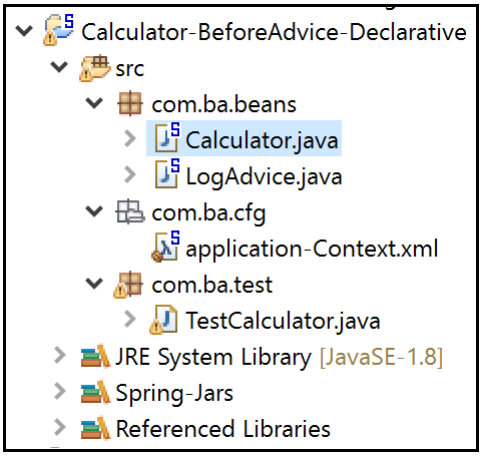
BillGenerator.java

```java
package com.bg.ar.beans;

import org.aspectj.lang.JoinPoint;

public class LogAfterAdvice {
	public void log(JoinPoint jp, double retval) {
	S.o.p("Method execution completed for : " + jp.getSignature().getName());
		// If bill amt > 5000 generate coupon
		if (retval > 5000.00) {
			// Generating coupon for Mobile number
			Object[] args = jp.getArgs();
			System.out.println("Coupon generated for : " + args[0]);
		}
	}
}
```

```xml
<bean id="bg" class="com.bg.ar.beans.BillGenerator" />
<bean id="logger" class="com.bg.ar.beans.LogAfterAdvice" />

<aop:config>
	<aop:pointcut expression="execution(* com.bg.ar.beans.*.*(..))" id="pc1" />
	<aop:aspect ref="logger">
		<aop:after-returning method="log" pointcut-ref="pc1" returning="retval" />
	</aop:aspect>
</aop:config>
```

```java
public class TestBillGenerationApp {

	public static void main(String[] args) {
		ApplicationContext context = new
ClassPathXmlApplicationContext("com/bg/ar/cfg/application-Context.xml");
		BillGenerator bg = context.getBean("bg", BillGenerator.class);
		double billAmt = bg.generateBill(8989797);
		System.out.println("Bill amount : " + billAmt);
	}
}
```

Before Advice-Declarative:



```java
package com.ba.beans;

public class Calculator {

	public int add(int i, int j) {
		return i + j;
	}

}
```

Calculator.java

```java
package com.ba.beans;

import java.util.Arrays;

import org.aspectj.lang.JoinPoint;

public class LogAdvice {

	public void log(JoinPoint jp) {
		System.out.println("Execution started for : " + jp.getSignature());
		System.out.println("Arguments : " + Arrays.toString(jp.getArgs()));
	}

}
```

LogAdvice.java

```xml
<bean id="calc" class="com.ba.beans.Calculator" />
<bean id="logger" class="com.ba.beans.LogAdvice" />
<aop:config>
	<aop:pointcut expression="execution(* com.ba.beans.*.*(..))" id="pc" />
	<aop:aspect ref="logger">
		<aop:before method="Log" pointcut-ref="pc" />
	</aop:aspect>
</aop:config>
```

```java
public class TestCalculator {

	public static void main(String[] args) {
		ApplicationContext context = new
ClassPathXmlApplicationContext("com/ba/cfg/application-Context.xml");
		Calculator c = context.getBean("calc", Calculator.class);
		System.out.println("Sum : " + c.add(10, 20));
	}
}
```

TestCalculator.java

Throws Advice – Declarative - Approach

```java
package com.ta.beans;

public class DepartmentDao {

	public void insert() {
		System.out.println("insert () : started");
		throw new NullPointerException();
	}
}
```

DepartmentDao.java

```
package com.ta.aspects;

public class ExceptionLogger {

	public void afterThrowing(Exception ex) {
		System.out.println("Exception occured: super ... " + ex);
		System.out.println("Sending Email...!!");
	}

	/*
	 public void afterThrowing(NullPointerException e) {
	   System.out.println("Exception occured : sub ... " + e);
	 }

	 public void afterThrowing(Method m, Object[] args, Object target, NullPointerException ne) {
		System.out.println("Exception occured in:" + m.getName());
		System.out.println("parameters : " + Arrays.toString(args));
		System.out.println("Exception details : " + ne);
	 }
	 */
}
```

ExceptionLogger.java

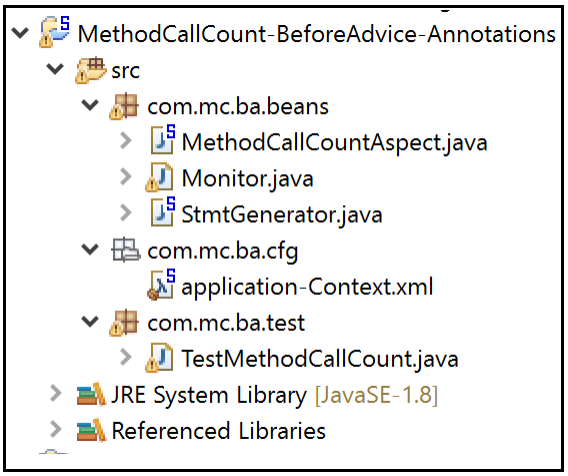
```
<bean id="dao" class="com.ta.beans.DepartmentDao" />
<bean id="elog" class="com.ta.aspects.ExceptionLogger" />

<aop:config>
	<aop:aspect ref="elog">
		<aop:after-throwing
		  throwing="ex"
		  method="afterThrowing"
		  pointcut="execution(* com.ta.beans.DepartmentDao.*(..))"/>
	</aop:aspect>
</aop:config>
```

Aop-Beans.xml

```
public class TestDeptDao {

    public static void main(String[] args) {
	ApplicationContext ctx =
		new ClassPathXmlApplicationContext("com/ta/cfg/Aop-Beans.xml");
	 DepartmentDao dao = ctx.getBean("dao", DepartmentDao.class);
	    dao.insert();
	}
}
```

TestDeptDao.java

AOP using Annotations:



```java
package com.mc.ba.beans;
@Aspect
public class MethodCallCountAspect {

	@Pointcut("execution(* com.mc.ba.beans.*.*(..))")
	public void methodPointCut() {
}
	@Before("methodPointCut()")
	public void monitor(JoinPoint jp) {
		Signature sig = jp.getSignature();
		String methodName = sig.getName();
		Monitor.increment(methodName);
	}
	@Around("methodPointCut()")
	public Object log(ProceedingJoinPoint pjp) {
		Object retVal = null;
		try {
			System.out.println("before");
			retVal = pjp.proceed();
		} catch (Throwable e) {
			e.printStackTrace();
		}
		System.out.println("after");
		return retVal;
	}
}
```

MethodCallCountAspect.java

```java
package com.mc.ba.beans;

import java.util.HashMap;
import java.util.Map;

public class Monitor {

	private static Map<String, Integer> map = new HashMap();

	public static void increment(String methodName) {
		if (map.containsKey(methodName)) {
			Integer cnt = map.get(methodName);
			cnt++;
			map.put(methodName, cnt);
		} else {
			map.put(methodName, 1);
		}
	}

	public static int getAccessCount(String methodName) {
		return map.get(methodName);
	}

}
```

Monitor.java

```xml
<aop:aspectj-autoproxy />

<bean id="sg" class="com.mc.ba.beans.StmtGenerator" />
<bean id="mc" class="com.mc.ba.beans.MethodCallCountAspect" />
```

```
package com.mc.ba.test;

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

import com.mc.ba.beans.Monitor;
import com.mc.ba.beans.StmtGenerator;

public class TestMethodCallCount {

	public static void main(String[] args) {

		ApplicationContext context = new ClassPathXmlApplicationContext("com/mc/ba/cfg/application-Context.xml");
		StmtGenerator sg = context.getBean("sg", StmtGenerator.class);

		sg.generateStmt();
		sg.generateStmt();
		sg.generateStmt();

		System.out.println("Count : " + Monitor.getAccessCount("generateStmt"));

	}

}
```

TestMethodCallCount.java

Spring DAO

Spring

Introduction

The DAO (Data Access Object) support in Spring is primarily aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way. This allows you to switch between them fairly easily and it also allows you to code without worrying about catching exceptions that are specific to each technology.

Consistent Exception Hierarchy

Spring provides a convenient translation from technology specific exceptions like SQLException to its own exception hierarchy with the DataAccessException as the root exception. These exceptions wrap the original exception so there is never any risk that you would lose any information as to what might have gone wrong.

In addition to JDBC exceptions, Spring can also wrap Hibernate exceptions, converting them from proprietary, checked exceptions, to a set of abstracted runtime exceptions. The same is true for JDO exceptions. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. As we mentioned above, JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

The above is true for the Template versions of the ORM access framework. If you use the Interceptor based classes then the application must care about handling Hibernate Exceptions and JDOExceptions itself, preferably via delegating to SessionFactoryUtils' convertHibernateAccessException or convertJdoAccessException methods respectively. These methods converts the exceptions to ones that are compatible with the org.springframework.dao exception hierarchy. As JDOExceptions are unchecked, they can simply get thrown too, sacrificing generic DAO abstraction in terms of exceptions though.

Consistent Abstract Classes for DAO Support

To make it easier to work with a variety of data access technologies like JDBC, JDO and Hibernate in a consistent way, Spring provides a set of abstract DAO classes that you can extend. These abstract classes have methods for setting the data source and any other configuration settings that are specific to the technology you currently are using.

Dao Support classes:

JdbcDaoSupport - super class for JDBC data access objects. Requires a DataSource to be set, providing a JdbcTemplate based on it to subclasses.

HibernateDaoSupport - super class for Hibernate data access objects. Requires a SessionFactory to be set, providing a HibernateTemplate based on it to subclasses. Can alternatively be initialized directly via a HibernateTemplate, to reuse the latter's settings like SessionFactory, flush mode, exception translator, etc.

JdoDaoSupport - super class for JDO data access objects. Requires a PersistenceManagerFactory to be set, providing a JdoTemplate based on it to subclasses.

Data Access using JDBC

Spring JdbcTemplate is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

Problems of JDBC API

The problems of JDBC API are as follows:

- We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
- We need to perform exception handling code on the database logic.

- We need to handle transaction.
- Repetition of all these codes from one to another database logic is a time consuming task.

Advantage of Spring JdbcTemplate

Spring JdbcTemplate eliminates all the above-mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

Spring jdbc provides extraction over plain jdbc by providing various templates such jdbctemplate, namedparameter template and better exception handling compared to plain jdbc. It makes mapping easier between relational database and java beans with the help of different mapper classes.

Who Does what in Spring Jdbc?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

All the classes in Spring JDBC are divided into four separate packages. They are core, datasource, object, and support.

1. **The org.springframework.jdbc.core package** contains the JdbcTemplate class and its various callback interfaces, plus a variety of related classes.
2. **The org.springframework.jdbc.datasource package** contains a utility class for easy DataSource access, and various simple DataSource implementations that can be used for testing and running unmodified JDBC code outside of a J2EE container. The utility class provides static methods to obtain connections from JNDI and to close connections if necessary. It has support for thread-bound connections, e.g. for use with DataSourceTransactionManager.
3. **Next, the org.springframework.jdbc.object package** contains classes that represent RDBMS queries, updates, and stored procedures as thread safe, reusable objects. This approach is modeled by JDO, although of course objects returned by queries are “disconnected” from the database. This higher level of JDBC abstraction depends on the lower-level abstraction in the org.springframework.jdbc.core package.
4. **Finally the org.springframework.jdbc.support package** is where you find the SQLException translation functionality and some utility classes. Exceptions thrown during JDBC processing are translated to exceptions defined in the org.springframework.dao package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated exceptions are unchecked giving you the option of catching the exceptions that you can recover from while allowing other exceptions to be propagated to the caller.

Spring Jdbc Approaches

Spring framework provides following approaches for JDBC database access:

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert and SimpleJdbcCall
- Mapping Queries to Sub Classes

Configuring DataSource

In order to work with data from a database, we need to obtain a connection to the database. The way Spring does this is through a DataSource. A DataSource is part of the JDBC specification and can be seen as a generalized connection factory. It allows a container or a framework to hide connection pooling and transaction management issues from the application code. As a developer, you don't need to know any details about how to connect to the database, that is the responsibility for the administrator that sets up the datasource. You will most likely have to fulfill both roles while you are developing and testing you code though, but you will not necessarily have to know how the production data source is configured.

When using Spring's JDBC layer, you can either obtain a data source from JNDI or you can configure your own, using an implementation that is provided in the Spring distribution. The latter comes in handy for unit testing outside of a web container. We will use the DriverManagerDataSource implementation for this section but there are several additional implementations that will be covered later on. The DriverManagerDataSource works the same way that you probably are used to work when you obtain a JDBC connection. You have to specify the fully qualified class name of the JDBC driver that you are using so that the DriverManager can load the driver class. Then you have to provide a url that varies between JDBC drivers. You have to consult the documentation for your driver for the correct value to use here. Finally you must provide a username and a password that will be used to connect to the database. Here is an example of how to configure a DriverManagerDataSource:

```
@Configuration
@ComponentScan("com.ashok.jdbc")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUsername("guest_user");
        dataSource.setPassword("guest_password");

        return dataSource;
    }
}
```

Alternatively, you can also make good use of an embedded database for development or testing – here is a quick configuration that creates an instance of HSQL embedded database and pre-populates it with a simple SQL scripts:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("classpath:jdbc/schema.sql")
        .addScript("classpath:jdbc/test-data.sql").build();
}
```

Finally – the same can of course be done using XML configuring for the DataSource:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/springjdbc"/>
    <property name="username" value="guest_user"/>
    <property name="password" value="guest_password"/>
</bean>
```

JdbcTemplate

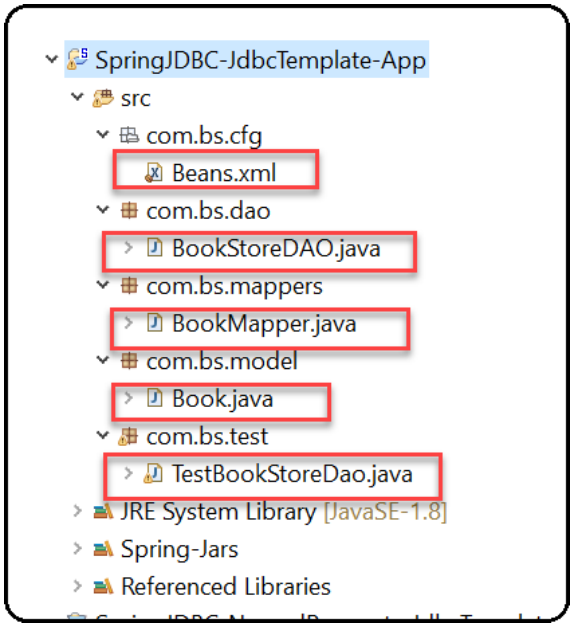
This is the central class in the JDBC core package. It simplifies the use of JDBC since it handles the creation and release of resources. This helps to avoid common errors like forgetting to always close the connection. It executes the core JDBC workflow like statement creation and execution, leaving application code to provide SQL and extract results. This class executes SQL queries, update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Example: -

Step-1: Create Database table like below

```
CREATE TABLE BOOK_STORE
(
        BOOK_ID NUMBER(10),
        BOOK_NAME VARCHAR2(50),
        BOOK_PRICE NUMBER(10,2),
        BOOK_ISBN VARCHAR2(10),
        AUTHOR_NAME VARCHAR2(20),
        PRIMARY KEY(BOOK_ID)
)
```

Step-2: Create Spring Application and add Spring Jars + ojdbc14.jar to project build path



Step-3: Create Spring Bean Configuration with DataSource, JdbcTemplate and Dao class

Beans.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="ds"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@localhost:1521/XE" />
        <property name="username" value="SYSTEM" />
        <property name="password" value="admin" />
    </bean>

    <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="ds" />
    </bean>

    <bean id="dao" class="com.bs.dao.BookStoreDAO">
        <property name="jdbcTemplate" ref="jt" />
    </bean>
</beans>
```

1 2 3

Beans.xml

1 – Configuring Data Source bean and injecting Database details through SI

2 – Configuring JdbcTemplate as Spring Bean and injecting DataSource into JdbcTemplate through SI

3 – Configuring BookStoreDao as Spring Bean and injecting JdbcTemplate through SI.

Step-4: Create a Model class to set/get the data like below

```java
public class Book {

	private Integer bookId;
	private String bookName;
	private String autorName;
	private String isbn;
	private Double price;

	//setters & getters
	//toString()
}
```

Book.java

Step-5: Create Dao class to perform DB operations

```java
public class BookStoreDAO {

	private static final String FIND_BOOK_BY_ID = "SELECT * FROM BOOK_STORE WHERE BOOK_ID=?";
	private static final String INSERT_BOOK = "INSERT INTO BOOK_STORE VALUES (?,?,?,?,?)";
	private static final String FIND_ALL_BOOKS = "SELECT * FROM BOOK_STORE";

	private JdbcTemplate jdbcTemplate;

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	//CURD operation method goes here

}
```

BookStoreDAO.java

```java
/**
 * This method is used to insert a record
 *
 * @param bid
 * @param aname
 * @param bname
 * @param price
 * @param isbn
 */
public void insertBook(int bid, String aname, String bname, double price, String isbn) {
	int no = jdbcTemplate.update(INSERT_BOOK, bid, aname, bname, price, isbn);
	if (no > 0) {
		System.out.println("Inserted....");
	} else {
		System.out.println("failed....");
	}
}
```

```java
/**
 * This method is used to insert a record using PSTMT
 * @param bid
 * @param aname
 * @param bname
 * @param price
 * @param isbn
 */
public void insertBookWithPstmt(int bid, String aname, String bname, double price, String isbn) {
	PreparedStatementCallback<Boolean> pstmtCallBk = new PreparedStatementCallback<Boolean>() {
		@Override
		public Boolean doInPreparedStatement(PreparedStatement pstmt) throws SQLException, DataAccessException {
			pstmt.setInt(1, bid);
			pstmt.setString(2, aname);
			pstmt.setString(3, bname);
			pstmt.setDouble(4, price);
			pstmt.setString(5, isbn);
			return pstmt.execute();
		}
	};
	jdbcTemplate.execute(INSERT_BOOK, pstmtCallBk);
}
```

```java
/**
 * This method is used to insert multiple records using batch operation
 *
 * @param books
 */
public void insertBooksUsingBatch(final List<Book> books) {
	jdbcTemplate.batchUpdate(INSERT_BOOK, new BatchPreparedStatementSetter() {

		@Override
		public void setValues(PreparedStatement pstmt, int i) throws SQLException {
			Book b = books.get(i);
			pstmt.setInt(1, b.getBookId());
			pstmt.setString(2, b.getAutorName());
			pstmt.setString(3, b.getBookName());
			pstmt.setDouble(4, b.getPrice());
			pstmt.setString(5, b.getIsbn());
		}

		@Override
		public int getBatchSize() {
			return books.size();
		}
	});
}
```

Below is the RowMapper class which is used to map Query Results to Java object on row basis.

For every row in ResultSet object , mapRow method will be called internally to map the data to java object

BookMapper.java

```java
package com.bs.mappers;

import java.sql.ResultSet;

/**
 * This class is used to Map every ResultSet Row into Book object
 *
 * @author Ashok
 *
 */
public class BookMapper implements RowMapper<Book> {

    @Override
    public Book mapRow(ResultSet rs, int rowIndex) throws SQLException {
        // Creating Book Object
        Book b = new Book();

        // Setting ResultSet data to Book obj
        b.setBookId(rs.getInt(1));
        b.setAutorName(rs.getString(2));
        b.setBookName(rs.getString(3));
        b.setPrice(rs.getDouble(4));
        b.setIsbn(rs.getString(5));

        // return Book obj
        return b;
    }
}
```

```java
/**
 * This method is used to Retrieve a record
 *
 * @param bookId
 * @return
 */
public Book findBookById(int bookId) {
      return jdbcTemplate.queryForObject(FIND_BOOK_BY_ID, new Object[] { bookId }, new BookMapper());
}
```

```java
/**
 * This method is used to retrieve all records
 *
 * @return
 */
public List<Book> findAllBooks() {
      return jdbcTemplate.query(FIND_ALL_BOOKS, new BookMapper());
}
```

Step-6: Create Test class to test Dao class methods

```java
public class TestBookStoreDao {

	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext("com/bs/cfg/Beans.xml");
		BookStoreDAO dao = context.getBean("dao", BookStoreDAO.class);

		// Inserting record
		dao.insertBook(102, "Rod Johnson", "Spring Boot", 800.00, "ISBN002");

		// Inserting record with Prepared-Statment
		dao.insertBookWithPstmt(103, "Rod Johnson", "Spring Cloud", 400.00, "ISBN003");

		// Retrieveing book using id
		Book book = dao.findBookById(101);
		System.out.println(book);

		// Retriving all books
		List<Book> books = dao.findAllBooks();
		if (!books.isEmpty()) {
			for (Book b : books) {
				System.out.println(b);
			}
		}
		// Testing batch
		List<Book> booksList = new ArrayList<Book>();
		booksList.add(new Book(105, "Spring", "Rod", "ISN001", 450.00));
		booksList.add(new Book(106, "Webservices", "Ashok", "ISBN002", 550.00));
		dao.insertBooksUsingBatch(booksList);
	}
}
```

TestBookStoreDao.java

What is RowMapper?

An interface used by JdbcTemplate for mapping rows of a ResultSet on a per-row basis. Implementations of this interface perform the actual work of mapping each row to a result object, but don't need to worry about exception handling. SQLExceptions will be caught and handled by the calling JdbcTemplate.

Mapping Query Results to java Object Using RowMapper

To map query results to Java object we will create a class by implementing RowMapper interface like below.

```java
public class EmployeeRowMapper implements RowMapper<Employee> {
  @Override
  public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
    Employee employee = new Employee();

    employee.setId(rs.getInt("ID"));
    employee.setFirstName(rs.getString("FIRST_NAME"));
    employee.setLastName(rs.getString("LAST_NAME"));
    employee.setAddress(rs.getString("ADDRESS"));

    return employee;
  }
}
```

EmployeeRowMapper.java

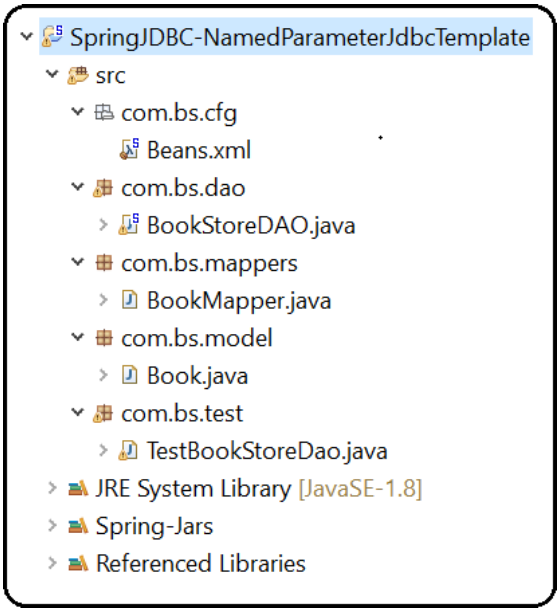
We can now pass the row mapper to the query API and get fully populated Java objects back:

```
String query = "SELECT * FROM EMPLOYEE WHERE ID = ?";

List<Employee> employees =
jdbcTemplate.queryForObject(query, new Object[] { id }, new EmployeeRowMapper());
```

NamedParameterJdbcTemplate: NamedParameterJdbcTemplate wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.

Example:

Step-1: Create Spring Application and add Spring Jars + ojdbc14.jar to project build path



Step-2: Create Spring Bean Configuration with DataSource, NamedParameterJdbcTemplate and Dao class

```
<bean id="ds"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
      <property name="url" value="jdbc:oracle:thin:@localhost:1521/XE" />
      <property name="username" value="SYSTEM" />
      <property name="password" value="admin" />
</bean>

<bean id="jt"
      class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
      <constructor-arg name="dataSource" ref="ds" />
</bean>

<bean id="dao" class="com.bs.dao.BookStoreDAO">
      <property name="jdbcTemplate" ref="jt" />
</bean>
```

Step-3: Create a Model class to set/get the data like below

```java
public class Book {

	private Integer bookId;
	private String bookName;
	private String autorName;
	private String isbn;
	private Double price;

	//setters & getters
	//toString()
}
```

Book.java

Step-4: Create Dao class to perform DB operations

```java
public class BookStoreDAO {

	private NamedParameterJdbcTemplate jdbcTemplate;

	// Injecting NPJT into DAO class through setter injection
	public void setJdbcTemplate(NamedParameterJdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	/**
	 * This method is used to Insert record with given values with Named
	 * positional parameters
	 */
	public void insertBook(int bid, String aname, String bname, double price, String isbn) {
		Map<String, Object> paramMap = new HashMap<String, Object>();
		paramMap.put("bid", bid);
		paramMap.put("aname", aname);
		paramMap.put("bname", bname);
		paramMap.put("bprice", price);
		paramMap.put("bisbn", isbn);
		jdbcTemplate.update(INSERT_BOOK, paramMap);
	}
}
```

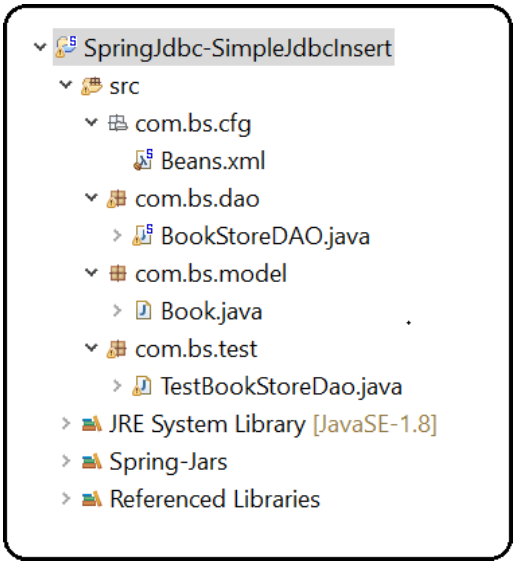
```java
/**
 * This method is used to Retrieve Book obj using Book ID
 */
public Book findById(int bid) {
	Map<String, Object> param = new HashMap();
	param.put("BID", bid);
	return jdbcTemplate.queryForObject(FIND_BOOK_BY_ID, param, new BookMapper());
}

/**
 * This method is used to update book price using BookId
 */
public int updatePriceById(int bid, double price) {
	Map<String, Object> params = new HashMap();
	params.put("PRICE", price);
	params.put("BID", bid);
	return jdbcTemplate.update(UPDATE_PRICE_BY_ID, params);
}
```

Step-5: Create Test class to test Dao class methods

SimpleJdbcInsert: A SimpleJdbcInsert 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. All you need to provide is the name of the table and a Map containing the column names and the column values.

Example:

Step-1: Create the Java project and add Spring Jars + Ojdbc14.jar file to project build path.



Step-2: Create Spring Bean Configuration file with below details

```
<bean id="ds"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
      <property name="url" value="jdbc:oracle:thin:@Localhost:1521/XE" />
      <property name="username" value="SYSTEM" />
      <property name="password" value="admin" />
</bean>

<bean id="sji" class="org.springframework.jdbc.core.simple.SimpleJdbcInsert">
      <constructor-arg name="dataSource" ref="ds" />
</bean>

<bean id="dao" class="com.bs.dao.BookStoreDAO">
      <property name="jdbcInsert" ref="sji" />
</bean>
```

Step-3: Create a model class to set/get the data

```
public class Book {

      private Integer bookId;
      private String bookName;
      private String autorName;
      private String isbn;
      private Double price;

      //setters & getters
      //toString()
}
```

Book.java

Step-4: Create Dao class to perform Insert operation

```
public class BookStoreDAO {

    private SimpleJdbcInsert jdbcInsert;

    /**
     * Injecting SimpleJdbcInsert thru setter injection
     */
    public void setJdbcInsert(SimpleJdbcInsert jdbcInsert) {
        this.jdbcInsert = jdbcInsert;
    }

    /**
     * This method is used to insert a record using table meta-data
     */
    public int insertBook(Book b) {

        // setting table name
        jdbcInsert.setTableName("BOOK_STORE");

        Map<String, Object> params = new HashMap();

        // Setting columns data
        params.put("BOOK_ID", b.getBookId());
        params.put("AUTHOR_NAME", b.getAutorName());
        params.put("BOOK_NAME", b.getBookName());
        params.put("BOOK_PRICE", b.getPrice());
        params.put("BOOK_ISBN", b.getIsbn());

        // executing
        return jdbcInsert.execute(params);
    }
}
```

Step-5: Create Test class to test Dao class methods

```
public class TestBookStoreDao {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext("com/bs/cfg/Beans.xml");
        BookStoreDAO dao = context.getBean("dao", BookStoreDAO.class);

        // setting data into Book obj
        Book b = new Book();
        b.setBookId(101);
        b.setBookName("Spring");
        b.setAutorName("Rod Johnson");
        b.setPrice(450.50);
        b.setIsbn("ISBN001");

        // calling dao method
        dao.insertBook(b);
    }
}
```

SimpleJdbcCall: A SimpleJdbcCall is a multi-threaded, reusable object representing a call to a stored procedure or a stored function. It provides meta-data processing to simplify the code needed to access basic stored procedures/functions. All you need to provide is the name of the procedure/function and a Map containing the parameters when you execute the call. The names of the supplied parameters will be matched up with in and out parameters declared when the stored procedure was created.

The meta-data processing is based on the DatabaseMetaData provided by the JDBC driver. Since we rely on the JDBC driver, this "auto-detection" can only be used for databases that are known to provide accurate meta-data. These currently include Derby, MySQL, Microsoft SQL Server, Oracle, DB2, Sybase and PostgreSQL. For any other databases you are required to declare all parameters explicitly. You can of course declare all parameters explicitly even if the database provides the necessary meta-data. In that case your declared parameters will take precedence. You can also turn off any meta-data processing if you want to use parameter names that do not match what is declared during the stored procedure compilation.

Creating Stored Procedure in Database

```
CREATE OR REPLACE  PROCEDURE GET_BOOK_PRICE_BY_ID
(
   BID IN NUMBER,
   BPRICE OUT NUMBER
)
AS
BEGIN
        SELECT BOOK_PRICE INTO BPRICE FROM BOOK_STORE WHERE BOOK_ID=BID;

END GET_BOOK_PRICE_BY_ID;
```

Below is the Dao class contains a method to call Stored Procedure from Database

```
public class BookStoreDAO {

    private SimpleJdbcCall sjc;

    public BigDecimal findPriceById(int bid) {
        BigDecimal price = null;

        // Setting procedure name
        sjc.setProcedureName("GET_BOOK_PRICE_BY_ID");

        Map<String, Object> inMap = new HashMap();
        inMap.put("BID", bid);

        // Executing procedure
        Map<String, Object> outMap = sjc.execute(inMap);

        if (!outMap.isEmpty()) {
            price = (BigDecimal) outMap.get("BPRICE");
        }
        return price;
    }

    public void setSjc(SimpleJdbcCall sjc) {
        this.sjc = sjc;
    }
}
```

Spring Bean Configuration file to Inject SimpleJdbcCall into Dao class

```
<bean id="ds"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
      <property name="url" value="jdbc:oracle:thin:@localhost:1521/XE" />
      <property name="username" value="SYSTEM" />
      <property name="password" value="admin" />
</bean>

<bean id="sjc" class="org.springframework.jdbc.core.simple.SimpleJdbcCall">
      <constructor-arg name="dataSource" ref="ds" />
</bean>

<bean id="dao" class="com.bs.dao.BookStoreDAO">
      <property name="sjc" ref="sjc" />
</bean>
```

Below is the Test class to call Dao class method

TestBookStoreDao.java

```
1  package com.bs.test;
2
3  import java.math.BigDecimal;
9
10 public class TestBookStoreDao {
11
12     public static void main(String[] args) {
13
14         ApplicationContext context = new ClassPathXmlApplicationContext("com/bs/cfg/Beans.xml");
15         BookStoreDAO dao = context.getBean("dao", BookStoreDAO.class);
16         BigDecimal price = dao.findPriceById(104);
17         System.out.println("Book price : " + price);
18     }
19 }
```

Working with BLOB and CLOB

You can store images, other binary objects, and large chunks of text. These large objects are called BLOB for binary data and CLOB for character data. In Spring you can handle these large objects by using the JdbcTemplate directly and also when using the higher abstractions provided by RDBMS Objects and the SimpleJdbc classes. All of these approaches use an implementation of the LobHandler interface for the actual management of the LOB data. The LobHandler provides access to a LobCreator class, through the getLobCreator method, used for creating new LOB objects to be inserted.

The LobCreator/LobHandler provides the following support for LOB input and output:

BLOB

- byte [ ] – getBlobAsBytes and setBlobAsBytes
- InputStream – getBlobAsBinaryStream and setBlobAsBinaryStream

CLOB

- String – getClobAsString and setClobAsString
- InputStream – getClobAsAsciiStream and setClobAsAsciiStream
- Reader – getClobAsCharacterStream and setClobAsCharacterStream

```
final File blobIn = new File("spring2004.jpg");
final InputStream blobIs = new FileInputStream(blobIn);
final File clobIn = new File("large.txt");
final InputStream clobIs = new FileInputStream(clobIn);
final InputStreamReader clobReader = new InputStreamReader(clobIs);
jdbcTemplate.execute(
  "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)",
  new AbstractLobCreatingPreparedStatementCallback(lobHandler) {❶
      protected void setValues(PreparedStatement ps, LobCreator lobCreator)
          throws SQLException {
        ps.setLong(1, 1L);
        lobCreator.setClobAsCharacterStream(ps, 2, clobReader, (int)clobIn.length());❷
        lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int)blobIn.length());❸
      }
  }
);
blobIs.close();
clobReader.close();
```

❶ Pass in the lobHandler that in this example is a plain `DefaultLobHandler`

❷ Using the method `setClobAsCharacterStream`, pass in the contents of the CLOB.

❸ Using the method `setBlobAsBinaryStream`, pass in the contents of the BLOB.

