https://www.youtube.com/watch?v=mLMvGeb-e-0

**SPRING**

***BeanPostProcessor / BeanFactoryPostProcessor***

Many processes in the [IoC container](http://www.dineshonjava.com/2012/06/spring-ioc-container.html) were made to be extensible. A specific extensible process can be referred to as an extension point. One extension point, the **BeanPostProcessor** interface, allows the modification of a bean instance before and after the properties are set. Another extension point is the **BeanFactoryPostProcessor** interface which allows direct modification of bean definitions before a bean is instantiated.

An [**ApplicationContext**](http://www.dineshonjava.com/2012/06/application-context-in-spring.html)will automatically register and process a bean that implements either of these interfaces (***BeanPostProcessor***, ***BeanFactoryPostProcessor***), but a [**BeanFactory**](http://www.dineshonjava.com/2012/06/what-is-bean-factory-in-spring.html) would have to have a ***BeanPostProcessor***or ***BeanFactoryPostProcessor***registered with it programatically as given below.

1. .....
2. // create BeanFactory
3. ConfigurableBeanFactory  factory = **new** XmlBeanFactory(**new** FileSystemResource("spring.xml"));
4. // now register some beans
5. // now register any needed BeanPostProcessors
6. DisplayNameBeanPostProcessor postProcessor = **new** DisplayNameBeanPostProcessor();
7. factory.addBeanPostProcessor(postProcessor);
8. .....
9. // now start using the factory

Since this manual registration step is not convenient, and [ApplictionContexts](http://www.dineshonjava.com/2012/06/application-context-in-spring.html) are functionally supersets of [BeanFactories](http://www.dineshonjava.com/2012/06/what-is-bean-factory-in-spring.html), it is generally recommended that [ApplicationContext](http://www.dineshonjava.com/2012/06/application-context-in-spring.html) variants are used when bean post-processors are needed.

**BeanPostProcessor:** A **BeanPostProcessor** gives you a chance to process an instance of a bean created by the [IoC container](http://www.dineshonjava.com/2012/06/spring-ioc-container.html)after it's instantiation and then again after the initialization life cycle event has occurred on the instance. You could use this to process fields that were set, perform validation on a bean, or even look up values from a remote resource to set on the bean as defaults.

Spring's different AOP proxies for caching, transactions, etc. are all applied by BeanPostProcessor .

BeanPostProcessor interface has two method...

1. **postProcessAfterInitialization** (Object bean, String beanName) execute after initialization of each beans in the [Spring IoC Container](http://www.dineshonjava.com/2012/06/spring-ioc-container.html).

2. **postProcessBeforeInitialization**(Object bean, String beanName) execute before initialization of each beans in the [Spring IoC Container](http://www.dineshonjava.com/2012/06/spring-ioc-container.html).

# Implementing RowMapper in Spring with Example

We implement a custom RowMapper class to map our domain objects. We then use this class to write fetchmethods that return custom model objects.  
  
An interface used by [JdbcTemplate](http://www.dineshonjava.com/2012/12/using-jdbctemplate-in-spring-chapter-33.html) 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](http://www.dineshonjava.com/2012/12/using-jdbctemplate-in-spring-chapter-33.html#.UMMg0IaeWk9).

Typically used either for JdbcTemplate's query methods or for out parameters of stored procedures. **RowMapper**objects are typically stateless and thus reusable; they are an ideal choice for implementing row-mapping logic in a single place.   
  
Alternatively, consider subclassing **MappingSqlQuery**from the **jdbc.object package**: Instead of working with separate JdbcTemplate and **RowMapper**objects, you can build executable query objects (containing row-mapping logic) in that style.  
  
Say for example, when we are selecting records from an employee table, we will iterate over the result set to get the individual values which won’t be ideal for situations, especially in Java where we want to map records from a database to individual Javaobjects. Also the question of re-usability comes into the picture as the above code doesn’t represent for getting itself re-used. Spring Row Mapper interfaces come into the rescue for such situations.  
  
 **Querying for Single Row:**  
Here’s two ways to query or extract a single row record from database, and convert it into a model class(Employee).  
**1. Custom RowMapper:**  
In general, It’s always recommended to implement the RowMapper interface to create a custom **RowMapper**to suit your needs..

1. **package** com.dineshonjava.sdnext.jdbc.utils;
2. **import** java.sql.ResultSet;
3. **import** java.sql.SQLException;
4. **import** org.springframework.jdbc.core.RowMapper;
5. **import** com.dineshonjava.sdnext.domain.Employee;
6. /\*\*
7. \* @author Dinesh Rajput
8. \*
9. \*/
10. **public** **class** EmployeeMapper **implements** RowMapper {
11. **public** Employee mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {
12. Employee employee = **new** Employee();
13. employee.setEmpid(rs.getInt("empid"));
14. employee.setName(rs.getString("name"));
15. employee.setAge(rs.getInt("age"));
16. employee.setSalary(rs.getLong("salary"));
17. **return** employee;
18. }
19. }

Pass it to ***queryForObject()***method, the returned result will call your custom ***mapRow()*** method to match the value into the property.

1. **public** Employee getEmployee(Integer empid) {
2. String SQL = "SELECT \* FROM Employee WHERE empid = ?";
3. Employee employee = (Employee) jdbcTemplateObject.queryForObject(SQL, **new** Object[]{empid}, **new** EmployeeMapper());
4. **return** employee;
5. }

**2. BeanPropertyRowMapper:**  
In Spring 2.5, comes with a handy **RowMapper**implementation called ‘**BeanPropertyRowMapper**’, which can maps a row’s column value to a property by matching their names. Just make sure both the property and column has the same name, e.g property ‘empid’ will match to column name ‘EMPID’ or with underscores ‘EMP\_ID’.

1. **public** Employee getEmployee(Integer empid) {
2. String SQL = "SELECT \* FROM Employee WHERE empid = ?";
3. Employee employee = (Employee) jdbcTemplateObject.queryForObject(SQL, **new** Object[]{empid}, **new** BeanPropertyRowMapper(Employee.**class**));
4. **return** employee;
5. }

**Querying for Multiple Rows:**  
Now, query or extract multiple rows from database, and convert it into a List.  
**1. Map it manually:**  
In mutiple return rows, RowMapper is not supported in queryForList() method, you need to map it manually.

1. **public** List<Employee> findAll(){
2. String sql = "SELECT \* FROM Employee";
4. List employees= **new** ArrayList();
6. List rows = getJdbcTemplate().queryForList(sql);
7. **for** (Map row : rows) {
8. Employee employee = **new** Employee();
9. employee.setEmpid((Integer)(row.**get**("EMPID")));
10. employee.setName((String)row.**get**("NAME"));
11. employee.setAge((Integer)row.**get**("AGE"));
12. employee.setSalary((Long)row.**get**("SALARY"));
13. employees.add(employee);
14. }
15. **return** employees;
16. }

**2. BeanPropertyRowMapper:**  
The simplest solution is using the BeanPropertyRowMapper class.

1. **public** List findAll(){
2. String sql = "SELECT \* FROM Employee";
3. List employees= getJdbcTemplate().query(sql,
4. **new** BeanPropertyRowMapper(Employee.**class**));
5. **return** employees;
6. }

# *DAO Support Classes in Spring*

**• JdbcDaoSupport** – superclass for JDBC data access objects. Requires a DataSource to be provided; in turn, this class provides a [**JdbcTemplate**](http://www.dineshonjava.com/2012/12/using-jdbctemplate-in-spring-chapter-33.html) instance initialized from the supplied DataSource to subclasses.  
**• HibernateDaoSupport** – superclass for Hibernate data access objects. Requires a SessionFactory to be provided; in turn, this class provides a HibernateTemplate instance initialized from the supplied SessionFactory to subclasses. Can alternatively be initialized directly via a HibernateTemplate, to reuse the latters settings like SessionFactory, flush mode, exception translator, and so forth.  
**• JdoDaoSupport** – super class for JDO data access objects. Requires a PersistenceManagerFactoryto be provided; in turn, this class provides a JdoTemplate instance initialized from the supplied PersistenceManagerFactory to subclasses.  
**• JpaDaoSupport** – super class for JPA data access objects. Requires a EntityManagerFactory to be provided; in turn, this class provides a JpaTemplate  
  
**In Spring JDBC Framework**there are many DAO support classes which help to reduce the configuration of JdbcTemplate, SimpleJdbcTemplate and NamedParamJdbcTemplate with dataSource object.  
**For**[**JdbcTemplate**](http://www.dineshonjava.com/2012/12/using-jdbctemplate-in-spring-chapter-33.html)**:**  
*org.springframework.jdbc.core.support.JdbcDaoSupport*  
**For**[**SimpleJdbcTemplate**](http://www.dineshonjava.com/2012/12/spring-simplejdbctemplate-example.html)**:**  
*org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport*  
**For**[**NamedParamJdbcTemplate**](http://www.dineshonjava.com/2012/12/using-namedparameterjdbctemplate-in.html)**:**  
*org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport*

***Difference between ref and idref:***In ***ref***can have the value of ***id of bean*** as well as ***name of the bean***.

But the ***idref***can have only and only ***id of the bean***.

**Using *ref*:**

1. **<bean** class="com.sdnext.innerBean.tutorial.Triangle" id="triangle"**>**
2. **<property** name="pointA" ref="zeroPoint"**></property>**
3. **</bean>**
4. **<bean** class="com.dineshonjava.sdnext.innerBean.tutorial.Point" name="zeroPoint"**>**
5. **</bean>**

**Using *idref*:**

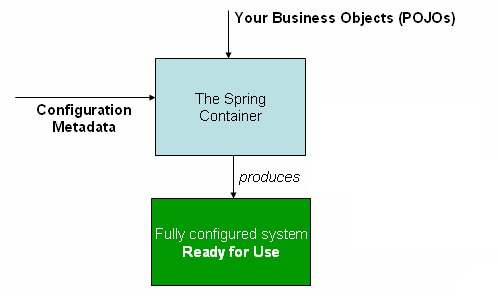
1. **<bean** class="com.sdnext.innerBean.tutorial.Triangle" id="triangle"**>**
2. **<property** name="pointA"**>**
3. **<idref** bean="zeropoint"**>**
4. **</idref>**
5. **</property>**
6. **</bean>**
7. **<bean** class="com.dineshonjava.sdnext.innerBean.tutorial.Point" id="zeroPoint"**>**
8. **</bean>**

**Constructor v/s Setter Injection**

* Setter injection gets preference over constructor injection when both are specified.
* Constructor injection cannot partially initialize values.
* Circular dependency can be achieved by setter injection.
* Security is lesser in setter injection as it can be overridden.
* Constructor injection fully ensures dependency injection but setter injection does not
* Setter injection is more readable.

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.

**Figure 7.1. The Spring IoC container**



The [Core Container](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-introduction) consists of the spring-core, spring-beans, spring-context, spring-context-support, and spring-expression (Spring Expression Language) modules.

The spring-core and spring-beans modules [provide the fundamental parts of the framework](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-introduction), including the IoC and Dependency Injection features. TheBeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The [Context](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#context-introduction) (spring-context) module builds on the solid base provided by the [Core and Beans](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-introduction) modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event propagation, resource loading, and the transparent creation of contexts by, for example, a Servlet container. The Context module also supports Java EE features such as EJB, JMX, and basic remoting.

The ApplicationContext interface is the focal point of the Context module. spring-context-support provides support for integrating common third-party libraries into a Spring application context for caching (EhCache, Guava, JCache), mailing (JavaMail), scheduling (CommonJ, Quartz) and template engines (FreeMarker, JasperReports, Velocity).

The spring-expression module provides a powerful [Expression Language](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#expressions) for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the content of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring’s IoC container. It also supports list projection and selection as well as common list aggregations.

### 2.2.2 AOP and Instrumentation

The spring-aop module provides an [AOP](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#aop-introduction) Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of .NET attributes.

The separate spring-aspects module provides integration with AspectJ.

The spring-instrument module provides class instrumentation support and classloader implementations to be used in certain application servers. The spring-instrument-tomcat module contains Spring’s instrumentation agent for Tomcat.

### 2.2.3 Messaging

Spring Framework 4 includes a spring-messaging module with key abstractions from the Spring Integration project such as Message, MessageChannel, MessageHandler, and others to serve as a foundation for messaging-based applications. The module also includes a set of annotations for mapping messages to methods, similar to the Spring MVC annotation based programming model.

### 2.2.4 Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS, and Transaction modules.

The spring-jdbc module provides a [JDBC](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jdbc-introduction)-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The spring-tx module supports [programmatic and declarative transaction](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#transaction) management for classes that implement special interfaces and for all your POJOs (Plain Old Java Objects).

The spring-orm module provides integration layers for popular [object-relational mapping](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-introduction) APIs, including [JPA](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-jpa), [JDO](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-jdo), and [Hibernate](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#orm-hibernate). Using the spring-orm module you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

The spring-oxm module provides an abstraction layer that supports [Object/XML mapping](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#oxm) implementations such as JAXB, Castor, XMLBeans, JiBX and XStream.

The spring-jms module ([Java Messaging Service](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jms)) contains features for producing and consuming messages. Since Spring Framework 4.1, it provides integration with the spring-messaging module.

### 2.2.5 Web

The Web layer consists of the spring-web, spring-webmvc, spring-websocket, and spring-webmvc-portlet modules.

The spring-web module provides basic web-oriented integration features such as multipart file upload functionality and the initialization of the IoC container using Servlet listeners and a web-oriented application context. It also contains an HTTP client and the web-related parts of Spring’s remoting support.

The spring-webmvc module (also known as the Web-Servlet module) contains Spring’s model-view-controller ([MVC](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#mvc-introduction)) and REST Web Services implementation for web applications. Spring’s MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.

The spring-webmvc-portlet module (also known as the Web-Portlet module) provides the MVC implementation to be used in a Portlet environment and mirrors the functionality of the spring-webmvc module.

The spring-test module supports the [unit testing](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#unit-testing) and [integration testing](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#integration-testing) of Spring components with JUnit or TestNG. It provides consistent [loading](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#testcontext-ctx-management) of SpringApplicationContexts and [caching](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#testcontext-ctx-management-caching) of those contexts. It also provides [mock objects](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#mock-objects) that you can use to test your code in isolation.

**Instantiating a container**

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {"services.xml", "daos.xml"});

#### Composing XML-based configuration metadata

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the previous section. Alternatively, use one or more occurrences of the <import/> element to load bean definitions from another file or files. For example:

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

**Circular dependencies**

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws aBeanCurrentlyInCreationException.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

#### Inner beans

A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called inner bean.

