# Chapter 1 – Introduction to Spring framework

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing enterprise applications. Spring handles the infrastructure so you can focus on your application.

Spring framework is an open source Java platform and it was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.

The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

## Benefits of Using Spring Framework:

Following is the list of few of the great benefits of using Spring Framework:

- Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.
- Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.
- Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBean-style POJOs, it becomes easier to use dependency injection for injecting test data.
- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks.
- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

Examples of how you, as an application developer, can use the Spring platform advantage:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

## Dependency Injection and Inversion of Control

## Background

Java applications -- a loose term that runs the gamut from constrained applets to n-tier server-side enterprise applications -- typically consist of objects that collaborate to form the application proper. Thus the objects in an application have *dependencies* on each other.

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers. True, you can use design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* to compose the various classes and object instances that make up an application. However, these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth. Patterns are formalized best practices that *you must implement yourself* in your application.

The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own applications.

## Dependency Injection (DI):

The technology that Spring is most identified with is the **Dependency Injection (DI)** flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection helps in gluing these classes together and same time keeping them independent.

What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent on class B. Now, let's look at the second part, injection. All this means is that class B will get injected into class A by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework, so I will explain this concept in a separate chapter with a nice example.

## Aspect Oriented Programming (AOP):

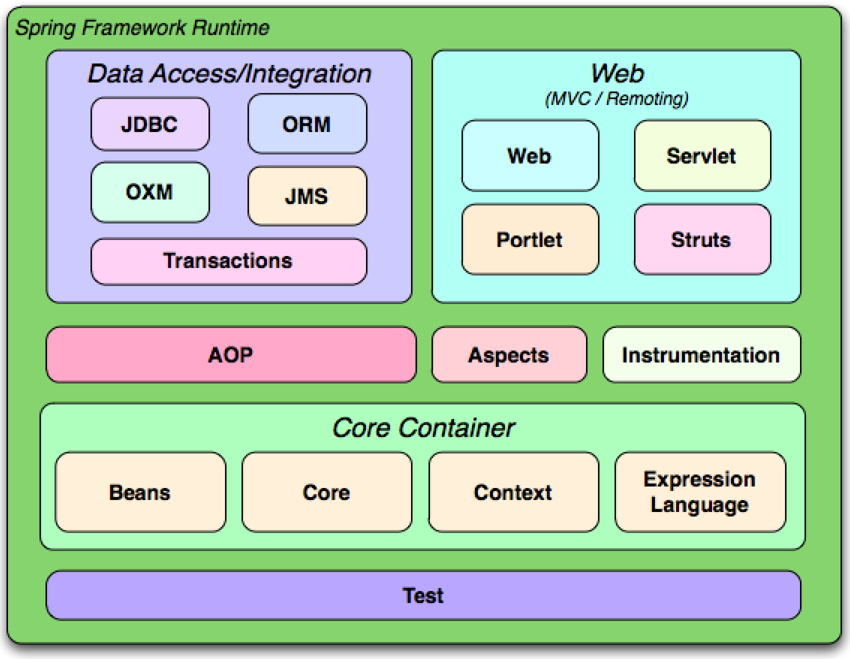
One of the key components of Spring is the **Aspect oriented programming (AOP)** framework. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, and caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Whereas DI helps you decouple your application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect.

The AOP module of Spring Framework provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. I will discuss more about Spring AOP concepts in a separate chapter.

## Spring Modules

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.



Overview of the Spring Framework

## Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules.

The Core and Beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory 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 module builds on the solid base provided by the Core and Beans 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.

The Expression Language module provides a powerful expression language 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 context 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.

## Data Access/Integration

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

- The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.
- The *ORM* module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis. Using the ORM package 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 OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service (JMS) module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs (plain old Java objects)*.

## Web

The *Web* layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

Spring's *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 the web-related parts of Spring's remoting support.

The *Web-Servlet* module contains Spring's model-view-controller (*MVC*) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.

The *Web-Struts* module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.

The *Web-Portlet* module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

## AOP and Instrumentation

Spring's AOP module provides an AOP 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 Aspects module provides integration with AspectJ.

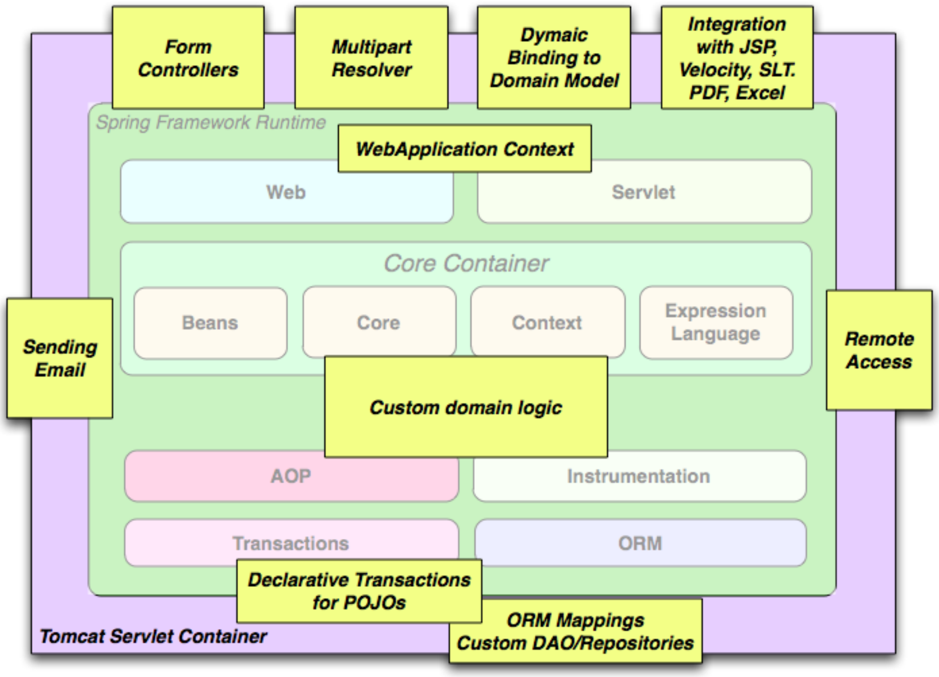
The Instrumentation module provides class instrumentation support and classloader implementations to be used in certain application servers.

## Test

The *Test* module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

## Usage scenarios

The building blocks described previously make Spring a logical choice in many scenarios, from applets to full-fledged enterprise applications that use Spring's transaction management functionality and web framework integration.



## Simple Bean Application

### Spring bean (Java class)

Create a normal Java class HelloWorld.java.Spring's bean is just a normal Java class, and declare in Spring bean configuration file .

```
package com.seed;
 /**
 * Spring bean
 *
 */
public class HelloWorld {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public void printHello() {
        System.out.println("Hello ! " + name);
    }
}
```

## Spring bean configuration file

Create an xml file (Spring-Module.xml) at “src/main/resources/Spring-Module.xml“. This is the Spring’s bean configuration file, which declares all the available Spring beans.

File : Spring-Module.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="helloBean" class="com.seed.HelloWorld">
        <property name="name" value="Hello World" />
    </bean>

</beans>
```

## Client Code

Run App.java, it will load the Spring bean configuration file (Spring-Module.xml) and retrieve the Spring bean via getBean() method.

File : App.java

```
package com.seed;

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

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "Spring-Module.xml");

        HelloWorld obj = (HelloWorld) context.getBean("helloBean");
        obj.printHello();
    }
}
```

# Chapter 2 – Spring Core

## Bean Management and Life cycle:

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container.

When defining a <bean> in Spring, you have the option of declaring a scope for that bean. For example, To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton**.

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.

| Scope | Description |
|---|---|
| singleton | This scopes the bean definition to a single instance per Spring IoC container (default). |
| prototype | This scopes a single bean definition to have any number of object instances. |
| request | This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

For example:

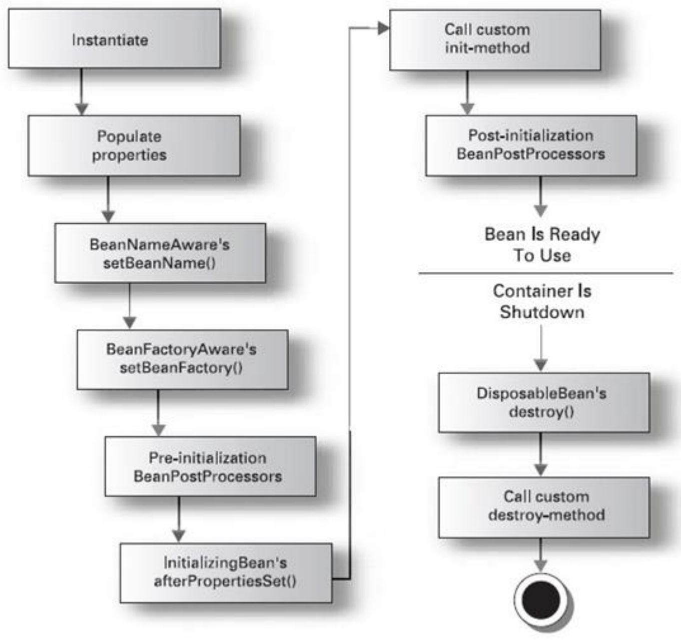
```
<!-- A bean definition with singleton scope -->
<bean id="..." class="..." scope="prototype">
   <!-- collaborators and configuration for this bean go here -->
</bean>
Bean Life Cycle:
```

## Bean Life Cycle

A ***Spring Bean*** represents a ***POJO component*** performing some useful operation. All ***Spring Beans*** reside within a Spring ***Container*** also known as ***IOC Container***. The Spring Framework is transparent and thereby hides most of the complex infrastructure and the communication that happens between the Spring Container and the Spring Beans. This section lists the sequence of activities that will take place between the time of Bean Instantiation and hand over of the Bean reference to the Client Application.

1) The Bean Container finds the definition of the Spring Bean in the Configuration file.
2) The Bean Container creates an instance of the Bean using Java Reflection API.
3) If any properties are mentioned, then they are also applied. If the property itself is a Bean, then it is resolved and set.

4) If the Bean class implements the `BeanNameAware` interface, then the `setBeanName()` method will be called by passing the name of the Bean.
5) If the Bean class implements the `BeanClassLoaderAware` interface, then the method `setBeanClassLoader()` method will be called by passing an instance of the `ClassLoader` object that loaded this bean.
6) If the Bean class implements the `BeanFactoryAware` interface, then the method `setBeanFactory()` will be called by passing an instance of `BeanFactory` object.
7) If there are any `BeanPostProcessors` object associated with the `BeanFactory` that loaded the Bean, then the method `postProcessBeforeInitialization()` will be called even before the properties for the Bean are set.
8) If the Bean class implements the `InitializingBean` interface, then the method `afterPropertiesSet()` will be called once all the Bean properties defined in the Configuration file are set.
9) If the Bean definition in the Configuration file contains a `'init-method'` attribute, then the value for the attribute will be resolved to a method name in the Bean class and that method will be called.
10) The `postProcessAfterInitialization()` method will be called if there are any Bean Post Processors attached for the Bean Factory object.
11) If the Bean class implements the `DisposableBean` interface, then the method `destroy()` will be called when the Application no longer needs the bean reference.
12) If the Bean definition in the Configuration file contains a `'destroy-method'` attribute, then the corresponding method definition in the Bean class will be called.


Instantiate
Populate properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean Is Ready To Use
Container Is Shutdown
DisposableBean's destroy()
Call custom destroy-method

## Spring Ioc Containers

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

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

Spring provides following two distinct types of containers.

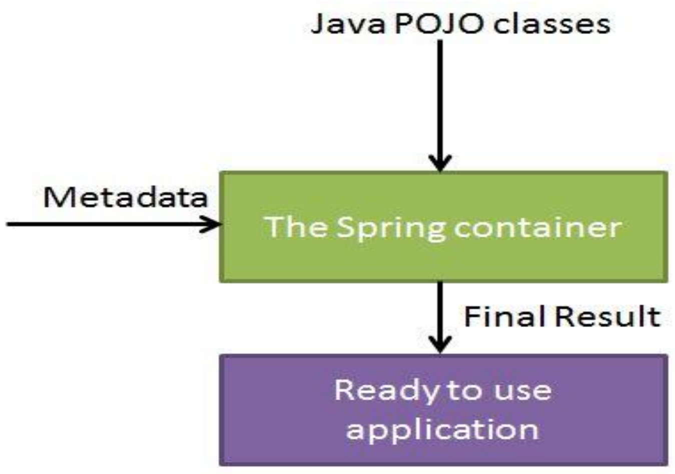
## 1. Spring BeanFactory Container

This is the simplest container providing basic support for DI and defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.

## 2. Spring ApplicationContext Container

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContext interface.

The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.

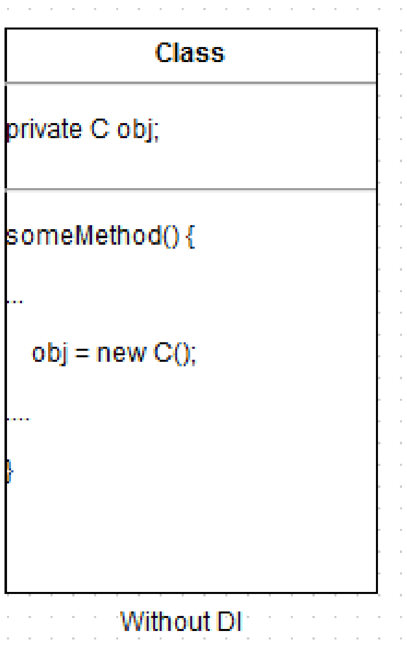


## Dependency Injection

Dependency injection is a jargon created by Martin Fowler and it is also called as inversion of control. In object oriented design, objects have relationship with one another. A class (A) can have attributes (B) and methods. Those attributes are again instances of another class (C). If class (A) wants to work and perform its objective, attributes (B) should be instantiated.

There are different ways to instantiate an object and we have seen a lot in our design pattern tutorial series. A simple and direct way is to use the "new" operator and call the constructor of Class (C) where we need that instance in class (A). This is class A has obsolute control over creation of attribute (B). It decides which class (C) to call and how to call etc.

Now, if we outsource that ‘instantiation and supplying an instance’ job to some third party. Class (A) needs instance of class (C) to operate, but it outsources that responsibility to some third party. The designated third party, decides the moment of creation and the type to use to create the instance. The dependency between class (A) and class (C) is injected by a third party. Whole of this agreement involves some configuration information too. This whole process is called dependency injection.



## Difference between Dependency Injection and Factory

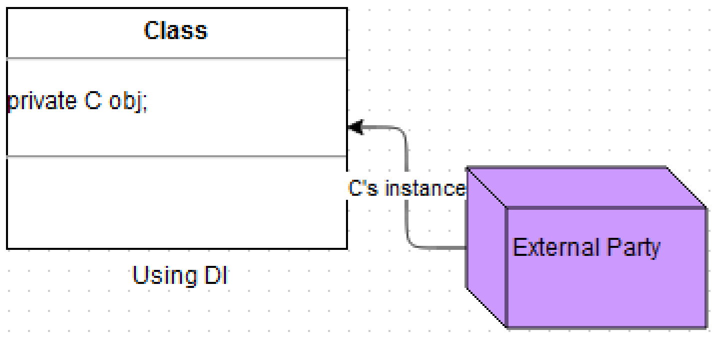
Factory design pattern and dependency injection may look related but we look at them with microscope then we can understand the are different. Even if we use a factory the dependent class has the responsibility of creating the instance but the core of dependency injection is separating that responsibility to external component.

## Factory Example:

```
Class A {
private C obj;

public void someMethod() {
   ...
   this.obj = MyObjectFactory.getC();
   ...
}
}
```

## With Dependency Injection:



```
Class A {
private C obj;


public void someMethod(C obj) {
  ...
  this.obj = obj;
  ...
}
}
```

With DI the contract is different, pass C’s instance to get the job done. So the responsibility is with an external person to decide.

## Advantages of Dependency Injection

- Loosely couple architecture.
- Separation of responsibility
- Configuration and code is separate.
- Using configuration, a different implementation can be supplied without changing the dependent code.
- Testing can be performed using mock objects.

## Dependency Injection Types

Dependency injection is classified into three categories namely,

1) Constructor Injection
2) Setter Injection
3) Interface-based Injection

- **Constructor Injection** - as the name suggests, in this form the dependencies are injected using a constructor and hence the constructor needs to have all the dependencies (either scalar values or object references) declared into it. Though, this form of DI is directly supported by the **Spring IoC Container**. But, Constructor Injection is mainly used by a highly embeddable, lightweight, and full-service IoC Container named PicoContainer.

**User.java File**

```
package com.seed;
```

```
public class User
{

  private String name;
  private int age;
  private String country;

  User(String name, int age, String country)
  {
    this.name=name;
    this.age=age;
    this.country=country;
  }
}
```

**Bean Configuration**

```
<beans....>
<bean id="user" class="com.seed.User" >
  <constructor-arg value="Eswar" />
  <constructor-arg value="24"/>
  <constructor-arg value="India"/>
</bean>
</beans>
```

- **Setter/Mutator Injection** - this form of DI uses the setter methods (also known as mutators) to inject the dependencies into the dependent components. Evidently all the dependent objects will have setter methods in their respective classes which would eventually be used by the Spring IoC container. This form of DI is also directly supported by Spring IoC Container. Though Spring framework supports both forms of DI namely the Constructor Injection and the Setter/Mutator Injection directly, but the IoC Container prefers the latter over the first.

**User.java File**

```
package com.vaannila;

public class User
{
   private String name;
  private int age;
  private String country;

  public String getName()
 {
   return name;
 }
  public void setName(String name)
 {
  this.name = name;
 }
 public int getAge()
 {
  return age;
 }
  public void setAge(int age)
 {
   this.age = age;
 }
  public String getCountry()
 {
  return country;
 }
  public void setCountry(String country)
 {
  this.country = country;
 }


}
```

**Bean configuration File**

```
<beans....>
<bean id="user" class="com.seed.User" >
  <property name="name" value="Eswar" />
  <property name="age" value="24"/>
  <property name="country" value="India"/>
</bean>
```

```
</beans>
```

- **Interface Injection** - in this case any implementation of the interface can be injected and hence it's relatively complex than the other two where the objects of specified classes are injected. It's not supported directly by the Spring IoC Container. Interface Injection is used by The Apache Avalon. In 2004 Apache Avalon project closed after growing into several sub-projects including Excalibur, Loom, Metro, and Castle.

## Collection Injection In Spring

You have seen how to configure primitive data type using value attribute and object references using ref attribute of the <property> tag in your Bean configuration file. Both the cases deal with passing singular value to a bean.

Now what about if you want to pass plural values like Java Collection types List, Set, Map, and Properties. To handle the situation, Spring offers four types of collection configuration elements which are as follows:

| Element | Description |
|---|---|
| <list> | This helps in wiring ie injecting a list of values, allowing duplicates |
| <set> | This helps in wiring a set of values but without any duplicates. |
| <map> | This can be used to inject a collection of name-value pairs where name and value can be of any type. |
| <props> | This can be used to inject a collection of name-value pairs where the name and value are both Strings. |

You can use either <list> or <set> to wire any implementation of java.util.Collection or an array.

You will come across two situations (a) Passing direct values of the collection and (b) Passing a reference of a bean as one of the collection elements.

Following Customer Example is showing four collection properties.

## Customer.java File

```
package com.seed;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;
public class Customer
{
   private List<Object> lists;
   private Set<Object> sets;
```

```
    private Map<Object, Object> maps;
    private Properties pros;
    //...
}
```

## List Mapping:

```
<property name="lists">
      <list>
         <value>1</value>
         <ref bean="PersonBean" />
         <bean class="com.mkyong.common.Person">
            <property name="name" value="mkyongList" />
            <property name="address" value="address" />
            <property name="age" value="28" />
         </bean>
      </list>
   </property>
```

## Set example

```
   <property name="sets">
      <set>
         <value>1</value>
         <ref bean="PersonBean" />
         <bean class="com.mkyong.common.Person">
            <property name="name" value="mkyongSet" />
            <property name="address" value="address" />
            <property name="age" value="28" />
         </bean>
      </set>
   </property>
```

## Map example

```
   <property name="maps">
      <map>
         <entry key="Key 1" value="1" />
         <entry key="Key 2" value-ref="PersonBean" />
         <entry key="Key 3">
            <bean class="com.mkyong.common.Person">
               <property name="name" value="mkyongMap" />
               <property name="address" value="address" />
               <property name="age" value="28" />
            </bean>
            </entry>
        </map>
    </property>
```

## Properties example

```
    <property name="pros">
        <props>
            <prop key="admin">admin@nospam.com</prop>
            <prop key="support">support@nospam.com</prop>
        </props>
    </property>
```

## Bean Auto wiring

You have learnt how to declare beans using the <bean> element and inject <bean> with using <constructor-arg> and <property> elements in XML configuration file.

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.

## Autowiring Modes:

| Mode | Description |
|---|---|
| no | This is default setting which means no autowiring and you should use explicit bean reference for wiring. |
| byName | Autowiring by property name. Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. |
| byType | Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown. |
| constructor | Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |
| autodetect | Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType. |

There are following autowiring modes which can be used to instruct Spring container to use autowiring for dependency injection. You use the autowire attribute of the <bean/> element to specify autowire mode for a bean definition.

You can use byType or constructor autowiring mode to wire arrays and other typed-collections

## Autowiring Example

## Customer.java File

```
package com.seed;


public class Customer
{
   //you want autowired this field.
   private Person person;
   private int type;
   private String action;
   //getter and setter method
 }
```

## Bean Configuration file

## Auto-Wiring 'no'

This is the default mode, you need to wire your bean via 'ref' attribute.

```
   <bean id="customer" class="com.seed..Customer">
                  <property name="person" ref="person" />
   </bean>
   <bean id="person" class="com.mkyong.common.Person" />
```

## Auto-Wiring 'byName'

Auto-wire a bean by property name. In this case, since the name of "person" bean is same with the name of the "customer" bean's property ("person"), so, Spring will auto wired it via setter method – "setPerson(Person person)".

```
   <bean id="customer" class="com.seed..Customer" autowire="byName" />
   <bean id="person" class="com.mkyong.common.Person" />
```

## Auto-Wiring 'byType'

Auto-wire a bean by property data type. In this case, since the data type of "person" bean is same as the data type of the "customer" bean's property (Person object), so, Spring will auto wired it via setter method – "setPerson(Person person)".

```
    <bean id="customer" class="com.seed..Customer" autowire="byType" />

    <bean id="person" class="com.mkyong.common.Person" />
```

## Auto-Wiring ‘constructor’

Auto-wire a bean by property data type in constructor argument. In this case, since the data type of “person” bean is same as the constructor argument data type in “customer” bean’s property (Person object), so, Spring auto wired it via constructor method – “public Customer(Person person)“.

```
<bean id="customer" class="com.seed..Customer" autowire="constructor" />

<bean id="person" class="com.mkyong.common.Person" />
```

## Auto-Wiring ‘autodetect’

If a default constructor is found, uses “constructor”; Otherwise, uses “byType”. In this case, since there is a default constructor in “Customer” class, so, Spring auto wired it via constructor method – “public Customer(Person person)“.

```
<bean id="customer" class="com.seed..Customer" autowire="autodetect" />

<bean id="person" class="com.mkyong.common.Person" />
```

# Chapter 3 – Spring AOP

Some programming tasks cannot be neatly encapsulated in objects, but must be scattered throughout the code

- Examples:
  - Logging (tracking program behavior to a file)
  - Profiling (determining where a program spends its time)
  - Tracing (determining what methods are called when)
  - Transaction Management
  - Security Management
- The result is crosscuting code--the necessary code "cuts across" many different classes and methods.

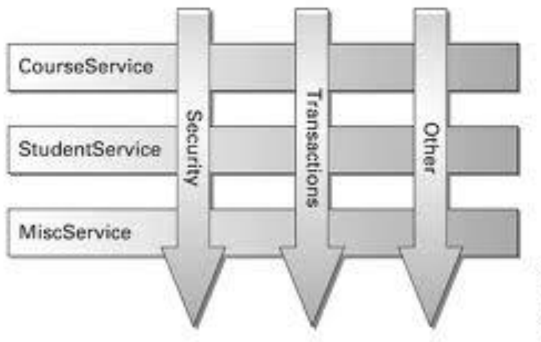


Figure Cross-cutting concerns

Above-mentioned problems are easey solvable via AOP.

One of the key components of Spring Framework is the **Aspect oriented programming (AOP)** framework. Aspect Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, and caching etc.

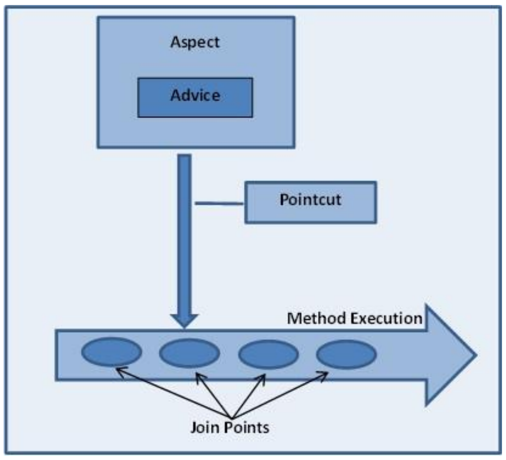
The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java and others.

Spring AOP module provides interceptors to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.

## AOP Terminologies:

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

| Terms | Description |
|---|---|
| Aspect | A module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. |

| | |
|---|---|
| Join point | This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework. |
| Advice | This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework. |
| Pointcut | This is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples. |
| Introduction | An introduction allows you to add new methods or attributes to existing classes. |
| Target object | The object being advised by one or more aspects, this object will always be a proxied object. Also referred to as the advised object. |
| Weaving | Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime. In Spring AOP makes it possible to modularize and separate logging, transaction like services and apply them declaratively to the components Hence programmer can focus on specific concerns. Aspects are wired into objects in the spring XML file in the way as JavaBean. This process is known as 'Weaving'.   Aspect<br>Advice<br>Pointcut<br>Method Execution<br>Join Points |

## Types of Advice

Spring aspects can work with five kinds of advice mentioned below:

| Advice | Description |
|---|---|
| before | Run advice before the a method execution. |
| after | Run advice after the a method execution regardless of its outcome. |
| after-returning | Run advice after the a method execution only if method completes successfully. |

| | |
|---|---|
| after-throwing | Run advice after the a method execution only if method exits by throwing an exception. |
| around | Run advice before and after the advised method is invoked. |

## Example of AOP:

### IEmployee Interface

```
package com.emp;
import com.emp.account.Account;
import com.service.LoggingService;

public interface IEmployee
{
  public void withDrawFromAccount(int amt); // throws LowBalanceException

  public String getName();
  public void setName(String name);

    public Account getAccount();
  public void setAccount(Account account);

}
```

**Employee.java file**

```
package com.emp;

import com.emp.account.Account;
import com.service.LoggingService;

public class Employee implements IEmployee
{
  private String name;

  private Account account;
  //private SalaryAccount account;
  //private DematAccount account;

    //following line is a traditional way, without making use of IoC
  //Instead of using loggingService here its better to treat it as
  //another Aspect and apply LoggingServiceAdvice to this Employee
declaratively.
  //LoggingService loggingservice = new LoggingService();


  public Employee(String name)
  {
     this.name = name;
     //following line introduces the strong coupling between
     //Employee and one of the Account class i.e. SalaryAccount
        //account = new SalaryAccount();
  }

  public void withDrawFromAccount(int amt) // throws LowBalanceException
  {
     /* following line is going to give call to logging service
       limitations of tight-coupling this LoggingService with Employee
       1.If tommorrow if this concern changes, one has to modify all the
         components having following line in it.
       2. Employee is coupled with this LoggingService which is actually
          not core functionality of this Bean. */
     //loggingservice.audit(name);

     account.withDraw(amt);
  }

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

    public Account getAccount()
  {
     return this.account;
  }
  public void setAccount(Account account)
  {
     System.out.println("am inside setAccount(), the dependency is
getting injected...");
     //here the dependent object Account has been injected into
     //this Employee class. Depending on XML configuration, any
     //subclass of Account can be injected into this Employee.
     this.account = account;
  }

}

package com.emp.account;

public interface Account
{
  public void withDraw(int amt);

}
package com.emp.account;

public class SalaryAccount implements Account
{
  public void withDraw(int amt)
  {
     System.out.println("withdrawing from SalaryAccount ...");
  }
}
```

Applying Services

```
package com.service;
//import org.apache.log4j.Logger;
public class LoggingService
{
  //Logger emplogger = Logger.getLogger(Employee.class);

  public LoggingService()
  {

  }

  public void audit(String name)
  {
     //emplogger.debug(name + " :: is going to withDraw !!");
     System.out.println(name + " :: is going to withDraw !!");

  }
}

package com.service;

//import org.apache.log4j.Logger;
import java.lang.reflect.Method;
import com.emp.Employee;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingServiceAdvice implements MethodBeforeAdvice
{
  //Logger emplogger = Logger.getLogger(Employee.class);

  public LoggingServiceAdvice()
  {
  }

    public void before(Method method, Object args[], Object target)
throws Throwable
  {
      Employee emp = (Employee) target;
    String name = emp.getName();
    System.out.println("inside before method and " + name + " :: is going
to withDraw !!");
  }

}
```

**Configuring bean:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<bean id="emloyeeTarget" class="com.emp.Employee">
<property name="account">
  <ref bean="acct" />
</property>
```

```

<constructor-arg>
<value>amit</value>
</constructor-arg>
</bean>

<bean id="acct" class="com.emp.account.SalaryAccount" />


<bean id="logging" class="com.service.LoggingServiceAdvice"/>


<bean id="emloyee"
class="org.springframework.aop.framework.ProxyFactoryBean">
<!-- the value of list for propery=proxyInterfaces should be an
Interface only -->
<property name="proxyInterfaces">
<list>
<value>com.emp.IEmployee</value>
</list>
</property>
<property name="interceptorNames">
<list>
<value>logging</value>
</list>
</property>
<property name="target"><ref bean="emloyeeTarget"/></property>
</bean>
</beans>
```

# Chapter 4 – Spring-DAO

## Spring JDBC

Spring provides a simplification in handling database access with the Spring JDBC Template.