```
List<Map<String, Object>> l = jdbcTemplate.query("select id, a_clob, a_blob from
 lob_table",
        new RowMapper<Map<String, Object>>() {
          public Map<String, Object> mapRow(ResultSet rs, int i) throws SQLException {
            Map<String, Object> results = new HashMap<String, Object>();
            String clobText = lobHandler.getClobAsString(rs, "a_clob");❶
            results.put("CLOB", clobText);
            byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a_blob");❷
            results.put("BLOB", blobBytes);
            return results;
          }
        });
```

❶ Using the method `getClobAsString`, retrieve the contents of the CLOB.

❷ Using the method `getBlobAsBytes`, retrieve the contents of the BLOB.

Spring MVC

Spring

We spend our lives fiddling with countless devices and we spend many hours surfing the Internet and visiting all kinds of websites. Sometimes it's due to work-related reasons, although most of the time it's just for leisure or sheer entertainment. In recent years, it seems that any concern can be solved by means of a simple click. The app revolution and — by extension — the web app development revolution, has brought everyone closer to a world in which new technologies are growing by leaps and bounds.

To begin, we should start with what is web application?

A web-based application is any application that uses a website as the interface or front-end. Users can easily access the application from any computer connected to the Internet using a standard browser.

This contrasts with traditional desktop applications, which are installed on a local computer. For example, most of us are familiar with Microsoft Word, a common word-processing application that is a desktop application.

On the other hand, Google Docs is also a word-processing application but users perform all the functions using a web browser instead of using software installed on their computer. This means that it is a web-based application.

What are the business advantages of web applications?

Cost effective development: With web-based applications, users access the system via a uniform environment—the web browser. While the user interaction with the application needs to be thoroughly tested on different web browsers, the application itself needs only be developed for a single operating system.

There is no need to develop and test it on all possible operating system versions and configurations. This makes development and troubleshooting much easier and for web applications that use a Flash front end testing and troubleshooting is even easier.

Accessible anywhere: Unlike traditional applications, web systems are accessible anytime, anywhere and via any PC with an Internet connection. This puts the user firmly in charge of where and when they access the application.

It also opens up exciting, modern possibilities such as global teams, home working and real-time collaboration. The idea of sitting in front of a single computer and working in a fixed location is a thing of the past with web-based applications.

Easily customizable: The user interface of web-based applications is easier to customize than is the case with desktop applications. This makes it easier to update the look and feel of the application or to customize the presentation of information to different user groups.

Therefore, there is no longer any need for everyone to settle for using exactly the same interface at all times. Instead, you can find the perfect look for each situation and user.

Accessible for a range of devices: In addition to being customizable for user groups, content can also be customized for use on any device connected to the internet. This includes the likes of PDAs, mobile phones and tablets.

This further extends the user's ability to receive and interact with information in a way that suits them. In this way, the up to date information is always at the fingertips of the people who need it.

Improved interoperability: It is possible to achieve a far greater level of interoperability between web applications than it is with isolated desktop systems. For example, it is much easier to integrate a web-based shopping cart system with a web-based accounting package than it is to get two proprietary systems to talk to each other.

Because of this, web-based architecture makes it possible to rapidly integrate enterprise systems, improving work-flow and other business processes. By taking advantage of internet technologies you get a flexible and adaptable business model that can be changed according to shifting market demands.

Easier installation and maintenance: With the web-based approach installation and maintenance becomes less complicated too. Once a new version or upgrade is installed on the host server all users can access it straight away and there is no need to upgrade the PC of each and every potential user.

Rolling out new software can be accomplished more easily, requiring only that users have up-to-date browsers and plugins. As the upgrades are only performed by an experienced professional to a single server the results are also more predictable and reliable.

Adaptable to increased workload: Increasing processor capacity also becomes a far simpler operation with web-based applications. If an application requires more power to perform tasks only the server hardware needs to be upgraded.

The capacity of web-based software can be increased by "clustering" or running the software on several servers simultaneously. As workload increases, new servers can be added to the system easily.

For example, Google runs on thousands of inexpensive Linux servers. If a server fails, it can be replaced without affecting the overall performance of the application.

Increased Security: Web-based applications are typically deployed on dedicated servers, which are monitored and maintained by experienced server administrators. This is far more effective than monitoring hundreds or even thousands of client computers as is the case with desktop applications.

This means that security is tighter and any potential breaches should be noticed far more quickly.

Flexible core technologies: Any of three core technologies can be used for building web-based applications, depending on the requirements of the application. The Java-based solutions (J2EE) from Sun Microsystems involve technologies such as JSP and Servlets.

The newer Microsoft .NET platform uses Active Server Pages, SQL Server and .NET scripting languages. The third option is the Open Source platform (predominantly PHP and MySQL), which is best suited to smaller websites and lower budget applications.

Conclusion:

Web-based applications are:

- **Easier to develop**
- **More useful for your users**
- **Easier to install, maintain and keep secure**

Spring Web MVC

Spring framework makes the development of web applications very easy by providing the Spring MVC module. Spring MVC module is based on two most popular design patterns, they are

1) **Front controller Design pattern**
2) **MVC Design pattern**

Let us have glance at these Design Patterns before starring Spring Web MVC.

Web is having more and more demand these days. Since the desire of the companies and organizations are increasing so the complexity and the performance of the web programming matters. Complexity with the different types of communication devices is increasing. The business is demanding applications using the web and many communication devices. So with the increase load of the data on the internet we have to take care of the architecture issue. To overcome these issues, we need to have idea about design models. There are two types of design models available in java, they are

1. **Model 1 Architecture**
2. **Model 2 (MVC) Architecture**

Model 1 Architecture

Servlet and JSP are the main technologies to develop the web applications.

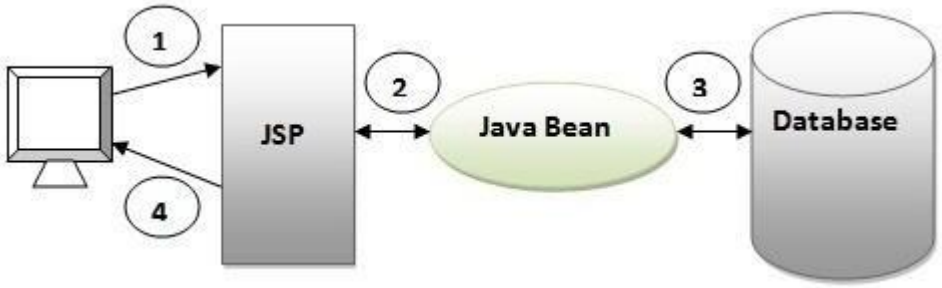
Servlet was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet.

Problem in Servlet Technology Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

Spring

JSP overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web

application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.



As you can see in the above figure, there is picture which show the flow of the model1 architecture.

- **Browser sends request for the JSP page**
- **JSP accesses Java Bean and invokes business logic**
- **Java Bean connects to the database and get/save data**
- **Response is sent to the browser which is generated by JSP**

Advantage of Model 1 Architecture

- Easy and Quick to develop web application

Disadvantage of Model 1 Architecture

- Navigation control is decentralized since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- Time consuming you need to spend more time to develop custom tags in JSP. So that we don't need to use script let tag
- Hard to extend It is better for small applications but not for large applications.

Model 2 Architecture

Model View Controller (MVC) is a software architecture pattern, commonly used to implement user interfaces: it is therefore a popular choice for architecting web apps. In general, it separates out the application logic into three separate parts, promoting modularity and ease of collaboration and reuse. It also makes applications more flexible and welcoming to iterations.

The MVC pattern was first described in 1979 by Trygve Reenskaug, then working on Smalltalk at Xerox research labs. The original implementation is described in depth in the influential paper “Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller”.

Smalltalk’s MVC implementation inspired many other GUI frameworks such as:

- **The NeXTSTEP and OPENSTEP development environments encourage the use of MVC. Cocoa and GNUstep, based on these technologies, also use MVC.**
- **Microsoft Foundation Classes (MFC) (also called Document/View architecture)**
- **Java Swing**
- **The Qt Toolkit (since Qt4 Release).**
- **XForms has a clear separation of model (stored inside the HTML head section) from the presentation (stored in the HTML body section). XForms uses simple bind commands to link the presentation to the model.**

MVC stands for Model, View and Controller. MVC separates application into three components - Model, View and Controller.

Spring

Model: The Model is where data from the controller and sometimes the view is actually passed into, out of, and manipulated. Keeping in mind our last example of logging into your web-based email, the model will take the username and password given to it from the controller, check that data against the stored information in the database, and then render the view accordingly. For example, if you enter in an incorrect password, the model will tell the controller that it was incorrect, and the controller will tell the view to display an error message saying something to the effect of "Your username or password is incorrect."

Model is a data and business logic.

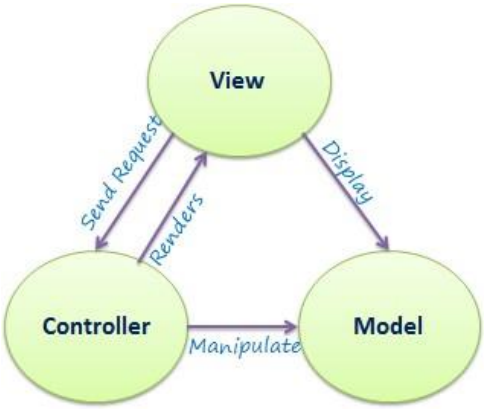
View: In a web-based application, the view is exactly what it sounds like: the visible interface that the user interacts with, displaying buttons, forms, and information. Generally speaking, the controller calls up the view after interacting with the model, which is what gathers the information to display in the particular view.

View is a User Interface.

Controller: The Controller is essentially the traffic cop of the application, directing traffic to where it needs to go, figuring out which view it needs to load up, and interacting with the appropriate models. For example, when you go to login to your email on a website, the controller is going to tell the application that it needs to load the login form view. Upon attempting to login, the controller will load the model that handles logins, which will check if the username and password match what exists within the system. If successful, the controller will then pass you off to the first page you enter when logging in, such as your inbox. Once there, the inbox controller will further handle that request.

Controller is a request handler.

The following figure illustrates the interaction between Model, View and Controller



The following figure illustrates the flow of the user's request

As per the above figure, when the user enters a URL in the browser, it goes to the server and calls appropriate controller. Then, the Controller uses the appropriate View and Model and creates the response and sends it back to the user.

Though MVC comes in different flavors, the control flow generally works as follows:

- The user interacts with the user interface in some way (e.g., user presses a button)

- A controller handles the input event from the user interface, often via a registered handler or callback.
- The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart). Complex controllers are often structured using the command pattern to encapsulate actions and simplify extension.
- A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. (However, the observer pattern can be used to allow the model to indirectly notify interested parties, potentially including views, of a change.)
- The user interface waits for further user interactions, which begins the cycle anew.

Conclusion

Now we know the basic concepts of the MVC pattern. Essentially, it allows for the programmer to isolate these very separate pieces of code into their own domain, which makes code maintenance and debugging much simpler than if all of these items were chunked into one massive piece. If I have a problem with an application not displaying an error message when it should, I have a very specific set of locations to look to see why this is not happening. First I would look at the "Login Controller" to see if it is telling the view to display the error. If that's fine, I would look at the "Login Model" to see if it is passing the data back to the controller to tell it that it needs to show an error. Then if that's correct, the last place it could be happening would be in the "Login View."

Front Controller design pattern

When the user access the view directly without going thought any centralized mechanism each view is required to provide its; own system services, often resulting in duplication of code and view navigations is left to the views, this may result in commingled view content and view navigation.

A centralized point of contact for handling a request may be useful, for example, to control and log a user's progress through the site. .

Introducing a controller as the initial point of contact for handling a request. The controller manages the handling of the request, including invoking security services such as authentication and authorization, delegating business processing, managing the choice of an appropriate view, handling errors, and managing the selection of content creation strategies.

The controller provides a centralized entry point that controls and manages Web request handling. By centralizing decision points and controls, the controller also helps reduce the amount of Java code, called scriptlets, embedded in the JavaServer Pages (JSP) page.

Centralizing control in the controller and reducing business logic in the view promotes code reuse across requests. It is a preferable approach to the alternative-embedding code in multiple views-because that approach may lead to a more error-prone, reuse-by-copy- and-paste environment.

Typically, a controller coordinates with a dispatcher component. Dispatchers are responsible for view management and navigation. Thus, a dispatcher chooses the next view for the user and vectors control to the resource. Dispatchers may be encapsulated within the controller directly or can be extracted into a separate component.

The responsibilities of the components participating in this pattern are:

Controller: The controller is the initial contact point for handling all requests in the system. The controller may delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.

Dispatcher: A dispatcher is responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource. A dispatcher can be encapsulated within a controller or can be a separate component working in coordination. The dispatcher provides either a static dispatching to the view or a more sophisticated dynamic dispatching mechanism. The dispatcher uses the RequestDispatcher object (supported in the servlet specification) and encapsulates some additional processing.

Helper: A helper is responsible for helping a view or controller complete its processing. Thus, helpers have numerous responsibilities, including gathering data required by the view and storing this intermediate model, in which case the helper is sometimes referred to as a value bean. Additionally, helpers may adapt this data model for use by the view. Helpers can service requests for data from the view by simply providing access to the raw data or by formatting the data as Web content. A view may work with any number of helpers, which are typically implemented as JavaBeans components (JSP 1.0+) and custom tags (JSP 1.1+). Additionally, a helper may represent a Command object, a delegate, or an XSL Transformer, which is used in combination with a stylesheet to adapt and convert the model into the appropriate form.

View: A view represents and displays information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display.

Front Controller centralizes control. A controller provides a central place to handle system services and business logic across multiple requests. A controller manages business logic processing and request handling. Centralized access to an application means that requests are easily tracked and logged. Keep in mind, though, that as control centralizes, it is possible to introduce a single point of failure. In practice, this rarely is a problem, though, since multiple controllers typically exist, either within a single server or in a cluster.

Front Controller improves manageability of security. A controller centralizes control, providing a choke point for illicit access attempts into the Web application. In addition, auditing a single entrance into the application requires fewer resources than distributing security checks across all pages.

Front Controller improves reusability. A controller promotes cleaner application partitioning and encourages reuse, as code that is common among components moves into a controller or is managed by a controller.

Benefits

The following lists the benefits of using the Front Controller pattern:

- ✓ **Centralizes control logic**
- ✓ **Improves reusability**
- ✓ **Improves separation of concerns**

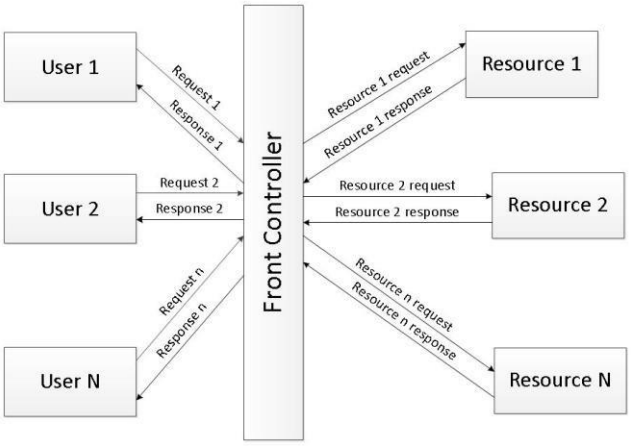
When to Use

You should use the Front Controller pattern to:

- ✓ **Apply common logic to multiple requests**
- ✓ **Separate processing logic from view**

This design pattern enforces a single point of entry for all the incoming requests. All the requests are handled by a single piece of code which can then further delegate the responsibility of processing the request to further application objects.

Front Controller Design pattern



Introduction to the spring web MVC framework

Spring's web MVC framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for upload files. The default handler is a very simple Controller interface, just offering a ModelAndView handleRequest (request, response) method. This can already be used for application controllers, but you will prefer the included implementation hierarchy, consisting of, for example AbstractController, AbstractCommandController and SimpleFormController. Application controllers will typically be subclasses of those. Note that you can choose an appropriate base class: If you don't have a form, you don't need a FormController. This is a major difference to Struts.

You can use any object as a command or form object - there's no need to implement an interface or derive from a base class. Spring's data binding is highly flexible, for example, it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. So you don't need to duplicate your business objects' properties as Strings in your form objects, just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it is often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes like Action and Action Form - for every type of action.

Compared to Web Work, spring has more differentiated object roles. It supports the notion of a Controller, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data. Instead, a Web Work Action combines all those roles into one single object. Web Work does allow you to use existing business objects as part of your form, but only by making them bean properties of the respective Action class.

Finally, the same Action instance that handles the request is used for evaluation and form population in the view. Thus, reference data needs to be modeled as bean properties of the Action too. These are arguably too many roles for one object.

Spring's view resolution is extremely flexible. A Controller implementation can even write a view directly to the response, returning null as ModelAndView. In the normal case, a ModelAndView instance consists of a view name and a model Map, containing bean names and corresponding objects (like a command or form, containing reference data). View name resolution is highly configurable, either via bean names, via a properties file, or via your own ViewResolver implementation. The abstract model Map

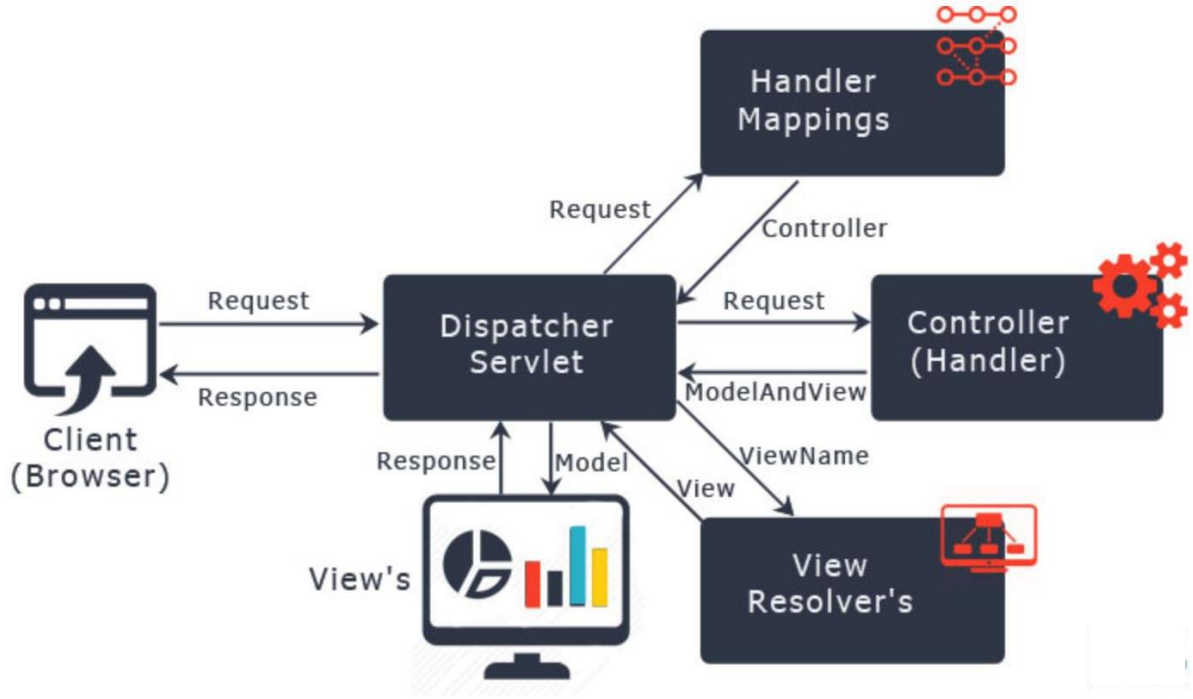
allows for complete abstraction of the view technology, without any hassle. Any renderer can be integrated directly, whether JSP, Velocity, or any other rendering technology. The model Map is simply transformed into an appropriate format, such as JSP request attributes or a Velocity template model.

Features of Spring MVC

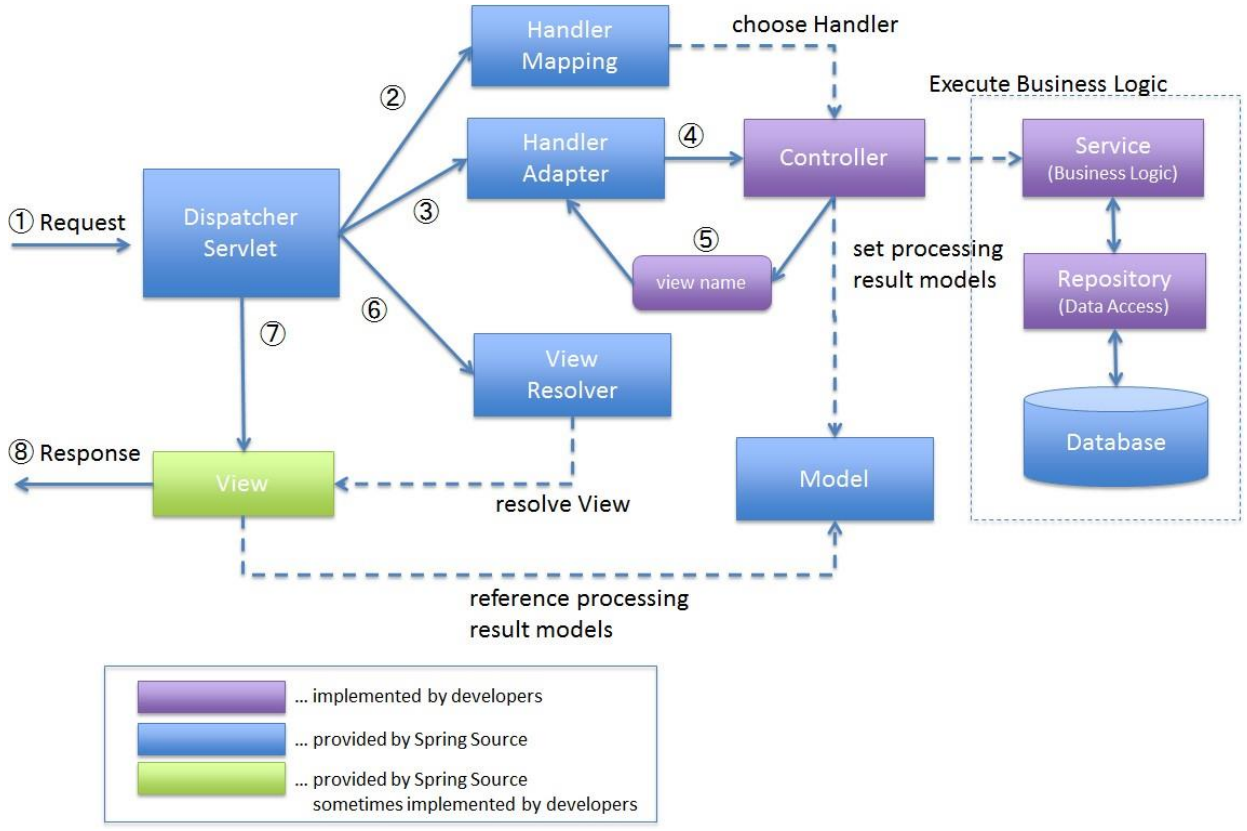
Spring's web module provides a wealth of unique web support features, including:

- Clear separation of roles - controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy referencing across contexts, such as from web controllers to business objects and validators.
- Adaptability, non-intrusiveness - Use whatever controller subclass you need (plain, command, form, wizard, multi-action, or a custom one) for a given scenario instead of deriving from a single controller for everything.
- Reusable business code - no need for duplication. You can use existing business objects as command or form objects instead of mirroring them in order to extend a particular framework base class.
- Customizable binding and validation - type mismatches as application-level validation errors that keep the offending value, localized date and number binding, etc instead of String-only form objects with manual parsing and conversion to business objects.
- Customizable handler mapping and view resolution - handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. This is more flexible than some web MVC frameworks which mandate a particular technique.
- Flexible model transfer - model transfer via a name/value Map supports easy integration with any view technology.
- Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, etc.
- A simple but powerful tag library - That avoids HTML generation at any cost, allowing for maximum flexibility in terms of markup code.

Spring MVC Architecture



Spring MVC Architecture



Request Flow execution:

1. **Client sends HTTP requests**
2. **Client's requests are intercepted by "DispatcherServlet", which tries to figure out appropriate handler mapping. DispatcherServlet consults Handler Mapping to identify Controller who is supposed to handle this request.**
3. **Based on Handler Mapper response Dispatcher Servlet invokes the Controller associated with the request. Now, the controller processes the requests invoking appropriate service method and returns the response/result (i.e.; ModelAndView instance which contains model data and view name) back to the DispatcherServlet.**
4. **Then DispatcherServlet sends the view name from ModelAndView instance to view resolver object.**
5. **Time to show output to the user: view name along with model data will render the response/result back to the user.**

DispatcherServlet

Spring's web MVC framework is, like many other web MVC frameworks, a request-driven web MVC framework, designed around a servlet that dispatches requests to controllers and offers other functionality facilitating the development of web applications.

Like ordinary servlets, the DispatcherServlet is declared in the web.xml of your web application. Requests that you want the DispatcherServlet to handle, will have to be mapped, using a URL mapping in the same web.xml file like below

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <display-name>SpringMVCApp1</display-name>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>
```

As per above configuration, all requests ending with .htm will be handled by the DispatcherServlet.

While Initializing the DispatcherServlet, it looks for the configuration file (DispatcherServletName-servlet.xml) for the web component beans declarations. Based on the above configuration it looks for dispatcher-servlet.xml file. Now loads all the bean declarations and creates a WebApplicationContainer with the web component beans.

Note: We can customize the [DispatcherServletName]-servlet.xml file name by configuring init-parameter to DispatcherServlet. Where param-name will be 'contextConfigLocation' and param-value will be path of the configuration file.

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <display-name>SpringMVCApp1</display-name>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/Web-MVC-Beans.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>
```

But in typical web application we will have 2 types of components. They are

4) **Web Components (Handler Mappers, Controller, View Resolvers, Themes etc)**
5) **Business Components (Service , Dao and DataSource etc)**

It is not recommended to combine Business components with Web Components in Single Bean configuration file. So we will create another ApplicationContext for managing business components.

- WebComponents will be handled by WebApplicationContext
- Business components will be handled by ApplicationContext

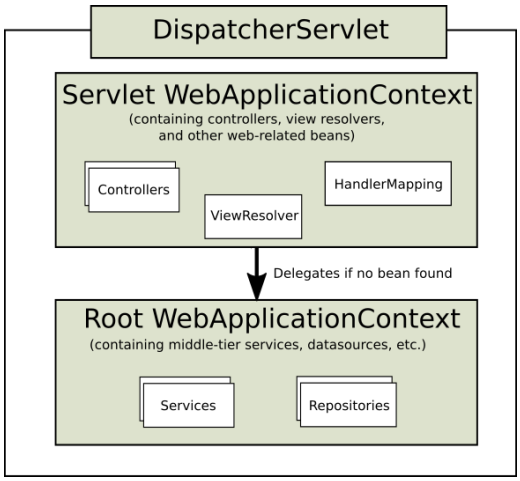
Configuring ApplicationContext

```xml
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootApplicationContext.xml</param-value>
    </context-param>
```

ContextLoaderListener

Performs the actual initialization work for the root application context. Reads a “contextConfigLocation” context-param and passes its value to the context instance, parsing it into potentially multiple file paths which can be separated by any number of commas and spaces, e.g. “WEB-INF/applicationContext1.xml, WEB-INF/applicationContext2.xml”.

Typical Context hierarchy in Spring Web MVC



Handler Mapping

HandlerMapping is an interface that is implemented by all Objects that map the request to the corresponding Handler Object. The default Implementations used by the DispatcherServlet are BeanNameUrlHandlerMapping and DefaultAnnotationHandlerMapping.

Below are the some Handler mappers in Spring

1. **BeanNameUrlHandlerMapping**
2. **DefaultAnnotationHandlerMapping**
3. **ControllerBeanNameHandlerMapping**
4. **ControllerClassNameHandlerMapping**
5. **SimpleUrlHandlerMapping**
6. **RequestMappingHandlerMapping (This was introduced in spring 3.1)**

BeanNameURLHandlerMapping

BeanNameUrlHandlerMapping maps request URLs to beans with the same name.

Two ways to define bean name for BeanNameUrlHandlerMapping :

1. You can define static url in bean name as we have do with the name “/hello.htm”, when ever the url “/hello.htm” will be requested by browser, BeanNameUrlHandlerMapping will return respective url.
2. You can define “*” in bean name to hold any number of characters in between the name. As we have done with the bean name “/sayHello*”. All the requests with url starting with “/sayHello” will be parsed to this bean. We will see this in our example.

dispatcher-servlet.xml

```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"
xmlns:mvc="http://www.springframework.org/schema/mvc"
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
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <bean name="/hello.htm" class="com.webapp.controller.HelloController"/>

    <bean name="/sayHello*" class="com.webapp.controller.HelloController"/>

    <bean id="urlHandler" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>


    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

Note : BeanNameUrlHandlerMapping is the default url handler used by Spring MVC. That means if we do not specify any url handler in our Spring MVC configuration, Spring will automatically load this url handler and pass requested urls to BeanNameUrlHandlerMapping to get the controller bean to be executed.

SimpleURLHandlerMapping

This type of HandlerMapping is the simplest of all handler mappings which allows you specify URL pattern and handler explicitly.

There are two ways of defining SimpleUrlHandlerMapping, using <value> tag and <props> tag. SimpleUrlHandlerMapping has a property called mappings we will be passing the URL pattern to it.

```xml
    <!-- Using Value Tag -->
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /welcome.htm=welcomeController
                /welcome*=welcomeController
                /hell*=helloWorldController
                /helloWorld.htm=helloWorldController
            </value>
        </property>
    </bean>

    <!--  Using Prop Tag -->
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/welcome.htm">welcomeController</prop>
                <prop key="/welcome*">welcomeController</prop>
                <prop key="/helloworld">helloWorldController</prop>
                <prop key="/hello*">helloWorldController</prop>
                <prop key="/HELLOworld">helloWorldController</prop>
            </props>
        </property>
    </bean>
```

Left Side of "=" is URL Pattern and right side is the id or name of the bean

ControllerClassNameHandlerMapping

ControllerClassHandlerMapping uses a convention to map the requested URL to the Controller. It will take the Controller name and converts them to lower case with a leading "/".

```
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />

    <bean class="com.app.controller.HelloWorldController"></bean>
    <bean class="com.app.controller.WelcomeController"></bean>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
```

As per above configuratoin

- helloworld is requested, the DispatcherServlet redirects it to the HelloWorldController.
- helloworld123 is requested, the DispatcherServlet redirects it to the HelloWorldController.
- welcome is requested, the DispatcherServlet redirects it to the WelcomeController.
- welcome123 is requested, the DispatcherServlet redirects it to the WelcomeController.
- helloWorld is requested, you will get 404 error as "W" is capitalized here.

RequestMappingHandlerMapping

This was introduced in spring 3.1. Earlier the DefaultAnnotationHandlerMapping decided which controller to use and the AnnotationMethodHandlerAdapter selected the actual method that handled the request. RequestMappingHandlerMapping does both the tasks. Therefore the request is directly mapped right to the method. The following types can be passed to method handlers.

MessageController.java

```
package com.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class MessageController {

    @RequestMapping("/wish")
    public ModelAndView generateWishMsg() {
        String message = "Welcome to Ashok IT School..!!";

        ModelAndView mav = new ModelAndView();
        mav.setViewName("welcome");
        mav.addObject("msg", message);

        return mav;
    }

}
```

Controllers

In Spring MVC, we write a controller class to handle requests coming from the client. Then the controller invokes a business class to process business-related tasks, and then redirects the client to a logical view name which is resolved by the Spring's dispatcher servlet in order to render results or output. That completes a round trip of a typical request-response cycle.

Spring MVC provides many abstract controllers, which is designed for specific tasks. Here is the list of anstract controllers that comes with the Spring MVC module:

1. **SimpleFormController**
2. **AbstractController**
3. **AbstractCommandController**
4. **CancellableFormController**
5. **AbstractCommandController**
6. **MultiActionController**
7. **ParameterizableViewController**
8. **ServletForwardingController**
9. **ServletWrappingController**
10. **UrlFilenameViewController**
11. **AbstractController**
12. **AbstractCommandController**
13. **SimpleFormController**
14. **CancellableFormController etc.**

Controller.java

```
1  package com.controller;
2
3  import javax.servlet.http.HttpServletRequest;
4  import javax.servlet.http.HttpServletResponse;
5
6  import org.springframework.web.servlet.ModelAndView;
7
8  public interface Controller {
9
10     /**
11      * Process the request and return a ModelAndView object which the
12      * DispatcherServlet will render.
13      */
14     public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res) throws Exception;
15
16 }
```

As you can see, the Controller interface requires a single method that should be capable of handling a request and returning an appropriate model and view. These three concepts are the basis for the Spring MVC implementation - *ModelAndView* and *Controller*. While the Controller interface is quite abstract, Spring offers a lot of controllers that already contain a lot of the functionality you might need. The Controller interface just defines the most common functionality required of every controller - handling a request and returning a model and a view.

Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as @RequestMapping, @RequestParam, @ModelAttribute, and so on. This annotation support is available for both Servlet MVC and Portlet MVC. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. Furthermore, they do not usually have direct dependencies on Servlet or Portlet APIs, although you can easily configure access to Servlet or Portlet facilities.

```
@Controller
public class HelloWorldController {
```

```
  @RequestMapping("/helloWorld")
  public String helloWorld(Model model) {
    model.addAttribute("message", "Hello World!");
    return "helloWorld";
  }
}
```

As you can see, the @Controller and @RequestMapping annotations allow flexible method names and signatures. In this particular example the method accepts a Model and returns a view name as a String, but various other method parameters and return values can be used as explained later in this section. @Controller and @RequestMapping and a number of other annotations form the basis for the Spring MVC implementation. This section documents these annotations and how they are most commonly used in a Servlet environment.

The @Controller annotation indicates that a particular class serves the role of a ***controller***. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to. The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans suchannotated classes for mapped methods and detects @RequestMapping annotations (see the next section).
You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the @Controller stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.

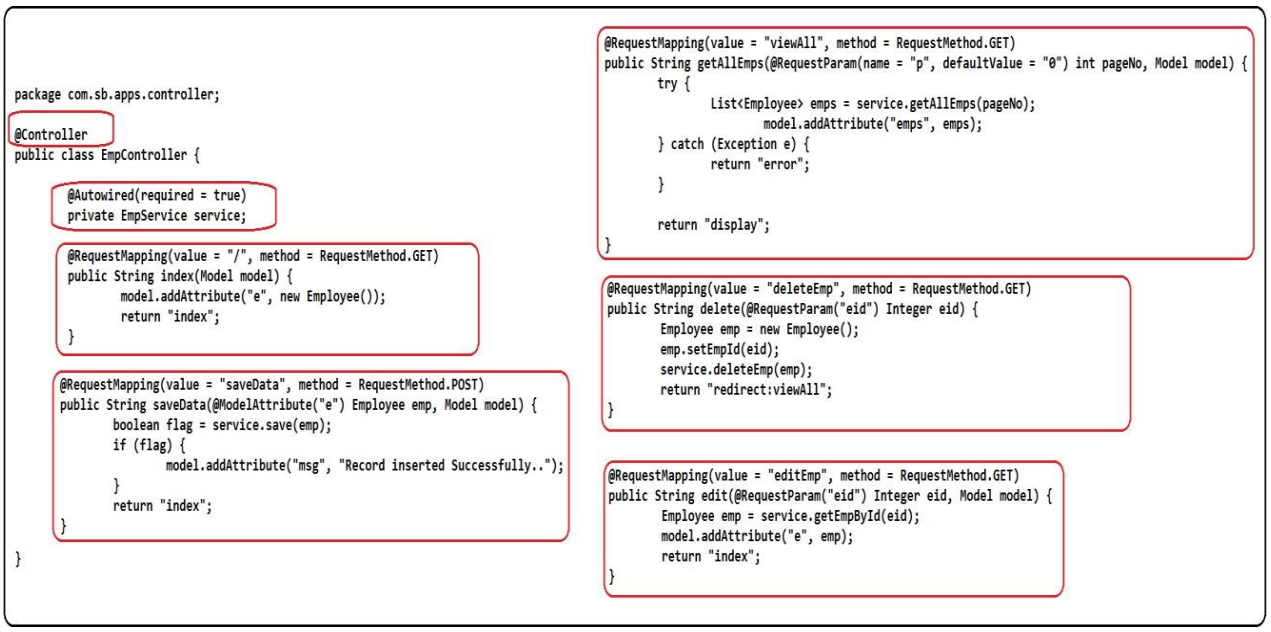
To enable autodetection of such annotated controllers, you add component scanning to your configuration. Use the ***spring-context*** schema as shown in the following XML snippet:

dispatcher-servlet.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"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <!-- To Enable Annotations -->
    <mvc:annotation-driven />

    <!-- To Scan Component classes -->
    <context:component-scan base-package="com.app.*" />

    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

Mapping Requests with @RequestMapping

You use the @RequestMapping annotation to map URLs such as /appointments onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.



```
package com.sb.apps.controller;

@Controller
public class EmpController {

	@Autowired(required = true)
	private EmpService service;

	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String index(Model model) {
		model.addAttribute("e", new Employee());
		return "index";
	}

	@RequestMapping(value = "saveData", method = RequestMethod.POST)
	public String saveData(@ModelAttribute("e") Employee emp, Model model) {
		boolean flag = service.save(emp);
		if (flag) {
			model.addAttribute("msg", "Record inserted Successfully..");
		}
		return "index";
	}
}
```

```
@RequestMapping(value = "viewAll", method = RequestMethod.GET)
public String getAllEmps(@RequestParam(name = "p", defaultValue = "0") int pageNo, Model model) {
	try {
		List<Employee> emps = service.getAllEmps(pageNo);
			model.addAttribute("emps", emps);
	} catch (Exception e) {
		return "error";
	}

	return "display";
}
```

```
@RequestMapping(value = "deleteEmp", method = RequestMethod.GET)
public String delete(@RequestParam("eid") Integer eid) {
	Employee emp = new Employee();
	emp.setEmpId(eid);
	service.deleteEmp(emp);
	return "redirect:viewAll";
}
```

```
@RequestMapping(value = "editEmp", method = RequestMethod.GET)
public String edit(@RequestParam("eid") Integer eid, Model model) {
	Employee emp = service.getEmpById(eid);
	model.addAttribute("e", emp);
	return "index";
}
```

Any request coming in, will be intercepted by the TimeBasedAccessInterceptor, and if the current time is outside office hours, the user will be redirected to a static html file, saying, for example, he can only access the website during office hours.
As you can see, spring has an adapter to make it easy for you to extend the HandlerInterceptor.

Views and resolving them

All MVC frameworks for web applications provide a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. Out of the box, Spring enables you to use Java Server Pages, Velocity templates and XSLT views, The two interfaces which are important to the way Spring handles viewsare ViewResolver and View. The ViewResolverprovides a mapping between view names and actual views. The View interface addresses the preparation of the request and hands the request over to one of the view technologies.

ViewResolvers

As discussed all controllers in the spring web MVC framework, return a ModelAndViewinstance. Views in spring are addressed by a view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. We'll list most of them

ViewResolver	Description
AbstractCachingViewResolver	**An abstract view resolver which takes care of caching views. Often views need preparation before they can be used, extending this view resolver provides caching of views.**
XmlViewResolver	**An implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's bean factories. The default configuration file is /WEB-INF/views.xml.**
ResourceBundleViewResolver	**An implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath. The default file name is views.properties.**
UrlBasedViewResolver	**A simple implementation of ViewResolver that allows for direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.**

ViewResolver	Description
InternalResourceViewResolver	A convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses like JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass. See UrlBasedViewResolver's javadocs for details.
VelocityViewResolver / FreeMarkerViewResolver	A convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them.

As an example, when using JSP for a view technology you can use the UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over the RequestDispatcher to render the view.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
   <property name="prefix">
          <value>/WEB-INF/jsp/</value>
    </property>
  <property name="suffix">
          <value>.jsp</value>
    </property>
</bean>
```

When returning test as a viewname, this view resolver will hand the request over to the RequestDispatcher that will send the request to /WEB-INF/jsp/test.jsp. When mixing different view technologies in a web application, you can use the ResourceBundleViewResolver:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"  />
  <property name="defaultParentView" value="parentView" />
</bean>
```

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].class as the view class and the value of the property[viewname].url as the view url. As you can see, you can identify a parent view, from which all views in the properties file sort of extend. This way you can specify a default view class, for example A note on caching - subclasses of AbstractCachingViewResolver cache view instances they have resolved. This greatly improves performance when using certain view technology. It's possible to turn off the cache, by setting the cacheproperty to false. Furthermore, if you have the requirement to be able to refresh a certain view at runtime (for example when a Velocity template has been modified), you can use the removeFromCache (String viewName, Locale loc) method.

Chaining View Resolvers

Spring supports more than just one view resolver. This allows you to chain resolvers and, for example, override specific views in certain circumstances. Chaining view resolvers is pretty straightforward - just add more than one resolver to your application context and, if necessary, set the order property to specify an order. Remember, the higher the order property, the later the view resolver will be positioned in the chain.

In the following example, the chain of view resolvers consists of two resolvers, a InternalResourceViewResolver (which is always automatically positioned as the last resolver in the chain) and an XmlViewResolver for specifying Excel views (which are not supported by the InternalResourceViewResolver):

```
<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

### views.xml
<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

If a specific view resolver does not result in a view, Spring will inspect the context to see if other view resolvers are configured. If there are additional view resolvers, it will continue to inspect them. If not, it will throw an Exception.
You have to keep something else in mind - the contract of a view resolver mentions that a view resolver *can* return null to indicate the view could not be found. Not all view resolvers do this however! This is because in some cases, the resolver simply cannot detect whether or not the view exists. For example, the InternalResourceViewResolver uses the RequestDispatcher internally, and dispatching is the only way to figure out if a JSP exists -this can only be done once. The same holds for the VelocityViewResolver and some others. Check the JavaDoc for the view resolver to see if you're dealing with a view resolver that does not report non-existing views. As a result of this, putting anInternalResourceViewResolver in the chain in a place other than the last, will result in the chain not being fully inspected, since the InternalResourceViewResolver will *always* return a view!

Redirecting to views

As has been mentioned, a controller normally returns a logical view name, which a view resolver resolves to a particular view technology. For view tehcnologies such as JSPs that are actually processed via the Servlet/JSP engine, this is normally handled via InternalResourceViewResolver/InternalResourceView which will ultimately end up issuing an internal forward or include, via the Servlet API's RequestDispatcher.forward() or RequestDispatcher.include(). For other view technologies, such as Velocity, XSLT, etc., the view itself produces the content on the response stream.

It is sometimes desireable to issue an HTTP redirect back to the client, before the view is rendered. This is desireable for example when one controller has been called with POSTed data, and the response is actually a delegation to another controller (for example on a successful form submission). In this case, a normal internal forward will mean the other controller will also see the same POST data, whih is potentially problematic if it can confuse it with other expected data Another reason to do a redirect before displaying the result is that this will eliminate the possiblity of the user doing a double submission of form data. The browser will have sent the initial POST, will have seen a redirect back and done a subsequent GET because of that, and thus as far as it is concerned, the current page does not reflect the result of a POST, but rather of a GET, so there is no way the user can accidentally re-POST the same data by doing a refresh. The refresh would just force a GET of the result page, not a resend of the initial POST data.

Redirect View

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring's RedirectView. In this case, DispatcherServlet will not use the normal view resolution mechanism, but rather as it has beengiven the (redirect) view already, will just ask it to do it's work.

The RedirectView simply ends up issuing an HttpServletResponse.sendRedirect() call, which will come back to the client browser as an HTTP redirect. All model attributes are simply exposed as HTTP query parameters. This does mean that the model must contain only objects (generally Strings or convertible to Strings) which can be readily converted to a string-form HTTP query parameter.

If using RedirectView, and the view is created by the Controller itself, it is generally always preferable if the redirect URL at least is injected into the Controller, so that it is not baked into the controller but rather configured in the context along with view names and the like.

The redirect: prefix

While the use of RedirectView works fine, if the controller itself is creating the RedirectView, there is no getting around the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really care about how the response gets handled. It should generally think only in terms of view names, that have been injected into it.

The special redirect: prefix allows this to be achived. If a view name is returned which has the prefix redirect:, thenUrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a RedirectView, but now the controller itself can deal just in terms of logical view names. A logical view name such as redirect:/my/response/controller.html will redirect relative to the current servlet

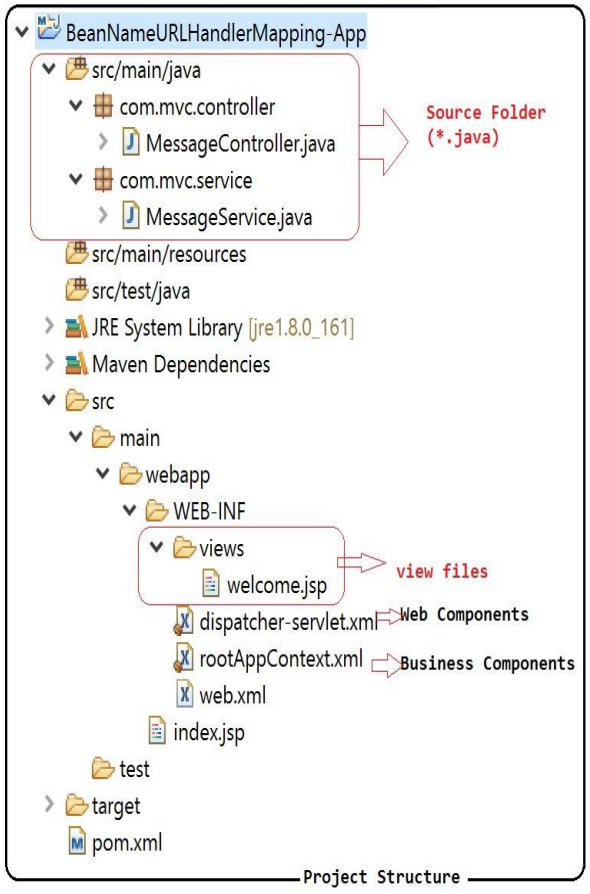
context, while a name such as redirect:http://myhost.com/some/arbitrary/path.htmlwill redirect to an absolute URL. The important thing is that as long is this redirect view name is injected into the controller like any other logical view name, the controller is not even aware that redirection is happening.

The forward: prefix

It is also possible to use a special forward: prefix for view names that will ultimately be resolved byUrlBasedViewResolver and subclasses. All this does is create an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL. Therefore, there is never any use in using this prefix when using InternalResourceViewResolver/InternalResourceView anyway (for JSPs for example), but it's of potential use when you are primarilly using another view technology, but want to still be able to in some cases force a forward to happen to a resource to be handled by the Servlet/JSP engine. Note that if you need to do this a lot though, you may also just chain multiple view resolvers. As with the redirect: prefix, if the view name with the prefix is just injected into the controller, the controller does not have to be aware that anything special is happening in terms of handling the response.

BeanNameUrlHandlerMapping-Application

Step-1: Create Maven web application and configure Spring Web Mvc and java.servlet-api dependencies in pom.xml file like below



BeanNameURLHandlerMapping-App/pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.web</groupId>
    <artifactId>BeanNameURLHandlerMapping-App</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>BeanNameURLHandlerMapping-App Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.3.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>BeanNameURLHandlerMapping-App</finalName>
    </build>
</project>
```

pom.xml

Step-2: Create Controller and Service classes

MessageController.java

```java
package com.mvc.controller;

import javax.servlet.http.HttpServletRequest;

@Controller
public class MessageController extends AbstractController {

    public MessageController() {
        System.out.println("****MessageController:::Constructor****");
    }

    @Autowired(required = true)
    private MessageService msgService;

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        ModelAndView mav = new ModelAndView();

        mav.setViewName("welcome");

        mav.addObject("msg", msgService.getMessage());

        return mav;
    }
}
```

MessageController.java

MessageService.java

```java
package com.mvc.service;

import org.springframework.stereotype.Service;

@Service
public class MessageService {

    public MessageService() {
        System.out.println("****MessageService:::Constructor****");
    }

    public String getMessage() {
        return "Welcome to Spring MVC";
    }
}
```

MessageService.java

Step-3: Create view file to display as response to end user (here we are using jsp as view technology)

welcome.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Spring MVC App</title>
</head>
<body>
    <h1>${msg}</h1>
</body>
</html>
```

welcome.jsp

Step-4: Create rootApplicationContext configuration file to configure Business components

rootAppContext.xml

```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:aop="http://www.springframework.org/schema/aop"
    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/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!-- Declaring Business components -->
    <bean id="msgService" class="com.mvc.service.MessageService" />

</beans>
```

rootAppContext.xml

Step-5: Create dispatcher-servlet.xml to declare Web components

dispatcher-servlet.xml

```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"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <!-- To Enable Annotations -->
    <mvc:annotation-driven />

    <!-- To Scan Component classes -->
    <context:component-scan base-package="com.mvc.*" />

    <bean name="/welcome.htm" class="com.mvc.controller.MessageController" />

    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

dispatcher.servlet.xml

Step-6: Configure DispatcherServlet & rootAppContext file in web.xml file like below

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">
    <display-name>SpringMVCApp1</display-name>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootAppContext.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>

</web-app>
```

web.xml

Step-7: Deploy the application in JEE server

Once project deployed into Server below operations will be performed

- **Server will try to load web.xml file from project WEB-INF folder.**
- **If web.xml is available, server checks for well-formness and validness of xml**
- **If xml is well-formed and valid, server will use jax-p api to parse web.xml content in xml style**
- **Server will create ServletContext object and will trigger contextCreated event**
- **If ContextLoaderListener is configured, then it will start IOC using ApplicationContext by passing context param value as input. This IoC will act as rootAppliationContext.**

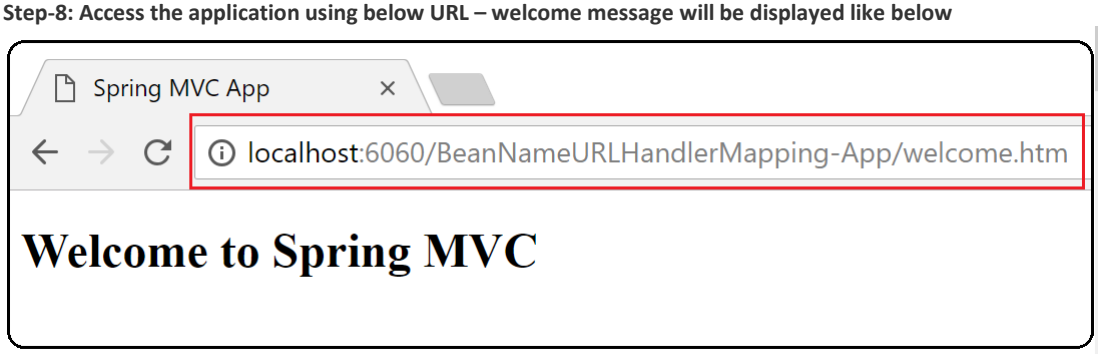
Note: context-param name should be contextConfigLocation and value will be path of the rootAppliationContext config file

- **If the given file is available, IOC will load all business components configured in given bean spring bean configuration file and will instantiate all singleton beans.**
- **Then Server will check for Servlets which are having load-on-startup, if available container will instantiate that servlet and it will call init method and service method of that servlet.**
- **Note : In Spring application we will configure DispatcherServlet with load-on-startup to create WebApplicationContext before receiving first request itself**
- **DispatcherServlet init method will start WebApplicationContext by passing disparcher-servlet.xml file as input.**
- **WebApplicationContext will read web components configured in Bean Configuration and will instantiate all singleton beans.**

Note: If no handler mapper configured, then DispatcherServlet will use BeanNameUrlHandlerMapper as a default handler mapper.

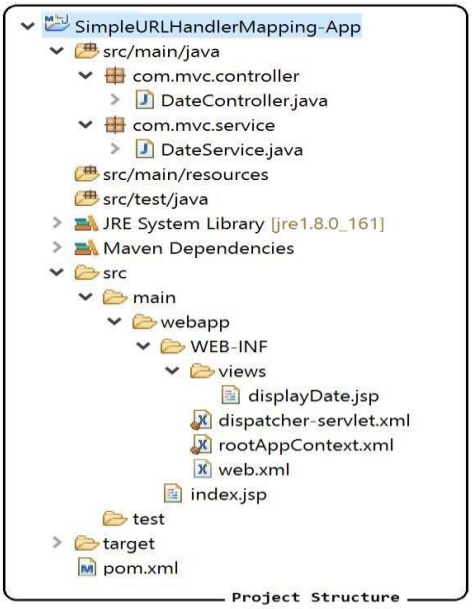
Once Deployment process is completed, RootAppliationContext and WebApplicationContext specific beans objects will be created and they will be ready to use.

Step-8: Access the application using below URL – welcome message will be displayed like below



SimpleUrlHandlerMapping-Application

Step-1: Create Maven web application and configure Spring Web Mvc and java.servlet-api dependencies in pom.xml file like below




```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.3.6.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
    </dependency>
</dependencies>
```

pom.xml

Step-2: Create Controller and Service classes

DateController.java

```
package com.mvc.controller;

import javax.servlet.http.HttpServletRequest;

public class DateController extends AbstractController {

    @Autowired(required = true)
    private DateService dateService;

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req,
            HttpServletResponse res) throws Exception {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("displayDate");
        mav.addObject("date", dateService.getTodaysDate());
        return mav;
    }

}
```

DateController.java

DateService.java

```
package com.mvc.service;

import java.util.Date;

@Service
public class DateService {

    public Date getTodaysDate() {
        return new Date();
    }
}
```

DateServie.java

Step-3: Create view file to display as response to end user (here we are using jsp as view technology)

displayDate.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Simple URL Handler Mapping App</title>
</head>
<body>
    <h1>Today's Date is : ${date}</h1>
</body>
</html>
```

displayDate.jsp

Step-4: Create rootApplicationContext & WebApplicationContext configuration file to configure Business components & web Components

dispatcher-servlet.xml

```xml
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/date.htm">dateController</prop>
            </props>
        </property>
    </bean>

    <bean id="dateController" class="com.mvc.controller.DateController" />


    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
```

dispatcher-servlet.xml

rootAppContext.xml

```xml

    <!-- Declaring Business components -->
    <bean id="msgService" class="com.mvc.service.DateService" />
```

rootAppContext.xml

Step-6: Configure DispatcherServlet & rootAppContext file in web.xml file like below

web.xml

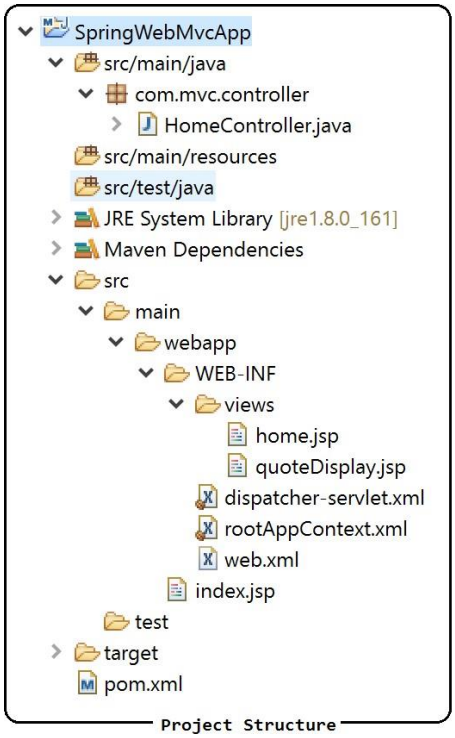
```xml
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootAppContext.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
```

web.xml

Step-7: Deploy the application in JEE server and access Controller using below URL.

