IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-2, Issue-8 E-ISSN: 2347-2693

High Performance Spring Programming

Vamsi Krishna Myalapalli

Open Text Corporation, Mind Space IT Park, Hi-Tec City, Hyderabad, India

www.ijcaonline.org

Received: 02/07/ 2014 Revised: 22/07/ 2014 Accepted: 17/08/ 2014 Published: 31 /08/ 2014

***Abstract*—**In the contemporary world Spring Application Framework is the most prevalently used application framework due to its IoC (Inversion of Control) property. Many of the Spring developers simply do programming with less concern towards optimized processing. Though Spring Framework offers sundry ways of programming techniques to reach the same end, certain practices are pre requisite to ensure that consequences will be prolific. This paper proposes miscellaneous, simple, reliable, flexible and easy techniques to make programs more efficient. The exploration in this paper would serve as a benchmarking tool for assessing best programming practices. Experimental results of analysis designate that maintainability, flexibility and reusability are enhanced.

***Keywords*—**Spring Best Practices; Spring Tuning; Spring Tactics; Spring Core; Spring Framework Tactics; Efficient Spring Practices.

I. INTRODUCTION

The Spring Framework is an open source application framework created to simplify the development of enterprise Java software. The framework achieves this goal by providing developers with a component model and a set of simplified and consistent APIs that effectively insulate developers from the complexity and error-prone boilerplate code required to create complex applications.

The Core of the Spring Framework is its IoC (Inversion of Control) container, which offers a reliable means of configuring and managing Java objects using reflection. The container takes responsibility of managing object lifecycles of specific objects i.e. creating objects, calling their initialization methods and configuring these objects by wiring them together.

Dependency lookup or DI (Dependency Injection) is the means through which Objects can be acquired.

This paper exposes proactive best practices or tactics that the programmers should glean at.

II. BACKGROUND AND RELATED WORK

Spring advocates that not to create objects using 'new' operator i.e. objects will be created explicitly from the class in a configuration file passively. (IoC technique externalizes the creation and management of component dependencies). POJO (Plain Old Java Objects) and POJI (Plain Old Java Interfaces) makes Spring Framework as light weight.

The following are several benefits offered by DI (Dependency Injection) over traditional programming approach:

a) Coupling among modules is reduced.
b) Reduced Code.
c) Simplified Application Configuration.
d) Ability to manage common dependencies in a single repository.
e) Enhanced Testability.
f) Fosters good Application Design.

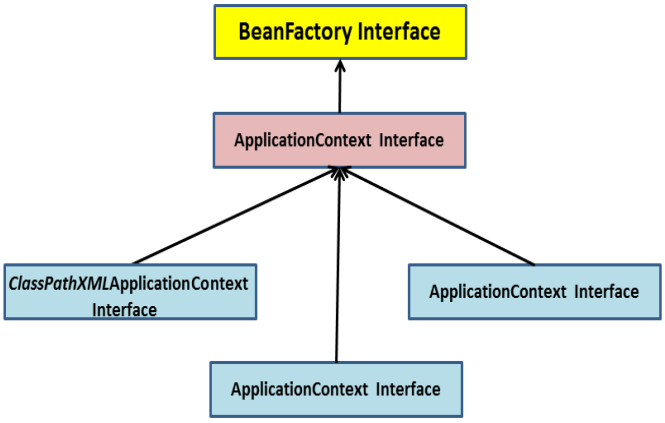

Figure 2.1: Hierarchy of Interfaces in Spring

The above portrait depicts the Hierarchy of Interfaces in Spring Framework.

The Spring Framework features its own MVC (Model View Controller) web application framework.

Spring supports classes for writing unit tests and integration tests through its Testing module, and enables implementing cross-cutting concerns through Aspect Oriented Programming.

III. PROPOSED BENCHMARK

This paper brings out several best programming practices in Spring Framework and serves as a Benchmarking tool. Though there exists several tactics some best practices are explained here and their corresponding programmatic implementation is explained in Section 4.