The Spring JDBC Template has the following advantages compared with standard JDBC.

- The Spring JDBC template allows to clean-up the resources automatically, e.g. release the database connections.
- The Spring JDBC template converts the standard JDBC SQLExceptions into RuntimeExceptions. This allows the programmer to react more flexible to the errors. The Spring JDBC template converts also the vendor specific error messages into better understandable error messages.

The Spring JDBC template offers several ways to query the database. queryForList() returns a list of HashMaps. The name of the column is the key in the hashmap for the values in the table.

More convenient is the usage of ResultSetExtractor or RowMapper which allows to translates the SQL result direct into an object (ResultSetExtractor) or a list of objects (RowMapper). Both these methods will be demonstrated in the coding.

Spring JDBC template internally uses JDBC api which eliminates lot of problems of JDBC api.

Problem of JDBC API:

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

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

**JDBC Template class:**

It is a central class in Spring JDBC support classes.It takes care of creation and release of resources such as creating and closing the connection object etc. so it will not lead to any problem if forget to close connection.

It handles the exception and provides the informative exception messages with the help of exception classes defined in the **org.springframework.dao** package.

All the database operations can be performed with the help of JDBC template such as insertion,updation,deletion and retrieval of data from the database.

Spring JdbcTemplate exposes many helpful methods for performing CRUD operations on database. Following are most common methods that we use from JdbcTemplate.

| **Method Name** | **Use** |
|---|---|
| execute(String sql) | Issue a single SQL execute, typically a DDL statement. |
| queryForList(String sql, Object[] args) | Query given SQL to create a prepared statement from SQL and a list of arguments to bind to the query, expecting a result list. |
| update(String sql) | Issue a single SQL update operation (such as an insert, update |

| **Method Name** | **Use** |
|---|---|
| | or delete statement). |

**Example With JdbcTemplate**

```
package com.seed;
public class Person {
  private String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
}

package com.seed;

import java.util.List;
import javax.sql.DataSource;

import domainmodel.Person;

public interface IDao {

  void setDataSource(DataSource ds);

  void create(String firstName, String lastName);

  List<Person> select(String firstname, String lastname);

  List<Person> selectAll();

  void deleteAll();

  void delete(String firstName, String lastName);

}
```

With JdbcTemplate, you save a lot of typing on the redundant codes, becuase JdbcTemplate will handle it automatically.

```
package com.seed;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import dao.mapper.PersonRowMapper;
import domainmodel.Person;

public class DerbyDao implements IDao {
  private DataSource dataSource;
  JdbcTemplate jdbctemp;

  public void setDataSource(DataSource ds) {
    dataSource = ds;
  }

  public void create(String firstName, String lastName) {
   jdbctemp = new JdbcTemplate(dataSource);
    jdbctemp.update("INSERT INTO PERSON (FIRSTNAME, LASTNAME)
VALUES(?,?)",
        new Object[] { firstName, lastName });
  }

  public List<Person> select(String firstname, String lastname) {
    jdbctemp = new JdbcTemplate(dataSource);
    return jdbctemp
        .query("select  FIRSTNAME, LASTNAME from PERSON where FIRSTNAME
= ? AND LASTNAME= ?",
            new Object[] { firstname, lastname },
            new PersonRowMapper());
  }

  public List<Person> selectAll() {
    jdbctemp = new JdbcTemplate(dataSource);
    return jdbctemp.query("select FIRSTNAME, LASTNAME from PERSON",
        new PersonRowMapper());
  }

  public void deleteAll() {
    jdbctemp = new JdbcTemplate(dataSource);
    jdbctemp.update("DELETE from PERSON");
  }

  public void delete(String firstName, String lastName) {
    jdbctemp = new JdbcTemplate(dataSource);
    jdbctemp.update("DELETE from PERSON where FIRSTNAME= ? AND LASTNAME
= ?",
        new Object[] { firstName, lastName });
  }
}
```

Spring JdbcTemplate is a class that takes care of all the boilerplate code required for creating a database connection and releasing the resources. It makes our life a lot easier by saving the effort and development time. To initialize the object of JdbcTemplate, we will use the datasource bean defined above. Once created, we could easily inject its reference to any class that wants to communicate with database.

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

<bean id="userDao" class="com.seed.DerbyDao">
  <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
```

## Integrating Hibernate with Spring

When we were kids, riding a bike was fun, wasn’t it? We’d ride to school in the morn- ings. When school let out, we’d cruise to our best friend’s house. When it got late and our parents were yelling at us for staying out past dark, we’d peddle home for the night. Gee, those days were fun. Then we grew up, and now we need more than a bike. Sometimes we have to travel a long distance to work. Groceries have to be hauled, and ours kids need to get to soc- cer practice. And if you live in Texas, air conditioning is a must! Our needs have sim- ply outgrown our bikes.

JDBC is the bike of the persistence world. It’s great for what it does, and for some jobs it works fine. But as our applications become more complex, so do our persis- tence requirements. We need to be able to map object properties to database columns and have our statements and queries created for us, freeing us from typing an endless string of question marks. We also need features that are more sophisticated:  *Lazy loading*—As our object graphs become more complex, we sometimes don’t want to fetch entire relationships immediately. To use a typical example, sup- pose we’re selecting a collection of PurchaseOrder objects, and each of these objects contains a collection of LineItem objects. If we’re only interested in PurchaseOrder attributes, it makes no sense to grab the LineItem data. This could be expensive. Lazy loading allows us to grab data only as it’s needed.

*Eager fetching*—This is the opposite of lazy loading. Eager fetching allows you to grab an entire object graph in one query. In the cases where we know that we need a PurchaseOrder object and its associated LineItems, eager fetching lets us get this from the database in one operation, saving us from costly round- trips.  *Cascading*—Sometimes changes to a database table should result in changes to other tables as well. Going back to our purchase order example, when an Order object is deleted, we also want to delete the associated LineItems from the database.

Several frameworks are available that provide these services. The general name for these services is *object-relational mapping (ORM)*. Using an ORM tool for your persistence layer can save you literally thousands of lines of code and hours of development time. This lets you switch your focus from writing error-prone SQL code to addressing your application requirements. Spring provides support for several persistence frameworks, including Hibernate, iBATIS, Java Data Objects (JDO), and the Java Persistence API (JPA). As with Spring’s JDBC support, Spring's support for ORM frameworks provides inte- gration points to the frameworks as well as some additional services:  Integrated support for Spring declarative transactions  Transparent exception handling  Thread-safe, lightweight template classes  DAO support classes  Resource management

Hibernate is an open source persistence framework that has gained significant popularity in the developer community. It provides not only basic object-relational mapping but also all the other sophisticated features you’d expect from a full-featured ORM tool, such as caching, lazy loading, eager fetching, and distributed caching.

Spring's beans declaration for Business Object (BO) and Data Access Object (DAO). The DAO class (CustomerDaoImpl.java) is extends Spring's "**HibernateDaoSupport**" class to access the Hibernate function easily.

Refer site for demo

http://www.javatpoint.com/hibernate-and-spring-integration

# Chapter 5 – Spring MVC

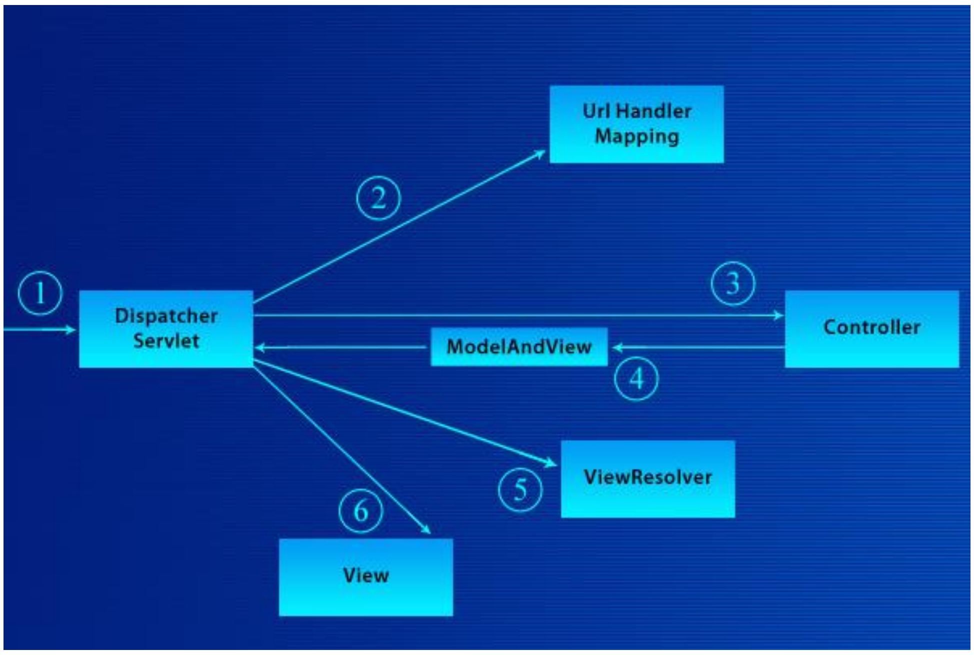
## Getting started with Spring MVC

At first glance, you may think that Spring's MVC framework is a lot like Mousetrap. Instead of moving a ball around through various ramps, teeter-totters, and wheels, Spring moves requests around between a dispatcher servlet, handler mappings, controllers, and view resolvers.

## Following a request through Spring MVC

Every time a user clicks a link or submits a form in their web browser, a request goes to work. A request's job description is that of a courier. Just like a postal carrier or a FedEx delivery person, a request lives to carry information from one place to another.

The request is a busy fellow. From the time it leaves the browser until it returns with a response, it'll make several stops, each time dropping off a bit of information and picking up some more.



When the request leaves the browser, it carries information about what the user is asking for. At least, the request will be carrying the requested URL. But it may also carry additional data such as the information submitted in a form by the user.

The first stop in the request's travels is at Spring's `DispatcherServlet`. Like most Java-based web frameworks, Spring MVC funnels requests through a single front controller servlet. A front controller is a common web application pattern where a single servlet delegates responsibility for a request to other components of an application to perform actual processing. In the case of Spring MVC, DispatcherServlet is the front controller.

The DispatcherServlet's job is to send the request on to a Spring MVC controller. A controller is a Spring component that processes the request. But a typical application may have several controllers and DispatcherServlet needs some help deciding which controller to send the request to. So the DispatcherServlet consults one or more handler mappings to figure out where the request's next stop will be. The handler mapping will pay particular attention to the URL carried by the request whenmaking its decision.

Once an appropriate controller has been chosen, DispatcherServlet sends the request on its merry way to the chosen controller. At the controller, the request will drop off its payload (the information submitted by the user) and patiently wait while the controller processes that information. (Actually, a well-designed controller performs little or no processing itself and instead delegates responsibility for the business logic to one or more service objects.)

The logic performed by a controller often results in some information that needs to be carried back to the user and displayed in the browser. This information is referred to as the *model*. But sending raw information back to the user isn't sufficient—it needs to be formatted in a user-friendly format, typically HTML. For that the information needs to be given to a *view*, typically a JSP.

One of the last things that a controller does is package up the model data and identify the name of a view that should render the output. It then sends the request, along with the model and view name, back to the DispatcherServlet. So that the controller doesn't get coupled to a particular view, the view name passed back to DispatcherServlet doesn't directly identify a specific JSP. In fact, it doesn't even necessarily suggest that the view is a JSP at all. Instead, it only carries a logical name which will be used to look up the actual view that will produce the result.

The DispatcherServlet will consult a view resolver to map the logical view name to a specific view implementation, which may or may not be a JSP. Now that DispatcherServlet knows which view will render the result, the request's job is almost over. Its final stop is at the view implementation (probably a

JSP) where it delivers the model data. The request's job is finally done. The view will use the model data to render output that will be carried back to the client by the (not-so-hardworking) response object.

We'll dive into each of these steps in more detail throughout this chapter. But first things first—we need to set up Spring MVC and DispatcherServlet in the Spitter application.

## Setting up Spring MVC

At the heart of Spring MVC is DispatcherServlet, a servlet that functions as Spring MVC's front controller. Like any servlet, DispatcherServlet must be configured in the web application's web.xml file. So the first thing we must do to use Spring MVC in our application is to place the following <servlet> declaration in the web.xml file:

```
<servlet>
<servlet-name>spitter</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
```

```
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
```

The <servlet-name> given to the servlet is significant. By default, when Dispatcher-Servlet is loaded, it'll load the Spring application context from an XML file whose name is based on the name of the servlet. In this case, because the servlet is named spitter, DispatcherServlet will try to load the application context from a file named spitter-servlet.xml (located in the application's WEB-INF directory).

Next we must indicate what URLs will be handled by the DispatcherServlet. It's common to find DispatcherServlet mapped to URL patterns such as *.htm, /*, or /app. But these URL patterns have a few problems:

- The*.htm pattern implies that the response will always be in HTML form (which isn't necessarily the case).
- Mapping it to /* doesn't imply any specify type of response, but indicates that DispatcherServlet will serve *all* requests. That makes serving static content such as images and stylesheets more difficult than necessary.
- The/app pattern (or something similar) helps us distinguish Dispatcher-Servlet-served content from other types of content. But then we have an implementation detail (specifically, the /app path) exposed in our URLs. That leads to complicated URL rewriting tactics to hide the /app path.