Spring Web MVC Application using @Controller and @RequestMapping

Step-1: Create Maven web project and configure maven dependencies in project pom.xml file



Project Structure

```
12  <dependencies>
13      <dependency>
14          <groupId>junit</groupId>
15          <artifactId>junit</artifactId>
16          <version>3.8.1</version>
17      </dependency>
18      <dependency>
19          <groupId>org.springframework</groupId>
20          <artifactId>spring-webmvc</artifactId>
21          <version>4.3.6.RELEASE</version>
22      </dependency>
23      <dependency>
24          <groupId>javax.servlet</groupId>
25          <artifactId>javax.servlet-api</artifactId>
26          <version>3.0.1</version>
27      </dependency>
28  </dependencies>
```

Maven Dependencies

Step-2: Create controller class using @Controller annotations (This class acts as Request Handler for client requests). Map request to controller method using @RequestMapping annotation like below

HomeController.java

```java
package com.mvc.controller;

import java.text.DateFormat;

@Controller
public class HomeController {

    @RequestMapping(value = "/home.htm", method = RequestMethod.GET)
    public String home(Locale locale, Model model) {

        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate);
        model.addAttribute("msg", "Welcome to Ashok IT School..!!");

        return "home";
    }

    @RequestMapping(value = "/getDailyQuote.htm", method = RequestMethod.GET)
    public String getDailyQuote(Model model) {
        model.addAttribute("quote", "Don't Drink and Drive..!!");
        return "quoteDisplay";
    }

}
```

HomeController.java

Step-3: Create view files (Using JSP as view Technology)

home.jsp

```jsp
<%@ page session="false"%>
<html>
<head>
<title>Home</title>
</head>
<body>
    <h1>${msg}</h1>
    <P>The time on the server is ${serverTime}.</P>
</body>
</html>
```

home.jsp

quoteDisplay.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Spring MVC App</title>
</head>
<body>
    <h1>Today's Quote : ${quote}</h1>
</body>
</html>
```

quoteDisplay.jsp

Step-4: Configure Web components in WebApplicationContext configuration file

dispatcher-servlet.xml

```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"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <mvc:annotation-driven />

    <context:component-scan base-package="com.mvc.controller" />

    <!-- View Resolver -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

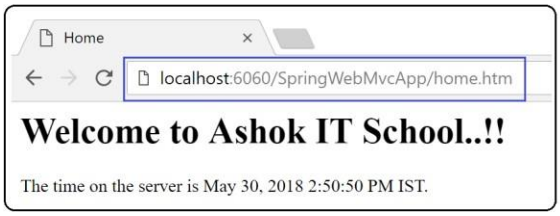
dispatcher-servlet.xml

Step-5: Configuring DispatcherServlet and rootAppliationContext in web.xml file

```
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/rootAppContext.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
```

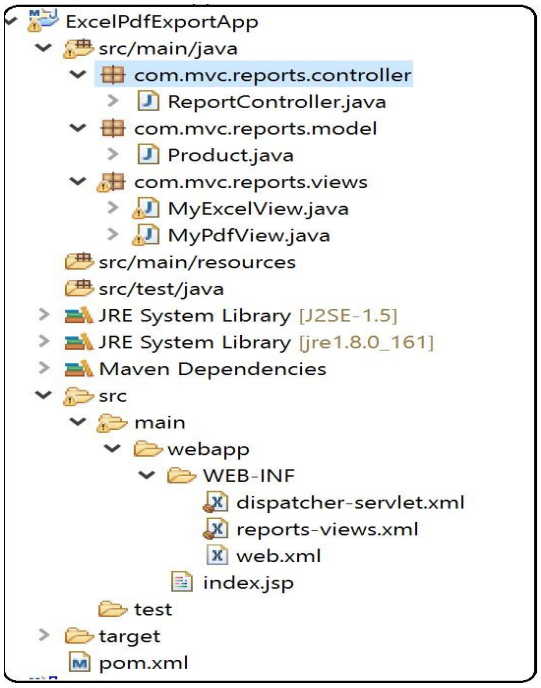
web.xml

Step-6: Deploy the application in Run server and access the controller method using http request



Working with Excel and PDF views

Step-1: Create Maven project and add poi and itext pdf dependencies like below




```
<dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
</dependency>

<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.3.6.RELEASE</version>
</dependency>

<dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.17</version>
</dependency>

<dependency>
        <groupId>com.lowagie</groupId>
        <artifactId>itext</artifactId>
        <version>1.4.8</version>
</dependency>
```

pom.xml

Step-2: Create Model class to present the data

```
package com.mvc.reports.model;

public class Product {

	private Integer pid;
	private String pname;

	public Product(int pid, String name) {
		this.pid = pid;
		this.pname = name;
	}

	//setters & getters
	//toString()
}
```

Product.java

```
package com.mvc.reports.controller;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import com.mvc.reports.model.Product;

@Controller
public class ReportController {

	@RequestMapping("/excel")
	public ModelAndView generateExcel() {
		List<Product> pList = getProductsData();
		return new ModelAndView("excelView", "products", pList);
	}

	@RequestMapping("/pdf")
	public ModelAndView generatePdf() {
		List<Product> pList = getProductsData();
		return new ModelAndView("pdfView", "products", pList);
	}

	private List<Product> getProductsData() {
		List<Product> pList = new ArrayList<Product>();
		pList.add(new Product(501, "Keyboard"));
		pList.add(new Product(502, "Mouse"));
		pList.add(new Product(503, "Hard disk"));
		return pList;
	}

}
```

ReportController.java

```java
package com.mvc.reports.views;
public class MyExcelView extends AbstractXlsView {

	@Override
	protected void buildExcelDocument(Map<String, Object> map, Workbook book,
HttpServletRequest req,
			HttpServletResponse res) throws Exception {

		Sheet sheet = book.createSheet("Products Details");

		Row headerRow = sheet.createRow(0);

		headerRow.createCell(0).setCellValue("S.No");
		headerRow.createCell(1).setCellValue("PID");
		headerRow.createCell(2).setCellValue("PName");

		List<Product> pList = (List) map.get("products");

		if (!pList.isEmpty()) {
			int rowIndex = 1;
			for (Product p : pList) {
				Row dataRow = sheet.createRow(rowIndex);
				dataRow.createCell(0).setCellValue(rowIndex);
				dataRow.createCell(1).setCellValue(p.getPid());
				dataRow.createCell(2).setCellValue(p.getPname());
				rowIndex++;
			}
		}
	}
}
```

MyExcelView.java

```java
package com.mvc.reports.views;

public class MyPdfView extends AbstractPdfView {
	@Override
	protected void buildPdfDocument(Map map, Document doc, PdfWriter writer, HttpServletRequest request,
			HttpServletResponse response) throws Exception {

		List<Product> pList = (List) map.get("products");

		Paragraph p = new Paragraph("Product Details");
		p.setAlignment("center");

		Table t = new Table(3);
		t.setAlignment("center");

		t.addCell("S.No");
		t.addCell("Product ID");
		t.addCell("Product Name");

		if (!pList.isEmpty()) {
			int rowIndex = 1;
			for (Product prod : pList) {
				t.addCell(rowIndex + "");
				t.addCell(prod.getPid() + " ");
				t.addCell(prod.getPname() + " ");
				rowIndex++;
			}
		}
		doc.add(p);
		doc.add(t);
	}
}
```

MyPdfView.java

```xml
<mvc:annotation-driven />

<context:component-scan base-package="com.mvc.reports.controller" />

<bean class="org.springframework.web.servlet.view.XmlViewResolver">
	<property name="location" value="/WEB-INF/reports-views.xml" />
	<property name="order" value="0" />
</bean>

<bean id="viewResolver"
	class="org.springframework.web.servlet.view.InternalResourceViewResolver">
	<property name="prefix" value="/WEB-INF/views/" />
	<property name="suffix" value=".jsp" />
	<property name="order" value="1" />
</bean>
```

dispatcher-Servlet.xml

```xml
<bean id="excelView" class="com.mvc.reports.views.MyExcelView" />
<bean id="pdfView" class="com.mvc.reports.views.MyPdfView" />
```

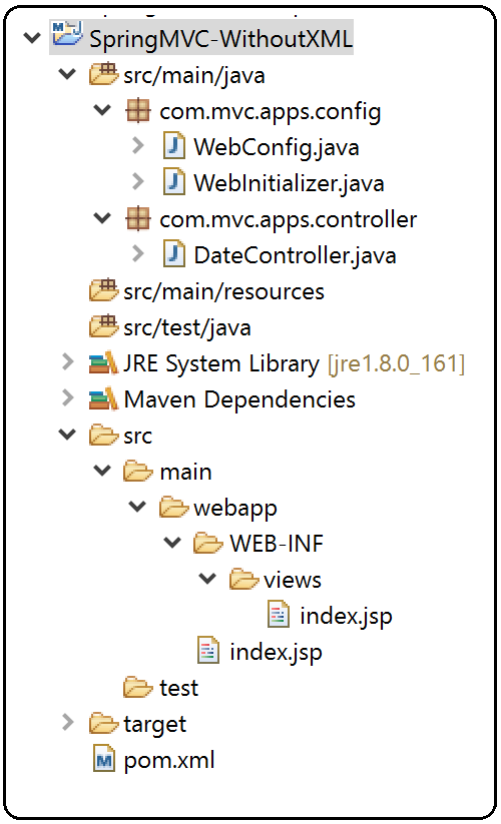
reports-views.xml

```jsp
<html>
<body>
	<a href="excel">Excel</a>
	<br />

	<a href="pdf">PDF</a>
</body>
</html>
```

index.jsp

Spring MVC Application with zero XML

Step-1: Create a Maven web application with below dependencies in Pom.xml for the Required Libraries




```
<properties>
        <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>

<dependencies>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>3.8.1</version>
        </dependency>

        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>3.0.1</version>
        </dependency>

        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>4.3.6.RELEASE</version>
        </dependency>
</dependencies>
```

pom.xml

This Property needs to be configured in the absence of web.xml file

Step-2: Creating a WebConfig Class

As we want to do Java-based configuration, we will create a class called SpringConfig, where we will register all Spring-related beans using Spring's Java-based configuration style.

This class will replace the need to create a SpringApplicationContext.xml file, where we use two important tags

- **<context:component-scan/>**
- **<mvc:annotation-driven/>**

* **Please note that this class has to extend the org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter class**

* **Please note that here, we will use three different annotations at the top level. They will serve the purpose of the XML-based tags used earlier.**

XML Tag	Annotation	Description
<context:component-scan/>	@ComponentScan()	Scan starts from base package and registers all controllers, repositories, service, beans, etc.
<mvc:annotation-driven/>	@EnableWebMvc	Enable Spring MVC-specific annotations like @Controller
Spring config file	@Configuration	Treat as the configuration file for Spring MVC-enabled applications.

```
package com.mvc.apps.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.mvc.apps.controller" })
public class WebConfig {

	@Bean
	public InternalResourceViewResolver createViewResolver() {
		InternalResourceViewResolver vr = new InternalResourceViewResolver();
		vr.setPrefix("/WEB-INF/views/");
		vr.setSuffix(".jsp");
		return vr;
	}
}
```

WebConfig.java

Step 3: Replacing Web.xml

Create another class, which will replace our traditional web.xml. We use Servlet 3.0 and extend the org.springframework.web.WebApplicationInitializer class.

Here we provide our SpringConfig class and add DispatcherServlet, which acts as the FrontController of the Spring MVC application. SpringConfig class is the source of Spring beans, before which we used contextConfiglocation.

```
package com.mvc.apps.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
	@Override
	protected Class<?>[] getRootConfigClasses() {
		return null;
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

WebInitializer.java

Step 4: Create a Controller Class

Now we will create a Controller class, which will take a parameter from request URL and greet a message in the browser.

```
package com.mvc.apps.controller;

import java.util.Date;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloController {

	@RequestMapping("/hello")
	public String display(Model model) {
		model.addAttribute("msg", "Hello, This is produced by HelloController..!!");
		return "index";
	}
}
```

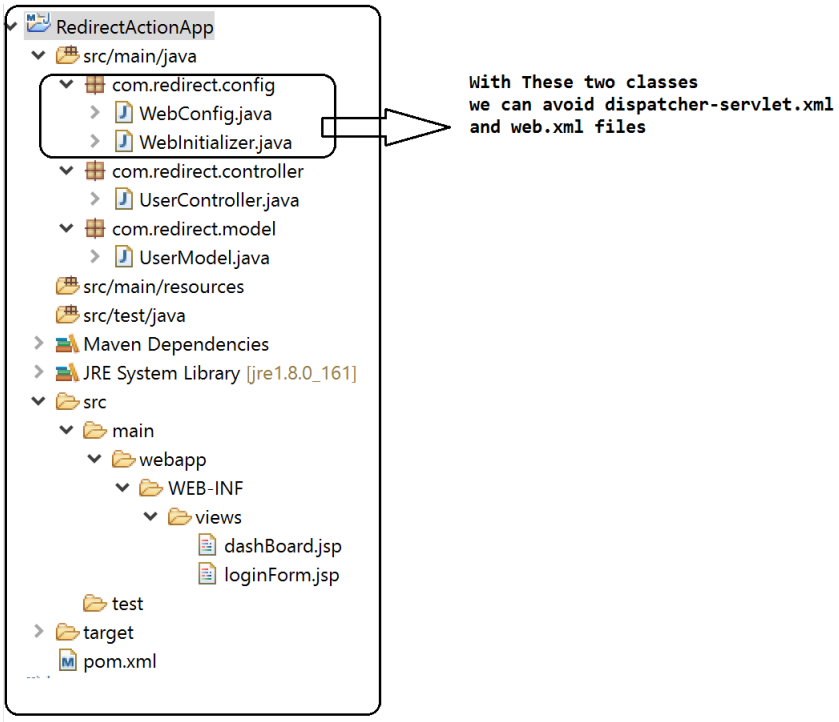
HelloController.java

Step 5: Create a JSP Page to Show the Message

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Spring App</title>
</head>
<body>
	Response : ${msg}
</body>
</html>
```

index.jsp

LogingForm with Redirection – App




```
package com.redirect.config;


@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.redirect.controller" })
public class WebConfig {

	@Bean
	public InternalResourceViewResolver createViewResolver() {
		InternalResourceViewResolver vr = new InternalResourceViewResolver();
		vr.setPrefix("/WEB-INF/views/");
		vr.setSuffix(".jsp");
		return vr;
	}
}
```

```
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
	@Override
	protected Class<?>[] getRootConfigClasses() {
		return null;
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

```java
public class UserModel {

	private String uname;
	private String pwd;

	//setters & getters
}
```

```java
package com.redirect.controller;

@Controller
public class UserController {

	@RequestMapping("/loginForm")
	public String displayLoginForm(Model model) {
		model.addAttribute("u", new UserModel());
		return "loginForm";
	}

	@RequestMapping(value = "/login", method = RequestMethod.POST)
	public String login(@ModelAttribute UserModel um, Model model) {
		if (um.getUname().equals("admin") && um.getPwd().equals("admin123")) {
			// Login successfull
			return "redirect:/buildDashboard";
		} else {
			// login failure
			model.addAttribute("u", new UserModel());
			model.addAttribute("errMsg", "Invalid Credentials");
			return "loginForm";
		}
	}

	@RequestMapping("/buildDashboard")
	public String dashboard(Model model) {
		model.addAttribute("msg", "Reports generating...!!");
		return "dashBoard";
	}
}
```

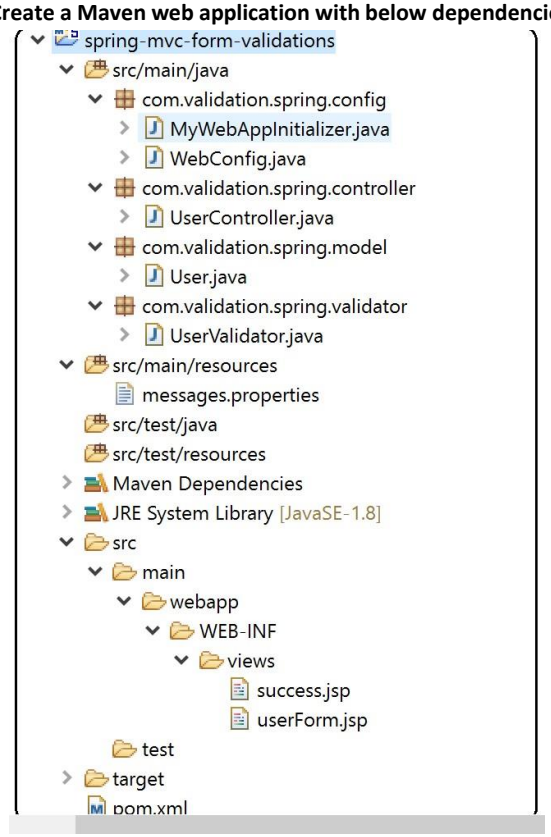
UserController.java

```jsp
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<h2>Login Here</h2>
	<b style="color: red">${errMsg}</b>
	<form:form action="login" method="post" modelAttribute="u">
		<table>
			<tr>
				<td>Username :</td>
				<td><form:input path="uname" /></td>
			</tr>
			<tr>
				<td>Password :</td>
				<td><form:password path="pwd" /></td>
			</tr>
			<tr>
				<td></td>
				<td><input type="submit" value="Login" /></td>
			</tr>
		</table>
	</form:form>
</body>
</html>
```

login.jsp

Spring MVC Application with Form validations

Step-1: Create a Maven web application with below dependencies in pom.xml for the Required Libraries



Step-2: Creating a WebConfig Class

WebConfig.java

```
1  package com.validation.spring.config;
2
3  import org.springframework.context.MessageSource;
12
13 @Configuration
14 @EnableWebMvc
15 @ComponentScan(basePackages = { "com.validation.spring.controller", "com.validation.spring.validator" })
16 public class WebConfig extends WebMvcConfigurerAdapter {
17
18     @Bean
19     public InternalResourceViewResolver resolver() {
20         InternalResourceViewResolver resolver = new InternalResourceViewResolver();
21         resolver.setViewClass(JstlView.class);
22         resolver.setPrefix("/WEB-INF/views/");
23         resolver.setSuffix(".jsp");
24         return resolver;
25     }
26
27     @Bean
28     public MessageSource messageSource() {
29         ResourceBundleMessageSource source = new ResourceBundleMessageSource();
30         source.setBasename("messages");
31         return source;
32     }
33 }
```

Step 3: Replacing Web.xml

MyWebAppInitializer.java

```java
package com.validation.spring.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] {};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Step-4 : Create messages.properties file for validation messages

messages.properties

```
user.name.empty = Enter a valid name.
user.email.empty = Enter a valid email.
user.email.invalid = Invalid email! Please enter valid email.
user.gender.empty = Select gender.
user.languages.empty = Select at least one language.
```

Step-5: Create a Model class for holding form data

User.java

```java
package com.validation.spring.model;

import java.util.List;

public class User {
    private String name;
    private String email;
    private String gender;
    private List<String> languages;

    // setters & getters

}
```

Step-6: Create Validator class by implementing Validator interface

UserValidator.java

```
1  package com.validation.spring.validator;
2
3  import java.util.regex.Pattern;
11
12 @Component
13 public class UserValidator implements Validator {
14
15     @Override
16     public boolean supports(Class<?> clazz) {
17         return User.class.equals(clazz);
18     }
19
20     @Override
21     public void validate(Object obj, Errors err) {
22
23         ValidationUtils.rejectIfEmpty(err, "name", "user.name.empty");
24         ValidationUtils.rejectIfEmpty(err, "email", "user.email.empty");
25         ValidationUtils.rejectIfEmpty(err, "gender", "user.gender.empty");
26         ValidationUtils.rejectIfEmpty(err, "languages", "user.languages.empty");
27
28         User user = (User) obj;
29
30         Pattern pattern = Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}$", Pattern.CASE_INSENSITIVE);
31         if (!(pattern.matcher(user.getEmail()).matches())) {
32             err.rejectValue("email", "user.email.invalid");
33         }
34     }
35 }
```

Step 7: Create a Controller Class

UserController.java

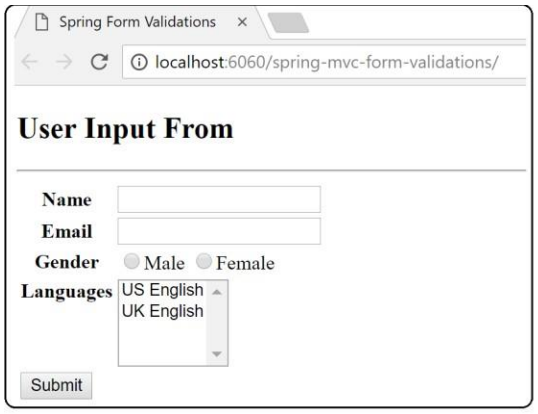
```
1  package com.validation.spring.controller;
2
3  import java.util.Locale;
18
19 @Controller
20 public class UserController {
21
22     @Autowired
23     private UserValidator userValidator;
24
25     @InitBinder
26     protected void initBinder(WebDataBinder binder) {
27         binder.addValidators(userValidator);
28     }
29
30     @GetMapping("/")
31     public String userForm(Locale locale, Model model) {
32         model.addAttribute("user", new User());
33         return "userForm";
34     }
35
36     /*
37      * Save user object
38      */
39     @PostMapping("/saveUser")
40     public String saveUser(@ModelAttribute("user") @Validated User user, BindingResult result, Model model) {
41         if (result.hasErrors()) {
42             return "userForm";
43         }
44         return "success";
45     }
46 }
```

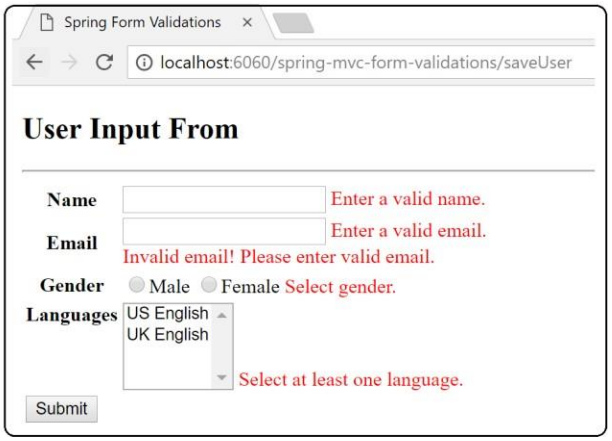
Step-8: Create jsp form for user input

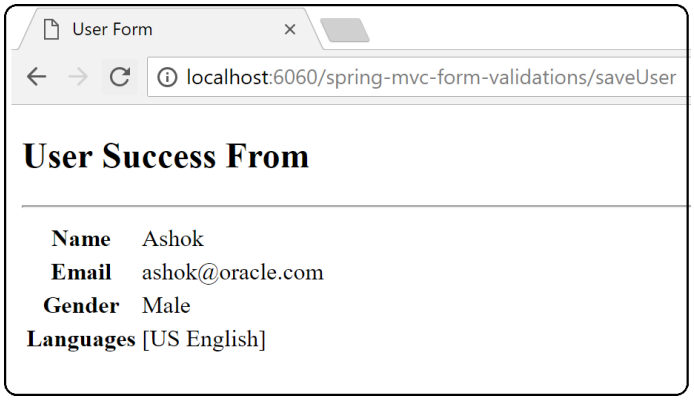
```
<style type="text/css">
.error {
        color: red;
}
</style>
</head>
<body>
        <h2>User Input From</h2>
        <hr />
        <form:form action="saveUser" method="post" modelAttribute="user">
                <table>
                        <tr>
                                <th>Name</th>
                                <td><form:input path="name" /> <form:errors path="name" cssClass="error" /></td>
                        </tr>
                        <tr>
                                <th>Email</th>
                                <td><form:input path="email" /> <form:errors path="email" cssClass="error" /></td>
                        </tr>
                        <tr>
                                <th>Gender</th>
                                <td><form:radiobutton path="gender" value="Male" label="Male" />
                                        <form:radiobutton path="gender" value="Female" label="Female" /> <form:errors
                                                path="gender" cssClass="error" /></td>
                        </tr>
                        <tr>
                                <th valign="top">Languages</th>
                                <td><form:select path="languages" multiple="true">
                                                <form:option value="US English">US English</form:option>
                                                <form:option value="UK English">UK English</form:option>
                                        </form:select>
                                        <form:errors path="languages" cssClass="error" />
                                </td>
                        </tr>
                        <tr>
                                <td><button type="submit">Submit</button></td>
                        </tr>
                </table>
        </form:form>
```

userForm.jsp

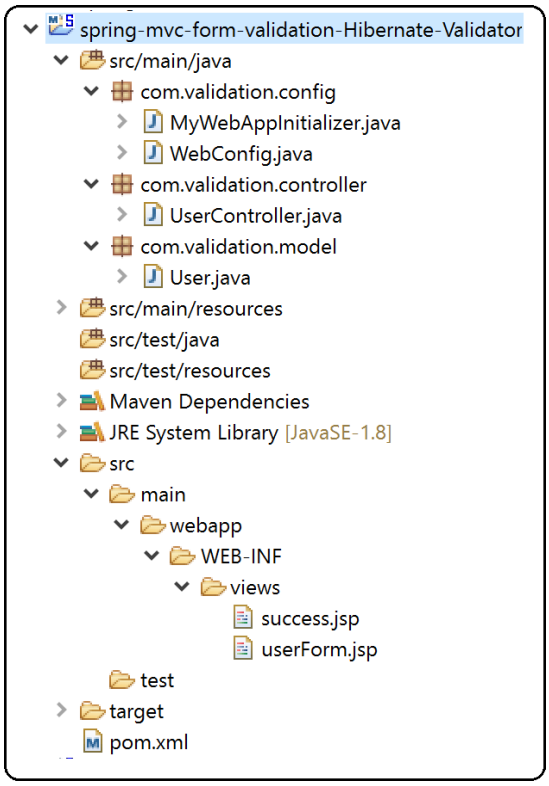
Step-9: Deploy the application and test it







Form Validations using Hibernate Validator - Application




```
<properties>
        <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>
<dependencies>
        <!-- Spring MVC Dependency -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>4.3.7.RELEASE</version>
        </dependency>

        <!-- Hibernate Validator -->
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-validator</artifactId>
                <version>5.4.1.Final</version>
        </dependency>

        <!-- JSTL Dependency -->
        <dependency>
                <groupId>javax.servlet.jsp.jstl</groupId>
                <artifactId>javax.servlet.jsp.jstl-api</artifactId>
                <version>1.2.1</version>
        </dependency>

        <dependency>
                <groupId>taglibs</groupId>
                <artifactId>standard</artifactId>
                <version>1.1.2</version>
        </dependency>

        <!-- Servlet Dependency -->
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>3.1.0</version>
                <scope>provided</scope>
        </dependency>

        <!-- JSP Dependency -->
        <dependency>
                <groupId>javax.servlet.jsp</groupId>
                <artifactId>javax.servlet.jsp-api</artifactId>
                <version>2.3.1</version>
                <scope>provided</scope>
        </dependency>

</dependencies>
```

pom.xml

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.validation.controller" })
public class WebConfig extends WebMvcConfigurerAdapter {

	@Bean
	public InternalResourceViewResolver resolver() {
		InternalResourceViewResolver resolver = new InternalResourceViewResolver();
		resolver.setViewClass(JstlView.class);
		resolver.setPrefix("/WEB-INF/views/");
		resolver.setSuffix(".jsp");
		return resolver;
	}

	@Bean
	public MessageSource messageSource() {
		ResourceBundleMessageSource source = new ResourceBundleMessageSource();
		source.setBasename("messages");
		return source;
	}

	@Override
	public Validator getValidator() {
		LocalValidatorFactoryBean validator = new LocalValidatorFactoryBean();
		validator.setValidationMessageSource(messageSource());
		return validator;
	}

}
```

WebConfig.java

```
package com.validation.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

	@Override
	protected Class<?>[] getRootConfigClasses() {
		return new Class[] {};
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

MyWebAppInitializer.java

```
user.name.empty = Name entered is invalid. It must be
between {2} and {1} characters.
user.email.invalid = Invalid email! Please enter valid email.
user.gender.empty = Select gender.
user.languages.empty = Select at least one language.
```

messages.properties

```
public class User {

		@Size(max = 20, min = 3, message = "{user.name.empty}")
		private String name;

		@Email(message = "{user.email.invalid}")
		private String email;

		@NotEmpty(message = "{user.gender.empty}")
		private String gender;

		@NotEmpty(message = "{user.languages.empty}")
		private List<String> languages;

		//setters & getters
}
```

User.java

```
@Controller
public class UserController {

	@GetMapping("/")
	public String userForm(Locale locale, Model model) {
		model.addAttribute("user", new User());
		return "userForm";
	}

	@PostMapping("/saveUser")
	public String saveUser(@ModelAttribute("user") @Valid User user, BindingResult result, Model model) {

		if (result.hasErrors()) {
			return "userForm";
		}

		return "success";
	}
}
```

UserController.java

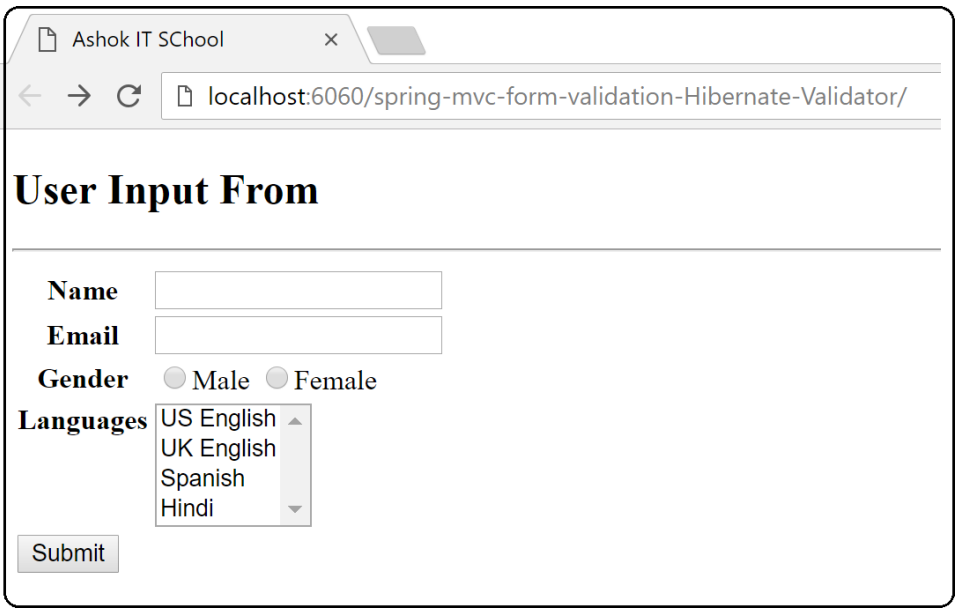
```
<body>
	<h2>User Input From</h2>
	<hr />
	<form:form action="saveUser" method="post" modelAttribute="user">
		<table>
			<tr>
				<th>Name</th>
				<td><form:input path="name" /> <form:errors path="name" cssClass="error" /></td>
			</tr>
			<tr>
				<th>Email</th>
				<td><form:input path="email" /> <form:errors path="email"  cssClass="error" /></td>
			</tr>
			<tr>
				<th>Gender</th>
				<td><form:radiobutton path="gender" value="Male" label="Male" />
					<form:radiobutton path="gender" value="Female" label="Female" />
					<form:errors   path="gender" cssClass="error" />
				</td>
			</tr>

		</table>
	</form:form>
</body>
```

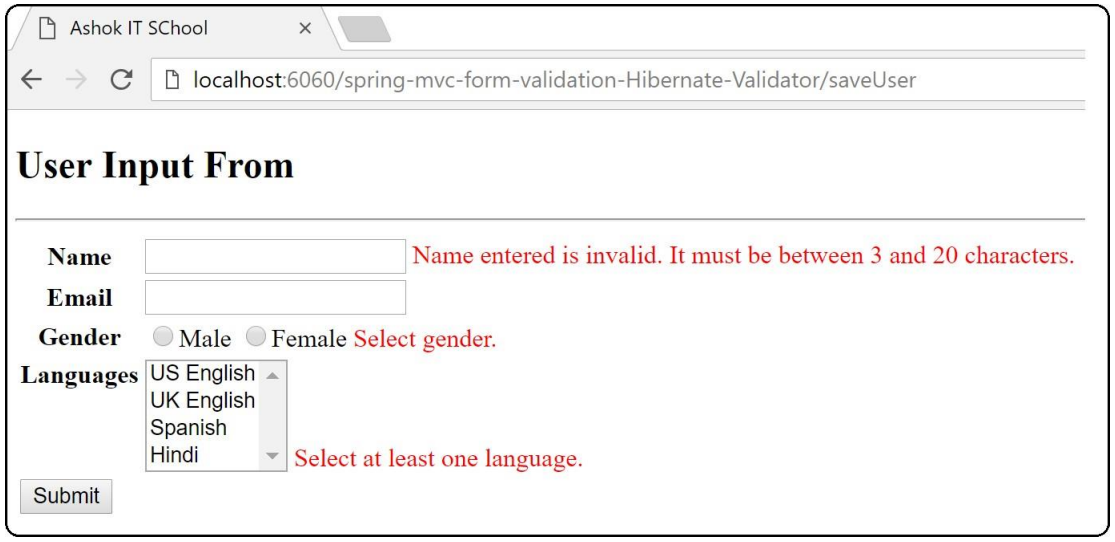
```
<tr>
	<th valign="top">Languages</th>
	<td><form:select path="languages" multiple="true">
			<form:option value="US English">US English</form:option>
			<form:option value="UK English">UK English</form:option>
			<form:option value="Spanish">Spanish</form:option>
			<form:option value="Hindi">Hindi</form:option>
	</form:select> <form:errors path="languages" cssClass="error" />
	</td>
</tr>
<tr>
	<td><button type="submit">Submit</button></td>
</tr>
```

userForm.jsp

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Ashok IT School</title>
</head>
<body>
 <h2>User Success From</h2>
 <hr />

 <table>
  <tr>
   <th>Name</th>
   <td>${user.name}</td>
  </tr>
  <tr>
   <th>Email</th>
   <td>${user.email}</td>
  </tr>
  <tr>
   <th>Gender</th>
   <td>${user.gender}</td>
  </tr>
  <tr>
   <th valign="top">Languages</th>
   <td>${user.languages}</td>
  </tr>
 </table>
</body>
</html>
```

success.jsp





Adding HandlerInterceptors

Spring's handler mapping mechanism has a notion of handler interceptors, that can be extremely useful when you want to apply specific functionality to certain requests, for example, checking for a principal.

Interceptors located in the handler mapping must implement HandlerInterceptor from theorg.springframework.web.servlet package. This interface defines three methods, one that will be called *before* the actual handler will be executed, one that will be called *after* the handler is executed, and one that is called *after the complete request has finished*. These three methods should provide enough flexibility to do all kinds of pre- and post-processing.
The preHandle method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue, when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.interceptor.controller" })
public class WebConfig extends WebMvcConfigurerAdapter {

	@Bean
	public ViewResolver createViewResolver() {
		InternalResourceViewResolver res = new InternalResourceViewResolver();
		res.setPrefix("/WEB-INF/views/");
		res.setSuffix(".jsp");
		return res;
	}

	@Override
	public void addInterceptors(InterceptorRegistry registry) {
		registry.addInterceptor(new ExecutionTimeInterceptor());
		registry.addInterceptor(new BrowserCheckInterceptor());
	}
}
```

WebConfig.java

```
public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

	@Override
	protected Class<?>[] getRootConfigClasses() {
		return null;
	}

	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

WebInitializer.java

```
public class BrowserCheckInterceptor extends HandlerInterceptorAdapter {

	@Override
	public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
			throws Exception {
		String browserName = request.getHeader("user-agent");

		if (browserName.contains("Chrome")) {
			response.sendRedirect("invalidBrowser.jsp");
			return false;
		}
		return true;
	}
}
```

BrowserCheckInterceptor.java

```
public class ExecutionTimeInterceptor extends HandlerInterceptorAdapter {

	@Override
	public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
			throws Exception {
		long startTime = System.currentTimeMillis();
		request.setAttribute("start", startTime);
		return true;
	}

	@Override
	public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
			ModelAndView modelAndView) throws Exception {
		long endTime = System.currentTimeMillis();
		long startTime = (Long) request.getAttribute("start");
		long diff = endTime - startTime;
		System.out.println("Time taken : " + diff);
	}

	@Override
	public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
			throws Exception {
		System.out.println("after completion");
	}
}
```

ExecutionTimeInterceptor.java

```
@Controller
public class WelcomeController {

	@RequestMapping("/welcome")
	public String welcome(Model model) {
		model.addAttribute("msg", "Welcome to My Project...!!");
		return "welcomeFile";
	}

	@RequestMapping("/wish")
	public String wish(Model model) {
		model.addAttribute("msg", "Good Morning...!!");
		for (int i = 0; i <= 100; i++) {
			// TODO:
		}
		return "wishFile";
	}
}
```

WelcomeController.java

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<h1>${msg}</h1>
</body>
</html>
```

welcomeFile.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<h1>Please use only IE browser to access this project..!!</h1>
</body>
</html>
```

invalidBrowser.jsp

Spring Form Tag Library

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). The Spring Web MVC framework provides a set of tags in the form of a tag library, which is used to construct views (web pages). The Spring Web MVC integrates spring's form tag library. Spring's form tag accesses to the command object, and also it refers to the data our spring controller deals with. A Command object can be defined as a JavaBean that stores user input, usually entered through HTML form is called the Command object or Model Object. The spring form tag makes it easier to develop, maintain, and read JSPs. The spring form tags are used to construct user interface elements such as text and buttons. Spring form tag library has a set of tags such as <form> and <input>. Each form tag provides support for the set of attributes of its corresponding HTML tag counterpart, which allows a developer to develop UI components in JSP or HTML pages.

Configuration – spring-form.tld

The spring form tag library comes bundled in spring-webmvc.jar. The springform.tld is known as Tag Library Descriptor (tld) file, which is available in a web application and generates HTML tags. The spring form tag library must be defined at the top of the JSP page. The following directive needs to be added to the top of your JSP pages, in order to use spring form tags from this library:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

Here, form is the tag name prefix, which will be used for the tags from this spring form tag library in JSP pages.

The following table shows few important tags of the spring form tag library:

Tag	Description
form:form	Generates the HTML <form> tag. It has the name attribute that specifies the command object that the inner tags should bind to.
form: input	Represents the HTML input text tag.
form:password	Represents the HTML input password tag.
form:radiobutton	Represents the HTML input radio button tag.
form: checkbox	Represents the HTML input checkbox tag.
form:select	Represents the HTML select list tag.
form:options	Represents the HTML options tag.
form:errors	Represents the HTML span tag. It also generates span tag from the error created as a result of validations.

The difference between Html tags and spring form tags

Tags	HTML	Spring
Input	<input type = "text">	<form:input>
Radiobutton	<input type="radiobutton">	<form:radiobutton> <form:radiobuttons> Specify list of values for radiobuttons in one shot
Checkbox	<input type="checkbox">	<form:checkbox> <form:checkboxes> Specify list of values for checkboxes in one shot
Password	<input type="password">	<form:password>
Select and option	<select name="course"> <option value="java">java</option> <option value="spring">spring</option> </select>	<form:select name="course"> <form:option value="java" label="java"/> <form:option value="spring" label="spring"/> </form:select> <form:options> Specify the list of options in one shot
Text area	<input type="textarea">	<form:textarea>
Hidden	<input type="hidden">	<form:hidden>

The <form: form> tag

The <form: form> tag is used to generate an HTML <form: form> tag in any of the JSP pages of a Spring application. It is used for binding the inner tags, which means that all the other tags are the nested tags in the <form:form> tag. Using this <form: form> tag, the inner tags can access the Command object, which resides in the JSP's PageContext class.

```
<form:form method="POST" modelAttribute="user" action="register">
        ……..
</form: form>
```

The commandName & modelAttribute attribute

The attributes commandName and modelAttribute on the form:form tag do primarily the same thing, which is to map the form's fields to an Object of some type in the Controller. I believe modelAttribute is the preferred method, and commandName is only there for backwards compatibility.

The <form: input>tag: The <form: input> tag is used for entering the text by the user in any of the JSP pages of the Spring web application. The following code snippet shows the use of the <form: input> tag in JSP pages:

```
<form:form method="POST" modelAttribute="user" action="register">
    <table>
      <tr>
        <td>Enter your name:</td>
        <td><form:input path="name" /></td>
      </tr>
      <tr>
        <td>Enter your mail:</td>
        <td><form:input path="email" /></td>
      </tr>
    </table>
</form:form>
```

The path attribute

The <form: input> tag renders an HTML <input type="text"/> element. The path attribute is the most important attribute of the input tag. This path attribute binds the input field to the form-backing object's property.

Let's take an example, if user is assigned as modelAttribute of the enclosing <form/> tag, then the path attribute of the input tag will be given as name or email. It should be noted that the User class contains getter and setter for name and email properties.

The <form:checkbox> tag

The <form:checkbox> tag is same as the HTML <input> tag, which is of the checkbox type.