Corresponding Author: *Vamsi, vamsikrishna.vasu@gmail.com*

1) ApplicationContext vs. BeanFactory interfaces: Prefer ApplicationContext over BeanFactory. The ApplicationContext is Spring's main object registry and integration point. It is usually configured via an XML file in which beans and their dependencies are declared. The ApplicationContext has many features, but its central role is object creation and Dependency Injection. In most cases, the ApplicationContext will be a transparent piece of our applications, freeing our application logic from Spring-specific integration.

BeanFactory provides basic functionality while ApplicationContext provides advance features to our spring applications which make them enterprise level applications, like i18n, event publishing, JNDI access, EJB integration, Remoting etc.

BeanFactory serves only as backward compatibility in existing systems.

2) Every Configuration file with Header: Represent a commenting header for every configuration file. This helps in summarizing the beans or properties defined in Configuration file(s). Commenting can be implemented either with XML or using <description> tag.

3) Minimize Dependency: Minimize direct dependency using the concept of Dependency Injection (DI). This ensures minimizing the rate of coupling among classes.

Example: Using the object of one class in another class is a kind of direct dependency.

4) Implementing Naming Conventions Consistently: Naming conventions should be consistent across every configuration file(s). Implementing consistent, clear and descriptive naming convention(s) enhances readability of configuration files and allows rest of the developers to prevent abrupt bugs.

5) Sparingly represent Version in Schema References: If versions are represented at higher rate, it necessitates maintenance. Since Spring automatically chooses the recent version from project dependencies (jars) the version specification should be kept to minimum. On the other hand as the project evolves the Spring version will be updated, so we need not maintain every XML configuration file to look into new features.

6) Choose Setter Injection method against Constructor Injection: Among all Dependency Injections (Constructor Injection, Setter Injection, Method Injection) Setter Injection allows higher maintainability and flexibility. On the other hand Constructor Injection provides less thread safety and advocates that object will not be handed over to other beans without complete initialization.

Use Constructor Injection to populate mandatory dependencies, and Setter Injection to populate optional dependencies.

7) Type vs. Index for Constructor Argument matching in Constructor Injection: As per the previous tactic Constructor injection should be deterred. On the occasion if there is extreme requirement to use Constructor Injection pick up parameter matching based on type instead of index. Type based argument exhibits higher readability and are less error prone.

8) Implement Shortcut forms for Expanded Forms: Choose Short form(s) over Expanded form(s) for better readability in XML files etc.

9) Bean Definition Reuse: Reuse bean definition(s) to full extent inorder to ensure higher functional cohesion. Example: Bean definition reuse for setter injection and constructor injection.

10) Use ID as Bean Identifiers: Though Spring allows recognizing beans using 'id' or by 'name', we should pick up a bean with its 'id' instead of 'name'. This is due to the reason that 'id' would be unique every time, whereas the 'name' might not.

11) Deter Autowiring: The Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements which helps cut down on the amount of XML configuration you write for a big Spring based application.

Though Autowiring offers lot of benefits, it has its downside. As the size of project increases it triggers trouble in recognizing precise dependency to use. Apart from this Autowiring make the process of debugging harder.

Merits of Auto wiring:

a) Is less verbose than explicit configuration.
b) Keeps itself up to date.
   Example: If we add more properties, they will automatically be satisfied without the need to update bean definition(s) of that class.

Demerits of Auto wiring:

a) Is less self-documenting than explicit wiring.
b) Cannot be used for simple configuration properties, rather than object dependencies. But, we can still explicitly set configuration properties on auto wired beans.

The container's behavior at runtime isn't described in our configuration, merely implied. However, Spring's policy of raising a fatal error on any ambiguity means that unwanted effects are unlikely.

12) Make use of Classpath Prefix: Use classpath as prefix if XML config, resources or properties are imported. This leads to clarity and consistency of resource location. The IDE and Build tool determines the class path and is as follows

src/main/java and src/test/java for Java code.

src/main/resources for Non-Java Dependencies and for Tests.
src/test/resources for Non-Java Resources.

13) Externalize the Properties: Never hard code the values in configuration file(s), rather externalize them onto property file(s). This practice prevents disturbing the actual code for modifications. It is highly appreciated if properties file(s) are grouped depending on its module or usage. Example: All Java Database Connectivity related grouped in 'jdbc.properties' file.