```
<servlet-mapping>
<servlet-name>spitter</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
```

If it concerns you that DispatcherServlet will be handling those kinds of requests, then hold on for a bit. A handy configuration trick frees you, the developer, from having to worry about that detail much. Spring's mvc namespace includes a new <mvc:resources> element that handles requests for static content for you. All you must do is configure it in the Spring configuration.

That means that it's now time to create the spitter-servlet.xml file that Dispatcher-Servlet will use to create an application context. The following listing shows the beginnings of the spitter-servlet.xml file.

```
<?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:mvc="http://www.springframework.org/schema/mvc"
xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<mvc:resources mapping="/resources/**"
location="/resources/" />
</beans>
```

As said earlier, all requests that go through DispatcherServlet must be handled in some way, commonly via controllers. Since requests for static content are also being handled by DispatcherServlet, we're going to need some way to tell Dispatcher-Servlet how to serve those resources. But writing and maintaining a controller for that purpose seems too involved. Fortunately, the <mvc:resources> element is on the job.

<mvc:resources> sets up a handler for serving static content. The mapping attribute is set to

/resources/**, which includes an Ant-style wildcard to indicate that the path must begin with /resources, but may include any subpath thereof. The location attribute indicates the location of the files to be served. As configured here, any requests whose paths begin with /resources will be automatically served from the /resources folder at the root of the application. Therefore, all of our images, stylesheets, JavaScript, and other static content needs to be kept in the application's /resources folder.

Now that we've settled the issue of how static content will be served, we can start thinking about how our application's functionality can be served. Since we're just getting started, we'll start simple by developing the Spitter application's home page.

## Writing a basic controller

As we develop the web functionality for the Spitter application, we're going to develop resource-oriented controllers. Rather than write one controller for each use case in our application, we're going to write a single controller for each kind of resource that our application serves.

The Spitter application, being rather simple, has only two primary resource types: Spitters who are the users of the application and the spittles that they use to communicate their thoughts. Therefore, you'll need to write a spitter-oriented controller and a spittle-oriented controller.

In addition to controllers for each of the application's core concepts, we also have two other utility controllers. These controllers handle a few requests that are necessary, but don't directly map to a specific concept.

One of those controllers, HomeController, performs the necessary job of displaying the home page—a page that isn't directly associated with either Spitters or Spittles. That will be the first controller we write. But first, since we're developing annotation-driven controllers, there's a bit more setup to do.

## Configuring an annotation-driven Spring MVC

As mentioned earlier, DispatcherServlet consults one or more handler mappings in order to know which controller to dispatch a request to. Spring comes with a handful of handler mapping implementations to choose from, including

- BeanNameUrlHandlerMapping—Maps controllers to URLs that are based on the controllers' bean names.
- ControllerBeanNameHandlerMapping—Similar to BeanNameUrlHandlerMapping, maps controllers to URLs that are based on the controllers' bean names. In this case, the bean names aren't required to follow URL conventions.
- ControllerClassNameHandlerMapping—Maps controllers to URLs by using the controllers' class names as the basis for their URLs.
- DefaultAnnotationHandlerMapping—Maps request to controller and controller methods that are annotated with @RequestMapping.
- SimpleUrlHandlerMapping—Maps controllers to URLs using a property collection defined in the Spring application context.

Using one of these handler mappings is usually just a matter of configuring it as a bean in Spring. But if no handler mapping beans are found, then DispatcherServlet creates and usesBeanNameUrlHandlerMapping and DefaultAnnotationHandler- Mapping. Fortunately, we'll be working primarily with annotated controller classes, so the DefaultAnnotationHandlerMapping that DispatcherServlet gives us will do fine. DefaultAnnotationHandlerMapping maps requests to controller methods that are annotated with @RequestMapping (which we'll see in the next section). But there's more to annotation-driven Spring MVC than just mapping requests to methods. As we build our controllers, we'll also use annotations to bind request parameters to handler method parameters, perform validation, and perform message conversion. Therefore, DefaultAnnotationHandlerMapping isn't enough.

Fortunately, you only need to add a single line of configuration to spitter-servlet.xml to flip on all of the annotation-driven features you'll need from Spring MVC:

<mvc:annotation-driven/>

Although small, the <mvc:annotation-driven> tag packs a punch. It registers several features, including JSR-303 validation support, message conversion, and support for field formatting.

We'll talk more about those features as we need them. For now, we have a home page controller to write.

## Defining the home page controller

The home page is usually the first thing that visitors to a website will see. It's the front door to the rest of the site's functionality. In the case of the Spitter application, the home page's main job is to welcome visitors and to display a handful of recent spittles, hopefully enticing the visitors to join in on the conversation.

Writing a basic controller

HomeController is a basic Spring MVC controller that handles requests for the

home page.

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;
@Controller
public class HomeController {
public static final int DEFAULT_SPITTLES_PER_PAGE = 25;
private SpitterService spitterService;
@Inject
public HomeController(SpitterService spitterService) {
this.spitterService = spitterService;
}
@RequestMapping({"/","/home"})
public String showHomePage(Map<String, Object> model) {
model.put("spittles", spitterService.getRecentSpittles(
DEFAULT_SPITTLES_PER_PAGE));
}

}

return "home";
```

Although HomeController is simple, there's a lot to talk about here. First, the @Controller annotation indicates that this class is a controller class. This annotation is a specialization of the@Component annotation, which means that <context:component-scan> will pick up and register @Controller-annotated classes as beans, just as if they were annotated with @Component.

That means that we need to configure a <context:component-scan> in spitter-servlet.xml so that the HomeController class (and all of the other controllers we'll write) will be automatically discovered and registered as beans. Here's the relevant snippet of XML:

<context:component-scan base-package="com.habuma.spitter.mvc" />

Going back to the HomeController class, we know that it'll need to retrieve a list of the most recent spittles via a SpitterService. Therefore, we've written the constructor to take a SpitterService as an argument and have annotated it with @Inject annotation so that it'll automatically be injected when the controller is instantiated.

The real work takes place in the showHomePage() method. As you can see, it's annotated with @RequestMapping. This annotation serves two purposes. First, it identifies

showHomePage() as a request-handling method. And, more specifically, it specifies that this method should handle requests whose path is either / or /home.

As a request-handling method, showHomePage() takes a Map of String-to-Object as a parameter. This Map represents the model—the data that's passed between the controller and a view. After retrieving a list of recent Spittles from the SpitterService's getRecentSpittles() method, that list is placed into the model Map so that it can be displayed when the view is rendered.

As we write more controllers we'll see that the signature of a request-handling method can include almost anything as an argument. Even though showHomePage() only needed the model Map, we could've added HttpServletRequest, HttpServletResponse, String, or numeric parameters that correspond to query parameters in the request, cookie values, HTTP request header values, or a number of other possibilities. For now, though, the model Map is all we need.

The last thing that showHomePage() does is return a String value that's the logical name of the view that should render the results. A controller class shouldn't play a direct part in rendering the results to the client, but should only identify a view implementation that'll render the data to the client. After the controller has finished its work, DispatcherServlet will use this name to look up the actual view implementation by consulting a view resolver.

We'll configure a view resolver soon. But first let's write a quick unit test to assert that HomeController is doing what we expect it to do.

## TESTING THE CONTROLLER

What's most remarkable about HomeController (and most Spring MVC controllers) is that there's little that's Spring-specific about it. In fact, if you were to strip away the three annotations, this would be a POJO.

From a unit testing perspective, this is significant because it means that HomeController can be tested easily without having to mock anything or create any Spring-specific objects. HomeControllerTest demonstrates how you might test HomeController.

```
package com.habuma.spitter.mvc;
import static com.habuma.spitter.mvc.HomeController.*;
import static java.util.Arrays.*;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;
import java.util.HashMap;
import java.util.List;
import org.junit.Test;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;
public class HomeControllerTest {
@Test
public void shouldDisplayRecentSpittles() {
List<Spittle> expectedSpittles =
asList(new Spittle(), new Spittle(), new Spittle());
SpitterService spitterService = mock(SpitterService.class);
when(spitterService.getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE)).
thenReturn(expectedSpittles);
HomeController controller =
new HomeController(spitterService);
HashMap<String, Object> model = new HashMap<String, Object>();
String viewName = controller.showHomePage(model);
assertEquals("home", viewName);
assertSame(expectedSpittles, model.get("spittles"));
}
}
verify(spitterService).getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE);
```

The only thing that HomeController needs to do its job is an instance of SpitterService, which Mockito2 graciously provides as a mock implementation. Once the mock SpitterService is ready, you just need to create a new instance of Home-Controller and then call the showHomePage() method. Finally, you assert that the list of spittles returned from the mock SpitterService ends up

in the model Map under the spittles key and that the method returns a logical view name of home.

As you can see, testing a Spring MVC controller is like testing any other POJO in your Spring application. Even though it'll ultimately be used to serve a web page, we didn't have to do anything special or web-specific to test it.

At this point we've developed a controller to handle requests for the home page. And we've written a test to ensure that the controller does what we think it should.

One question is still unanswered, though. The showHomePage() method returned a logical view name. But how does that view name end up being used to render output to the user?

## Resolving views

The last thing that must be done in the course of handling a request is rendering output to the user. This job falls to some view implementation—typically JavaServer Pages (JSP), but other view technologies such as Velocity or FreeMarker may be used. In order to figure out which view should handle a given request, DispatcherServlet consults a view resolver to exchange the logical view name returned by a controller for an actual view that should render the results.

In reality, a view resolver's job is to map a logical view name to some implementation of org.springframework.web.servlet.View. But it's sufficient for now to think of a view resolver as something that maps a view name to a JSP, as that's effectively what it does.

Spring comes with several view resolver implementations to choose from, as described table:

| | |
|---|---|
| BeanNameViewResolver | Finds an implementation of View that's registered as a <bean> whose ID is the same as the logical view name. |
| ContentNegotiatingViewResolver | Delegates to one or more other view resolvers, the choice of which is based on the content type being requested. |
| FreeMarkerViewResolver | Finds a FreeMarker-based template whose path is determined by prefixing and suffixing the logical view name |
| . InternalResourceViewResolver | Finds a view template contained within the web application's WAR file. The path to the view template is derived by prefixing and suffixing the logical view name. |
| JasperReportsViewResolver | Finds a view path is derived by prefixing and suffixing the logical view defined as a Jasper Reports report file whose name. |
| ResourceBundleViewResolver | Looks up View implementations from a properties file. |
| TilesViewResolver | Looks up a view that is defined as a Tiles template. The name of the template is the same as the logical view name |
| . UrlBasedViewResolver | This is the base class for some of the other view resolvers, such as InternalResourceViewResolver. It can be used on its own, but it's not as powerful as its sub classes. For example, UrlBasedViewResolver is unable to resolve views based on the current locale. |
| VelocityLayoutViewResolver | This is a subclass of VelocityViewResolver that supports page composition via Spring's VelocityLayout- View (a view implementation that emulates Velocity's VelocityLayoutServlet). |

| | |
|---|---|
| VelocityViewResolver | Resolves a Velocity-based view where the path of a Velocity template is derived by prefixing and suffixing the logical view name. |
| XmlViewResolver | Finds an implementation of View that's declared as a<bean> in an XML file (/WEB-INF/views.xml). This view resolver is a lot like BeanNameViewResolver except that the view <bean>s are declared separately from those for the application's Spring context |
| XsltViewResolver | Resolves an XSLT-based view where the path of the XSLT stylesheet is derived by prefixing and suffixing the logical view name. |

## RESOLVING INTERNAL VIEWS

A lot of Spring MVC embraces a convention-over-configuration approach to development. InternalResourceViewResolver is one such convention-oriented element. It resolves a logical view name into a View object that delegates rendering responsibility to a template (usually a JSP) located in the web application's context. it does this by taking the logical view name and surrounding it with a prefix and a suffix to arrive at the path of a template that's a resource within the web application.

Let's say that we've placed all of the JSPs for the Spitter application in the /WEB-INF/views/ directory. Given that arrangement, we'll need to configure an InternalResourceViewResolver bean in spitter-servlet.xml as follows:

```
<bean class=
"org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/views/"/>
<property name="suffix" value=".jsp"/>
</bean>
```

When DispatcherServlet asks InternalResourceViewResolver to resolve a view, it takes the logical view name, prefixes it with /WEB-INF/views/ and suffixes it with .jsp. The result is the path of a JSP that will render the output. Internally, InternalResourceViewResolver then hands that path over to a View object that dispatches the request to the JSP. So, when HomeController returns home as the logical view name, it'll end up being resolved to the path /WEB-INF/views/home.jsp.

By default the View object that InternalResourceViewResolver creates is an instance of InternalResourceView, which simply dispatches the request to the JSP for rendering. But since home.jsp uses some JSTL tags, we may choose to replace InternalResourceView with JstlView by setting the viewClass property as follows:

```
<bean class=
"org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="viewClass"
value="org.springframework.web.servlet.view.JstlView" />
<property name="prefix" value="/WEB-INF/views/"/>
```

```
<property name="suffix" value=".jsp"/>
</bean>
```

JstlView dispatches the request to JSP, just like InternalResourceView. But it also exposes JSTL-specific request attributes so that you can take advantage of JSTL's internationalization support.