```
<form:checkbox path="skills" value="Excel" label="Excel"/>
<form:checkbox path="skills" value="Word" label="Word"/>
<form:checkbox path="skills" value="Powerpoint"  label="Powerpoint"/>
```

In the preceding code snippet, the User class property called skills is used in the checkbox option. If the skills checkbox is checked, the User class's skills property is set accordingly.

The <form: radiobutton> tag

The <form: radiobutton> tag is used to represent the HTML <input> tag with the radio type in any of the JSP pages of a Spring web application. It is used when there are many tag instances having the same property with different values, and only one radio value can be selected at a time. The following code snippet shows how we use the <form:radiobutton> tag in JSP pages:

```
<tr>
  <td>Gender:</td>
  <td>
    Male: <form:radiobutton path="gender" value="male"  label="male"/>
    Female: <form:radiobutton path="gender" value="female"  label="female"/>
  </td>
</tr>
```

The <form : password> tag

The <form: password> tag is used to represent the HTML <input> tag of the password type, in any of the JSP pages of a Spring web application. By default, the browser does not show the value of the password field. We can show the value of the password field by setting the showPassword attribute to true. The following code snippet shows how we use the <form: password> tag in the JSP page:

```
<tr>
  <td>Password :</td>
  <td>
     <form:password path="password" />
  </td>
</tr>
```

The <form: select> tag The <form: select> tag is used to represent an HTML <select> tag in any of the JSP pages of a Spring application. We use the <form: select> tag for binding the selected option with its value. The <form: option> tag is the nested tag in the <form:select> tag. The following code snippet shows how we use the <select> tag in the JSP pages:

```
<tr>
  <td>Department</td>
  <td>
     <form:select path="department" items="${departmentMap}"  />
  </td>
</tr>
```

Note: Here departmentMap is collection with multiple keys and values

The <form:option> tag

The <form: option> tag is used to represent an HTML <option> tag in any of the JSP pages of a Spring application. This tag is used when we need to add all the options to be <form: select> tag. Here, we need to add all the options inside the <form: select> tag. The following code snippet shows how to use the <option> tag in a JSP page:

```
<tr>
  <td>Department</td>
  <td><form:select path="department">
    <form:option value="technical" label="Technical" />
    <form:option value="non_technical" label="Non Technical" />
    <form:option value="r&d" label="R & D" />
  </form:select></td>
</tr>
```

The alternative code is below (with collection object)

```
<tr>
  <td>Department</td>
  <td><form:select path="department">
    <form:options items="${departmentMap}" />
  </form:select></td>
</tr>
```

The <form: textarea> tag

The <form: textarea> tag is used to represent an HTML <textarea> tag in any of the JSP pages of a Spring application. The following code snippet shows how to use the <form: textarea> tag in a JSP pages:

```
<td>Remarks :</td>
<td>
  <form:textarea path="remarks" rows="3"  cols="20"></form:textarea>
</td>
```

The <form: hidden> tag

The <form: hidden> tag is used to represent an HTML hidden field in a JSP page of a Spring application. The following code snippet shows how we use the <hidden> tag in JSP pages:

```
<form:hidden path="empId" />
```

The <form: errors> tag

The <form: errors> tag is used to represent an HTML errors in a JSP of a

Spring application. This tag is used for accessing the error defined in an org. springframework.validation.Validator interface. For example, if we want to submit a form, and find all the validator error related to the name and password fields in that form, we have to define all the validation errors related to these fields in a Validator class, which must implements the Validator interface as follows:

```
<form:form method="POST" modelAttribute="user" action="register">
    <table>
      <tr>
        <td>Enter your name:</td>
        <td><form:input path="name" /></td>
        <td><form:errors path="name" cssStyle="color: #ff0000;"/></td>
      </tr>
      <tr>
        <td>Enter your mail:</td>
        <td><form:input path="email" /></td>
        <td><form:errors path="email" cssStyle="color: #ff0000;" /></td>
      </tr>
      <tr>
        <td>Select your gender</td>
        <td><form:radiobuttons path="gender" items="${genders}" /></td>
        <td><form:errors path="gender" cssStyle="color: #ff0000;"/></td>
      </tr>
    </table>
</form>
```

Registration Form using Spring Form Tags

Step 1: Create one Maven Project like below

Project Structure

```
<properties>
        <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>
<dependencies>
        <!-- Spring MVC Dependency -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>4.3.7.RELEASE</version>
        </dependency>

        <!-- Servlet Dependency -->
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>3.1.0</version>
                <scope>provided</scope>
        </dependency>

  </dependencies>
```

```
<!-- JSTL Dependency -->
<dependency>
        <groupId>javax.servlet.jsp.jstl</groupId>
        <artifactId>javax.servlet.jsp.jstl-api</artifactId>
        <version>1.2.1</version>
</dependency>
<dependency>
        <groupId>taglibs</groupId>
        <artifactId>standard</artifactId>
        <version>1.1.2</version>
</dependency>

<!-- JSP Dependency -->
<dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
</dependency>
```

pom.xml

Step-2: Creating a WebConfig Class

WebConfig.java

```
1  package com.app.config;
2
3  import org.springframework.context.annotation.Bean;
10
11 @Configuration
12 @EnableWebMvc
13 @ComponentScan(basePackages = { "com.app.controller" })
14 public class WebConfig extends WebMvcConfigurerAdapter {
15
16     @Bean
17     public InternalResourceViewResolver resolver() {
18         InternalResourceViewResolver resolver = new InternalResourceViewResolver();
19         resolver.setViewClass(JstlView.class);
20         resolver.setPrefix("/WEB-INF/views/");
21         resolver.setSuffix(".jsp");
22         return resolver;
23     }
24 }
25
```

WebConfig.java

Step-3: Create WebInitializer.java class

WebConfig.java | MyWebAppInitializer.java

```
1  package com.app.config;
2
3  import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
4
5  public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
6
7      @Override
8      protected Class<?>[] getRootConfigClasses() {
9          return new Class[] {};
10     }
11
12     @Override
13     protected Class<?>[] getServletConfigClasses() {
14         return new Class[] { WebConfig.class };
15     }
16
17     @Override
18     protected String[] getServletMappings() {
19         return new String[] { "/" };
20     }
21 }
22
```

MyWebAppInitializer.java

Step 4 : Create User.java (model class) like below

User.java

```
1  package com.app.model;
2
3  import java.util.List;
4
5  public class User {
6
7      private String name;
8
9      private String email;
10
11     private String gender;
12
13     private String password;
14
15     private String passwordConfirm;
16
17     private List<String> courses;
18
19     private List<String> batches;
20
21     private String hiddenMsg;
22
23     //setters && getters
24
25 }
```

User.java

Step-5 : Create RegistrationController class to handle the request

In the controller we have added method to display the Registration page and success page on submit.
We have also written a method to initialize the form with all the required values.

RegistrationController.java

```
1  package com.app.controller;
2
3  import java.util.ArrayList;
13
14 @Controller
15 public class RegistrationController {
16
17     @RequestMapping(value = "/", method = RequestMethod.GET)
18     public String displayUserPage(Model model) {
19         User user = new User();
20         user.setHiddenMsg("Ashok IT School");
21         model.addAttribute("user", user);
22         //to set form default values into model
23         initializeFormValues(model);
24         return "registration";
25     }
26
27     @RequestMapping(value = "/register", method = RequestMethod.POST)
28     public String displayUserDetails(@ModelAttribute User user, Model model) {
29         model.addAttribute("user", user);
30         return "success";
31     }
32
33     private void initializeFormValues(Model model) {
34         List<String> courses = new ArrayList<String>();
35         courses.add("JSE"); courses.add("J2EE"); courses.add("Spring"); courses.add("Hibernate");
36         courses.add("RESTful Services");
37         model.addAttribute("courses", courses);
38
39         List<String> genders = new ArrayList<String>();
40         genders.add("Male"); genders.add("Female");
41         model.addAttribute("genders", genders);
42
43         List<String> batches = new ArrayList<String>();
44         batches.add("Morning"); batches.add("Afternoon");
45         batches.add("Evening");
46
47         model.addAttribute("batches", batches);
48
49     }
50 }
```

Step 6: Create registration.jsp page to display registration form with input fields like below

- **path is the spring form tag attribute used to bind the form field with the model class.**
- **<form: errors> tag is used to specify error to be displayed for the corresponding field.**
- **We can specify the list of strings directly using items as we used in < form: checkboxes> and < form:select>**
- **List specified with items is used to display multiple values and path is used to bind the selected value into the model class variable.**

----------registration.jsp----------

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
<title>Registraion</title>
</head>
<body>
        <h2>Register Here</h2>
        <hr />
        <form:form method="POST" modelAttribute="user" action="register">
```

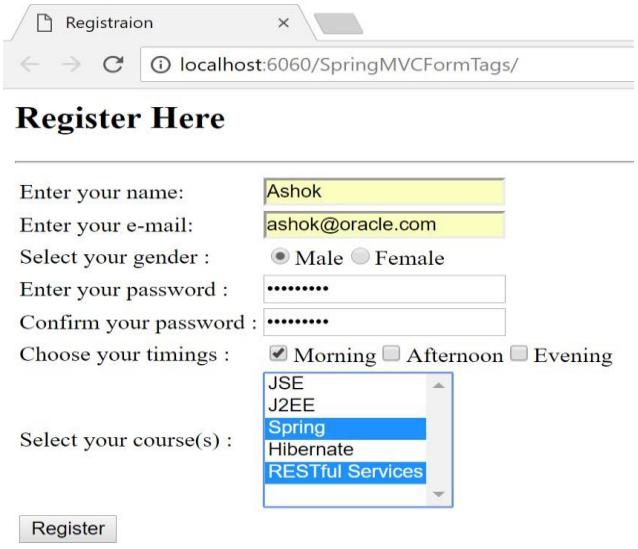
```
		<table>
			<tr>
				<td>Enter your name:</td>
				<td><form:input path="name" /></td>

			<td><form:errors path="name" cssStyle="color: #ff0000;" /></td>
			</tr>
			<tr>
				<td>Enter your e-mail:</td>
				<td><form:input path="email" /></td>
				<td><form:errors path="email" cssStyle="color: #ff0000;" /></td>
			</tr>
			<tr>
				<td>Select your gender :</td>
				<td><form:radiobuttons path="gender" items="${genders}" /></td>
				<td><form:errors path="gender" cssStyle="color: #ff0000;" /></td>
			</tr>
			<tr>
				<td>Enter your password :</td>
				<td><form:password path="password" showPassword="true" /></td>
				<td><form:errors path="password" cssStyle="color: #ff0000;" /></td>
			<tr>
				<td>Confirm your password :</td>
				<td><form:password path="passwordConfirm" showPassword="true" /></td>
				<td><form:errors path="passwordConfirm"
					cssStyle="color: #ff0000;" /></td>
			</tr>
			<tr>
				<td>Choose your timings :</td>
				<td><form:checkboxes path="batches" items="${batches}" /></td>
				<td><form:errors path="batches" cssStyle="color: #ff0000;" /></td>
			</tr>
			<tr>
				<td>Select your course(s) :</td>
				<td><form:select path="courses" size="6">
					<form:options items="${courses}" />
				</form:select></td>
				<td><form:errors path="courses" cssStyle="color: #ff0000;" /></td>
			</tr>
			<tr>
				<form:hidden path="hiddenMsg" />
			</tr>
			<tr>
				<td><input type="submit" name="submit" value="Register"></td>
			</tr>
			<tr>
		</table>
	</form:form>
</body>
```

Step 7 : Create success.jsp page to display captured form data as a response

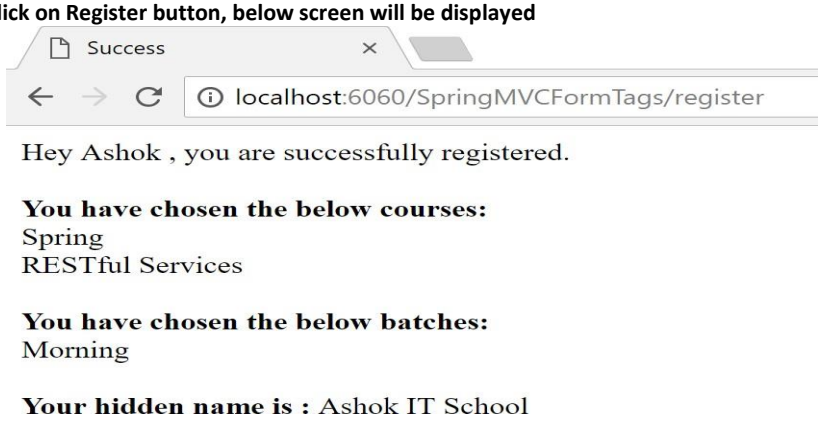
success.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Success</title>
</head>
<body>
    Hey ${user.name} , you are successfully registered.
    <br />
    <b>You have chosen the below courses: </b>
    <br>
    <c:forEach var="course" items="${user.courses}">
        <c:out value="${course}" />
        <br>
    </c:forEach>
    <b>You have chosen the below batches:</b>
    <c:forEach var="batch" items="${user.batches}">
        <c:out value="${batch}" />
        <br>
    </c:forEach>
    <br>
    <b>Your hidden name is : </b> ${user.hiddenMsg}
</body>
</html>
```

Step 8: Deploy the application into server and access using below URL and fill the form

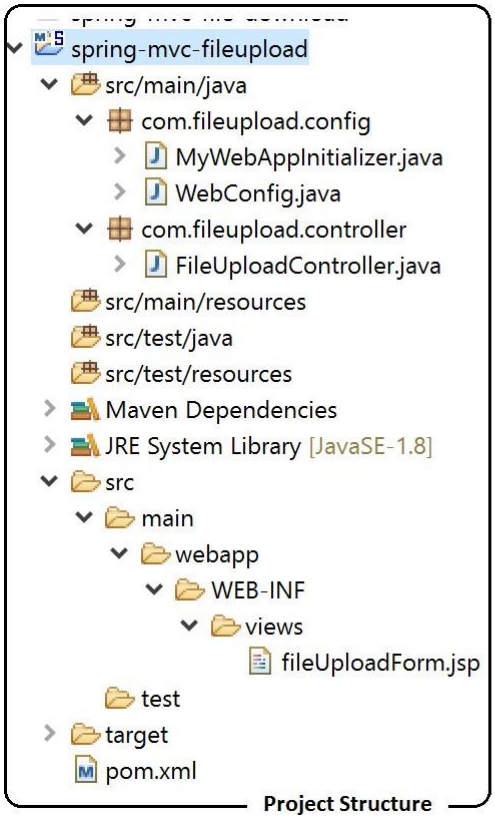


Step 9: Click on Register button, below screen will be displayed



Spring MVC File(s) Uploading - Application

Step-1: Create Maven Project and add dependencies



Project Structure

```
<properties>
    <failOnMissingWebXml>false</failOnMissingWebXml>
</properties>
<dependencies>
    <!-- Spring MVC Dependency -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.3.6.RELEASE</version>
    </dependency>

    <!-- Servlet Dependency -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>

    <!-- JSP Dependency -->
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
    </dependency>

</dependencies>
```

Maven Dependencies

Step-2: Create WebConfig class

```
package com.fileupload.config;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.fileupload.controller" })
public class WebConfig extends WebMvcConfigurerAdapter {

        @Bean
        public InternalResourceViewResolver resolver() {
                InternalResourceViewResolver resolver = new InternalResourceViewResolver();
                resolver.setViewClass(JstlView.class);
                resolver.setPrefix("/WEB-INF/views/");
                resolver.setSuffix(".jsp");
                return resolver;
        }

        @Bean
        public MultipartResolver multipartResolver() {
                StandardServletMultipartResolver multipartResolver = new StandardServletMultipartResolver();
                return multipartResolver;
        }

}
```

WebConfig.java

Step-3: Create Web Initializer class

```
package com.fileupload.config;

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
	@Override
	protected Class<?>[] getRootConfigClasses() {
		return new Class[] {};
	}
	@Override
	protected Class<?>[] getServletConfigClasses() {
		return new Class[] { WebConfig.class };
	}

	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
	@Override
	protected void customizeRegistration(Dynamic registration) {

		// Parameters:-
		// location - the directory location where files will be stored
		// maxFileSize - the maximum size allowed for uploaded files
		// maxRequestSize - the maximum size allowed for multipart/form-data
		// requests
		// fileSizeThreshold - the size threshold after which files will be
		// written to disk
		MultipartConfigElement multipartConfig = new MultipartConfigElement("C:\\Desktop\\File", 1048576, 10485760, 0);
		registration.setMultipartConfig(multipartConfig);
	}
}
```

MyWebAppInitializer.java

Step-4: Create Controller class to handle file upload request

FileUploadController.java

```
1  package com.fileupload.controller;
2
3  import java.io.BufferedOutputStream;
14
15 @Controller
16 public class FileUploadController {
17
18     @GetMapping("/")
19     public String fileUploadForm(Model model) {
20         return "fileUploadForm";
21     }
22     // Handling single file upload request
23     @PostMapping("/singleFileUpload")
24     public String singleFileUpload(@RequestParam("file") MultipartFile file, Model model) throws IOException {
25         // Save file on system
26         if (!file.getOriginalFilename().isEmpty()) {
27             File f = new File("C:\\Desktop\\File\\SingleFileUpload", file.getOriginalFilename());
28             FileOutputStream fis = new FileOutputStream(f);
29             BufferedOutputStream outputStream = new BufferedOutputStream(fis);
30             outputStream.write(file.getBytes());
31             outputStream.flush();
32             outputStream.close();
33             model.addAttribute("smsg", "File uploaded successfully.");
34         } else {
35             model.addAttribute("smsg", "Please select a valid file..");
36         }
37
38         return "fileUploadForm";
39     }
```

FileUploadController.java

```java
// Handling multiple files upload request
@PostMapping("/multipleFileUpload")
public String multipleFileUpload(@RequestParam("file") MultipartFile[] files, Model model) throws IOException {
    // Save file on system
    for (MultipartFile file : files) {
        if (!file.getOriginalFilename().isEmpty()) {
            FileOutputStream fos = new FileOutputStream(new File("C:\\Desktop\\File\\MultiFileUpload", file.getOriginalFilename()));
            BufferedOutputStream outputStream = new BufferedOutputStream(fos);
            outputStream.write(file.getBytes());
            outputStream.flush();
            outputStream.close();
        } else {
            model.addAttribute("mmsg", "Please select at least one file..");
            return "fileUploadForm";
        }
    }
    model.addAttribute("mmsg", "Multiple files uploaded successfully.");
    return "fileUploadForm";
}
```

To Support Multiple files at a time

FileController method to upload multiple files

Step-5: Deploy the application, below screen will be displayed where we can upload single or multiple files

Spring with REST

Introduction

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See the @Path Annotation and URI Path Templates for more information.

Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.

Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.

Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and “Building URIs” in the JAX-RS Overview document for more information.

RESTful Web Services utilize the features of the HTTP Protocol to provide the API of the Web Service. It uses the HTTP Request Types to indicate the type of operation:

GET: Retrieve / Query of existing records.

POST: Creating new records.

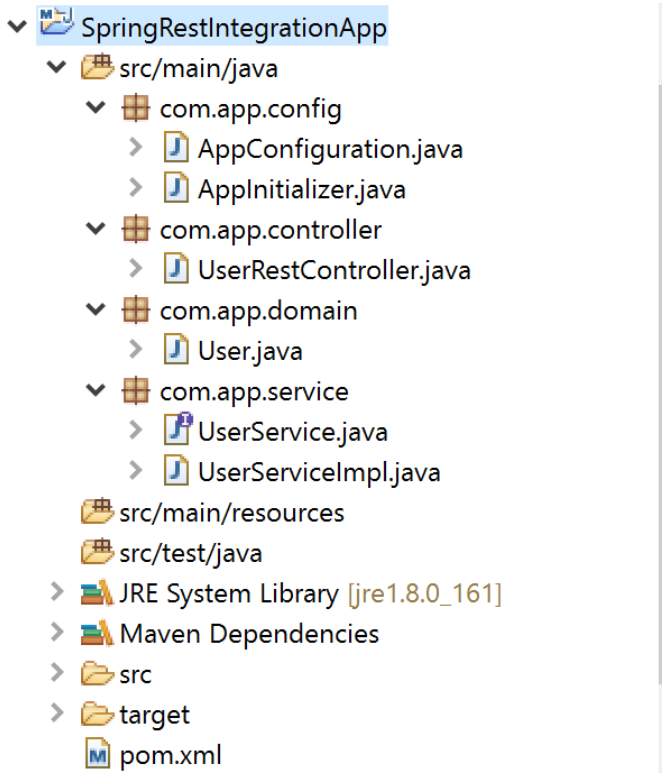
DELETE: Removing records.

PUT: Updating existing records.

Using these 4 HTTP Request Types a RESTful API mimics the CRUD operations (Create, Read, Update & Delete). REST is stateless, each call the to a RESTful Web Service is completely stand-alone, it has no knowledge of previous requests.

Below REST application performs CURD operations using Spring Service class

Step 1: Create Maven Web Project



Step 2: Configure maven dependencies in project pom.xml file

```
<properties>
	<springframework.version>4.3.0.RELEASE</springframework.version>
	<jackson.library>2.7.5</jackson.library>
</properties>
<dependencies>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-core</artifactId>
		<version>${springframework.version}</version>
	</dependency>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-web</artifactId>
		<version>${springframework.version}</version>
	</dependency>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>${springframework.version}</version>
	</dependency>
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>javax.servlet-api</artifactId>
		<version>3.1.0</version>
	</dependency>
	<dependency>
		<groupId>com.fasterxml.jackson.core</groupId>
		<artifactId>jackson-databind</artifactId>
		<version>${jackson.library}</version>
	</dependency>
</dependencies>
```

Step 3: Create Domain class for Storing and Retrieving the data (User.java)

User.java

```
package com.app.domain;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "user")
@XmlAccessorType(XmlAccessType.FIELD)
public class User {

    private Integer userid;
    private String username;
    private String email;
    private String gender;

    //setters & getters

    //toString( )

}
```

Step 4: Create UserService.java class to perform Business operations

```
package com.app.service;

@Service(value = "service")
public class UserServiceImpl implements UserService {

	private static Map<Integer, User> usersData = new HashMap<Integer, User>();

	public boolean add(User user) {
		if (usersData.containsKey(user.getUserid())) {
			return false;
		} else {
			usersData.put(user.getUserid(), user);
			return true;
		}
	}
	public User get(String uid) {
		System.out.println(usersData);
		if (usersData.containsKey(Integer.parseInt(uid))) {
			return usersData.get(Integer.parseInt(uid));
		}
		return null;
	}
	public boolean update(String uid, User user) {
		if (usersData.containsKey(Integer.parseInt(uid))) {
			usersData.put(Integer.parseInt(uid), user);
			return true;
		}
		return false;
	}
	public boolean delete(String uid) {
		if (usersData.containsKey(Integer.parseInt(uid))) {
			usersData.remove(usersData.get(Integer.parseInt(uid)));
			return true;
		}
		return false;
	}
}
```

UserServiceImpl.java

Step 5: Create RestController (UserRestController.java)

```
package com.app.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import com.app.domain.User;
import com.app.service.UserService;

@RestController
public class UserRestController {

	@Autowired(required = true)
	private UserService service;

	@RequestMapping(value = "/add", method = RequestMethod.POST, consumes = { "application/xml",
"application/json" })
	public @ResponseBody String addUser(@RequestBody User user) {
		boolean isAdded = service.add(user);
		if (isAdded) {
			return "User Added successfully";
		} else {
			return "Failed to Add the User..!";
		}
	}


	@RequestMapping(value = "/get", produces = { "application/xml", "application/json" }, method =
RequestMethod.GET)
```

```
	public User getUserById(@RequestParam(name = "uid") String uid) {
		System.out.println("Getting User with User Id : " + uid);
		User user = service.get(uid);
		return user;
	}

	@RequestMapping(value = "/update", method = RequestMethod.PUT,
			consumes = { "application/xml", "application/json" }
	)
	public @ResponseBody String update(@RequestParam("uid") String uid, @RequestBody User user) {
		boolean isAdded = service.update(uid, user);
		if (isAdded) {
			return "User updated successfully";
		} else {
			return "Failed to update the User..!";
		}
	}


	@RequestMapping(value = "/delete", method = RequestMethod.DELETE)
	public @ResponseBody String delete(@RequestParam("uid") String uid) {
		boolean isAdded = service.delete(uid);
		if (isAdded) {
			return "User Deleted successfully";
		} else {
			return "Failed to Delete the User..!";
		}
	}


	public void setService(UserService service) {
		this.service = service;
	}
}
```

Step 6: Create AppConfig and AppInitiazer classes

AppConfiguration.java

AppConfiguration.java

```
package com.app.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.app.*")
public class AppConfiguration {

}
```

AppInitializer.java

AppInitializer.java

```
package com.app.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { AppConfiguration.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/rest/*" };
    }

}
```

Step 7: Deploy the Application into server (I have used Apache Tomcat 8.0)

Step 8: Test the Application using POSTMAN plugin in Google Chrome

Testing Add User

URI: http://localhost:6060/SpringRestIntegrationApp/rest/add

Method Type: POST

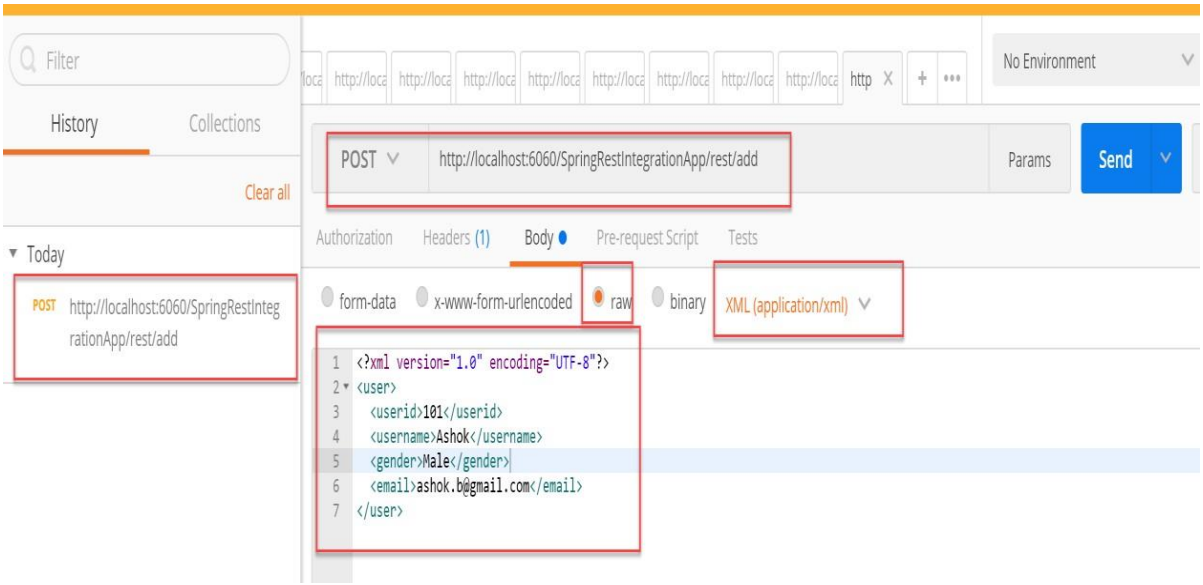
Consumes: {application/xml, application/json}

Produces: text/plain

Request Body Data: In XML Format

```
<? xml version="1.0" encoding="UTF-8"?>
<user>
 <userid>101</userid>
 <username>Raju</username>
 <gender>Male</gender>
 <email>Raj@gmail.com</email>
</user>
```

POSTMAN Screenshot



Testing Fetch User

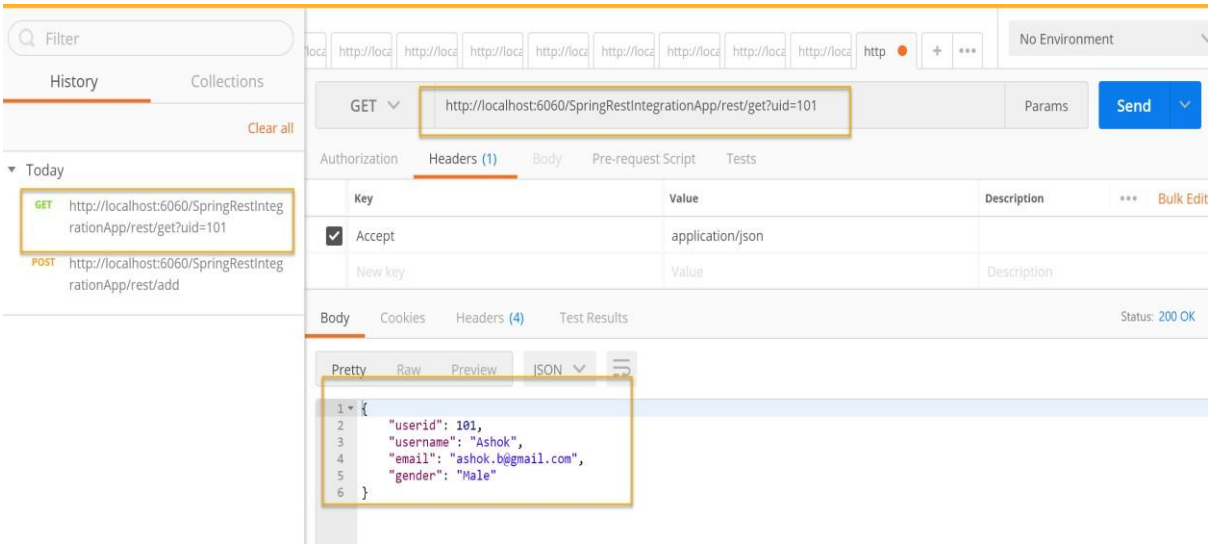
URI: http://localhost:6060/SpringRestIntegrationApp/rest/get?uid=101

Method Type: GET

Input: Request Parameter (? uid=101)

Produces: {application/xml, application/json}

POSTMAN Screenshot



Testing Update User

URL: http://localhost:6060/SpringRestIntegrationApp/rest/update?uid=101

Method Type: PUT

Input in URL: User Id (Request Parameter)? uid=101

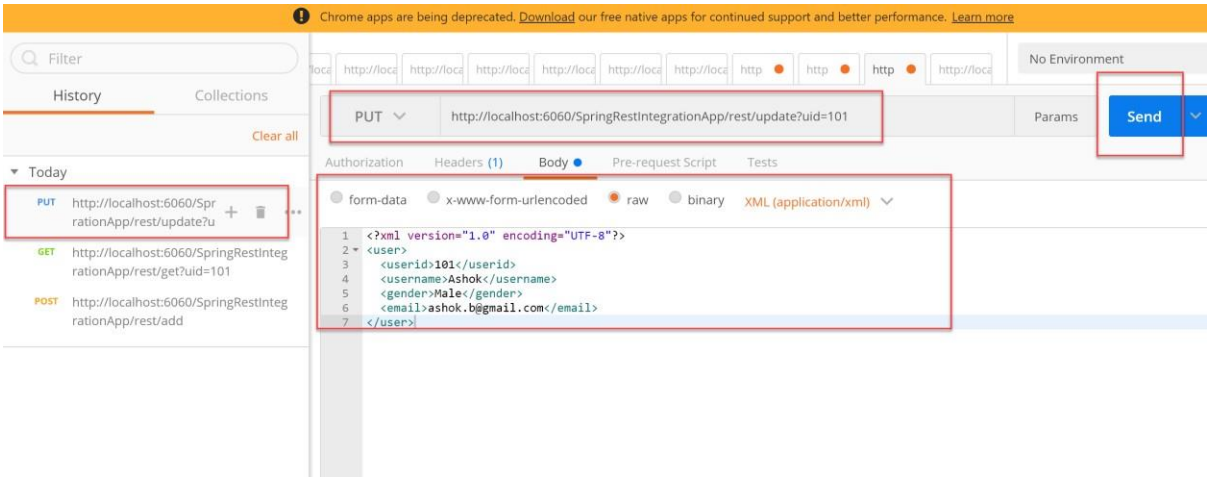
Consumes: {application/xml, application/json}

Produces: text/plain

Request Body Data: in XML format

```
<? xml version="1.0" encoding="UTF-8"?>
<user>
 <userid>101</userid>
 <username>Savitasoft</username>
 <gender>Male</gender>
 <email>savitasoft@gmail.com</email>
</user>
```

POSTMAN Screenshot



Testing Delete User

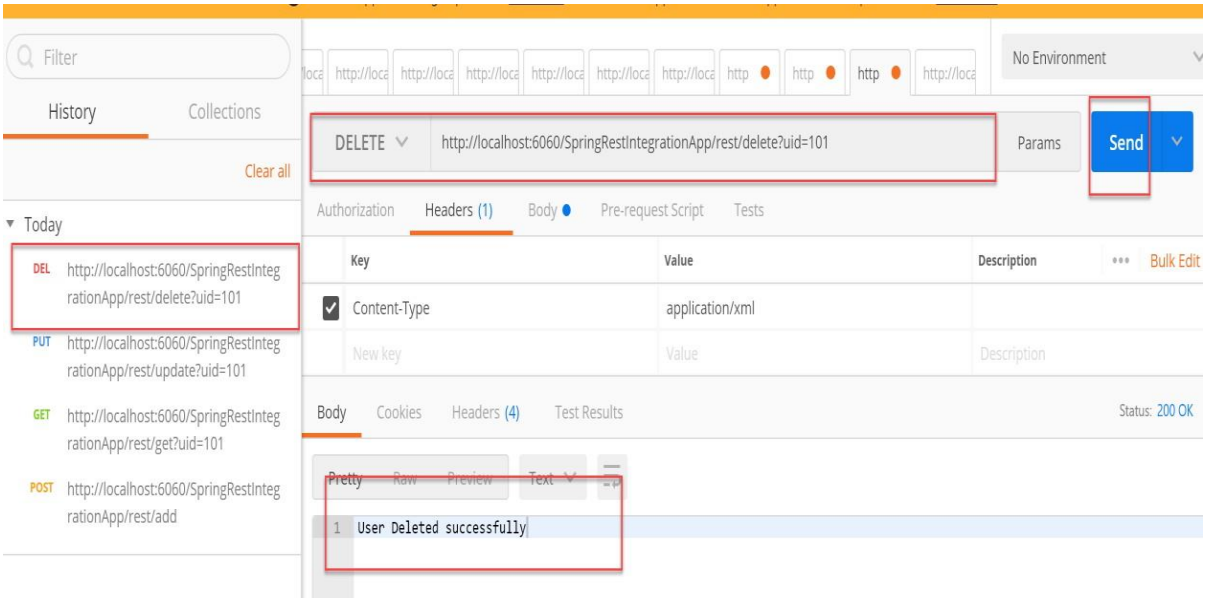
URL: http://localhost:6060/SpringRestIntegrationApp/rest/delete?uid=101

Method Type: DELETE

Input: Request Parameter (? uid=101)

Produces: text/plain

POSTMAN Screenshot



Spring Transactions

Introduction

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the following four key properties described as ACID −

Atomicity – A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.

Consistency – this represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.

Isolation – There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.

Durability − Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows −

Begin the transaction using begin transaction command.

Perform various deleted, update or insert operations using SQL queries.

If all the operation are successful then perform commit otherwise rollback all the operations.

Comprehensive transaction support is among the most compelling reasons to use the Spring Framework. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits

- **Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO).**
- **Support for declarative transaction management.**
- **Simpler API for programmatic transaction management than complex transaction APIs such as JTA.**
- **Excellent integration with spring's data access abstractions.**

Traditionally, J2EE developers have had two choices for transaction management: global or local transactions. Global transactions are managed by the application server, using the Java Transaction API (JTA). Local transactions are resource-specific: the most common example would be a transaction associated with a JDBC connection. This choice has profound implications. For instance, global transactions provide the ability to work with multiple transactional resources (typically relational databases and message queues). With local transactions, the application server is not involved in transaction management and cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.)

Global Transactions: Global transactions have a significant downside, in that code needs to use JTA, and JTA is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA UserTransaction normally needs to be sourced from JNDI: meaning that we need to use both JNDI and JTA to use JTA. Obviously all use of global transactions limits the reusability of application code, as JTA is normally only available in an application server environment. Previously, the preferred way to use global transactions was via EJB CMT (Container Managed Transaction): CMT is a form of declarative transaction management (as distinguished from programmatic transaction management). EJB CMT removes the need for transaction-related JNDI lookups - although of course the use of EJB itself necessitates the use of JNDI. It removes most of the need (although not entirely) to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives around EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

Local Transactions: Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Another downside is that local transactions tend to be invasive to the programming model.

Spring resolves these problems. It enables application developers to use a consistent programming model in any environment. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Declarative transaction management is preferred by most users, and is recommended in most cases.

With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model, developers typically write little or no code related to transaction management, and hence don't depend on the Spring Framework's transaction API (or indeed on any other transaction API).

The key to the Spring transaction abstraction is the notion of a transaction strategy. A transaction strategy is defined by the org.springframework.transaction.PlatformTransactionManager interface, shown below:

```
public interface PlatformTransactionManager {

 TransactionStatus getTransaction(TransactionDefinition definition)  throws TransactionException;

 void commit(TransactionStatus status) throws TransactionException;

 void rollback(TransactionStatus status) throws TransactionException;

}
```

This is primarily an SPI interface, although it can be used programmatically. Note that in keeping with the Spring Framework's philosophy, PlatformTransactionManager is an interface, and can thus be easily mocked or stubbed as necessary. Nor is it tied to a lookup strategy such as JNDI: PlatformTransactionManager implementations are defined like any other object (or bean) in the Spring Framework's IoC container. This benefit alone makes it a worthwhile abstraction even when working with JTA: transactional code can be tested much more easily than if it used JTA directly.

Again in keeping with spring's philosophy, the TransactionException that can be thrown by any of the PlatformTransactionManager interface's methods is unchecked (that is it extends the java.lang.RuntimeException class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle TransactionException. The salient point is that developers are not forced to do so.

The getTransaction (...) method returns a TransactionStatus object, depending on a TransactionDefinition parameter. The returned TransactionStatus might represent a new or existing transaction (if there were a matching transaction in the current call stack - with the implication being that (as with J2EE transaction contexts) a TransactionStatus is associated with a thread of execution).

The Transaction Definition interface specifies:

Isolation: the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?

Propagation: normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case); or suspending the existing transaction and creating a new transaction. Spring offers all of the transaction propagation options familiar from EJB CMT. (Some details regarding the semantics of transaction propagation in spring can be found in the section entitled Section 9.5.7, "Transaction propagation".

Timeout: how long this transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).

Read-only status: a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

Configuring TransactionManager with DataSource

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
```

The related PlatformTransactionManager bean definition will look like this:

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

If we use JTA in a J2EE container, as in the 'dataAccessContext-jta.xml' file from the same sample application, we use a container DataSource, obtained via JNDI, in conjunction with Spring's JtaTransactionManager. The JtaTransactionManager doesn't need to know about the DataSource, or any other specific resources, as it will use the container's global transaction management infrastructure.

```
<?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:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
   http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

 <jee:jndi-lookup id="dataSource" jndi-name="jdbc/myds"/>

 <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />

 <!-- other <bean/> definitions here -->

</beans>
```

We can also use Hibernate local transactions easily, as shown in the following examples from the Spring Framework's PetClinic sample application. In this case, we need to define a Hibernate LocalSessionFactoryBean, which application code will use to obtain Hibernate Session instances.

The DataSource bean definition will be similar to the one shown previously (and thus is not shown). If the DataSource is managed by the JEE container it should be non-transactional as the Spring Framework, rather than the JEE container, will manage transactions.

The 'txManager' bean in this case is of the HibernateTransactionManager type. In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the SessionFactory.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
 <property name="dataSource" ref="dataSource" />
 <property name="mappingResources">
 <list>
  <value>product.hbm.xml</value>
 </list>
 </property>
 <property name="hibernateProperties">
 <value>
              hibernate.dialect=${hibernate.dialect}
            </value>
 </property>
</bean>

<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
 <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

In all these cases, application code will not need to change at all. We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

Declarative transaction management

Most users of the Spring Framework choose declarative transaction management. It is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

The Spring Framework's declarative transaction management is made possible with Spring AOP, although, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with the Spring Framework's declarative transaction management. The basic approach is similar: it is possible to specify transaction behavior (or lack of it) down to individual method level. It is possible to make a setRollbackOnly() call within a transaction context if necessary. The differencesare:

Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only.

The Spring Framework enables declarative transaction management to be applied to any class, not merely special classes such as EJBs.

The Spring Framework offers declarative rollback rules: this is a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.

The Spring Framework gives you an opportunity to customize transactional behavior, using AOP. For example, if you want to insert custom behavior in the case of transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you have no way to influence the container's transaction management other than setRollbackOnly().

The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature, because normally, one does not want transactions to span remote calls.

The concept of rollback rules is important: they enable us to specify which exceptions (and throwables) should cause automatic roll back. We specify this declaratively, in configuration, not in Java code. So, while we can still call setRollbackOnly()on the TransactionStatus object to roll the current transaction back programmatically, most often we can specify a rule that MyApplicationException must always result in rollback. This has the significant advantage that business objects don't need to depend on the transaction infrastructure. For example, they typically don't need to import any Spring APIs, transaction or other.

While the EJB default behavior is for the EJB container to automatically roll back the transaction on a system exception (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an application exception (that is, a checked exception other than java.rmi.RemoteException). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it is often useful to customize this.

Understanding the Spring Framework's declarative transaction implementation

Let's assume that the first two methods of the UserService interface (findUser(String) and findUser(String, String)) have to execute in the context of a transaction with read-only semantics, and that the other methods (insertUser(User) and updateUser(User)) have to execute in the context of a transaction with read-write semantics.

```
<beans>
<!-- this is the service object that we want to make transactional -->
<bean id="userService" class="x.y.service.UserService"/>

<!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below) -->
<tx:advice id="txAdvice" transaction-manager="txManager">
<!-- the transactional semantics... -->
<tx:attributes>
 <!-- all methods starting with 'get' are read-only -->
 <tx:method name="get*" read-only="true"/>
 <!-- other methods use the default transaction settings (see below) -->
 <tx:method name="*"/>
</tx:attributes>
</tx:advice>

<!-- ensure that the above transactional advice runs for any execution
  of an operation defined by the FooService interface -->

<aop:config>
<aop:pointcut id="userServiceOperation" expression="execution(* x.y.service.FooService.*(..))"/>
<aop:advisor advice-ref="txAdvice" pointcut-ref="userServiceOperation"/>
</aop:config>

<!-- don't forget the DataSource -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
</bean>

<!-- similarly, don't forget the PlatformTransactionManager -->
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
</bean>

<!-- other <bean/> definitions here -->

</beans>
```

Let's pick apart the above configuration. We have a service object (the 'userService' bean) that we want to make transactional. The transaction semantics that we want to apply are encapsulated in the <tx:advice/> definition. The <tx:advice/> definition reads as "... all methods on starting with 'get' are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics". The 'transaction-manager' attribute of the <tx:advice/> tag is set to the name of the PlatformTransactionManager bean that is going to actually drive the transactions (in this case the 'txManager' bean).

The <aop:config/> definition ensures that the transactional advice defined by the 'txAdvice' bean actually executes at the appropriate points in the program. First we define a pointcut that matches the execution of any operation defined in the UserService interface ('userServiceOperation'). Then we associate the pointcut with the 'txAdvice' using an advisor. The result indicates that at the execution of a 'userServiceOperation', the advice defined by 'txAdvice' will be run.

The expression defined within the <aop:pointcut/> element is an AspectJ pointcut expression.

A common requirement is to make an entire service layer transactional. The best way to do this is simply to change the pointcut expression to match any operation in your service layer.

For example:

```
<aop:config>
        <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="userServiceMethods"/>
</aop:config>
```

Rolling back

The previous section outlined the basics of how to specify the transactional settings for the classes, typically service layer classes, in your application in a declarative fashion. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an Exception from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code will catch any unhandled Exception as it bubbles up the call stack, and will mark the transaction for rollback.

Note however that the Spring Framework's transaction infrastructure code will, by default, only mark a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of RuntimeException. (Errors will also - by default - result in a rollback.) Checked exceptions that are thrown from a transactional method will not result in the transaction being rolled back.

Exactly which Exception types mark a transaction for rollback can be configured. Find below a snippet of XML configuration that demonstrates how one would configure rollback for a checked, application-specific Exception type.

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
            <tx:method name="get*" read-only="true" rollback-for="NoProductInStockException"/>
             <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

It is also possible to specify 'no rollback rules', for those times when you do not want a transaction to be marked for rollback when an exception is thrown. In the example configuration below, we effectively are telling the Spring Framework's transaction infrastructure to commit the attendant transaction even in the face of an unhandled InstrumentNotFoundException.

```
<tx:advice id="txAdvice">
     <tx:attributes>
            <tx:method name="updateStock" no-rollback-for="InstrumentNotFoundException"/>
             <tx:method name="*"/>
      </tx:attributes>
</tx:advice>
```

When the Spring Framework's transaction infrastructure has caught an exception and is consulting any configured rollback rules to determine whether or not to mark the transaction for rollback, the strongest matching rule wins. So in the case of the following configuration, any exception other than an InstrumentNotFoundException would result in the attendant transaction being marked for rollback.

```
<tx:advice id="txAdvice">
    <tx:attributes>
            <tx:method name="*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>
     </tx:attributes>
</tx:advice>
```

The second way to indicate that a rollback is required is to do so programmatically. Although very simple, this way is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure, as can be seen below:

```
public void resolvePosition() {
  try {
    // some business logic...
  } catch (NoProductInStockException ex) {
    // trigger rollback programmatically
    TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
  }
}
```

You are strongly encouraged to use the declarative approach to rollback if at all possible. Programmatic rollback is available should you absolutely need it, but its usage flies in the face of achieving a nice, clean POJO-based architecture.

