Hi Friends,

Just go through the following small story.. (taken from **"*You Can Win* "** - by Shiv Khera)

Once upon a time a very strong woodcutter asks for a job in a timber merchant, and he got it. The paid was really good and so were the work conditions. For that reason, the woodcutter was determined to do his best. His boss gave him an axe and showed him the area where he was supposed to work. The first day, the woodcutter brought 18 trees "Congratulations," the boss said. "Go on that way!" Very motivated for the boss' words, the woodcutter try harder the next day, but he only could bring 15 trees. The third day he try even harder, but he only could bring 10 trees. Day after day he was bringing less and less trees. The woodcutter thought that "I must be losing my strength". He went to the boss and apologized, saying that he could not understand what was going on.

The boss asked, "When was the last time you sharpened your axe?"

"Sharpen? I had no time to sharpen my axe. I have been very busy trying to cut trees.

If we are just busy in applying for jobs & work, when we will sharpen our skills to chase the job selection process?

My aim is to provide good and quality content to readers to understand easily. All contents are written and practically tested by me before publishing. If you have any query or questions regarding any article feel free to leave a comment or you can get in touch with me on venus.kumaar@gmail.com.

This is just a start, I have achieved till now is just 0.000001n% .

With Warm Regards

Venu Kumaar.S
venus.kumaar@gmail.com

Spring FrameWork

Spring Frame Work Core module

Model → Business Logic(**BL**) + Persistence Logic(**PL**)→ JavaBean/EJB Session Bean/Spring JEE Application

View → Presentation Logic(**PL**) → HTML/JSP etc.

Controller → Integration Logic(**IL**)/Connectivity Logic(**CL**) → Servlet / ServletFilter

BL → The main logic of the application based on input values.

PL → The logic that is there to interact with DB to manipulate DB data.

IL → The logic that performs the connectivity View layer & Model layer resources.

View	**Controller**	**Model (BL+PL)**		
		BL	**PL**	
JSP	→ Servlet →	JavaBean	→ DAO Class	→ DB
JSP	→ Servlet →	EJB Session Bean	→ EJB Entity Bean	→ DB
JSP	→ Servlet →	EJB Session Bean	→ DAO Class	→ DB
Struts App. (**V,C**)	→	EJB Session Bean	→ DAO /Hibernate	→ DB
Struts App. (**V,C**)	→	SpringJEEApp	→ Hibernate	→ DB
Spring MVC App (**V,C**)	→ →	SpringJEEApp	→ DAO/Hibernate	→ DB

==========

==========

Spring is an open-source framework, created by Rod Johnson.1 It was created to address the complexity of enterprise application development.

Spring makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with EJBs. However, Spring's usefulness isn't limited to server-side development.

Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.

Spring is a lightweight inversion of control and aspect-oriented container framework.

1) Lightweight :

Spring is lightweight in terms of both size and overhead. The entire Spring framework can be distributed in a single JAR file that weighs in at just over 1 MB. And the processing overhead required by Spring is negligible.

Spring is a non-invasive framework. This is the key departure from most previous frameworks. It means the application classes no need to extend or implement spring framework API classes or interfaces respectively.

2) Inversion of control :

Spring promotes loose coupling through a technique known as inversion of control (IoC). When IoC is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves.

You can think of IoC as JNDI in reverse, instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked.

3) Aspect-oriented :

Spring comes with rich support for aspect-oriented programming that enables cohesive development by separating application business logic from system services (such as auditing and transaction management).

Application objects do what they're supposed to do, perform business logic and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging or transactional support.

4) Container

Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects.

You can configure how your each of your beans should be created—either create one single instance of your bean or produce a new instance every time one is needed based on a configurable prototype—and how they should be associated with each other.

Spring should not, however, be confused with traditionally heavyweight EJB containers, which are often large and cumbersome to work with.

5) Framework :

Spring makes it possible to configure and compose complex applications from simpler components. In Spring, application objects are composed declaratively, typically in an XML file.

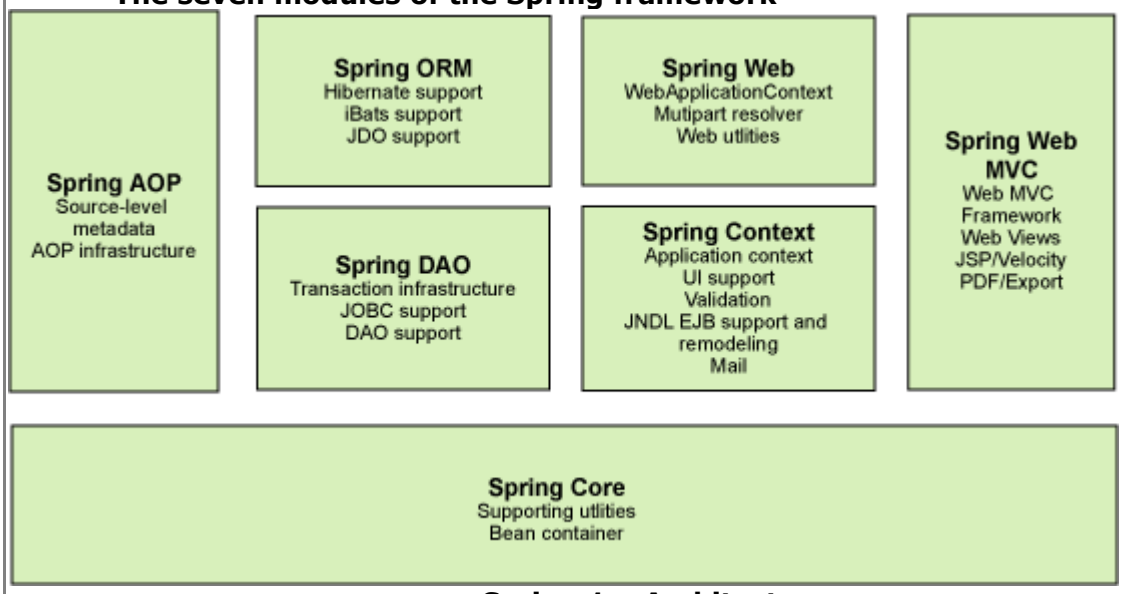
Spring also provides much infrastructure functionality (transaction management, persistence, framework integration, etc.), leaving the development of application logic to you.

All of these attributes of Spring enable you to write code that is cleaner, more manageable, and easier to test. They also set the stage for a variety of subframeworks within the greater Spring framework.

Spring modules:

The Spring framework is a layered architecture consisting of seven well-defined modules. The Spring modules are built on top of the core container, which defines how beans are created, configured, and managed.

The seven modules of the Spring framework



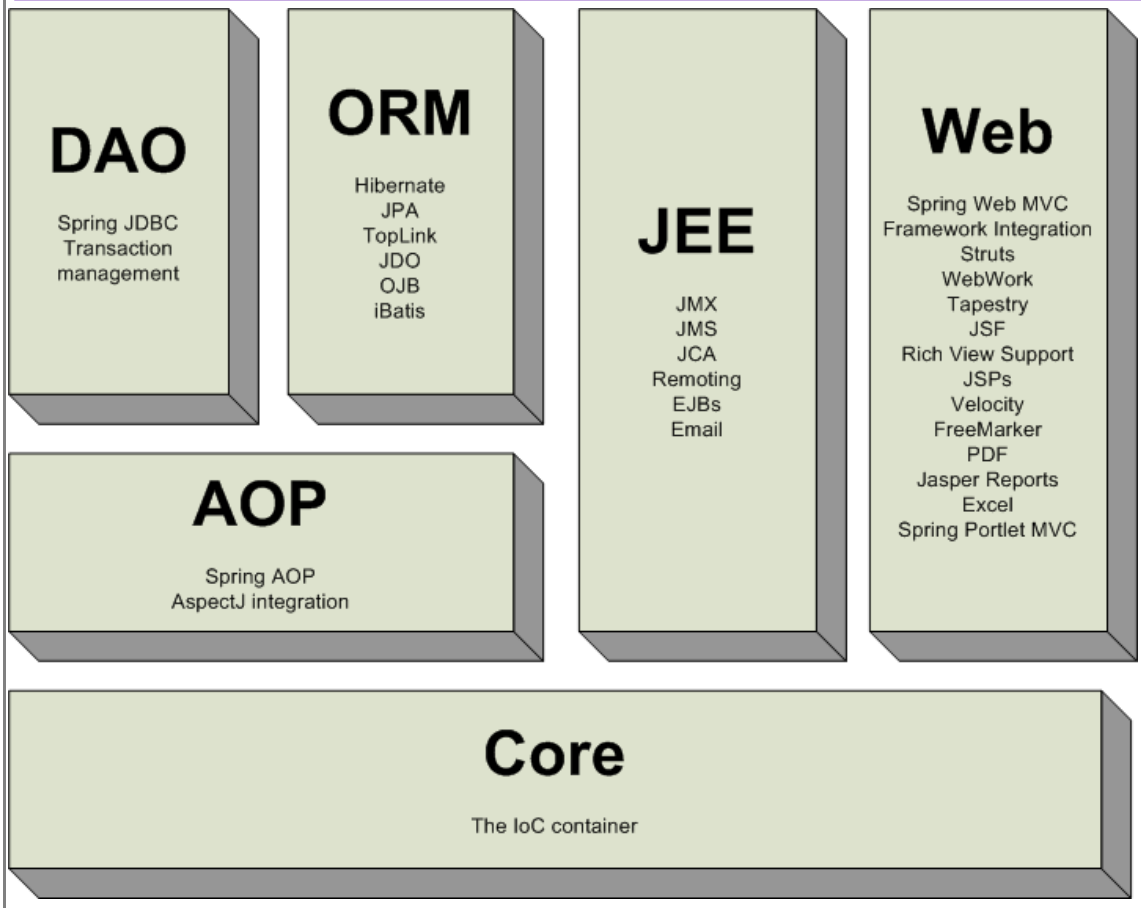
Spring 1.x Architecture

Each of the modules (or components) that comprise the Spring framework can stand on its own or be implemented jointly with one or more of the others. The functionality of each component is as follows:

- **The core container:**
  -- The core container provides the essential functionality of the Spring framework. -- A primary component of the core container is the BeanFactory, an implementation of the Factory pattern.
  -- The BeanFactory applies the *Inversion of Control* (IOC) pattern to separate an application's configuration and dependency specification from the actual application code.
- **Spring context:**
  -- The Spring context is a configuration file that provides context information to the Spring framework.
  -- The Spring context includes enterprise services such as JNDI, EJB, e-mail, internalization, validation, and scheduling functionality.
- **Spring AOP**:
  -- The Spring AOP module integrates aspect-oriented programming functionality directly into the Spring framework, through its configuration management feature. As a result you can easily

AOP-enable any object managed by the Spring framework.
-- The Spring AOP module provides transaction management services for objects in any Spring-based application. With Spring AOP you can incorporate declarative transaction management into your applications without relying on EJB components.

- **Spring DAO**:
  -- The Spring JDBC DAO abstraction layer offers a meaningful exception hierarchy for managing the exception handling and error messages thrown by different database vendors.
  -- The exception hierarchy simplifies error handling and greatly reduces the amount of exception code you need to write, such as opening and closing connections. Spring DAO's JDBC-oriented exceptions comply to its generic DAO exception hierarchy.
- **Spring ORM**:
  -- The Spring framework plugs into several ORM frameworks to provide its Object Relational tool, including JDO, Hibernate, and iBatis SQL Maps. All of these comply to Spring's generic transaction and DAO exception hierarchies.
- **Spring Web module**:
  -- The Web context module builds on top of the application context module, providing contexts for Web-based applications. As a result, the Spring framework supports integration with Jakarta Struts.
  -- The Web module also eases the tasks of handling multi-part requests and binding request parameters to domain objects.
- **Spring MVC framework**:
  -- The Model-View-Controller (MVC) framework is a full-featured MVC implementation for building Web applications.
  -- The MVC framework is highly configurable via strategy interfaces and accommodates numerous view technologies including JSP, Velocity, Tiles, iText, and POI.

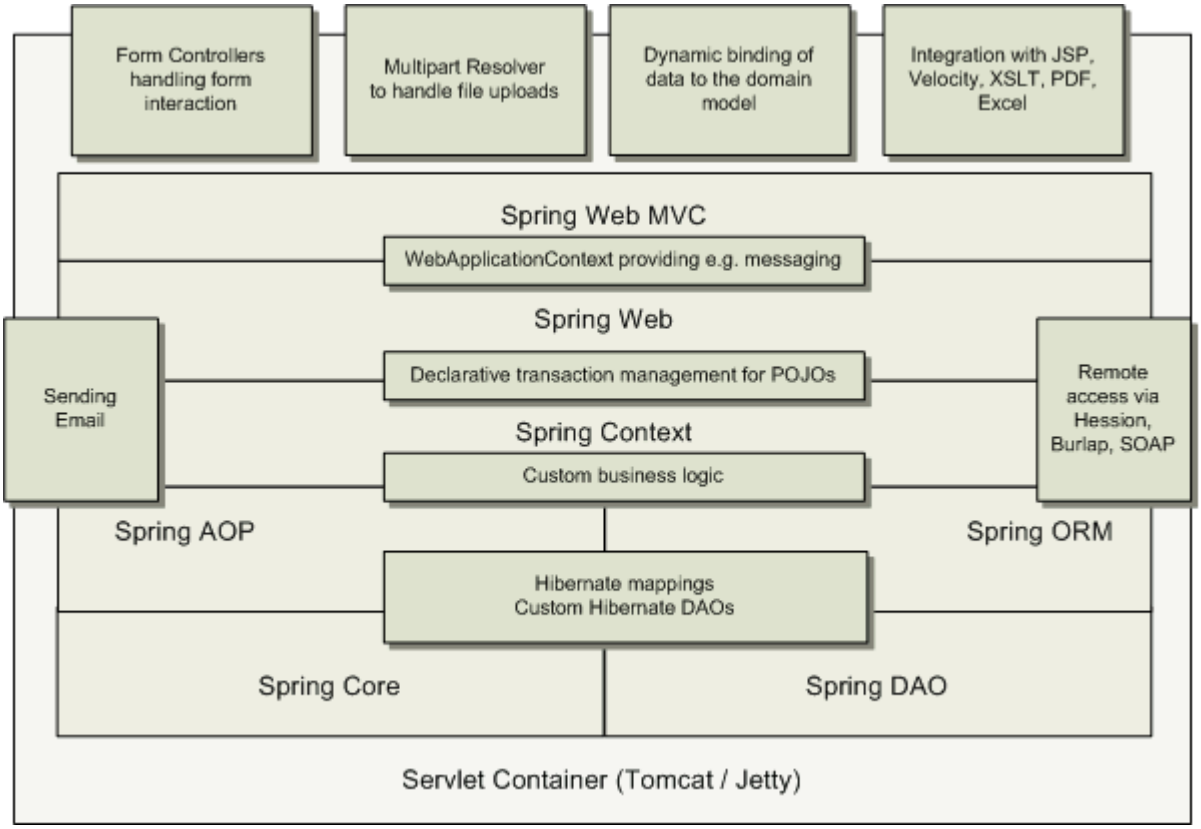


Spring 2.x Architecture

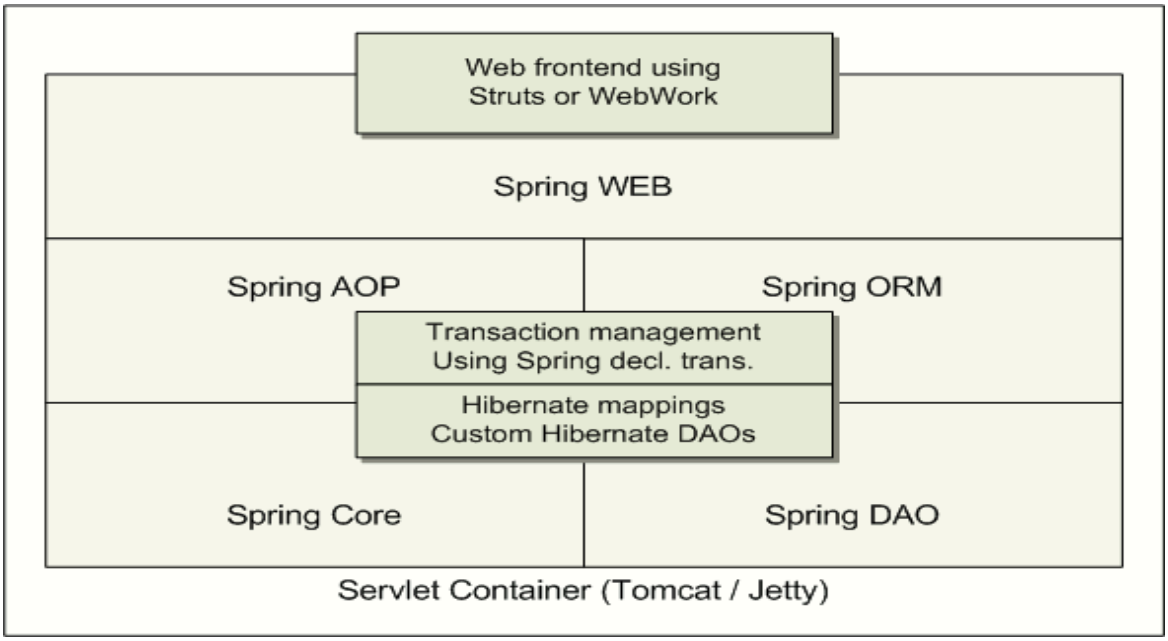
Spring Usage Scenarios:

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

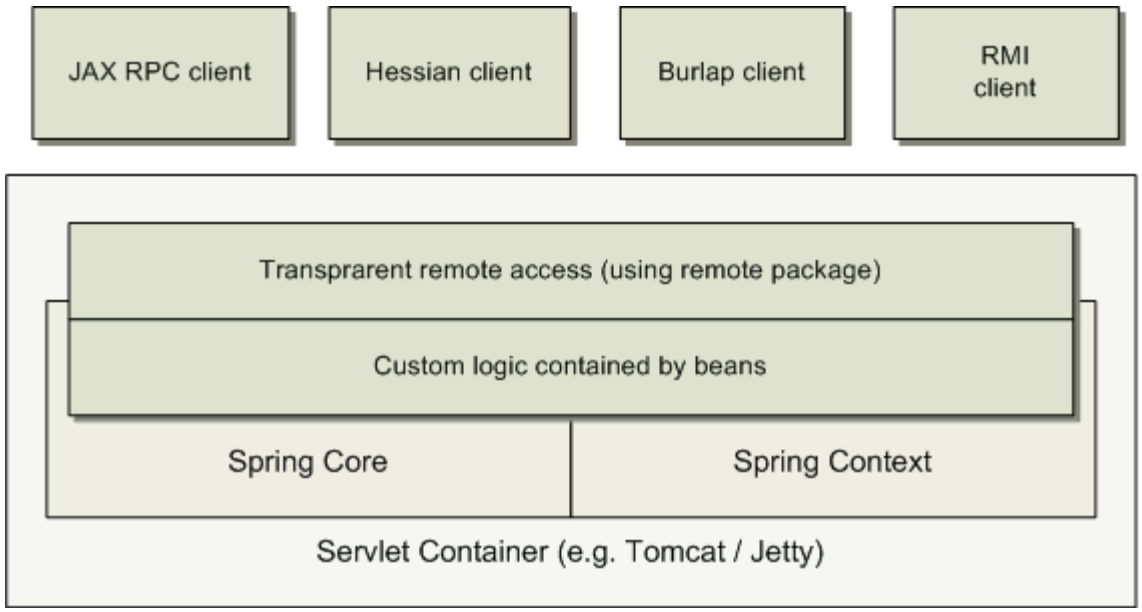
Scenario #1 : full-fledged Spring web application



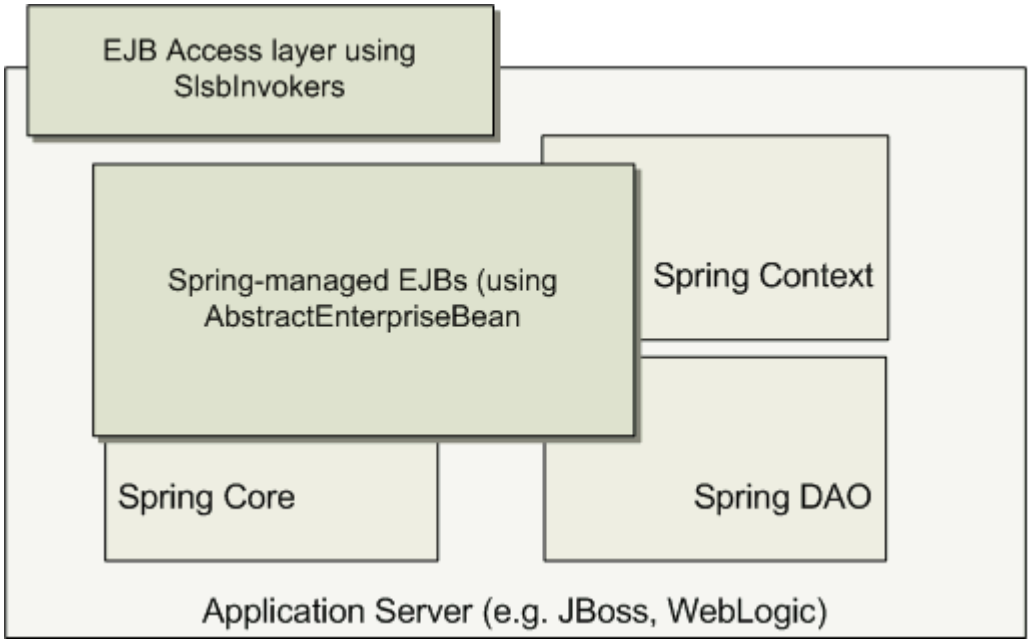
Scenario #2 : Spring middle-tier using a third-party web framework



Scenario #3 : Remoting usage scenario



Scenario #4 : Spring to EJB access scenario



Dependency LookUp: Resource explicitly searches and gathers dependent values from others(like other resources).

→ In this resource "pulls" the values from others

Ex1: If "**student**" (resource) gets "**course material**"(like dependent value) only after passing request for it, then it is called dependency lookup.

Ex2: The way we collect data source object reference from JNDI registry is called Dependency LookUp.

Advantage: Resource can gather only required dependent values.
Disadvantage: Resource should spend some time to gather it's dependent values before utilizing them.

Dependency Injection: The under-laying server or container or framework or runtime or special resource assigns values to the resource automatically and dynamically.

→ In Dependency Injection(also called IOC) the under-laying container or server or runtime or framework and etc. "pushes" the values to the resource.

Ex1: If course materiel is assigned to "Student" the moment student registers for a course is called DI.
Ex2: The way JVM calls constructor automatically to assign initial values to object.
Ex3: The way ActionServlet writes form data to FormBean class properties.
Ex4: The way Servlet container calls life cycle method "***service(-,-)***" to expose req, res objects to out servlet program.

Advantage: Resource can use the injected dependent values directly without spending time to get them.

DisAdvantage: The under-laying environment like Server/Framework may inject both necessary & unnecessary values.

→ Spring supports both dependency lookup & dependency Injection. spring containers are design to perform dependency injection on the resources of spring application.

→ A file or program of an application is called resource (generally it is class or interface)

Spring-Core Container General Example:

/*Connection.java */

```
	package com.venus;

	public interface Connection {
		public String getConnection();
	}

/* OracleConnectionProvider.java */
	package com.venus;

	public class OracleConnectionProvider implements Connection {
		public String getConnection() {
			return "Oracle DB ConnectionProvider";
		}
	}

/*MySqlConnectionProvider.java */
	package com.venus;

	public class MySqlConnectionProvider implements Connection {
		public String getConnection() {
			return "MySQL DB ConnectionProvider";
		}
	}


/* MyProperties.properties */

	provider=com.venus.MySqlConnectionProvider

/*MyContainer.java*/

	package com.venus;

	import java.io.IOException;
	import java.util.Properties;

	public class MyContainer {
		private static Properties properties;
		static {
			try {
				properties = new Properties();
				properties.load(MyContainer.class.getClassLoader()
						.getResourceAsStream("MyProperties.properties"));
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
```

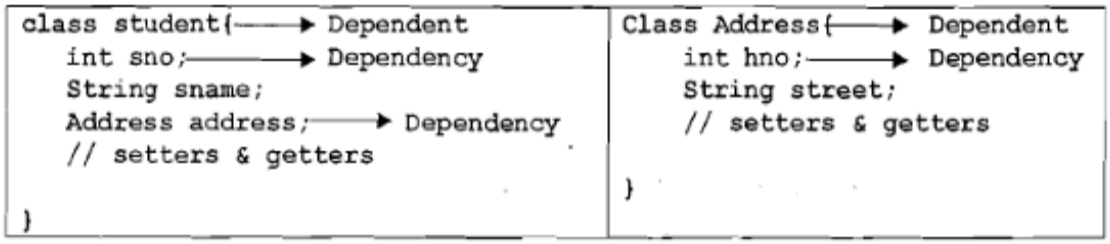
```
	public static Object getBean(String providerKey) {
		Object object = null;
		String providerClassName = properties.getProperty(providerKey);
		try {
			Class c = Class.forName(providerClassName);
			object = c.newInstance();
		} catch (Exception e) {
			e.printStackTrace();
		}
		return object;
	}
}

/*ConnectionUtitilizer.java*/

	package com.venus;

/**
 * Getting specified(in MyProperties.properties file) Provider object
 * dynamically by calling "getBean()" method. to this we are passing the key of
 * "provider" class.
 * */

	public class ConnectionUtitilizer {
		private static MyContainer container = new MyContainer();

		public static void main(String args[]) {
			Connection contract = (Connection) container.getBean("provider");
			String connectionString = contract.getConnection();
			System.out.println(connectionString);
		}
	}
```

Spring Bean→ Java class that are controlled by spring container are known as spring bean.

Injection → Setter injection is always preferable because we can change the dependency value any no. of times we required. But with constructor we can't change dependency object value, as and when we require.

IOC → Is a design principle using which an external entity injecting dependencies into dependent objects. In Spring, "Spring Container" is the external entity, which injects dependency objects into dependent object.

IOC terminology



- Here the dependencies can be primitive datatypes or primitive object types or userdefined datatypes.

DI (Dependency Injection)→ Physical realization of IOC is nothing but DI.

Container : It is a software or special resource or special application that can take care to manage complete life cycle of a given resource.

It takes cares of all the operations related to resource which are required for execution.

Two ways to locate spring configuration file for BeanFactory.

1) **By using FileSystemResource class :**
   org.springframework.core.io.FileSystemResource

This checks for given configuration file in the specified path.

FileSystemResource resource = new FileSystemResource("..\\spring-config.xml");

2) **By using ClassPathResource class :**
   org.springframework.core.io.ClassPathResource

This class takes or locates spring configuration file from the directories or jar files added in the classpath.

Place spring-config.xml in any folder of your choice ex: c:\store.

Note: Both provides abstract layer on "**java.io.file**" class.

Spring built-in Containers:

1) **BeanFactory** → (Core Module) → Activity BeanFactory container is nothing but creating object for a class that implements "org.springframework.Beans.factory.BeanFactory" interface.

There are several implementation of BeanFactory, the most useful one is

"org.springframework.beans.factory.xml.XmlBeanFactory". It loads its beans based on the definition contained in an XML file. To create an **XmlBeanFactory**, pass a **InputStream** to the constructor. The resource will provide the XML to the factory.

```
BeanFactory factory = new XmlBeanFactory(new
                              FileInputStream("myBean.xml"));
```

This line tells the bean factory to read the bean definition from the XML file. The bean definition includes the description of beans and their properties. But the bean factory doesn't instantiate the bean yet. To retrieve a bean from a 'BeanFactory', the getBean() method is called. When getBean() method is called, factory will instantiate the bean and begin setting the bean's properties using dependency injection.

```
myBean bean1 = (myBean)factory.getBean("myBean");
```

Ex:

```
// 1.Locate Spring configuration file
Resource resource = new ClassPathResource("spr-config.xml");

// 2. Activate BeanFactory Spring Container

BeanFactory factory = new XmlBeanFactory(resource);

// 3.get access to Spring Bean class object from Spring Container
Hello h1 = (Hello) factory.getBean("id1"); //Interface
```

2) **ApplicationContext** → (Core Module, J2EE module advance container) → It is an enhancement of BeanFactory, it has the following advantages:

a) pre-initialization of bean. Events to beans that are registered as listeners.
b) ability to read values of BeanProperty from property file, a generic way to load file resources.
c) a means for resolving text messages, including support for internationalization(I18n).

Because of additional functionality, '**Application Context**' is preferred over a **BeanFactory**. Only when the resource is *scarce like mobile devices*, '**BeanFactory**' is used. The three commonly used implementation of 'Application Context' are

ApplicationContext related file systems are as follows:

1) FileSystemXMLApplicatonContext → In specific path.
It loads context definition from an XML file in the file system. The application context is loaded from the file system by using the code

```
ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");
```

2) ClassPathXMLApplicationContext → In specific directory

It Loadscontext definition from an XML file located in the classpath, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code

```
ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
```

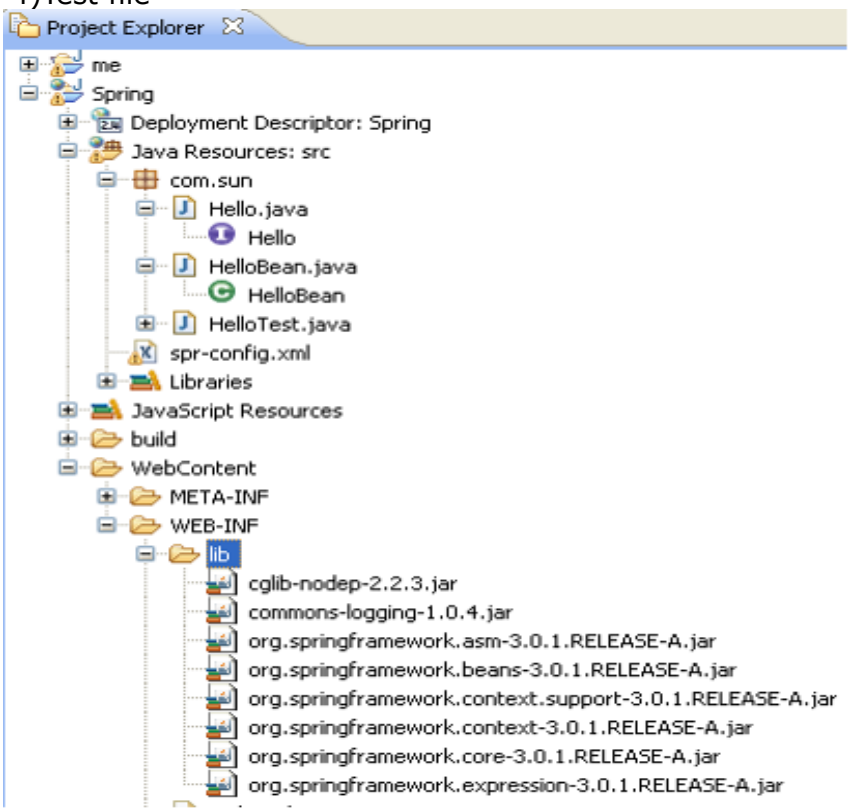
3) XMLWebApplicationContext → This is used in Spring Web MVC module. This container object created by web container internally. we no need to create our own .

By locating spring-config.xml in deployment directory structure of WEB-APP. It loads context definition from an XML file contained within a web application.

==========

Files required to work with Spring:

1)POJI
2)POJO Class
3)Spring Configuration file
4)Test file



All jars are not required - main jars are : commons-logging.jar, org.springframework.beans.jar

and org.springframework.core.jar

1) POJI – Plain Old Java Interface – **Hello.java**

```
package com.sun;

public interface Hello {
	public void show();
}
```

2) Bean class – **HelloBean.java**

```
package com.sun;

public class HelloBean implements Hello {

	private String message;

	public void setMessage(String message) {
		System.out.println("setMessage():: of HelloBean");
		this.message = message;
	}

	public void show() {
		System.out.println("Your Message.....: " + message);
	}
}
```

3) Spring configuration file - **spr-config.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
	<bean id="id1" class="com.sun.HelloBean">
		<property name="message" value="Hello! Welcome to spring" />
	</bean>
</beans>
```

4)Test File - **HelloTest.java** OR **HelloTest1.java**

```
package com.sun;
```

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

public class HelloTest {
	public static void main(String[] args) {

		// 1.Locate Spring configuration file
		Resource resource = new ClassPathResource("spr-config.xml");

		// 2. Activate BeanFactory Spring Container
		BeanFactory factory = new XmlBeanFactory(resource);

		// 3.get access to Spring Bean class object from Spring Container
		Hello h1 = (Hello) factory.getBean("id1"); //Interface

		// 4.Call Business methods of bean class
		h1.show();
	}
}
```

Output:

```
Dec 17, 2012 3:34:22 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [spr-config.xml]
setMessage():: of HelloBean
Your Message.....: Hello! Welcome to spring
```

OR

```
package com.sun;

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

public class HelloTest1 {

	public static void main(String[] args) {

//Locating Spring configuration file and Activating BF (1&2 steps combine in HelloTest)
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"spr-config.xml"));
		//get access to Spring Bean class object from Spring Container
		 HelloBean h1 = (HelloBean) factory.getBean("id1"); //BeanClass

		//Call Business methods of bean class
		h1.show();
```

```
        }
    }
```

Output:

```
Dec 17, 2012 3:35:33 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [spr-config.xml]
setMessage():: of HelloBean
Your Message.....: Hello! Welcome to spring
```

==========

Explanation:

1.**(HelloBean) factory.getBean("id1");**
gets Bean class from **spring configuration file** (spr-config.xml) based on the given bean id **"id1"**.

2. Spring container loads **HelloBean** class from the memory by using **"Class.forName()"**.

3.Crates **HelloBean object** using **new Instance()**.

4. Spring container performs setter Injection on **"message"** based on **properties** done in spring configuration file.

****

Note: Regarding **2,3 & 4** points →

HelloBean h1 = (HelloBean) factory.getBean("id1");

Point **2&3**. HelloBean h1=(HelloBean)**Class.forName(**"HelloBean"**).new Instance()**;
Class.forName() is capable of loading class into application container from memory.

Point **4**. h1.**setMessage(***"Hello! Welcome to spring"***)**;

****

5. **getBean()** returns **HelloBean** class object by collecting it from spring container. Programmer can refer this object either by using **Spring Interface reference** or **Spring Bean Class reference** object.

==========

Spring-Core Container Example:

```
/*Connection.java */

    package com.venus;
```

```
	public interface Connection {
		public String getConnection();
	}

/* OracleConnectionProvider.java */
	package com.venus;

	public class OracleConnectionProvider implements Connection {
		public String getConnection() {
			return "Oracle DB ConnectionProvider";
		}
	}

/*MySqlConnectionProvider.java */
	package com.venus;

	public class MySqlConnectionProvider implements Connection {
		public String getConnection() {
			return "MySQL DB ConnectionProvider";
		}
	}


<!-- applicationContext.xml -->

	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<bean id="provider" class="com.venus.OracleConnectionProvider">
		</bean>

		<!-- <bean id="provider1" class="com.venus.MySqlConnectionProvider">
		</bean> -->

	</beans>


/*ConnectionUtitilizer.java*/

	package com.venus;

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

	public class ConnectionUtitilizer {
		private static XmlBeanFactory factory = new XmlBeanFactory(new
```

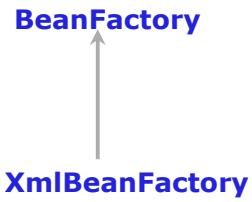
```
ClassPathResource("applicationContext.xml"));

        public static void main(String args[]) {
                Connection contract = (Connection) factory.getBean("provider");
                String connectionString = contract.getConnection();
                System.out.println(connectionString);
        }
}
```

Different types of containers in spring framework:

Spring containers are of 2 types:

1) **BeanFactory based** 2) **ApplicationContext based**

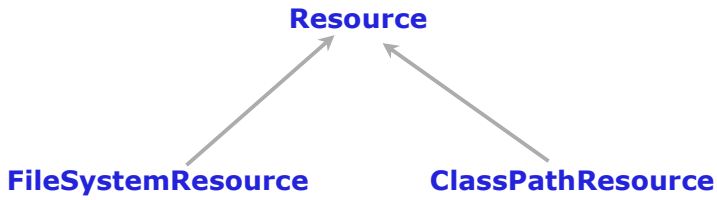
Bean Factory based: BeanFactory is an interface. It's implementation class is XmlBeanFactory.



→ Create instance of XmlBeanFactory class, is nothing but we create spring container.

→ Constructor of XmlBeanFactory class takes "Resource" object as arg,which represents spring congifuration file.

There are two implementation classes of "Resource" interface,which are used to specify spring configuration file.