<bean id="outer" class="...">

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

**Primefaces Spring Hibernate EclipseLink**

**Hibernate & EclipseLink** are two most popular implementations used for persisting given business model against some sort of persistence store like relational database. As such, this tutorial will provide you a full-fledged example containing all required configuration steps to developer a layered application that uses:

1. Primefaces components to develop a compelling User Interface that aimed to handle user’s interactions and verify user’s inputs.
2. Hibernate/EclipseLink implementations to develop an Object/Relational Mapping beneath JPA umbrella.
3. [Spring framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial) as a kind of glue that get everything attached each together.

**What is Java Server Faces?**

* JSF is a component based framework with great focus on user interfaces. Whereas [Spring framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial) core principle is [**Dependency Injection**](http://www.journaldev.com/2394/java-dependency-injection-design-pattern-example-tutorial)**.** So it makes sense to integrate JSF with Spring framework where JSF will be used for user interfaces and Spring framework will be used for backend server side business logic.

Springs can be integrated with JSF through **DelegatingVariableResolver**.

Add the **DelegatingVariableResolver** in faces-config.xml file as shown below. Here el-resolver is the delegating variable resolver.

**<application>**

**<el-resolver>**

**org.springframework.web.jsf.el.SpringBeanFacesELResolver**

**</el-resolver>**

**</application>**

The el-resolver mentioning SpringBeanFacesELResolver connects the JSF front end values to the Spring backend service layer.

**Key Components of Spring Boot Framework**

Spring Boot Framework has mainly four major Components.

* Spring Boot Starters
* Spring Boot AutoConfigurator
* Spring Boot CLI
* Spring Boot Actuator

### Spring Boot Starter

Spring Boot Starters is one of the major key features or components of Spring Boot Framework. The main responsibility of Spring Boot Starter is to combine a group of common or related dependencies into single dependencies. We will explore this statement in detail with one example.

For instance, we would like to develop a Spring WebApplication with Tomcat WebServer. Then we need to add the following minimal jar dependencies in your Maven’s pom.xml file or Gradle’s build.gradle file

Spring core Jar file(spring-core-xx.jar)

Spring Web Jar file(spring-web-xx.jar)

Spring Web MVC Jar file(spring-webmvc-xx.jar)

Servlet Jar file(servlet-xx.jar)

If we want to add some database stuff, then we need to add database related jars like Spring JDBC jar file, Spring ORM jar files,Spring Transaction Jar file etc.

Spring JDBC Jar file(spring-jdbc-xx.jar)

Spring ORM Jar file(spring-orm-xx.jar)

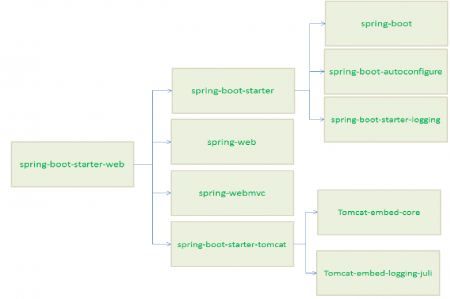
Spring Transaction Jar file(spring-transaction-xx.jar)

We need to define lot of dependencies in our build files. It is very tedious and cumbersome tasks for a Developer. And also it increases our build file size.

What is the solution to avoid this much dependencies definitions in our build files? The solution is Spring Boot Starter component.

Spring Boot Starter component combines all related jars into single jar file so that we can add only jar file dependency to our build files. Instead of adding above 4 jars files to our build file, we need to add one and only one jar file: “spring-boot-starter-web” jar file.

When we add “spring-boot-starter-web” jar file dependency to our build file, then Spring Boot Framework will automatically download all required jars and add to our project classpath.

[](http://cdn.journaldev.com/wp-content/uploads/2015/06/spring-boot-starter-dependencies1.png)

In the same way, “spring-boot-starter-logging” jar file loads all it’s dependency jars like “jcl-over-slf4j, jul-to-slf4j,log4j-over-slf4j, logback-classic” to our project classpath.

#### Major Advantages of Spring Boot Starter

Spring Boot Starter reduces defining many dependencies simplify project build dependencies.

Spring Boot Starter simplifies project build dependencies.

That’s it about Spring Boot Starter component. We will discuss some more details with some Spring Boot examples in coming posts.

### Spring Boot AutoConfigurator

Another important key component of Spring Boot Framework is Spring Boot AutoConfigurator. Most of the Spring IO Platform ([Spring Framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial)) Critics opinion is that “To develop a Spring-based application requires lot of configuration (Either XML Configuration of Annotation Configuration). Then how to solve this problem.

The solution to this problem is Spring Boot AutoConfigurator. The main responsibility of Spring Boot AutoConfigurator is to reduce the Spring Configuration. If we develop Spring applications in Spring Boot,then We dont need to define single XML configuration and almost no or minimal Annotation configuration. Spring Boot AutoConfigurator component will take care of providing those information.

For instance, if we want to declare a Spring MVC application using Spring IO Platform, then we need to define lot of XML Configuration like views, view resolvers etc. But if we use Spring Boot Framework, then we dont need to define those XML Configuration. Spring Boot AutoConfigurator will take of this.

If we use “spring-boot-starter-web” jar file in our project build file, then Spring Boot AutoConfigurator will resolve views, view resolvers etc. automatically.

And also Spring Boot reduces defining of Annotation configuration. If we use @SpringBootApplication annotation at class level, then Spring Boot AutoConfigurator will automatically add all required annotations to Java Class ByteCode.

[SpringBootApplicationAnnotation](http://cdn.journaldev.com/wp-content/uploads/2015/06/SpringBootApplicationAnnotation.png)

If we go through Spring Boot Documentation, we can find the following definition for @SpringBootApplication.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Target(value=TYPE)  @Retention(value=RUNTIME)  @Documented  @Inherited  @Configuration  @EnableAutoConfiguration  @ComponentScan  public @interface SpringBootApplication |

That is, @SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiration.

That’s it about Spring Boot AutoConfigurate component. We will discuss some more details with some Spring Boot examples in coming posts.

**NOTE:-**

In simple words, Spring Boot Starter reduces build’s dependencies and Spring Boot AutoConfigurator reduces the Spring Configuration.

As we discussed that Spring Boot Starter has a dependency on Spring Boot AutoConfigurator, Spring Boot Starter triggers Spring Boot AutoConfigurator automatically.

### Spring Boot CLI

Spring Boot CLI(Command Line Interface) is a Spring Boot software to run and test Spring Boot applications from command prompt. When we run Spring Boot applications using CLI, then it internally uses Spring Boot Starter and Spring Boot AutoConfigurate components to resolve all dependencies and execute the application.

We can run even Spring Web Applications with simple Spring Boot CLI Commands.

Spring Boot CLI has introduced a new “spring” command to execute Groovy Scripts from command prompt.

**spring command example:**

|  |  |
| --- | --- |
| 1 | spring run HelloWorld.groovy |

Here HelloWorld.groovy is a Groovy script FileName. Like Java source file names have \*.java extension, Groovy script files have \*.groovy extension. “spring” command executes HelloWorld.groovy and produces output.

Spring Boot CLI component requires many steps like CLI Installation, CLI Setup, Develop simple Spring Boot application and test it. So we are going to dedicate another post to discuss it in details with some Spring Boot Examples. Please refer my next post on Spring Boot CLI.

### Spring Boot Actuator

Spring Boot Actuator components gives many features, but two major features are

Providing Management EndPoints to Spring Boot Applications.

Spring Boot Applications Metrics.

When we run our Spring Boot Web Application using CLI, Spring Boot Actuator automatically provides hostname as “localhost” and default port number as “8080”. We can access this application using “http://localhost:8080/” end point.

We actually use HTTP Request methods like GET and POST to represent Management EndPoints using Spring Boot Actuator.

We will discuss some more details about Spring Boot Actuator in coming posts.

### Internals of Spring Boot Framework

It’s always recommended to understand how Spring Boot Framework reduces build’s dependencies,Spring Configuration, etc. How Spring Boot works under-the-hood.

If you are familiar with Groovy Programming language, then you know most of the stuff. In Groovy, we don’t need to add some some imports and no need to add some dependencies to Groovy project. When we compile Groovy scripts using Groovy Compiler(groovyc), it will automatically adds all default import statements then compile it.

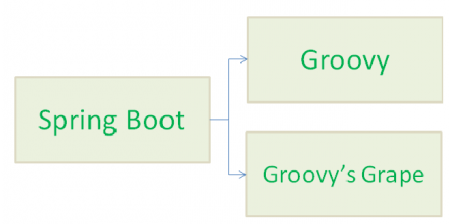
In the same way, Groovy Programming language contains a JAR Dependency Resolver to resolve and add all required jar files to Groovy Project classpath.

Spring Boot Framework internally uses Groovy to add some defaults like Default import statements, Application main() method etc. When we run Groovy Scripts from CLI Command prompt, it uses this main() method to run the Spring Boot Application.

#### Grape

Grape is an Embedded Dependency Resolution engine. Grape is a JAR Dependency Manager embedded into Groovy. Grape lets us quickly add maven repository dependencies to our project classpath to reduce build file definitions.

Spring Boot Framework programming model is mainly inspired by Groovy Programming model. Spring Boot Framework internally depends on these two major components: Groovy and Grape.

[](http://cdn.journaldev.com/wp-content/uploads/2015/06/springboot-internals.png)

You can refer Grape documentation http://docs.groovy-lang.org/latest/html/documentation/grape.html for more details.

That’s it about Spring Components and Internals. We will discuss about these components in details with some Spring Boot examples in coming posts.

**Spring Boot Tutorial**

Spring Boot uses completely new development model to make Java Development very easy by avoiding some tedious development steps and boilerplate code and configuration.

**What is Spring Boot?**

Spring Boot is a Framework from “The Spring Team” to ease the bootstrapping and development of new Spring Applications.  
 It provides defaults for code and annotation configuration to quick start new Spring projects within no time. It follows “Opinionated Defaults Configuration” Approach to avoid lot of boilerplate code and configuration to improve Development, Unit Test and Integration Test Process.

**What is NOT Spring Boot?**

Spring Boot Framework is not implemented from the scratch by The Spring Team, rather than implemented on top of existing Spring Framework (Spring IO Platform).  
It is not used for solving any new problems. It is used to solve same problems like Spring Framework.

**Why Spring Boot?**

* To ease the Java-based applications Development, Unit Test and Integration Test Process.
* To reduce Development, Unit Test and Integration Test time by providing some defaults.
* To increase Productivity.
* Don’t worry about what is “Opinionated Defaults Configuration” Approach at this stage. We will explain this in detail with some examples in coming posts.
* Advantages of Spring Boot:
* It is very easy to develop Spring Based applications with Java or Groovy.
* It reduces lots of development time and increases productivity.
* It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
* It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
* It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort
* It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
* It provides CLI (Command Line Interface) tool to develop and test Spring Boot(Java or Groovy) Applications from command prompt very easily and quickly.
* It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
* It provides lots of plugins to work with embedded and in-memory Databases very easily.
* In Simple Terminology, What Spring Boot means

[What Is Spring Boot, Spring Boot Tutorial](http://cdn.journaldev.com/wp-content/uploads/2015/05/WhatIsSpringBoot1.png)

* That means Spring Boot is nothing but existing Spring Framework + Some Embedded HTTP Servers (Tomcat/Jetty etc.) – XML or Annotations Configurations.  
  Here minus means we don’t need to write any XML Configuration and few Annotations only.
* **Main Goal of Spring Boot:**
* The main goal of Spring Boot Framework is to reduce Development, Unit Test and Integration Test time and to ease the development of Production ready web applications very easily compared to existing Spring Framework, which really takes more time.

**Spring MVC controller and the RESTful web service controller**

Spring’s annotation based MVC framework simplifies the process of creating RESTful web services. The key difference between a traditional Spring MVC controller and the RESTful web service controller is the way the HTTP response body is created. While the traditional MVC controller relies on the View technology, the RESTful web service controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML. For a detailed description of creating RESTful web services using the Spring framework.

Using the @RestController Annotation

Spring 4.0 introduced @RestController, a specialized version of the controller which is a convenience annotation that does nothing more than add the @Controller and @ResponseBodyannotations. By annotating the controller class with @RestController annotation, you no longer need to add @ResponseBody to the entire request mapping methods. The @ResponseBody annotation is active by default**.**

@RestController is composition of @Controller and @ResponseBody, if we are not using the @ResponseBody in Method signature then we need to use the @Restcontroller.

**List of View Resolvers in Spring MVC**

In Spring MVC, view resolvers enable you to render models in a browser without tying you to a specific view technology like JSP, Velocity, XML…etc.

There are two interfaces that are important to the way Spring handles views are **ViewResolver** and **View**. The **ViewResolver** provides 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.

Below are the important viewresolvers provided by spring framework:

1. **AbstractCachingViewResolver** : Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.
2. **XmlViewResolver** : Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.
3. **ResourceBundleViewResolver** : Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is **views.properties**.
4. **UrlBasedViewResolver** : Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
5. **InternalResourceViewResolver** :  Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).
6. **VelocityViewResolver**/**FreeMarkerViewResolver**: Convenient subclass of UrlBasedViewResolver that supports **VelocityView** (in effect, Velocity templates) or **FreeMarkerView**, respectively, and custom subclasses of them.
7. **ContentNegotiatingViewResolver** : Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header.

**Spring Hibernate Integration**

**Spring** is one of the most used Java EE Framework and **Hibernate** is the most popular ORM framework. That’s why **Spring Hibernate** combination is used a lot in enterprise applications.

We will use Spring 4 and integrate it with Hibernate 3 and then update the same project to use Hibernate 4. . Since there are a lot of versions for spring and Hibernate both and Spring ORM artifact supports both Hibernate 3 and Hibernate 4.

Note that I have noticed that all spring and hibernate versions are not compatible, below versions have worked for me so I think they are compatible. If you are using some other versions and getting java.lang.NoClassDefFoundError, then it means that they are not compatible. Mostly it’s because Hibernate classes are moved from one package to another causing this error. For example org.hibernate.engine.FilterDefinition class is moved to org.hibernate.engine.spi.FilterDefinition in latest hibernate versions.