14) Enforce Dependency check at Development Phase: The attribute 'dependency-check' on bean definition should be set to 'simple', 'objects' or 'all' (default is 'none') in order to ensure that container perform explicit dependency validation. This behavior is beneficial if all or some categories of properties of a bean should be set explicitly or through Auto wiring.

15) Singleton vs. Prototype Scope: Singleton beans are not thread-safe in Spring framework. Absence of locks in getSingleton method makes Singleton bean thread unsafe. Locking makes sure that only one process goes on at a time irrespective of the circumstance.
Example for unsafe threading:

Thread 1: Instantiates bean "X". Adds "X" to the singleton Objects map even before completely resolving all the dependencies of that bean.
Thread 2: Asks for bean "A" and till this time Thread 1 has already added it into singleton Objects map but Thread 1 is still resolving dependencies.

Prototype beans incur hit on performance during creation. It should be avoided completely or designed carefully if it uses resources such as database or network connections. They are useful for factory to create new bean instances.
But now Thread 2 recognizes an instance of "A" which is not completely initialized.

16) Enforce Single XML File: Use single XML configuration file to bootstrap the application or tests.

17) Deem Inner Beans: Inner beans are beans that are defined within the scope of another bean. Use Inner beans if a bean is to be created/exist without its parent.

18) Isolate Deployment from Application Details: Isolate Application Context from Deployment details.

19) Nulling a Property: Use </null> to set any property to null value.

20) AOP vs. AspectJ: Prefer Spring AOP (Runtime Weaving) or AspectJ (Compile time weaving) based on requirement. Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.

Spring AOP is simpler than using AspectJ as there is no requirement to introduce the compiler into build processes for Compile Time Weaving.

21) Smart Logging: To ensure smart logging enforce the following

a) Avoid using System.out
b) Avoid using System.err
c) Always use SLF4J
d) Always use Logback
e) Prohibit Apache Commons Logging also referred as Jakarta Commons Logging
f) Prohibit Java Util Logging (JUL)

22) Deter the Use of Component Scanning: Spring defines a set of stereotype annotations, which are markers for any class that fulfills a role within an application. For example, the @Repository annotation, which was introduced in Spring 2.0, is used for classes that fulfill the role of Data Access Object of a repository. Spring provides an option to automatically detect stereotyped classes and register corresponding Bean definitions with the ApplicationContext, eliminating the need to specify the stereotyped classes in the XML configuration metadata file. This is achieved by implicitly detecting the candidate components by scanning the class path and matching against filters.

When component scanning is enabled, during the application initialization phase Spring will have to read a large amount of data from the file system in real-time to scan for the stereotyped classes on the class path. This can cause the initial request to an App Engine application to take an in deterministic amount of time to complete, despite the fact that the App Engine's virtual file system is a very high performance system.

23) Breaking Larger Jar Files: Breaking up larger JAR files can improve performance of class path scanning. This is because the application does not have as many classes to scan if Spring can determine which JARS are relevant.

24) Class path Scanning: It is not advisable to attempt to perform class path scanning with UberJars or any other form of JAR files generated from "JAR composing" build tools.

25) Avoid huge Up loadings: Try to avoid uploading a large amount of .class files; attempt to "JAR up" these .class files into smaller JAR files that will be compressed and therefore faster to load.

26) Disabling XML Validation in Production: To further reduce the loading time of an application, we can disable XML validation in production.

27) Avoiding Constructor Injection by Name: Spring supports using the constructor parameter name for value

disambiguation. Let's consider the following sample class as an example:

```
public class Movie
  {
    private String name;
    private String description;
    public Movie(String name, String description)
      {
        this.name = name;
        this.description = description;
      }
  }
```

The corresponding configuration using injection by name is shown as below:

```
<bean id="movie" class="example.Movie">
<constructor-arg name="name" value="ET" />
<constructor-arg name="synopsis" value="ItsMe
"/>
</bean>
```