Although we won't delve into the details of FreeMarkerViewResolver, Jasper ReportsViewResolver, VelocityViewResolver, VelocityLayoutViewResolver, or XsltViewResolver, they're all similar to InternalResourceViewResolver in that they resolve views by adding a prefix and a suffix to the logical view name to find a view template. Once you know how to use InternalResourceViewResolver, working with those other view resolvers should feel natural.

Using InternalResourceViewResolver to resolve to JSP views is fine for a simple web application with an uncomplicated look and feel. But websites often have interesting user interfaces with some common elements shared between pages. For those kinds of sites, a layout manager such as Apache Tiles is in order. Let's see how to configure Spring MVC to resolve Tiles layout views.

## RESOLVING TILES VIEWS

Apache Tiles is a templating framework for laying out pieces of a page as fragments that are assembled into a full page at runtime. Although it was originally created as part of the Struts framework, Tiles proved to be useful with other web frameworks. In fact, we'll use it with Spring MVC to lay out the look and feel of the Spitter application.

To use Tiles views in Spring MVC, the first thing to do is to register Spring's TilesViewResolver as a <bean> in spitter-servlet.xml:

```
<beanclass="org.springframework.web.servlet.view.tiles2.TilesViewResolver"/>
```

This modest <bean> declaration sets up a view resolver that attempts to find views that are Tiles template definitions where the logical view name is the same as the Tiles definition name.

What's missing here is how Spring knows about Tiles definitions. By itself, TilesViewResolver doesn't know anything about any Tiles definitions, but instead relies on a TilesConfigurer to keep track of that information. So we'll need to add a TilesConfigurer bean to spitter-servlet.xml:

```
<bean
class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
<property name="definitions">
<list>
<value>/WEB-INF/viewsviews.xml</value>
</list>
</property>
</bean>
```

TilesConfigurer loads one or more Tiles definition files and make them available for TilesViewResolver to resolve views from. For the Spitter application we're going to have a few Tiles definition files, all named views.xml, spread around under the /WEB-INF/views folder. So we wire /WEB-INF/views/**/views.xml into the definitions property. The Ant-style ** pattern indicates that the entire directory hierarchy under /WEB-INF/views should be searched for files named views.xml.

As for the contents of views.xml files, we'll build them up throughout this chapter, starting with just enough to render the home page. The following views.xml file defines the home tile definition as well as a common template definition to be used by other tile definitions.

```
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
"http://tiles.apache.org/dtds/tiles-config_2_1.dtd">
<tiles-definitions>
<definition name="template"
template="/WEB-INF/views/main_template.jsp">
<put-attribute name="top"
value="/WEB-INF/views/tiles/spittleForm.jsp" />
<put-attribute name="side"
value="/WEB-INF/views/tiles/signinsignup.jsp" />
</definition>
<definition name="home" extends="template">
<put-attribute name="content" value="/WEB-INF/views/home.jsp" />
</definition>
</tiles-definitions>
```

The home definition extends the template definition, using home.jsp as the JSP that renders the main content of the page, but relying on template for all of the common features of the page.

It's the home template that TilesViewResolver will find when it tries to resolve the logical view name returned by HomeController's showHomePage() methods. DispatcherServlet will send the request to Tiles to render the results using the home definition.

## Defining the home page view

As you can see from listing, the home page is made up of several distinct pieces. The main_template.jsp file describes the common layout for all pages in the Spitterapplication, while home.jsp displays the main content for the home page. Plus, spittleForm.jsp and signinsignup.jsp provide some additional common elements.

For now we'll focus on home.jsp, as it's most pertinent to our discussion of displaying the home page. This JSP is where the home page request finishes its journey. It picks up the list of Spittles that HomeController placed into the model and renders them to be displayed in the user's browser. The following shows what home.jsp is made of.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="t" uri="http://tiles.apache.org/tags-tiles"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<div>
<h2>A global community of friends and strangers spitting out their
inner-most and personal thoughts on the web for everyone else to
see.</h2>
<h3>Look at what these people are spitting right now...</h3>
<olclass="spittle-list">
<c:forEach var="spittle" items="${spittles}">
<s:url value="/spitters/{spitterName}"
var="spitter_url" >
<s:param name="spitterName"

</s:url>
<li>
value="${spittle.spitter.username}" />
<span class="spittleListImage">
<img src=

"http://s3.amazonaws.com/spitterImages/${spittle.spitter.id}.jpg"
width="48"
border="0"
align="middle"
onError=
"this.src='<s:url value="/resources/images"/>/spitter_avatar.png';"/>
</span>
<span class="spittleListText">
<a href="${spitter_url}">
<c:out value="${spittle.spitter.username}" /></a>
- <c:out value="${spittle.text}" /><br/>
<small><fmt:formatDate value="${spittle.when}"
pattern="hh:mma MMM d, yyyy" /></small>
</span>
      </li>
</c:forEach>
</ol>
</div>
```

Aside from a few friendly messages at the beginning, the crux of home.jsp is contained in the<c:forEach> tag, which cycles through the list of Spittles, rendering the details of each one as it goes. Since the Spittles were placed into the model with the key spittles, the list is referenced in the JSP using ${spittles}.

## THE MODEL AND REQUEST ATTRIBUTES: THE INSIDE STORY

It's not obvious, but ${spittles} in home.jsp refers to a servlet request attribute named spittles. After HomeController finished its work and before home.jsp was called into action, DispatcherServlet copied all of the members of the model into request attributes with the same name.

Take notice of the <s:url> tag near the middle. We use this tag to create a servlet context–relative URL to the Spitter that authored each Spittle. The <s:url> tag is new to Spring 3.0 and works much like JSTL's <c:url> tag.

The main difference between Spring's <s:url> and JSTL's <c:url> is that <s:url> supports parameterized URL paths. In this case, the path is parameterized with the Spitter's username. For example, if the Spitter's username is habuma and the servlet context name is Spitter, then the resulting path will be /Spitter/spitters/habuma.

When rendered, this JSP along with the other JSPs in the same Tiles definition will display the Spitter application's home page.

At this point, we've written our first Spring MVC controller, configured a view resolver, and defined a basic JSP view to display the results of invoking the controller.

There's one tiny problem, though. An exception is waiting to happen in Home-Controller because DispatcherServlet's Spring application context won't know where to find a SpitterService bean. Fortunately, it's an easy fix.

## Rounding out the Spring application context

As I mentioned earlier, DispatcherServlet loads its Spring application context from a singleXML file whose name is based on its <servlet-name>. But what about the other beans we've declared in previous chapters, such as the SpitterService bean? If DispatcherServlet is going to load its beans from a file named spitter-servlet.xml, then won't we need to declare those other beans in spitter-servlet.xml?

In the earlier chapters we've split our Spring configuration across multiple XML files: one for the service layer, one for the persistence layer, and another for the data source configuration. Although not strictly required, it's a good idea to organize our Spring configuration across multiple files. With that in mind, it makes sense to put all of the web layer configuration in spitter-servlet.xml, the file loaded by Dispatcher-Servlet. But we still need a way to load the other configuration files.

That's where ContextLoaderListener comes into play. ContextLoaderListener is a servlet listener that loads additional configuration into a Spring application context alongside the application context created by DispatcherServlet. To use Context-LoaderListener, add the following <listener> declaration to the web.xml file:

```
<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>
```

We also need to tell ContextLoaderListener which Spring configuration file(s) it should load. If not specified otherwise, the context loader will look for a Spring configuration file at /WEB-INF/applicationContext.xml. But this single file doesn't lend itself to breaking up the application context into several pieces. So we'll need to override this default.

To specify one or more Spring configuration files for ContextLoaderListener to load, set the contextConfigLocation parameter in the servlet context:

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
/WEB-INF/spitter-security.xml
classpath:service-context.xml
classpath:persistence-context.xml
classpath:dataSource-context.xml
</param-value>
</context-param>
```

The contextConfigLocation parameter is specified as a list of paths. Unless specified otherwise, the paths are relative to the application root. But since our Spring configuration is split across multiple XML files that are scattered across several JAR files in the web application, we've prefixed some of them with classpath: to load them as resources from the application classpath and others with a path local to the web application.

You'll recognize that we've included the Spring configuration files that we created in previous chapters. You may also notice a few extra configuration files that we've not covered yet. Don't worry...we'll get to those in later chapters.

Now we have our first controller written and ready to serve requests for the Spitter application's home page. If all we needed is a home page, we'd be done. But there's more to Spitter than just the home page, so let's continue building out the application. The next thing we'll try is to write a controller that can handle input.

## Handling controller input

HomeController had it easy. It didn't have to deal with user input or any parameters. It just handled a basic request and populated the model for the view to render. It couldn't have been much simpler.

But not all controllers live such simple lives. Controllers are often asked to perform some logic against one or more pieces of information that are passed in as URL parameters or as form data. Such is the case for both SpitterController and SpittleController. These two controllers will handle several kinds of requests, many of which take input of some kind.

One example of how SpitterController will handle input is in how it supports displaying a list of Spittles for a given Spitter. Let's drive out that functionality now to see how to write controllers that process input.

Writing a controller that processes input

One way that we could implement SpitterController is to have it respond to a URL with the Spitter's username as a request query parameter. For example, http://localhost:8080/spitter/spitters/spittles?spitter=habuma could be the URL for displaying all of the Spittles for a Spitter whose username is habuma.

The following shows an implementation of SpitterController that can respond to this kind of request.

```
package com.habuma.spitter.mvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.service.SpitterService;
import static org.springframework.web.bind.annotation.RequestMethod.*;
@Controller
@RequestMapping("/spitter")
public class SpitterController {
private final SpitterService spitterService;
@Inject
//Root URL path
public SpitterController(SpitterService spitterService) {
this.spitterService = spitterService;
}
//Handle GET requests
@RequestMapping(value="/spittles", method=GET)
public String listSpittlesForSpitter(
//for /spitter/spittles
@RequestParam("spitter") String username, Model model) {
Spitter spitter = spitterService.getSpitter(username);
model.addAttribute(spitter);
//Fill model
model.addAttribute(spitterService.getSpittlesForSpitter(username));
}
}
return "spittles/list";
```

As you can see, we've annotated SpitterController with @Controller and @Request-Mapping at the class level. As we've already discussed, @Controller is a clue to <context:component-scan> that this class should be automatically discovered and registered as a bean in the Spring application context.

You'll also notice that SpitterController is annotated with @RequestMapping at the class level. In HomeController we used @RequestMapping on the showHomePage() handler method, but this class-level use of @RequestMapping is different.

As used here, the class-level @RequestMapping defines the root URL path that this controller will handle. We'll ultimately have several handler methods in Spitter-Controller, each handling different types of requests. But here @RequestMapping is saying that all of those requests will have paths that start with /spitters.

Within SpitterController we currently have a single method: listSpittlesForSpitter(). Like any good handler method, this method is annotated with @Request-Mapping. It's not dramatically different from the one we used in HomeController. But there's more to this @RequestMapping than meets the eye.

Method-level @RequestMappings narrow the mapping defined by any class-level @RequestMapping. Here, SpitterController is mapped to /spitters at the class level and to /spittles at the method level. Taken together, that means that listSpittlesForSpitter() handles requests for /spitters/spittles. Moreover, the method attribute is set to GET indicating that this method will only handle HTTP GET requests for /spitters/spittles.

The listSpittlesForSpitter() method takes a Stringusername and a Model object as parameters.

The username parameter is annotated with@RequestParam("spitter") to indicate that it should be given the value of the spitter query parameter in the request. listSpittlesForSpitter() will use that parameter to look up the Spitter object and its list of Spittles.

## Do I really need @RequestParam?

The @RequestParam annotation isn't strictly required. @RequestParam is useful for binding query parameters to method parameters where the names don't match. As a matter of convention, any parameters of a handler method that aren't annotated otherwise will be bound to the query parameter of the same name.

In the case of listSpittlesForSpitter(), if the parameter were named spitter or if the query parameter were called username, then we could leave the @RequestParam annotation off. @RequestParam also comes in handy when you compile your Java code without debugging information compiled in. In that circumstance, the name of the method parameter is lost and so there's no way to bind the query parameter to the method parameter by convention. For that reason, it's probably best to always use @RequestParam and not rely too heavily on the convention.

## Handling controller input

When we wrote HomeController, we passed in a Map<String, Object> to represent the model. But here we're using a new Model parameter.

The truth be known, the object passed in as a Model likely is a Map<String,Object> under the covers. But Model provides a few convenient methods for populating the model, such as addAttribute(). The addAttribute() method does pretty much the same thing as Map's put() method, except that it figures out the key portion of the map on its own.

When adding a Spitter object to the model, addAttribute() gives it the name spitter, a name it arrives at by applying JavaBeans property naming rules to the object's class name. When adding a List of Spittles, it tacks List to the end the member type of the List, naming the attribute spittleList.

We're almost ready to call listSpittlesForSpitter() done. We've written the SpitterController and a handler method. All that's left is to write the view that will display that list of Spittles.

## Rendering the view

When the list of Spittles is displayed to the user, we don't need much different than what we did for the home page. We just need to show the name of the Spitter (so that it's clear who the list of Spittles belongs to) and then list each Spittle.

To enable that, we first need to create a new Tiles definition. listSpittlesForSpitter() returns spittles/list as its logical view name, so the following Tile definition should do the trick:

```
<definition name="spittles/list" extends="template">
<put-attribute name="content"
value="/WEB-INF/views/spittles/list.jsp" />
</definition>
```

Just like the home Tile, this one adds another JSP page to the content attribute to be rendered within main_template.jsp. The list.jsp file used to display the list of Spittles is shown next.

```
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<div>
<h2>Spittles for ${spitter.username}</h2>
<table cellspacing="15">
<c:forEach items="${spittleList}" var="spittle">
<tr>
<td>
<img src="<s:url value="/resources/images/spitter_avatar.png"/>"
width="48" height="48" /></td>

<td>
<a href="<s:url value="/spitters/${spittle.spitter.username}"/>">
${spittle.spitter.username}</a>
<c:out value="${spittle.text}" /><br/>
<c:out value="${spittle.when}" />
</td>
</tr>
</c:forEach>
</table>
</div>
```

Aesthetics aside, this JSP does what we need. Near the top, it displays a header indicating who the list of Spittles belongs to. This header references the username property of the Spitter object that listSpittlesForSpitter() placed into the model with ${spitter.username}.