* [Spring Framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial) Version: 4.0.3.RELEASE
* Hibernate Core and Hibernate EntityManager Version: 3.6.9.Final and 4.3.5.Final
* Spring ORM Version: 4.0.3.RELEASE
* spring-context and spring-tx for core Spring functionalities. Notice that I am using version 4.0.3.RELEASE.
* spring-orm dependency for Spring ORM support, it’s required for hibernate integration in our spring project.
* hibernate-entitymanager and hibernate-core dependencies for Hibernate framework. Notice that version is 3.6.9.Final, for using Hibernate 4 all we need is to change it to 4.3.5.Final as commented in above pom.xml file.
* mysql-connector-java for MySQL driver for database connection.

<?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.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

<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/TestDB" />

<property name="username" value="pankaj" />

<property name="password" value="pankaj123" />

</bean>

<!-- Hibernate 3 XML SessionFactory Bean definition-->

<!-- <bean id="hibernate3SessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="mappingResources">

<list>

<value>person.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<value>

hibernate.dialect=org.hibernate.dialect.MySQLDialect

</value>

</property>

</bean> -->

<!-- Hibernate 3 Annotation SessionFactory Bean definition-->

<bean id="hibernate3AnnotatedSessionFactory"

class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="annotatedClasses">

<list>

<value>com.journaldev.model.Person</value>

</list>

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>

<prop key="hibernate.current\_session\_context\_class">thread</prop>

<prop key="hibernate.show\_sql">false</prop>

</props>

</property>

</bean>

<bean id="personDAO" class="com.journaldev.dao.PersonDAOImpl">

<property name="sessionFactory" ref="hibernate3AnnotatedSessionFactory" />

</bean>

</beans>

There are two ways we can provide database connection details to Hibernate, first by passing everything in **hibernateProperties** and second by creating a **DataSource** and then passing it to hibernate. I prefer the second approach, that’s why we have Apache Commons DBCP dependency to create a BasicDataSource by setting database connection properties.

For Spring and Hibernate 3 integration, Spring ORM provides two classes – org.springframework.orm.hibernate3.LocalSessionFactoryBean when hibernate mappings are XML based and org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean for annotations based mapping. I have provided simple bean configuration of LocalSessionFactoryBean in comments, if you are using XML based mappings. AnnotationSessionFactoryBean extends LocalSessionFactoryBean class, so it has all the basic properties for hibernate integration.

Spring 4 Hibernate 4 integration changes

<?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.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

<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/TestDB" />

<property name="username" value="pankaj" />

<property name="password" value="pankaj123" />

</bean>

<!-- Hibernate 4 SessionFactory Bean definition -->

<bean id="hibernate4AnnotatedSessionFactory"

class="org.springframework.orm.**hibernate4**.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="annotatedClasses">

<list>

<value>com.journaldev.model.Person</value>

</list>

</property>

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>

<prop key="hibernate.current\_session\_context\_class">thread</prop>

<prop key="hibernate.show\_sql">false</prop>

</props>

</property>

</bean>

<bean id="personDAO" class="com.journaldev.dao.PersonDAOImpl">

<property name="sessionFactory" ref="hibernate4AnnotatedSessionFactory" />

</bean>

</beans>

**For hibernate 4,** we need to use org.springframework.orm.hibernate4.LocalSessionFactoryBean

for SessionFactory bean, Spring ORM has merged both the classes for Hibernate 3 and there is a single class now, this is good to avoid confusion.

**Spring JDBC**

[Spring Framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial) provides excellent integration with JDBC API and provides JdbcTemplate utility class that we can use to avoid bolier-plate code from our database operations logic such as Opening/Closing Connection, ResultSet, and PreparedStatement etc. JdbcTemplate class can help us in writing modular code with ease, without worrying whether resources are closed properly or not.

If you look at the DAO implementation class, there is a lot of boiler-plate code where we are opening and closing Connection, PreparedStatements and ResultSet. This can lead to resource leak if someone forgets to close the resources properly. We can use org.springframework.jdbc.core.JdbcTemplate class to avoid these errors. Spring JdbcTemplate is the central class in Spring JDBC core package and provides a lot of methods to execute queries and automatically parse ResultSet to get the Object or list of Objects.

All we need is to provide the arguments as Object array and implement Callback interfaces such as **PreparedStatementSetter** and **RowMapper** for mapping arguments or converting ResultSet data to bean objects.

* No code related to opening and closing connections, statements or result set. All that is handled internally by Spring JdbcTemplate class.
* RowMapper anonymous class implementation to map the ResultSet data to Employee bean object in ***queryForObject****()* method.
* ***queryForList****()* method returns list of Map whereas Map contains the row data mapped with key as the column name and value from the database row matching the criteria.

We know that [DataSource with JNDI](http://www.journaldev.com/2513/tomcat-datasource-jndi-example-java) is the preferred way to achieve connection pooling and get benefits of container implementations.

<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns="http://www.springframework.org/schema/mvc"

xmlns:jee="http://www.springframework.org/schema/jee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:beans="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd

http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

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

<!-- DispatcherServlet Context: defines this servlet's request-processing

infrastructure -->

<!-- Enables the Spring MVC @Controller programming model -->

<annotation-driven />

<!-- Handles HTTP GET requests for /resources/\*\* by efficiently serving

up static resources in the ${webappRoot}/resources directory -->

<resources mapping="/resources/\*\*" location="/resources/" />

<!-- Resolves views selected for rendering by @Controllers to .jsp resources

in the /WEB-INF/views directory -->

<beans:bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<beans:property name="prefix" value="/WEB-INF/views/" />

<beans:property name="suffix" value=".jsp" />

</beans:bean>

<!-- Configure to plugin JSON as request and response in method handler -->

<beans:bean

class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">

<beans:property name="messageConverters">

<beans:list>

<beans:ref bean="jsonMessageConverter" />

</beans:list>

</beans:property>

</beans:bean>

<!-- Configure bean to convert JSON to POJO and vice versa -->

<beans:bean id="jsonMessageConverter"

class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

</beans:bean>

<!-- Create DataSource Bean -->

<beans:bean id="dbDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

<beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB"/>

</beans:bean>

<!-- using JEE namespace for lookup -->

<!--

<jee:jndi-lookup id="dbDataSource" jndi-name="jdbc/MyLocalDB"

expected-type="javax.sql.DataSource" />

-->

<context:component-scan base-package="com.journaldev.spring.jdbc.controller" />

</beans:beans>

**Spring 4 Security**

Now-a-days, Developing Secure Applications is very crucial aspect to avoid Malfunctioning, Stealing or hacking our confidential data or unauthorized access. We can develop secure applications using Spring Security Module to restrict access to our applications.

Initially, Spring Framework was using a separate Third-Party Framework to support Spring Applications Security: Acegi Security. But it was not easy approach to develop secure applications and had some drawbacks.

**Drawbacks of Spring Acegi Security**

1. **Lot of XML Configuration**
2. **Too much learning curve**
3. **Does not support Annotations**

To avoid all these issues, The Spring Team (Pivotal Team) has integrated “Acegi Security” framework into Spring Framework as “Spring Security” module.

Spring 4 Framework has the following modules to provide Security to the Spring-Based Applications: In Spring Framework, “Spring Security” module is the base module for rest of the Spring Security modules. Spring Security is one of the Spring Framework’s Security modules. It is a Java SE/Java EE Security Framework to provide Authentication, Authorization, SSO and other Security features for Web Applications or Enterprise Applications.

1. **Spring Security**
2. **Spring Security SAML**
3. **Spring Security OAuth**
4. **Spring Security Kerberos**
5. **Spring Cloud Security**

**Spring 4 Security Features**

Spring 3.x Security Framework provides the following Features:

1. Authentication and Authorization.
2. Supports BASIC, Digest and Form-Based Authentication.
3. Supports LDAP Authentication.
4. Supports OpenID Authentication.
5. Supports SSO (Single Sign-On) Implementation.
6. Supports Cross-Site Request Forgery (CSRF) Implementation.
7. Supports “Remember-Me” Feature through HTTP Cookies.
8. Supports Implementation of ACLs
9. Supports “Channel Security” that means automatically switching between HTTP and HTTPS.
10. Supports I18N (Internationalisation).
11. Supports JAAS (Java Authentication and Authorization Service).
12. Supports Flow Authorization using Spring WebFlow Framework.
13. Supports WS-Security using Spring Web Services.
14. Supports Both XML Configuration and Annotations. Very Less or minimal XML Configuration.

Spring 4.x Security Framework supports the following New Features:

1. Supports WebSocket Security.
2. Supports Spring Data Integration.
3. CSRF Token Argument Resolver.

**Spring 4 Security Levels**

Spring Security supports the following two Levels of Authorization

1. Method Level Authorization
2. URL Level Authorization

NOTE  
Spring Security supports “Method Level Security” by using AOP (Aspect-Oriented Programming) that means through Aspects. Spring Security supports “URL Level Security” by using Servlet filters.

**Spring 4 Security Advantages**

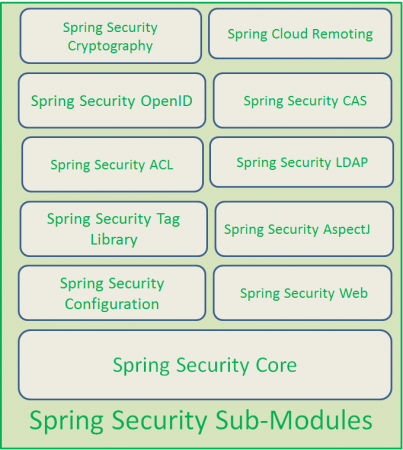
Spring 4 Security Framework provides the following Advantages:

1. Open Source Security Framework
2. Flexible, Easy to Develop and Unit Test the applications
3. Declarative Security Programming
4. Easy of Extendability
5. Easy of Maintenance
6. Takes full advantage of Spring DI(Dependency Injection) and AOP.
7. We can develop Loosely-Coupled Applications.

**Spring 4 Security Sub-Modules**

Spring 4 Security Module is again divided into 11 sub-modules. It has the following sub-modules:

1. Spring Security Core Module
2. Spring Security Configuration Module
3. Spring Security Web Module
4. Spring Security Tag Library Module
5. Spring Security AspectJ Module
6. Spring Security ACL Module
7. Spring Security LDAP Module
8. Spring Security OpenID Module
9. Spring Security CAS Module
10. Spring Security Cryptography Module
11. Spring Security Remoting Module

[](http://cdn.journaldev.com/wp-content/uploads/2015/07/spring-secruity-submodules.png)

In Spring Framework’s Security Sub-Module, Spring Security Core Sub-Module is the base module for rest of all Security Sub-modules.

To support these 11 Spring Security modules, Spring framework has the following jars:

spring-security-core-4.0.2.RELEASE.jar

spring-security-config-4.0.2.RELEASE.jar

spring-security-web-4.0.2.RELEASE.jar

spring-security-taglibs-4.0.2.RELEASE.jar

spring-security-aspects-4.0.2.RELEASE.jar

spring-security-acl-4.0.2.RELEASE.jar

spring-security-ldap-4.0.2.RELEASE.jar

spring-security-openid-4.0.2.RELEASE.jar

spring-security-cas-4.0.2.RELEASE.jar

spring-security-crypto-4.0.2.RELEASE.jar

spring-security-remoting-4.0.2.RELEASE.jar

Almost all Spring Security JARs have similar kind of Maven or Gradle dependency entries

Some of the benefits of using Spring Security are:

1. Proven technology, it’s better to use this than reinvent the wheel. Security is something where we need to take extra care, otherwise our application will be vulnerable for attackers.
2. Prevents some of the common attacks such as CSRF, session fixation attacks.
3. Easy to integrate in any web application. We don’t need to modify web application configurations; spring automatically injects security filters to the web application.
4. Provides support for authentication by different ways – in-memory, DAO, JDBC, LDAP and many more.
5. Provides option to ignore specific URL patterns, good for serving static HTML, image files.
6. Support for groups and roles.

**Web.xml**

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<!-- Spring Security Configuration File -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/appServlet/spring-security.xml</param-value>

</context-param>

<!-- Creates the Spring Container shared by all Servlet and Filters -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<listener>

<listener-class>org.springframework.security.web.session.HttpSessionEventPublisher</listener-class>

</listener>

<session-config>

<session-timeout>15</session-timeout>

</session-config>

<!-- Spring Security Filter -->

<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>

<!-- Spring MVC - START -->

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>

/WEB-INF/spring/appServlet/servlet-context.xml

</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<!-- Spring MVC - END -->

</web-app>

**spring-security.xml**

<?xml version="1.0" encoding="UTF-8"?>

<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/security http://www.springframework.org/schema/security/spring-security.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- Configuring RoleVoter bean to use custom access roles, by default roles

should be in the form ROLE\_{XXX} -->

<beans:bean id="roleVoter"

class="org.springframework.security.access.vote.RoleVoter">

<beans:property name="rolePrefix" value=""></beans:property>

</beans:bean>

<beans:bean id="accessDecisionManager"

class="org.springframework.security.access.vote.AffirmativeBased">

<beans:constructor-arg name="decisionVoters"

ref="roleVoter" />

</beans:bean>

<http authentication-manager-ref="jdbc-auth"

access-decision-manager-ref="accessDecisionManager">

<intercept-url pattern="/emp/\*\*" access="Admin" />

<form-login login-page="/login" authentication-failure-url="/denied"

username-parameter="username" password-parameter="password"

default-target-url="/home" />

<logout invalidate-session="true" logout-success-url="/login"

logout-url="/j\_spring\_security\_logout" />

<access-denied-handler error-page="/denied"/>

<session-management invalid-session-url="/login">

<concurrency-control max-sessions="1"

expired-url="/login" />

</session-management>

</http>

<authentication-manager id="in-memory-auth">

<authentication-provider>

<user-service>

<user name="pankaj" password="pankaj123" authorities="Admin" />

</user-service>

</authentication-provider>

</authentication-manager>

<authentication-manager id="dao-auth">

<authentication-provider user-service-ref="userDetailsService">

</authentication-provider>

</authentication-manager>

<beans:bean id="userDetailsService"

class="com.journaldev.spring.security.dao.AppUserDetailsServiceDAO" />

<authentication-manager id="jdbc-auth">

<authentication-provider>

<jdbc-user-service data-source-ref="dataSource"

users-by-username-query="select username,password,enabled from Employees where username = ?"

authorities-by-username-query="select username,role from Roles where username = ?" />

</authentication-provider>

</authentication-manager>

<!-- MySQL DB DataSource -->

<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/TestDB" />

<beans:property name="username" value="pankaj" />

<beans:property name="password" value="pankaj123" />

</beans:bean>

<!-- If DataSource is configured in Tomcat Servlet Container -->

<beans:bean id="dbDataSource"

class="org.springframework.jndi.JndiObjectFactoryBean">

<beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB" />

</beans:bean>

</beans:beans>

Spring 4 Security Module supports the following options to store and manage User Credentials:

1. In-Memory Store
2. Relations Databases(RDBMS)
3. No SQL Data Stores
4. LDAP

**Spring Transaction Management** is one of the most widely used and important feature of [Spring framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial). Transaction Management is a trivial task in any enterprise application. We have already learned how to use [JDBC API for Transaction Management](http://www.journaldev.com/2483/java-jdbc-transaction-management-savepoint). Spring provides extensive support for transaction management and help developers to focus more on business logic rather than worrying about the integrity of data incase of any system failures.