Configuring different transactional semantics for different beans

Consider the scenario where you have a number of service layer objects, and you want to apply totally different transactional configuration to each of them. This is achieved by defining distinct <aop:advisor/> elements with differing 'pointcut' and 'advice-ref' attribute values.

Let's assume that all of your service layer classes are defined in a root 'x.y.service' package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in 'Service' have the default transactional configuration, you would write the following:

```
<beans
 <aop:config>
  <aop:pointcut id="serviceOperation"   expression="execution(* x.y.service..*Service.*(..))"/>
  <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>
 </aop:config>

 <!-- these two beans will be transactional... -->
 <bean id="fooService" class="x.y.service.DefaultFooService"/>
 <bean id="barService" class="x.y.service.extras.SimpleBarService"/>

 <!-- ... and these two beans won't -->
 <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right package) -->
 <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in 'Service') -->

 <tx:advice id="txAdvice">
  <tx:attributes>
   <tx:method name="get*" read-only="true"/>
   <tx:method name="*"/>
  </tx:attributes>
 </tx:advice>

  <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

 </beans>
```

Find below an example of configuring two distinct beans with totally different transactional settings.

```
<beans>
 <aop:config>
  <aop:pointcut id="defaultServiceOperation"  expression="execution(* x.y.service.*Service.*(..))"/>
  <aop:pointcut id="noTxServiceOperation"  expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>
  <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>
  <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>
 </aop:config>

 <!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->
 <bean id="fooService" class="x.y.service.DefaultFooService"/>

 <!-- this bean will also be transactional, but with totally different transactional settings -->
 <bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

 <tx:advice id="defaultTxAdvice">
  <tx:attributes>
   <tx:method name="get*" read-only="true"/>
   <tx:method name="*"/>
  </tx:attributes>
 </tx:advice>

 <tx:advice id="noTxAdvice">
  <tx:attributes>
   <tx:method name="*" propagation="NEVER"/>
  </tx:attributes>
 </tx:advice>

 <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->
</beans>
```

<tx:advice/> settings

This section summarizes the various transactional settings that can be specified using the <tx:advice/> tag. The default <tx:advice/> settings are:

- The propagation setting is REQUIRED
- The isolation level is DEFAULT
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any RuntimeException will trigger rollback, and any checked Exception will not

These default settings can be changed; the various attributes of the <tx:method/> tags that are nested within <tx:advice/> and <tx:attributes/> tags are summarized below:

<tx:method/> settings

Attribute	Required?	Default	Description
`name`	Yes		Method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, `get*`, `handle*`, `on*Event`, and so forth.
`propagation`	No	REQUIRED	Transaction propagation behavior.
`isolation`	No	DEFAULT	Transaction isolation level.
`timeout`	No	-1	Transaction timeout value (in seconds).
`read-only`	No	false	Is this transaction read-only?
`rollback-for`	No		`Exception(s)` that trigger rollback; comma-delimited. For example, `com.foo.MyBusinessException,ServletException.`
`no-rollback-for`	No		`Exception(s)` that do *not* trigger rollback; comma-delimited. For example, `com.foo.MyBusinessException,ServletException.`

Using @Transactional

Note: The functionality offered by the @Transactional annotation and the support classes is only available to you if you are using at least Java 5 (Tiger).

In addition to the XML-based declarative approach to transaction configuration, you can also use an annotation-based approach to transaction configuration. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code, and there is generally not much danger of undue coupling, since code that is meant to be used transactionally is almost always deployed that way anyway.

The ease-of-use afforded by the use of the @Transactional annotation is best illustrated with an example, after which all of the details will be explained. Consider the following class definition:

```
// the service class that we want to make transactional

@Transactional
public class ProductServiceImpl implements ProductService {

        Product  getProduct(String productName);

        Product  getProduct(String productName, String category);

        void insertProduct(Product p);

        void updateProduct(Product p);

}
```

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely one line of XML configuration, like so:

```
<?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:aop="http://www.springframework.org/schema/aop"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

 <!-- this is the service object that we want to make transactional -->
 <bean id="productService" class="x.y.service.ProductServiceImpl"/>

 <!-- enable the configuration of transactional behavior based on annotations -->
 <tx:annotation-driven transaction-manager="txManager"/>

 <!-- a PlatformTransactionManager is still required -->
 <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

        <!-- (this dependency is defined somewhere else) -->
        <property name="dataSource" ref="dataSource"/>

 </bean>

 <!-- other <bean/> definitions here -->

</beans>
```

The @Transactional annotation may be placed before an interface definition, a method on an interface, a class definition, or a public method on a class. However, please note that the mere presence of the @Transactional annotation is not enough to actually turn on the transactional behavior - the @Transactional annotation is simply metadata that can be consumed by something that is @Transactional-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the case of the above example, it is the presence of the <tx:annotation-driven/> element that switches on the transactional behavior.

The Spring team's recommendation is that you only annotate concrete classes with the @Transactional annotation, as opposed to annotating interfaces. You certainly can place the @Transactional annotation on an interface (or an interface method), but this will only work as you would expect it to if you are using interface-based proxies. The fact that annotations are not inherited means that if you are using class-based proxies (proxy-target-class="true") or the weaving-based aspect (mode="aspectj") then the transaction settings will not be recognised by the proxying/weaving infrastructure and the object will not be wrapped in a transactional proxy (which would be decidedly bad). So please do take the Spring team's advice and only annotate concrete classes (and the methods of concrete classes) with the @Transactional annotation.

Note: In proxy mode (which is the default), only 'external' method calls coming in through the proxy will be intercepted. This means that 'self-invocation', i.e. a method within the target object calling some other method of the target object, won't lead to an actual transaction at runtime even if the invoked method is marked with @Transactional!

Consider the use of AspectJ mode (see below) if you expect self-invocations to be wrapped with transactions as well. In this case, there won't be a proxy in the first place; instead, the target class will be 'weaved' **(i.e. its byte code will be modified) in order to turn @Transactional into runtime behavior on any kind of method.**

Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the TransactionTemplate may be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure. When using the Spring Framework, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

Spring ORM

Introduction

Spring provides integration with Hibernate, JDO, Oracle TopLink, Apache OJB and iBATIS SQL Maps: in terms of resource management, DAO implementation support, and transaction strategies. For example for Hibernate, there is first-class support with lots of IoC convenience features, addressing many typical Hibernate integration issues. All of these support packages for O/R mappers comply with spring's generic transaction and DAO exception hierarchies. There are usually two integration styles: either using Spring's DAO 'templates' or coding DAOs against plain Hibernate/JDO/TopLink/etc APIs. In both cases, DAOs can be configured through Dependency Injection and participate in spring's resource and transaction management.

Spring's adds significant support when using the O/R mapping layer of your choice to create data access applications. First of all, you should know that once you started using spring's support for O/R mapping, you don't have to go all the way. No matter to what extent, you're invited to review and leverage the Spring approach, before deciding to take the effort and risk of building a similar infrastructure in-house. Much of the O/R mapping support, no matter what technology you're using may be used in a library style, as everything is designed as a set of reusable JavaBeans. Usage inside an ApplicationContext does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside an ApplicationContext.

Some of the benefits of using Spring ORM based DAO:

Ease of testing: Spring's inversion of control approach makes it easy to swap the implementations and config locations of Hibernate SessionFactory instances, JDBC Data Sources, transaction managers, and mapper object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.

Common data access exceptions: Spring can wrap exceptions from you O/R mapping tool of choice, converting them from proprietary (potentially checked) exceptions to a common runtime DataAccessException hierarchy. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. Remember that JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

General resource management: Spring application contexts can handle the location and configuration of Hibernate SessionFactory instances, JDBC DataSources, iBATIS SQL Maps configuration objects, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy and safe handling of persistence resources. For example: Related code using Hibernate generally needs to use the same Hibernate Session for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a Session to the current thread, either by using an explicit 'template' wrapper class at the Java code level or by exposing a current Session through the Hibernate SessionFactory (for DAOs based on plain Hibernate3 API). Thus Spring solves many of the issues that repeatedly arise from typical Hibernate usage, for any transaction environment (local or JTA).

Integrated transaction management: Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the Java code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your Hibernate/JDO related code being affected: for example, between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. As an additional benefit, JDBC-related code can fully integrate transactionally with the code you use to do O/R mapping. This is useful for data access that's not suitable for O/R mapping, such as batch processing or streaming of BLOBs, which still needs to share common transactions with O/R mapping operations.

To avoid vendor lock-in, and allow mix-and-match implementation strategies: While Hibernate is powerful, flexible, open source and free, it still uses a proprietary API. Furthermore one could argue that iBATIS is a bit lightweight,

although it's excellent for use in application that don't require complex O/R mapping strategies. Given the choice, it's usually desirable to implement major application functionality using standard or abstracted APIs, in case you need to switch to another implementation for reasons of functionality, performance, or any other concerns. For example, Spring's abstraction of Hibernate transactions and exceptions, along with its IoC approach which allows you to easily swap in mapper/DAO objects implementing data access functionality, makes it easy to isolate all Hibernate-specific code in one area of your application, without sacrificing any of the power of Hibernate. Higher level service code dealing with the DAOs has no need to know anything about their implementation. This approach has the additional benefit of making it easy to intentionally implement data access with a mix-and-match approach (i.e. some data access performed using Hibernate, and some using JDBC, others using iBATIS) in a non-intrusive fashion, potentially providing great benefits in terms of continuing to use legacy code or leveraging the strength of each technology.

Hibernate

Spring offers Hibernate and JDO support, consisting of a HibernateTemplate / JdoTemplate analogous to JdbcTemplate, a HibernateInterceptor / JdoInterceptor, and a Hibernate / JDO transaction manager. The major goal is to allow for clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business service dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. One simple and consistent approach to wiring up application objects, keeping them as reusable and free from container dependencies as possible. All the individual data access features are usable on their own but integrate nicely with spring's application context concept, providing XML-based configuration and cross-referencing of plain JavaBean instances that don't need to be Spring-aware. In a typical spring app, many important objects are JavaBeans: data access templates, data access objects (that use the templates), transaction managers, business services (that use the data access objects and transaction managers), web view resolvers, web controllers (that use the business services), etc.

SessionFactory setup in a spring application context

To avoid tying application objects to hard-coded resource lookups, spring allows you to define resources like a JDBC DataSource or a Hibernate SessionFactory as beans in an application context. Application objects that need to access resources just receive references to such pre-defined instances via bean references (the DAO definition in the next section illustrates this). The following excerpt from an XML application context definition shows how to set up a JDBCDataSource and a Hibernate SessionFactory on top of it:

```
<bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
    <property name="url" value="jdbc:oracle:thin:@localhost:1521/XE" />
    <property name="username" value="SYSTEM" />
    <property name="password" value="admin" />
</bean>
<bean id="mySessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="ds" />
    <property name="annotatedClasses">
        <list>
            <value>com.orm.entity.Emp</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>
```

Note that switching from a local Jakarta Commons DBCP BasicDataSource to a JNDI-located DataSource (usually managed by the J2EE server) is just a matter of configuration like below

```
<beans>
 <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
     <property name="jndiName" value="java:comp/env/jdbc/myds"/>
 </bean>
 ...
</beans>
```

JNDI Located Datasource Name

Inversion of Control: HibernateTemplate and HibernateCallback

The basic programming model for templating looks as follows, for methods that can be part of any custom data access object or business service. There are no restrictions on the implementation of the surrounding object at all, it just needs to provide a Hibernate SessionFactory. It can get the latter from anywhere, but preferably as bean reference from a spring application context - via a simple setSessionFactory bean property setter. The following snippets show a DAO definition in a spring application context, referencing the above defined SessionFactory, and an example for a DAO method implementation.

```
<beans>
 ...
    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>
</beans>
```

```
public class ProductDaoImpl implements ProductDao {

  private static final String FIND_BY_CATEGORY = "from test.Product product where product.category=?" ;

  private SessionFactory sessionFactory;

  public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }

  public Collection loadProductsByCategory(final String category) throws DataAccessException {
    HibernateTemplate ht = new HibernateTemplate(this.sessionFactory);

     return (Collection) ht.execute(new HibernateCallback() {
       public Object doInHibernate(Session session) throws HibernateException {
         Query query = session.createQuery(FIND_BY_CATEGORY);
         query.setString(0, category);
         return query.list();
       }
     });
  }
}
```

ProductDaoImpl.java

A callback implementation can effectively be used for any Hibernate data access. HibernateTemplate will ensure that Sessions are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single step actions like a single find, load, saveOrUpdate, or delete call, HibernateTemplate offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenientHibernateDaoSupport base class that provides a setSessionFactory

method for receiving a SessionFactory, andgetSessionFactory and getHibernateTemplate for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

  private  static  final  String  FIND_BY_CATEGORY =  "from Product p where p.category=?" ;

  public Collection loadProductsByCategory(String category) throws DataAccessException {
        return  getHibernateTemplate().find(FIND_BY_CATEGORY , category);
  }

}
```

Implementing Spring-based DAOs without callbacks

As alternative to using Spring's HibernateTemplate to implement DAOs, data access code can also be written in a more traditional fashion, without wrapping the Hibernate access code in a callback, while still complying to Spring's generic DataAccessException hierarchy. Spring's HibernateDaoSupport base class offers methods to access the current transactional Session and to convert exceptions in such a scenario; similar methods are also available as static helpers on the SessionFactoryUtils class. Note that such code will usually pass "false" into getSession's the "allowCreate" flag, to enforce running within a transaction (which avoids the need to close the returned Session, as it its lifecycle is managed by the transaction).

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

private  static  final  String  FIND_BY_CATEGORY =   " from Product p  where p.category=? " ;

  public Collection loadProductsByCategory(String category)
      throws DataAccessException, MyException {

    Session session = getSession(getSessionFactory(), false);
    try {
      List result = session.find(FIND_BY_CATEGORY,  category, Hibernate.STRING);
      if (result == null) {
        throw new MyException("invalid search result");
      }
      return result;
    }
    catch (HibernateException ex) {
      throw convertHibernateAccessException(ex);
    }
  }
}
```

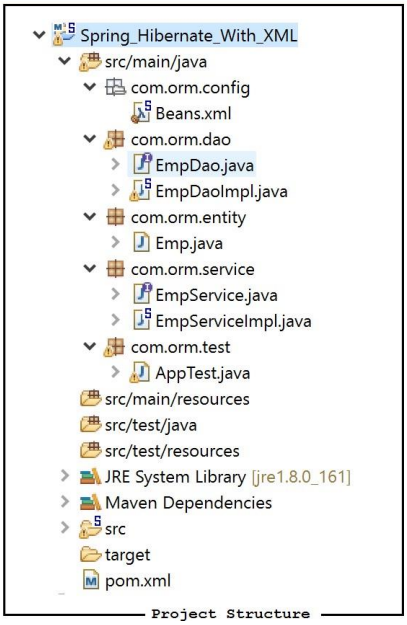
ProductDaoImpl.java

The major advantage of such direct Hibernate access code is that it allows any checked application exception to be thrown within the data access code, while HibernateTemplate is restricted to unchecked exceptions within the callback. Note that one can often defer the corresponding checks and the throwing of application exceptions to after the callback, which still allows working with HibernateTemplate. In general, HibernateTemplate's convenience methods are simpler and more convenient for many scenarios.

Hibernate Integration with spring (XML approach)

Step-1: Create Standalone Maven project and configure spring, Hibernate and ojdbc Dependencies like below

Note: Observe package structure



Project Structure

Spring_Hibernate_With_XML/pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.hib</groupId>
    <artifactId>Spring_Hibernate_With_XML</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.2.5.Final</version>
        </dependency>
        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc14</artifactId>
            <version>1.1.1</version>
        </dependency>
    </dependencies>
</project>
```

Maven Dependencies

Step-2: Create Entity class and Dao interface and its Implementation class like below

EmpDao.java

```java
package com.orm.dao;

import java.util.List;

public interface EmpDao {

    public boolean save(Emp e);

    public Emp findById(int id);

    public List<Emp> findAll();
}
```

EmpDao.java

EmpDaoImpl.java

```java
package com.orm.dao;

import java.io.Serializable;

@Repository
public class EmpDaoImpl implements EmpDao {

    @Autowired(required = true)
    private HibernateTemplate ht;

    @Transactional(readOnly = false)
    public boolean save(Emp e) {
        Serializable ser = ht.save(e);
        return (ser != null) ? true : false;
    }

    public Emp findById(int id) {
        return ht.get(Emp.class, id);
    }

    public List<Emp> findAll() {
        String hql = "From Emp";
        return (List<Emp>) ht.find(hql, null);
    }
}
```

EmpDaoImpl.java

Emp.java

```java
package com.orm.entity;

import javax.persistence.Entity;

@Entity
@Table(name = "EMP_TBL")
public class Emp {

    @Id
    @GeneratedValue
    private Integer eid;
    private String ename;
    private Double salary;

    // setters & getters

    // toString
}
```

Emp.java

Step - 3: Create Service interface and its Implementation like below (Service class will call daoimpl class methods)

EmpService.java

```java
package com.orm.service;

import java.util.List;

public interface EmpService {

    public boolean save(Emp e);

    public Emp findById(int id);

    public List<Emp> findAll();
}
```

EmpService.java

EmpServiceImpl.java

```java
package com.orm.service;

import java.util.List;

@Service
public class EmpServiceImpl implements EmpService {

    @Autowired(required = true)
    private EmpDao dao;

    public boolean save(Emp e) {
        return dao.save(e);
    }

    public Emp findById(int id) {
        return dao.findById(id);
    }

    public List<Emp> findAll() {
        return dao.findAll();
    }

}
```

EmpServiceImpl.java

Step – 4: Create Spring Bean Configuration file with DataSource, SessionFactory, Transaction Manager and Hibernate Template like below

Beans.xml

```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"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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-4.3.xsd
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <context:component-scan base-package="com.orm" />
    <tx:annotation-driven transaction-manager="txManager" />
    <bean id="ds"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@Localhost:1521/XE" />
        <property name="username" value="SYSTEM" />
        <property name="password" value="admin" />
    </bean>
    <bean id="sf"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="ds" />
        <property name="annotatedClasses">
            <list>
                <value>com.orm.entity.Emp</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>
    <bean id="txManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sf" />
    </bean>
    <bean id="ht" class="org.springframework.orm.hibernate5.HibernateTemplate">
        <property name="sessionFactory" ref="sf" />
    </bean>
</beans>
```

Step – 5: Create Test class to test our application

AppTest.java

```
package com.orm.test;

import java.util.List;

public class AppTest {

    public static void main(String[] args) {

        // Starting IOC
        @SuppressWarnings("resource")
        ApplicationContext context = new ClassPathXmlApplicationContext("com/orm/config/Beans.xml");

        // Getting EmpService bean obj
        EmpService service = context.getBean(EmpService.class);

        // Setting Data into Entity class obj
        Emp e = new Emp();
        e.setEname("Gita");
        e.setSalary(35000.00);

        // Saving record
        System.out.println(" Record Saved ? : " + service.save(e));

        // Getting One Record using Id
        System.out.println(service.findById(109));

        // Getting All Records
        List<Emp> emps = service.findAll();

        if (!emps.isEmpty()) {
            for (Emp emp : emps) {
                System.out.println(emp);
            }
        }
    }
}
```

Hibernate Integration with spring (Annotation Approach)

Step-1: Create Maven Standalone project and configure maven dependencies for Spring, Hibernate and ojdbc

Project Structure

SpringWithHibernate_Without_XML/pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.aits</groupId>
    <artifactId>SpringWithHibernate-WithoutXML</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.3.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.2.5.Final</version>
        </dependency>
        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc14</artifactId>
            <version>1.1.1</version>
        </dependency>
    </dependencies>
</project>
```

Maven Dependencies

Step-2: Create Application Configuration class like below

```java
@Configuration
@PropertySource("classpath:DB.properties")
@EnableTransactionManagement
@ComponentScan(basePackages = { "com.orm.dao", "com.orm.service" })
public class AppConfig {

	@Autowired
	private Environment env;

	@Bean
	public DataSource getDataSource() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setDriverClassName(env.getProperty("db.driver"));
		dataSource.setUrl(env.getProperty("db.url"));
		dataSource.setUsername(env.getProperty("db.username"));
		dataSource.setPassword(env.getProperty("db.password"));
		return dataSource;
	}
```

DataSource Creation

```java
	@Bean
	public HibernateTransactionManager getTransactionManager() {

		HibernateTransactionManager transactionManager = new HibernateTransactionManager();

		transactionManager.setSessionFactory(getSessionFactory().getObject());

		return transactionManager;
	}
```

TransactionManager Creation

```java
	@Bean
	public HibernateTemplate getHibernateTemple() {

		HibernateTemplate hibernateTemplate = new HibernateTemplate(getSessionFactory().getObject());

		return hibernateTemplate;
	}
```

HibernateTemplate Creation

```java
	@Bean
	public LocalSessionFactoryBean getSessionFactory() {
		LocalSessionFactoryBean factoryBean = new LocalSessionFactoryBean();
		factoryBean.setDataSource(getDataSource());
		Properties props = new Properties();
		props.put("hibernate.show_sql", env.getProperty("hibernate.show_sql"));
		props.put("hibernate.hbm2ddl.auto", env.getProperty("hibernate.hbm2ddl.auto"));
		factoryBean.setHibernateProperties(props);
		factoryBean.setAnnotatedClasses(Emp.class);
		return factoryBean;
	}
```

SessionFactory Creation

AppConfig.java (This class works as a replacement for xml file)

Step-3: Create Properties file in resource folder to configure database configuration details

DB.properties

```
# Database properties
db.driver=oracle.jdbc.driver.OracleDriver
db.url=jdbc:oracle:thin:@localhost:1521/XE
db.username=system
db.password=admin

# Hibernate properties
hibernate.show_sql=true
hibernate.hbm2ddl.auto=update
```

Database Properties

Step-4: Create Entity class and Dao interface and its Implementation class like below

EmpDao.java

```java
package com.orm.dao;

import java.util.List;

public interface EmpDao {

	public boolean save(Emp e);

	public Emp findById(int id);

	public List<Emp> findAll();
}
```

EmpDao.java

EmpDaoImpl.java

```java
package com.orm.dao;

import java.io.Serializable;

@Repository
public class EmpDaoImpl implements EmpDao {

	@Autowired(required = true)
	private HibernateTemplate ht;

	@Transactional(readOnly = false)
	public boolean save(Emp e) {
		Serializable ser = ht.save(e);
		return (ser != null) ? true : false;
	}

	public Emp findById(int id) {
		return ht.get(Emp.class, id);
	}

	public List<Emp> findAll() {
		String hql = "From Emp";
		return (List<Emp>) ht.find(hql, null);
	}
}
```

EmpDaoImpl.java

Emp.java

```java
package com.orm.entity;

import javax.persistence.Entity;

@Entity
@Table(name = "EMP_TBL")
public class Emp {

	@Id
	@GeneratedValue
	private Integer eid;
	private String ename;
	private Double salary;

	// setters & getters

	// toString
}
```

Emp.java

Step-5: Create Service interface and its Implementation like below

EmpService.java

```java
package com.orm.service;

import java.util.List;

public interface EmpService {

    public boolean save(Emp e);

    public Emp findById(int id);

    public List<Emp> findAll();
}
```

EmpService.java

EmpServiceImpl.java

```java
package com.orm.service;

import java.util.List;

@Service
public class EmpServiceImpl implements EmpService {

    @Autowired(required = true)
    private EmpDao dao;

    public boolean save(Emp e) {
        return dao.save(e);
    }

    public Emp findById(int id) {
        return dao.findById(id);
    }

    public List<Emp> findAll() {
        return dao.findAll();
    }

}
```

EmpServiceImpl.java

Step-6: Create Test class to test our application

AppTest.java

```java
package com.orm.test;

import java.util.List;

public class AppTest {

    public static void main(String[] args) {

        // Starting IOC
        @SuppressWarnings("resource")
        ApplicationContext context = new ClassPathXmlApplicationContext("com/orm/config/Beans.xml");

        // Getting EmpService bean obj
        EmpService service = context.getBean(EmpService.class);

        // Setting Data into Entity class obj
        Emp e = new Emp();
        e.setEname("Gita");
        e.setSalary(35000.00);

        // Saving record
        System.out.println(" Record Saved ? : " + service.save(e));

        // Getting One Record using Id
        System.out.println(service.findById(109));

        // Getting All Records
        List<Emp> emps = service.findAll();

        if (!emps.isEmpty()) {
            for (Emp emp : emps) {
                System.out.println(emp);
            }
        }
    }
}
```

Spring Boot

Introduction

Spring is a very popular Java framework for building web and enterprise applications. Unlike many other frameworks which focuses on only one area(web or persistence etc), Spring framework provides a wide verity of features addressing the modern business needs via its portfolio projects.

Spring framework provides flexibility to configure beans in multiple ways such as XML, Annotations and JavaConfig. With the number of features increased the complexity also gets increased and configuring Spring applications becomes tedious and error-prone. Spring team created SpringBoot to address the complexity of configuration.

But before diving into SpringBoot, we will take a quick look at Spring framework and see what kind of problems SpringBoot is trying to address.

Spring framework was created primarily as a Dependency Injection container but it is much more than that.

Spring is very popular because of several reasons:

- Spring's dependency injection approach encourages writing testable code
- Easy to use and powerful database transaction management capabilities
- Spring simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts/JSF etc web frameworks
- State of the art Web MVC framework for building web applications

Along with Spring framework there are many other Spring portfolio projects which helps to build applications addressing modern business needs:

- **Spring Data**: Simplifies data access from relational and NoSQL data stores.
- **Spring Batch:** Provides powerful batch processing framework.
- **Spring Security:** Robust security framework to secure applications.
- **Spring Social:** Supports integration with social networking sites like Facebook, Twitter, LinkedIn, GitHub etc.
- **Spring Integration:** An implementation of Enterprise Integration Patterns to facilitate integration with other enterprise applications using lightweight messaging and declarative adapters.

There are many other interesting projects addressing various other modern application development needs. For more information take a look at http://spring.io/projects.

In the initial days, Spring provides XML based approach for configuring beans. Later Spring introduced XML based DSLs, Annotations and JavaConfig based approaches for configuring beans.

XML Configuration Approach

```xml
<bean id="userService" class="com.ashok.service.UserService">
        <property name="userDao" ref="userDao"/>
</bean>

<bean id="userDao" class="com.ashok.dao.JdbcUserDao">
        <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/test"/>
        <property name="username" value="root"/>
        <property name="password" value="secret"/>
</bean>
```

Annotation Based Configuration Approach

```java
@Service
public class UserService {

private UserDao userDao;

@Autowired
public UserService(UserDao dao){
        this.userDao = dao;
}
...
...
}
```

UserService.java

```java
@Repository
public class JdbcUserDao {

private DataSource dataSource;

@Autowired
public JdbcUserDao(DataSource dataSource){
        this.dataSource = dataSource;
}
...
...
}
```

JdbcUserDao.java

Java Config Approach

```java
@Configuration
public class AppConfig {

@Bean
public UserService userService(UserDao dao){
    return new UserService(dao);
}

@Bean
public UserDao userDao(DataSource dataSource){
  return new JdbcUserDao(dataSource);
}

@Bean
public DataSource dataSource(){

        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/test");
        dataSource.setUsername("root");
        dataSource.setPassword("secret");

        return dataSource;
}
}
```

AppConfig.java

Spring provides many approaches for doing the same thing and we can even mix the approaches as well like you can use both JavaConfig and Annotation based configuration styles in the same application. That is a lot of flexibility and it is one way good and one way bad. People new to Spring framework may gets confused about which approach to follow.

As of now the Spring team is suggesting to follow JavaConfig based approach as it gives more flexibility. But there is no one-size fits all kind of solution. One has to choose the approach based on their own application needs.

Let us take a quick look at the configuration of a typical SpringMVC + JPA/Hibernate based web application configuration looks like.

Developing Web Application using SpringMVC and JPA

Before getting to know what is SpringBoot and what kind of features it provides, let us take a look at how a typical Spring web application configuration looks like, what are the pain points and then we will discuss about how SpringBoot addresses those problems.

Step 1: Configure Maven Dependencies

First thing we need to do is configure all the dependencies required in our pom.xml.

```
</dependency>
        <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.13</version>
</dependency>
<dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
</dependency>
<dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.4.190</version>
</dependency>
<dependency>
        <groupId>commons-dbcp</groupId>
        <artifactId>commons-dbcp</artifactId>
        <version>1.4</version>
</dependency>
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.38</version>
</dependency>
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.3.11.Final</version>
</dependency>
<dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
</dependency>
<dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring4</artifactId>
        <version>2.1.4.RELEASE</version>
</dependency>
```

pom.xml

We have configured all our Maven jar dependencies to include Spring MVC, Spring Data JPA, JPA/Hibernate, Thymeleaf and Log4j.

Step 2: Configure Service/DAO layer beans using JavaConfig.

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages="com.savitasoft.demo"
) @PropertySource(value = { "classpath:application.properties"
})public class AppConfig {
@Autowired
```

```
private Environment env;
@Bean
public static PropertySourcesPlaceholderConfigurer placeHolderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
}
@Value("${init-db:false}")
private String initDatabase;

@Bean
public PlatformTransactionManager transactionManager() {
        EntityManagerFactory factory = entityManagerFactory().getObject();
        return new JpaTransactionManager(factory);
}
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
vendorAdapter.setGenerateDdl(Boolean.TRUE);
vendorAdapter.setShowSql(Boolean.TRUE);
factory.setDataSource(dataSource());
factory.setJpaVendorAdapter(vendorAdapter);
factory.setPackagesToScan(env.getProperty("packages-to-scan"));

Properties jpaProperties = new Properties();
jpaProperties.put("hibernate.hbm2ddl.auto",env.getProperty("hibernate.hbm2ddl.auto"));
factory.setJpaProperties(jpaProperties);
factory.afterPropertiesSet();
factory.setLoadTimeWeaver(new InstrumentationLoadTimeWeaver());
        return factory;
}
```

```
@Bean
public HibernateExceptionTranslator hibernateExceptionTranslator() {
        return new HibernateExceptionTranslator();
}
@Bean
public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
        dataSource.setUrl(env.getProperty("jdbc.url"));
        dataSource.setUsername(env.getProperty("jdbc.username"));
        dataSource.setPassword(env.getProperty("jdbc.password"));
        return dataSource;
}
@Bean
public DataSourceInitializer dataSourceInitializer(DataSource dataSource) {
        DataSourceInitializer dsInitializer = new DataSourceInitializer();
        dsInitializer.setDataSource(dataSource);
        ResourceDatabasePopulator dbPopulator = new ResourceDatabasePopulator();
        dbPopulator.addScript(new ClassPathResource("data.sql"));
        dsInitializer.setDatabasePopulator(dbPopulator);
        dsInitializer.setEnabled(Boolean.parseBoolean(initDatabase));
        return dsInitializer;
}
}
```

In our AppConfig.java configuration class we have done the following:

- Marked it as a Spring Configuration class using @Configuration annotation.
- Enabled Annotation based transaction management using @EnableTransactionManagement
- Configured @EnableJpaRepositories to indicate where to look for Spring Data JPA repositories
- Configured PropertyPlaceHolder bean using @PropertySource annotation and PropertySourcesPlaceholderConfigurer bean definition which loads properties from application properties file.
- Defined beans for DataSource, JPA EntityManagerFactory, JpaTransactionManager.
- Configured DataSourceInitializer bean to initialize the database by executing data.sql script on application start up.

we can configure property placeholder values in application.properties as follows:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test
jdbc.username=root
jdbc.password=admin
init-db=true
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.hbm2ddl.auto=update
```

application.properties

Create log4j.properties file with below basic configurations

```
log4j.rootCategory=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p %t %c{2}:%L - %m%n
log4j.category.com.sivalabs=DEBUG
log4j.category.org.springframework=INFO
```

log4j.properties

Step 3: Configure Spring MVC web layer beans

We will have to configure Thymeleaf ViewResolver, static ResourceHandlers, MessageSource for i18n etc.

```
@Configuration
@ComponentScan(basePackages = {
"com.savitasoft.demo"})@EnableWebMvc
public class WebMvcConfig extends WebMvcConfigurerAdapter {
@Bean
public TemplateResolver templateResolver() {
TemplateResolver templateResolver = new ServletContextTemplateResolver();
        templateResolver.setPrefix("/WEB-INF/views/");
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML5");
        templateResolver.setCacheable(false);
        return templateResolver;
}
@Bean

public SpringTemplateEngine templateEngine() {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver());
        return templateEngine;
```

```
}
@Bean
public ThymeleafViewResolver viewResolver() {
        ThymeleafViewResolver thymeleafViewResolver = new ThymeleafViewResolver();
        thymeleafViewResolver.setTemplateEngine(templateEngine());
        thymeleafViewResolver.setCharacterEncoding("UTF-8");
}
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**") .addResourceLocations("/resources/");
}
@Bean(name = "messageSource")
public MessageSource configureMessageSource() {
ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();
        messageSource.setBasename("classpath:messages");
        messageSource.setCacheSeconds(5);
        messageSource.setDefaultEncoding("UTF-8");
        return messageSource;
}
}
```

In our WebMvcConfig.java configuration class we have done the following:

- Marked it as a Spring Configuration class using @Configuration annotation.
- Enabled Annotation based Spring MVC configuration using @EnableWebMvc
- Configured Thymeleaf ViewResolver by registering TemplateResolver, SpringTemplateEngine, ThymeleafViewResolver beans.
- Registered ResourceHandlers bean to indicate requests for static resources with URI /resources/** will be served from the location /resources/ directory.
- Configured MessageSource bean to load i18n messages from ResourceBundle messages_-
- {country-code}.properties from classpath.

Create messages.properties file in src/main/resources folder and add the following property.

```
app.title=SpringMVC JPA Demo (Without SpringBoot)
```

Step 4: Register Spring MVC FrontController servlet DispatcherServlet.

Prior to Servlet 3.x specification we have to register Servlets/Filters in web.xml. Since Servlet 3.x specification we can register Servlets/Filters programatically using ServletContainerInitializer. Spring MVC provides a convenient class AbstractAnnotationConfigDispatcherServletInitializer to register DispatcherServlet.

```
public class SpringWebAppInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer
{
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
        return new Class<?>[] { AppConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses()
    {
        return new Class<?>[] { WebMvcConfig.class };
    }

    @Override
    protected String[] getServletMappings()
    {
        return new String[] { "/" };
    }

    @Override
    protected Filter[] getServletFilters() {
       return new Filter[]{ new OpenEntityManagerInViewFilter() };
    }
}
```

In our SpringWebAppInitializer.java configuration class we have done the following:

- We have configured AppConfig.class as RootConfirationClasses which will become the parent ApplicationContext that contains bean definitions shared by all child (DispatcherServlet) contexts.
- We have configured WebMvcConfig.class as ServletConfigClasses which is child ApplicationContext that contains WebMvc bean definitions.
- We have configured ”/” as ServletMapping means all the requests will be handled by DispatcherServlet.
- We have registered OpenEntityManagerInViewFilter as a Servlet Filter so that we can lazy load the JPA Entity lazy collections while rendering the view.

Step 5: Create a JPA Entity and Spring Data JPA Repository

Create a JPA entity User.java and a Spring Data JPA repository for User entity.

```
@Entity
public class User
{
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
    private String name;

    public User()
    {
    }

    public User(Integer id, String name)
    {
        this.id = id;
        this.name = name;
    }

    //setters and getters
}
```

```
public interface UserRepository extends JpaRepository<User, Integer>
{

}
```

If you don't understand what is JpaRepository don't worry. We will learn more about Spring Data JPA in next chapter Working With JPA in detail.

Step 6: Create a SpringMVC Controller

Create a SpringMVC controller to handle URL "/" and render list of users.

```
@Controller
public class HomeController
{
    @Autowired UserRepository userRepo;

    @RequestMapping("/")
    public String home(Model model)
    {
        model.addAttribute("users", userRepo.findAll());
        return "index";
    }
}
```

Step 7: Create a thymeleaf view /WEB-INF/views/index.html to render list of Users.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8"/>
<title>Home</title>
</head>
<body>
    <h2 th:text="#{app.title}">App Title</h2>
    <table>
        <thead>
            <tr>
                <th>Id</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="user : ${users}">
                <td th:text="${user.id}">Id</td>
                <td th:text="${user.name}">Name</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

We are all set now to run the application. But before that we need to download and configure the server like Tomcat or Jetty or Wildfly etc in your IDE. You can download Tomcat 8 and configure in your favorite IDE, run the application and point your browser to http://localhost:8080/springmvcjpa- demo. You should see the list of users details in a table.

Users List

Yay…We did it. But wait..Isn't it too much work to just show a list of user details pulled from a database table?

Let us be honest and fair. All this configuration is not just for this one use-case. This configuration is basis for rest of the application also. But again, this is too much of work to do if you want to quickly get up and running. Another problem with it is, assume you want to develop another SpringMVC application with similar technical stack? Well, you copy-paste the configuration and tweak it. Right?

But remember one thing: if you have to do the same thing again and again, you should find an automated way to do it.

Apart from writing the same configuration again and again, do you see any other problems here?

Well, let me list our what are the problems I am seeing here.

1. You need to hunt for all the compatible libraries for the specific Spring version and configure them.
2. 95% of the times we configure the DataSource, EntitymanagerFactory, TransactionManager etc beans in the same way. Wouldn't it be great if Spring can do it for me automatically.
3. Similarly we configure SpringMVC beans like ViewResolver, MessageSource etc in the same way most of the times. If Spring can automatically do it for me that would be awesome. Imagine, what if Spring is capable of configuring beans automatically? What if you can customize the automatic configuration using simple customizable properties? For example, instead of mapping DispatcherServlet url-pattern to "/" you want to map it to "/app/". Instead of putting thymeleaf views in "/WEB-INF/views" folder you may want to place them in "/WEB-INF/templates/" folder. So basically you want Spring to do things automatically but provide the flexibility to override the default configuration in a simpler way?

   Well, you are about to enter into the world of SpringBoot where your dreams come true!!!

Welcome to SpringBoot! SpringBoot do what exactly you are looking for. It will do things automatically for you but allows you to override the defaults if you want to.

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration. You can use Spring Boot to create Java applications that can be started using java -jar or more traditional war deployments. We also provide a command line tool that runs "spring scripts".

Spring Boot primary goals are:

1. Provide a radically faster and widely accessible getting started experience for all Spring development.
2. Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
3. Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
4. Absolutely no code generation and no requirement for XML configuration.

Instead of explaining in theory I prefer to explain by example. So let us implement the same application that we built earlier but this time using SpringBoot.

Step 1: Create a Maven based SpringBoot Project

Create a Maven project and configure the dependencies as follows:

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.6.RELEASE</version>
</parent>

<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
</dependencies>
```

Wow our pom.xml suddenly become so small!

Step 2: Configure datasource/JPA properties in src/main/resources/application.properties as follows

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.initialize=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

application.properties

you can copy the same data.sql file into src/main/resources folder.

Step 3: Create a JPA Entity and Spring Data JPA Repository Interface for the entity

Create User.java, UserRepository.java and HomeController.java same as in springmvc-jpa-demo application.

```java
@Entity
public class User
{
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
    private String name;

    public User()
    {
    }

    public User(Integer id, String name)
    {
        this.id = id;
        this.name = name;
    }

    //setters and getters
}
```

```java
public interface UserRepository extends JpaRepository<User, Integer>
{

}
```

```java
@Controller
public class HomeController
{
    @Autowired UserRepository userRepo;

    @RequestMapping("/")
    public String home(Model model)
    {
        model.addAttribute("users", userRepo.findAll());
        return "index";
    }
}
```

Step 4: Create Thymeleaf view to show list of users

Copy /WEB-INF/views/index.html that we created in springmvc-jpa-demo application into src/main/resources/templates folder in our new project.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8"/>
<title>Home</title>
</head>
<body>
    <h2 th:text="#{app.title}">App Title</h2>
    <table>
        <thead>
            <tr>
                <th>Id</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="user : ${users}">
                <td th:text="${user.id}">Id</td>
                <td th:text="${user.name}">Name</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

Step 5: Create SpringBoot EntryPoint Class.

Create a Java class Application.java with main method as follows:

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

Now run Application.java as a Java Application and point your browser to http://localhost:8080/.

You should see the list of users in table format.

Let me explain what just happened in Spring Boot Application

1. Easy dependency Management

First thing to observe is we are using some dependencies named like spring-boot-starter-*.

Remember I said “95% of the times I use same configuration”. So when you add springboot-starter-web dependency by default it will pull all the commonly used libraries while developing Spring MVC applications such as spring-webmvc, jackson-json, validation-api and tomcat.

We have added spring-boot-starter-data-jpa dependency. This pulls all the spring-datajpa dependencies and also adds Hibernate libraries because majority of the applications use Hibernate as JPA implementation.

2. Auto Configuration

Not only the spring-boot-starter-web adds all these libraries but also configures the commonly registered beans like DispatcherServlet, ResourceHandlers, MessageSource etc beans with sensible defaults.

We also added spring-boot-starter-thymeleaf which not only adds the thymeleaf library dependencies but also configures ThymeleafViewResolver beans as well automatically.

We haven't defined any of the DataSource, EntityManagerFactory, TransactionManager etc beans but they are automatically gets created. How? If we have any in-memory database drivers like H2 or HSQL in our classpath then SpringBoot will automatically creates an in-memory DataSource and then registers EntityManagerFactory, TransactionManager beans automatically with sensible defaults. But we are using MySQL, so we need to explicitly provide MySQL connection details. We have configured those MySQL connection details in application.properties file and SpringBoot creates a DataSource using these properties.

3. Embedded Servlet Container Support

The most important and surprising thing is we have created a simple Java class annotated with some magical annotation @SpringApplication having a main method and by running that main we are able to run the application and access it at http://localhost:8080/.

Where is the servlet container comes from?

We have added spring-boot-starter-web which pull the spring-boot-starter-tomcat automatically and when we run the main() method it started tomcat as an embedded container so that we don't have to deploy our application on any externally installed tomcat server. By the way have you observe that our packaging type in pom.xml is 'jar' not 'war'. Wonderful! Ok, but what if I want to use Jetty server instead of tomcat? Simple, exclude spring-bootstarter- tomcat from spring-boot-starter-web and include spring-boot-starter-jetty. That's it.

But, this looks all magical!!!

I can imagine what you are thinking. You are thinking like SpringBoot looks cool and it is doing lot of things automatically for me. But still I am not fully understand how it is all really working behind the scenes. Right?

I can understand. Watching magic show is fun normally, but not in Software Development. Don't worry, we will be looking at each of those things and explain in-detail how things are happening behind the scenes. But I don't want to overwhelm you by dumping everything on to your hear right now in this chapter.

What is SpringBoot?

SpringBoot is an opinionated framework which helps to build Spring based applications quickly and easily. The main goal of SpringBoot is to quickly create Spring based applications without requiring the developers to write the same boilerplate configuration again and again.

The key SpringBoot features include:

- SpringBoot starters
- SpringBoot AutoConfiguration
- Elegant Configuration Management
- SpringBoot Actuator
- Easy to use Embedded Servlet container support

SpringBoot starters

SpringBoot offers many starter modules to get started quickly with many of the commonly used technologies like SpringMVC, JPA, MongoDB, Spring Batch, SpringSecurity, Solr, ElasticSearch etc. These starters are pre-configured with the most commonly used library dependencies so that we don't have to search for the compatible library versions and configure them manually.

For ex: spring-boot-starter-data-jpa starter module includes all the dependencies required to use Spring Data JPA along with Hibernate library dependencies as Hibernate is the most commonly used JPA implementation.

You can find list of all the SpringBoot starters that comes out-of-the-box in official documentation at Starter POMs

SpringBoot AutoConfiguration

SpringBoot addresses the "Spring applications needs complex configuration" problem by eliminating the need to manually configuring the boilerplate configuration by using AutoConfiguration. SpringBoot takes opinionated view of the application and configures various components automatically by registering beans based on various criteria.

The criteria can be:

- Availability of a particular class in classpath
- Presence or Absence of a Spring Bean
- Presence of a System Property
- Absence of configuration file etc.

For example:

If you have spring-webmvc dependency in your classpath SpringBoot assumes you are trying to build a SpringMVC based web application and automatically tries to register DispatcherServlet if it is not already registered by you. If you have any Embedded database driver in classpath like H2 or HSQL and if you haven't configured any DataSource bean explicitly then SpringBoot will automatically register a DataSource bean using in-memory database settings.

Elegant Configuration Management

Spring supports externalizing configurable properties using @PropertySource configuration. Spring-Boot takes it even further by using the sensible defaults and powerful type-safe property binding to bean properties. SpringBoot supports having separate configuration files for different profiles without requiring much configuration.

SpringBoot Actuator

Being able to get the various details of an application running in production is very crucial to many applications. SpringBoot Actuator provides a wide variety of such production ready features without requiring to write much code.

Some of the Spring Actuator features are:

- Can view the application bean configuration details
- Can view the application URL Mappings, Environment details and configuration parameter values
- Can view the registered Health Check Metrics and many more.

Easy to use Embedded Servlet container support

Traditionally while building web applications we used to create war type modules and then deploy on external servers like tomcat, wildlfy etc. But by using SpringBoot we can create jar type module and embed the servlet container within application itself very easily so that our application will be a self-contained deployment unit. Also, during development

we can easily run the SpringBoot jar type module as a Java application from IDE or from command-line using build tool like Maven or Gradle.

Our First SpringBoot Application

There are many ways to create a SpringBoot application. The simplest way is to use Spring Initializer

http://start.spring.io/ which is an online SpringBoot application generator.

Using Spring Initializer

You can point your browser to http://start.spring.io/ and enter the project details as follows:

1. Click on Switch to the full version link

2. Select Maven Project and SpringBoot version (As of writing this book latest is 1.3.2)

3. Enter the maven project details as follows:

Group: com.savitasoft

Artifact : springboot-basic

Name: springboot-basic

Package Name:

com.savitasoft.demoPackaging:

Jar

Java version: 1.8

Language: Java

4. Either you can search for the starters if you are already familiar with the names or click on

Switch to the full version link. to see all the available starter.

You can see many starter modules organized into various categories like Core, Web, Data etc.

Select Web checkbox from Web category Click on Generate Project button. Now you can extract the downloaded zip file and import into your favorite IDE.

Using Spring Tool Suite

Spring Tool Suite (STS https://spring.io/tools/sts) is an extension of Eclipse IDE with lot of Spring framework related plugins.

You can easily create a SpringBoot application from STS as follows:

• Select File -> New -> Other -> Spring -> Spring Starter Project -> Next - > Enter Project Name -> Click Next -> Select Dependencies required -> Click Finish.

Step 1: First, let us take a look at pom.xml file.