To make this work "out of the box", Spring requires that the code must be compiled with the debug flag enabled (-g for all debugging info, or -g:vars to be precise, for the local variable debug information). This allows Spring to look up the parameter name in the debug information. However, since this information is not cached in the JVM, it must be loaded from disk which causes significant I/O time penalty.
To solve this problem we can use one of the following guidelines:
a) Use the @ConstructorProperties annotation to explicitly name your constructor arguments as shown next.

```
public class Movie
  {
  // Fields omitted
  @ConstructorProperties({"name", "description"})
    public Movie(String name, String description)
      {
        this.name = name;
        this.description = description;
      }
  }
```

b) Define the bean without using constructor injection by name:

```
<bean id="movie" class="example.Movie">
<constructor-arg  index=0 value="ET" />
<constructor-arg index=1 value="Help ET go home" />
</bean>
```

However, constructor injection by name is a bad programming practice. It is an anti-pattern of dependency injection which relies on the consistent naming of constructor arguments. Here are the potential problems that may lead to problems that are difficult to debug.
a) When utilizing software that we do not control, the effective definition of the interface itself is extended to include information that is not part of the standard method signature in Java. Therefore, it is entirely possible that someone could change the name of a constructor argument without revising the major or minor versions of the built artifact, causing your application to fail.
b) Constructor injection can become problematic with any framework or system that rewrites the class byte code using something similar to ASM. In Java, it is entirely possible to obtain the byte code of a specific class and modify it to be reloaded by a class loader. Doing so can break linkage of the subsequently defined class with the source it was originally defined with. This means the lookup of the names of the constructor arguments will fail and this again may break the application unpredictably.

28) Use "depends" if that Bean depends on any other bean: If any bean is depended on other bean like any static variable is using or something then we must specify that bean inside the bean definition using the keyword "depends". This helps Spring to initialize that bean before the initialization of the actual bean.

29) Aspect Oriented Programming in the necessary situations: Use the ability of Aspect Oriented Programming in the necessary times.

30) Every bean must have an id: Always use id to specify the default name of the bean. If you need characters in your bean name that are not allowed in the id attribute, then you can provide additional names with the name attribute.

31) Declarative Caching services for Spring: Declarative Caching Services for Spring provides declarative caching for Spring-powered applications. Declarative caching does not involve any programming and therefore it is a much easier and more rapid way of applying and tuning caching services. Configuration of caching services can be completely done in the Spring IoC container. Spring provides support for different cache providers such as EHCache, JBoss Cache, Java Caching System (JCS), OSCache, and Tangosol Coherence.

32) Configuring Spring Application context for different locations: If the configuration is the same for all the environments except for the developer's machine, then make (a) separate configuration file(s) with the configuration that is different. Let this different configuration overwrite the definition(s) in the original file(s).Make sure this different configuration will never be placed in the other environments. (b) Beans defined in the configuration locations (first constructor-arg) overwrite the beans in the parent application context (second constructor-arg).

33) Prefer static point cut over dynamic point cut: Static point cut refers to a point cut that can be evaluated when a proxy is created. Criteria for static point cuts cannot be changed afterwards. Dynamic point cuts depend on runtime information such as argument values or call stack. Dynamic point cuts are slower to evaluate than static point cuts and allow less potential for optimization. It's always necessary to evaluate them on each invocation.

34) Use regular expression advisors to fine tune interceptor scope: Instead of using broad method point cuts and filtering target methods in interceptor, use more sophisticated and elegant regular expression at the application context level.

35) Use Auto proxying for large applications: ProxyFactoryBean works well for small application but it requires more verbose configuration. It allows control over every aspect of the proxy. Spring ease the use of ProxyFactoryBean by providing dedicated proxies such as TransactionProxyFactoryBean and LocalStatelessSessionProxyFactoryBean. Proxies also prevent code duplication in configurations.

36) Aware of thread safe issues with AOP advice: Advice instances are most often shared among threads, so we need to consider thread safety issues. For example if the method interceptor is responsible for generating unique id or count, then consider using ThreadLocal variable with synchronized method for incrementing the count.

37) Prefer to use apache Connection pooling bean: In the spring DataSource Configuration we are commonly using class is "org. springframework. jdbc. datasource. DriverManagerDataSource". We can implement connection pooling using the class "org.apache.commons.dbcp.BasicDataSource". In order to implement this, the necessary jars should be installed.