Ex:

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

public class Client {
        private static XmlBeanFactory factory = new XmlBeanFactory(new
FileSystemResource("D:/venukumarsIndigo/IndigoSpringWorkSpace/Spring-Core-Ex1-
Container/src/applicationContext.xml"));
```

```
        public static void main(String args[]) {
                Connection contract = (Connection) factory.getBean("provider");
                String connectionString = contract.getConnection();
                System.out.println(connectionString);
        }
}
```

→ In the FileSystemResource takes complete system path(absolute path). So in the futue if we change location of the project then again we need to change the locatiom of spring configuration file in the program. that's why it is not advisable to use FileSystemResource.

Ex2:

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

public class Client {
        private static XmlBeanFactory factory = new XmlBeanFactory(new
ClassPathResource("applicationContext.xml"));

        public static void main(String args[]) {
                Connection contract = (Connection) factory.getBean("provider");
                String connectionString = contract.getConnection();
                System.out.println(connectionString);
        }
}
```

→ In ClassPathResource, the container reading the configuration file from class path, but we are not specifying the complete path. even we change the location of the project in the system, still out application work without modification, because it reads configuration file from classpath. So it is advisable to use ClassPathResource instead of FileSystemResource.

Key notes:

→ It is a light weight container

→ It loads spring beans configured in spring configuration file, and manages the life cycle of the spring bean when we call getBean("springbeanref"). So when we call getBean("springbeanref") then only spring bean life cycle starts.

→ Generally it is used to develop stand alone applications, mobile applications.

ApplicationContext: It is a child interface of BeanFactory

BeanFactory
ApplicationContext
ClassPathXmlApplicationContext
FileSystemXmlApplicaionContext
XmlWebApplicationContext

Ex1: FileSystemXmlApplicationContext

```
package com.venus;

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

public class EmployeeTest {
	private static ApplicationContext context = new
FileSystemXmlApplicationContext("D:/venukumarsIndigo/IndigoSpringWorkSpace/Spring-Core-Ex12-ConstructorAmbiguity/src/applicationContext.xml");

	public static void main(String args[]) {
		Employee employee = (Employee) context.getBean("empref");
		employee.getEmployeeDetails();
	}
}
```

Ex2: ClassPathXmlApplicationContext

```
package com.venus;

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

public class EmployeeTest {
	private static ApplicationContext context = new
		ClassPathXmlApplicationContext("applicationContext.xml");

	public static void main(String args[]) {
		Employee employee = (Employee) context.getBean("empref");
		employee.getEmployeeDetails();
	}
}
```

XmlWebApplicationContext : This is used in Spring Web MVC module. This container object created by web container internally. we no need to create our own.

Key notes:

→ It loads spring beans configured in spring configuration file, and manages the life cycle of the spring bean as and when container starts, it won't wait until getBean() method is called.

→ ApplicationContext based container provides all the features of BeanFactory. In addition to them it provides the following features also.

1) Event-handling support
2) Internationalization support
3) Remoting
4) EJB configuration
5) Scheduling
6) JNDI look-up.. etc.

→ In enterprise level application always ApplicationContext based container is used.

Spring bean Life Cycle

→ Spring bean has 4 life cycle states:

1) **Instantiation**
2) **Initialization**
3) **Ready to use(method ready to use state)**
4) **Destruction**

→ Initialization method, destruction method can be given to spring bean in the following ways:

→ By implementing spring framework given interface they are
1) IntializingBean
2) DisposableBean

→ By configuring custom initialization and destuction method in the spring configuration file using <bean> tag attribute
1)init-method
2)destroy-method

→ By using annotations
1)preDestroy
2)postConstruct

→ If spring bean is implementing framework interface "InitializingBean", we need to define afterPropertiesSet() in bean class. It becomes the initialization method. similarly if bean implements "DesposableBean" interface, destroy method becomes destruction method & no need to inform.

Spring Bean: Java class that are controlled by spring container are known as spring bean.

Injection : Setter injection is always preferable because we can change the dependency value any no. of times we required. But with constructor we can't change dependency object value, as and when we require.

Steps to develop Spring Application:

1) Develop the Spring beans(classes) according to the business requirement.
2) Develop the spring configuration file and configure the spring beans developed in step1.
3) Inject the dependencies into dependent objects either in setter approach(identify how the bean is expecting Dependency object values based on that we are going to define either of one).
4) Create the instance of spring container.
5) Get the instance of dependent objects from the container by making a method call **getBean()**, by passing id of bean.
6) After we get the bean object we will call the required methods on dependent bean.

Setter injection in a Spring Container:

1) Declare the dependency variables in spring bean class.
2) Define setter method for those variables created in step1.
3) Use <property/> tag in bean configuration file & supply the value.

```
<property name="" value=""/> where dependency is primitive type
<property name="" ref=""/>  where dependency is object type
```

Example:

```
package com.venus;
```

Bean class:

```
public class GreetingBean { // Dependent
	private String name; // Dependency

	public GreetingBean() {
	}

	public GreetingBean(String name) {
		this.name = name;
	}

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

Injection:

1) **premitive type:**

```
<bean id="Gb" class="com.venus.GreetingBean">
	<property name="name" value="VagmeeChandraGuru" />
</bean>
```

2) **Object Type:**

```
<bean id="Gb" class="com.venus.GreetingBean">
	<property name="name" ref="VagmeeChandraGuru" />
</bean>

<beans>
	<bean id="addr" class="com.venus.Address">
		<property name="hno" value="411" />
		<property name="city" value="Nalgonda" />
	</bean>

	<bean id="emp" class="com.venus.Employee">
		<property name="eno" value="1" />
		<property name="name" value="VagmeeChandraGuru" />
		<property name="address" ref="addr" />
	</bean>
```

```
</beans>
```

Constructor injection in a Spring Container:

1) Declare the dependency variables in spring bean class.
2) Define constuctor which will take these variables (created in step1) as parameters.
3) Use <constructor-arg> tag in bean configuration file & supply the value.
   <constructor-arg value="value" /> where dependency is primitive type
   <constructor-arg ref="ref-identifier"/> where dependency is object type

→ **In config.xml**

1)**Premitive type:**

```
<bean id="Gb" class="com.venus.GreetingBean">
    <constructor-arg value="VagmeeChandra"></constructor-arg>
</bean>
```

2) **Object type:**

```
<bean id="Gb" class="com.venus.GreetingBean">
    <constructor-arg ref="VagmeeChandra"></constructor-arg>
</bean>

    <bean id="addr" class="com.venus.Address">
        <constructor-arg value="411" />
        <constructor-arg value="Nalgonda" />
    </bean>
    <bean id="emp" class="com.venus.Employee">
        <constructor-arg value="1" />
        <constructor-arg value="VagmeeChandraGuru" />
        <constructor-arg ref="addr" />
    </bean>

<bean id="gb" name="gb1, gb2, gb3, gb4" class="com.venus.bean.GreetingBean">
```

id value should be unique, whereas name no need to be unique.
id value won't allow special characters. whereas name attribute allows any king of special characters.
name attribute is useful when we develop spring MVC based applications, because as we know every URL should start with "/".
Generally most of the cases we use id attribute only but not name attribute

The IOC Container (Spring Core)

IOC → Is a design principle using which an external entity injecting dependencies into dependent objects. In Spring, "Spring Container" is the external entity, which injects dependency objects into dependent object.

☐ Inversion of control is at the heart of the Spring framework. IoC is not nearly as complex as it sounds. In fact, by applying IoC in your projects, you'll find that your code will become significantly

simpler, easier to understand, and easier to test.

☐Any nontrivial application (pretty much anything more complex than Hello-World.java) is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). As you'll see, this can lead to highly coupled and hard-to-test code.

☐Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. That is, dependencies are *injected* into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

☐ The basic concept of the Inversion of Control pattern (also known as dependency injection) is that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (in the case of the Spring framework, the IOC container) is then responsible for hooking it all up.

☐The org.springframework.beans and org.springframework.context packages provide the basis for the Spring Framework's IoC container.

☐ TheBeanFactory interface provides an advanced configuration mechanism capable of managing objects of any nature.

☐ The ApplicationContext interface builds on top of the BeanFactory (it is a sub-interface) and adds other functionality such as easier integration with Spring's AOP features, message resource handling (for use in internationalization), event propagation, and application-layer specific contexts such as the WebApplicationContext for use in web applications.

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

☐ In Spring, those objects that form the backbone of your application and that are managed by the Spring IoC *container* are referred to as *beans*. A bean is simply an object that is instantiated, assembled and otherwise managed by a Spring IoC container; other than that, there is nothing special about a bean

☐ The org.springframework.beans.factory.BeanFactory is the actual representation of the Spring

IoC *container* that is responsible for containing and otherwise managing the aforementioned bean

Dependency Injection (DI)

DI → Physical realization of IOC is nothing but DI

→ Resolving the dependencies of among the beans is nothing but Dependency Injection.

→ In case of spring, dependencies are injected into dependent object (beans) through DI by spring container.

If underlying server software or framework software or container software or special resource of the application gathers dependent values and assigns to resource of the application dynamically is called **Dependency Injection(IOC)**.

Dependent values will be pushed to resource dynamically.

Dependency injection can help you design your applications so that the architecture links the components rather than the components linking themselves.

Dependency injection eliminates tight coupling between objects to make both the objects and applications that use them more flexible, reusable, and easier to test. It facilitates the creation of loosely coupled objects and their dependencies.

The basic idea behind Dependency Injection is that you should isolate the implementation of an object from the construction of objects on which it depends. Dependency Injection is a form of the Inversion of Control Pattern where a factory object carries the responsibility for object creation and linking.

The factory object ensures loose coupling between the objects and promotes seamless testability.

The primary advantages of dependency injection are:

- Loose coupling
- Centralized configuration
- Easily testable

Types of Dependency Injection

There are three common forms of dependency injection:

1. Setter Injection
2. Constructor Injection
3. Interface-based injection

Mainly we are using **1) setter injection 2) constructor injection** to achieve DI in spring

→ In a well designed spring bean, interfaces are specified as dependencies.

→ If dependency is provided to dependent obj via setter method is known as setter injection.

→ If dependency is provided to dependent obj via constructor is known as constructor injection.

Advantages:

1) Dynamically we can change dependency objects.
2) Even dependency changing dependent object code remains same, there by it is offering loose coupling among dependent and dependencies.

Setter injection Example:

/*GreetingBean.java*/

```
package com.venus;

public class GreetingBean {
	private String name;

	public GreetingBean() {
	}

	public GreetingBean(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}

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

<!-- applicationContext.xml -->

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
	<bean id="Gb" class="com.venus.GreetingBean">
		<property name="name" value="VagmeeChandraGuru" />
	</bean>

</beans>
```

/*ConnectionUtitilizer.java*/

```
package com.venus;
```

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

public class ConnectionUtitilizer {
	private static XmlBeanFactory factory = new XmlBeanFactory(new
ClassPathResource("applicationContext.xml"));

	public static void main(String args[]) {
		GreetingBean bean = (GreetingBean) factory.getBean("Gb");
		String name =  bean.getName();
		System.out.println("Hello! "+name+" welcome to Spring");
	}
}
```

constructor injection Example:

/*GreetingBean.java*/

```
package com.venus;

public class GreetingBean {
	private String name;

	public GreetingBean() {
	}

	public GreetingBean(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}

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

<!-- applicationContext.xml -->

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
	<bean id="Gb" class="com.venus.GreetingBean">
```

```
                <constructor-arg value="VagmeeChandra"></constructor-arg>
        </bean>

    </beans>
```

```
/*ConnectionUtitilizer.java*/
    package com.venus;

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

    public class ConnectionUtitilizer {
        private static XmlBeanFactory factory = new XmlBeanFactory(new
    ClassPathResource("applicationContext.xml"));

        public static void main(String args[]) {
            GreetingBean bean = (GreetingBean) factory.getBean("Gb");
            String name =  bean.getName();
            System.out.println("Hello! "+name+" welcome to Spring");
        }
    }
```

Spring-Core-Ex4-ObjectTypeConstructorInjection

```
/* Address.java */

    package com.venus;

    public class Address {
        private int hno;
        private String city;

        public Address() {
        }

        public Address(int hno, String city) {
            this.hno = hno;
            this.city = city;
        }

        public int getHno() {
            return hno;
        }

        public void setHno(int eno) {
            this.hno = eno;
```

```
		}

		public String getCity() {
			return city;
		}

		public void setCity(String city) {
			this.city = city;
		}

	}

/* Employee.java */

	package com.venus;

	public class Employee {
		private int eno;
		private String name;
		private Address address;

		public Employee() {
		}

		public Employee(int eno, String name, Address address) {
			this.eno = eno;
			this.name = name;
			this.address = address;
		}

		public int getEno() {
			return eno;
		}

		public void setEno(int eno) {
			this.eno = eno;
		}

		public String getName() {
			return name;
		}

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

		public Address getAddress() {
			return address;
		}
```

```
	public void setAddress(Address address) {
		this.address = address;
	}

}
```

```
<!-- applicationContext.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>

		<bean id="addr" class="com.venus.Address">
			<constructor-arg value="411" />
			<constructor-arg value="Nalgonda" />
		</bean>
		<bean id="emp" class="com.venus.Employee">
			<constructor-arg value="1" />
			<constructor-arg value="VagmeeChandraGuru" />
			<constructor-arg ref="addr" />
		</bean>

	</beans>

<!-- <property name = "name" value = "VagmeeChandraGuru"/> →
```

```
/* ConnectionUtitilizer.java */
	package com.venus;

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

	public class ConnectionUtitilizer {
		private static XmlBeanFactory factory = new XmlBeanFactory(
				new ClassPathResource("applicationContext.xml"));

		public static void main(String args[]) {
			Employee employee = (Employee) factory.getBean("emp");

			System.out.println("Employee Details ....");
			System.out.println("\tEmployee Number ==> " + employee.getEno());
			System.out.println("\tEmployee Name ==> " + employee.getName());

			Address address = employee.getAddress();
			System.out.println("\tAddress ==> \n\t\tH.No.: " + address.getHno());
			System.out.println("\t\tCity ==> " + address.getCity());
```

```
		}
	}
```

Spring-Core-Ex5-ObjectTypeSetterInjection :

/* Address.java */

```
	package com.venus;

	public class Address {
		private int hno;
		private String city;

		public Address() {
		}

		public Address(int hno, String city) {
			this.hno = hno;
			this.city = city;
		}

		public int getHno() {
			return hno;
		}

		public void setHno(int eno) {
			this.hno = eno;
		}

		public String getCity() {
			return city;
		}

		public void setCity(String city) {
			this.city = city;
		}

	}
```

/* Employee.java */

```
	package com.venus;

	public class Employee {
		private int eno;
		private String name;
		private Address address;

		public Employee() {
```

```
	}

	public Employee(int eno, String name, Address address) {
		this.eno = eno;
		this.name = name;
		this.address = address;
	}

	public int getEno() {
		return eno;
	}

	public void setEno(int eno) {
		this.eno = eno;
	}

	public String getName() {
		return name;
	}

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

	public Address getAddress() {
		return address;
	}

	public void setAddress(Address address) {
		this.address = address;
	}

}
```

<!-- applicationContext.xml -->

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
	<bean id="addr" class="com.venus.Address">
		<property name="hno" value="411" />
		<property name="city" value="Nalgonda" />
	</bean>

	<bean id="emp" class="com.venus.Employee">
		<property name="eno" value="1" />
		<property name="name" value="VagmeeChandraGuru" />
		<property name="address" ref="addr" />
```

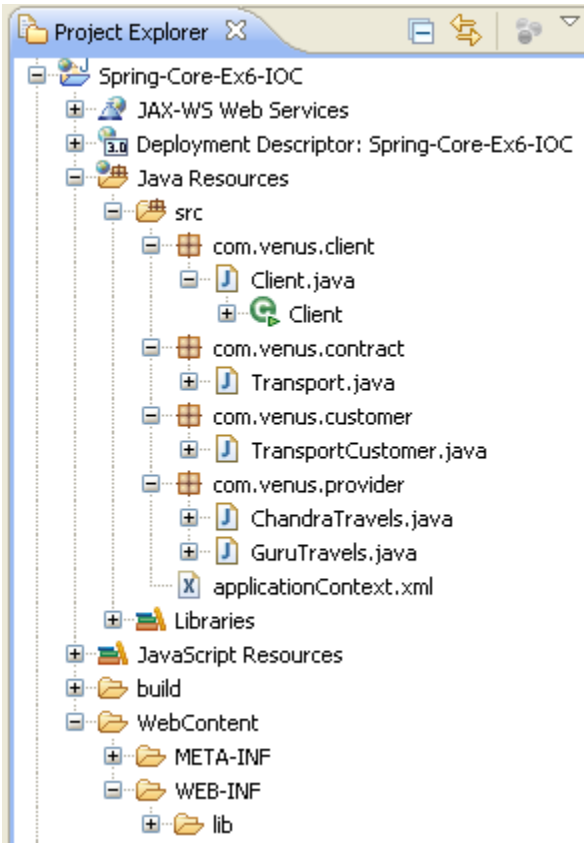
```
		</bean>
	</beans>

	<!-- <property name = "name" value = "VagmeeChandraGuru"/> -->
```

```
/* ConnectionUtitilizer.java */
	package com.venus;

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

	public class ConnectionUtitilizer {
		private static XmlBeanFactory factory = new XmlBeanFactory(
				new ClassPathResource("applicationContext.xml"));

		public static void main(String args[]) {
			Employee employee = (Employee) factory.getBean("emp");

			System.out.println("Employee Details ....");
			System.out.println("\tEmployee Number ==> " + employee.getEno());
			System.out.println("\tEmployee Name ==> " + employee.getName());

			Address address = employee.getAddress();
			System.out.println("\tAddress ==> \n\t\tH.No.: " + address.getHno());
			System.out.println("\t\tCity ==> " + address.getCity());