The better part of this JSP iterates through the list of Spittles, displaying their details. The JSTL <c:forEach> tag's items attribute references the list with ${spittleList}—the name that Model's addAttribute() gave it.

One minor thing to take note of is that we're using a hardcoded reference to spitter_avatar.png as the user's profile image. We'll see how to let the user upload an image to their profile.

The result of list.jsp, as rendered in the context of the spittles/list view.

Processing forms.

But first, we need to create a way for users to register to the application. In doing so, we'll get a chance to write a controller that handles form submissions.

## Processing forms

Working with forms in a web application involves two operations: displaying the form and processing the form submission. Therefore, in order to register a new Spitter in our application, we're going to need to add two handler methods to SpitterController to handle each of the operations. Since we're going to need the form in the browser before we can submit it, we'll start by writing the handler method that displays the registration form.

## Displaying the registration form

When the form is displayed, it'll need a Spitter object to bind to the form fields. Since this is a new Spitter that we're creating, a newly constructed, uninitialized Spitter object will be perfect. The following createSpitterProfile() handler method will create a Spitter object and place it in the model.

```
@RequestMapping(method=RequestMethod.GET, params="new")
public String createSpitterProfile(Model model) {
model.addAttribute(new Spitter());return "spitters/edit";
}
```

As with other handler methods, createSpitterProfile() is annotated with @RequestMapping. But, unlike previous handler methods, this one doesn't specify a path. Therefore, this method handles requests for the path specified in the class-level @RequestMapping—/spitters in the case of SpitterController.

What the method's @RequestMapping does specify is that this method will handle HTTP GET requests only. What's more, note the params attribute, which is set to new.

This means that this method will only handle HTTP GET requests for /spitters if the request includes a new query parameter. Figure 7.6 illustrates the kind of URL that createSpitterProfile will handle.

As for the inner workings of createSpitterProfile(), it simply creates a new instance of a Spitter and adds it to the model. It then wraps up by returning spitters/edit as the logical name of the view that will render the form.

Speaking of the view, let's create it next

DEFINING THE FORM VIEW

As before, the logical view name returned from createSpitterProfile() will ultimately be mapped to a Tiles definition for rendering the form to the user. So we need to add a tile definition named spitters/edit to the Tiles configuration file. The following <definition> entry should do the trick.

```
<definition name="spitters/edit" extends="template">
<put-attribute name="content"
value="/WEB-INF/views/spitters/edit.jsp" />
</definition>
```

As before, the content attribute is where the main content of the page will go. In this case, it's the JSP file at /WEB-INF/views/spitters/edit.jsp, as shown next.

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
```

```
<div>
<h2>Create a free Spitter account</h2>
<sf:form method="POST" modelAttribute="spitter">
<fieldset>
<table cellspacing="0">
<tr>
<th><label for="user_full_name">Full name:</label></th>
<td><sf:input path="fullName" size="15" id="user_full_name"/></td>
</tr>
<tr>
<th><label for="user_screen_name">Username:</label></th>
<td><sf:input path="username" size="15" maxlength="15"
id="user_screen_name"/>
</tr>
<tr>
<small id="username_msg">No spaces, please.</small>

</td>
<th><label for="user_password">Password:</label></th>
<td><sf:password path="password" size="30"
showPassword="true"
id="user_password"/> <small>6 characters or more (be tricky!)</small>
</td>
</tr>

<tr>
<th><label for="user_email">Email Address:</label></th>

<td><sf:input path="email" size="30"
</tr>
<tr>
id="user_email"/>
<small>In case you forget something</small>
</td>
<th></th>
<td>
<sf:checkbox path="updateByEmail"
id="user_send_email_newsletter"/>
<label for="user_send_email_newsletter"
```

```
>Send me email updates!</label>


</td>
</tr>
</table>
</fieldset>
</sf:form>
</div>
```

What makes this JSP file different than the others we've created so far is that it uses Spring's form binding library. The <sf:form> tag binds the Spitter object (identified by themodelAttribute attribute) that createSpitterProfile() placed into the model to the various fields in the form.

The <sf:input>, <sf:password>, and <sf:checkbox> tags each have a path attribute that references the property of the Spitter object that the form is bound to.

When the form is submitted, whatever values these fields contain will be placed into a Spitter object and submitted to the server for processing.

Note that the <sf:form> specifies that it'll be submitted as an HTTP POST request. What it doesn't specify is the URL. With no URL specified, it'll be submitted back to /spitters, the same URL path that displayed the form. That means that the next thing to do is to write another handler method that accepts POST requests for /spitters.

## Processing form input

After the form is submitted, we'll need a handler method that takes a Spitter object (populated with data from the form) and saves it. Then, the last thing it should do is redirect to the user's profile page. The following listing shows addSpitterFromForm(), a method that processes the form submission.

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
BindingResult bindingResult) {
if(bindingResult.hasErrors()) {
return "spitters/edit";
}
spitterService.saveSpitter(spitter);
}
return "redirect:/spitters/" + spitter.getUsername();
```

Note that the addSpitterFromForm() method is annotated with an @RequestMapping annotation that isn't much different than the @RequestMapping that adorns the createSpitterProfile() method. Neither specify a URL path, meaning that both handle requests for /spitters. The difference is that where createSpitterProfile() handles GET requests, addSpitterFromForm() handles POST requests. That's perfect, since that's how the form will be submitted.

And when that form is submitted, the fields in the request will be bound to the Spitter object that's passed in as an argument to addSpitterFromForm(). From there, it's sent to the SpitterService's saveSpitter() method to be stored away in the database.

You may have also noticed that the Spitter parameter is annotated with @Valid. This indicates that the Spitter should pass validation before being passed in. We'll talk about validation in the next section.

Like the handler methods we've written before, this one ends by returning a String to indicate where the request should be sent next. This time, instead of specifying a logical view name, we're returning a special redirect view. The redirect: prefix signals that the request should be redirected to the path that it precedes. By redirecting to another page, we can avoid duplicate submission of the form if the user clicks the Refresh button in their browser.

As for the path that it's redirecting to, it'll take the form of /spitters/{username} where {username} represents the username of the Spitter that was just submitted. For example, if the user registered under the name habuma, then they'd be redirected to /spitters/habuma after the form submission.

## HANDLING REQUESTS WITH PATH VARIABLES

The big question is what will respond to requests for /spitters/{username}? Actually, that's another handler method that we'll add to SpitterController:

```
@RequestMapping(value="/{username}", method=RequestMethod.GET)
public String showSpitterProfile(@PathVariable String username,
Model model) {
model.addAttribute(spitterService.getSpitter(username));
return "spitters/view";
}
```

The showSpitterProfile() method isn't too dissimilar from the other handler methods we've seen. It's given a String parameter containing a username and uses it to retrieve a Spitter object. It then places that Spitter into the model and wraps up by returning the logical name of the view that will render the output.

But by now you've probably noticed a few things that make showSpitterProfile() different. First, the value attribute in the @RequestMapping contains some strange looking curly braces. And the username parameter is annotated with @PathVariable.

Those two things work together to enable the showSpitterProfile() method to handle requests whose URLs have parameters embedded in their path. The {username} portion of the path is actually a placeholder that corresponds to the username method parameter that's annotated with@PathVariable. Whatever value is in that location in a request's path will be passed in as the value of username.

For example, if the request path is /username/habuma, then habuma will be passed in to showSpitterProfile() for the username.

But we still have some unfinished business with regard to addSpitterFromForm(). You've probably noticed that addSpitterFromForm()'s Spitter parameter is annotated with @Valid. Let's see how this annotation can be used to keep bad data from being submitted in a form.

## Validating input

When a user registers with the Spitter application, there are certain requirements that we'd like to place on that registration. Specifically, a new user must give us their full name, email address, a username, and a password. Not only that, but the email address can't be just freeform text—it must look like an email address. Moreover, the password should be at least six characters long.

The @Valid annotation is the first line of defense against faulty form input. @Valid is actually a part of the JavaBean validation specification.4 Spring 3 includes support for JSR-303, and we're using @Valid here to tell Spring that the Spitter object should be validated as it's bound to the form input.

Should anything go wrong while validating the Spitter object, the validation error will be carried to the addSpitterFromForm() method via the BindingResult that's passed in on the second parameter. If the BindingResult's hasErrors() method returns true, then that means that validation failed. In that case, the method will return spitters/edit as the view name to display the form again so that the user can correct any validation errors.

But how will Spring know the difference between a valid Spitter and an invalid Spitter?

## DECLARING VALIDATION RULES

Among other things, JSR-303 defines a handful of annotations that can be placed on properties to specify validation rules. We can use these annotations to define what "valid" means with regard to a Spitter object. The following shows the properties of the Spitter class that are annotated with validation annotations.

```
@Size(min=3, max=20, message=
"Username must be between 3 and 20 characters long.")
@Pattern(regexp="^[a-zA-Z0-9]+$",
message="Username must be alphanumeric with no spaces")
private String username;
@Size(min=6, max=20,
message="The password must be at least 6 characters long.")
private String password;

@Size(min=3, max=50, message=
"Your full name must be between 3 and 50 characters long.")
private String fullName;
@Pattern(regexp="[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
message="Invalid email address.")
private String email;
```

The first three properties in listing are annotated with JSR-303's @Size annotation to validate that those fields meet certain expectations on their length. The username property must be at least 3 and at most 20 characters long, whereas the fullName property must be between 3 and 50 characters in length. As for the password property, it must be at least 6 characters long and not exceed 20 characters.

To make sure that the value given to the email property fits the format of an email address, we've annotated it with @Pattern and specified a regular expression to match it against in the regexp attribute.[5] Similarly, we've used @Pattern on the username property to ensure that the username is only made up of alphanumeric characters with no spaces.

In all of the validation annotations, we've set the message attribute with the message to be displayed in the form when validation fails so that the user knows what needs to be corrected.

With these annotations in place, when a user submits a registration form to SpitterController's addSpitterFromForm() method, the values in the Spitter object's fields will be weighed against the validation annotations. If any of those rules are broken, then the handler method will send the user back to the form to fix the problem.

When they arrive back at the form, we'll need a way to tell them what the problem was. So we're going to have to go back to the form JSP and add some code to display the validation messages.

## DISPLAYING VALIDATION ERRORS

Recall that the BindingResult passed in as a parameter to addSpitterFromForm() knew whether the form had any validation errors. And we were able to ask if there were any errors by calling its hasErrors() method. But what we didn't see was that the actual error messages are also in there, associated with the fields that failed validation.

One way of displaying those errors to the users is to access those field errors through BindingResult's getFieldError() method. But a much better way is to use Spring's form binding JSP tag library to display the errors. More specifically, the <sf:errors> tag can render field validation errors. All we need to do is sprinkle a few <sf:errors> tags around our form JSP.

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<div>
<h2>Create a free Spitter account</h2>
<sf:form method="POST" modelAttribute="spitter"
enctype="multipart/form-data">
<fieldset>
<table cellspacing="0">
<tr>
<th><sf:label path="fullName">Full name:</sf:label></th>
<td><sf:input path="fullName" size="15" /><br/>
<sf:errors path="fullName" cssClass="error" />
</td>
</tr>
<tr>
<th><sf:label path="username">Username:</sf:label></th>
<td><sf:input path="username" size="15" maxlength="15" />
<small id="username_msg">No spaces, please.</small><br/>

</tr>
<tr>
<sf:errors path="username" cssClass="error" />
</td>
<th><sf:label path="password">Password:</sf:label></th>
<td><sf:password path="password" size="30"
showPassword="true"/>
<small>6 characters or more (be tricky!)</small><br/>
<sf:errors path="password" cssClass="error" />
</td>
</tr>
<tr>
```

```
<th><sf:label path="email">Email Address:</sf:label></th>
<td><sf:input path="email" size="30"/>
<small>In case you forget something</small><br/>
<sf:errors path="email" cssClass="error" />
</td>
</tr>
<tr>
<th></th>
<td>
<sf:checkbox path="updateByEmail"/>
<sf:label path="updateByEmail"
>Send me email updates!</sf:label>
</td>
</tr>
<tr>
<th><label for="image">Profile image:</label></th>
<td><input name="image" type="file"/>
</tr>
<tr>
<th></th>
<td><input name="commit" type="submit"
</tr>
</table>
</fieldset>
</sf:form>
</div>
value="I accept. Create my account." /></td>
```

The <sf:errors> tag's path attribute specifies the form field for which errors should be displayed. For example, the following <sf:errors> displays errors (if there are any) for the field whose name is fullName:

<sf:errors path="fullName" cssClass="error" />

If there are multiple errors for a single field, they'll all be displayed, separated by an HTML <br/> tag. If you'd rather have them separated some other way, then you can use the delimiter attribute. The following <sf:errors> snippet uses delimiter to separate errors with a comma and a space:

<sf:errors path="fullName" delimiter=", " cssClass="error" />

Note that there are four <sf:errors> tags in this JSP, one on each of the fields for which we declared validation rules. The cssClass attribute refers to a class that's declared in CSS to display in red so that it catches the user's attention.

With these in place, errors will be displayed on the page if any validation errors occur.

As you can see, validation errors are displayed on a per-field basis. But if you'd refer to display all of the errors in one place (perhaps at the top of the form), you'll only need a single <sf:errors>

tag, with its path attribute set to *:

<sf:errors path="*" cssClass="error" />

Now you know how to write controller handler methods that process form data. The one thing that's common about all of the form fields we've seen thus far is that they're textual data and were probably entered into the form by the user typing on a key-board. But what if the thing that the user needs to submit in a form can't be banged out on a keyboard? What if the user needs to submit an image or some other kind of file?

The key to the Spring transaction abstraction is the notion of a transaction strategy. A transaction strategy is defined by the org.springframework.transaction.PlatformTransactionManager interface, shown below:

```
public interface PlatformTransactionManager {


    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;


    void commit(TransactionStatus status) throws TransactionException;


    void rollback(TransactionStatus status) throws TransactionException;
}
```

This is primarily an SPI interface, although it can be used programmatically. Note that in keeping with the Spring Framework's philosophy, PlatformTransactionManager is an interface, and can thus be easily mocked or stubbed as necessary. Nor is it tied to a lookup strategy such as JNDI: PlatformTransactionManager implementations are defined like any other object (or bean) in the Spring Framework's IoC container. This benefit alone makes it a worthwhile abstraction even when working with JTA: transactional code can be tested much more easily than if it used JTA directly.

Again in keeping with Spring's philosophy, the TransactionException that can be thrown by any of the PlatformTransactionManager interface's methods is unchecked (i.e. it extends the java.lang.RuntimeException class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle TransactionException. The salient point is that developers are not forced to do so.

The getTransaction(..) method returns a TransactionStatus object, depending on a TransactionDefinition parameter. The returned TransactionStatus might represent a new or existing transaction (if there were a matching transaction in the current call stack - with the implication being that (as with J2EE transaction contexts) a TransactionStatus is associated with a thread of execution).

The `TransactionDefinition` interface specifies:

- **Isolation**: the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- **Propagation**: normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case); or suspending the existing transaction and creating a new transaction. *Spring offers all of the transaction propagation options familiar from EJB CMT*.
- **Timeout**: how long this transaction may run before timing out (and automatically being rolled

back by the underlying transaction infrastructure).

- **Read-only status**: a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

These settings reflect standard transactional concepts. If necessary, please refer to a resource discussing transaction isolation levels and other core transaction concepts because understanding such core concepts is essential to using the Spring Framework or indeed any other transaction management solution.

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

```
public interface TransactionStatus {


    boolean isNewTransaction();


    void setRollbackOnly();


    boolean isRollbackOnly();
}
```

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct PlatformTransactionManager implementation is absolutely essential. In good Spring fashion, this important definition typically is made using via Dependency Injection.

PlatformTransactionManager implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, etc The following examples from the dataAccessContext-local.xml file from Spring's jPetStore sample application show how a local PlatformTransactionManager implementation can be defined. (This will work with plain JDBC.)

We must define a JDBC DataSource, and then use the Spring DataSourceTransactionManager, giving it a reference to the DataSource.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">

  <property name="driverClassName" value="${jdbc.driverClassName}" />

  <property name="url" value="${jdbc.url}" />

  <property name="username" value="${jdbc.username}" />

  <property name="password" value="${jdbc.password}" />

</bean>
```

The related `PlatformTransactionManager` bean definition will look like this:

```
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>
```

If we use JTA in a J2EE container, as in the 'dataAccessContext-jta.xml' file from the same sample application, we use a container DataSource, obtained via JNDI, in conjunction with Spring's JtaTransactionManager. The JtaTransactionManager doesn't need to know about the DataSource, or any other specific resources, as it will use the container's global transaction management infrastructure.

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

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

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

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

xsi:schemaLocation="

       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd

       http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">


  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>


  <bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager" />


  <!-- other <bean/> definitions here -->


</beans>
```

We can also use Hibernate local transactions easily, as shown in the following examples from the Spring Framework's PetClinic sample application. In this case, we need to define a Hibernate LocalSessionFactoryBean, which application code will use to obtain Hibernate Session instances.

The DataSource bean definition will be similar to the one shown previously (and thus is not shown). If the DataSource is managed by the JEE container it should be non-transactional as the Spring Framework, rather than the JEE container, will manage transactions.

The 'txManager' bean in this case is of the HibernateTransactionManager type. In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the SessionFactory.

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

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

  <property name="mappingResources">

    <list>


<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</v
alue>

    </list>

  </property>

  <property name="hibernateProperties">

    <value>

         hibernate.dialect=${hibernate.dialect}

       </value>

  </property>

</bean>
```

```
<bean id="txManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">

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

</bean>
```

With Hibernate and JTA transactions, we can simply use the JtaTransactionManager as with JDBC or any other resource strategy.

<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

Note that this is identical to JTA configuration for any resource, as these are global transactions, which can enlist any transactional resource.

In all these cases, application code will not need to change at all. We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

It should now be clear how different transaction managers are created, and how they are linked to related resources which need to be synchronized to transactions (i.e. DataSourceTransactionManager to a JDBC DataSource, HibernateTransactionManager to a Hibernate SessionFactory, etc). There remains the question however of how the application code, directly or indirectly using a persistence API (JDBC, Hibernate, JDO, etc), ensures that these resources are obtained and handled properly in terms of proper creation/reuse/cleanup and trigger (optionally) transaction synchronization via the relevant PlatformTransactionManager.

## High-level approach

The preferred approach is to use Spring's highest level persistence integration APIs. These do not replace the native APIs, but internally handle resource creation/reuse, cleanup, optional transaction synchronization of the resources and exception mapping so that user data access code doesn't have to worry about these concerns at all, but can concentrate purely on non-boilerplate persistence logic. Generally, the same template approach is used for all persistence APIs, with examples including the JdbcTemplate, HibernateTemplate, and JdoTemplate classes (detailed in subsequent chapters of this reference documentation.

## Low-level approach

At a lower level exist classes such as `DataSourceUtils` (for JDBC), `SessionFactoryUtils` (for Hibernate), `PersistenceManagerFactoryUtils` (for JDO), and so on. When it is preferable for application code to deal directly with the resource types of the native persistence APIs, these classes ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions which happen in the process are properly mapped to a consistent API.

For example, in the case of JDBC, instead of the traditional JDBC approach of calling the `getConnection()` method on the `DataSource`, you would instead use Spring's `org.springframework.jdbc.datasource.DataSourceUtils` class as follows:

Connection conn = DataSourceUtils.getConnection(dataSource);

If an existing transaction exists, and already has a connection synchronized (linked) to it, that instance will be returned. Otherwise, the method call will trigger the creation of a new connection, which will be (optionally) synchronized to any existing transaction, and made available for subsequent reuse in that same transaction. As mentioned, this has the added advantage that any `SQLException` will be wrapped in a Spring Framework `CannotGetJdbcConnectionException` - one of the Spring Framework's hierarchy of unchecked DataAccessExceptions. This gives you more information than can easily be obtained from the `SQLException`, and ensures portability across databases: even across different persistence technologies.

It should be noted that this will also work fine without Spring transaction management (transaction synchronization is optional), so you can use it whether or not you are using Spring for transaction management.

Of course, once you've used Spring's JDBC support or Hibernate support, you will generally prefer not to use `DataSourceUtils` or the other helper classes, because you'll be much happier working via the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring `JdbcTemplate` or `jdbc.object` package to simplify your use of JDBC, correct connection retrieval happens behind the scenes and you won't need to write any special code.

## TransactionAwareDataSourceProxy

At the very lowest level exists the `TransactionAwareDataSourceProxy` class. This is a proxy for a target `DataSource`, which wraps the target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource` as provided by a J2EE server.

It should almost never be necessary or desirable to use this class, except when existing code exists which must be called and passed a standard JDBC `DataSource` interface implementation. In that case, it's possible to still have this code be usable, but participating in Spring managed transactions. It is preferable to write your new code using the higher level abstractions mentioned above.

*Most users of the Spring Framework choose declarative transaction management. It is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.*

The Spring Framework's declarative transaction management is made possible with Spring AOP, although, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with the Spring Framework's declarative transaction management. The basic approach is similar: it is possible to specify transaction behavior (or lack of it) down to individual method level. It is possible to make a `setRollbackOnly()` call within a transaction context if necessary. The differences are:

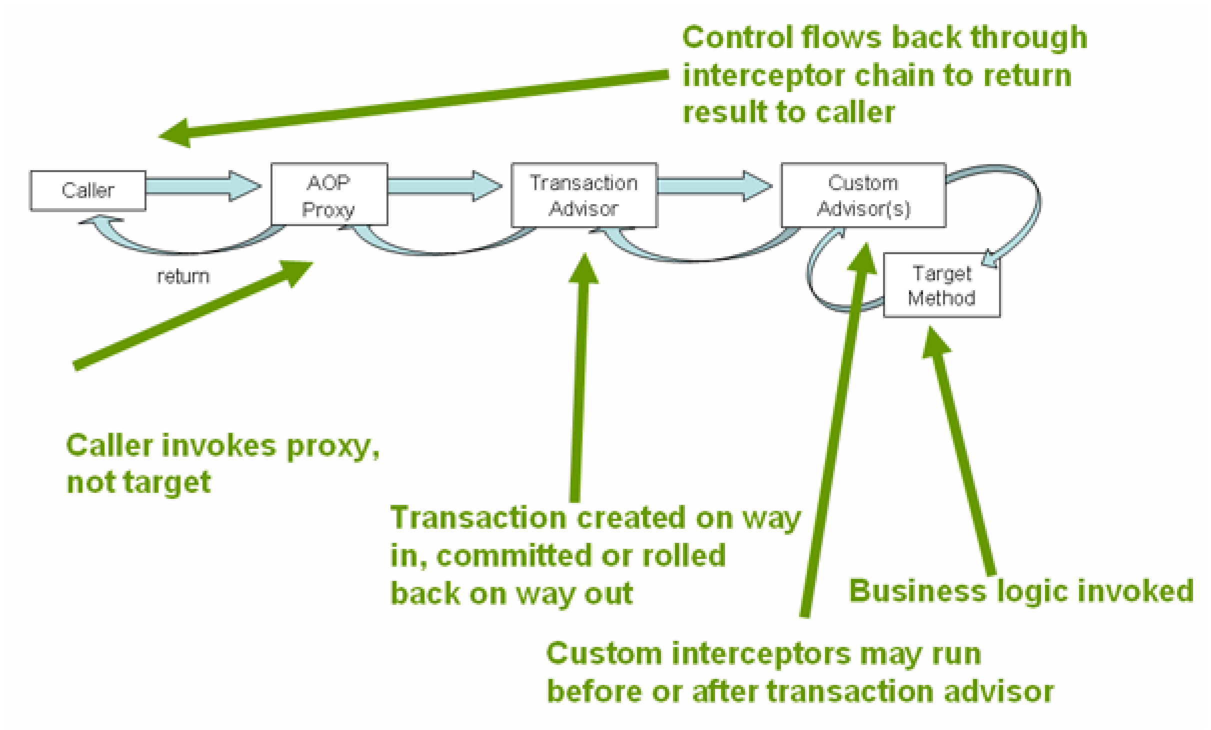
- Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only.
- The Spring Framework enables declarative transaction management to be applied to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative *rollback rules*: a feature with no EJB equivalent, which we'll discuss below. Rollback can be controlled declaratively, not merely programmatically.
- The Spring Framework gives you an opportunity to customize transactional behavior, using AOP. For example, if you want to insert custom behavior in the case of transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you have no way to influence the container's transaction management other than `setRollbackOnly().`
- The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature. Normally, we do not want transactions to span remote calls.

### Understanding the Spring Framework's declarative transaction implementation

The aim of this section is to dispel the mystique that is sometimes associated with the use of declarative transactions. It is all very well for this reference documentation simply to tell you to annotate your classes with the @Transactional annotation, add the line ('<tx:annotation-driven/>') to your configuration, and then expect you to understand how it all works. This section will explain the inner workings of the Spring Framework's declarative transaction infrastructure to help you navigate your way back upstream to calmer waters in the event of transaction-related issues.

The most important concepts to grasp with regard to the Spring Framework's declarative transaction support are that this support is enabled via AOP proxies, and that the transactional advice is driven by metadata (currently XML- or annotation-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a TransactionInterceptor in conjunction with an appropriate PlatformTransactionManager implementation to drive transactions around method invocations.

Conceptually, calling a method on a transactional proxy looks like this...


Control flows back through interceptor chain to return result to caller
Caller
AOP Proxy
Transaction Advisor
Custom Advisor(s)
Target Method
return
Caller invokes proxy, not target
Transaction created on way in, committed or rolled back on way out
Business logic invoked
Custom interceptors may run before or after transaction advisor

## First example

Consider the following interface, and its attendant implementation. (The intent is to convey the concepts, and using the rote `Foo` and `Bar` tropes means that you can concentrate on the transaction usage and not have to worry about the domain model.)

```
// the service interface that we want to make transactional


package x.y.service;


public interface FooService {


    Foo getFoo(String fooName);


    Foo getFoo(String fooName, String barName);


    void insertFoo(Foo foo);


    void updateFoo(Foo foo);
}
// an implementation of the above interface

package x.y.service;

public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }

    public Foo getFoo(String fooName, String barName) {
        throw new UnsupportedOperationException();
    }

    public void insertFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

}
```

Let's assume that the first two methods of the `FooService` interface (`getFoo(String)` and `getFoo(String, String)`) have to execute in the context of a transaction with read-only semantics, and that the other methods (`insertFoo(Foo)` and `updateFoo(Foo)`) have to execute in the context of a transaction with read-write semantics. Don't worry about taking the following configuration in all at once; everything will be explained in detail in the next few paragraphs.

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
```

```
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">


  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>


  <!-- the transactional advice (i.e. what 'happens'; see the
<aop:advisor/> bean below) -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
    <!-- the transactional semantics... -->
    <tx:attributes>
      <!-- all methods starting with 'get' are read-only -->
      <tx:method name="get*" read-only="true"/>
      <!-- other methods use the default transaction settings (see below) -
->
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>


  <!-- ensure that the above transactional advice runs for any execution
      of an operation defined by the FooService interface -->
  <aop:config>
    <aop:pointcut id="fooServiceOperation" expression="execution(*
x.y.service.FooService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
  </aop:config>


  <!-- don't forget the DataSource -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
  </bean>


  <!-- similarly, don't forget the PlatformTransactionManager -->
  <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
  </bean>
  <!-- other <bean/> definitions here -->


</beans>
```

Let's pick apart the above configuration. We have a service object (the `'fooService'` bean) that we want to make transactional. The transaction semantics that we want to apply are encapsulated in the `<tx:advice/>` definition. The `<tx:advice/>` definition reads as "*... all methods on starting with `'get'` are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics*". The `'transaction-manager'` attribute of the `<tx:advice/>` tag is set to the name of the `PlatformTransactionManager` bean that is going to actually *drive* the transactions (in this case the `'txManager'` bean).

The `<aop:config/>` definition ensures that the transactional advice defined by the `'txAdvice'` bean actually executes at the appropriate points in the program. First we define a pointcut that matches the execution of any operation defined in the `FooService` interface (`'fooServiceOperation'`). Then we associate the pointcut with the `'txAdvice'` using an advisor. The result indicates that at the execution of a `'fooServiceOperation'`, the advice defined by `'txAdvice'` will be run.

The expression defined within the `<aop:pointcut/>` element is an AspectJ pointcut expression;

A common requirement is to make an entire service layer transactional. The best way to do this is simply to change the pointcut expression to match any operation in your service layer. For example:

```
<aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(*
x.y.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
  </aop:config>
```

The above configuration is going to effect the creation of a transactional proxy around the object that is created from the `'fooService'` bean definition. The proxy will be configured with the transactional advice, so that when an appropriate method is invoked *on the proxy*, a transaction *may* be started, suspended, be marked as read-only, etc., depending on the transaction configuration associated with that method. Consider the following program that test drives the above configuration.