1. Support for Declarative Transaction Management. In this model, Spring uses AOP over the transactional methods to provide data integrity. This is the preferred approach and works in most of the cases.
2. Support for most of the transaction APIs such as JDBC, Hibernate, JPA, JDO, JTA etc. All we need to do is use proper transaction manager implementation class. For example org.springframework.jdbc.datasource.DriverManagerDataSource for JDBC transaction management and org.springframework.orm.hibernate3.HibernateTransactionManager if we are using hibernate as ORM tool.
3. Support for programmatic transaction management by using TransactionTemplate or PlatformTransactionManager implementation.

Most of the features that we would want in a transaction manager is supported by Declarative transaction management, so we would use this approach for our example project.

We will create a simple Spring JDBC project where we will update multiple tables in a single transaction. The transaction should commit only when all the JDBC statements execute successfully otherwise it should rollback to avoid data inconsistency.

If you know JDBC transaction management, you might argue that we can get do it easily by setting auto-commit to false for the connection and based on the result of all the statements, either commit or rollback the transaction. Obviously we can do it, but that will result in a lot of boiler-plate code just for transaction management. Also the same code will present in all the places where we are looking for transaction management, causing tightly coupled and non-maintainable code.

Spring declarative transaction management addresses these concerns by using Aspect Oriented Programming to achieve loose coupling and avoid boiler-plate code in our application.

<?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-4.0.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

<!-- Enable Annotation based Declarative Transaction Management -->

<tx:annotation-driven proxy-target-class="true"

transaction-manager="transactionManager" />

<!-- Creating TransactionManager Bean, since JDBC we are creating of type

DataSourceTransactionManager -->

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

<!-- MySQL DB DataSource -->

<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/TestDB" />

<property name="username" value="pankaj" />

<property name="password" value="pankaj123" />

</bean>

<bean id="customerDAO" class="com.journaldev.spring.jdbc.dao.CustomerDAOImpl">

<property name="dataSource" ref="dataSource"></property>

</bean>

<bean id="customerManager" class="com.journaldev.spring.jdbc.service.CustomerManagerImpl">

<property name="customerDAO" ref="customerDAO"></property>

</bean>

</beans>

tx:annotation-driven element is used to tell spring context that we are using annotation based transaction management configuration.

transaction-manager attribute is used to provide the transaction manager bean name. transaction-manager default value is transactionManager but I am still having it to avoid confusion.

proxy-target-class attribute is used to tell Spring context to use class based proxies, without it you will get runtime exception with message such as Exception in thread “main” org.springframework.beans.factory.BeanNotOfRequiredTypeException: Bean named ‘customerManager’ must be of type [com.journaldev.spring.jdbc.service.CustomerManagerImpl], but was actually of type [com.sun.proxy.$Proxy6]

Spring ORM example – JPA, Hibernate, and Transaction

The Spring Framework supports integration with **Hibernate, Java Persistence API (JPA)** and **Java Data Objects (JDO)** for resource management, data access object (DAO) implementations, and transaction strategies.

We use JPA**@PersistenceContext a**nnotation that indicate dependency injection to an **EntityManager**. Spring injects a proper instance of EntityManager according to the spring.xml configuration.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

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

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

">

<!-- Scans the classpath for annotated components that will be auto-registered as Spring beans -->

<context:component-scan base-package="hu.daniel.hari.learn.spring" />

<!-- Activates various annotations to be detected in bean classes e.g: @Autowired -->

<context:annotation-config />

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="org.hsqldb.jdbcDriver" />

<property name="url" value="jdbc:hsqldb:mem://productDb" />

<property name="username" value="sa" />

<property name="password" value="" />

</bean>

<bean id="entityManagerFactory"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"

p:packagesToScan="hu.daniel.hari.learn.spring.orm.model"

p:dataSource-ref="dataSource"

>

<property name="jpaVendorAdapter">

<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

<property name="generateDdl" value="true" />

<property name="showSql" value="true" />

</bean>

</property>

</bean>

<!-- Transactions -->

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="entityManagerFactory" ref="entityManagerFactory" />

</bean>

<!-- enable the configuration of transactional behavior based on annotations -->

<tx:annotation-driven transaction-manager="transactionManager" />

</beans>

ORM with AspectJ for Transaction

<!-- AOP Configuration for selecting transactional methods -->

<!-- the transactional advice (what 'happens'; see the <aop:advisor/> ) -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!-- all methods starting with 'list' or 'get' are read-only -->

<tx:method name=*"list\*"* read-only=*"true"*/>

<tx:method name=*"get\*"* read-only=*"true"*/>

<!-- for other methods use the default transaction settings -->

<tx:method name=*"\*"* />

</tx:attributes>

</tx:advice>

<!-- ensure that the above transactional advice runs for any execution

of a method in the service package -->

<aop:config>

<aop:pointcut id=*"serviceMethods"*

expression=*"execution(\* hu.daniel.hari.learn.spring.orm.service.\*.\*(..))"* />

<aop:advisor pointcut-ref=*"serviceMethods"* advice-ref=*"txAdvice"* />

</aop:config>

Spring MVC Hibernate MySQL Integration

**dataSource** bean is defined for org.apache.commons.dbcp.BasicDataSource class for basic connection pooling.

<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns="http://www.springframework.org/schema/mvc"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:beans="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

<!-- DispatcherServlet Context: defines this servlet's request-processing

infrastructure -->

<!-- Enables the Spring MVC @Controller programming model -->

<annotation-driven />

<!-- Handles HTTP GET requests for /resources/\*\* by efficiently serving

up static resources in the ${webappRoot}/resources directory -->

<resources mapping="/resources/\*\*" location="/resources/" />

<!-- Resolves views selected for rendering by @Controllers to .jsp resources

in the /WEB-INF/views directory -->

<beans:bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<beans:property name="prefix" value="/WEB-INF/views/" />

<beans:property name="suffix" value=".jsp" />

</beans:bean>

<beans:bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<beans:property name="driverClassName" value="com.mysql.jdbc.Driver" />

<beans:property name="url"

value="jdbc:mysql://localhost:3306/springjdbc" />

<beans:property name="username" value="root" />

<beans:property name="password" value="raoSB\*12" />

</beans:bean>

<!-- Hibernate 4 SessionFactory Bean definition -->

<beans:bean id="hibernate4AnnotatedSessionFactory"

class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

<beans:property name="dataSource" ref="dataSource" />

<beans:property name="annotatedClasses">

<beans:list>

<beans:value>com.journaldev.spring.model.Person</beans:value>

</beans:list>

</beans:property>

<beans:property name="hibernateProperties">

<beans:props>

<beans:prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect

</beans:prop>

<beans:prop key="hibernate.show\_sql">true</beans:prop>

</beans:props>

</beans:property>

</beans:bean>

<beans:bean id="personDAO" class="com.journaldev.spring.dao.PersonDAOImpl">

<beans:property name="sessionFactory" ref="hibernate4AnnotatedSessionFactory" />

</beans:bean>

<beans:bean id="personService" class="com.journaldev.spring.service.PersonServiceImpl">

<beans:property name="personDAO" ref="personDAO"></beans:property>

</beans:bean>

<context:component-scan base-package="com.journaldev.spring" />

<tx:annotation-driven transaction-manager="transactionManager"/>

<beans:bean id="transactionManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager">

<beans:property name="sessionFactory" ref="hibernate4AnnotatedSessionFactory" />

</beans:bean>

</beans:beans>

org.springframework.orm.hibernate4.LocalSessionFactoryBean bean is used for Hibernate 4 SessionFactory. For Hibernate 3, you will find similar classes as org.springframework.orm.hibernate3.LocalSessionFactoryBean and org.springframework.orm.hibernate3.AnnotationSessionFactoryBean.

One important point is that when we are depending on spring framework for Hibernate Session management, we should not define hibernate.current\_session\_context\_class property, otherwise you will get a lot of session transaction related issues.

**transactionManager** bean definition for **org.springframework.orm.hibernate4.HibernateTransactionManager** is required for Spring ORM to support hibernate session transaction management. For Hibernate 3, you will find similar class as **org.springframework.orm.hibernate3.HibernateTransactionManager.** Spring uses AOP for transaction management, you can now relate it with **@Transactional** annotation.

# Spring MVC Internationalization (i18n) and Localization (L10n)

Any web application with users all around the world, **internationalization** (i18n) or **localization** (L10n) is very important for better user interaction.

Most of the web application frameworks provide easy ways to localize the application based on user locale settings. Spring also follows the pattern and provides extensive support for internationalization (i18n) through the use of Spring interceptors, Locale Resolvers and Resource Bundles for different locales.

**messageSource** bean is configured to enable i18n for our application. **basename** property is used to provide the location of resource bundles. Classpath: messages means that resource bundles are located in the classpath and follows name pattern as messages\_{locale}.properties. **defaultEncoding** property is used to define the encoding used for the messages.

**localeResolver** bean of type org.springframework.web.servlet.i18n.CookieLocaleResolver is used to set a cookie in the client request so that further requests can easily recognize the user locale. For example, we can ask user to select the locale when he launches the web application for the first time and with the use of cookie, we can identify the user locale and automatically send locale specific response. We can also specify the default locale, cookie name and maximum age of the cookie before it gets expired and deleted by the client browser.

If your application maintains user sessions, then you can also use org.springframework.web.servlet.i18n.SessionLocaleResolver as localeResolver to use a locale attribute in the user’s session. The configuration is similar to CookieLocaleResolver.

<bean id="localeResolver"

class="org.springframework.web.servlet.i18n.SessionLocaleResolver">

<property name="defaultLocale" value="en" />

</bean>

If we don’t register any “localeResolver”, **AcceptHeaderLocaleResolver** will be used by default, which resolves user locale by checking the accept-language header in the client HTTP request.

**org.springframework.web.servlet.i18n.LocaleChangeInterceptor** interceptor is configured to intercept the user request and identify the user locale. The parameter name is configurable and we are using request parameter name for locale as “locale”. Without this interceptor, we won’t be able to change the user locale and send the response based on the new locale settings of the user. It needs to be part of **interceptors** element otherwise Spring won’t configure it as an interceptor.

class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<beans:property name="basename" value="classpath:messages" />

<beans:property name="defaultEncoding" value="UTF-8" />

</beans:bean>

<beans:bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

<beans:property name="defaultLocale" value="en" />

<beans:property name="cookieName" value="myAppLocaleCookie"></beans:property>

<beans:property name="cookieMaxAge" value="3600"></beans:property>

</beans:bean>

<interceptors>

<beans:bean

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">

<beans:property name="paramName" value="locale" />

</beans:bean>

</interceptors>

# Spring MVC Interceptor

Spring Interceptor are used to intercept client requests and process them. Sometimes we want to intercept the HTTP Request and do some processing before handing it over to the controller handler methods. That’s where Spring MVC Interceptor come handy.

Just like we have [Struts2 Interceptors](http://www.journaldev.com/2210/struts-2-interceptor-example), we can create our own Spring interceptor by either implementing org.springframework.web.servlet.HandlerInterceptor interface or by overriding abstract class org.springframework.web.servlet.handler.HandlerInterceptorAdapter that provides the base implementation of HandlerInterceptor interface.

# Spring Interceptor – HandlerInterceptor

Spring **HandlerInterceptor** declares three methods based on where we want to intercept the HTTP request.

**boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**: This method is used to intercept the request before it’s handed over to the handler method. This method should return ‘true’ to let Spring know to process the request through another spring interceptor or to send it to handler method if there are no further spring interceptors.

If this method returns ‘false’ [Spring framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial) assumes that request has been handled by the spring interceptor itself and no further processing is needed. We should use response object to send response to the client request in this case.

Object *handler* is the chosen handler object to handle the request. This method can throw Exception also, in that case [Spring MVC Exception Handling](http://www.journaldev.com/2651/spring-mvc-exception-handling-controlleradvice-exceptionhandler-handlerexceptionresolver) should be useful to send error page as response.

**void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)**: This HandlerInterceptor interceptor method is called when HandlerAdapter has invoked the handler but DispatcherServlet is yet to render the view. This method can be used to add additional attribute to the ModelAndView object to be used in the view pages. We can use this spring interceptor method to determine the time taken by handler method to process the client request.

**void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)**: This is a HandlerInterceptor callback method that is called once the handler is executed and view is rendered.

If there are multiple spring interceptors configured, *preHandle()* method is executed in the order of configuration whereas *postHandle()* and *afterCompletion()* methods are invoked in the reverse order.

<!-- Configuring interceptors based on URI -->

<interceptors>

<interceptor>

<mapping path="/home" />

<beans:bean class="com.journaldev.spring.RequestProcessingTimeInterceptor"></beans:bean>

</interceptor>

</interceptors>

# Spring MVC File Upload

File Uploading is a very common task in any web application. We have earlier seen how to [upload files in Servlet](http://www.journaldev.com/1964/servlet-upload-file-download-example) and [Struts2 File Uploading](http://www.journaldev.com/2192/struts-2-file-upload-example). Spring MVC framework provide support for uploading files by integrating Apache Commons FileUpload API.

<form method="POST" action="uploadFile" enctype="multipart/form-data">

Notice that these files are simple HTML files, I am not using any JSP or Spring tags to avoid complexity. The important point to note is that form **enctype** should be **multipart/form-data**, so that Spring web application knows that the request contains file data that needs to be processed.

Also note that for multiple files, the form field “file” and “name” are same in the input fields, so that the data will be sent in the form of array. We will take the input array and parse the file data and store it in the given file name.