		}
	}
```

Spring-Core-Ex6-IOC

```
/*Transport.java*/
	package com.venus.contract;

	public interface Transport {
		public void doTravel();
	}
/*ChandraTravels.java*/
	package com.venus.provider;

	import com.venus.contract.Transport;

	public class ChandraTravels implements Transport {

		public ChandraTravels() {
		}

		@Override
		public void doTravel() {
```

```
		System.out.println("Chandra Travels is the Bestone, Thank u to choose this. Happy journey");
		}
```

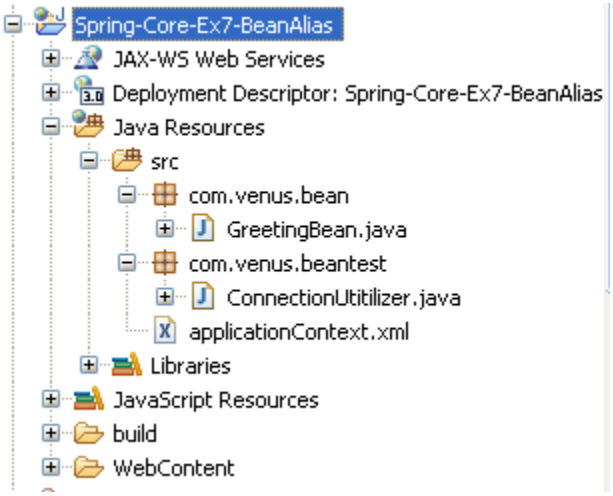



```
	}
/*GuruTravels.java*/
	package com.venus.provider;

	import com.venus.contract.Transport;

	public class GuruTravels implements Transport {

		public GuruTravels() {
		}

		@Override
		public void doTravel() {
	System.out.println("Guru Travels is the Bestone, Thank u to choose this. Happy journey");
		}

	}
```

```
/*TransportCustomer.java */
	package com.venus.customer;

	import com.venus.contract.Transport;

	public class TransportCustomer {
		private Transport transport;

		public TransportCustomer() {
		}

		public TransportCustomer(Transport transport) {
			this.transport = transport;
		}

		public void useTransport() {
			transport.doTravel();
		}

		public void setTransport(Transport transport) {
			this.transport = transport;
		}

	}

<!-- applicationContext.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>

		<bean id="Gt" class="com.venus.provider.GuruTravels" />
		<bean id="Ct" class="com.venus.provider.ChandraTravels" />
		<bean id="Tc" class="com.venus.customer.TransportCustomer">
			<constructor-arg ref="Gt" />
			<!-- <constructor-arg ref = "Ct"/> -->
		</bean>

	</beans>



/* Client.java */
	package com.venus.client;

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

```
import com.venus.customer.TransportCustomer;

public class Client {
	private static XmlBeanFactory factory = new XmlBeanFactory(
			new ClassPathResource("applicationContext.xml"));

	public static void main(String args[]) {
		TransportCustomer customer = (TransportCustomer) factory.getBean("Tc");
		customer.useTransport();

	}
}
```

Spring-Core-Ex7-BeanAlias:




```
/*GreetingBean.java*/
	package com.venus.bean;

	public class GreetingBean {
		private String name;

		public GreetingBean() {
		}

		public GreetingBean(String name) {
			this.name = name;
		}

		public String getName() {
			return name;
		}
```

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

<!-- applicationContext.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<bean id="gb" name="gb1, gb2, gb3, gb4" class="com.venus.bean.GreetingBean">
			<property name="name" value="VagmeeChandraGuru" />
		</bean>

	</beans>

<bean name="gb1, gb2, gb3, gb4" class="com.venus.bean.GreetingBean">
	we can create like this also, the container creates objects based on
	the alias names.


/*ConnectionUtitilizer.java*/
	package com.venus.beantest;

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

	import com.venus.bean.GreetingBean;

	public class ConnectionUtitilizer {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"applicationContext.xml");

		public static void main(String args[]) {
			GreetingBean bean = (GreetingBean) context.getBean("gb");
			String name = bean.getName();
			System.out.println("Hello! " + name + " welcome to Spring");
			String[] aliaaNames = context.getAliases("gb");
			System.out.println("Alias Names of Gb...");
			for (int i = 0; i < aliaaNames.length; i++) {
				System.out.print("\t" + aliaaNames[i] + ",");
			}

		}
	}

	/*
```

```
 Output:
Hello! VagmeeChandraGuru welcome to Spring Alias Names of Gb... gb2, gb1, gb3, gb4,
 */
```

=

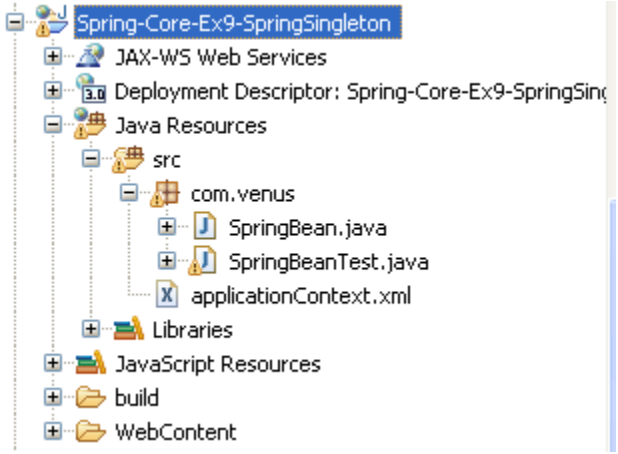
Scope:
singleton: We create singleton when there is no instance variables and static variables to save the memory.

Spring-Core-Ex8-Simple singleton Creation

```
/*Car.java*/
	public class Car {
		private static Car car = new Car();

		private Car() {
			System.out.println("Car() constructor");
		}

		public static Car getInstance() {
			return car;

		}

	}

/*CarTest.java*/
	public class CarTest {

		public static void main(String[] args) {
			Car car1 = Car.getInstance();
			Car car2 = Car.getInstance();
			Car car3 = Car.getInstance();
			Car car4 = Car.getInstance();
			Car car5 = Car.getInstance();
			Car car6 = Car.getInstance();
			System.out.println(car1);
			System.out.println(car2);
			System.out.println(car3);
			System.out.println(car4);
			System.out.println(car5);
			System.out.println(car6);
		}
	}
```

→ Here we are implementing our own logic to implement singleton.

Spring-Core-Ex9-SpringSingleton




```
package com.venus;

	public class SpringBean {

	}
```

```
<!-- applicationContext.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<bean id="sb" class="com.venus.SpringBean">
		</bean>

	</beans>
```

```
	package com.venus;

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

	public class SpringBeanTest {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"applicationContext.xml");

		public static void main(String args[]) {
			SpringBean springBean1 = (SpringBean) context.getBean("sb");
			System.out.println("Object 1 ==>" + springBean1);
```

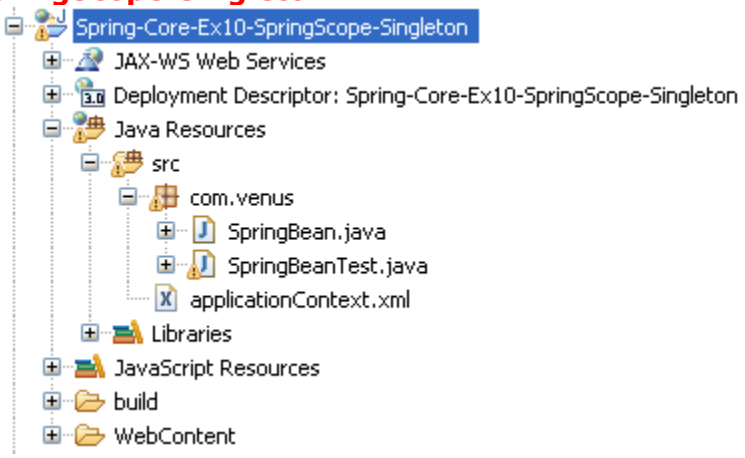
```
                SpringBean springBean2 = (SpringBean) context.getBean("sb");
                System.out.println("Object 2 ==>" + springBean2);
                SpringBean springBean3 = (SpringBean) context.getBean("sb");
                System.out.println("Object 3 ==>" + springBean3);
        }
}
```

→ In the case of spring singleton design pattern is implicitly implemented. To make one object as singleton, or non-singleton just we need change configuration in the spring configuration file. But we no need to implement our own logic.

Spring Scopes

→ **Program:**

Spring-Core-Ex10-SpringScope-Singleton




```
/*SpringBean.java*/
        package com.venus;

        public class SpringBean {

        }


<!-- applicationContext.xml -->
        <?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

        <beans>
                <bean id="sb" class="com.venus.SpringBean" scope="singleton">
                </bean>
```

```
    </beans>

/*SpringBeanTest.java*/
    package com.venus;

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

    public class SpringBeanTest {
        private static ApplicationContext context = new ClassPathXmlApplicationContext(
                    "applicationContext.xml");

        public static void main(String args[]) {
            SpringBean springBean1 = (SpringBean) context.getBean("sb");
            System.out.println("Object 1 ==>" + springBean1);
            SpringBean springBean2 = (SpringBean) context.getBean("sb");
            System.out.println("Object 2 ==>" + springBean2);
            SpringBean springBean3 = (SpringBean) context.getBean("sb");
            System.out.println("Object 3 ==>" + springBean3);
        }
    }
    /*
      Output:
      Object 1 ==>com.venus.SpringBean@ecd7e
      Object 2 ==>com.venus.SpringBean@ecd7e
      Object 3 ==>com.venus.SpringBean@ecd7e
     */
```

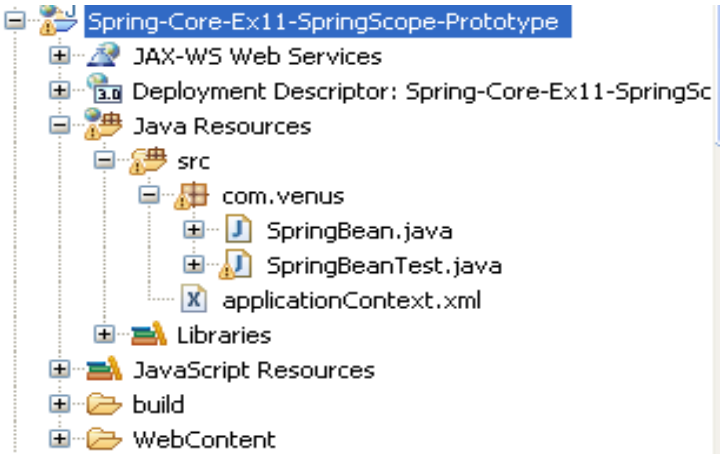
Analysis: By observing the above application the spring container creates only one object(returning same address) even we call getBean("objRef") "n" number of times. so by default a spring bean is "singleton".

→ If we want to change its nature to "non-singleton" i.e for each "getBean()" method call on container one brand new instance has to be created. To achieve this we can make use of "scope" attribute of "<bean>" tag.

→ scope attribute takes two values: 1) **singleton (default)** 2) **prototype**

Spring-Core-Ex11-SpringScope-Prototype:

```
/*SpringBean.java*/
    package com.venus;

    public class SpringBean {

    }
```




```
<!-- applicationContext.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<bean id="sb" class="com.venus.SpringBean" scope="prototype">
		</bean>

	</beans>

/*SpringBeanTest.java*/
	package com.venus;

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

	public class SpringBeanTest {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"applicationContext.xml");

		public static void main(String args[]) {
			SpringBean springBean1 = (SpringBean) context.getBean("sb");
			System.out.println("Object 1 ==>" + springBean1);
			SpringBean springBean2 = (SpringBean) context.getBean("sb");
			System.out.println("Object 2 ==>" + springBean2);
			SpringBean springBean3 = (SpringBean) context.getBean("sb");
			System.out.println("Object 3 ==>" + springBean3);
		}
	}
	/*
	Output:
```

```
Object 1 ==>com.venus.SpringBean@ecd7e
Object 2 ==>com.venus.SpringBean@1d520c4
Object 3 ==>com.venus.SpringBean@15a3d6b

*/
```

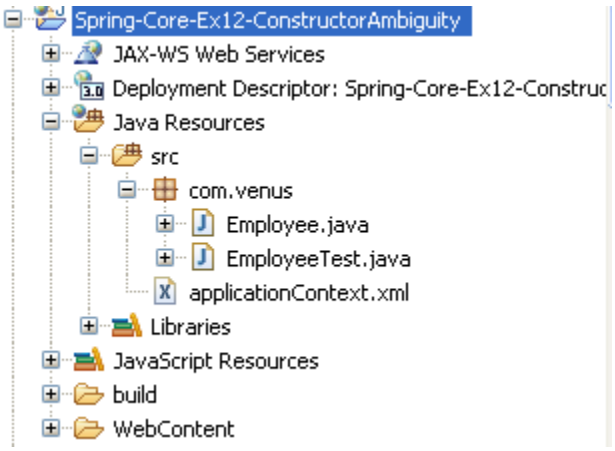
→ All possible values of scope attribute:

1)singleton : specifies to container to create the configured spring bean only once in the container.
2)prototype: specifies to container to create the configured spring bean for each time when it is requested (i.e. getBean("springref"))
3)request: specifies to container to create the configured spring bean once in per web request. It is applicable only in web module.
4)session: specifies to container to create the configured spring bean once in per HttpSession. It is applicable only in web module.
5)globalsession: specifies to container to create the configured spring bean once in a global session. It is applicable in portals.
6)thread: specifies to container to create the configured spring bean once in per each thread. It is not implicitly registered. If we want to specify thread as a scope bean, we have to register explicitly.

Constructor Ambiguity:

→ When two constructors have the same no. of parameters, then it occurs.

Spring-Core-Ex12-ConstructorAmbiguity




```
/*Employee.java*/
	package com.venus;

	public class Employee {
		private int eno = 1;
		private String name = null;
		private double sal = 0.0;
```

```
	private String designation = "Manager";

	public Employee() {
	}

	public Employee(int eno, String name) {
		this.eno = eno;
		this.name = name;
	}

	public Employee(double sal, String designation) {
		this.sal = sal;
		this.designation = designation;
	}

	public void getEmployeeDetails() {
		System.out.println("eno = " + eno);
		System.out.println("name = " + name);
		System.out.println("sal = " + sal);
		System.out.println("designation = " + designation);
	}

}

<!-- applicationContext.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<bean id="empref" class="com.venus.Employee">
			<constructor-arg value="5" />
			<constructor-arg value="VagmeeChandraGuru" />
			<!-- <constructor-arg value="100000" /> <constructor-arg value="Analyst"
				/> -->
		</bean>

	</beans>

	<!-- <property name = "name" value = "VagmeeChandraGuru"/> -->

/*EmployeeTest.java*/
	package com.venus;

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

	public class EmployeeTest {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
```

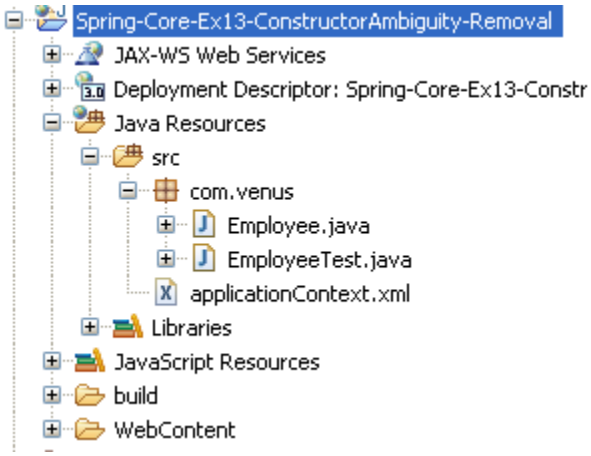
```
                        "applicationContext.xml");

                public static void main(String args[]) {
                        Employee employee = (Employee) context.getBean("empref");
                        employee.getEmployeeDetails();
                }
        }
```

To overcome this we want to use the following two attributes:

1) index → it specifies the index of the arg (it takes from 0)

2) type → it specifies the type of the arg. for object type we have to give fully qualified name, for primitives directly we can give the corresponding primitive key words.

Spring-Core-Ex13-ConstructorAmbiguity-Removal




```
/*Employee.java*/
        package com.venus;

        public class Employee {
                private int eno = 1;
                private String name = null;
                private double sal = 0.0;
                private String designation = "Manager";

                public Employee() {
                }

                public Employee(int eno, String name) {
                        this.eno = eno;
```

```
                this.name = name;
        }

        public Employee(double sal, String designation) {
                this.sal = sal;
                this.designation = designation;
        }

        public void getEmployeeDetails() {
                System.out.println("eno = " + eno);
                System.out.println("name = " + name);
                System.out.println("sal = " + sal);
                System.out.println("designation = " + designation);
        }

}

<!-- applicationContext.xml -->
        <?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

        <beans>
                <bean id="empref" class="com.venus.Employee">
                        <constructor-arg value="5" index="0" type="int" />
                        <constructor-arg value="VagmeeChandraGuru" index="1"
                                type="java.lang.String" />
                </bean>
                <bean id="empref1" class="com.venus.Employee">

                        <constructor-arg value="100000" index="0" type="double" />
                        <constructor-arg value="Analyst" index="1"
                                type="java.lang.String" />
                </bean>
        </beans>

<!-- <property name = "name" value = "VagmeeChandraGuru"/> →

/*EmployeeTest.java*/
        package com.venus;

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

        public class EmployeeTest {
                private static ApplicationContext context = new ClassPathXmlApplicationContext(
                        "applicationContext.xml");

                public static void main(String args[]) {
```

```
                Employee employee = (Employee) context.getBean("empref1");
                employee.getEmployeeDetails();
        }
}
```

AutoWiring:

In all the examples so far, we have had to define explicitly, via the configuration file, how individual beans are wired together. If you don't like having to wire all your components together, you can have Spring attempt to do so automatically.

By default, automatic wiring is disabled. To enable it, you specify which method of automatic wiring you wish to use using the autowire attribute of the bean you wish to automatically wire.

→ AutoWiring means automatically injecting the dependencies.
→ Instead of manually configuring the injection we done it automatically by using atuto wiring.
→ To implement autowiring we use "autowire" attribute of <bean> tag.

There are five possible values to the autowire attribute of <bean> tag.

1) **no** → It won't allow autowiring
2) **byName**
3) **constructor**
4) **byType**
5) **autodetect**

2) byName: mainly it checks for 3 conditions if all these are valid then it injects the values by setter approach.

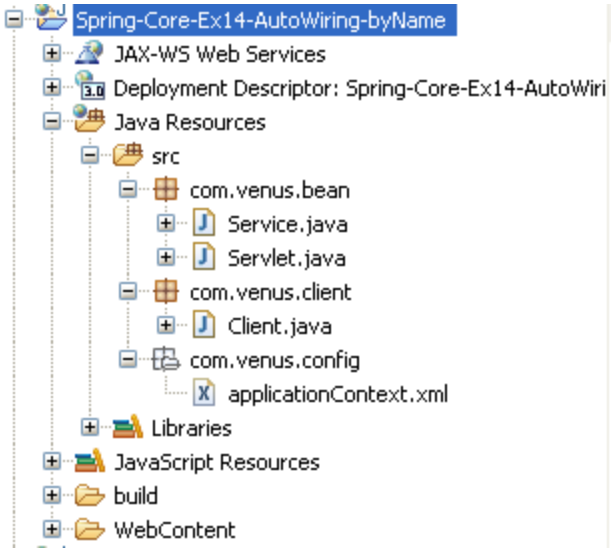
1) Dependency bean name
2) configured bean name
3) Setter method name

→ If dependency name is "**abc**", bean configuration should be "**abc**" and setter name method should be "**setAbc(Abc abc)**".

→ When it finds "**autowire=byName**" for any bean configuration, then it first checks for dependency bean name in te depedent bean, then it will checks weather any bean is configured in the spring configuration file with the same name. If it finds then it will call corresponding setter method of the dependent bean.

****Note:** It does't bother about the arg type, it only bothers the names because it is **autowiring byName**.

Spring-Core-Ex14-AutoWiring-byName




```
/*Service.java*/
	package com.venus.bean;

	public class Service {
		public void serviceMethod() {
			System.out.println("Service.serviceMethod() called");
		}

	}

/*Servlet.java*/
	package com.venus.bean;

	public class Servlet {
		private Service service;
		public void setService(Service service){//method name must match the object name
	service
			this.service = service;
			System.out.println("....AutoWiring byName Injection Happened");
		}
		public void servletMethod()
		{
			System.out.println("Servlet.servletMethod() called");
			service.serviceMethod();
		}

	}

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

```
            xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

            <bean id="service" class="com.venus.bean.Service" />
            <bean id="servletref" class="com.venus.bean.Servlet" autowire="byName" />

        </beans>


/*Client.java*/
        package com.venus.client;

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

        import com.venus.bean.Servlet;

        public class Client {
            private static ApplicationContext context = new ClassPathXmlApplicationContext(
                        "com/venus/config/applicationContext.xml");

            public static void main(String args[]) {
                Servlet servlet = (Servlet) context.getBean("servletref");
                servlet.servletMethod();
            }
        }
        /*
        Output:
            ....Injection Happened
            Servlet.servletMethod() called
            Service.serviceMethod() called
        */
```

Ex14-1-AutoWiring-byName

When using **byName** wiring, Spring attempts to wire each property to a bean of the same name. So, if the target bean has a property named foo and a foo bean is defined in the BeanFactory, the foo bean is assigned to the foo property of the target.