```
<parent>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.6.RELEASE</version>
</parent>

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

1. First thing to note here is our springboot-basic maven module is inheriting from springboot-starter-parent module.

By inheriting from spring-boot-starter-parent module we will automatically get the following benefits:

• We only need to specify the springboot version once in parent module configuration. We don't need to specify the version for all the starter dependencies and other supporting libraries.

To see all the list of supporting libraries see the pom.xml of org.springframework.boot:springboot-dependencies:1.3.6.RELEASE maven module.

• The parent module spring-boot-starter-parent already included the most commonly used plugin such as maven-jar-plugin, maven-surefire-plugin, maven-war-plugin, execmaven-plugin, maven-resources-plugin with sensible defaults.

• In addition to the above mentioned plugins, module spring-boot-starter-parent module also configured the spring-boot-maven-plugin which will be used to build fat jar. We will look into spring-boot-maven-plugin in more details in further sections in this chapter.

2. We have selected only web starter but test starter also be included by default.

3. We have selected 1.8 as the Java Version, hence the property <java.version>1.8</java.version> is included. This java.version value will be used to configure the JDK version for maven compiler in spring-boot-starter-parent module.

```
<maven.compiler.source>${java.version}</maven.compiler.source>

<maven.compiler.target>${java.version}</maven.compiler.target>
```

4. Application EntryPoint class SpringbootBasicApplication.java

A SpringBoot jar type module will have an application entry point Java class with public static void main(String[] args) method which you can run to start the application.

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Here SpringbootBasicApplication class is annotated with @SpringBootApplication annotation which is a composed annotation.

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {
    ....
    ....
}
```

Let's explore the meaning of these annotations.

1. **@Configuration** annotation indicates that this class is a Spring configuration class.
2. **@ComponentScan** annotation enables component scanning for Spring beans in the package in which the current class is defined
3. **@EnableAutoConfiguration** annotation is the one which triggers all the SpringBoot's auto configuration mechanism.

We are bootstrapping the application by calling SpringApplication.run(SpringbootBasicApplication.class, args) in our main() method. We can pass one or more Spring Configuration classes to SpringApplication.run() method. But if we have our application entry-point class in root package then it is sufficient to pass our application entry class only which takes care of scanning other Spring Configuration classes with in all the sub-packages.

Step 2: Now create a simple SpringMVC controller, HomeController, as follows:

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

@Controller
public class HomeController
{
    @RequestMapping("/")
    public String home(Model model) {
        return "index.html";
    }
}
```

This is a simple SpringMVC controller with one request handler method for URL "/" which returns the view name "index.html".

Step 3: Create a HTML view index.html

By default SpringBoot serves the static content from src/main/public/ and src/main/static/ directories.

So create index.html in src/main/public/ as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>Home</title>
</head>
<body>
    <h2>Hello World!!</h2>
</body>
</html>
```

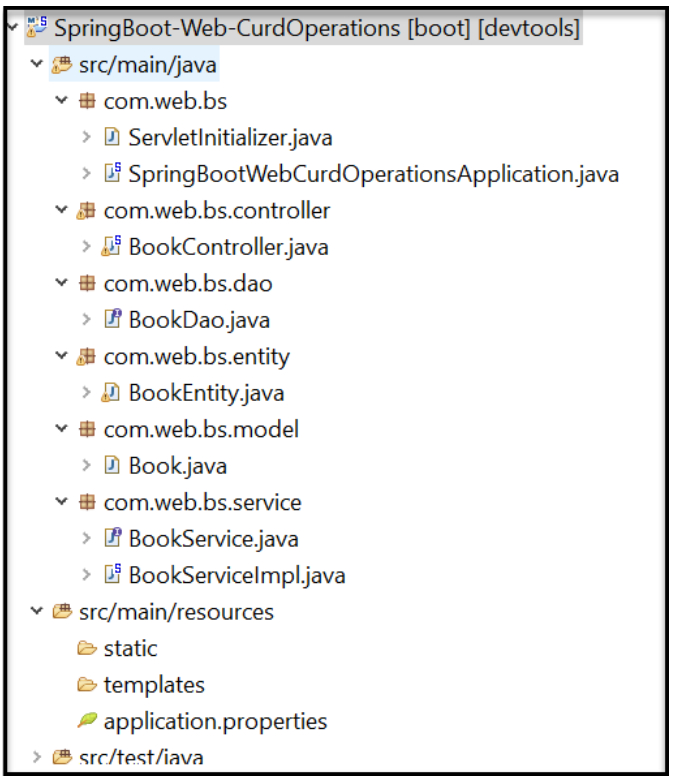
Now from your IDE run the SpringbootBasicApplication.java main() method as stand-alone Java class which will start the embedded tomcat server on port 8080 and point your browser to http://localhost:8080/. You should be able to see the response Hello World!!.

Limitations of Spring Boot

It is very tough and time consuming process to convert existing or legacy Spring Framework projects into Spring Boot Applications. It is applicable only for brand new/Greenified Spring Projects.

CURD Operations Application Development using Spring Boot and Spring Data JPA

Step-1: Create Spring Starter project using STS IDE and (select required maven dependencies)



----------pom.xml----------

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

        http://maven.apache.org/xsd/maven-4.0.0.xsd">

        <modelVersion>4.0.0</modelVersion>
        <groupId>com.web.bs</groupId>
        <artifactId>SpringBoot-Jpa-DBOperations</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>war</packaging>
        <name>SpringBoot-Web-CurdOperations</name>
        <description>Demo project for Spring Boot</description>
        <parent>

                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.5.14.RELEASE</version>
                <relativePath /> <!-- lookup parent from repository -->
        </parent>
```

```
	<properties>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
		<java.version>1.8</java.version>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-jetty</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-devtools</artifactId>
		</dependency>

		<!-- JSTL -->
		<dependency>
			<groupId>javax.servlet</groupId>
			<artifactId>jstl</artifactId>
		</dependency>

		<!-- To compile JSP files -->
		<dependency>
			<groupId>org.apache.tomcat.embed</groupId>
			<artifactId>tomcat-embed-jasper</artifactId>
		</dependency>

		<!-- Oracle -->
		<dependency>
			<groupId>com.oracle</groupId>
			<artifactId>ojdbc14</artifactId>
			<version>1.1.1</version>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
		</dependency>
	</dependencies>
</project>.
```

----------End of Pom.xml----------

Step-2 : Create Entity class and Jpa Repository for Entity for Persistence layer to perform DB operations.

```java
@Entity
@Table(name = "BOOK_STORE_TBL")
public class BookEntity implements Serializable {

	@Id
	@GeneratedValue
	@Column(name = "BOOK_ID")
	private Integer bookId;

	@Column(name = "BOOK_NAME")
	private String bookName;

	@Column(name = "BOOK_ISBN")
	private String bookIsbn;

	@Column(name = "BOOK_PRICE")
	private Double bookPrice;

	@Column(name = "ACTIVE_SW")
	private String activeSw;

	@CreationTimestamp
	@Temporal(TemporalType.DATE)
	private Date createDt;

	@UpdateTimestamp
	@Temporal(TemporalType.DATE)
	private Date updateDt;

	//setters & getters
}
```

BookEntity.java

```java
/**
 * This interface implemented methods performs CURD operations with BOOK_STORE_TBL
 *
 * @author Ashok
 *
 */
public interface BookDao extends JpaRepository<BookEntity, Serializable> {
    //Spring Data provides implementation for our BookDao interface
}
```

Step-3: Create Model class to transfer data from controller to service and vice versa

```java
public class Book {

	private Integer bid;
	private String bname;
	private String isbn;
	private Double price;

	//setters & getters
}
```

Book.java

Step-4: Create Service and ServiceImpl classes to perform business operations (Service will talk with Dao)

```
/**
 *
 * @author Ashok
 *
 */
public interface BookService {

    public boolean insert(Book book);

    public List<Book> findAllBooks();

    public boolean delete(Integer bid);

    public Book edit(Integer bid);

    public boolean update(Book b);

}
```

----------BookServiceImpl.java----------

```
@Service("bookService")
public class BookServiceImpl implements BookService {

	@Autowired(required = true)
	private BookDao bookDao;

	@Override
	public boolean insert(Book book) {

		// Converting model obj to entity obj
		BookEntity entity = new BookEntity();
		entity.setBookName(book.getBname());
		entity.setBookIsbn(book.getIsbn());
		entity.setBookPrice(book.getPrice());
		entity.setActiveSw("Y");

		// calling dao method
		BookEntity be = bookDao.save(entity);

		return (be.getBookId() != null) ? true : false;
	}

	@Override
	public List<Book> findAllBooks() {
		List<BookEntity> entities = bookDao.findAll();
		List<Book> books = new ArrayList<Book>();

		if (!entities.isEmpty()) {
			for (BookEntity entity : entities) {
				if ("Y".equals(entity.getActiveSw())) {
					Book b = new Book();

					b.setBid(entity.getBookId());
					b.setBname(entity.getBookName());
					b.setIsbn(entity.getBookIsbn());
					b.setPrice(entity.getBookPrice());

					books.add(b);
				}
			}
		}
		return books;
	}
```

```
	/**
	 * This method performs soft delete
	 *
	 * @param bid
	 * @return
	 */
	public boolean delete(Integer bid) {
		BookEntity entity = bookDao.findOne(bid);
		entity.setActiveSw("N");
		BookEntity be = bookDao.save(entity);
		return (be != null) ? true : false;
	}

	@Override
	public Book edit(Integer bid) {
		BookEntity entity = bookDao.findOne(bid);

		// Converting entity to model obj
		Book b = new Book();
		b.setBid(entity.getBookId());
		b.setBname(entity.getBookName());
		b.setIsbn(entity.getBookIsbn());
		b.setPrice(entity.getBookPrice());

		return b;
	}

	public boolean update(Book b) {

		BookEntity entity = bookDao.findOne(b.getBid());

		entity.setBookIsbn(b.getIsbn());
		entity.setBookName(b.getBname());
		entity.setBookPrice(b.getPrice());

		BookEntity be = bookDao.save(entity);

		return (be != null) ? true : false;
	}

}
```

----------End of ServiceImpl class----------

Step-4: Create Controller class to handle user requests

----------BookController.java----------

```
@Controller
public class BookController {

	@Autowired(required = true)
	private BookService bookService;

	@ModelAttribute("book")
	public Book bookForm() {
		return new Book();
	}

	@RequestMapping(value = "/index", method = RequestMethod.GET)
	public String index() {
		return "index";
	}

	@RequestMapping(value = "/store", method = RequestMethod.POST)
	public String storeBook(Model model, @ModelAttribute("book") Book b) {
		// calling service method
		boolean isInserted = bookService.insert(b);

		if (isInserted) {
```

```
                model.addAttribute("sMsg", "Book Stored Successfully");
            } else {
                model.addAttribute("eMsg", "Failed to store");
            }

            return "index";
        }

        @RequestMapping(value = "/retriveBooks", method = RequestMethod.GET)
        public String viewBooks(Model model) {
            List<Book> booksList = bookService.findAllBooks();
            model.addAttribute("books", booksList);
            return "viewBooks";
        }

        @RequestMapping("/delete")
        public String deleteBook(Model model, HttpServletRequest req) {
            boolean isDeleted = false;
            String bid = req.getParameter("bid");
            if (bid != null && !bid.equals("")) {
                int bookId = Integer.parseInt(bid);
                isDeleted = bookService.delete(bookId);
            }
            return "redirect:retriveBooks";
        }

        @RequestMapping("/edit")
        public String edit(Model model, HttpServletRequest req) {

            String bid = req.getParameter("bid");

            if (null != bid && !"".equals(bid)) {
                int bookId = Integer.parseInt(bid);
                Book b = bookService.edit(bookId);
                model.addAttribute("book", b);
            }
            return "editBook";
        }

        @RequestMapping(value = "/update", method = RequestMethod.POST)
        public String update(Model model, @ModelAttribute("book") Book b) {
            boolean isUpdated = bookService.update(b);
            if (isUpdated) {
                model.addAttribute("sMsg", "Updated Successfully..");
            } else {
                model.addAttribute("eMsg", "Failed to update..");
            }
            return "editBook";
        }
}
```

----------End Of BookController.java----------

Step-5 : Create view files

Index.jsp → to display first page to add the book
viewBooks.jsp → To display all books which are active in table
editBook.jsp → To display book details in editable format

----------index.jsp----------

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" isELIgnored="false"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Book Store</title>
</head>
<body>
	<h3>Store Book Info</h3>

	<font color="red">${eMsg}</font>
	<font color="green">${sMsg}</font>

	<form:form action="store" method="POST" modelAttribute="book">
		<table>
			<tr>
				<td>Book Name :</td>
				<td><form:input path="bname" /></td>
			</tr>
			<tr>
				<td>Book Isbn:</td>
				<td><form:input path="isbn" /></td>
			</tr>
			<tr>
				<td>Book Price:</td>
				<td><form:input path="price" /></td>
			</tr>
			<tr>
				<td><input type="reset" value="Reset" /></td>
				<td><input type="Submit" value="Submit" /></td>
			</tr>
		</table>
	</form:form>

	<a href="retriveBooks">View All Books</a>
</body>
</html>

-----------------------------------ViewBooks.jsp-----------------------------------------------------
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1" isELIgnored="false"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>View Books</title>
<script>
	function confirmDelete() {

		var status = confirm("Are you sure, you want to delete?");
		if (status) {
			return true;
		} else {
			return false;
		}
	}
</script>

</head>
<body>

	<a href="index">+Add new book</a>
	<table>
		<thead>
			<tr>
				<th>S.No</th>
				<th>Book Name</th>
```

```
<th>Book Isbn</th>        <    th>Book Price</th>
```

```
                        <th>Action</th>
                    </tr>
                </thead>
                <tbody>
                    <c:forEach items="${books}" var="book" varStatus="index">
                        <tr>
                            <td><c:out value="${index.count}" /></td>
                            <td><c:out value="${book.bname }" /></td>
                            <td><c:out value="${book.isbn }" /></td>
                            <td><c:out value="${book.price }" /></td>
                            <td><a href="edit?bid=${book.bid}">Edit</a> | <a
                                href="delete?bid=${book.bid}" onclick="return
confirmDelete()">Delete

                            </a>
                        </tr>
                    </c:forEach>
                </tbody>
        </table>
</body>
</html>

----------------------------------------editBook.jsp----------------------------------------------------
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1" isELIgnored="false"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Book Store</title>
</head>
<body>
        <h3>Update Book Info</h3>

        <font color="red">${eMsg}</font>
        <font color="green">${sMsg}</font>

        <form:form action="update" method="POST" modelAttribute="book">
                <table>
                    <tr>
                        <td>Book ID :</td>
                        <td><c:out value="${book.bid}" /> <input type="hidden"
                                name="bid" value="${book.bid}" /></td>
                    </tr>

                    <tr>
                        <td>Book Name :</td>
                        <td><form:input path="bname" /></td>
                    </tr>
                    <tr>
                        <td>Book Isbn:</td>
                        <td><form:input path="isbn" /></td>
                    </tr>
                    <tr>
                        <td>Book Price:</td>
                        <td><form:input path="price" /></td>
                    </tr>
                    <tr>
                        <td><input type="Submit" value="Update" /></td>
                    </tr>
                </table>
```

```
</form:form>
<a href="retriveBooks">View All Books</a>
```

```
</body>
</html>
```

----------End of view files----------

Below is the application.properties file

application.properties

```
#Server Port No
server.port=7070

server.context-path=/SpringBootApp

#View Resolver Properties
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp

## Db Details
spring.datasource.url=jdbc:oracle:thin:@localhost:1521/XE
spring.datasource.username=system
spring.datasource.password=admin
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

##ORM Details
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.OracleDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
```

ServletInitializer.java

```
package com.web.bs;

import org.springframework.boot.builder.SpringApplicationBuilder;

public class ServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SpringBootWebCurdOperationsApplication.class);
    }
}
```

SpringBootWebCurdOperationsApplication.java

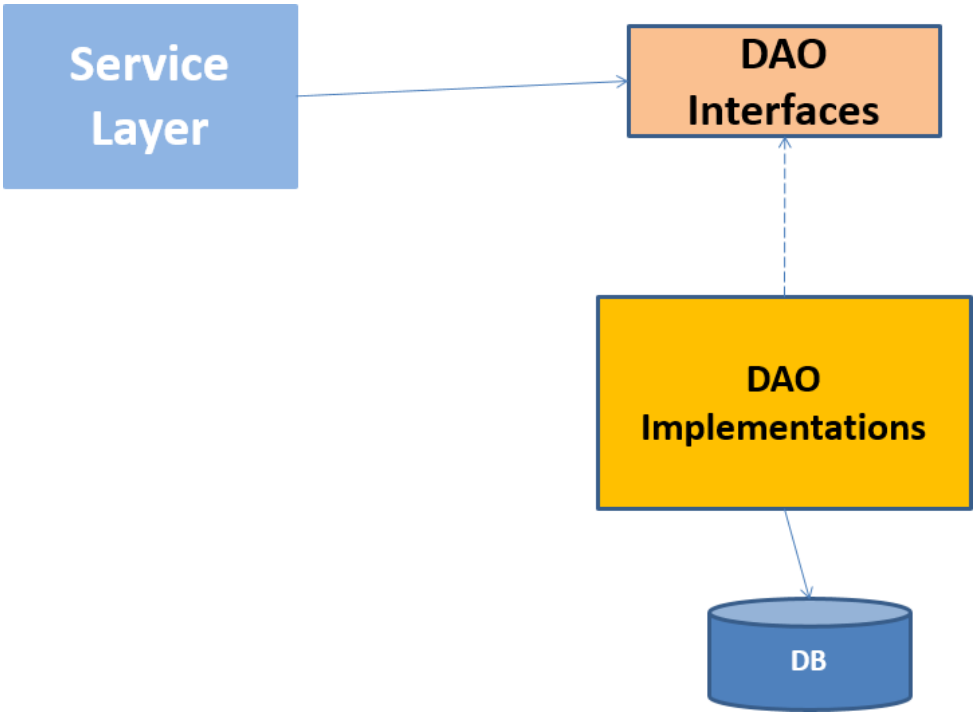
```
package com.web.bs;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class SpringBootWebCurdOperationsApplication {

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

Spring Data JPA

Introduction

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to perform CURD operations and to execute simple queries as well as to perform pagination and auditing.

In traditional approach for persistence layer we will write a Dao interface and we will write DaoImpl class to perform Persistence operations. In this approach lot of DaoImpl classes will contain boiler-plate code. As the DAO layer usually consists of a lot of boilerplate code that can and should be simplified.



To simply boiler-plate code in DAO layer we can use Spring Data. Spring Data makes it easier to create Spring driven applications that use new ways to access data, such as non-relational databases, map-reduction frameworks, cloud services, as well as well-advanced relational database support.

Some of the cool features provided by Spring Data JPA are:

- **Create and support repositories created with Spring and JPA**
- **Support Query DSL and JPA queries**
- **Audit of domain classes**
- **Support for batch loading, sorting, dynamical queries**
- **Supports XML mapping for entities**
- **Reduce code size for generic CRUD operations by using CrudRepository**

The Spring Data generated DAO – No More DAO Implementations

As discussed in an earlier article, the DAO layer usually consists of a lot of boilerplate code that can and should be simplified. The advantages of such a simplification are many: a decrease in the number of artifacts that need to be defined and maintained, consistency of data access patterns and consistency of configuration.

Spring Data takes this simplification one step forward and makes it possible to remove the DAO implementations entirely – the interface of the DAO is now the only artifact that needs to be explicitly defined.

In order to start leveraging the Spring Data programming model with JPA, a DAO interface needs to extend the JPA specific Repository interface – JpaRepository. This will enable Spring Data to find this interface and automatically create an implementation for it.

By extending the interface we get the most relevant CRUD methods for standard data access available in a standard DAO out of the box.

Custom Access Method and Queries

As discussed, by implementing one of the Repository interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented.

To define more specific access methods, Spring JPA supports quite a few options – you can:

- **simply define a new method in the interface**
- **provide the actual JPQ query by using the @Query annotation**
- **use the more advanced Specification and Querydsl support in Spring Data**
- **define custom queries via JPA Named Queries**

The third option – the Specifications and Querydsl support – is similar to JPA Criteria but using a more flexible and convenient API – making the whole operation much more readable and reusable. The advantages of this API will become more pronounced when dealing with a large number of fixed queries that could potentially be more concisely expressed through a smaller number of reusable blocks that keep occurring in different combinations.

This last option has the disadvantage that it either involves XML or burdening the domain class with the queries.

Automatic Custom Queries

When Spring Data creates a new Repository implementation, it analyses all the methods defined by the interfaces and tries to automatically generate queries from the method names. While this has some limitations, it is a very powerful and elegant way of defining new custom access methods with very little effort.

Let's look at an example: if the managed entity has a name field (and the Java Bean standard getName and setName methods), we'll define the findByName method in the DAO interface; this will automatically generate the correct query:

```
public interface IFooDAO extends JpaRepository< Foo, Long >{

  Foo findByName( String name );

}
```

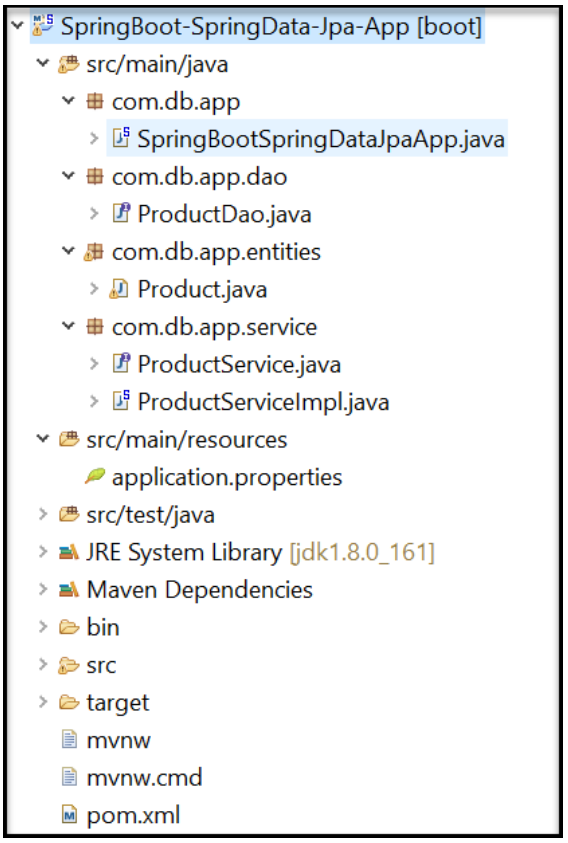
Manual Custom Queries

Let's now look at a custom query that we will define via the @Query annotation:

```
@Query("SELECT f FROM Foo f WHERE LOWER(f.name) = LOWER(:name)")
Foo retrieveByName(@Param("name") String name);
```

Spring Data JPA application with Spring Boot + MYSQL DB + Maven

Step-1: Create Spring Boot Application



Step-2: Add Below dependencies in project pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
</dependencies>
```

Note : Added mysql-connector-java to connect with MySQL database.

Step-3: Configure Database properties and ORM specific properties in application.properties file under src/main/resource folder

application.properties

```
## Db Details
spring.datasource.url=jdbc:mysql://localhost:3306/IEAppDB
spring.datasource.username=root
spring.datasource.password=ashok_123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

##ORM Details
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
```

Step-4: Create Entity class and Map with Database table

Product.java

```
package com.db.app.entities;

import java.io.Serializable;

@Entity
@Table(name = "PRODUCT_TBL")
public class Product implements Serializable {

    private static final long serialVersionUID = -2812313882891523128L;

    @Id
    @GeneratedValue
    private Integer pid;
    private String pname;
    private Double price;

    // setters & getters
    // toString()
}
```

Step-5: Create DAO interface using JpaRepository interface

ProductDao.java

```
package com.db.app.dao;

import java.io.Serializable;

@Repository("prodDao")
public interface ProductDao extends JpaRepository<Product, Serializable> {

    @Query("select pname from Product")
    public List<String> findProductNames();
}
```

Note : Spring Data provides implementation for ProductDao interface.

Step-6 : Create Service interface with required methods

ProductService.java

```
package com.db.app.service;

import java.io.Serializable;
import java.util.List;

import com.db.app.entities.Product;

public interface ProductService {

    public Serializable insert(Product p);

    public Product findProductById(Integer pid);

    public List<Product> findAllProducts();

    public List<String> findAllProductNames();

}
```

Step-7 : Create ServiceImpl class (This will call Dao methods)

ProductServiceImpl.java

```
package com.db.app.service;

import java.io.Serializable;

@Service("prodService")
public class ProductServiceImpl implements ProductService {

    @Autowired(required = true)
    private ProductDao prodDao;

    @Override
    public Serializable insert(Product p) {
        return prodDao.save(p);
    }

    @Override
    public Product findProductById(Integer pid) {
        return prodDao.findOne(pid);
    }

    @Override
    public List<Product> findAllProducts() {
        return prodDao.findAll();
    }

    @Override
    public List<String> findAllProductNames() {
        return prodDao.findProductNames();
    }
}
```

Step-8: Below is the main class which generated by STS IDE when SpringBoot project is created. This is entry for Spring Boot project execution.

Getting Service object and calling service layer method which internally calls Dao methods to perform our DB operations.

SpringBootSpringDataJpaApp.java

```java
@SpringBootApplication
public class SpringBootSpringDataJpaApp {

    public static void main(String[] args) {
        ConfigurableApplicationContext ctx = SpringApplication.run(SpringBootSpringDataJpaApp.class, args);
        ProductService service = ctx.getBean(ProductService.class);

        Product p = new Product();
        p.setPname("Keyboard");
        p.setPrice(4250.00);
        Serializable ser = service.insert(p);

        if (ser != null)
            System.out.println("Record inserted");
        else
            System.out.println("Failed to insert..");

        Product pd = service.findProductById(241);
        System.out.println(pd);

        List<Product> products = service.findAllProducts();

        if (!products.isEmpty()) {
            for (Product prod : products) {
                System.out.println(prod);
            }
        }

        List<String> names = service.findAllProductNames();
        if (!names.isEmpty()) {
            for (String name : names) {
                System.out.println(name);
            }
        }
    }
}
```

Spring Security

Introduction

Spring Security is a lightweight security framework that provides authentication and authorization support in order to Secure Spring-based applications. It integrates well with Spring MVC and comes bundled with popular security algorithm implementations.

Spring Security provides comprehensive security services for J2EE-based enterprise software applications. There is a particular emphasis on supporting projects built using The Spring Framework, which is the leading J2EE solution for enterprise software development.

People use Spring Security for many reasons, but most are drawn to the project after finding the security features of J2EE's Servlet Specification or EJB Specification lack the depth required for typical enterprise application scenarios. Whilst mentioning these standards, it's important to recognize that they are not portable at a WAR or EAR level. Therefore, if you switch server environments, it is typically a lot of work to reconfigure your application's security in the new target environment. Using Spring Security overcomes these problems, and also brings you dozens of other useful, customizable security features.

As you probably know two major areas of application security are “authentication” and “authorization” (or “access-control”). These are the two main areas that Spring Security targets. “Authentication” is the process of establishing a principal is who they claim to be (a “principal” generally means a user, device or some other system which can perform an action in your application). “Authorization” refers to the process of deciding whether a principal is allowed to perform an action within your application. To arrive at the point where an authorization decision is needed, the identity of the principal has already been established by the authentication process. These concepts are common, andnot at all specific to Spring Security.

At an authentication level, Spring Security supports a wide range of authentication models. Most of these authentication models are either provided by third parties, or are developed by relevant standards bodies such as the Internet Engineering Task Force. In addition, Spring Security provides its own set of authentication features. Specifically, Spring Security currently supports authentication integration with all of these technologies:

- **HTTP BASIC authentication headers (an IEFT RFC-based standard)**
- **HTTP Digest authentication headers (an IEFT RFC-based standard)**
- **HTTP X.509 client certificate exchange (an IEFT RFC-based standard)**
- **LDAP (a very common approach to cross-platform authentication needs, especially in large environments)**
- **Form-based authentication (for simple user interface needs) etc.**

History

The project was started in late 2003 as 'Acegi Security' (pronounced Ah-see-gee) by Ben Alex, with it being publicly released under the Apache License in March 2004. Subsequently, Acegi was incorporated into the spring portfolio as Spring Security, an official spring sub-project. The first public release under the new name was Spring Security 2.0.0 in April 2008, with commercial support and training available from Spring Source.

Getting Spring Security

For Non-Maven Projects we have to add below jars in build path

Spring-Security-Config.{version}.jar
Spring-Security-core.{version}.jar
Spring-Security-web.{version}.jar

For Maven project, add below dependencies in pom.xml file

```
<dependency>
	<groupId>org.springframework.security</groupId>
	<artifactId>spring-security-core</artifactId>
	<version>5.0.5.RELEASE</version>
</dependency>

<dependency>
	<groupId>org.springframework.security</groupId>
	<artifactId>spring-security-config</artifactId>
	<version>5.0.5.RELEASE</version>
</dependency>

<dependency>
	<groupId>org.springframework.security</groupId>
	<artifactId>spring-security-web</artifactId>
	<version>5.0.5.RELEASE</version>
</dependency>
```

Core - spring-security-core.jar

Contains core authentication and access-control classes and interfaces, remoting support and basic provisioning APIs. Required by any application which uses Spring Security. Supports standalone applications, remote clients, method (service layer) security and JDBC user provisioning. Contains the top-level packages:

- **org.springframework.security.core**
- **org.springframework.security.access**
- **org.springframework.security.authentication**
- **org.springframework.security.provisioning**
- **org.springframework.security.remoting**

Web - spring-security-web.jar

Contains filters and related web-security infrastructure code. Anything with a servlet API dependency. You'll need it if you require Spring Security web authentication services and URL-based access-control. The main package is

- **org.springframework.security.web.**

Config - spring-security-config.jar

Contains the security namespace parsing code (and hence nothing that you are likely yo use directly in your application). You need it if you are using the Spring Security XML namespace for configuration. The main package is

- **org.springframework.security.config.**

Security Namespace Configuration

Namespace configuration has been available since version 2.0 of the Spring framework. It allows you to supplement the traditional spring beans application context syntax with elements from additional XML schema.

To start using the security namespace in your application context, you first need to make sure that the spring-security-config jar is on your classpath. Then all you need to do is add the schema declaration to your application context file:

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="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-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.2.xsd">

    ...
    ...

</beans:beans>
```

Design of the Namespace

The namespace is designed to capture the most common uses of the framework and provide a simplified and concise syntax for enabling them within an application. The design is based around the large-scale dependencies within the framework, and can be divided up into the following areas:

Web/HTTP Security - the most complex part. Sets up the filters and related service beans used to apply the framework authentication mechanisms, to secure URLs, render login and error pages and much more.

Business Object (Method) Security - options for securing the service layer.

Authentication Manager - handles authentication requests from other parts of the framework.

AccessDecisionManager - provides access decisions for web and method security. A default one will be registered, but you can also choose to use a custom one, declared using normal spring bean syntax.

Authentication Providers - mechanisms against which the authentication manager authenticates users. The namespace provides supports for several standard options and also a means of adding custom beans declared using a traditional syntax.

UserDetailsService - closely related to authentication providers, but often also required by other beans.

Getting Started with Security Namespace Configuration

Spring security is defined in an XML document, just like maven configuration is defined in a pom.xml file. It has a namespace (i.e., a set of XML tag elements) which can be used to activate or configure Spring security features. Again, read the spring security appendix to learn about these in details. It is a must to understand Spring Security.

Some of its main elements are:

<html auto-config='true'> - It is the parent element of web related configuration elements. It creates the filter chain proxy bean called springSecurityFilterChain for security. It has an auto-config attribute, which can be set to install the default spring security configuration elements.

<access-denied-handler> - Can be used to set the default error page for access denials.

<intercept-url pattern="/" access="ROLE_USER"> - This element creates a relationship between a set of URLs and the required access role to visit these pages.**

requires-channel - This is an <intercept-url> attribute which can be used to require the usage of https to access a set of URLs (i.e., secured channels).

<form-login> - Can be used to define the login page URL, the URL for login processing, the target URL after login, the login failure URL, etc...

<remember-me> - If a user is not authenticated, and 'remembered' information about the user is available (for example, from a cookie), it will be used.

<session-management> and <concurrency-control> - To implement session management strategies.

<logout> - To configure the default logout page.

<http-firewall> - To implement a firewall filter.

<authentication-manager> - A required configuration element. It creates a provider manager. The child elements are <authentication-provider>.

<authentication-provider> - Can be used to create an in-memory authentication provider. The children <user-service> and <user> elements can be used to define user login-password combinations. Other types of authentication providers can be configured too.

<password-encoder> - If the users' login and password are stored in a database (for example), one can use this configuration element to specify how the password should be encrypted.

web.xml Configuration

The first thing we need to do is add the following filter declaration in web.xml file:

web.xml

```
<web-app>
    <!-- Spring Security Filter Configuration -->
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Delegating Filter Proxy – A servlet filter capturing every user requests and sending them to the configured security filters to make sure access is authorized.

This provides a hook into the Spring Security web infrastructure. DelegatingFilterProxy is a Spring Framework class which delegates to a filter implementation which is defined as a spring bean in your application context. In this case, the bean is named "springSecurityFilterChain", which is an internal infrastructure bean created by the namespace to handle web security. Note that you should not use this bean name yourself. Once you've added this to your web.xml, you're ready to start editing your application context file. Web security services are configured using the <http> element.

A Minimal <http> Configuration

All we need to enable web security to begin with is

```
<http auto-config='true'>
      <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

Which says that we want all URLs within our application to be secured, requiring the role ROLE_USER to access them. The <http> element is the parent for all web-related namespace functionality.

The <intercept-url> element defines a pattern which is matched against the URLs of incoming requests using an ant path style syntax.

The access attribute defines the access requirements for requests matching the given pattern.

With the default configuration, this is typically a comma-separated list of roles, one of which a user must have to be allowed to make the request. The prefix "ROLE_" is a marker which indicates that a simple comparison with the user's authorities should be made. In other words, a normal role-based check should be used. Access-control in Spring Security is not limited to the use of simple roles (hence the use of the prefix to differentiate between different types of security attributes).

Note

You can use multiple <intercept-url> elements to define different access requirements for different sets of URLs, but they will be evaluated in the order listed and the first match will be used. So you must put the most specific matches at the top. You can also add a method attribute to limit the match to a particular HTTP method (GET, POST, PUT etc.). If a request matches multiple patterns, the method-specific match will take precedence regardless of ordering.

To add some users, you can define a set of test data directly in the namespace:

```
<authentication-manager>
    <authentication-provider>
        <user-service>
            <user name="ashok" password="abc@123" authorities="ROLE_USER" />
            <user name="Smith" password="def@456" authorities="ROLE_USER" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

The configuration above defines two users, their passwords and their roles within the application (which will be used for access control). It is also possible to load user information from a standard properties file using the properties attribute on user-service.

Authentication Manager: Using <authentication-manager> means, it processes authentication requests via child authentication providers.

Authentication Provider: Using the <authentication-provider> element means that the user information will be used by the authentication manager to process authentication requests.

Note: We can have multiple <authentication-provider> elements to define different authentication sources and each will be consulted in turn.

What does auto-config include?

Auto-Config: A configuration element of the Spring Security namespace enabling (or not) the default configuration of Spring Security.

The auto-config attribute, as we have used it above, is just a shorthand syntax for:

```
<http>
    <form-login />
    <http-basic />
    <logout />
</http>
```

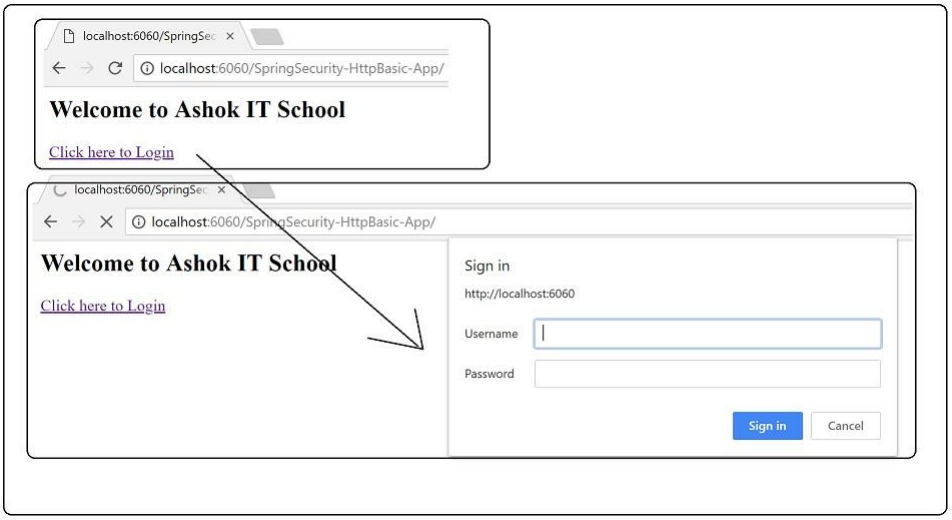
These elements are responsible for setting up form-login, basic authentication and logout handling services respectively. They each have attributes which can be used to alter their behavior.

<u><http-basic/></u>

If we want to use basic authentication, we have to write below configuration

```
<http auto-config='true'>
        <intercept-url pattern="/**" access="ROLE_USER" />
        <http-basic />
</http>
```

Basic authentication will then take precedence and will be used to prompt for a login when a user attempts to access a protected resource like below



<u><form-login/></u>

If we want to use default login page, we have to write below configuration

```
<http auto-config="true">
      <intercept-url pattern="/welcome" access="ROLE_USER" />
      <form-login/>
</http>
```

With the above configuration (<form-login/>) Spring Security will display one default login page. You might be wondering where the login form came from when you were asked to log in, because we have not written any HTML files or JSPs.

In fact, Spring Security generates one automatically, based on the features that are enabled and using standard values for the URL which processes the submitted login, the default target URL the user will be sent to after logging in and so on. However, the namespace offers plenty of support to allow you to customize these options.

Customized Login Form

If we want to user our own login page, we could use below configuration

```
<http auto-config='true'>
      <intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
      <intercept-url pattern="/**" access="ROLE_USER" />
     <form-login login-page='/login.jsp'/>
</http>
```

Note that we can still use auto-config. The form-login element just overrides the default settings. Also note that we've added an extra intercept-url element to say that any requests for the login page should be available to anonymous users. Otherwise the request would be matched by the pattern /** and it wouldn't be possible to access the login page itself! This is a common configuration error and will result in an infinite loop in the application. Spring Security will emit a warning in the log if your login page appears to be secured. It is also possible to have all requests matching a particular pattern bypass the security filter chain completely:

```
<http auto-config='true'>
   <intercept-url pattern="/css/**" filters="none"/>
   <intercept-url pattern="/login.jsp*" filters="none"/>
   <intercept-url pattern="/**" access="ROLE_USER" />
   <form-login login-page='/login.jsp'/>
</http>
```

It's important to realize that these requests will be completely oblivious to any further Spring Security web-related configuration or additional attributes such as requires-channel, so you will not be able to access information on the current user or call secured methods during the request. Use access='IS_AUTHENTICATED_ANONYMOUSLY' as an alternative if you still want the security filter chain to be applied.

Note

Using filters="none" operates by creating an empty filter chain in Spring Security's FilterChainProxy, whereas the access attributes are used to configure the FilterSecurityInterceptor in the single filter chain which is created by the namespace configuration. The two are applied independently, so if you have an access constraint for a sub-pattern of a pattern which has a filters="none" attribute, the access constraint will be ignored, even if it is listed first. It isn't possible to apply a filters="none" attribute to the pattern /** since this is used by the namespace filter chain. In version 3.1 things are more flexible. You can define multiple filter chains and the filters attribute is no longer supported.

Using other Authentication Providers

In practice we will need a more scalable source of user information than a few names added to the application context file. Most likely we want to store application user information in something like a database or an LDAP server. If we have a custom implementation of Spring Security's UserDetailsService, called "myUserDetailsService" in our application context, then we can authenticate against this using

```
<authentication-manager>
        <authentication-provider user-service-ref='myUserDetailsService'/>
</authentication-manager>
```

If we want to use a database, then we have to configure jdbc-user-service like below

```
<beans:bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <beans:property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <beans:property name="url" value="jdbc:mysql://localhost:3306/mydb" />
    <beans:property name="username" value="root" />
    <beans:property name="password" value="admin" />
</beans:bean>
```

DataSource Configuration

```
<authentication-manager>
    <authentication-provider>
        <jdbc-user-service data-source-ref="dataSource"
            users-by-username-query="select username,password, enabled from users where username=?"
            authorities-by-username-query="select username, role from user_roles where username =?  " />
    </authentication-provider>
</authentication-manager>
```

Adding a Password Encoder

Often your password data will be encoded using a hashing algorithm. This is supported by the <password-encoder> element. With SHA encoded passwords, the original authentication provider configuration would look like this:

```
<authentication-manager>
    <authentication-provider>
        <password-encoder hash="sha" />
        <user-service>
            <user name="ashok" password="d7e6351eaa13189a5a3641bab846c8e8c69ba39f"
                authorities="ROLE_USER, ROLE_ADMIN" />
            <user name="smith" password="4e7421b1b8765d8f9406d87e7cc6aa784c4ab97f"
                authorities="ROLE_USER" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

Adding HTTP/HTTPS Channel Security

If our application supports both HTTP and HTTPS, and we require that particular URLs can only be accessed over HTTPS, then this is directly supported using the requires-channel attribute on <intercept-url>:

```
<http>
  <intercept-url pattern="/secure/**" access="ROLE_USER" requires-channel="https"/>
  <intercept-url pattern="/**" access="ROLE_USER" requires-channel="any"/>
  ...
 </http>
```

With this configuration in place, if a user attempts to access anything matching the "/secure/" pattern using HTTP, they will first be redirected to an HTTPS URL. The available options are "http", "https" or "any". Using the value "any" means that either HTTP or HTTPS can be used.**

If our application uses non-standard ports for HTTP and/or HTTPS, you can specify a list of port mappings as follows:

```
<http>
  ...
     <port-mappings>
            <port-mapping http="9080" https="9443"/>
      </port-mappings>

 </http>
```

Session Management

We can configure Spring Security to detect the submission of an invalid session ID and redirect the user to an appropriate URL. This is achieved through the session-management element:

```
<http>
  ...
     <session-management invalid-session-url="/sessionTimeout.htm" />

 </http>
```

Concurrent Session Control

If we wish to place constraints on a single user's ability to log in to your application, Spring Security supports this out of the box with the following simple additions.