38) Handling Exceptions: Spring Gives a consistent exception hierarchy in its DAO level. All the SQL as well as DAO based exceptions are under the DataAccessException. With the effective handling of this exception, we can easily log the errors as well as we can more effectively assign "ERROR MESSAGES" to the error objects. We can also use Spring's "org.springframework.web.servlet.handler.SimpleMappingExceptionResolver" class to get the exceptions thrown and can be displayed in the web level. This will help us to check the "Exceptions in a real distributed environment".

39) Prefer to use Springs Declarative Transaction Capability: Spring provides Programmatic as well as Declarative Transaction capabilities. And if we are using any OR mapping tools then spring provides its own Transaction manager to handle it.

40) Transaction Attribute Settings: Spring provides a distinct "ISOLATION behaviors" for handling transactions. In which the most efficient isolation level, but isolates the transaction the least, leaving the transaction open to dirty, no repeatable, and phantom reads. At the other extreme, ISOLATION_SERIALIZABLE prevents all forms of isolation problems but least efficient. So it's better to choose according to our needs.

41) Perform unit testing in the DAO layer: The Data Access Part is the important part in which errors are popping up. So if we complete a unit test here then it will be more useful for our service layer programming. It's very easy to write unit tests in the spring DAO layers. Junit, easymock, unitils are some of the useful as well as mostly used Unit testing frameworks. Each one has its own advantages.

42) Prefer assembling bean definitions through ApplicationContext over imports: Spring imports elements that are useful for assembling modularized bean definitions. Using ApplicationContext makes the XML configurations easy to manage.

43) Communicate with team members for changes: When you are refactoring Java source code, you need to make sure to update the configuration files accordingly and notify team members. The XML configurations are still code, and they are critical parts of the application, but they are hard to read and maintain. Most of the time, we need to read both the XML configurations and Java source code to figure out what is going on.

44) Don't refer files directly: Do not reference any of the file(s) directly in the source code but rather use one single utility class responsible for returning references to different groups of the configuration files used in different places. Make sure that the methods returning these references uses proper signature return type like an String[] even if we have just one file for an specific domain or group of configuration files. Spring provides the possibility of composing a configuration file from multiple other files.

45) Commenting: Leave a one paragraph comment describing the purpose of the document rather than leaving it as it is. Do not leave room for next to come people to guess where to put the new bean definitions, etc. But rather specify it by leaving some comments if not documentation about the configuration file.

46) Designing Thread Safe Objects: We can make our Objects thread safe if we ensure the bean to be

a) Immutable
b) Stateless
c) Persistent
d) Lock enabled

47) IoC vs. Dependency Injection: Inversion of Control and Dependency Injection are used interchangeably, but in fact they are not the same. Inversion of Control is a much more general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control.

Inversion of Control (or IoC) covers a broad range of techniques that allow an object to become a passive participant in the system. When the IoC technique is applied, an object will relinquish control over some feature or aspect to the framework or environment. Some examples of control include the creation of objects or the delegation to dependent objects. IoC can remove these concerns from objects with

Dependency Injection and Aspect-Oriented programming, respectively.

48) Handling Mutable Objects: For objects that are not immutable, like Date, it's a best practice to make our own copies of the arguments before storing in the class or using with some business logic. By using this technique, we protect our class, as the client could change the internal value of the argument after passing in the object, potentially creating odd or inconsistent states.

49) Interface-driven Design: Interface-driven design is a traditional OOP best practice. When we use interface-driven design, the main components of our application are defined in terms of interfaces rather than concrete classes.

50) Handling Common Tests: For a group of common tests (e.g., test cases for controller classes, service layer testing classes, and so on), it's always a best practice to develop a common abstract parent class that has the mandatory testing infrastructure set up correctly.

51) Handling Action States: Extract reusable actions into standalone action states. An action state simply defines one or more actions to perform in a flow, and these actions will be performed in the declared order.

52) Autoproxy Infrastructure: If we need to advise a large number of objects, autoproxy infrastructure can produce significant simplification in application configuration.

53) Bean Definition "Inheritance": Use bean definition "inheritance" technique to eliminate duplication of interceptor chain definitions and other AOP configuration(s).

54) Avoid unnecessary dependence on Spring API: There should normally be no need to depend on Spring APIs for configuration. (Using Spring abstractions, such as data access APIs, is a different matter.) Part of the point of Dependency Injection is to minimize the need for dependency on any container. Becoming familiar with capabilities of Spring's IoC container helps us to minimize such dependencies, such as Method Injection.