Program :Autowiring by Name :

Book.java

```
package com.venus.wiringbyname;

public  class Book {

   private String bookname;
   private int bookprice;

   public String getBookname() {
```

```
        return bookname;
    }
    public void setBookname(String bookname) {
        this.bookname = bookname;
    }
    public int getBookprice() {
        return bookprice;
    }
    public void setBookprice(int bookprice) {
        this.bookprice = bookprice;
    }

}
```

Categories.java

```
package com.venus.wiringbyname;

public  class Categories {

    private String name;
    private Book bk;

    public String getName() {
        return name;
    }

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

    public Book getBk() {
        return bk;
    }

    public void setBk(Book bk) {
        this.bk = bk;
    }

    public void show()
    {
        System.out.println("Categories name :"+name);
        System.out.println("Book name :"+bk.getBookname()+" and Book Price :"+bk.getBookprice());
    }

}
```

spconfig-wireName.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
	<bean id="id1" class="com.venus.wiringbyname.Categories" autowire="byName">
		<property name="name" value="General Books" />
	</bean>
	<bean id="bk" class="com.venus.wiringbyname.Book">
		<property name="bookname" value="The Kids" />
		<property name="bookprice" value="300" />
	</bean>
</beans>
```

ClientLogic.java

```
package com.venus.wiringbyname;

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

public class ClientLogic {

   public static void main(String[] args)
   {
      Resource res = new ClassPathResource("spconfig-wireName.xml");
      BeanFactory factory = new XmlBeanFactory(res);

      Object o = factory.getBean("id1");
      Categories wb = (Categories)o;

      wb.show();

   }

}
```

```
	<!--
	public class MyBean
	{
	private DemoBean db;
	public void setDb(DemoBean db)
	{
	this.db=db;
	}
	}
```

```
<beans>
	<bean id="id1" class="MyBean" autowire="byName" />
	<bean id="db" class="DemoBean" />
</beans>

In MyBean.java-->setDb(DemoBean db), our class depends on DemoBean class object, now see
in the xml file, we have given autowire=”byName“, means when ever spring container notice
autowire=”byName” then it will verifies whether the id in xml file is matching with the property
name in the MyBean or not, if yes it will
wired automatically else unwired

-->
```

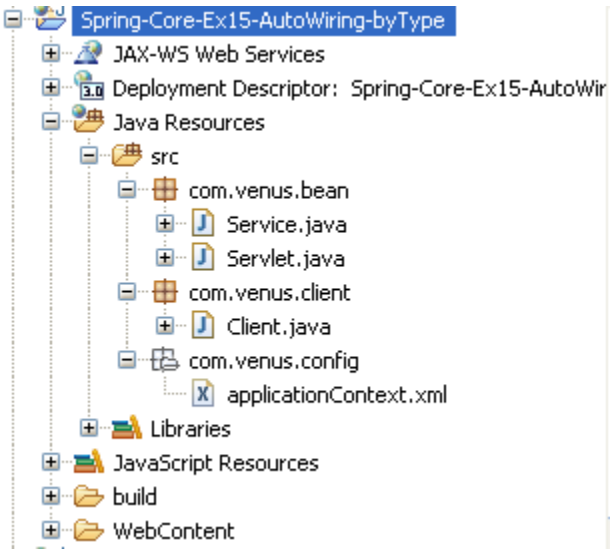
2) byType: Mainly it checks for 3 conditions if all these are valid then it injects the values by setter approach.

1) Dependency bean name
2) configured bean name
3) Setter method Argument Type

→ When it finds "**autowire=byType**" for any bean configuration, then it first checks for dependency bean Type in the dependent bean, then it will checks the spring configuration file, weather any bean is configured with the dependency type. If it finds it will check for any setter method that takes bean dependency type as an arg. If it finds then it will call that setter method.

****Note:** It doesn't bother about the setter method names, it only bothers the args Type.

Spring-Core-Ex15-AutoWiring-byType




```
/*Service.java*/
	package com.venus.bean;
```

```
	public class Service {
		public void serviceMethod()
		{
			System.out.println("Service.serviceMethod() called");
		}
	}

/*Servlet.java*/
	package com.venus.bean;

	public class Servlet {
		private Service service;
		public void setService1(Service service){ //change in method but arg must match the
								//service
			this.service = service;
			System.out.println("....Autowiring byType Injection Happened");
		}
		public void servletMethod()
		{
			System.out.println("Servlet.servletMethod() called");
			service.serviceMethod();
		}
	}

	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<bean id="service" class="com.venus.bean.Service" />
		<bean id="servletref" class="com.venus.bean.Servlet" autowire="byType" />

	</beans>


/*Client.java*/
	package com.venus.client;

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

	import com.venus.bean.Servlet;

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/applicationContext.xml");
```

```
        public static void main(String args[]) {
                Servlet servlet = (Servlet) context.getBean("servletref");
                servlet.servletMethod();
        }
}
/*
Output:
....Autowiring byType Injection Happened
Servlet.servletMethod() called
Service.serviceMethod() called
*/
```

Spring-Core-Ex15-1-AutoWiring-byType

When using byType automatic wiring, Spring attempts to wire each of the properties on the target bean automatically using a bean of the same type in the BeanFactory. So if you have a property of type String on the target bean and a bean of type String in the BeanFactory, Spring wires the String bean to the target bean's String property.

If you have more than one bean of the same type, in this case String, in the same BeanFactory, Spring is unable to decide which one to use for the automatic wiring and throws an exception.

Program AutoWiring by Type:

Book.java

```
package com.venus.wiringbytype;

public  class Book {

   private String bookname;
   private int bookprice;

   public String getBookname() {
      return bookname;
   }
   public void setBookname(String bookname) {
      this.bookname = bookname;
   }
   public int getBookprice() {
      return bookprice;
   }
   public void setBookprice(int bookprice) {
      this.bookprice = bookprice;
   }

}
```

Categories.java

```
package com.venus.wiringbytype;

public  class Categories {

   private String name;
   private Book bk;

   public String getName() {
      return name;
   }

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

   public Book getBk() {
      return bk;
   }

   public void setBk(Book bk) {
      this.bk = bk;
   }

   public void show()
   {
      System.out.println("Categories name :"+name);
System.out.println("Book name :"+bk.getBookname()+" and Book Price :"+bk.getBookprice());
   }

}
```

spconfig-wireType.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
        <bean id="id1" class="com.venus.wiringbytype.Categories" autowire="byType">
                <property name="name" value="General Books" />
        </bean>
        <bean id="SomeThing" class="com.venus.wiringbytype.Book">
                <property name="bookname" value="The Kids" />
                <property name="bookprice" value="300" />
        </bean>
</beans>
```

ClientLogic.java

```
package com.venus.wiringbytype;

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

public class ClientLogic {

   public static void main(String[] args)
   {
      Resource res = new ClassPathResource("spconfig-wireType.xml");
      BeanFactory factory = new XmlBeanFactory(res);

      Object o = factory.getBean("id1");
      Categories wb = (Categories)o;

      wb.show();

   }

}
```

/* We called "id1" from ClientLogic.java , and in spconfig.xml we have written autowire=byType, so first spring container will checks for the bean with class attribute Book [as autowire=byType and we have written private Book bk in Categories.java] and then inserts Book class properties into Book class object [jav4s.Book] and gives this book class object to Categories then injects value "General Books" into name property of Categories class. */

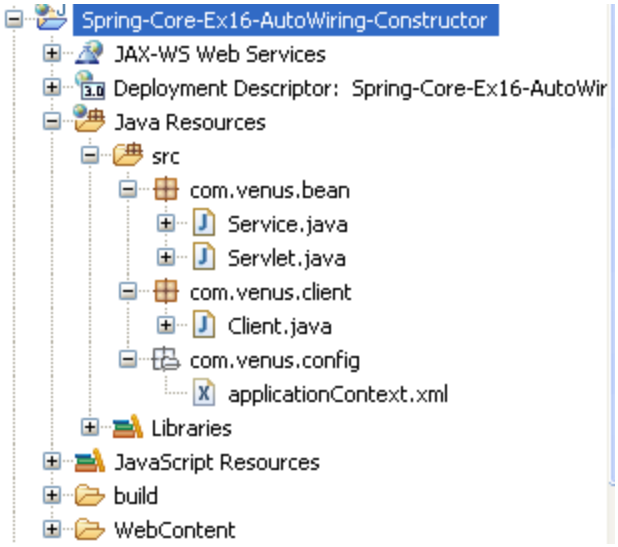
3) constructor: Mainly it checks for 3 conditions if all these are valid then it injects the values by constructor approach.

1) Dependency bean Type
2) Configured bean Type
3) Constructor Argument Type

→ When it finds **autowire=constructor** for any bean configuration, then it first checks for dependency bean Type in the dependent bean, then it will checks the spring configuration file, weather any bean is configured with the dependency type, if it finds it will check for constructor which will takes bean dependency type as an arg. If it finds then it will call corresponding constructor.

****Note:** It won't bother about the dependency name, it bothers about only dependency type, constructor arg type.

Spring-Core-Ex16-AutoWiring-Constructor




```
/*Service.java*/
	package com.venus.bean;

	public class Service {
		public void serviceMethod()
		{
			System.out.println("Service.serviceMethod() called");
		}
	}

/*Servlet.java*/
	package com.venus.bean;

	public class Servlet {
		private Service service;
		public Servlet(Service service){ //constructor
			this.service = service;
			System.out.println("....Autowiring Constructor Injection Happened");
		}
		public void servletMethod()
		{
			System.out.println("Servlet.servletMethod() called");
			service.serviceMethod();
		}
	}

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

```
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<bean id="service" class="com.venus.bean.Service" />
		<bean id="servletref" class="com.venus.bean.Servlet" autowire="constructor" />

	</beans>


/*Client.java*/
	package com.venus.client;

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

	import com.venus.bean.Servlet;

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/applicationContext.xml");

		public static void main(String args[]) {
			Servlet servlet = (Servlet) context.getBean("servletref");
			servlet.servletMethod();
		}
	}
	/*
	Output:
		....Autowiring Constructor Injection Happened
		Servlet.servletMethod() called
		Service.serviceMethod() called
	*/
```

The **constructor wiring** mode functions just like **byType** wiring, except that it uses constructors rather than setters to perform the injection. Spring attempts to match the greatest numbers of arguments it can in the constructor. So, if your bean has two constructors, one that accepts a String and one that accepts a String and an Integer, and you have both a String and an Integer bean in your BeanFactory, Spring uses the two-argument constructor.

Program AutoWiring By constructor

Spring-Core-Ex16-1-AutoWiring-Constructor

```
	package com.venus.constructor;

	public class  DemoBean
	{
```

```
    public String message;

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

    public void show()
    {
      System.out.println(message);
    }

  }
```

spconfig-con1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
<bean id="id1" class="com.venus.constructor.DemoBean">
  <constructor-arg  value="Welcome to Constructor Injection" />
<!--    <property name="message" value="Welcome to spring" />-->

</bean>
</beans>
```

ClientLogicList.java

```
package com.venus.constructor;

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

import com.venus.constructor.DemoBean;

public class ClientLogicList {

    public static void main(String[] args)
    {
        Resource res = new ClassPathResource("spconfig-con1.xml");
        BeanFactory factory = new XmlBeanFactory(res);

        Object o = factory.getBean("id1");
        DemoBean wb = (DemoBean)o;
```

```
        wb.show();

        }

    }
```

Program Autowiring bean class with two constructors

DemoBean.java

```
public class DemoBean {
        String uname, pwd;

        public DemoBean(String uname) {
                this.uname = uname;
        }

        public DemoBean(String uname, String pwd) {
                this.uname = uname;
                this.pwd = pwd;
        }

        public void show() {
                System.out.println("uname = " + uname + "   " + "Pwd = " + pwd);
        }
}
```

Demo.xml

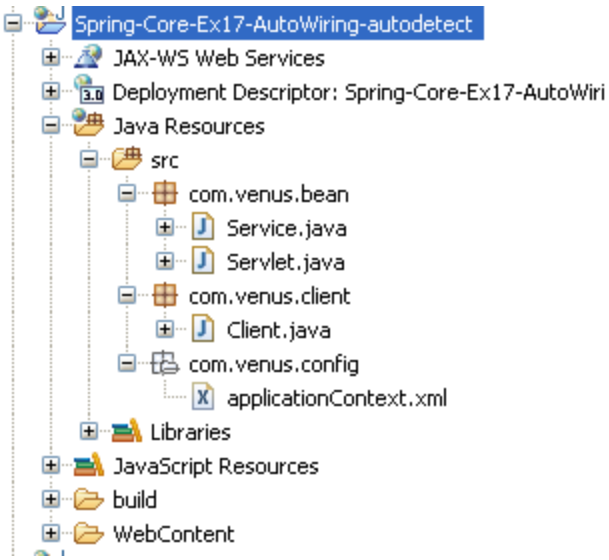
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="id1" class="DemoBean">
    <constructor-arg>
        <value> Sun </value>
    </constructor-arg>
  </bean>

  <bean   id="id2" class="DemoBean">
     <constructor-arg index="0">
        <value> Kumar </value>
     </constructor-arg>
     <constructor-arg index="1"  value="Guru"/>
 </bean>
 </beans>
```

Client.java

```
import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;

public class Client {
	public static void main(String args[]) {
		Resource res = new ClassPathResource("demo.xml");
		BeanFactory factory = new XmlBeanFactory(res);

		// Constructor with one arg.
		DemoBean ob1 = (DemoBean) factory.getBean("id1");

		// Constructor with two args.
		DemoBean ob2 = (DemoBean) factory.getBean("id2");
		ob1.show();
		ob2.show();
	}

}
```

4) autodetect: autodetect chooses "**constructor**" or "**byTtype**" through introspection of the bean class. If a default constructor is found, "**byType**" gets applied, if not found it will apply "**constructor**".

Spring-Core-Ex17-AutoWiring-autodetect




```
/*Service.java*/
	package com.venus.bean;

	public class Service {
		public void serviceMethod()
```

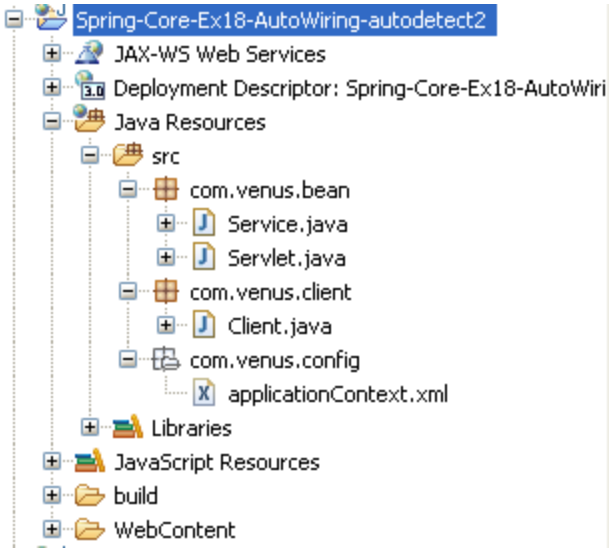
```
		{
			System.out.println("Service.serviceMethod() called");
		}

	}

/*Servlet.java*/
	package com.venus.bean;

/* In this application we provided both constructor and setter, but it injects by "constructor" approach
because there is no default constructor. */
	public class Servlet {
		private Service service;
		public Servlet(Service service){ //constructor
			this.service = service;
		System.out.println("....Autowiring \"autodetect\" byConstructor Injection Happed");
		}
		public void setService(Service service){ //setter
			this.service = service;
			System.out.println("....Autowiring Constructor Injection Happed");
		}
		public void servletMethod()
		{
			System.out.println("Servlet.servletMethod() called");
			service.serviceMethod();
		}


	}


	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<!-- <bean id="servletref" class="com.venus.bean.Servlet" autowire="constructor"
			dependency-check="objects" /> -->
		<bean id="service" class="com.venus.bean.Service" />
		<bean id="servletref" class="com.venus.bean.Servlet" autowire="autodetect" />

	</beans>


/*Client.java*/
	package com.venus.client;

	import org.springframework.context.ApplicationContext;
```

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

import com.venus.bean.Servlet;

public class Client {
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/config/applicationContext.xml");

	public static void main(String args[]) {
		Servlet servlet = (Servlet) context.getBean("servletref");
		servlet.servletMethod();
	}
}
/*
 Output:
 ....Autowiring "autodetect" byConstructor Injection Happed
Servlet.servletMethod() called
Service.serviceMethod() called
 */
```

From the above application we provided both **constructor** and **setter** in "**Servlet.java**" but it **injects** by "**constructor**" approach because there is **no default constructor**.