<beans:bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

<!-- setting maximum upload size -->

<beans:property name="maxUploadSize" value="100000" />

</beans:bean>

# Spring Validation

When we accept user inputs in any web application, it become necessary to validate them. We can validate the user input at client side using JavaScript but it’s also necessary to validate them at server side to make sure we are processing valid data incase user has javascript disabled.

[Spring MVC Framework](http://www.journaldev.com/2433/spring-mvc-tutorial) supports JSR-303 specs by default and all we need is to add JSR-303 and it’s implementation dependencies in Spring MVC application. Spring also provides @Validator annotation and BindingResult class through which we can get the errors raised by Validator implementation in the controller request handler method.

We can create our custom validator implementations by two ways –

* First one is to create an annotation that confirms to the JSR-303 specs and implement it’s Validator class.
* Second approach is to implement the org.springframework.validation.Validator interface and add set it as validator in the Controller class using @InitBinder annotation.

# Spring MVC Exception Handling – @ControllerAdvice, @ExceptionHandler, HandlerExceptionResolver

**Spring MVC Exception Handling is very important to make sure you are not sending server exceptions to client. Today we will look into Spring Exception Handling using @ExceptionHandler, @ControllerAdvice and HandlerExceptionResolver. Any**[**web application**](http://www.journaldev.com/1854/java-web-application-tutorial-for-beginners)**requires good design for exception handling because we don’t want to serve container generated page when any unhandled exception is thrown by our application.**

Spring MVC Framework provides following ways to help us achieving robust exception handling.

1. **Controller Based** – We can define exception handler methods in our controller classes. All we need is to annotate these methods with @**ExceptionHandler** annotation. This annotation takes Exception class as argument. So if we have defined one of these for Exception class, then all the exceptions thrown by our request handler method will have handled.

These exception handler methods are just like other request handler methods and we can build error response and respond with different error page. We can also send JSON error response that we will look later on in our example.

If there are multiple exception handler methods defined, then handler method that is closest to the Exception class is used. For example, if we have two handler methods defined for IOException and Exception and our request handler method throws IOException, then handler method for IOException will get executed.

1. **Global Exception Handler** – Exception Handling is a cross-cutting concern, it should be done for all the pointcuts in our application. We have already looked into [Spring AOP](http://www.journaldev.com/2583/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations) and that’s why Spring provides @**ControllerAdvice** annotation that we can use with any class to define our global exception handler.

The handler methods in Global Controller Advice is same as Controller based exception handler methods and used when controller class is not able to handle the exception.

1. **HandlerExceptionResolver**– For generic exceptions, most of the times we serve static pages. Spring Framework provides **HandlerExceptionResolver** interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

**SimpleMappingExceptionResolver** is the default implementation class, it allows us to configure exceptionMappings where we can specify which resource to use for a particular exception. We can also override it to create our own global handler with our application specific changes, such as logging of exception messages.

**annotation-driven element is used to let Controller servlet know that annotations will be used for bean configurations.**

<!-- Enables the Spring MVC @Controller programming model -->

<annotation-driven />

**resources element defines the location where we can put static files such as images, html pages etc that we don’t want to get through Spring framework.**

<!-- Handles HTTP GET requests for /resources/\*\* by efficiently serving up static resources in the ${webappRoot}/resources directory -->

<resources mapping="/resources/\*\*" location="/resources/" />

**InternalResourceViewResolver is the view resolver, we can provide view pages location through prefix and suffix properties. So all our JSP pages should be in /WEB-INF/views/ directory.**

<! -- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->

<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

* Spring @**Autowired** annotation is used for automatic dependency injection. [Spring framework](http://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial) is built on [dependency injection](http://www.journaldev.com/2410/spring-dependency-injection) and we inject the class dependencies through spring bean configuration file.

## Spring @Autowired Annotation

There are different ways through which we can autowire a spring bean.

1. **autowire byName** – For this type of autowiring, setter method is used for dependency injection. Also the variable name should be same in the class where we will inject the dependency and in the spring bean configuration file.
2. **autowire byType** – For this type of autowiring, class type is used. So there should be only one bean configured for this type in the spring bean configuration file.
3. **autowire by constructor** – This is almost similar to autowire byType, the only difference is that constructor is used to inject the dependency.
4. **autowire by autodetect** – If you are on Spring 3.0 or older versions, this is one of the autowire options available. This option was used for autowire by constructor or byType, as determined by Spring container. Since we already have so many options, this option is deprecated. I will not cover this option in this tutorial.
5. **@Autowired annotation** – We can use Spring @Autowired annotation for spring bean autowiring. @Autowired annotation can be applied on variables and methods for autowiring byType. We can also use @Autowired annotation on constructor for constructor based spring autowiring.

For @Autowired annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by **context:annotation-config** element or by defining a bean of type **org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.**

1. **@Qualifier annotation –** This annotation is used to avoid conflicts in bean mapping and we need to provide the bean name that will be used for autowiring. This way we can avoid issues where multiple beans are defined for same type. This annotation usually works with the @Autowired annotation. For constructors with multiple arguments, we can use this annotation with the argument names in the method.

By default spring bean autowiring is turned off. Spring bean autowire default value is “default” that means no autowiring is to be performed. autowire value “no” also have the same behavior.

* @**Configuration** annotation is used to let spring know that it’s a Configuration class.
* @**ComponentScan** annotation is used with @Configuration annotation to specify the packages to look for Component classes.

**Ex: @Configuration**

**@ComponentScan(value={"com.journaldev.spring.di.consumer"})**

* @**Bean** annotation is used to let spring framework know that this method should be used to get the bean implementation to inject in Component classes.

Important points about spring bean configuration file are:

* **beans** element default-autowire is used to define the default autowiring method. Here I am defining the default autowiring method to be byName.
* **beans** element default-autowire-candidates is used to provide the pattern for bean names that can be used for autowiring. For simplicity I am allowing all the bean definitions to be eligible for autowiring, however if we can define some pattern for autowiring. For example, if we want only DAO bean definitions for autowiring, we can specify it as **default-autowire-candidates="\*DAO".**
* **autowire-candidate="false"** is used in a bean definition to make it ineligible for autowiring. It’s useful when we have multiple bean definitions for a single type and we want some of them not to be autowired. For example, in above spring bean configurations “employee1” bean will not be used for autowiring.
* **autowire** attribute byName, byType and constructor is self understood, nothing much to explain there.
* **context:annotation-config** is used to enable annotation based configuration support. Notice that employeeAutowiredByTypeService and employeeAutowiredByConstructorService beans don’t have autowire attributes.

The project dependencies that I am interested in are;

* **spring-context**: Spring Core dependency. Notice the exclusion of commons logging in favor of SLF4J.
* **spring-webmvc:** Spring artifact for MVC support
* **aspectjrt:** AspectJ API reference
* **SLF4J and Log4j**: For logging purposes, Spring is very easy to configure for log4j or [Java Logging API](http://www.journaldev.com/977/logger-in-java-logging-example)because of SLF4J integration.
* **javax.inject** – JSR330 API for dependency injection.

## Spring Bean Configuration

* Spring Framework provides three ways to configure beans to be used in the application.
* **Annotation Based Configuration** – By using @**Service** or @**Component** annotations. Scope details can be provided with @**Scope** annotation.
* **XML Based Configuration** – By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
* **Java Based Configuration** – Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @**Configuration**, @**ComponentScan** and @**Bean**.

**Deployment Descriptor**

We need to configure our application for Spring Framework, so that the configuration metadata will get loaded and context will be initialized.

* <?xml version="1.0" encoding="UTF-8"?>
* <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
* xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
* xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">
* <!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
* **<context-param>**
* **<param-name>contextConfigLocation</param-name>**
* **<param-value>/WEB-INF/spring/root-context.xml</param-value>**
* **</context-param>**
* **<!-- Creates the Spring Container shared by all Servlets and Filters -->**
* **<listener>**
* **<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>**
* **</listener>**
* **<!-- Processes application requests -->**
* **<servlet>**
* **<servlet-name>appServlet</servlet-name>**
* **<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>**
* **<init-param>**
* **<param-name>contextConfigLocation</param-name>**
* **<param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>**
* **</init-param>**
* <load-on-startup>1</load-on-startup>
* </servlet>
* <servlet-mapping>
* <servlet-name>appServlet</servlet-name>
* <url-pattern>/</url-pattern>
* </servlet-mapping>
* </web-app>

## Spring Aware Interfaces

Sometimes we need Spring Framework objects in our beans to perform some operations, for example reading ServletConfig and ServletContext parameters or to know the bean definitions loaded by the ApplicationContext. That’s why spring framework provides a bunch of \*Aware interfaces that we can implement in our bean classes.

org.springframework.beans.factory.Aware is the root marker interface for all these Aware interfaces. All of the \*Aware interfaces are sub-interfaces of Aware and declare a single setter method to be implemented by the bean. Then spring context uses setter-based dependency injection to inject the corresponding objects in the bean and make it available for our use.

Spring Aware interfaces are similar to [servlet listeners](http://www.journaldev.com/1945/servletcontextlistener-servlet-listener-example) with callback methods and implementing [observer design pattern](http://www.journaldev.com/1739/observer-design-pattern-in-java).

Some of the important Aware interfaces are:

* **ApplicationContextAware** – to inject ApplicationContext object, example usage is to get the array of bean definition names.
* **BeanFactoryAware** – to inject BeanFactory object, example usage is to check scope of a bean.
* **BeanNameAware** – to know the bean name defined in the configuration file.
* **ResourceLoaderAware** – to inject ResourceLoader object, example usage is to get the input stream for a file in the classpath.
* **ServletContextAware** – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
* **ServletConfigAware** – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

# Spring AOP Method Profiling

This example shows how simple to configure Spring for profiling any method in any service (or other) classes **without writing a single line of profiling** (method process time logger) **code** **in** any of the service **classes**. Aspect Oriented Programming (AOP) let you to separate the (usually duplicated and boilerplate) profiling code from the service code. **No more boilerplate profiling code in your business logic** that always result in a mess.

We write our profiler code only once in a separated class (SimpleProfiler.java), and that’s all, the rest is only AOP configuration in spring.xml that has to be done for working.

* Profiling any (service) classes,
* Without touching (service) classes’ code,
* Through Spring-AOP approach.

<!-- AOP Configuration for profiling -->

    <!-- Our profiler class that we want to use to measure process time of service methods. -->

    <bean id="profiler" class="hu.daniel.hari.learn.spring.aop.profiling.core.profiler.SimpleProfiler" />

    <!-- Spring AOP -->

    <aop:config>

        <aop:aspect ref="profiler">

            <!-- Catch all method in the service package through AOP. -->

            <aop:pointcut id="serviceMethod"

                expression="execution(\* hu.daniel.hari.learn.spring.aop.profiling.service.\*.\*(..))" />

            <!-- For these methods use the profile named method in the profiler class we defined below. -->

            <aop:around pointcut-ref="serviceMethod" method="profile" />

        </aop:aspect>

    </aop:config>

# Spring 4.x Features

* Spring 4.0 was released in December 2013 and developers are migrating to it gradually. New Java projects are recommended to adopt it directly. It was one of the major version release for Spring framework. Major features on Spring 4 is the complete support for Java SE 8 and Groovy. Following are the Spring 4.x features highlights and lets see about each of them in \*/+-detail in coming tutorials.



## Spring 4.0 Features

## Many improvements to Spring Core Container like generics as autowiring qualifiers, conditional annotation, ordered auto wiring qualifiers and ordered auto wiring into lists and arrays.

## Complete support for [Java 8 features](http://javapapers.com/java/java-8-features/). One important point to note is, minimum requirement to using Spring 4 is Java SE6. Similarly requires Servlet 3.0+ for Spring Web modules

## Supports conditional Bean Creation.

## Spring 4.0.x has added modules new spring-websocket and spring-messaging for WebSocket-based communication and support STOMP.

## Support for external bean configuration using Groovy DSL.

## Supports SockJS, STOMP Messaging and Java Websocket API.

## Improvements to RestTemplate

## Loads of improvements to Spring Test module.

## Lots of third party dependencies have been upgraded their respective latest versions. Like Hibernate 3.6+, EhCache 2.1+, Quartz 1.8+, Groovy 1.8+, and Joda-Time 2.0+.

## Removed many deprecated packages and methods

## Spring 4.1 Improvements

* Spring 4.1.5 was released on February 20, 2015 with lots of new features and improvements.
* Many improvements to Spring JMS Module. Added new annotations and also provided some improvements to “**jms:annotation-driven**”namespace.
* Many improvements to Spring Web modules
* Improvements to WebSocket module
* Improvements to SockJS &STOM Messaging
* Improvements to Spring Test module
* Improvements in Spring Cache
* Supports JCache Annotations
* If you are using Spring framework it is good time to upgrade to Spring 4.x

**Learn what you can do with Spring Boot?**

Spring Boot offers a fast way to build applications. It looks at your classpath and at beans you have configured, makes reasonable assumptions about what you’re missing, and adds it. With Spring Boot you can focus more on business features and less on infrastructure.

For example:

* Got Spring MVC? There are several specific beans you almost always need, and Spring Boot adds them automatically. A Spring MVC app also needs a servlet container, so Spring Boot automatically configures embedded Tomcat.
* Got Jetty? If so, you probably do NOT want Tomcat, but instead embedded Jetty. Spring Boot handles that for you.
* Got Thymeleaf? There are a few beans that must always be added to your application context; Spring Boot adds them for you.

These are just a few examples of the automatic configuration Spring Boot provides. At the same time, Spring Boot doesn’t get in your way. For example, if Thymeleaf is on your path, Spring Boot adds a SpringTemplateEngine to your application context automatically. But if you define your own SpringTemplateEngine with your own settings, then Spring Boot won’t add one. This leaves you in control with little effort on your part.

|  |  |
| --- | --- |
|  | Spring Boot doesn’t generate code or make edits to your files. Instead, when you start up your application, Spring Boot dynamically wires up beans and settings and applies them to your application context. |

**What is bean wiring?**

Combining together beans within the spring container is known as bean wiring or wiring. When wiring beans, you should tell the container what beans are needed and how the container should use dependency injection to tie them together.