First we need to add the following listener to your web.xml file to keep Spring Security updated about session lifecycle events:

```
<listener>
  <listener-class>
        org.springframework.security.web.session.HttpSessionEventPublisher
  </listener-class>
 </listener>
```

Then add the following lines to application context:

```
<http>
 ...
 <session-management>
    <concurrency-control max-sessions="1" />
 </session-management>

</http>
```

If we don't want a user logging in multiple times - a second login will cause the first to be invalidated. Often you would prefer to prevent a second login, in such scenario we can use below configuration

```
<http>
  ...
  <session-management>
     <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
  </session-management>

 </http>
```

The second login will then be rejected. By "rejected", we mean that the user will be sent to the authentication-failure-url if form-based login is being used. If the second authentication takes place through another non-interactive mechanism, such as "remember-me", an "unauthorized" (402) error will be sent to the client. If instead you want to use an error page, you can add the attribute session-authentication-error-url to the session-management element.

Method Security

From version 2.0 onwards Spring Security has improved support substantially for adding security to service layer methods. It provides support for JSR-250 annotation security as well as the framework's original @Secured annotation.

From 3.0 we can also make use of new expression-based annotations. We can apply security to a single bean, using the intercept-methods element to decorate the bean declaration, or we can secure multiple beans across the entire service layer using the AspectJ style pointcuts.

The <global-method-security> Element

This element is used to enable annotation-based security in an application (by setting the appropriate attributes on the element), and also to group together security pointcut declarations which will be applied across entire application context. You should only declare one <global-method-security> element. The following declaration would enable support for Spring Security's @Secured:

```
<global-method-security secured-annotations="enabled" />
```

Adding an annotation to a method (on an class or interface) would then limit the access to that method accordingly. Spring Security's native annotation support defines a set of attributes for the method. These will be passed to the AccessDecisionManager for it to make the actual decision:

```
public interface BankService {

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();

  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);

}
```

BankService.java

Support for JSR-250 annotations can be enabled using

```
<global-method-security    jsr250-annotations="enabled" />
```

These are standards-based and allow simple role-based constraints to be applied but do not have the power Spring Security's native annotations. To use the new expression-based syntax, you would use

```
<global-method-security    pre-post-annotations="enabled" />
```

And Equivalent java would be

```
public interface BankService {

  @PreAuthorize("isAnonymous()")
  public Account readAccount(Long id);

  @PreAuthorize("isAnonymous()")
  public Account[] findAccounts();

  @PreAuthorize("hasAuthority('ROLE_TELLER')")
  public Account post(Account account, double amount);

}
```

BankService.java

Expression-based annotations are a good choice if you need to define simple rules that go beyond checking the role names against the user's list of authorities. You can enable more than one type of annotation in the same application, but you should avoid mixing annotations types in the same interface or class to avoid confusion.

Adding Security Pointcuts using protect-pointcut

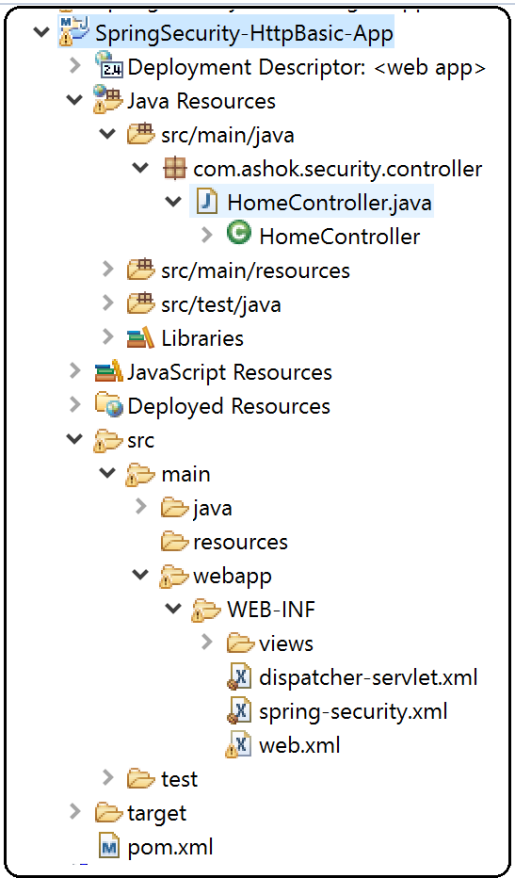
The use of protect-pointcut is particularly powerful, as it allows you to apply security to many beans with only a simple declaration. Consider the following example:

```
<global-method-security>
    <protect-pointcut expression="execution(* com.mycompany.*Service.*(..))"
                      access="ROLE_USER"/>
</global-method-security>
```

This will protect all methods on beans declared in the application context whose classes are in the com.mycompany package and whose class names end in "Service". Only users with the ROLE_USER role will be able to invoke these methods. As with URL matching, the most specific matches must come first in the list of pointcuts, as the first matching expression will be used.

Spring Security with HttpBasic – Application

Step-1: Create Maven web project with below project structure.



Step-2: Add below Maven dependencies in pom.xml

- Spring-webmvc
- Spring-security-web
- Spring-security-config
- Jstl
- servlet

```
<dependencies>
	<!-- Spring dependencies -->
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>4.3.6.RELEASE</version>
	</dependency>
	<!-- Spring Security -->
	<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-web</artifactId>
		<version>3.2.3.RELEASE</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-config</artifactId>
		<version>3.2.3.RELEASE</version>
	</dependency>
	<!-- jstl for jsp page -->
	<dependency>
		<groupId>jstl</groupId>
		<artifactId>jstl</artifactId>
		<version>1.2</version>
	</dependency>
	<!--Servlet -->
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>javax.servlet-api</artifactId>
		<version>3.0.1</version>
	</dependency>
</dependencies>
```

Maven Dependencies in pom.xml

Step-3: Create Controller class using @Controller like below with required methods

HomeController.java

```
package com.ashok.security.controller;

import java.security.Principal;

@Controller
public class HomeController {

    @RequestMapping("/")
    public String home(Model model) {
        return "home";
    }

    @RequestMapping("/welcome")
    public String welcome(Model model, Principal principle) {
        String uname = principle.getName();
        model.addAttribute("name", uname);
        model.addAttribute("message", "Spring Security Form");
        return "welcome";
    }

    @RequestMapping("/logout")
    public String logout(Model model) {
        return "home";
    }
}
```

HomeController.java

Step-4: create home.jsp to display welcome page user can select login screen (This page is accessible for all users. This is not secured page)

home.jsp

```
<%@page session="false" isELIgnored="false"%>
<html>
<body>
    <h2>Welcome to Ashok IT School</h2>
    <a href="welcome">Click here to Login</a>
</body>
</html>
```

home.jsp

Step-5: Create welcome.jsp (This is secured page). When user click on Click here to login in home page request comes to /welcome and this is secured URL hence it ask for credentials.

welcome.jsp

```
<%@page session="true" isELIgnored="false"%>
<html>
<body>
    <h1>Message : ${message}</h1>

    <b>Logged in User : ${name} </b>

    <p>
        <a href="logout">Logout</a>
    </p>
</body>
</html>
```

welcome.jsp

Note: If given user credentials are valid then only welcome.jsp contented will be rendered.

Step-6: Create dispatcher-servlet.xml to configure web components like below in WEB-INF folder

dispatcher-servlet.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"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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-4.3.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd">

    <mvc:annotation-driven />
    <context:component-scan base-package="com.ashok.security.controller" />

    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

dispatcher-servlet.xml

Step-7: Create Spring-security.xml file in WEB-INF folder

spring-security.xml

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="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-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.2.xsd">

    <http auto-config="true" create-session="never">
        <intercept-url pattern="/welcome" access="ROLE_USER" />
        <http-basic />
    </http>

    <authentication-manager>
        <authentication-provider>
            <user-service>
                <user name="ashok" password="abc@123" authorities="ROLE_USER" />
                <user name="Smith" password="def@456" authorities="ROLE_USER" />
            </user-service>
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

spring-security.xml

From the above configuration

- **/welcome URL pattern is getting intercepted. This is secured URL pattern.**
- **<http-basic /> represents windows based security.**
- **<user-service /> contains users credentials and their roles (These are in-memory credentials)**

Step-8: Configure Dispatcher Servlet, DelegatingProxyFilter, ContextLoaderListener and spring-security.xml file in web.xml file like below

web.xml

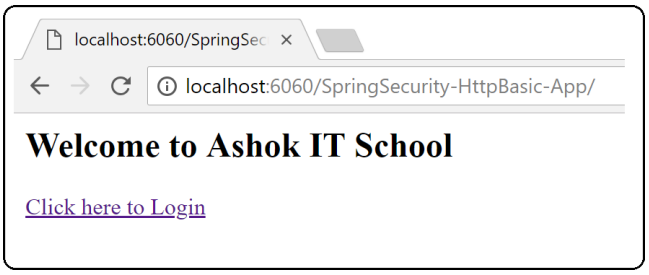
```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-security.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- Spring Security -->
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

web.xml

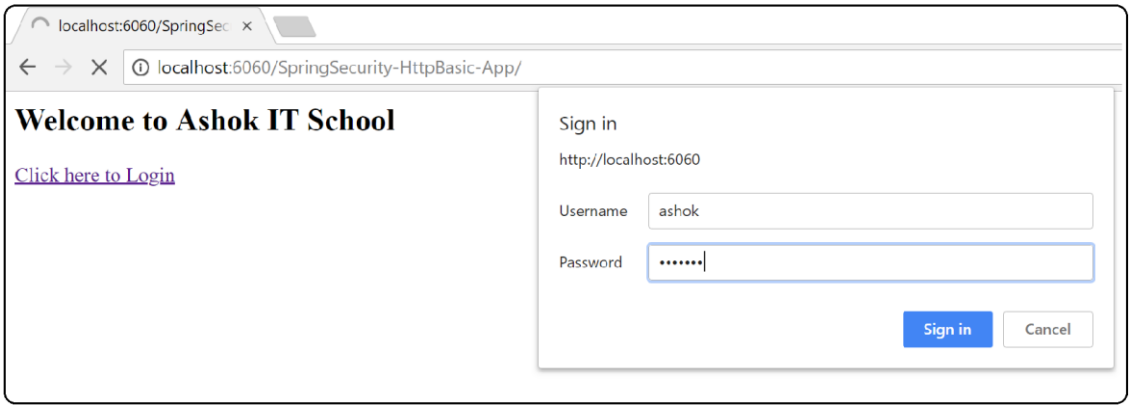
As per above configuration

- **Context-param contains spring-security.xml location**
- **ContextLoaderListener loads context-param value and starts rootApplicationContext**
- **DispatcherServlet acts as front controller in Spring MVC appplications**
- **<filter /> and <filter-mapping /> represents Spring Security Configuration**

Step-9: Deploy the application in Server (Below Screen will be displayed)



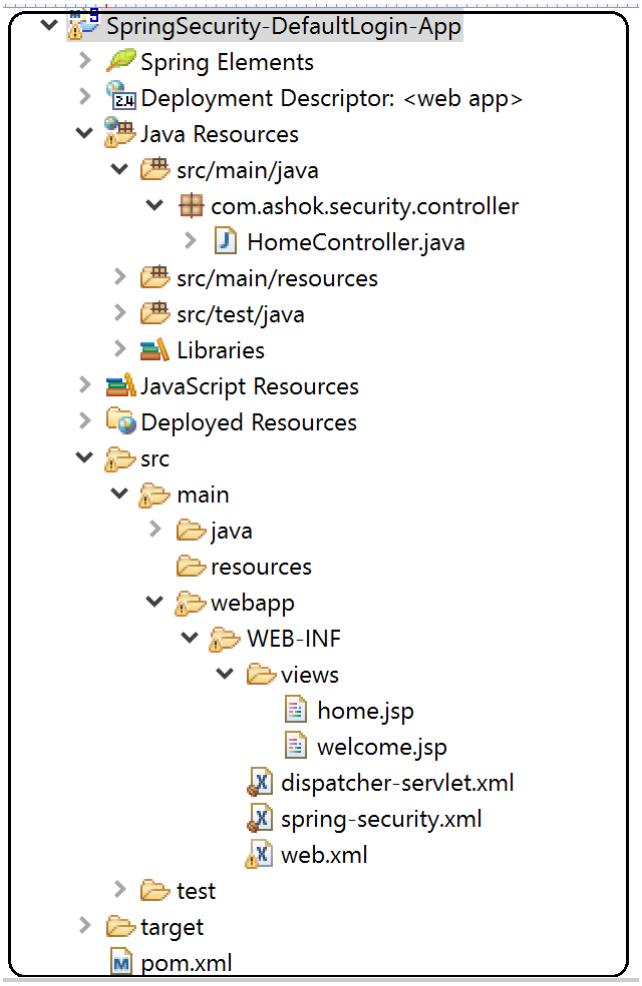
Step-10: Click on Click here to Login hyperlink (It tries to access /welcome resource but it is secured url-pattern as per our spring-security.xml hence credentials will asked to access /welcome resource) below screen will be displayed.



Step-11: Enter credentials and click on Sign in button, below screen will be displayed.

Spring Security with Default Form Login – Application

Step-1: Create Maven web project with below project structure



Step-2: Add maven dependencies in project pom.xml file

```
<dependencies>
	<!-- Spring dependencies -->
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>4.3.6.RELEASE</version>
	</dependency>
	<!-- Spring Security -->
	<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-web</artifactId>
		<version>3.2.3.RELEASE</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-config</artifactId>
		<version>3.2.3.RELEASE</version>
	</dependency>
</dependencies>
```

```
<!-- jstl for jsp page -->
<dependency>
	<groupId>jstl</groupId>
	<artifactId>jstl</artifactId>
	<version>1.2</version>
</dependency>
```

```
<!--Servlet -->
<dependency>
	<groupId>javax.servlet</groupId>
	<artifactId>javax.servlet-api</artifactId>
	<version>3.0.1</version>
</dependency>
```

Maven Dependencies in pom.xml

Step-3: Create Controller class with required methods

HomeController.java

```
package com.ashok.security.controller;

import java.security.Principal;

@Controller
public class HomeController {

    @RequestMapping("/")
    public String home(Model model) {
        return "home";
    }

    @RequestMapping("/welcome")
    public String welcome(Model model, Principal principle) {
        String uname = principle.getName();
        model.addAttribute("name", uname);
        model.addAttribute("message", "Spring Security Form");
        return "welcome";
    }

    @RequestMapping("/logout")
    public String logout(Model model) {
        return "home";
    }
}
```

HomeController.java

Step-4: create home.jsp to display welcome page user can select login screen (This page is accessible for all users. This is not secured page)

home.jsp

```
<%@page session="false" isELIgnored="false"%>
<html>
<body>
    <h2>Welcome to Ashok IT School</h2>
    <a href="welcome">Click here to Login</a>
</body>
</html>
```

home.jsp

Step-5: Create welcome.jsp (This is secured page). When user click on Click here to login in home page request comes to /welcome and this is secured URL hence it ask for credentials.

welcome.jsp

```
<%@page session="true" isELIgnored="false"%>
<html>
<body>
    <h1>Message : ${message}</h1>

    <b>Logged in User : ${name} </b>

    <p>
        <a href="logout">Logout</a>
    </p>
</body>
</html>
```

welcome.jsp

Note: If given user credentials are valid then only welcome.jsp contented will be rendered.

Step-6: Create dispatcher-servlet.xml to configure web components like below in WEB-INF folder

dispatcher-servlet.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"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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-4.3.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd">

    <mvc:annotation-driven />
    <context:component-scan base-package="com.ashok.security.controller" />

    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

dispatcher-servlet.xml

Step-7: Create spring-security.xml file in WEB-INF folder

spring-security.xml

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="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-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.2.xsd">

    <http auto-config="true">
        <intercept-url pattern="/welcome" access="ROLE_USER" />
        <form-login />
    </http>

    <authentication-manager>
        <authentication-provider>
            <user-service>
                <user name="ashok" password="abc@123" authorities="ROLE_USER" />
            </user-service>
        </authentication-provider>
    </authentication-manager>

</beans:beans>
```

spring-security.xml

Note : <form-login /> displays default login form provided by Spring Security when request made with /welcome URL pattern like below

Step-8: Configure Dispatcher Servlet, DelegatingProxyFilter, ContextLoaderListener and spring-security.xml file in web.xml file like below

web.xml

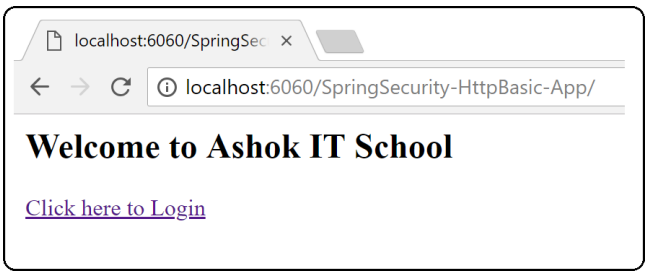
```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-security.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- Spring Security -->
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

web.xml

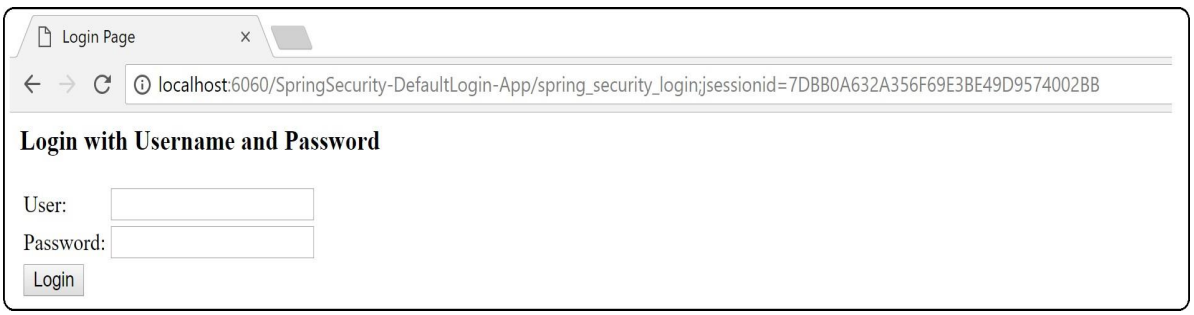
As per above configuration

- **Context-param contains spring-security.xml location**
- **ContextLoaderListener loads context-param value and starts rootApplicationContext**
- **DispatcherServlet acts as front controller in Spring MVC appplications**
- **<filter /> and <filter-mapping /> represents Spring Security Configuration**

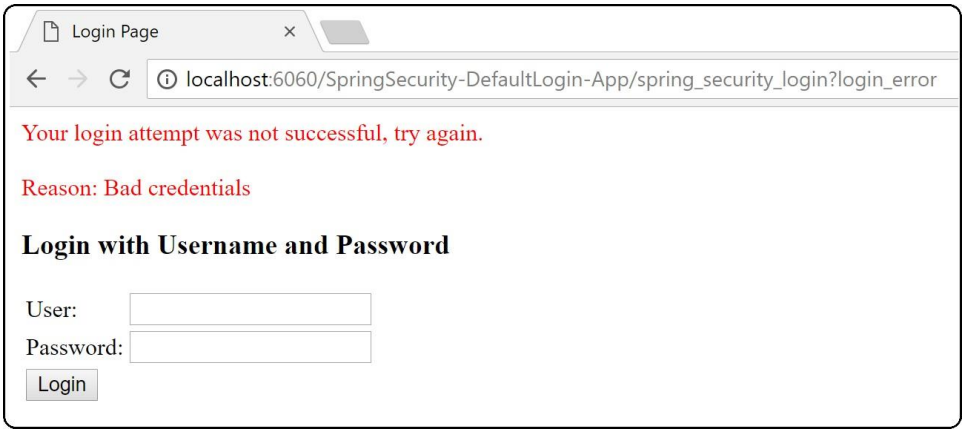
Step-9: Deploy the application in Server (Below Screen will be displayed)



Step-10: Click on Click here to Login hyperlink (It tries to access /welcome resource but it is secured url-pattern as per our spring-security.xml hence credentials will asked to access /welcome resource) below screen will be displayed.



If invalid credentials are entered, below error message will be displayed



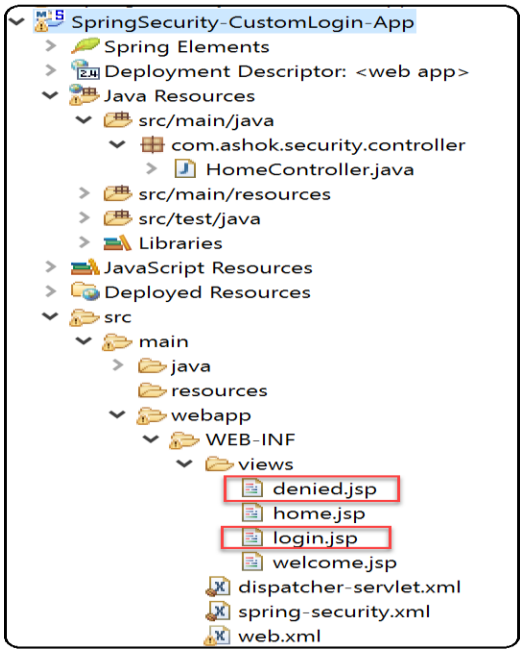
For Valid credentials, below, welcome.jsp content will be rendered like below

User: savitasoft

Password: abc@123

Spring Security with Custom Login – Application

Step1: Create Maven web application



Configure below dependencies in pom.xml

```
<dependencies>
	<!-- Spring dependencies -->
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>4.3.6.RELEASE</version>
	</dependency>
	<!-- Spring Security -->
	<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-web</artifactId>
		<version>3.2.3.RELEASE</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-config</artifactId>
		<version>3.2.3.RELEASE</version>
	</dependency>
	<!-- jstl for jsp page -->
	<dependency>
		<groupId>jstl</groupId>
		<artifactId>jstl</artifactId>
		<version>1.2</version>
	</dependency>
	<!--Servlet -->
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>javax.servlet-api</artifactId>
		<version>3.0.1</version>
	</dependency>
</dependencies>
```

Maven Dependencies in pom.xml

Step-2: Create Controller class with required methods like below

```
@Controller
public class HomeController {

	@RequestMapping("/")
	public String home(Model model) {
		return "home";
	}

	@RequestMapping("/welcome")
	public String welcome(Model model, Principal principle) {
		String uname = principle.getName();
		model.addAttribute("name", uname);
		model.addAttribute("message", "Spring Security Form");
		return "welcome";
	}

}
```

HomeController.java

```
@RequestMapping("/login")
public String login(Model model) {
	return "login";
}

@RequestMapping("/accessdenied")
public String denied(Model model) {
	model.addAttribute("error", "true");
	return "denied";
}

@RequestMapping("/logout")
public String logout(Model model) {
	return "home";
}
```

HomeController - methods

Step-4: Create view files respective to controller method return types in /WEB-INF/views/ folder

- **home.jsp : To display home page where user can click on login**
- **login.jsp : This is used to display custom login form to user to login**
- **denied.jsp : If authentication is failed this will display validation message to user**
- **welcome.jsp : If authentication is successful user will land on welcome.jsp**

```jsp
<%@page session="false" isELIgnored="false"%>
<html>
<body>
	<h2>Welcome to Ashok IT School</h2>
	<a href="welcome">Click here to Login</a>
</body>
</html>
```

home.jsp

```jsp
<form name="f" action="<c:url value='j_spring_security_check'/>"
	method="POST">
	<table>
		<tr>
			<td>Username:</td>
			<td><input type='text' name='username' /></td>
		</tr>
		<tr>
			<td>Password:</td>
			<td><input type='password' name='password'></td>
		</tr>
		<tr>
			<td colspan="2"> </td>
		</tr>
		<tr>
			<td colspan='2'><input name="submit" type="submit"
				value="Login"> <input name="reset" type="reset"></td>
		</tr>
	</table>
</form>
```

login.jsp

```jsp
<%@page session="true" isELIgnored="false"%>
<html>
<body>
	<h1>Message : ${message}</h1>
	<b>Logged in User : ${name} </b>
	<p>
		<a href="logout">Logout</a>
	</p>
</body>
</html>
```

welcome.jsp

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
<body>
	<h1 id="banner">Unauthorized Access !!</h1>
	<hr />
	<c:if test="${not empty error}">
		<div style="color: red">
			Your fake login attempt was bursted, Try again !!<br /> Caused :
			${sessionScope["SPRING_SECURITY_LAST_EXCEPTION"].message}
		</div>
	</c:if>
	<p class="message">Access denied!</p>
	<a href="login">Go back to login page</a>
</body>
</html>
```

denied.jsp

Step-5: Create Spring Security configuration file to configure authentication manager and http configuration details like below in WEB-INF folder

```xml
<http auto-config="true">
	<intercept-url pattern="/welcome" access="ROLE_USER" />
	<form-login login-page="/login"
		default-target-url="/welcome"
		authentication-failure-url="/accessdenied" />
</http>
<authentication-manager>
	<authentication-provider>
		<user-service>
		    <user name="ashok" password="abc@123" authorities="ROLE_USER,ROLE_ADMIN" />
		    <user name="john" password="def@123" authorities="ROLE_USER" />
		</user-service>
	</authentication-provider>
</authentication-manager>
```

spring-security.xml

Step-6: Create dispatcher-servlet xml with web components in /WEB-INF/ folder

- **<mvc:annotation-driven />**
- **ComponentScan basePackages="**"**
- **ViewResolver**

```xml
<mvc:annotation-driven />
<context:component-scan base-package="com.ashok.security.controller" />

<bean id="viewResolver"
	class="org.springframework.web.servlet.view.InternalResourceViewResolver">
	<property name="prefix" value="/WEB-INF/views/" />
	<property name="suffix" value=".jsp" />
</bean>
```

dispatcher-servlet.xml

Step-6: Configure listener, dispatcher servlet and filter in web.xml

- **configure context-param**
  - **Key: contextConfigLocation**
  - **Value: spring-security-xml path**
- **ContextLoaderListener to start rootAppliationContext**
- **DispatcherServlet as front controller**
- **DelegatingFilterProxy (security config)**

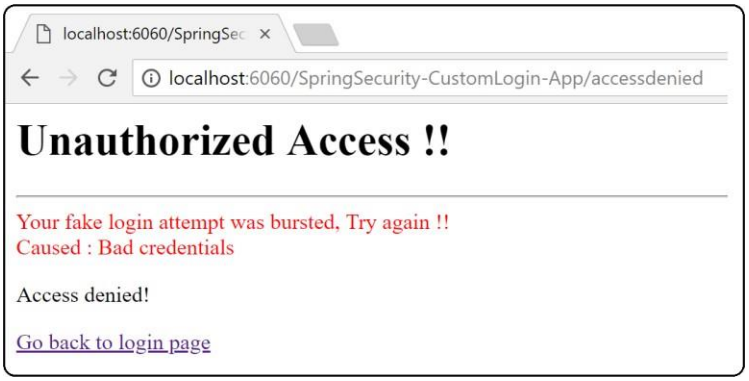
web.xml

```xml
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-security.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- Spring Security -->
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

web.xml

Step-7: Deploy the project and test it with valid and invalid credentials

For Invalid credentials, below error message will be displayed

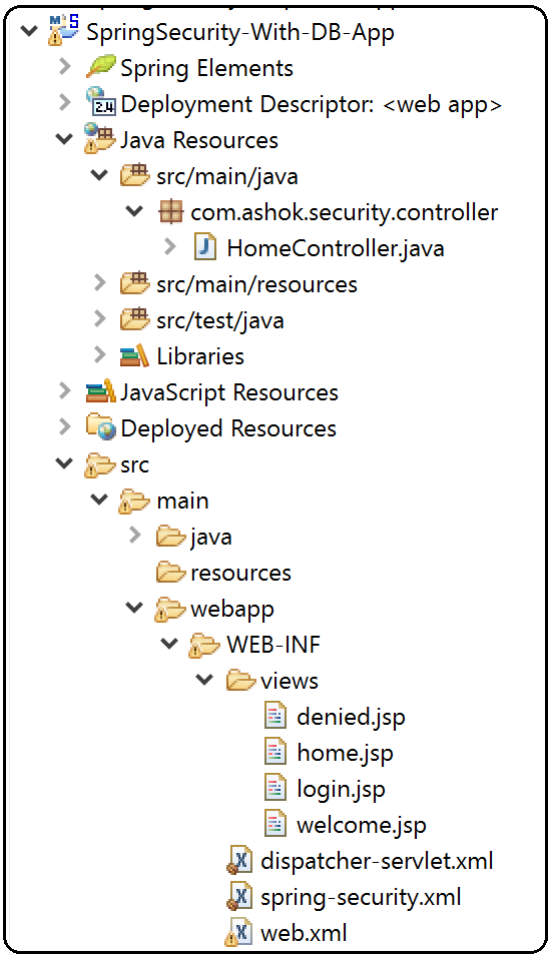


For valid credentials, below response will be displayed

Spring Security with Database Credentials – Application

Step-1: Create maven project and add below dependencies in pom.xml

- **spring-security-core**
- **spring-security-config**
- **spring-security-web**
- **spring-webmvc**
- **javax.servlet-api**
- **Jstl**
- **Jdbc driver (Oracle | MySQL)**



Step-2: In order to validate User credentials with Database, let us setup some sample data in database.

Create users table like below

```
CREATE TABLE users
  (
    username VARCHAR(45) NOT NULL ,
    password VARCHAR(45) NOT NULL ,
    enabled  TINYINT NOT NULL DEFAULT 1 ,
    PRIMARY KEY (username)
  );
```

Create USER_ROLES table like below

```
CREATE TABLE user_roles
  (
    user_role_id INT(11) NOT NULL auto_increment,
    username     VARCHAR(45) NOT NULL,
    role         VARCHAR(45) NOT NULL,
    PRIMARY KEY (user_role_id),
    UNIQUE KEY uni_username_role (role,username),
    KEY fk_username_idx (username),
    CONSTRAINT fk_username FOREIGN KEY (username) REFERENCES users (username)
  );
```

Note: username is foreign key between both tables

Insert sample data into USERS and USER_ROLES table using below queries

```
--Queries to insert sample data into USERS TABLE
INSERT INTO users(username,password,enabled) VALUES ('Ashok','123456', true);
INSERT INTO users(username,password,enabled) VALUES ('Smith','456789', true);

--Queries to insert sample data into USER_ROLES TABLE
INSERT INTO user_roles (username, role) VALUES ('Ashok', 'ROLE_USER');
INSERT INTO user_roles (username, role)  VALUES ('Ashok', 'ROLE_ADMIN');
INSERT INTO user_roles (username, role)  VALUES ('Smith', 'ROLE_USER');
```

Step-3: Create Controller class with required methods like below

```
@Controller
public class HomeController {

	@RequestMapping("/")
	public String home(Model model) {
		return "home";
	}

	@RequestMapping("/welcome")
	public String welcome(Model model, Principal principle) {
		String uname = principle.getName();
		model.addAttribute("name", uname);
		model.addAttribute("message", "Spring Security Form");
		return "welcome";
	}

}
```

HomeController.java

```
@RequestMapping("/login")
public String login(Model model) {
	return "login";
}

@RequestMapping("/accessdenied")
public String denied(Model model) {
	model.addAttribute("error", "true");
	return "denied";
}

@RequestMapping("/logout")
public String logout(Model model) {
	return "home";
}
```

HomeController - methods

Step-4: Create view files respective to controller method return types in /WEB-INF/views/ folder

- **home.jsp : To display home page where user can click on login**
- **login.jsp : This is used to display custom login form to user to login**
- **denied.jsp : If authentication is failed this will display validation message to user**
- **welcome.jsp : If authentication is successful user will land on welcome.jsp**

```
<%@page session="false" isELIgnored="false"%>
<html>
<body>
	<h2>Welcome to Ashok IT School</h2>
	<a href="welcome">Click here to Login</a>
</body>
</html>
```

home.jsp

```
<form name="f" action="<c:url value='j_spring_security_check'/>"
	method="POST">
	<table>
		<tr>
			<td>Username:</td>
			<td><input type='text' name='username' /></td>
		</tr>
		<tr>
			<td>Password:</td>
			<td><input type='password' name='password'></td>
		</tr>
		<tr>
			<td colspan="2"> </td>
		</tr>
		<tr>
			<td colspan='2'><input name="submit" type="submit"
				value="Login"> <input name="reset" type="reset"></td>
		</tr>
	</table>
</form>
```

login.jsp

```jsp
<%@page session="true" isELIgnored="false"%>
<html>
<body>
	<h1>Message : ${message}</h1>
	<b>Logged in User : ${name} </b>
	<p>
		<a href="logout">Logout</a>
	</p>
</body>
</html>
```

welcome.jsp

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
<body>
	<h1 id="banner">Unauthorized Access !!</h1>
	<hr />
	<c:if test="${not empty error}">
		<div style="color: red">
			Your fake login attempt was bursted, Try again !!<br /> Caused :
			${sessionScope["SPRING_SECURITY_LAST_EXCEPTION"].message}
		</div>
	</c:if>
	<p class="message">Access denied!</p>
	<a href="login">Go back to login page</a>
</body>
</html>
```

denied.jsp

Step-5: Create Spring Security xml with below details (in WEB-INF folder)

- **http configuration**
- **DataSource bean definition**
- **authentication-manager with jdbc-user-service**

```xml
<beans:bean id="dataSource"
	class="org.springframework.jdbc.datasource.DriverManagerDataSource">
	<beans:property name="driverClassName" value="com.mysql.jdbc.Driver" />
	<beans:property name="url" value="jdbc:mysql://localhost:3306/mydb" />
	<beans:property name="username" value="root" />
	<beans:property name="password" value="admin" />
</beans:bean>

<authentication-manager>
	<authentication-provider>
		<jdbc-user-service data-source-ref="dataSource"
			users-by-username-query="select username,password, enabled from users where username=?"
			authorities-by-username-query="select username, role from user_roles where username =?" />
	</authentication-provider>
</authentication-manager>

<http auto-config="true">
	<intercept-url pattern="/welcome" access="ROLE_USER" />
	<form-login login-page="/login"
		default-target-url="/welcome"
		authentication-failure-url="/accessdenied"
		username-parameter="username"
		password-parameter="password" />
	<logout logout-success-url="/logout" />
</http>
```

DataSource

JDBC Service

http configuration

spring-security.xml

Step-6: Create dispatcher-servlet xml with web components in /WEB-INF/ folder

- **<mvc:annotation-driven />**
- **ComponentScan basePackages="**"**
- **ViewResolver**

```
<mvc:annotation-driven />
<context:component-scan base-package="com.ashok.security.controller" />

<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/views/" />
      <property name="suffix" value=".jsp" />
</bean>
```

dispatcher-servlet.xml

Step-7: Configure listener, dispatcher servlet and filter in web.xml

- **configure context-param**
  - **Key: contextConfigLocation**
  - **Value: spring-security-xml path**
- **ContextLoaderListener to start rootAppliationContext**
- **DispatcherServlet as front controller**
- **DelegatingFilterProxy (security config)**

web.xml

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-security.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- Spring Security -->
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

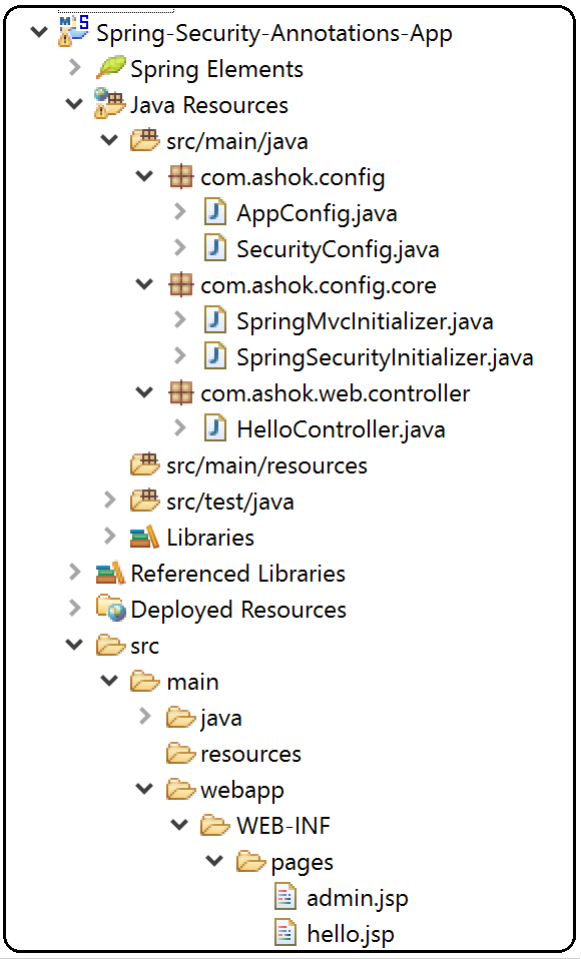
web.xml

Step-7: Deploy the project and test it.