55) Unit Testing (Testing Objects Individually): Unit testing should be done without any container, Spring or other, and without any further dependencies such as a database or JNDI environment.

56) Prevent Resource Leaks: Close the resources after they are used.

57) Lazy vs. Pre Loading: Lazy loading ensures that beans are loaded on fly, whereas Preloading ensures that all specified beans are loading even before they are used. Choose required methodology based on need.

Example: Preloading can be preferred on performance grounds i.e. on the occasion that there are less beans and application should run faster.
Note: Preloading can be achieved only with ApplicationContext and not with BeanFactory. But both support Lazy Loading.

58) Miscellaneous: Concern should be kept on following specious issues.

a) Bad coding
b) Not following standard
c) Not bothering about performance
d) History, Indentation, Comments are not appropriate.
e) Poor Readability
f) Open files are not closed
g) Allocated memory has not been released
h) Too many global variables.
i) Too much hard coding.
j) Poor error handling.
k) No modularity.
l) Repeated code

IV. EXPERIMENTAL SETUP

In this section some important practices which correspond to tactics in section 3 are explained via programs.

1. ApplicationContext context = new
ClassPathXmlApplicationContext("SpringConfig.xml");
Instead of
BeanFactory context = new
ClassPathXmlApplicationContext("SpringConfig.xml");
Where 'SpringConfig.xml' is the xml file holding bean configurations.

2.
```
<beans>
<description>
This configuration file will have all beans which control Database operations.
</description>
...
</beans>
```

3.
```
Class A
{
  B b = new B();
}
Class B
{
  …
}
```
In the above scenario, class A is directly/fully dependent on class B i.e. Coupling is higher.

6.
```
<!-- Setter injection -->
<bean id="employee"
class="com.opentext.myproject.Employee">
<property name="datasource" ref="datasource">
```

```
</bean>
```

Instead of

```
<!-- Constructor injection -->
<bean id="employee"
 class="com.opentext.myproject.Employee">
<constructor-arg ref="datasource"/>
</bean>
```

7.
```
<!-- Type based constructor injection -->
 <bean id="emp"
 class="com.opentext.Employee">
 <constructor-arg type="java.lang.String" value="rest"/>
 <constructor-arg type="int" value="8080"/>
 </bean>
```

Instead of

```
<!-- Index based constructor injection -->
<bean id="employee"
class="com.opentext.Employee">
 <constructor-arg index="0" value="rest"/>
<constructor-arg index="1" value="8080"/>
</bean>
```

8.
```
<!-- Shorter/shortcut version -->
<bean id="employeeDAO"
class="com.howtodoinjava.dao.EmployeeDAO">
<property name="datasource" ref="datasource"
value="datasource">
</bean>
```

Instead of

```
<!-- Expanded version -->
<bean id="employeeDAO"
class="com.howtodoinjava.dao.EmployeeDAO">
<property name="datasource">
<ref bean="datasource"></ref>
<value>datasource</value>
</property>
</bean>
```

10.
```
<bean id="restaurantBean"
        class="org.opentext.Springpractice.MyTest" >
```

11. Auto wiring may look like following
```
<bean id="myBean"
            class="com.opentext.spring.myTest"
                        autowire="byName"/>
```

12.
```
<!-- Always use classpath: prefix-->
<import resource = "classpath: /META-
    INF/spring/applicationContextsecurity.xml"/>
```

13. Content of 'Config.xml' file
```
<bean id="myBean"
        class="org.opentext.Springpractice.MyTest" >
<property name="Note" value="${Note}">
 </property>
 </bean>
 <bean
 class="org.springframework.beans.factory.config.Pr
                opertyPlaceholderConfigurer">
 <property name="locations">
 <value>classpath:spring.properties</value>
 </property>
 </bean>
 </beans>
```
The above xml file refers the following 'spring.properties' file
Content of 'spring.properties' file
Note = Value from external file.