**Explain Bean lifecycle in spring framework?**

1. The spring container finds the bean’s definition from the XML file and instantiates the bean.
2. Using the dependency injection, spring populates all of the properties as specified in the bean definition.
3. If the bean implements the **BeanNameAware** interface, the factory calls **setBeanName**() passing the bean’s ID.
4. If the bean implements the **BeanFactoryAware** interface, the factory calls **setBeanFactory**(), passing an instance of itself.
5. If there are any **BeanPostProcessor**s associated with the bean, their post- **ProcessBeforeInitialization**() methods will be called.
6. If an init-method is specified for the bean, it will be called.
7. Finally, if there are any BeanPostProcessors associated with the bean, their **postProcessAfterInitialization**() methods will be called.

**What are important ApplicationContext implementations in spring framework?**

**ClassPathXmlApplicationContext** – This context loads a context definition from an XML file located in the class path, treating context definition files as class path resources.

**FileSystemXmlApplicationContext** – This context loads a context definition from an XML file in the filesystem.

**XmlWebApplicationContext** – This context loads the context definitions from an XML file contained within a web application.

**Difference between Dirty Read, Non Repeatable Read and Phantom Read in Database.**

D**irty Read:-**

Dirty read occurs when one transaction is changing the record, and the other transaction can read this record before the first transaction has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid data.

**Dirty Read Example:-**

Transaction A begins.  
UPDATE EMPLOYEE SET SALARY = 10000 WHERE EMP\_ID= ‘123’;

Transaction B begins.  
SELECT \* FROM EMPLOYEE;  
(Transaction B sees data which is updated by transaction A. But, those updates have not yet been committed.)

**Non-Repeatable Read:-**

Non Repeatable Reads happen when in a same transaction same query yields to a different result. This occurs when one transaction repeatedly retrieves the data, while a difference transaction alters the underlying data. This causes the different or non-repeatable results to be read by the first transaction.

**Non-Repeatable Example:-**

Transaction A begins.  
SELECT \* FROM EMPLOYEE WHERE EMP\_ID= ‘123’;

Transaction B begins.  
UPDATE EMPLOYEE SET SALARY = 20000 WHERE EMP\_ID= ‘123’;  
(Transaction B updates rows viewed by the transaction A before transaction B commits.) If Transaction A issues the same SELECT statement, the results will be different.

**Phantom Read:-**

Phantom read occurs where in a transaction execute same query more than once, and the second transaction result set includes rows that were not visible in the first result set. This is caused by another transaction inserting new rows between the execution of the two queries. This is similar to a non-repeatable read, except that the number of rows is changed either by insertion or by deletion.

**Phantom Read Example:-**

Transaction A begins.  
SELECT \* FROM EMPLOYEE WHERE SALARY > 10000 ;

Transaction B begins.  
INSERT INTO EMPLOYEE (EMP\_ID, FIRST\_NAME, DEPT\_ID, SALARY) VALUES (‘111′, ‘Jamie’, 10, 35000);  
Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.

**Following are the possible values for isolation level:**

|  |  |
| --- | --- |
| S.N. | Isolation & Description |
| 1 | TransactionDefinition.ISOLATION\_DEFAULT  This is the default isolation level. |
| 2 | TransactionDefinition.ISOLATION\_READ\_COMMITTED  Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur. |
| 3 | TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED  Indicates that dirty reads, non-repeatable reads and phantom reads can occur. |
| 4 | TransactionDefinition.ISOLATION\_REPEATABLE\_READ  Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur. |
| 5 | TransactionDefinition.ISOLATION\_SERIALIZABLE  Indicates that dirty reads, non-repeatable reads and phantom reads are prevented. |

**Following are the possible values for propagation types:**

|  |  |
| --- | --- |
| S.N. | Propagation & Description |
| 1 | TransactionDefinition.PROPAGATION\_MANDATORY  Support a current transaction; throw an exception if no current transaction exists. |
| 2 | TransactionDefinition.PROPAGATION\_NESTED  Execute within a nested transaction if a current transaction exists. |
| 3 | TransactionDefinition.PROPAGATION\_NEVER  Do not support a current transaction; throw an exception if a current transaction exists. |
| 4 | TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED  Do not support a current transaction; rather always execute non-transactionally. |
| 5 | TransactionDefinition.PROPAGATION\_REQUIRED  Support a current transaction; create a new one if none exists. |
| 6 | TransactionDefinition.PROPAGATION\_REQUIRES\_NEW  Create a new transaction, suspending the current transaction if one exists. |
| 7 | TransactionDefinition.PROPAGATION\_SUPPORTS  Support a current transaction; execute non-transactionally if none exists. |
| 8 | TransactionDefinition.TIMEOUT\_DEFAULT  Use the default timeout of the underlying transaction system, or none if timeouts are not supported. |

**What are different modules in SPRING?**

* The Core container module
* Application context module
* AOP module (Aspect Oriented Programming)
* JDBC abstraction and DAO module
* O/R mapping integration module (Object/Relational)
* Web module
* MVC framework module

**What is the Core container module?**

This module provides the fundamental functionality of the spring framework. In this module **BeanFactory** is the heart of any spring-based application. The entire framework was built on the top of this module. This module makes the Spring container.

**Why most users of the Spring Framework choose declarative transaction management?**

Most users of the Spring Framework choose declarative transaction management because 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.

**What is RowCallbackHandler?**

The **RowCallbackHandler** interface extracts values from each row of a ResultSet.

Has one method – processRow (ResultSet) called for each row in ResultSet Typically stateful?

**What is Application context module?**

The Application context module makes spring a framework. This module extends the concept of BeanFactory, providing support for **internationalization** (I18N) **messages, application lifecycle events, and validation**. This module also supplies many enterprise services such **JNDI access, EJB integration, remoting, and scheduling**. It also provides support to other framework.

#### What is AOP module?

The AOP module is used for developing aspects for our Spring-enabled application. Much of the support has been provided by the AOP Alliance in order to ensure the interoperability between spring and other AOP frameworks. This module also introduces metadata programming to spring. Using spring’s metadata support, we will be able to add annotations to our source code that instruct spring on where and how to apply aspects.

#### What is JDBC abstraction and DAO module?

Using this module we can keep up the database code clean and simple, and prevent problems that result from a failure to close database resources. A new layer of meaningful exceptions on top of the error messages given by several database servers is bought in this module. In addition, this module uses Spring’s AOP module to provide transaction management services for objects in a spring application.

#### What are object/relational mapping integration module?

Spring also supports for using of an object/relational mapping (ORM) tool over straight JDBC by providing the ORM module. Spring provide support to tie into several popular ORM frameworks, including Hibernate, JDO, and iBATIS SQL Maps. Spring’s transaction management supports each of these ORM frameworks as well as JDBC.

#### What is web module?

Spring comes with a full-featured MVC framework for building web applications. Although spring can easily be integrated with other MVC frameworks, such as Struts, spring’s MVC framework uses IoC to provide for a clean separation of controller logic from business objects. It also allows you to declaratively bind request parameters to your business objects. It also can take advantage of any of spring’s other services, such as **I18N messaging and validation.**

**Struts-2 integration with spring**

**In Web.xml**

<filter-class>

**org.apache.struts2.dispatcher.ng.filter**.**StrutsPrepareAndExecuteFilter**

</filter-class>

</filter>

<listener-class>

**org.springframework.web.context.ContextLoaderListener**

</listener-class>

**What is Bean Factory, Have you used XMLBean Factory?**

A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application’s configuration and dependencies from the actual application code.

XMBeanFactory is one of the implementation of bean factory **org.springframework.beans.factory.xml.XmlBeanFactory** is used to create bean instance defined in our xml file.

BeanFactory has many implementations in spring. But one of the most useful one is **org.springframework.beans.factory.xml.XmlBeanFactory**, which loads its beans based on the definitions contained in an XML file. To create an **XmlBeanFactory**, pass a **java.io.InputStream** to the constructor. The InputStream will provide the XML to the factory. For example, the following code snippet uses a java.io.FileInputStream to provide a bean definition XML file to XmlBeanFactory.

BeanFactory factory=new XmlBeanFactory(new FileInputStream(“beans.xml””));

(Or)

ClassPathResource resource=new ClassPathResource(“beans.xml”);

XmlBeansFactory factory=new XmlBeansFactory(resource);

**Cache vs. Buffer**

The terms "**buffer**" and "cache" tend to be used interchangeably; note however they represent different things. **A buffer is used traditionally as an intermediate temporary store for data between a fast and a slow entity**. As one party would have to *wait* for the other affecting performance, the buffer alleviates this by allowing entire blocks of data to move at once rather then in small chunks. The data is written and read only once from the buffer. Furthermore, the buffers are *visible* to at least one party which is aware of it.

A **cache** on the other hand by definition is hidden and neither party is aware that caching occurs. It as well improves performance but does that by allowing the same data to be read multiple times in a fast fashion.

**@Cacheable annotation**

As the name implies, @**Cacheable** is used to demarcate methods that are cacheable - that is, methods for whom the result is stored into the cache so on subsequent invocations (with the same arguments), the value in the cache is returned without having to actually execute the method. In its simplest form, the annotation declaration requires the name of the cache associated with the annotated method:

**@Cacheable("books")**

**public Book findBook(ISBN isbn) {...}**

In the snippet above, the method findBook is associated with the cache named books. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated. While in most cases, only one cache is declared, the annotation allows multiple names to be specified so that more than one cache are being used. In this case, each of the caches will be checked before executing the method - if at least one cache is hit, then the associated value will be returned:

|  |  |
| --- | --- |
|  | All the other caches that do not contain the value will be updated as well even though the cached method was not actually executed. |

@Cacheable({"books", "isbns"})

public Book findBook(ISBN isbn) {...}

**Enable caching annotations**

It is important to note that even though declaring the cache annotations does not automatically trigger their actions - like many things in Spring, the feature has to be declaratively enabled (which means if you ever suspect caching is to blame, you can disable it by removing only one configuration line rather than all the annotations in your code).

To enable caching annotations add the annotation @**EnableCaching** to one of your @Configuration classes:

**@Configuration**

**@EnableCaching**

public class AppConfig {

}

Alternatively for XML configuration use the **cache:annotation-driven** element:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:cache="http://www.springframework.org/schema/cache"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

**http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-cache.xsd">**

**<cache:annotation-driven />**

</beans>

Both the **cache:annotation-driven** element and **@EnableCaching** annotation allow various options to be specified that influence the way the caching behavior is added to the application through AOP.

The configuration is intentionally similar with that of [@Transactional](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#tx-annotation-driven-settings):

The **tx** tags deal with configuring all of those beans in Spring’s comprehensive support for transactions.

xmlns:tx="http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

The **aop** tags deal with configuring all things AOP in Spring: this includes Spring’s own proxy-based AOP framework and Spring’s integration with the AspectJ AOP framework.

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"

The **context** tags deal with Application Context configuration that relates to plumbing - that is, not usually beans that are important to an end-user but rather beans that do a lot of grunt work in Spring, such as BeanfactoryPostProcessors.

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

The **jms** tags deal with configuring JMS-related beans such as Spring’s [MessageListenerContainers](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jms-mdp).

ttp://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms.xsd

The **lang** tags deal with exposing objects that have been written in a dynamic language such as JRuby or Groovy as beans in the Spring container.

http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd

The **jee** tags deal with Java EE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd

**Defining new Advice types**

Spring AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to the out-of-the-box interception around advice, before, throws advice and after returning advice.

The **org.springframework.aop.framework.adapter** package is an SPI package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom Advice type is that it must implement the **org.aopalliance.aop.Advice** tag interface.

**How to filter components in auto scanning?**

To include and exclude components based on your requirements. You can filter your components during enabling auto component scanning level, it should be configured at xml based configuration level. Here is an example for including auto scan components:

Here is the xml based configuration, which includes a filter to include components; the filter is specified using "include-filter" tag. Note that below filtering components based on regular expression, and annotation based. Below xml configuration including all Components annotated with @Component annotation and any class containing "Service" String.

 <context:component-scan base-package="com.java2novice">

        <context:include-filter type="regex"

            expression="com.java2novice.service.\*Service.\*" />

        <context:include-filter type="annotation"

            expression="org.springframework.stereotype.Component" />

     </context:component-scan>

**How to read property file in spring 3.0 using java based configuration?**

@Configuration

@PropertySource("classpath:/db.properties")

Public class MyApplicationConfig {

@Autowired

Environment env;

@Bean (name="dbConfig")

Public MyDbConfig getDbConfig (){

**How to read property file in spring using xml based configuration file?**

To load property file from the classpath using xml based configuration. Declare your property file in your xml based configuration file using "context: property-placeholder" tag, and refer property key any where in the xml based configuration file using ${db.host.url} syntax.

In the spring bean configurations, bean attribute called 'scope' defines what kind of object has to created and returned. There are 5 types of bean scopes available, they are:

1) **singleton**: Returns a single bean instance per Spring IoC container.

2) **prototype**: Returns a new bean instance each time when requested.

3) **request**: Returns a single instance for every HTTP request call.

4) **session**: Returns a single instance for every HTTP session.

5) **global session**: global session scope is equal as session scope on portlet-based web applications.

If no bean scope is specified in bean configuration file, then it will be by default 'singleton'.

In spring the beans are managed by Spring IoC container, these are backbone of the application. You can instantiate and manage them in your application using configurations. In xml based spring bean configurations, using <bean> tag, you can manage them. Here we have given complete list of bean tag properties:

**name / id:**

This attribute specifies the bean unique identifier. In XML based configuration metadata, you use the id and/or name attributes to specify the bean identifier.

**class:**

This attribute is mandatory and specify the bean class to be used to create the bean. You should specify fully qualified class name. Include package name.

**scope:**

This attribute specifies the scope of the objects created from a particular bean definition. The scope values can be prototype, singleton, request, session, and global session.

**constructor-arg:**

This is used to inject the dependencies through bean constructor.

**properties:**

This attribute is used to inject the dependencies through setter method.

**autowiring mode:**

This is used to inject the dependencies.

**lazy-init** (lazy-initialization mode):

A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

**init-method** (initialization method):

A callback to be called just after all necessary properties on the bean have been set by the container. This is part of bean lifecycle.

**destroy-method** (destruction method):

A callback to be used when the container containing the bean is destroyed. This is part of bean lifecycle.