→ By providing default constructor

Spring-Core-Ex18-AutoWiring-autodetect2




```
/*Service.java*/
	package com.venus.bean;
```

```
	public class Service {
		public void serviceMethod()
		{
			System.out.println("Service.serviceMethod() called");
		}

	}

/*Servlet.java*/
	package com.venus.bean;

/* In this application injection is happened by using "byType" because default constructor is there.*/
	public class Servlet {
		private Service service;

		public Servlet() {
			// TODO Auto-generated constructor stub
		}

		public Servlet(Service service) { // constructor
			this.service = service;
			System.out.println("....Autowiring \"autodetect\" byType Injection Happed");
		}

		public void setService(Service service) { // setter
			this.service = service;
			System.out.println("....Autowiring Constructor Injection Happed");
		}

		public void servletMethod() {
			System.out.println("Servlet.servletMethod() called");
			service.serviceMethod();
		}

	}


	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<bean id="service" class="com.venus.bean.Service" />
		<bean id="servletref" class="com.venus.bean.Servlet" autowire="constructor" />

	</beans>
```

```
/*Client.java*/
	package com.venus.client;

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

	import com.venus.bean.Servlet;

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/applicationContext.xml");

		public static void main(String args[]) {
			Servlet servlet = (Servlet) context.getBean("servletref");
			servlet.servletMethod();
		}
	}
	/*
	Output:
	....Autowiring "autodetect" byType Injection Happened
	Servlet.servletMethod() called
	Service.serviceMethod() called
	*/
```

Now here injection is happened by using "**byType**" because **default constructor** is there.

autodetect:

The final mode, autodetect, instructs Spring to choose between the constructor and byType modes automatically. If your bean has a default (no arguments) constructor, Spring uses byType; otherwise, it uses constructor.

Program Bean autowire with dependency-check

ExampleBean1.java

```
	package com.dependencyCheck;

	public class ExampleBean1
	{
		public void m1()
		{
			System.out.println("m1 called...");
		}
	}
```

ExampleBean2.java

```
	package com.dependencyCheck;
```

```
public class ExampleBean2
{
	int k;
	ExampleBean1  eb1;
	public void setK(int k) {
		this.k = k;
	}
	public void setEb1(ExampleBean1 eb1) {
		this.eb1 = eb1;
	}
	public void m2()
	{
		eb1.m1();
		System.out.println("m2 called..");
	}
}
```

Dependency.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
		"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
  <bean id="eb1"  class="com.dependencyCheck.ExampleBean1"/>
  <bean id="id2"  class="com.dependencyCheck.ExampleBean2"  autowire="byName"
dependency-check="objects" />
</beans>
```

Client.java

```
package com.dependencyCheck;

import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
public class Client
{
	public static void main(String... args)
	{
		Resource res = new ClassPathResource("Dependency.xml");
		BeanFactory factory = new XmlBeanFactory(res);
		ExampleBean2 ob =(ExampleBean2)factory.getBean("id2");
		ob.m2();
	}
}

/*
```

```
 m1 called...
 m2 called..
*/
```

*** **Note:** It is always good to combine both "**auto-wire**" and "**dependency-check**" together, to make sure the property is always auto-wire successfully.

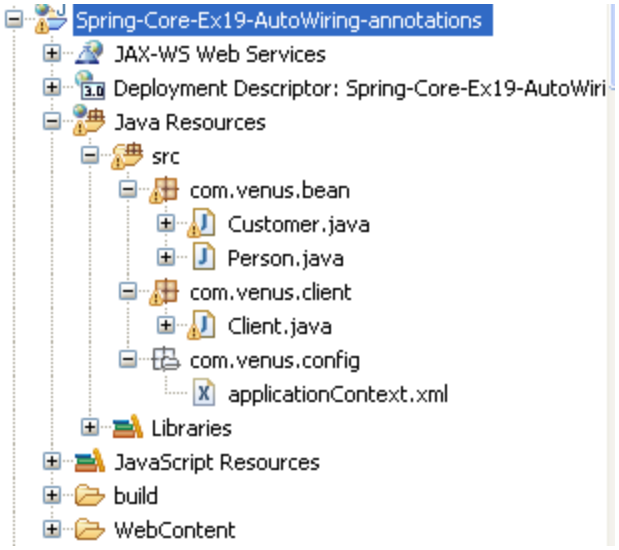
```
<bean id="person" class="com.venus.bean.person" />
<bean id="customer" class="com.venus.bean.Customer" autowire="autodetect"
        dependency-check="objects" />
```

Conclusion:

Spring auto-wiring make development faster with greater costs- it added complexity for the entire bean configuration file, and you don't even know which bean will auto wired in which bean.

In practice, wire it manually, it is always clean and work perfectly or better uses **@Autowired** annotation, which is more flexible and recommended.

Spring-Core-Ex19-AutoWiring-annotations:




```
/*Customer.java*/
        package com.venus.bean;

        import org.springframework.beans.factory.annotation.Autowired;

        public class Customer {
                @Autowired
                //@Autowired(required=false)
```

```
	private Person person;
	private int type;
	private String action;

	private Customer() {
		// TODO Auto-generated method stub
	}

/**
 * @param person
 */
@Autowired
public Customer(Person person) {
	this.person = person;
	System.out.println("..Injection Happened...Customer Constructor");
}

/**
 * @param person
 */
public void setPerson(Person person) {
	this.person = person;
	System.out.println("..Injection Happened...Setter Method");
}

/**
 * @return the type
 */
public int getType() {
	return type;
}

/**
 * @param type
 *          the type to set
 */
public void setType(int type) {
	this.type = type;
}

/**
 * @return the action
 */
public String getAction() {
	return action;
}

/**
 * @param action
```

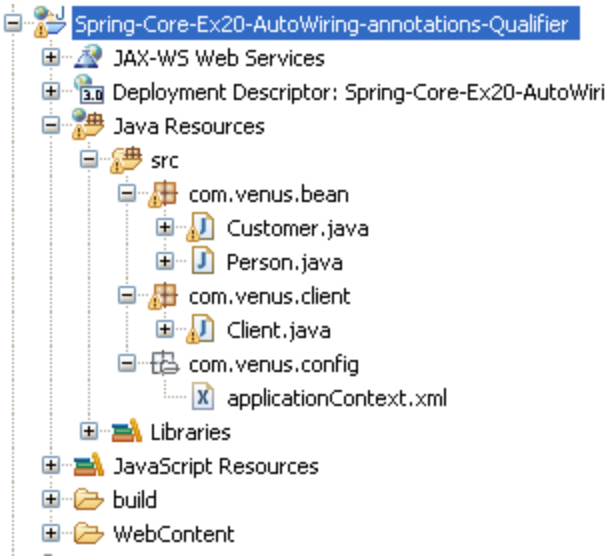
```
	 *		the action to set
	 */
	public void setAction(String action) {
		this.action = action;
	}

	/**
	 * @return the person
	 */
	public Person getPerson() {
		return person;
	}

	public void showCustomer() {
		final StringBuilder sb = new StringBuilder();
		sb.append("Customer Bean");
		sb.append("{Type=").append(type);
		sb.append(" Action = ").append(action);
		sb.append('}');
		System.out.println(sb);
		person.showPerson();
	}
}

/*Person.java*/
	package com.venus.bean;

/* In this application we provided both constructor and setter, but it injects by "constructor" approach
because there is no default constructor. */
	public class Person {
		private String name;
		private String address;
		private int age;

		/**
		 * @return the name
		 */
		public String getName() {
			return name;
		}

		/**
		 * @param name
		 *		the name to set
		 */
		public void setName(String name) {
			this.name = name;
		}
```

```
	/**
	 * @return the address
	 */
	public String getAddress() {
		return address;
	}

	/**
	 * @param address
	 *		the address to set
	 */
	public void setAddress(String address) {
		this.address = address;
	}

	/**
	 * @return the age
	 */
	public int getAge() {
		return age;
	}

	/**
	 * @param age
	 *		the age to set
	 */
	public void setAge(int age) {
		this.age = age;
	}


	  public void showPerson() {
	    final StringBuilder sb = new StringBuilder();
	    sb.append("Person Bean");
	    sb.append("{Name=").append(name);
	    sb.append(" Address = ").append(address);
	    sb.append(" Age = ").append(age);
	    sb.append('}');
	    System.out.println(sb);
	  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
	http://www.springframework.org/schema/context
```

```
            http://www.springframework.org/schema/context/spring-context-2.5.xsd">

        <context:annotation-config></context:annotation-config>
            <bean id="CustomerBean" class="com.venus.bean.Customer" >
           <property name="action" value="buy" />
                <property name="type" value="1" />
            </bean>

            <bean id="PersonBean" class="com.venus.bean.Person">
                <property name="name" value="Mee" />
                <property name="address" value="Hyd" />
                <property name="age" value="9" />
            </bean>

    <!--            <bean id="PersonBean2" class="com.venus.bean.Person">
                <property name="name" value="Mee" />
                <property name="address" value="Hyd" />
                <property name="age" value="9" />
            </bean> -->

    </beans>


/*Client.java*/
    package com.venus.client;

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

    import com.venus.bean.Customer;

    public class Client {
        private static ApplicationContext context = new ClassPathXmlApplicationContext(
                    "com/venus/config/applicationContext.xml");

        public static void main(String args[]) {
            Customer customer = (Customer) context.getBean("CustomerBean");
            customer.showCustomer();
        }
    }
```

Spring-Core-Ex20-AutoWiring-annotations-Qualifier




```
/*Customer.java*/
	package com.venus.bean;

	import org.springframework.beans.factory.annotation.Autowired;

	public class Customer {
		@Autowired
		//@Autowired(required=false)
		private Person person;
		private int type;
		private String action;

		private Customer() {
			// TODO Auto-generated method stub
		}

		/**
		 * @param person
		 */
		@Autowired
		public Customer(Person person) {
			this.person = person;
			System.out.println("..Injection Happened...Customer Constructor");
		}

		/**
		 * @param person
		 */
		public void setPerson(Person person) {
			this.person = person;
```

```
		System.out.println("..Injection Happened...Setter Method");
	}

	/**
	 * @return the type
	 */
	public int getType() {
		return type;
	}

	/**
	 * @param type
	 *            the type to set
	 */
	public void setType(int type) {
		this.type = type;
	}

	/**
	 * @return the action
	 */
	public String getAction() {
		return action;
	}

	/**
	 * @param action
	 *            the action to set
	 */
	public void setAction(String action) {
		this.action = action;
	}

	/**
	 * @return the person
	 */
	public Person getPerson() {
		return person;
	}

	public void showCustomer() {
		final StringBuilder sb = new StringBuilder();
		sb.append("Customer Bean");
		sb.append("{Type=").append(type);
		sb.append(" Action = ").append(action);
		sb.append('}');
		System.out.println(sb);
		person.showPerson();
	}
```

```
	}

/*Person.java*/
	package com.venus.bean;

/* In this application we provided both constructor and setter, but it injects by "constructor" approach
because there is no default constructor. */
	public class Person {
		private String name;
		private String address;
		private int age;

		/**
		 * @return the name
		 */
		public String getName() {
			return name;
		}

		/**
		 * @param name
		 *		the name to set
		 */
		public void setName(String name) {
			this.name = name;
		}

		/**
		 * @return the address
		 */
		public String getAddress() {
			return address;
		}

		/**
		 * @param address
		 *		the address to set
		 */
		public void setAddress(String address) {
			this.address = address;
		}

		/**
		 * @return the age
		 */
		public int getAge() {
			return age;
		}
```

```
	/**
	 * @param age
	 *            the age to set
	 */
	public void setAge(int age) {
		this.age = age;
	}


	  public void showPerson() {
	    final StringBuilder sb = new StringBuilder();
	    sb.append("Person Bean");
	    sb.append("{Name=").append(name);
	    sb.append(" Address = ").append(address);
	    sb.append(" Age = ").append(age);
	    sb.append('}');
	    System.out.println(sb);
	  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
	http://www.springframework.org/schema/context
	http://www.springframework.org/schema/context/spring-context-2.5.xsd">

	<context:annotation-config></context:annotation-config>
	<bean id="CustomerBean" class="com.venus.bean.Customer">
		<property name="action" value="buy" />
		<property name="type" value="1" />
	</bean>

	<bean id="PersonBean1" class="com.venus.bean.Person">
		<property name="name" value="Mee" />
		<property name="address" value="Hyd" />
		<property name="age" value="9" />
	</bean>

	<bean id="PersonBean2" class="com.venus.bean.Person">
		<property name="name" value="Mee2" />
		<property name="address" value="Hyd2" />
		<property name="age" value="92" />
	</bean>

</beans>
```

```
/*Client.java*/
	package com.venus.client;

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

	import com.venus.bean.Customer;

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/applicationContext.xml");

		public static void main(String args[]) {
			Customer customer = (Customer) context.getBean("CustomerBean");
			customer.showCustomer();
		}
	}
```

Excluding a bean from being available for autowiring: On a **per-bean** basis totally exclude a bean from being an autowire candidate.

→ When configuring beans using spring's XML format, the "autowiring-candidate" attribute of the **<bean/>** element can be set to "**false**". This has the effect of making the container totally exclude that specific bean definition from being available to the autowiring infrastructure.

→ This can be useful when you have a bean that you absolutely never ever want to have injected into other beans via autowiring.

null: <null/> element is used to handle null values.

→ null value handling is as follows:

```
<bean class="EmployeeBean">
	<property name="email">
		<null/>
	</property>
</bean>
```

The above xml is equivalent to the following java code :

```
EmployeeBean.setEmail(null);
```

→ Spring treats empty args for properties and the like as empty Strings.

```
<bean class="EmployeeBean">
        <property name="email">
                <value />
        </property>
</bean>
```

The above xml is equivalent to the following java code :

```
EmployeeBean.setEmail("");
```

Spring & Collections

Strongly-typed collection:

→ If we are using java5 or java6, we will be aware that it is possible to have strongly typed collections(using generic types). i.e., it is possible to declare a "**Collection**" type such that it can only contain "**String**" elements.

→ If we are using Spring to dependency inject a strongly-typed "**Collection**" into a bean, we can take advantage of Spring's type-conversion support such that the elements of our strongly-type "**Collection**" instances will be converted to the appropriate type prior to being added to the "**Collection**".

```
package com.venus;

import java.util.Map;

public class Foo {
        private Map<String, Float> accounts;

        public void setAccounts(Map<String, Float> accounts) {
                this.accounts = accounts;
        }
}
```

```
<beans>
        <bean id="foo" class="com.venus.Foo">
                <property name="accounts">
                        <map>
                                <entry key="one" value="9.99" />
                                <entry key="two" value="2.75" />
                                <entry key="three" value="3.55" />
                        </map>
                </property>
        </bean>
</beans>
```

→ When the "accounts" property of the "foo" bean is being prepared for injection, the generics

information about the element type of the strongly-typed "Map<String, Float>" is actually available via reflection, and so Spring's type conversion infrastructure will actually recognize the various value elements as being of type "Float" and so the String values "9.99", "2.75", and "3.55" will be converted into an actual Float type.

List

WelcomeBean.java

```
package com.venus.list;

import java.util.List;

public class WelcomeBean {

      private List studentsData;

    public void setStudentsData(List studentsData) {
       this.studentsData = studentsData;
    }

    public void show()
      {
         System.out.println(studentsData);
      }

}
```

spconfig-list.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
  <bean id="id2" class="com.venus.list.WelcomeBean">

    <property name="studentsData" >
      <list>
        <value>venus</value>
        <value>sun</value>
        <value>raam</value>
        <value>venus</value>
      </list>
     </property>
  </bean>
</beans>
```

ClientLogicList.java

```
package com.venus.list;

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

import com.venus.list.WelcomeBean;

public class ClientLogicList {

   public static void main(String[] args)
   {
      Resource res = new ClassPathResource("spconfig-list.xml");
      BeanFactory factory = new XmlBeanFactory(res);

      Object o = factory.getBean("id2");
      WelcomeBean wb = (WelcomeBean)o;

   wb.show();

   }

}
```

Set:

WelcomeBean.java

```
package com.venus.set;

import java.util.Set;
public class WelcomeBean {

     private Set studentsData;


   public void setStudentsData(Set studentsData) {
      this.studentsData = studentsData;
   }

   public void show()
     {
        System.out.println(studentsData);
     }
```

```
}
```

spconfig-set.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
  <bean id="id1" class="com.venus.set.WelcomeBean">

    <property name="studentsData" >
      <set>
        <value>Venus</value>
        <value>sun</value>
        <value>Raam</value>
        <value>sun</value>
        <value>sun</value>
      </set>
    </property>
  </bean>
</beans>
```

ClientLogic.java

```
package com.venus.set;

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

import com.venus.set.WelcomeBean;

public class ClientLogic {

   public static void main(String[] args)
   {
      Resource res = new ClassPathResource("spconfig-set.xml");
      BeanFactory factory = new XmlBeanFactory(res);

      Object o = factory.getBean("id1");
      WelcomeBean wb = (WelcomeBean)o;

   wb.show();

   }
```

```
}
```

Map:

```
import  java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class WelcomeBean {

   private Map data;

   public void setData(Map data) {
      this.data = data;
   }

   public void show()
     {
        Set s=data.entrySet();
        Iterator it = s.iterator();
        while(it.hasNext())
        {
           Map.Entry me = (Map.Entry)it.next();
           System.out.println(me.getKey()+ " - "+me.getValue());
        }
     }

   }
```

spconfig-map.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
<bean id="id1" class="com.venus.map.WelcomeBean">

<property name="data" >
  <map>
     <entry key="k1">
        <value>10</value>
     </entry>
     <entry key="k2">
        <value>java4s</value>
     </entry>
     <entry key="k3">
```

```
            <value>10.55</value>
        </entry>
    </map>
</property>

</bean>
</beans>
```

```
package com.venus.map;

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

import com.venus.map.WelcomeBean;

public class ClientLogic {

   public static void main(String[] args)
   {
      Resource res = new ClassPathResource("spconfig-map.xml");
      BeanFactory factory = new XmlBeanFactory(res);

      Object o = factory.getBean("id1");
      WelcomeBean wb = (WelcomeBean)o;

      wb.show();

   }

}
```

To work with multiple spring configuration files:

→ When the spring configuration file size is huge, it is difficult to main, so in that scenarios we generally have multiple configuration files.