SpringSecurity-Application-Annotations (Without XML)

Step-1: Create Maven project and add below dependencies in pom.xml

- **spring-security-core**
- **spring-security-config**
- **spring-security-web**
- **spring-webmvc**
- **javax.servlet-api**
- **Jstl**
- **Jdbc driver (Oracle | MySQL)**



Step-2: Create SecurityConfiguration class (This is replacement for spring-security.xml)

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().withUser("ashok").password("123456").roles("USER");
        auth.inMemoryAuthentication().withUser("admin").password("123456").roles("ADMIN")
        auth.inMemoryAuthentication().withUser("dba").password("123456").roles("DBA");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/admin/**")
            .access("hasRole('ROLE_ADMIN')")
            .antMatchers("/dba/**")
            .access("hasRole('ROLE_ADMIN') or hasRole('ROLE_DBA')")
            .and()
            .formLogin();
    }
}
```

SecurityConfig.java

Step-3: Create AppConfiguration class (This is replacement for dispatcher-servlet.xml)

AppConfig.java

```java
package com.ashok.config;

import org.springframework.context.annotation.Bean;

@EnableWebMvc
@Configuration
@ComponentScan({ "com.ashok.web.*" })
@Import({ SecurityConfig.class })
public class AppConfig {

    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-INF/pages/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

AppConfig.java

Step-4: Create SpringMvcInitializer class (This is replacement for web.xml file)

SpringMvcInitializer.java

```
package com.ashok.config.core;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class SpringMvcInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { AppConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
    }
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

SpringMvcInitializer.java

Step-5: Create SpringSecurityInitializer class (This is to load security configuration details)

SpringSecurityInitializer.java

```
package com.ashok.config.core;

import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

public class SpringSecurityInitializer extends AbstractSecurityWebApplicationInitializer {

}
```

SpringSecurityInitializer.java

Step-6: Create Controller class with required methods like below

```
@Controller
public class HelloController {
    @RequestMapping(value = { "/", "/welcome**" }, method = RequestMethod.GET)
    public ModelAndView welcomePage() {
        ModelAndView model = new ModelAndView();
        model.addObject("message", "This is welcome page!");
        model.setViewName("hello");
        return model;
    }
    @RequestMapping(value = "/admin**", method = RequestMethod.GET)
    public ModelAndView adminPage() {
        ModelAndView model = new ModelAndView();
        model.addObject("message", "This is protected page - Admin Page!");
        model.setViewName("admin");
        return model;
    }
    @RequestMapping(value = "/dba**", method = RequestMethod.GET)
    public ModelAndView dbaPage() {
        ModelAndView model = new ModelAndView();
        model.addObject("message", "This is protected page - Database Page!");
        model.setViewName("admin");
        return model;
    }
}
```

HelloController.java

Step-7: Create view files in WEB-INF folder

hello.jsp

```
<%@page session="false"%>
<html>
<body>
    <h1>Message : ${message}</h1>
</body>
</html>
```

hello.jsp

admin.jsp

```
<%@page session="true" isELIgnored="false"%>
<html>
<body>
    <h1>Message : ${message}</h1>
    <a href="<c:url value="/logout" />"> Logout</a>
</body>
</html>
```

admin.jsp

Step-8: Deploy the project and test it.

Spring Batch

Spring Batch Introduction

Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include automated, complex processing of large volumes of information that is most efficiently processed without user interaction. These operations typically include time based events (e.g. month-end calculations, notices or correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. Insurance benefit determination or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Batch processing is used to process billions of transactions every day for enterprises.

Spring Batch is a lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems. Spring Batch builds upon the productivity, POJO-based development approach, and general ease of use capabilities people have come to know from the Spring Framework, while making it easy for developers to access and leverage more advance enterprise services when necessary. Spring Batch is not a scheduling framework. There are many good enterprise schedulers available in both the commercial and open source spaces such as Quartz, Tivoli, Control-M, etc. It is intended to work in conjunction with a scheduler, not replace a scheduler.

Spring Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It also provides more advance technical services and features that will enable extremely high-volume and high performance batch jobs though optimization and partitioning techniques. Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information.

Background

While open source software projects and associated communities have focused greater attention on web-based and SOA messaging-based architecture frameworks, there has been a notable lack of focus on reusable architecture frameworks to accommodate Java-based batch processing needs, despite continued needs to handle such processing within enterprise IT environments. The lack of a standard, reusable batch architecture has resulted in the proliferation of many one-off, in-house solutions developed within client enterprise IT functions.

SpringSource and Accenture have collaborated to change this. Accenture's hands-on industry and technical experience in implementing batch architectures, SpringSource's depth of technical experience, and Spring's proven programming model together mark a natural and powerful partnership to create high-quality, market relevant software aimed at filling an important gap in enterprise Java. Both companies are also currently working with a number of clients solving similar problems developing Spring-based batch architecture solutions. This has provided some useful additional detail and real-life constraints helping to ensure the solution can be applied to the real-world problems posed by clients. For these reasons and many more, SpringSource and Accenture have teamed to collaborate on the development of Spring Batch.

Accenture has contributed previously proprietary batch processing architecture frameworks, based upon decades worth of experience in building batch architectures with the last several generations of platforms, (i.e., COBOL/Mainframe, C++/Unix, and now Java/anywhere) to the Spring Batch project along with committer resources to drive support, enhancements, and the future roadmap.

The collaborative effort between Accenture and SpringSource aims to promote the standardization of software processing approaches, frameworks, and tools that can be consistently leveraged by enterprise users when creating batch applications. Companies and government agencies desiring to deliver standard, proven solutions to their enterprise IT environments will benefit from Spring Batch.

Usage Scenarios

A typical batch program generally reads a large number of records from a database, file, or queue, processes the data in some fashion, and then writes back data in a modified form. Spring Batch automates this basic batch iteration, providing the capability to process similar transactions as a set, typically in an offline environment without any user interaction. Batch jobs are part of most IT projects and Spring Batch is the only open source framework that provides a robust, enterprise-scale solution.

Business Scenarios

- Commit batch process periodically
- Concurrent batch processing: parallel processing of a job
- Staged, enterprise message-driven processing
- Massively parallel batch processing
- Manual or scheduled restart after failure
- Sequential processing of dependent steps (with extensions to workflow-driven batches)
- Partial processing: skip records (e.g. on rollback)
- Whole-batch transaction: for cases with a small batch size or existing stored procedures/scripts

Technical Objectives

- Batch developers use the Spring programming model: concentrate on business logic; let the framework take care of infrastructure.
- Clear separation of concerns between the infrastructure, the batch execution environment, and the batch application.
- Provide common, core execution services as interfaces that all projects can implement.
- Provide simple and default implementations of the core execution interfaces that can be used 'out of the box'.
- Easy to configure, customize, and extend services, by leveraging the spring framework in all layers.
- All existing core services should be easy to replace or extend, without any impact to the infrastructure layer.
- Provide a simple deployment model, with the architecture JARs completely separate from the application, built using Maven.

Spring Batch Architecture

Spring Batch is designed with extensibility and a diverse group of end users in mind. The figure below shows a sketch of the layered architecture that supports the extensibility and ease of use for end-user developers.

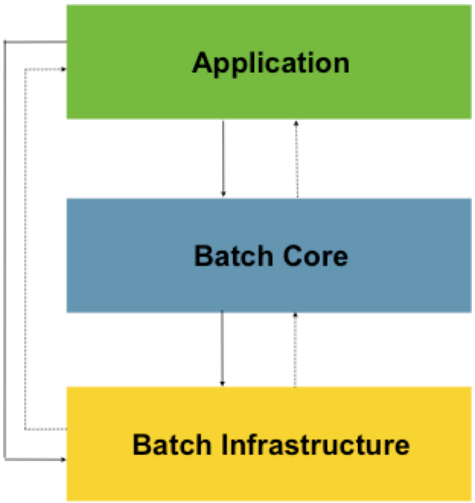


Figure 1.1: Spring Batch Layered Architecture

This layered architecture highlights three major high level components: Application, Core, and Infrastructure. The application contains all batch jobs and custom code written by developers using Spring Batch. The Batch Core contains the core runtime classes necessary to launch and control a batch job. It includes things such as a `JobLauncher`, `Job`, and `Step` implementations. Both Application and Core are built on top of a common infrastructure. This infrastructure contains common readers and writers, and services such as the `RetryTemplate`, which are used both by application developers(`ItemReader` and `ItemWriter`) and the core framework itself. (retry)

General Batch Principles and Guidelines

The following are a number of key principles, guidelines, and general considerations to take into consideration when building a batch solution.

- A batch architecture typically affects on-line architecture and vice versa. Design with both architectures and environments in mind using common building blocks when possible.
- Simplify as much as possible and avoid building complex logical structures in single batch applications.
- Process data as close to where the data physically resides as possible or vice versa (i.e., keep your data where your processing occurs).
- Minimize system resource use, especially I/O. Perform as many operations as possible in internal memory.
- Review application I/O (analyze SQL statements) to ensure that unnecessary physical I/O is avoided. In particular, the following four common flaws need to be looked for:
    - Reading data for every transaction when the data could be read once and kept cached or in the working storage;
    - Rereading data for a transaction where the data was read earlier in the same transaction;
    - Causing unnecessary table or index scans;
    - Not specifying key values in the WHERE clause of an SQL statement.
- Do not do things twice in a batch run. For instance, if you need data summarization for reporting purposes, increment stored totals if possible when data is being initially processed, so your reporting application does not have to reprocess the same data.
- Allocate enough memory at the beginning of a batch application to avoid time-consuming reallocation during the process.
- Always assume the worst with regard to data integrity. Insert adequate checks and record validation to maintain data integrity.
- Implement checksums for internal validation where possible. For example, flat files should have a trailer record telling the total of records in the file and an aggregate of the key fields.
- Plan and execute stress tests as early as possible in a production-like environment with realistic data volumes.
- In large batch systems backups can be challenging, especially if the system is running concurrent with on-line on a 24-7 basis. Database backups are typically well taken care of in the on-line design, but file backups should be considered to be just as important. If the system depends on flat files, file backup procedures should not only be in place and documented, but regularly tested as well.

Batch Processing Strategies

To help design and implement batch systems, basic batch application building blocks and patterns should be provided to the designers and programmers in form of sample structure charts and code shells. When starting to design a batch job, the business logic should be decomposed into a series of steps which can be implemented using the following standard building blocks:

- *Conversion Applications:* For each type of file supplied by or generated to an external system, a conversion application will need to be created to convert the transaction records supplied into a standard format required for processing. This type of batch application can partly or entirely consist of translation utility modules (see Basic Batch Services).

- *Validation Applications:* Validation applications ensure that all input/output records are correct and consistent. Validation is typically based on file headers and trailers, checksums and validation algorithms as well as record level cross-checks.
- *Extract Applications:* An application that reads a set of records from a database or input file, selects records based on predefined rules, and writes the records to an output file.
- *Extract/Update Applications:* An application that reads records from a database or an input file, and makes changes to a database or an output file driven by the data found in each input record.
- *Processing and Updating Applications:* An application that performs processing on input transactions from an extract or a validation application. The processing will usually involve reading a database to obtain data required for processing, potentially updating the database and creating records for output processing.
- *Output/Format Applications:* Applications reading an input file, restructures data from this record according to a standard format, and produces an output file for printing or transmission to another program or system.

Additionally a basic application shell should be provided for business logic that cannot be built using the previously mentioned building blocks.

In addition to the main building blocks, each application may use one or more of standard utility steps, such as:

- Sort - A Program that reads an input file and produces an output file where records have been re-sequenced according to a sort key field in the records. Sorts are usually performed by standard system utilities.
- Split - A program that reads a single input file, and writes each record to one of several output files based on a field value. Splits can be tailored or performed by parameter-driven standard system utilities.
- Merge - A program that reads records from multiple input files and produces one output file with combined data from the input files. Merges can be tailored or performed by parameter-driven standard system utilities.

Batch applications can additionally be categorized by their input source:

- Database-driven applications are driven by rows or values retrieved from the database.
- File-driven applications are driven by records or values retrieved from a file.
- Message-driven applications are driven by messages retrieved from a message queue.

The foundation of any batch system is the processing strategy. Factors affecting the selection of the strategy include: estimated batch system volume, concurrency with on-line or with another batch systems, available batch windows (and with more enterprises wanting to be up and running 24x7, this leaves no obvious batch windows).

Typical processing options for batch are:

- Normal processing in a batch window during off-line
- Concurrent batch / on-line processing
- Parallel processing of many different batch runs or jobs at the same time
- Partitioning (i.e. processing of many instances of the same job at the same time)
- A combination of these

The order in the list above reflects the implementation complexity, processing in a batch window being the easiest and partitioning the most complex to implement.

Some or all of these options may be supported by a commercial scheduler.

In the following section these processing options are discussed in more detail. It is important to notice that the commit and locking strategy adopted by batch processes will be dependent on the type of processing performed, and as a rule of thumb and the on-line locking strategy should also use the same principles. Therefore, the batch architecture cannot be simply an afterthought when designing an overall architecture.

The locking strategy can use only normal database locks, or an additional custom locking service can be implemented in the architecture. The locking service would track database locking (for example by storing the necessary information

in a dedicated db-table) and give or deny permissions to the application programs requesting a db operation. Retry logic could also be implemented by this architecture to avoid aborting a batch job in case of a lock situation.

1. Normal processing in a batch window For simple batch processes running in a separate batch window, where the data being updated is not required by on-line users or other batch processes, concurrency is not an issue and a single commit can be done at the end of the batch run.

In most cases a more robust approach is more appropriate. A thing to keep in mind is that batch systems have a tendency to grow as time goes by, both in terms of complexity and the data volumes they will handle. If no locking strategy is in place and the system still relies on a single commit point, modifying the batch programs can be painful. Therefore, even with the simplest batch systems, consider the need for commit logic for restart-recovery options as well as the information concerning the more complex cases below.

2. Concurrent batch / on-line processing Batch applications processing data that can simultaneously be updated by on-line users, should not lock any data (either in the database or in files) which could be required by on-line users for more than a few seconds. Also updates should be committed to the database at the end of every few transaction. This minimizes the portion of data that is unavailable to other processes and the elapsed time the data is unavailable.

Another option to minimize physical locking is to have a logical row-level locking implemented using either an Optimistic Locking Pattern or a Pessimistic Locking Pattern.

- Optimistic locking assumes a low likelihood of record contention. It typically means inserting a timestamp column in each database table used concurrently by both batch and on-line processing. When an application fetches a row for processing, it also fetches the timestamp. As the application then tries to update the processed row, the update uses the original timestamp in the WHERE clause. If the timestamp matches, the data and the timestamp will be updated successfully. If the timestamp does not match, this indicates that another application has updated the same row between the fetch and the update attempt and therefore the update cannot be performed.
- Pessimistic locking is any locking strategy that assumes there is a high likelihood of record contention and therefore either a physical or logical lock needs to be obtained at retrieval time. One type of pessimistic logical locking uses a dedicated lock-column in the database table. When an application retrieves the row for update, it sets a flag in the lock column. With the flag in place, other applications attempting to retrieve the same row will logically fail. When the application that set the flag updates the row, it also clears the flag, enabling the row to be retrieved by other applications. Please note, that the integrity of data must be maintained also between the initial fetch and the setting of the flag, for example by using db locks (e.g., SELECT FOR UPDATE). Note also that this method suffers from the same downside as physical locking except that it is somewhat easier to manage building a time-out mechanism that will get the lock released if the user goes to lunch while the record is locked.

These patterns are not necessarily suitable for batch processing, but they might be used for concurrent batch and on-line processing (e.g. in cases where the database doesn't support row-level locking). As a general rule, optimistic locking is more suitable for on-line applications, while pessimistic locking is more suitable for batch applications. Whenever logical locking is used, the same scheme must be used for all applications accessing data entities protected by logical locks.

Note that both of these solutions only address locking a single record. Often we may need to lock a logically related group of records. With physical locks, you have to manage these very carefully in order to avoid potential deadlocks. With logical locks, it is usually best to build a logical lock manager that understands the logical record groups you want to protect and can ensure that locks are coherent and non-deadlocking. This logical lock manager usually uses its own tables for lock management, contention reporting, time-out mechanism, etc.

3. Parallel Processing Parallel processing allows multiple batch runs / jobs to run in parallel to minimize the total elapsed batch processing time. This is not a problem as long as the jobs are not sharing the same files, db-tables or index spaces. If they do, this service should be implemented using partitioned data. Another option is to build an

architecture module for maintaining interdependencies using a control table. A control table should contain a row for each shared resource and whether it is in use by an application or not. The batch architecture or the application in a parallel job would then retrieve information from that table to determine if it can get access to the resource it needs or not.

If the data access is not a problem, parallel processing can be implemented through the use of additional threads to process in parallel. In the mainframe environment, parallel job classes have traditionally been used, in order to ensure adequate CPU time for all the processes. Regardless, the solution has to be robust enough to ensure time slices for all the running processes.

Other key issues in parallel processing include load balancing and the availability of general system resources such as files, database buffer pools etc. Also note that the control table itself can easily become a critical resource.

4. Partitioning Using partitioning allows multiple versions of large batch applications to run concurrently. The purpose of this is to reduce the elapsed time required to process long batch jobs. Processes which can be successfully partitioned are those where the input file can be split and/or the main database tables partitioned to allow the application to run against different sets of data.

In addition, processes which are partitioned must be designed to only process their assigned data set. A partitioning architecture has to be closely tied to the database design and the database partitioning strategy. Please note, that the database partitioning doesn't necessarily mean physical partitioning of the database, although in most cases this is advisable. The following picture illustrates the partitioning approach:

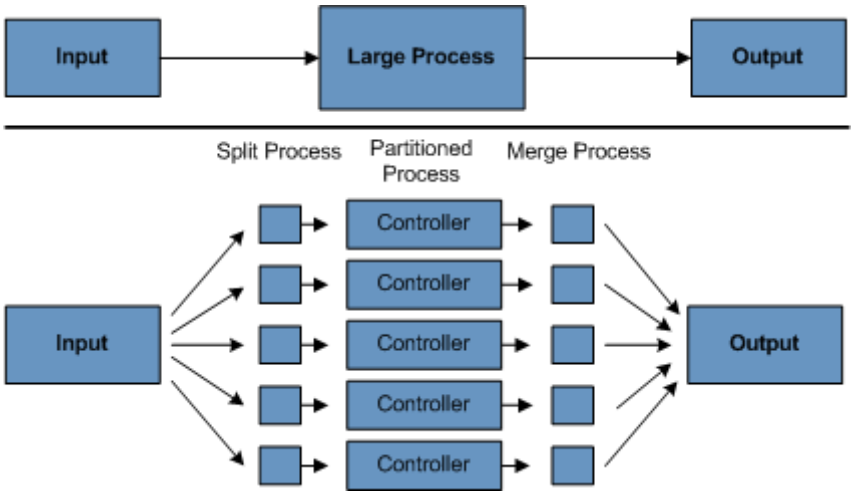


Figure 1.2: Partitioned Process

The architecture should be flexible enough to allow dynamic configuration of the number of partitions. Both automatic and user controlled configuration should be considered. Automatic configuration may be based on parameters such as the input file size and/or the number of input records.

Partitioning Approaches The following lists some of the possible partitioning approaches. Selecting a partitioning approach has to be done on a case-by-case basis.

1. Fixed and Even Break-Up of Record Set

This involves breaking the input record set into an even number of portions (e.g. 10, where each portion will have exactly 1/10th of the entire record set). Each portion is then processed by one instance of the batch/extract application.

In order to use this approach, preprocessing will be required to split the recordset up. The result of this split will be a lower and upper bound placement number which can be used as input to the batch/extract application in order to restrict its processing to its portion alone.

Preprocessing could be a large overhead as it has to calculate and determine the bounds of each portion of the record set.

2. Breakup by a Key Column

This involves breaking up the input record set by a key column such as a location code, and assigning data from each key to a batch instance. In order to achieve this, column values can either be

3. Assigned to a batch instance via a partitioning table (see below for details).

4. Assigned to a batch instance by a portion of the value (e.g. values 0000-0999, 1000 - 1999, etc.)

Under option 1, addition of new values will mean a manual reconfiguration of the batch/extract to ensure that the new value is added to a particular instance.

Under option 2, this will ensure that all values are covered via an instance of the batch job. However, the number of values processed by one instance is dependent on the distribution of column values (i.e. there may be a large number of locations in the 0000-0999 range, and few in the 1000-1999 range). Under this option, the data range should be designed with partitioning in mind.

Under both options, the optimal even distribution of records to batch instances cannot be realized. There is no dynamic configuration of the number of batch instances used.

5. Breakup by Views

This approach is basically breakup by a key column, but on the database level. It involves breaking up the recordset into views. These views will be used by each instance of the batch application during its processing. The breakup will be done by grouping the data.

With this option, each instance of a batch application will have to be configured to hit a particular view (instead of the master table). Also, with the addition of new data values, this new group of data will have to be included into a view. There is no dynamic configuration capability, as a change in the number of instances will result in a change to the views.

6. Addition of a Processing Indicator

This involves the addition of a new column to the input table, which acts as an indicator. As a preprocessing step, all indicators would be marked to non-processed. During the record fetch stage of the batch application, records are read on the condition that that record is marked non-processed, and once they are read (with lock), they are marked processing. When that record is completed, the indicator is updated to either complete or error. Many instances of a batch application can be started without a change, as the additional column ensures that a record is only processed once.

With this option, I/O on the table increases dynamically. In the case of an updating batch application, this impact is reduced, as a write will have to occur anyway.

7. Extract Table to a Flat File

This involves the extraction of the table into a file. This file can then be split into multiple segments and used as input to the batch instances.

With this option, the additional overhead of extracting the table into a file, and splitting it, may cancel out the effect of multi-partitioning. Dynamic configuration can be achieved via changing the file splitting script.

8. Use of a Hashing Column

This scheme involves the addition of a hash column (key/index) to the database tables used to retrieve the driver record. This hash column will have an indicator to determine which instance of the batch application will process this particular row. For example, if there are three batch instances to be started, then an indicator of 'A' will mark that row for processing by instance 1, an indicator of 'B' will mark that row for processing by instance 2, etc.

The procedure used to retrieve the records would then have an additional WHERE clause to select all rows marked by a particular indicator. The inserts in this table would involve the addition of the marker field, which would be defaulted to one of the instances (e.g. 'A').

A simple batch application would be used to update the indicators such as to redistribute the load between the different instances. When a sufficiently large number of new rows have been added, this batch can be run (anytime, except in the batch window) to redistribute the new rows to other instances.

Additional instances of the batch application only require the running of the batch application as above to redistribute the indicators to cater for a new number of instances.

Database and Application design Principles

An architecture that supports multi-partitioned applications which run against partitioned database tables using the key column approach, should include a central partition repository for storing partition parameters. This provides flexibility and ensures maintainability. The repository will generally consist of a single table known as the partition table.

Information stored in the partition table will be static and in general should be maintained by the DBA. The table should consist of one row of information for each partition of a multi-partitioned application. The table should have columns for: Program ID Code, Partition Number (Logical ID of the partition), Low Value of the db key column for this partition, High Value of the db key column for this partition.

On program start-up the program id and partition number should be passed to the application from the architecture (Control Processing Tasklet). These variables are used to read the partition table, to determine what range of data the application is to process (if a key column approach is used). In addition the partition number must be used throughout the processing to:

- Add to the output files/database updates in order for the merge process to work properly
- Report normal processing to the batch log and any errors that occur during execution to the architecture error handler

Minimizing Deadlocks

When applications run in parallel or partitioned, contention in database resources and deadlocks may occur. It is critical that the database design team eliminates potential contention situations as far as possible as part of the database design.

Also ensure that the database index tables are designed with deadlock prevention and performance in mind.

Deadlocks or hot spots often occur in administration or architecture tables such as log tables, control tables, and lock tables. The implications of these should be taken into account as well. A realistic stress test is crucial for identifying the possible bottlenecks in the architecture.

To minimize the impact of conflicts on data, the architecture should provide services such as wait-and-retry intervals when attaching to a database or when encountering a deadlock. This means a built-in mechanism to react to certain database return codes and instead of issuing an immediate error handling, waiting a predetermined amount of time and retrying the database operation.

Parameter Passing and Validation

The partition architecture should be relatively transparent to application developers. The architecture should perform all tasks associated with running the application in a partitioned mode including:

- Retrieve partition parameters before application start-up
- Validate partition parameters before application start-up
- Pass parameters to application at start-up

The validation should include checks to ensure that:

- the application has sufficient partitions to cover the whole data range
- there are no gaps between partitions

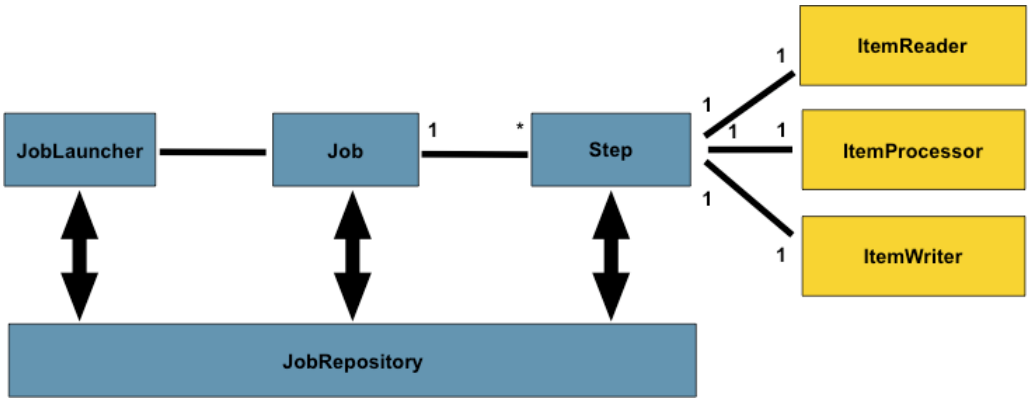
If the database is partitioned, some additional validation may be necessary to ensure that a single partition does not span database partitions.

The Domain Language of Batch

To any experienced batch architect, the overall concepts of batch processing used in Spring Batch should be familiar and comfortable. There are "Jobs" and "Steps" and developer supplied processing units called ItemReaders and ItemWriters. However, because of the Spring patterns, operations, templates, callbacks, and idioms, there are opportunities for the following:

- significant improvement in adherence to a clear separation of concerns
- clearly delineated architectural layers and services provided as interfaces
- simple and default implementations that allow for quick adoption and ease of use out-of-the-box
- significantly enhanced extensibility

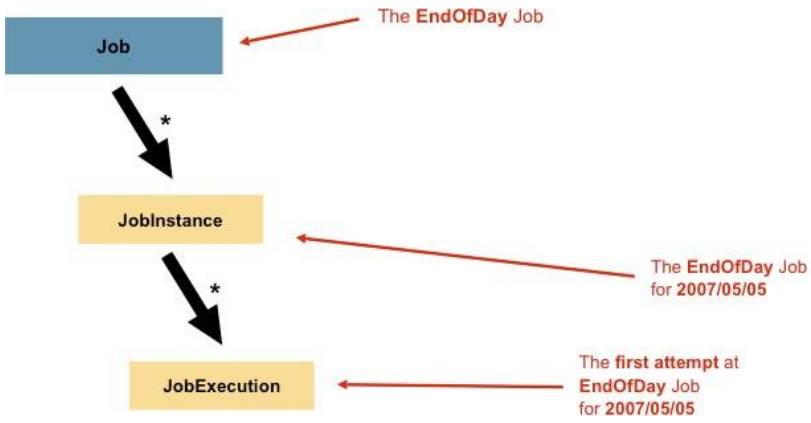
The diagram below is simplified version of the batch reference architecture that has been used for decades. It provides an overview of the components that make up the domain language of batch processing. This architecture framework is a blueprint that has been proven through decades of implementations on the last several generations of platforms (COBOL/Mainframe, C++/Unix, and now Java/anywhere). JCL and COBOL developers are likely to be as comfortable with the concepts as C++, C# and Java developers. Spring Batch provides a physical implementation of the layers, components and technical services commonly found in robust, maintainable systems used to address the creation of simple to complex batch applications, with the infrastructure and extensions to address very complex processing needs.



The diagram above highlights the key concepts that make up the domain language of batch. A Job has one to many steps, which has exactly one ItemReader, ItemProcessor, and ItemWriter. A job needs to be launched (JobLauncher), and meta data about the currently running process needs to be stored (JobRepository).

Job

A Job is an entity that encapsulates an entire batch process. As is common with other Spring projects, a Job will be wired together via an XML configuration file or Java based configuration. This configuration may be referred to as the "job configuration". However, Job is just the top of an overall hierarchy:



In Spring Batch, a Job is simply a container for Steps. It combines multiple steps that belong logically together in a flow and allows for configuration of properties global to all steps, such as restartability. The job configuration contains:

- The simple name of the job
- Definition and ordering of Steps
- Whether or not the job is restartable

A default simple implementation of the Job interface is provided by Spring Batch in the form of the SimpleJob class which creates some standard functionality on top of Job, however the batch namespace abstracts away the need to instantiate it directly. Instead, the <job> tag can be used:

```
<job id="footballJob">
    <step id="playerload" next="gameLoad"/>
    <step id="gameLoad" next="playerSummarization"/>
    <step id="playerSummarization"/>
</job>
```

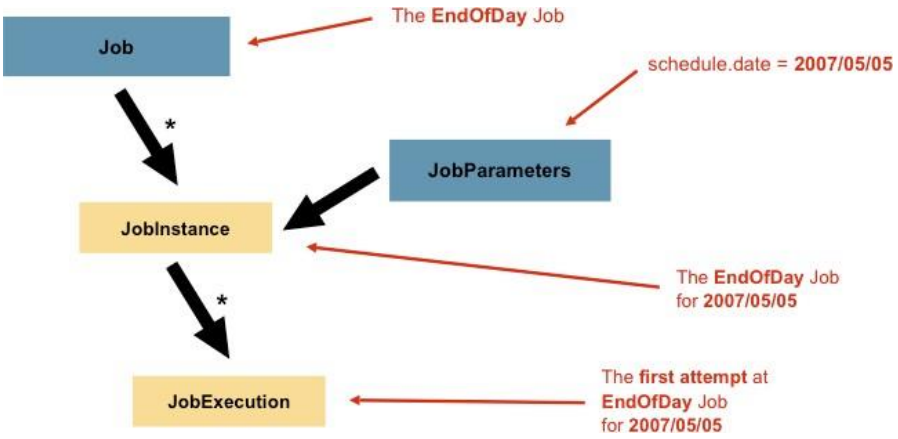
JobInstance

A JobInstance refers to the concept of a logical job run. Let's consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' job from the diagram above. There is one 'EndOfDay' Job, but each individual run of the Job must be tracked separately. In the case of this job, there will be one logical JobInstance per day. For example, there will be a January 1st run, and a January 2nd run. If the January 1st run fails the first time and is run again the next day, it is still the January 1st run. (Usually this corresponds with the data it is processing as well, meaning the January 1st run processes data for January 1st, etc). Therefore, each JobInstance can have multiple executions (JobExecution is discussed in more detail below) and only one JobInstance corresponding to a particular Job and identifying JobParameters can be running at a given time.

The definition of a JobInstance has absolutely no bearing on the data the will be loaded. It is entirely up to the ItemReader implementation used to determine how data will be loaded. For example, in the EndOfDay scenario, there may be a column on the data that indicates the 'effective date' or 'schedule date' to which the data belongs. So, the January 1st run would only load data from the 1st, and the January 2nd run would only use data from the 2nd. Because this determination will likely be a business decision, it is left up to the ItemReader to decide. What using the same JobInstance will determine, however, is whether or not the 'state' (i.e. the ExecutionContext, which is discussed below) from previous executions will be used. Using a new JobInstance will mean 'start from the beginning' and using an existing instance will generally mean 'start from where you left off'.

JobParameters

Having discussed JobInstance and how it differs from Job, the natural question to ask is: "how is one JobInstance distinguished from another?" The answer is: JobParameters. JobParameters is a set of parameters used to start a batch job. They can be used for identification or even as reference data during the run:



In the example above, where there are two instances, one for January 1st, and another for January 2nd, there is really only one Job, one that was started with a job parameter of 01-01-2008 and another that was started with a parameter of 01-02-2008. Thus, the contract can be defined as: JobInstance = Job + identifying JobParameters. This allows a developer to effectively control how a JobInstance is defined, since they control what parameters are passed in.

JobExecution

A JobExecution refers to the technical concept of a single attempt to run a Job. An execution may end in failure or success, but the JobInstance corresponding to a given execution will not be considered complete unless the execution completes successfully. Using the EndOfDay Job described above as an example, consider a JobInstance for 01-01-2008 that failed the first time it was run. If it is run again with the same identifying job parameters as the first run (01-01-2008), a new JobExecution will be created. However, there will still be only one JobInstance.

A Job defines what a job is and how it is to be executed, and JobInstance is a purely organizational object to group executions together, primarily to enable correct restart semantics. A JobExecution, however, is the primary storage mechanism for what actually happened during a run, and as such contains many more properties that must be controlled and persisted:

JobExecution Properties

status	A `BatchStatus` object that indicates the status of the execution. While running, it's BatchStatus.STARTED, if it fails, it's BatchStatus.FAILED, and if it finishes successfully, it's BatchStatus.COMPLETED
startTime	A `java.util.Date` representing the current system time when the execution was started.
endTime	A `java.util.Date` representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The `ExitStatus` indicating the result of the run. It is most important because it contains an exit code that will be returned to the caller. See chapter 5 for more details.
createTime	A `java.util.Date` representing the current system time when the `JobExecution` was first persisted. The job may not have been started yet (and thus has no start time), but it will always have a createTime, which is required by the framework for managing job level `ExecutionContext`s.
lastUpdated	A `java.util.Date` representing the last time a `JobExecution` was persisted.
executionContext	The 'property bag' containing any user data that needs to be persisted between executions.
failureExceptions	The list of exceptions encountered during the execution of a `Job`. These can be useful if more than one exception is encountered during the failure of a `Job`.

These properties are important because they will be persisted and can be used to completely determine the status of an execution. For example, if the EndOfDay job for 01-01 is executed at 9:00 PM, and fails at 9:30, the following entries will be made in the batch meta data tables:

Table 3.2. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob

Table 3.3. BATCH_JOB_EXECUTION_PARAMS

JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	DATE_VAL	IDENTIFYING
1	DATE	schedule.Date	2008-01-01	TRUE

Table 3.4. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00	2008-01-01 21:30	FAILED

Now that the job has failed, let's assume that it took the entire course of the night for the problem to be determined, so that the 'batch window' is now closed. Assuming the window starts at 9:00 PM, the job will be kicked off again for 01-01, starting where it left off and completing successfully at 9:30. Because it's now the next day, the 01-02 job must be run as well, which is kicked off just afterwards at 9:31, and completes in its normal one hour time at 10:30. There is no requirement that one JobInstance be kicked off after another, unless there is potential for the two jobs to attempt to access the same data, causing issues with locking at the database level. It is entirely up to the scheduler to determine when a Job should be run. Since they're separate JobInstances, Spring Batch will make no attempt to stop them from being run concurrently. (Attempting to run the same JobInstance while another is already running will result in a

JobExecutionAlreadyRunningException being thrown). There should now be an extra entry in both the JobInstance and JobParameters tables, and two extra entries in the JobExecution table:

Table 3.5. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob
2	EndOfDayJob

Table 3.6. BATCH_JOB_EXECUTION_PARAMS

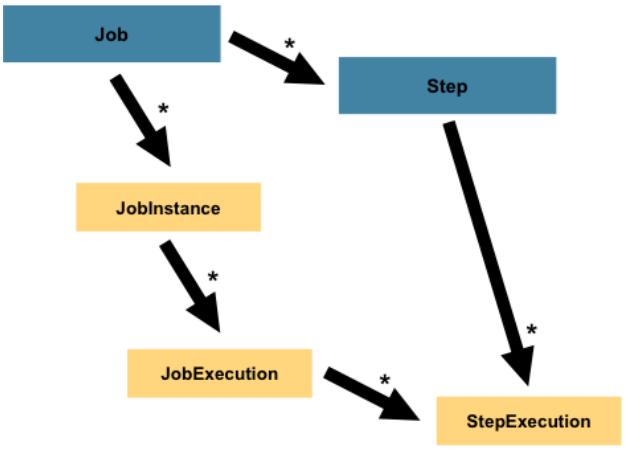
JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	DATE_VAL	IDENTIFYING
1	DATE	schedule.Date	2008-01-01 00:00:00	TRUE
2	DATE	schedule.Date	2008-01-01 00:00:00	TRUE
3	DATE	schedule.Date	2008-01-02 00:00:00	TRUE

Table 3.7. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00	2008-01-01 21:30	FAILED
2	1	2008-01-02 21:00	2008-01-02 21:30	COMPLETED
3	2	2008-01-02 21:31	2008-01-02 22:29	COMPLETED

Step

A Step is a domain object that encapsulates an independent, sequential phase of a batch job. Therefore, every Job is composed entirely of one or more steps. A Step contains all of the information necessary to define and control the actual batch processing. This is a necessarily vague description because the contents of any given Step are at the discretion of the developer writing a Job. A Step can be as simple or complex as the developer desires. A simple Step might load data from a file into the database, requiring little or no code. (depending upon the implementations used) A more complex Step may have complicated business rules that are applied as part of the processing. As with Job, a Step has an individual StepExecution that corresponds with a unique JobExecution:



StepExecution

A StepExecution represents a single attempt to execute a Step. A new StepExecution will be created each time a Step is run, similar to JobExecution. However, if a step fails to execute because the step before it fails, there will be no execution persisted for it. A StepExecution will only be created when its Step is actually started.

Step executions are represented by objects of the StepExecution class. Each execution contains a reference to its corresponding step and JobExecution, and transaction related data such as commit and rollback count and start and end times. Additionally, each step execution will contain an ExecutionContext, which contains any data a developer needs persisted across batch runs, such as statistics or state information needed to restart. The following is a listing of the properties for StepExecution:

StepExecution Properties

status	A `BatchStatus` object that indicates the status of the execution. While it's running, the status is BatchStatus.STARTED, if it fails, the status is BatchStatus.FAILED, and if it finishes successfully, the status is BatchStatus.COMPLETED
startTime	A `java.util.Date` representing the current system time when the execution was started.
endTime	A `java.util.Date` representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The `ExitStatus` indicating the result of the execution. It is most important because it contains an exit code that will be returned to the caller. See chapter 5 for more details.
executionContext	The 'property bag' containing any user data that needs to be persisted between executions.
readCount	The number of items that have been successfully read
writeCount	The number of items that have been successfully written
commitCount	The number transactions that have been committed for this execution
rollbackCount	The number of times the business transaction controlled by the `Step` has been rolled back.
readSkipCount	The number of times `read` has failed, resulting in a skipped item.
processSkipCount	The number of times `process` has failed, resulting in a skipped item.
filterCount	The number of items that have been 'filtered' by the `ItemProcessor`.
writeSkipCount	The number of times `write` has failed, resulting in a skipped item.

ExecutionContext

An ExecutionContext represents a collection of key/value pairs that are persisted and controlled by the framework in order to allow developers a place to store persistent state that is scoped to a StepExecution or JobExecution. For those familiar with Quartz, it is very similar to JobDataMap. The best usage example is to facilitate restart. Using flat file input as an example, while processing individual lines, the framework periodically persists the ExecutionContext at commit points. This allows the ItemReader to store its state in case a fatal error occurs during the run, or even if the power goes out. All that is needed is to put the current number of lines read into the context, and the framework willdo the rest:

```
executionContext.putLong(getKey(LINES_READ_COUNT), reader.getPosition());
```

JobRepository

JobRepository is the persistence mechanism for all of the Stereotypes mentioned above. It provides CRUD operations for JobLauncher, Job, and Step implementations. When a Job is first launched, a JobExecution is obtained from the repository, and during the course of execution StepExecution and JobExecution implementations are persisted by passing them to the repository:

```
<job-repository id="jobRepository"/>
```

JobLauncher

JobLauncher represents a simple interface for launching a Job with a given set of JobParameters:

```
public interface JobLauncher {

   public JobExecution run(Job job, JobParameters jobParameters)

            throws JobExecutionAlreadyRunningException, JobRestartException;

}
```

It is expected that implementations will obtain a valid JobExecution from the JobRepository and execute the Job.

Item Reader

ItemReader is an abstraction that represents the retrieval of input for a Step, one item at a time. When the ItemReader has exhausted the items it can provide, it will indicate this by returning null. More details about the ItemReader interface and its various implementations can be found in Chapter 6, ItemReaders and ItemWriters.

Item Writer

ItemWriter is an abstraction that represents the output of a Step, one batch or chunk of items at a time. Generally, an item writer has no knowledge of the input it will receive next, only the item that was passed in its current invocation. More details about the ItemWriter interface and its various implementations can be found in Chapter 6, ItemReaders and ItemWriters.

Item Processor

ItemProcessor is an abstraction that represents the business processing of an item. While the ItemReader reads one item, and the ItemWriter writes them, the ItemProcessor provides access to transform or apply other business processing. If, while processing the item, it is determined that the item is not valid, returning null indicates that the item should not be written out. More details about the ItemProcessor interface can be found in Chapter 6, ItemReaders and ItemWriters.

Batch Namespace

Many of the domain concepts listed above need to be configured in a Spring ApplicationContext. While there are implementations of the interfaces above that can be used in a standard bean definition, a namespace has been provided for ease of configuration:

```
<beans:beans xmlns="http://www.springframework.org/schema/batch"
     xmlns:beans="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
           http://www.springframework.org/schema/batch
           http://www.springframework.org/schema/batch/spring-batch-2.2.xsd">

    <job id="ioSampleJob">
        <step id="step1">
            <tasklet>
                <chunk reader="itemReader" writer="itemWriter" commit-interval="2"/>
            </tasklet>
        </step>
    </job>

</beans:beans>
```

Spring Batch Application

We will create a simple Spring Batch application which uses a CSV Reader and an XML Writer.

Reader – The reader we are using in the application is FlatFileItemReader to read data from the CSV files

Following is the input CSV file we are using in this application. This document holds data records which specify details like tutorial id, tutorial author, tutorial title, submission date, tutorial icon and tutorial description.

```
1001, "Ashok", "Learn Java", 06/05/2007
1002, "Smith", "Learn MySQL", 19/04/2007
1003, "Dean", "Learn JavaFX", 06/07/2017
```

Writer – The Writer we are using in the application is StaxEventItemWriter to write the data to XML file.

Processor – The Processor we are using in the application is a custom processor which just prints the records read from the CSV file.

jobConfig.xml

Following is the configuration file of our sample Spring Batch application. In this file, we will define the Job and the steps. In addition to these, we also define the beans for ItemReader, ItemProcessor, and ItemWriter. (Here, we associate them with respective classes and pass the values for the required properties to configure them.)

----------Start of jobConfig.xml----------

<beans xmlns = " http://www.springframework.org/schema/beans"

xmlns:batch = "http://www.springframework.org/schema/batch"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/batch

http://www.springframework.org/schema/batch/spring-batch-2.2.xsd

```
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

  <import resource = "../jobs/context.xml" />
   <bean id = "report" class = "Report" scope = "prototype" />
  <bean id = "itemProcessor" class = "CustomItemProcessor" />
    <batch:job id = "helloWorldJob">
      <batch:step id = "step1">
        <batch:tasklet>
        <batch:chunk reader = "cvsFileItemReader" writer = "xmlItemWriter"
          processor = "itemProcessor" commit-interval = "10">
        </batch:chunk>
      </batch:tasklet>
    </batch:step>
  </batch:job>
   <bean id = "cvsFileItemReader" class = "org.springframework.batch.item.file.FlatFileItemReader">
    <property name = "resource" value = "classpath:resources/report.csv" />
    <property name = "lineMapper">
      <bean  class = "org.springframework.batch.item.file.mapping.DefaultLineMapper">
        <property name = "lineTokenizer">
          <bean   class = "org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
            <property name = "names" value = "tutorial_id, tutorial_author, Tutorial_title, submission_date" />
          </bean>
        </property>
        <property name = "fieldSetMapper">
          <bean class = "ReportFieldSetMapper" />
        </property>
      </bean>
    </property>
```

```
  </bean>


  <bean id = "xmlItemWriter"  class = "org.springframework.batch.item.xml.StaxEventItemWriter">
    <property name = "resource" value = "file:xml/outputs/tutorials.xml" />
    <property name = "marshaller" ref = "reportMarshaller" />
    <property name = "rootTagName" value = "tutorials" />
  </bean>
  <bean id = "reportMarshaller"  class = "org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name = "classesToBeBound">
      <list>
        <value>Tutorial</value>
      </list>
    </property>
  </bean>
</beans>
```

----------End of jobConfig.xml----------

Context.xml

Following is the context.xml of our Spring Batch application. In this file, we will define the beans like job repository, job launcher, and transaction manager.

Start of Context.xml

```
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:jdbc = "http://www.springframework.org/schema/jdbc"
  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-3.2.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">


  <!-- stored job-meta in database -->
```

```
  <bean id = "jobRepository"
    class = "org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
    <property name = "dataSource" ref = "dataSource" />
    <property name = "transactionManager" ref = "transactionManager" />
    <property name = "databaseType" value = "mysql" />
  </bean>
   <bean id = "transactionManager"
    class = "org.springframework.batch.support.transaction.ResourcelessTransactionManager" />
  <bean id = "jobLauncher"
    class = "org.springframework.batch.core.launch.support.SimpleJobLauncher">
    <property name = "jobRepository" ref = "jobRepository" />
  </bean>
   <bean id = "dataSource" class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name = "driverClassName" value = "com.mysql.jdbc.Driver" />
    <property name = "url" value = "jdbc:mysql://localhost:3306/details" />
    <property name = "username" value = "myuser" />
    <property name = "password" value = "password" />
  </bean>
    <!-- create job-meta tables automatically -->
  <jdbc:initialize-database data-source = "dataSource">
    <jdbc:script location = "org/springframework/batch/core/schema-drop-mysql.sql" />
    <jdbc:script location = "org/springframework/batch/core/schema-mysql.sql" />
  </jdbc:initialize-database>
</beans>
```

----------End of Context.xml----------

CustomItemProcessor.java

Following is the Processor class. In this class, we write the code of processing in the application. Here, we are printing the contents of each record.

```
import org.springframework.batch.item.ItemProcessor;

public class CustomItemProcessor implements ItemProcessor<Tutorial, Tutorial> {

   @Override
   public Tutorial process(Tutorial item) throws Exception {
      System.out.println("Processing..." + item);
      return item;
   }
}
```

TutorialFieldSetMapper.java

Following is the TutorialFieldSetMapper class which sets the data to the Tutorial class.

```
import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;
import org.springframework.validation.BindException;

public class TutorialFieldSetMapper implements FieldSetMapper<Tutorial> {

   @Override
   public Tutorial mapFieldSet(FieldSet fieldSet) throws BindException {

      //Instantiating the report object
      Tutorial tutorial = new Tutorial();

      //Setting the fields
      tutorial.setTutorial_id(fieldSet.readInt(0));
      tutorial.setTutorial_author(fieldSet.readString(1));
      tutorial.setTutorial_title(fieldSet.readString(2));
      tutorial.setSubmission_date(fieldSet.readString(3));

      return tutorial;
   }
}
```

Tutorial.java class

Following is the Tutorial class. It is a simple Java class with setter and getter methods. In this class, we are using annotations to associate the methods of this class with the tags of the XML file.

```
@XmlRootElement(name = "tutorial")
public class Tutorial {
   private int tutorial_id;
   private String tutorial_author;
   private String tutorial_title;
   private String submission_date;

   @XmlAttribute(name = "tutorial_id")
   public int getTutorial_id() {
      return tutorial_id;
   }

   @XmlElement(name = "tutorial_author")
   public String getTutorial_author() {
      return tutorial_author;
   }

   @XmlElement(name = "tutorial_title")
   public String getTutorial_title() {
      return tutorial_title;
   }

   @XmlElement(name = "submission_date")
   public String getSubmission_date() {
      return submission_date;
   }
    //setter methods ( )
   //toString()
}
```

App.java

Following is the code which launches the batch process. In this class, we will launch the batch application by running the JobLauncher.

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {
   public static void main(String[] args) throws Exception {

      String[] springConfig  =  { "jobs/job_hello_world.xml" };

      // Creating the application context object
      ApplicationContext context = new ClassPathXmlApplicationContext(springConfig);

      // Creating the job launcher
      JobLauncher jobLauncher = (JobLauncher) context.getBean("jobLauncher");

      // Creating the job
      Job job = (Job) context.getBean("helloWorldJob");

      // Executing the JOB
      JobExecution execution = jobLauncher.run(job, new JobParameters());
      System.out.println("Exit Status : " + execution.getStatus());
   }
}
```

On executing this application, it will produce the following output.

```
May 08, 2017 10:10:12 AM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh

INFO: Refreshing  org.springframework.context.support.ClassPathXmlApplicationContext@3d646c37: startup date
[Mon May 08 10:10:12 IST 2018]; root of context hierarchy
May 08, 2018 10:10:12 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
May 08, 2018 10:10:15 AM org.springframework.jdbc.datasource.init.ScriptUtils executeSqlScript
INFO: Executing step: [step1]

Processing...  [Tutorial id=1001, Tutorial Author=Ashok, Tutorial Title=Learn Java, Submission Date=06/05/2007]
Processing...  [Tutorial id=1002, Tutorial Author=Smith, Tutorial Title=Learn MySQL, Submission Date=19/04/2007]
Processing...  [Tutorial id=1003, Tutorial Author=Dean, Tutorial Title=Learn JavaFX, Submission Date=06/07/2017]

May 08, 2018 10:10:21 AM org.springframework.batch.core.launch.support.SimpleJobLauncher run
INFO: Job: [FlowJob: [name=helloWorldJob]] completed with the following parameters: [{}] and the following
status: [COMPLETED]

Exit Status : COMPLETED
```

This will generate an XML file with the following contents.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<tutorials>
   <tutorial tutorial_id = "1001">
      <submission_date>06/05/2007</submission_date>
      <tutorial_author>Ashok</tutorial_author>
      <tutorial_title>Learn Java</tutorial_title>
   </tutorial>

   <tutorial tutorial_id = "1002">
      <submission_date>19/04/2007</submission_date>
      <tutorial_author>Smith</tutorial_author>
      <tutorial_title>Learn MySQL</tutorial_title>
   </tutorial>

   <tutorial tutorial_id = "1003">
      <submission_date>06/07/2017</submission_date>
      <tutorial_author>Dean</tutorial_author>
      <tutorial_title>Learn JavaFX</tutorial_title>
   </tutorial>
</tutorials>
```

==== Happy Coding ====