Configure default initialization and destroy methods in all spring beans

  default-init-method="init"

     default-destroy-method="destroy">

Configure spring bean initialization and destroy method calls using java annotations @**PostConstruct** and @**PreDestroy**. These annotations are not belong to spring API, these are part of J2ee library common-annotations.jar file

Spring Annotations: Contents:

|  |  |
| --- | --- |
| **Annotation** | **Package Detail/Import statement** |
| [@Service](http://www.techferry.com/articles/spring-annotations.html#Service) | import org.springframework.stereotype.Service; |
| [@Repository](http://www.techferry.com/articles/spring-annotations.html#Repository) | import org.springframework.stereotype.Repository; |
| [@Component](http://www.techferry.com/articles/spring-annotations.html#Component) | import org.springframework.stereotype.Component; |
| [@Autowired](http://www.techferry.com/articles/spring-annotations.html#Autowired) | import org.springframework.beans.factory.annotation.Autowired; |
| [@Transactional](http://www.techferry.com/articles/spring-annotations.html#Transactional) | import org.springframework.transaction.annotation.Transactional; |
| [@Scope](http://www.techferry.com/articles/spring-annotations.html#Scope) | import org.springframework.context.annotation.Scope; |
| [**Spring MVC Annotations**](http://www.techferry.com/articles/spring-annotations.html#MVC) | |
| [@Controller](http://www.techferry.com/articles/spring-annotations.html#Controller) | import org.springframework.stereotype.Controller; |
| [@RequestMapping](http://www.techferry.com/articles/spring-annotations.html#RequestMapping) | import org.springframework.web.bind.annotation.RequestMapping; |
| [@PathVariable](http://www.techferry.com/articles/spring-annotations.html#PathVariable) | import org.springframework.web.bind.annotation.PathVariable; |
| [@RequestParam](http://www.techferry.com/articles/spring-annotations.html#RequestParam) | import org.springframework.web.bind.annotation.RequestParam; |
| [@ModelAttribute](http://www.techferry.com/articles/spring-annotations.html#ModelAttribute) | import org.springframework.web.bind.annotation.ModelAttribute; |
| [@SessionAttributes](http://www.techferry.com/articles/spring-annotations.html#SessionAttributes) | import org.springframework.web.bind.annotation.SessionAttributes; |
| [**Spring Security Annotations**](http://www.techferry.com/articles/spring-annotations.html#SpringSecurity) | |
| [@PreAuthorize](http://www.techferry.com/articles/spring-annotations.html#PreAuthorize) | import org.springframework.security.access.prepost.PreAuthorize; |

For spring to process annotations, add the following lines in your application-context.xml file.

**<context:annotation-config />**

**<context:component-scan base-package="...specify your package name..." />**

@**Service**

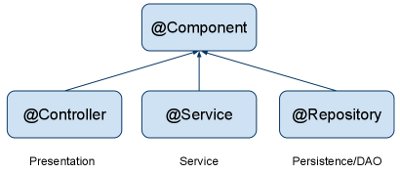
Annotate all your service classes with @**Service**. All your business logic should be in Service classes.

@**Repository**

Annotate all your DAO classes with @**Repository**. All your database access logic should be in DAO classes

@**Component**

Annotate your other components (for example REST resource classes).

@**Component** is a generic stereotype for any Spring-managed component. **@Repository, @Service, and @Controller** are specializations of @**Component** for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.   
  
@**Autowired**

Let spring auto-wire other beans into your classes using @Autowired annotation. Spring beans can be wired by name or by type.

@**Autowire** by default is a type driven injection.

@**Qualifier** spring annotation can be used to further fine-tune autowiring.

@**Resource** (javax.annotation.Resource) annotation can be used for wiring by name. Beans that are themselves defined as a **collection** or **map** type cannot be injected through @**Autowired**, because type matching is not properly applicable to them. Use @**Resource** for such beans, referring to the specific collection or map bean by unique name.

@**Transactional**

Configure your transactions with @Transactional spring annotation.

To activate processing of spring’s @Transactional annotation, use the **<tx: annotation-driven**/> element in your spring's configuration file.

The default @Transactional settings are as follows:

Propagation setting is PROPAGATION\_REQUIRED.

Isolation level is ISOLATION\_DEFAULT.

Transaction is read/write.

Transaction timeout defaults to the default timeout of the underlying transaction system or to none if timeouts are not supported.

Any Runtime Exception triggers rollback, and any checked Exception does not.

These default settings can be changed using various properties of the

@**Transactional** spring annotation.

Specifying the @Transactional annotation on the bean class means that it applies to all applicable business methods of the class. Specifying the annotation on a method applies it to that method only. If the annotation is applied at both the class and the method level, the method value overrides if the two disagree

@**Scope**

As with Spring-managed components in general, the default and most common scope for auto detected components is singleton. To change this default behavior, use @Scope spring annotation.

@**Controller**

Annotate your controller classes with @**Controller**.

@**RequestMapping**

You use the @RequestMapping spring annotation to map URLs 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.

@**PathVariable**

You can use the @PathVariable spring annotation on a method argument to bind it to the value of a URI template variable. In our example below, a request path of /company/techferry will bind companyName variable with 'techferry' value.

@**RequestParam**

You can bind request parameters to method variables using spring annotation

@**RequestParam**

Similarly, you can use spring annotation @RequestHeader to bind request headers

@**ModelAttribute**

An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

@SessionAttributes

@**SessionAttributes** spring annotation declares session attributes. This will typically list the names of model attributes which should be transparently stored in the session, serving as form-backing beans between subsequent requests.

@**SessionAttribute** works as follows:

is initialized when you put the corresponding attribute into model (either explicitly or using @ModelAttribute-annotated method).

is updated by the data from HTTP parameters when controller method with the corresponding model attribute in its signature is invoked.

@**SessionAttributes** are cleared when you call setComplete () on SessionStatus object passed into controller method as an argument.

@**PreAuthorize**

Using Spring Security @PreAuthorize annotation, you can authorize or deny functionality. In our example below, only a user with Admin role has the access to delete a contact.

**What is JdbcTemplate in spring?**

**org.springframework.jdbc.core.Jdbc. JdbcTemplate**

The **JdbcTemplate** class 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 such as 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.

The **NamedParameterJdbcTemplate** class helps you specify the named parameters instead of classic placeholder('?') argument. Named parameters improve readability and are easier to maintain.  
  
The **NamedParameterJdbcTemplate** provide better approach than [JdbcTemplate](http://www.dineshonjava.com/2012/12/using-jdbctemplate-in-spring-chapter-33.html#.UMNSg4aeWk9) ,where multiple parameters are in use for an SQL statement. It eliminated need of traditional JDBC "?" and provide named parameters. It is easy to use and provide better documentation. It functionality is similar to [JdbcTemplate](http://www.dineshonjava.com/2012/12/using-jdbctemplate-in-spring-chapter-33.html#.UMNSg4aeWk9) except it incorporate named parameters instead of "?" placeholder.   
  
In [**JdbcTemplate**](http://www.dineshonjava.com/2012/12/using-jdbctemplate-in-spring-chapter-33.html#.UMNSg4aeWk9), SQL parameters are represented by a special placeholder “?” symbol and bind it by position. The problem is whenever the order of parameter is changed, you have to change the parameters bindings as well, it’s error prone and cumbersome to maintain it.

<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/java2novice>" />

<property name="username" value="user\_name" />

<property name="password" value="password" />

</bean>

<bean id="employeeDAO" class="com.java2novice.dao.EmployeeDaoImpl">

<property name="dataSource" ref="dataSource" />

</bean>

**What is Dependency Injection in spring?**

Dependency Injection is an aspect of inversion of control is a general concept, and it can be expressed in many different ways. This concept says that you do not create your object but describe how they should be created. You don’t directly connect your components and services together in code but describe which services are needed by which components in a spring configuration file. A container (the IOC container) is then responsible for hooking it all up.

Java Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application **loosely coupled**, **extendable** and **maintainable**. We can implement dependency injection in java to move the dependency resolution from compile-time to runtime.

To use dependency injection pattern to achieve loose coupling and extendibility in the application

.

Whether to use Constructor based dependency injection or setter based is a design decision and depends on your requirements. For example, if my application can’t work at all without the service class then I would prefer constructor based DI or else I would go for setter method based DI to use it only when it’s really needed.

Dependency Injection in Java is a way to achieve Inversion of control (IoC) in our application by moving objects binding from compile time to runtime. We can achieve IoC through **Factory Pattern, Template Method Design Pattern, Strategy Pattern and Service Locator pattern** too.

Spring Dependency Injection, Google Guice and Java EE CDI frameworks facilitate the process of dependency injection through use of Java Reflection API and java annotations. All we need is to annotate the field, constructor or setter method and configure them in configuration xml files or classes.

**Benefits of Java Dependency Injection?**

Some of the benefits of using Dependency Injection in Java are:

Separation of Concerns Boilerplate Code reduction in application classes because all work to initialize dependencies is handled by the injector component Configurable components makes application easily extendable

Unit testing is easy with mock object.

**Disadvantages of Java Dependency Injection?**

**Java Dependency injection has some disadvantages too:**

If overused, it can lead to maintenance issues because effect of changes is known at runtime. Dependency injection in java hides the service class dependencies that can lead to runtime errors that would have been caught at compile time.

**What are the different types of IOC (dependency injection)?**

**Constructor based DI** is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

**Setter-based dependency injection**: Setter based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

**Aspect Oriented Programming Core Concepts?**

Most of the enterprise applications have some common crosscutting concerns that is applicable for different types of Objects and modules. Some of the common crosscutting concerns are **logging**, **transaction management**, **data validation** etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in Aspect Oriented Programming application modularity is achieved by Aspects and they are configured to cut across different classes.

Spring AOP takes out the direct dependency of crosscutting tasks from classes that we can’t achieve through normal object oriented programming model. For example, we can have a separate class for logging but again the functional classes will have to call these methods to achieve logging across the application.

**Aspect**: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management. Aspects can be a normal class configured through Spring XML configuration or we can use Spring **AspectJ** integration to define a class as Aspect using @Aspect annotation.

**<!-- Enable AspectJ style of Spring AOP -->**

**<aop:aspectj-autoproxy />**

**Join Point**: A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.

**Advice**: Advices are actions taken for a particular join point. In terms of programming, they are methods that gets executed when a certain join point with matching pointcut is reached in the application. You can think of Advices as [Struts2 interceptors](http://www.journaldev.com/2210/struts-2-interceptor-example) or [Servlet Filters](http://www.journaldev.com/1933/java-servlet-filter-example-tutorial).

**Before advice** - @Aspect , @**Before**

**These advices runs before the execution of join point methods. We can use @Before annotation to mark an advice type as Before advice.**

**After advice** - @**After**

**An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using @After annotation.**

**After returning advice** - @**AfterRunning**

**Sometimes we want advice methods to execute only if the join point method executes normally. We can use @AfterReturning annotation to mark a method as after returning advice.**

**After throwing advice** - @**AfterThrowing**

**This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively.We use @AfterThrowing annotation for this type of advice.**

**Around advice - @Around**

**This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something. We use @Around annotation to create around advice methods.**

org.aspectj.lang.annotation.Aspect

org.aspectj.lang.annotation.Before

org.aspectj.lang.annotation.After

org.aspectj.lang.annotation.AfterReturning

org.aspectj.lang.annotation.AfterThrowing

org.aspectj.lang.annotation.Around

**Pointcut**: **Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression language.**

**@Before("execution(public String getName())")**

**public void getNameAdvice(){**

**System.out.println("Executing Advice on getName()");**

**}**

**Target Object**: **They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object. What is means is that a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.**

**AOP proxy**: **Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations; these are called AOP proxy classes. We can also use CGLIB proxy by adding it as the dependency in the Spring AOP project.**

**Weaving**: **It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time**

**or at runtime. Spring AOP performs weaving at the runtime**.

**What is the difference between concern and cross-cutting concern in Spring AOP?**

Concern **is behavior which we want to have in a module of an application. Concern may be defined as a functionality we want to implement to solve a specific business problem. E.g. in any ecommerce application different concerns (or modules) may be inventory management, shipping management, user management etc.**

**Cross-cutting concern is a concern which is applicable throughout the application (or more than one module). e.g. logging , security and data transfer are the concerns which are needed in almost every module of an application, hence they are termed as cross-cutting concerns.**

**What is load-on-startup**?

**As stated earlier load-on-startup is a tag element which appear inside <servlet> tag in web.xml. load-on-startup tells the web container about loading of a particular servlet. if you don't specify load-on-startup then container will load a particular servlet when it feels necessary most likely when first request for that servlet will come, this may lead to longer response time for that query if Servlet is making**[**database connections**](http://javarevisited.blogspot.com/2011/11/database-transaction-tutorial-example.html)**or performing**[**ldap authentication**](http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html)**which contribute network latency or any other time consuming job, to avoid this, web container provides you a mean to specify certain servlet to be loaded during deployment time of application by using load-on-startup parameter.**

**If you specify load-on-startup parameter inside a servlet than based upon its value Container will load it.you can specify any value to this element but in case of load-on-startup>0 , servlet with less number will be loaded first. For example in below web.xml**AuthenticationServlet**will be loaded before**AuthorizationServlet**because load-on-startup value for AuthenticationServlet is less (2) while for AuthorizationServlet is 4.**

**load-on-startup Example in web.xml**

**here is an example of how to use load on startup tag inside servlet element in web.xml:**

<servlet>

**<servlet-name>AuthenticationServlet</servlet-name>**

**<display-name>AuthenticationServlet</display-name>**

**<servlet-class>com.trading.AuthenticationServlet</servlet-class>**

<load-on-startup>2</load-on-startup>

</servlet>

<servlet>

**<servlet-name>AuthorizationServlet</servlet-name>**

**<display-name>AuthorizationServlet</display-name>**

**<servlet-class>com.trading.AuthorizationServlet</servlet-class>**

<load-on-startup>4</load-on-startup>

</servlet>

Important points on load-on-startup element

**1. If <load-on-startup> value is same for two servlet than they will be loaded in an order on which they are declared inside web.xml file.**

**2. if <load-on-startup> is 0 or negative integer than Servlet will be loaded when Container feels to load them.**

**3. <load-on-startup> guarantees loading, initialization and call to init() method of servlet by web container.**

**4. If there is no <load-on-startup> element for any servlet than they will be loaded when web container decides to load them.**

**When to use <load-on-startup> in web.xml?**

**<load-on-startup> is suitable for those servlet which performs time consuming jobs e.g. Creating Database Connection pool, downloading files or data from network or prepare environment ready for servicing client in terms of initializing cache , clearing pipelines and loading important data in memory. If any of your servlet performs these jobs then declare them using <load-on-startup> element and specify order as per your business logic or what suites your application.**Remember lower the value of <load-on-startup>, servlet will be loaded first. **You can also check your web container documentation on how exactly load on start-up is supported.**

@RequestMapping is one of the most widely used Spring

MVC annotation. org.springframework.web.bind.annotation.RequestMapping annotation is used to map web requests onto specific handler classes and/or handler methods.