→ We can work with multiple spring configuration files in two ways

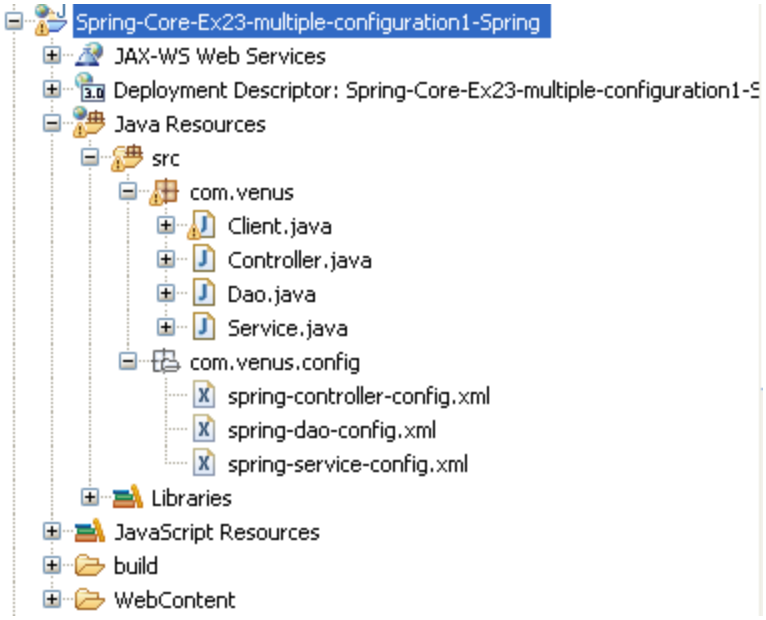
→ By importing other spring file into the current configuration file

Ex: <import resource="other-spring-beans.xml"/>

→ By passing spring bean configuration files as a string array to the constructor of the container

→ new ClassPathXmlApplicationContext(new String[] {"beans-1.xml", "beans-2.xml", "beans-3.xml"});

Spring-Core-Ex23-multiple-configuration1-Spring



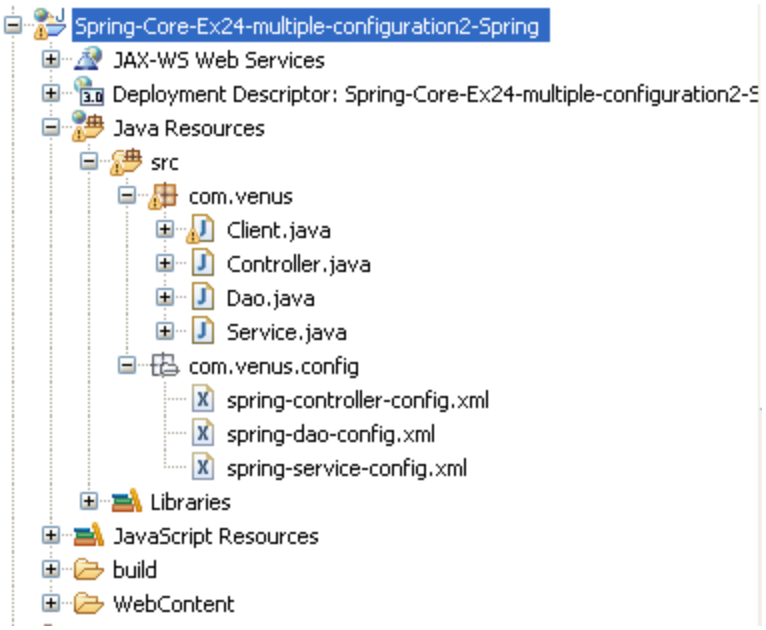

```
/*Dao.java*/
	package com.venus;

	public class Dao {
		public void daoMethod() {
			System.out.println("Dao.daoMethod()");
		}
	}

/*Service.java*/
	package com.venus;

	public class Service {
		private Dao dao;

		public void setDao(Dao dao) {
			this.dao = dao;
		}

		public void serviceMethod() {
			System.out.println("Service.serviceMethod()");
			dao.daoMethod();
		}
	}

/*Controller.java*/
```

```
package com.venus;

public class Controller {
	private Service service;

	public void setService(Service service) {
		this.service = service;
	}

	public void controllerMehtod() {
		System.out.println("Controller.controllerMethod()");
		service.serviceMethod();
	}
}
```

```
<!-- spring-dao-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<bean id="daoRef" class="com.venus.Dao">
		</bean>

	</beans>

<!-- spring-service-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<import resource="spring-dao-config.xml" />
		<bean id="serviceRef" class="com.venus.Service">
			<property name="dao" ref="daoRef"></property>
		</bean>

	</beans>

<!-- spring-controller-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<import resource="spring-service-config.xml" />
		<bean id="ControllerRef" class="com.venus.Controller">
			<property name="service" ref="serviceRef"></property>
		</bean>
```

```
	</beans>


/*Client.java*/
	package com.venus;

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

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/spring-controller-config.xml");

		public static void main(String args[]) {
			Controller controller = (Controller) context.getBean("ControllerRef");
			controller.controllerMehtod();
		}
	}

	/*
		Output:
			Controller.controllerMethod()
			Service.serviceMethod()
			Dao.daoMethod()
	 */
```

Spring-Core-Ex24-multiple-configuration2-Spring

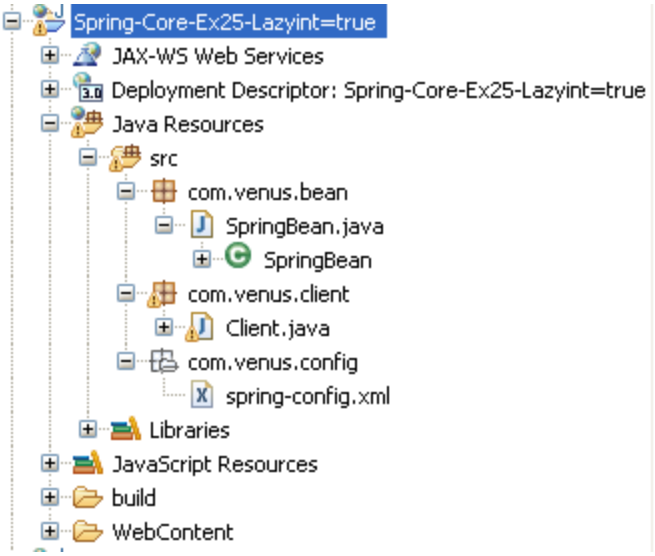



```
/*Dao.java*/
	package com.venus;

	public class Dao {
		public void daoMethod() {
			System.out.println("Dao.daoMethod()");
		}
	}

/*Service.java*/
	package com.venus;

	public class Service {
		private Dao dao;

		public void setDao(Dao dao) {
			this.dao = dao;
		}

		public void serviceMethod() {
			System.out.println("Service.serviceMethod()");
			dao.daoMethod();
		}
	}

/*Controller.java*/
	package com.venus;
```

```
	public class Controller {
		private Service service;

		public void setService(Service service) {
			this.service = service;
		}

		public void controllerMehtod() {
			System.out.println("Controller.controllerMethod()");
			service.serviceMethod();
		}
	}

<!-- spring-dao-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<bean id="daoRef" class="com.venus.Dao">
		</bean>

	</beans>

<!-- spring-service-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<import resource="spring-dao-config.xml" />
		<bean id="serviceRef" class="com.venus.Service">
			<property name="dao" ref="daoRef"></property>
		</bean>

	</beans>

<!-- spring-controller-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
	"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

	<beans>
		<import resource="spring-service-config.xml" />
		<bean id="ControllerRef" class="com.venus.Controller">
			<property name="service" ref="serviceRef"></property>
		</bean>

	</beans>
```

```
/*Client.java*/
	package com.venus;

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

	public class Client {
		private static String[] configs = {
				"com/venus/config/spring-controller-config.xml",
				"com/venus/config/spring-service-config.xml",
				"com/venus/config/spring-dao-config.xml" };
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				configs);

		public static void main(String args[]) {
			Controller controller = (Controller) context.getBean("ControllerRef");
			controller.controllerMehtod();
		}
	}

	/*
		Output:
			Controller.controllerMethod()
			Service.serviceMethod()
			Dao.daoMethod()
	 */
```

Spring-Core-Ex25-Lazyint=true

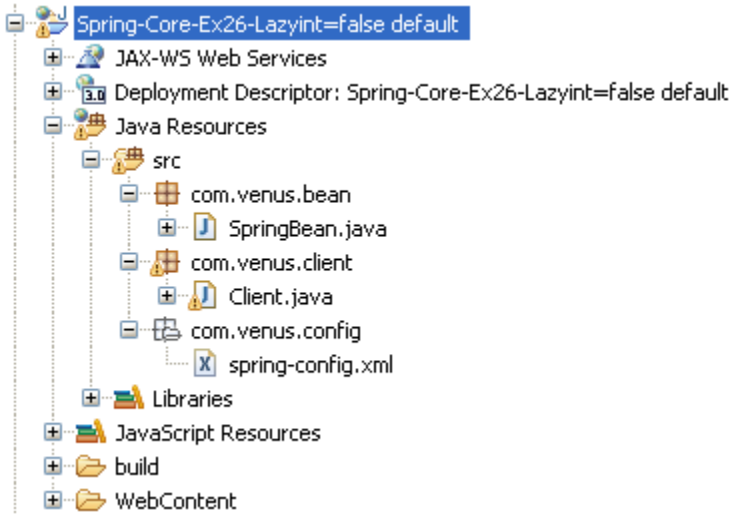



```
/*SpringBean.java*/
	package com.venus.bean;

	public class SpringBean {
		public SpringBean() {
			System.out.println("SpringBean is Created...");
		}
	}

<!-- spring-config.xml  -->
	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<bean id="sbref" class="com.venus.bean.SpringBean" lazy-init="true" />
	</beans>


/*Client.java*/
	package com.venus.client;

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

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/spring-config.xml");
```

```
		public static void main(String args[]) {
			System.out.println("....Lazy init testing....");
		}
	}

	/*
	Output:
		....Lazy init testing....
	*/
```

Spring-Core-Ex26-Lazyint=false default

```
/*SpringBean.java*/
	package com.venus.bean;

	public class SpringBean {
		public SpringBean() {
			System.out.println("SpringBean is Created...");
		}

	}
```




```
<!-- spring-config.xml  -->
	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

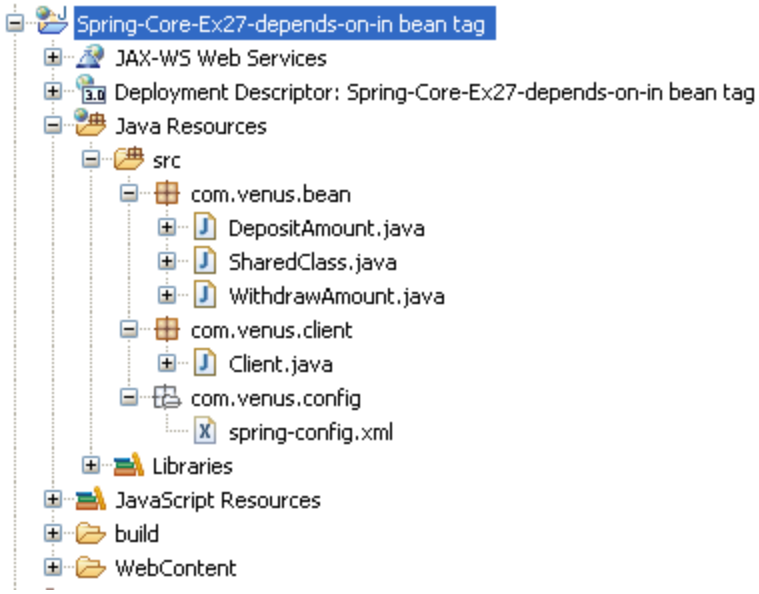
```
		<bean id="sbref" class="com.venus.bean.SpringBean" />
		<!-- <bean id="sbref" class="com.venus.bean.SpringBean" lazy-init="false"/> -->
	</beans>

/*Client.java*/
	package com.venus.client;

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

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/spring-config.xml");

		public static void main(String args[]) {
			System.out.println("....Lazy init=false testing....");
		}
	}
	/*
	Output:
		SpringBean is Created...
		....Lazy init=false testing....
	*/
```

depends-on attribute of <bean> tag:

→ Sometimes some bean creation is meaningful only when its dependency is created,

→ For example, Vehicle bean creation is meaningful, when the Petrol bean is available.

Spring-Core-Ex27-depends-on-in bean tag




```
/*SharedClass.java*/
	package com.venus.bean;

	public class SharedClass {
		private static int amount;

		public static int getAmount() {
			return amount;
		}

		public static void setAmount(int Amount) {
			amount = Amount;
		}
	}

/*DepositAmount.java*/
	package com.venus.bean;

	public class DepositAmount {
		/* creates a new instance of DepositAmount */
		public DepositAmount() {
			System.out.println("Before Depositing : " + SharedClass.getAmount());
			SharedClass.setAmount(SharedClass.getAmount() + 100);
			System.out.println("After Depositing : " + SharedClass.getAmount());
		}
	}

/*WithdrawAmount.java*/
	package com.venus.bean;
```

```
	public class WithdrawAmount {
		/* Creates a new instance of WithdrawAmount */
		public WithdrawAmount() {
			System.out.println("Before Withdrawal :" + SharedClass.getAmount());
			SharedClass.setAmount(SharedClass.getAmount() - 100);
			System.out.println("After Withdrawal :" + SharedClass.getAmount());
		}
	}

<!-- spring-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<bean id="withdraw" class="com.venus.bean.WithdrawAmount"
			depends-on="deposit" />
		<!-- <bean id="withdraw" class="com.venus.bean.WithdrawAmount"/> -->

		<bean id="deposit" class="com.venus.bean.DepositAmount" />

	</beans>


/*Client.java*/
package com.venus.client;

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

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/spring-config.xml");

		public static void main(String args[]) {
			System.out.println("....depends-on....");
			context.getBean("withdraw");
			context.getBean("deposit");
		}
	}

	/*
	Output:
		Before Depositing : 0
		After Depositing : 100
		Before Withdrawal :100
		After Withdrawal :0
```

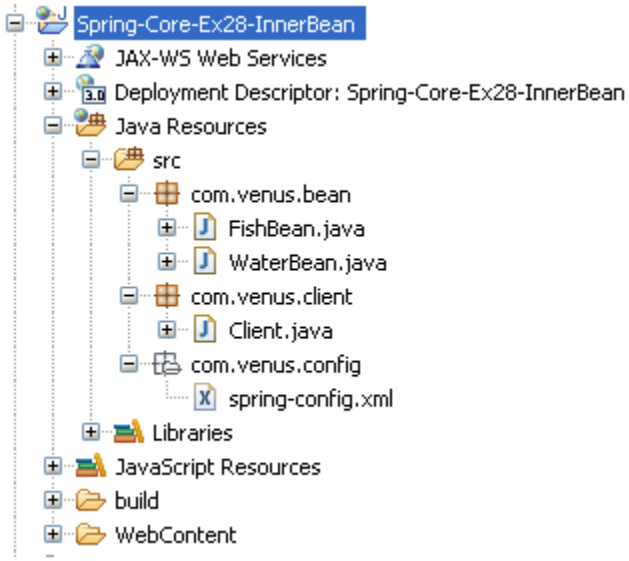
```
        ....depends-on....
    */
```

Inner bean configuration:

→ A Spring bean configured with in <property> or <constructor-arg> tag using <bean> tag is known as inner bean configuration.

→ While configuring inner bean name/id attributes are not useful. So we say inner bean also called anonymous bean.

Spring-Core-Ex28-InnerBean




```
/*FishBean.java*/
    package com.venus.bean;

    public class FishBean {
        /* creates a new instance of DepositAmount */
        public void swim() {
            System.out.println(" Fishes are swimming: ~:> ~:> ~:> ~:> ~:> ~:>");
        }
    }

/*WaterBean.java*/
    package com.venus.bean;

    public class WaterBean {
        private FishBean fishBean;
```

```
		public void setFishBean(FishBean fishBean) {
			this.fishBean = fishBean;
		}

		public void flowing() {
		System.out.println("Water is flowing :~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
			fishBean.swim();
		System.out.println("Water is flowing :~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
		}
	}

<!-- spring-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<bean id="water" class="com.venus.bean.WaterBean">
			<property name="fishBean">
				<bean class="com.venus.bean.FishBean"></bean>
			</property>
		</bean>

	</beans>

/*Client.java*/
	package com.venus.client;

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

	import com.venus.bean.WaterBean;

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/spring-config.xml");

		public static void main(String args[]) {
			WaterBean waterBean = (WaterBean) context.getBean("water");
			waterBean.flowing();
		}
	}

	/*
	Output:
		Water is flowing :~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
		 Fishes are swimming: ~:> ~:> ~:> ~:> ~:> ~:>
```

```
        Water is flowing :~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    */
```

MethodInjection

→ If dependent, dependency both are singleton(scope="singleton") objects, then there is no problem in the injection of dependency into dependent.

→ If dependent, dependency both are prototype(scope="prototype") objects, then there is no problem in the injection of dependency into dependent.

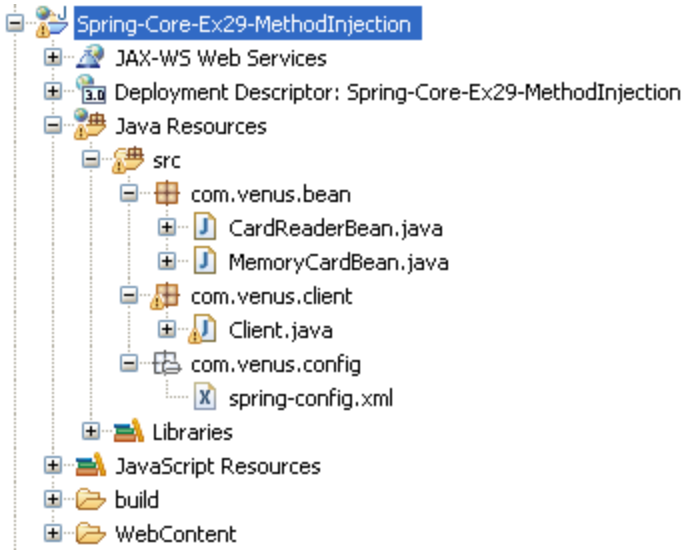
→ If dependent is prototype (scope="prototype") object , dependency is singleton (scope="singleton") objects, then there is no problem in the injection of dependency into dependent.

→ If dependent is singleton (scope="singleton") object , dependency is prototype (scope="prototype") objects, then there is problem in the injection of dependency into dependent.

→ Because dependent object created only once, but dependent object we are expecting as prototype. but as because of dependent object creation happens only once, then there is no chance of creating multiple dependency objects, and inject them into singleton dependent object.

→ To solve this problem, spring has given feature called "Method Injection".

→ Lookup method injection refers to the ability of the container to override methods on container managed beans, to return the result of lookup another named bean in the container. The lookup will typically be of a prototype bean as in the scenario described above. The spring framework implements this method injection by dynamically generating a subclass overriding the method, using bytecode generation via the CGLIB library.



Spring-Core-Ex29-MethodInjection