```
public final class Boot {


    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("context.xml", Boot.class);
        FooService fooService = (FooService) ctx.getBean("fooService");
        fooService.insertFoo (new Foo());
    }
}
```

The output from running the above program will look something like this. *(Please note that the Log4J output and the stacktrace from the `UnsupportedOperationException` thrown by the `insertFoo(..)` method of the `DefaultFooService` class have been truncated in the interest of clarity.)*

```
    <!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating
implicit proxy
        for bean 'fooService' with 0 common interceptors and 1 specific
interceptors
    <!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for
[x.y.service.DefaultFooService]

    <!-- ... the insertFoo(..) method is now being invoked on the proxy -->

[TransactionInterceptor] - Getting transaction for
x.y.service.FooService.insertFoo
    <!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name
[x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC
transaction

    <!-- the insertFoo(..) method from DefaultFooService throws an
exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether
transaction should
        rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on
x.y.service.FooService.insertFoo
        due to throwable [java.lang.UnsupportedOperationException]

   <!-- and the transaction is rolled back (by default, RuntimeException
instances cause rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on
Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after
transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException
        at
x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)
   <!-- AOP infrastructure stack trace elements removed for clarity -->
        at $Proxy0.insertFoo(Unknown Source)
        at Boot.main(Boot.java:11)
```

## Rolling back

The previous section outlined the basics of how to specify the transactional settings for the classes, typically service layer classes, in your application in a declarative fashion. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an `Exception` from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code will catch any unhandled `Exception` as it bubbles up the call stack, and will mark the transaction for rollback.

However, please note that the Spring Framework's transaction infrastructure code will, by default, *only* mark a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of `RuntimeException`. (`Errors` will also - by default - result in a rollback.) Checked exceptions that are thrown from a transactional method will *not* result in the transaction being rolled back.

Exactly which `Exception` types mark a transaction for rollback can be configured. Find below a snippet of XML configuration that demonstrates how one would configure rollback for a checked, application-specific `Exception` type.

```
<tx:advice id="txAdvice" transaction-manager="txManager">

  <tx:attributes>

         <tx:method name="get*" read-only="false" rollback-
for="NoProductInStockException"/>

         <tx:method name="*"/>

  </tx:attributes>

</tx:advice>
```

The second way to indicate to the transaction infrastructure that a rollback is required is to do so *programmatically*. Although very simple, this way is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure. Find below a snippet of code that does programmatic rollback of a Spring Framework-managed transaction:

```
public void resolvePosition() {

    try {

        // some business logic...

    } catch (NoProductInStockException ex) {

        // trigger rollback programmatically


TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

    }

}
```

You are strongly encouraged to use the declarative approach to rollback if at all possible. Programmatic rollback is available should you need it, but its usage flies in the face of achieving a clean POJO-based application model.

**Configuring different transactional semantics for different beans**

Consider the scenario where you have a number of service layer objects, and you want to apply *totally different* transactional configuration to each of them. This is achieved by defining distinct `<aop:advisor/>` elements with differing `'pointcut'` and `'advice-ref'` attribute values.

Let's assume that all of your service layer classes are defined in a root `'x.y.service'` package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in `'Service'` have the default transactional configuration, you would write the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">


    <aop:config>


        <aop:pointcut id="serviceOperation"
                    expression="execution(* x.y.service..*Service.*(..))"/>


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


    </aop:config>


    <!-- these two beans will be transactional... -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>
    <bean id="barService" class="x.y.service.extras.SimpleBarService"/>


    <!-- ... and these two beans won't -->
    <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in
the right package) -->
    <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!--
(doesn't end in 'Service') -->


    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>
    <!-- other transaction infrastructure beans such as a
PlatformTransactionManager omitted... -->


</beans>
```

Find below an example of configuring two distinct beans with totally different transactional settings.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">


    <aop:config>


        <aop:pointcut id="defaultServiceOperation"
                    expression="execution(* x.y.service.*Service.*(..))"/>


        <aop:pointcut id="noTxServiceOperation"
                    expression="execution(*
x.y.service.ddl.DefaultDdlManager.*(..))"/>


        <aop:advisor pointcut-ref="defaultServiceOperation" advice-
ref="defaultTxAdvice"/>


        <aop:advisor pointcut-ref="noTxServiceOperation" advice-
ref="noTxAdvice"/>


    </aop:config>


    <!-- this bean will be transactional (see the 'defaultServiceOperation'
pointcut) -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>


    <!-- this bean will also be transactional, but with totally different
transactional settings -->
    <bean id="anotherFooService"
class="x.y.service.ddl.DefaultDdlManager"/>
```

```
    <tx:advice id="defaultTxAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <tx:advice id="noTxAdvice">
        <tx:attributes>
            <tx:method name="*" propagation="NEVER"/>
        </tx:attributes>
    </tx:advice>

    <!-- other transaction infrastructure beans such as a
PlatformTransactionManager omitted... -->

</beans>
```

**`<tx:advice/>` settings**

This section summarises the various transactional settings that can be specified using the `<tx:advice/>` tag. The default `<tx:advice/>` settings are:

- The propagation setting is `REQUIRED`
- The isolation level is `DEFAULT`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any `RuntimeException` will trigger rollback, and any checked `Exception` will not

These default settings can, of course, be changed; the various attributes of the `<tx:method/>` tags that are nested within `<tx:advice/>` and `<tx:attributes/>` tags are summarized below:

**`<tx:method/>`** settings

| **Attribute** | **Required?** | **Default** | **Description** |
|---|---|---|---|
| `name` | Yes | | The method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, `'get*'`, `'handle*'`, `'on*Event'`, etc. |
| `propagation` | No | REQUIRED | The transaction propagation behavior |
| `isolation` | No | DEFAULT | The transaction isolation level |

| | | | |
|---|---|---|---|
| `timeout` | No | -1 | The transaction timeout value (in seconds) |
| `read-only` | No | false | Is this transaction read-only? |
| `rollback-for` | No | | The `Exception(s)` that will trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,ServletException'` |
| `no-rollback-for` | No | | The `Exception(s)` that will *not* trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,ServletException'` |

## Introduction to Lab Manual

This lab manual aims at giving complete understanding of the core concepts of the topic and applying them in the exercises.

The lab manual consists of a set of lab exercises defined chapter wise. Each exercise has a definite objective defined. These objectives map with the terminal objectives defined at the beginning of each chapter. The problem statement is defined with clear instructions.

Some exercises have specific configuration or pre-condition mentioned. Advanced lab exercises are also given for some topics.

Appendix B given at the end of lab manual contains stepwise instructions to use eclipse IDE.

## Configuration

jdk 1.6 should be installed. The lab exercises should be coded using Java language.

Specific configuration related to a particular exercise is mentioned. If it is not mentioned then the above configuration should be considered.

## Structure of Lab Manual

Lab manual consists of different sections. The explanation of each structure is given below.

### Objective

It states what you will achieve after completing a particular application. At the end of each lab exercise you should keep a track whether the objectives of that session are achieved.

### Configuration

It is optional section and is present for specific lab exercises. If absent, then the configuration mentioned earlier should be considered.

### Pre-condition

You should have understood the concepts explained in a particular chapter in the courseware thoroughly to solve the exercises mentioned.

But some lab exercises will have pre-condition explicitly mentioned.

### Problem Statement

Problem statement for each lab exercise is given. It defines clear instructions to achieve the defined objective.

## How to use the Lab Manual?

Whenever a programmer has to transform a problem statement into a program which a computer can execute, you should split the activity in the following manner:

- Read the problem statement carefully. Hint is given for some problem statements to help you to solve the problem.
- Preparation
  - Decide the User interface (CUI or GUI) before hand.
  - Write the algorithm or steps to be followed to solve the problem. You can also draw flowcharts if required.
  - The program is made modular by writing functions. So pen down expected function prototypes and arguments on paper.

  - Also decide how one module (function) would communicate with other module (function).
- Give a dry run to the algorithm written.
- Write the code.
- Execute the code.

## Coding practices

The maintenance of code is easy if the code is written using coding practices. You should follow following coding practices while solving the lab exercises in this lab manual:

- Use meaningful names for variables, functions, file.
- Use uniform notation throughout your code.
- Code should be properly indented.
- Code should be commented. While documenting your code, write the purpose of your piece of code. What task is assigned to a method, what arguments are passed to it, what it returns should be clearly stated. Write a clear comment if you have added any statements for testing purpose.

  Writing a right kind of well-documented software is an art along with your technical skills. Enough necessary documentation should be done. This makes the code easily maintainable.

# Chapter 1 – Introduction to Spring framework

## Lab Exercise - 1

### Objective

- Construct `"has-a"` relationship between classes

### Problem Statement

Create a POJO classes `Employee` and `Date`.
Date contains `day, month, year` attributes.
`Employee` contains `empno, ename` and `salary, date_OF_joining` attributes.
Data type of `date_Of_joining` class is `Date`.
Write a main method to print the entire information of employees.

# Chapter 2 – Spring Core

## Lab Exercise - 2

### Objective

- Construct the bean life cycle.

### Pre-condition

- `Employee and Date` should be created.

### Problem Statement

Construct and observe the spring bean life cycle.

## Lab Exercise - 3

### Objective

- Construct the simple bean.
- Identify how DI/IOC container works.

### Pre-condition

- `Employee and Date` should be created.

### Problem Statement

Construct a spring Test class to inject the date information inside employee and print employee information.

## Lab Exercise - 4

### Objective

- Construct the simple bean using ApplicationContext.
- Identify how DI/IOC container works.

**Pre-condition**

- `Employee and Date` should be created.

**Problem Statement**

Construct a spring Test class to inject the date information inside employee and print employee information.

## Lab Exercise - 5

**Objective**

- Injecting the dependencies by setter/getter, constructor and method.
- Identify how DI/IOC container works.

**Pre-condition**

- `Employee and Date` should be created.

**Problem Statement 1**

Construct a spring Test class to inject the date information inside employee using setter and getter and print employee information.

**Problem Statement 2**

Construct a spring Test class to inject the date information inside employee using Constructor and print employee information.

**Problem Statement 3**

Construct a spring Test class to inject the date information inside employee using method and print employee information.

## Lab Exercise - 6

**Objective**

- Construct Inheritance in Bean.
- Construct Collections in Bean.
- Check controlling the bean creation.
- Identify how DI/IOC container works.

**Pre-condition**

- `Employee and Date` should be created.

**Problem Statement 1**

Construct a spring Test class to inject the date information inside employee to check inheritance in bean and print employee information. Also check the magic of `paren`t and `abstract` special attributes in xml.

**Problem Statement 2**

Construct a spring Test class to inject the date information inside employee to check collections in bean and print employee information with List of certifications and collection of experience like (`companyName->No_of_years_exp`). Also check the magic of parent and abstract special attributes in xml.

### Problem Statement 3

Construct a spring Test class to inject the date information inside employee and print employee information. Also check the magic of prototype and singleton scope attributes in xml.

## Lab Exercise - 7

### Objective

- Construct Auto Wiring of Bean.
- Identify how DI/IOC container works.

### Pre-condition

- `Employee and Date` should be created.

### Problem Statement

Construct a spring Test class to inject the date information inside employee to bean and print employee information.

Hint: Use Auto Wire concept byType and constructor.

# Chapter 3 – Spring AOP

## Lab Exercise - 8

### Objective

- Construct the Proxy classes.
- Indentify concept of AOP
- Check the xml tags and attributes.

### Pre-condition

- `Employee and Date` should be created.

### Problem Statement 1

Construct a spring Test class to check "`cross-cutting concerns`".
Create a Logging service advice and whenever their requirement waves this advice using proxy bean class.
Create `LoggingService` and `LoggingServiceAdvice`.

### Problem Statement 2

Construct a spring Test class to check "cross-cutting concerns".
Create a Logging service advice and whenever their requirement waves this advice using proxy bean class.
Create `LoggingService` and `LoggingServiceAdvice`. Use Around advice.

# Chapter 4 – Spring MVC

## Lab Exercise - 9

### Objective

- Construct the Spring MVC application.
- Construct the web.xml and springapp-servlet.xml files.

### Pre-condition

- `Employee and Date` should be created.

### Problem Statement 1

Construct a spring MVC application for employee.
Accept the information of employee form `emp.htm` file and print list of employees using spring MVC.

# Chapter 5 – Spring-DAO

## Lab Exercise – 10

### Objective

- Construct the JDBC application without using spring-DAO.
- Access data from the tables in the MySQL database and perform operations like insert, update and delete using JDBC.

**Configuration:** MySQL should be installed.

### Pre-condition

- `employee` table should be present in the database. The table should have fields `employee_id, name, salary.`

  `Employee` classes should be created.

### Problem Statement

Construct a spring application to perform CRUD operation without using spring-DAO.

## Lab Exercise – 11

### Objective

- Construct the JDBC application using spring-DAO.
- Access data from the tables in the MySQL database and perform operations like insert, update and delete using JDBC.

**Configuration:** MySQL should be installed.

### Pre-condition

- `employee` table should be present in the database. The table should have fields `employee_id, name, salary.`
- `Employee` classes should be created.

### Problem Statement

Construct a spring application to perform CRUD operation without using spring-DAO.

## Lab Exercise – 12

### Objective

- Construct the hibernate application using spring-DAO.
- Access data from the tables in the MySQL database and perform operations like insert, update and delete using JDBC.

**Configuration:** MySQL should be installed.

### Pre-condition

- `employee` table should be present in the database. The table should have fields `employee_id, name, salary.`

Hint: Use spring-hibernate.