Instead of

```
<bean id="restaurantBean"
   class="org.opentext.Springpractice.MyTest" >
<property name="welcomeNote" value="Value
                from External file"> </property>
</bean>
```

15. Bean with singleton scope.
```
<bean id="myBean"
 class="org.opentext.Springpractice.MyTest"
                        scope="singleton">
</bean>
```
Bean with prototype scope
```
<bean id="myBean"
 class="org.opentext.Springpractice.MyTest"
                        scope="prototype">
</bean>
```

17.
```
<bean id="outerBean" class="...">
<property name="target">
<bean id="innerBean" class="..."/>
</property>
</bean>
```

19.
```
<bean class="myBean">
<property name="address"><null/></property>
</bean>
```

23. Disable component scanning by not using the following configuration element in the Spring XML configuration file:
```
<!-- Component scanning will significantly slow down
                application initialization time -->
<context:component-scan base-package=""/>
```
Instead, explicitly declare your dependencies, such as:
```
<bean id = "myComponentBean"
class="org.opentext.MyComponent"/>
 <bean id="myOtherComponentBean"
class="org.opentext.MyComponent"/>
```

28.
```
<beanid="first" class="com.opentext.First"
                        depends-on="second"/>
<beanid = "second" class="com.opentext.Second"/>
```

32. Configuration example for this Coherence Cache is
```
<bean id="customerTarget"
 class="org.opentext.cache.samples.dao.Customer" />
```

```
<!-- Properties -->
</bean>
<coherence:proxy id="customerDao"
                                refId="customerTarget“>
 <coherence:caching  methodName="load"
                          cacheName="customerCache" />
<coherence:flushing   methodName="update"
                          cacheNames="customerCache" />
</coherence:proxy>
<bean id="customerManager"
class =
 "org.opentext.cache.samples.business.
                                        CustomerManager" />
<property name="customerDao" ref="customerDao" />
</bean>
```

42.
```
String[] requiredResources =
 {"file1.xml",
 "file2.xml"};
ApplicationContext orderServiceContext = new
 ClassPathXmlApplicationContext(requiredResources);
```

Instead of

```
<beans>
 <import resource="file1.xml"/>
 <import resource="file2.xml"/>
 <bean id="requiredService"
    class="com.opentext.spring.RequiredService"/>
 <beans>
```

56.
```
ApplicationContext context = new
ClassPathXmlApplicationContext("SpringConfig.xml");
// Actual Code
 ((ClassPathXmlApplicationContext)context).close();
```

57. Lazy Loading

```
    <bean id="myBean"
      class="org.opentext.Springpractice.MyTest”
       lazy-init=”true”>
       </bean>
```

Pre Loading

```
    <bean id="myBean"
      class="org.opentext.Springpract6ice.MyTest”
       lazy-init=”false”>
       </bean>
```

V. CLOSING COMMENTS

Spring is a light weight application framework and can be treated as Framework of frameworks. Spring can be used for developing any sort of applications. It can be stand alone, web application, distributed and enterprise applications. Spring is flexible, when compared to Struts (a framework) as Struts is used for developing only web applications, whereas Spring can be used for developing any sort of applications. Spring simplifies the usage of existing java API's by providing abstractions for different APIs.

This paper brought several best practices and tactics in Spring Application Framework upfront.

FUTURE WORK: I extend this work by bringing out more appraisals, methodologies by testing and discussing with several performance engineers and other experts.

VI. REFERENCES

[1] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu, “Professional Java Development with the Spring Framework”, Wiley Publishing, Copyright 2005.

[2] Willie Wheeler, “Spring in Practice”, Manning Publications, Copyright 2013.

[3] Gary Mak, Josh Long, Daniel Rubio, “Spring Recipes”, 2nd Edition, Apress Publications.Copyright 2010.

[4] http://en.wikipedia.org/wiki/Spring_Framework reffered on 27th December, 2013.

ABOUT THE AUTHOR

Vamsi Krishna completed his Bachelors in Computer Science from JNTUK University College of Engineering, Vizianagaram. He is Oracle Certified JAVA SE7 Programmer, DBA Professional 11g, Performance Tuning Expert, Database Security Implementation Specialist & SQL Expert. He is fervor towards Database Security and associated research. His other areas of interests entail SQL & Database Tuning, Performance and Security Engineering in Programming etc.