Spring @RequestMapping

@RequestMapping with Class: We can use it with class definition to create the base URI. For example:

@Controller

@RequestMapping("/home")

public class HomeController {

}

Now /home is the URI for which this controller will be used. This concept is very similar to servlet context of a web application.

@RequestMapping with Method: We can use it with method to provide the URI pattern for which handler method will be used. For example:

@RequestMapping(value="/method0")

@ResponseBody

public String method0(){

return "method0";

}

Above annotation can also be written as @RequestMapping("/method0"). On a side note, I am using @ResponseBody to send the String response for this web request, this is done to keep the example simple. Like I always do, I will use these methods in Spring MVC application and test them with a simple program or script.

@RequestMapping with Multiple URI: We can use a single method for handling multiple URIs, for example:

@RequestMapping(value={"/method1","/method1/second"})

@ResponseBody

public String method1(){

return "method1";

}

If you will look at the source code of RequestMapping annotation, you will see that all of it’s variables are arrays. We can create String array for the URI mappings for the handler method.

@RequestMapping with HTTP Method: Sometimes we want to perform different operations based on the HTTP method used, even though request URI remains same. We can use @RequestMapping method variable to narrow down the HTTP methods for which this method will be invoked. For example:

@RequestMapping(value="/method2", method=RequestMethod.POST)

@ResponseBody

public String method2(){

return "method2";

}

@RequestMapping(value="/method3", method={RequestMethod.POST,RequestMethod.GET})

@ResponseBody

public String method3(){

return "method3";

}

@RequestMapping with Headers: We can specify the headers that should be present to invoke the handler method. For example:

@RequestMapping(value="/method4", headers="name=pankaj")

@ResponseBody

public String method4(){

return "method4";

}

@RequestMapping(value="/method5", headers={"name=pankaj", "id=1"})

@ResponseBody

public String method5(){

return "method5";

}

@RequestMapping with Produces and Consumes: We can use header Content-Type and Accept to find out request contents and what is the mime message it wants in response. For clarity, @RequestMapping provides produces and consumes variables where we can specify the request content-type for which method will be invoked and the response content type. For example:

@RequestMapping(value="/method6", produces={"application/json","application/xml"}, consumes="text/html")

@ResponseBody

public String method6(){

return "method6";

}

Above method can consume message only with Content-Type as text/html and is able to produce messages of type application/json and application/xml.

Spring @PathVariable

@RequestMapping with @PathVariable: RequestMapping annotation can be used to handle dynamic URIs where one or more of the URI value works as a parameter. We can even specify [Regular Expression](http://www.journaldev.com/634/regular-expression-in-java-regex-example) for URI dynamic parameter to accept only specific type of input. It works with @PathVariable annotation through which we can map the URI variable to one of the method arguments. For example:

@RequestMapping(value="/method7/{id}")

@ResponseBody

public String method7(@PathVariable("id") int id){

return "method7 with id="+id;

}

@RequestMapping(value="/method8/{id:[\\d]+}/{name}")

@ResponseBody

public String method8(@PathVariable("id") long id, @PathVariable("name") String name){

return "method8 with id= "+id+" and name="+name;

}

Spring @RequestParam

@RequestMapping with @RequestParam for URL parameters: Sometimes we get parameters in the request URL, mostly in GET requests. We can use @RequestMapping with @RequestParam annotation to retrieve the URL parameter and map it to the method argument. For example:

@RequestMapping(value="/method9")

@ResponseBody

public String method9(@RequestParam("id") int id){

return "method9 with id= "+id;

}

For this method to work, the parameter name should be “id” and it should be of type int.

@RequestMapping default method: If value is empty for a method, it works as default method for the controller class. For example:

@RequestMapping()

@ResponseBody

public String defaultMethod(){

return "default method";

}

As you have seen above that we have mapped /home to HomeController, this method will be used for the default URI requests.

@RequestMapping fallback method: We can create a fallback method for the controller class to make sure we are catching all the client requests even though there are no matching handler methods. It is useful in sending custom 404 response pages to users when there are no handler methods for the request.

@RequestMapping("\*")

@ResponseBody

public String fallbackMethod(){

return "fallback method";

}

Spring RestMapping Test Program

We can use [Spring RestTemplate](http://www.journaldev.com/2552/spring-rest-example-tutorial-spring-restful-web-services) to test the different methods above, but today I will use cURL commands to test these methods because these are simple and there are not much data flowing around.

What is the difference between Spring3.0 and Spring2.5?

**The Spring Framework is the widely used framework for java based application development. Spring framework 3.0 is the next major release after framework 2.5. The difference between Spring Framework3.0 and Spring Framework 2.5 are:**

|  |  |
| --- | --- |
| **Spring Framework3.0** | **Spring Framework2.5** |
| is compatible with java5 or higher versions | is compatible with java 2.4 and higher versions. |
| Introduces Spring Expression Language which defines bean definitions based XML and Annotation. | Native expression language exists which is less powerful than spring expression language of spring 3.0 and it has different parsing rules. |
| has type converting system and field formatting. | Does not support type conversions and field formatting. |
| Fully supports the JSR303 bean validation API. | Does not offer complete support for JSR303 bean validation API. |
| Offers support for various embedded database engines like HSQL, Derby and H2. | Does not support the embedded database engines. |
| Automatically validates the @Controller inputs. | Does not validate the @Controller inputs automatically. |
|  |  |

**New Features and Enhancements in Spring Framework 4.0**  
  
**1. Removed Deprecated Packages and Methods**

All deprecated packages, and many deprecated classes and methods have been removed with version 4.0. If you are upgrading from a previous release of Spring, you should ensure that you have fixed any deprecated calls that you were making to outdated APIs. This should be the challenging point for those who want to migrate their old spring applications to the new version. As a caution, if you are running the old spring application, then first thing to get rid of your deprecated APIs before migrating to the latest version.  
  
For a complete set of changes, [check out the API Differences Report](http://docs.spring.io/spring-framework/docs/3.2.4.RELEASE_to_4.0.0.RELEASE/).  
  
**2. Java 8 Support**  
Spring Framework 4.0 provides support for several Java 8 features. You can make use of lambda expressions and method references with Spring’s callback interfaces. There is first-class support for java.time (JSR-310), and several existing annotations have been retrofitted as @Repeatable. You can also use Java 8’s parameter name discovery (based on the -parameters compiler flag) as an alternative to compiling your code with debug information enabled.  
**Some of the Java SE 8 features to be supported include:**

* JSR-335 Lambda expressions.
* JSR-310 Date-Time value types for Spring data binding and formatting.
* Support for the new 1.8 byte code format (required to support Lambda expressions).

Spring 4.0 has increased the minimum recommendation to Java 6.0. Also it states that any new project recommended to use the Java 7.0 for their projects with Spring 4.0. It still supports the lower version Java 6 and 7 without any problem.  
  
**3. Java EE 6 and 7**  
Java EE version 6 or above is now considered the baseline for Spring Framework 4, with the JPA 2.0 and Servlet 3.0 specifications being of particular relevance. It is possible to run your application in Servlet 2.5, but it is recommended to use Servlet 3.0 environment.  
  
**4. Groovy Bean Definition DSL**  
With Spring Framework 4.0 it is now possible to define external bean configuration using a Groovy DSL. [Read more about this API](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/groovy/GroovyBeanDefinitionReader.html).  
  
**5. Core Container Improvements**  
There have been several general improvements to the core container:

* Spring now treats [generic types as a form of qualifier](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-generics-as-qualifiers) when injecting Beans. For example, if you are using a Spring Data Repository you can now easily inject a specific implementation: @Autowired Repository<Customer> customerRepository.
* If you use Spring’s meta-annotation support, you can now develop custom annotations that [expose specific attributes from the source annotation](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-meta-annotations).
* Beans can now be Ordered when they are [autowired intolists and arrays](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-autowired-annotation). Both the @Ordered annotation and Ordered interface are supported.
* The @Lazy annotation can now be used on injection points, as well as @Bean definitions.
* The [@Description annotation has been added](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-java-bean-description) for developers using Java-based configuration.
* A generalized model for [conditionally filtering beans](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-java-conditional) has been added via the @Conditionalannotation. This is similar to @Profile but allows for user-defined strategies to be developed.
* [CGLIB-based proxy classes](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#aop-pfb-proxy-types) no longer require a default constructor. Support is provided via the [objenesis](http://code.google.com/p/objenesis/) library which is repackaged *inline* and distributed as part of the Spring Framework. With this strategy, no constructor at all is being invoked for proxy instances anymore.
* There is managed time zone support across the framework now, e.g. on LocaleContext.

**6. General Web Improvements**  
Deployment to Servlet 2.5 servers remains an option, but Spring Framework 4.0 is now focused primarily on Servlet 3.0+ environments. If you are using the Spring MVC Test Framework you will need to ensure that a Servlet 3.0 compatible JAR is in your test classpath.  
  
In addition to the WebSocket support mentioned later, the following general improvements have been made to Spring’s Web modules:

* You can use the new ***@RestController*** annotation with Spring MVC applications, removing the need to add ***@ResponseBody*** to each of your ***@RequestMapping*** methods.
* The ***AsyncRestTemplate***class has been added, allowing non-blocking asynchronous support when developing REST clients.
* Spring now offers comprehensive timezone support when developing Spring MVC applications.

**7. WebSocket, SockJS, and STOMP Messaging**  
A new spring-websocket module provides comprehensive support for WebSocket-based, two-way communication between client and server in web applications. It is compatible with JSR-356, the Java WebSocket API, and in addition provides SockJS-based fallback options (i.e. WebSocket emulation) for use in browsers that don’t yet support the WebSocket protocol (e.g. Internet Explorer < 10).  
  
A new spring-messaging module adds support for STOMP as the ***WebSocket***sub-protocol to use in applications along with an annotation programming model for routing and processing STOMP messages from ***WebSocket***clients. As a result an ***@Controller*** can now contain both ***@RequestMapping*** and ***@MessageMapping*** methods for handling HTTP requests and messages from WebSocket-connected clients. The new spring-messaging module also contains key abstractions from the Spring Integration project such as ***Message, MessageChannel, MessageHandler***, and others to serve as a foundation for messaging-based applications.  
  
**8. Testing Improvements**  
In addition to pruning of deprecated code within the spring-test module, Spring Framework 4.0 introduces several new features for use in unit and integration testing.

* Almost all annotations in the spring-test module (e.g., *@ContextConfiguration, @WebAppConfiguration, @ContextHierarchy, @ActiveProfiles, etc.*) can now be used as meta-annotations to create custom composed annotations and reduce configuration duplication across a test suite.
* Active bean definition profiles can now be resolved programmatically, simply by implementing a custom ***ActiveProfilesResolver***and registering it via the resolver attribute of *@ActiveProfiles*.
* A new ***SocketUtils***class has been introduced in the spring-core module which enables you to scan for free TCP and UDP server ports on localhost. This functionality is not specific to testing but can prove very useful when writing integration tests that require the use of sockets, for example tests that start an in-memory SMTP server, FTP server, Servlet container, etc.
* As of Spring 4.0, the set of mocks in the *org.springframework.mock.web* package is now based on the Servlet 3.0 API. Furthermore, several of the Servlet API mocks (e.g., **MockHttpServletRequest**, **MockServletContext**, etc.) have been updated with minor enhancements and improved configurability.

#### XML shortcut with the p-namespace

The p-namespace enables you to use the bean element’s attributes, instead of nested <property/> elements, to describe your property values and/or collaborating beans.

Spring supports extensible configuration formats [with namespaces](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#xsd-configuration), which are based on an XML Schema definition. The beans configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

The following example shows two XML snippets that resolve to the same result: The first uses standard XML format and the second uses the p-namespace.

<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="classic" class="com.example.ExampleBean">

<property name="email" value="foo@bar.com"/>

</bean>

<bean name="p-namespace" class="com.example.ExampleBean"

p:email="foo@bar.com"/>

</beans>

The example shows an attribute in the p-namespace called email in the bean definition. This tells Spring to include a property declaration. As previously mentioned, the p-namespace does not have a schema definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

<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="john-classic" class="com.example.Person">

<property name="name" value="John Doe"/>

<property name="spouse" ref="jane"/>

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person">

<property name="name" value="Jane Doe"/>

</bean>

</beans>

As you can see, this example includes not only a property value using the p-namespace, but also uses a special format to declare property references. Whereas the first bean definition uses <property name="spouse" ref="jane"/> to create a reference from bean john to bean jane, the second bean definition uses p:spouse-ref="jane" as an attribute to do the exact same thing. In this case spouse is the property name, whereas the -ref part indicates that this is not a straight value but rather a reference to another bean.

|  |
| --- |
| [Note] |
| The p-namespace is not as flexible as the standard XML format. For example, the format for declaring property references clashes with properties that end in Ref, whereas the standard XML format does not. We recommend that you choose your approach carefully and communicate this to your team members, to avoid producing XML documents that use all three approaches at the same time. |

#### XML shortcut with the c-namespace

Similar to the [the section called “XML shortcut with the p-namespace”](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-p-namespace), the c-namespace, newly introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments rather then nested constructor-arg elements.

Let’s review the examples from [the section called “Constructor-based dependency injection”](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-constructor-injection) with the c: namespace:

<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 id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

*<!-- traditional declaration -->*

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

<constructor-arg value="foo@bar.com"/>

</bean>

*<!-- c-namespace declaration -->*

<bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com"/>

</beans>

The c: namespace uses the same conventions as the p: one (trailing -ref for bean references) for setting the constructor arguments by their names. And just as well, it needs to be declared even though it is not defined in an XSD schema (but it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was compiled without debugging information), one can use fallback to the argument indexes:

*<!-- c-namespace index declaration -->*

<bean id="foo" class="x.y.Foo" c:\_0-ref="bar" c:\_1-ref="baz"/>

|  |
| --- |
| [Note] |
| Due to the XML grammar, the index notation requires the presence of the leading \_ as XML attribute names cannot start with a number (even though some IDE allow it). |

In practice, the constructor resolution [mechanism](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-factory-ctor-arguments-resolution) is quite efficient in matching arguments so unless one really needs to, we recommend using the name notation through-out your configuration.