```
/*MemoryCardBean.java*/
	package com.venus.bean;

	public class MemoryCardBean {
		public void provideData() {
		System.out.println("MemoryCardReaderBean.providing data to card reader. ..");
		}
	}

/*CardReaderBean.java*/
	package com.venus.bean;

	public abstract class CardReaderBean {

		public abstract MemoryCardBean getMemoryCardBean();

		public void readDataFromCard() {
			System.out.println(getMemoryCardBean());
			System.out.println("CardReaderBean.reading data from card. ..");
			getMemoryCardBean().provideData();
		}
	}

<!-- spring-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

		<bean id="memoryCardBean" class="com.venus.bean.MemoryCardBean"
			scope="prototype">
		</bean>
		<bean id="cardReaderBean" class="com.venus.bean.CardReaderBean"
			scope="singleton">
			<lookup-method name="getMemoryCardBean" bean="memoryCardBean" />
		</bean>

	</beans>


/*Client.java*/
	package com.venus.client;

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

	import com.venus.bean.CardReaderBean;
```

```
public class Client {
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/config/spring-config.xml");

	public static void main(String args[]) {
		System.out.println("First time calling method on CardReader...");
		CardReaderBean cardReaderBean = (CardReaderBean) context
				.getBean("cardReaderBean");
		cardReaderBean.readDataFromCard();

		System.out.println("Second time calling method on CardReader...");
		cardReaderBean.readDataFromCard();

		System.out.println("Third time calling method on CardReader...");
		cardReaderBean.readDataFromCard();

	}
}
```

PropertyEditor:

→ PropertyEditor object is used to convert String representation of data into Object representation and vice-versa.

→ PropertyEditor is an interface that belongs to java bean API.

Benefits of Property editors in spring: If we use property editors in spring, we get the flexibility in populating the bean fields directly with string type even though bean fields are of some object type.

Built-in spring framework property editors:

ByteArrayPropertyEditor, ClassEditor, CustomBooleanEditor, LocaleEditor, URLEditor, FileEditor, CustomCollectionEditor.. etc like this we have so many built-in property editors are there. These all are used by spring framework internally.

CustomPropertyEditor: means our own property editor class used to perform our own application specific conversations.

Steps to develop custom property editor:

Step1: Develop a spring bean(dependency bean) whose values are populated from String representation to object representation. In our project the dependency bean is ContactNumber.java

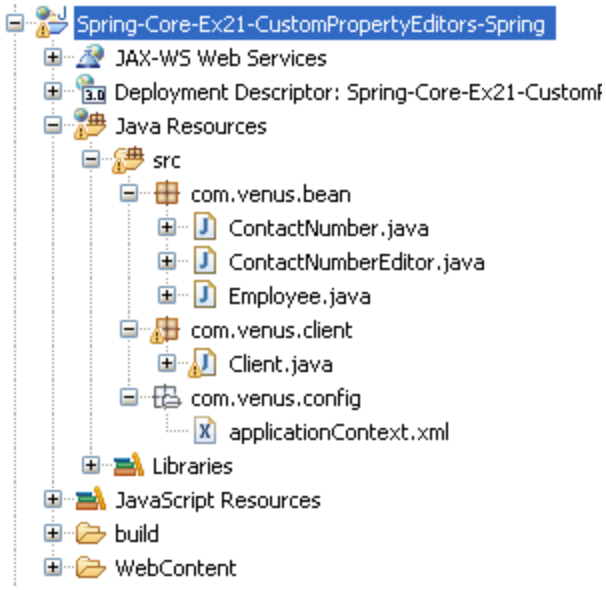
Step2: Develop the custom property editor class by extending java.beans.PropertyEditorSupport class. In our project the editor bean is ContactNumberEditor.java

***Note:** CustomPropertyEditor class name should be suffixed with Editor to the dependency bean class and should be placed in the same package where dependency bean resides. If we do so we no need to perform 4th step.

Step3: Override "setAsText(String value)" method in the custom property editor class and implement convertion code(String to required object). After conversion call "setValue(ConvertedObject)" method.

Step4: Configure the CustomPropertyEditor in the configuration file(applicationContext.xml) by configuring the class "org.springframework.beans.factory.config.CustomEditorConfigurer" with property name = "customEditors".

Spring-Core-Ex21-CustomPropertyEditors-Spring




```
/*ContactNumber.java*/
	package com.venus.bean;

	public class ContactNumber {
		private String countryNumber;
		private String stdCode;
		private String actualNumber;
		/**
		 * @param countryNumber
		 * @param stdCode
		 * @param actualNumber
		 */
		public ContactNumber(String countryNumber, String stdCode,
				String actualNumber) {
			this.countryNumber = countryNumber;
			this.stdCode = stdCode;
			this.actualNumber = actualNumber;
		}
		public String getContactNumber(){
```

```
                return countryNumber+"-"+stdCode+"-"+actualNumber;
        }


    }

/*Employee.java*/
    package com.venus.bean;

    public class Employee {
        private int eno;
        private String name;
        private String city;
        private ContactNumber contactNumber;

        /**
         * @param eno
         *          the eno to set
         */
        public void setEno(int eno) {
            this.eno = eno;
        }

        /**
         * @param name
         *          the name to set
         */
        public void setName(String name) {
            this.name = name;
        }

        /**
         * @param city
         *          the city to set
         */
        public void setCity(String city) {
            this.city = city;
        }

        /**
         * @param contactNumber
         *          the contactNumber to set
         */
        public void setContactNumber(ContactNumber contactNumber) {
            this.contactNumber = contactNumber;
        }

        public void displayDetails() {
            System.out.println("Eno = " + eno);
```

```
                System.out.println("Name = " + name);
                System.out.println("City = " + city);
                System.out.println("Contact Number = "
                                + contactNumber.getContactNumber());
        }

    }

/*ContactNumberEditor.java*/
    package com.venus.bean;

    import java.beans.PropertyEditorSupport;

    public class ContactNumberEditor extends PropertyEditorSupport {
        @Override
        // container will pass injected contact number as a String
        public void setAsText(String text) throws IllegalArgumentException {
                String[] token = text.split("-");
                ContactNumber contactNumber = new ContactNumber(token[0], token[1],
                                token[2]);
                setValue(contactNumber);
        }
    }

    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

        <bean id="EmployeeBean" class="com.venus.bean.Employee">
            <property name="eno" value="100"></property>
            <property name="name" value="Guru"></property>
            <property name="city" value="Hyd"></property>
            <property name="contactNumber" value="91-9523-789395"></property>
        </bean>
        <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
            <property name="customEditors">
                <map>
                    <entry key="com.venus.bean.ContactNumber">
                        <bean
    class="com.venus.bean.ContactNumberEditor"></bean>
                    </entry>
                </map>
            </property>
        </bean>
```

```
	</beans>


/*Client.java*/

	package com.venus.client;

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

	import com.venus.bean.Employee;

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/applicationContext.xml");

		public static void main(String args[]) {
			Employee employee = (Employee) context.getBean("EmployeeBean");
			employee.displayDetails();
		}
	}
```

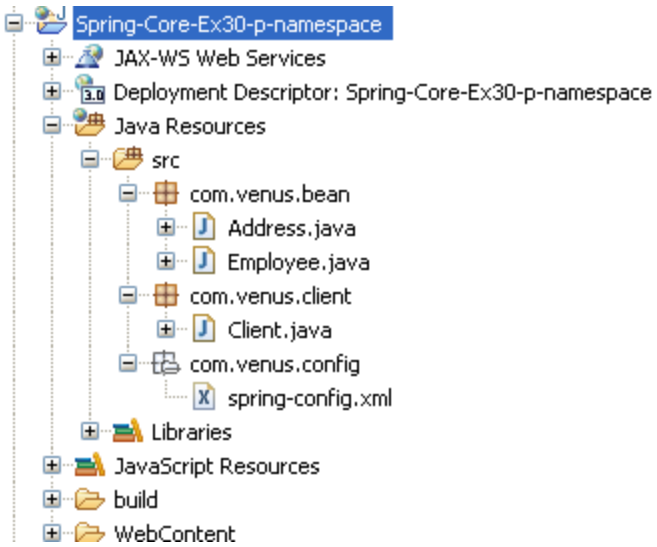
p-namespace:

→ Use the special **p-namespace** to limit the amount of XML you have to write, to configure your beans.

→ Instead of using nested property elements =, using the **p-namespace** you can use attributes as part of the bean element that descibe your property values. The values of the attribute will be taken as the values for your properties.

→ **p-namespace** is used for setter injection.

Spring-Core-Ex30-p-namespace




```
/*Address.java*/
	package com.venus.bean;

	public class Address {
		private String hno;
		private String city;

		/**
		 * @return the hno
		 */
		public String getHno() {
			return hno;
		}

		/**
		 * @param hno
		 *            the hno to set
		 */
		public void setHno(String hno) {
			this.hno = hno;
		}

		/**
		 * @return the city
		 */
		public String getCity() {
			return city;
		}

		/**
		 * @param city
		 *            the city to set
```

```
		 */
		public void setCity(String city) {
			this.city = city;
		}

	}

/*Employee.java*/
	package com.venus.bean;

	public class Employee {
		private int eno;
		private String ename;
		private Address address;

		public Employee() {
		}

		/**
		 * @param eno
		 *		the eno to set
		 */
		public void setEno(int eno) {
			this.eno = eno;
		}

		/**
		 * @param ename
		 *		the ename to set
		 */
		public void setEname(String ename) {
			this.ename = ename;
		}

		/**
		 * @param address
		 *		the address to set
		 */
		public void setAddress(Address address) {
			this.address = address;
		}

		public void getEmployeeDetails() {
			System.out.println("Employee Details are .....");
			System.out.println("eno = " + eno);
			System.out.println("name = " + ename);
			System.out.println("hno = " + address.getHno());
			System.out.println("city = " + address.getCity());
		}
```

```
	}

<!-- spring-config.xml -->
	<?xml version="1.0" encoding="UTF-8"?>
	<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:p="http://www.springframework.org/schema/p"
	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="address" class="com.venus.bean.Address"> <property name="hno"
			value="4-11" /> <property name="city" value="Hyderabad" /> </bean> <bean
	id="employee" class="com.venus.bean.Employee"> <property name="eno" value="100" />
	 <property name="ename" value="Guru" /> <property name="address" ref="address" />
	</bean> →

		<bean id="address" class="com.venus.bean.Address" p:hno="4-11"
			p:city="Hyderabad" />
		<bean id="employee" class="com.venus.bean.Employee" p:eno="100"
			p:ename="Guru" p:address-ref="address" />

	</beans>

/*Client.java*/
	package com.venus.client;

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

	import com.venus.bean.Employee;

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/spring-config.xml");

		public static void main(String args[]) {
			Employee employee = (Employee) context.getBean("employee");
			employee.getEmployeeDetails();

		}
	}
	/*
	Output:
		Employee Details are .....
		eno = 100
		name = Guru
		hno = 4-11
		city = Hyderabad
```

```
*/
```

Internationalization(I18N): In software system Internationalization means based on the users country and language we need to display respective content in the browser.

→ In JAVA in order to get the user country and language we can achieve by using "**java.util.Locale**" object.

LOCALE = Language + Country

→ A "**java.util.Locale**" is a lightweight object that contains only a few important members:

- → A language code
- → An optional country or region code
- → An optional variant (in India it is called as State) code

→**The notataion of Locale:**

<language code>[_<country code>[_<variant code>]]

→ In the above notation each code is separated with an underscore symbol(_).
The country code and variant codes are optional.

→ **Language codes:** Language codes are defined by ISO 639 that assigns two and three letter codes to most languages of the world. Locale uses the two-letter codes to identify the target language.

→ Language code Examples in the ISO 639 Standard

Arabic → Ar
German→ De
English→ En
Spanish→ Es
Japanese → Ja
Hebrew→ He

→ **Country(Region) Codes:** Country codes are defined by ISO 3166, another international standard. It defines two and three letter abbreviations for each country or major region in the world. In contrast to the language code, country codes are set uppercase.

→ Some country codes defined in the ISO 3166 Standard

United States → US
Canada → CA
France → FR
Japan → JP
Germany → DE

→ **Variant(State) code:** To provide additional functionality or customization we use Variant code,

that isn't possible with just a language and country.

Ex: de_DE → German-speaking German locale
de_DE_EURO → German-Speaking German locale with a euro variant
en_US_siliconValley → English speaking US locale for silicon valley people.

→ In java syntax of above notation by three constructors:
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)

Example on the notations:

```
//Creates a generic English-speaking locale.
Locale locale1 = new Locale("en");

//Creates an English-speaking, Canadian locale.
Locale locale2 = new Locale("en","CA");

//Creates a very specific English-speaking, US locale for Silicon Valley
Locale locale2 = new Locale("en","US","SiliconValley");
```

Steps to write I18N application in Spring:

Step1: Create the required properties files:

Syntax for naming the properties file:
We can write this in three ways:
1. Propertiesfilename_languagecode.properties
2. Propertiesfilename_languagecode_countrycode.properties
3.Propertiesfilename_languagecode_countrycode_variantcode.properties

Our project property files:

In this we have created foue properties files. The properties file should be in the form of key and value pair. In our properties files the key is "welcome.message".

1) ApplicationResources_en_US_SiliconVally.properties
welcome.message=welcome to US silicon vally english users {0}.{1}
{0}.{1} used to pass some values dynamically, called as place holders.

2) ApplicationResources_fr_CA.properties
welcome.message=welcome to French CANADA english users

3) ApplicationResources_zh_CN.properties
welcome.message=welcome to Chineese english users

4) ApplicationResources.properties
welcome.message=welcome to users

Note: In the above properties file we didn't mention any language code and country code this will execute by default if it doesn't find any codes.

Step2: Configure the properties files in the configuration file.

→ Inject the properties files prefix to spring provided bean "ResourceBundleMessageSource" using "basename" or "basenames" values for the "name" attribute of "<bean>" tag and also the id reference for this bean should be "messageSource".

→ If we are injecting one properties file then we can use "basename" or "basenames" for the "name" attribute without "<list>" tag.

→ In case if we are injecting more than one property file then we should use only "basenames" for the "name" attribute with "<list>" tag.

Client Program: Here to get the language content we should call "getMessage(key, arguments, locale)" method of ApplicationContext of by passing → 1.Key

→

2.Arguments to the message

→ 3.

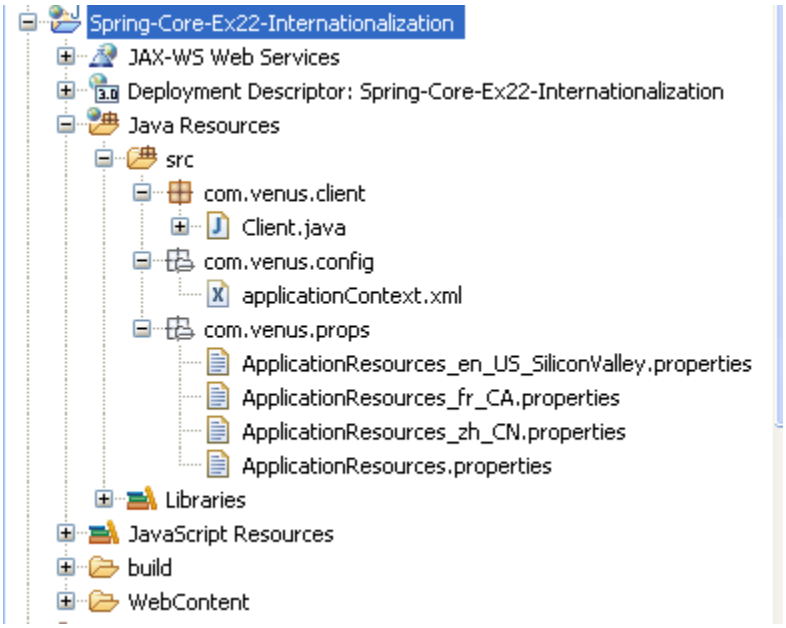
Locale Object

In our project:

Key → welcome.message

Message Argument → dynamic values for the place holders i.e.,{0}{1}

****Note:** If we don't have any arguments then we should pass "null" as an argument.



Spring-Core-Ex22-Internationalization

ApplicationResources_en_US_SiliconValley.properties

welcome.message=This is the **{0}{1}** Welcome message

ApplicationResources_fr_CA.properties

```
welcome.message=This is CANADIAN French Welcome message
```

ApplicationResources_zh_CN.properties

```
welcome.message=This is CHINESE Welcome message
```

ApplicationResources.properties

```
welcome.message=This is the DEFAULT Welcome message
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
	http://www.springframework.org/schema/context
	http://www.springframework.org/schema/context/spring-context-2.5.xsd">

	<bean id="messageSource"
		class="org.springframework.context.support.ResourceBundleMessageSource">
		<property name="basename" value="com.venus.props.ApplicationResources">
		</property>
	</bean>
</beans>
```

```
/*Client.java*/
	package com.venus.client;

	import java.util.Locale;

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

	public class Client {
		private static ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/applicationContext.xml");

		public static void main(String args[]) {
			String message = context.getMessage("welcome.message", new Object[] {
					"US", "English" }, new Locale("en", "US", "SiliconValley"));
			System.out.println(message);

		message = context.getMessage("welcome.message", null, new Locale("zh","CN"));
			System.out.println(message);
```

```
                message = context.getMessage("welcome.message", null, new Locale("Ar"));
                System.out.println(message);
        }
}

/*
  Output:
    This is the USEnglish Welcome message
    This is CHINESE Welcome message
    This is the DEFAULT Welcome message
*/
```

==========

Spring JDBC(DAO) Module

==========

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

The purpose of JDBC is to provide APIs through which we can execute SQL statements against a database. When using JDBC, we have to manage database resources and handle database exceptions explicitly. To make JDBC easier to use, spring establishes a JDBC accessing framework by defining an abstraction layer on top of the existing JDBC APIs.

As the heart of the spring JDBC framework, jdbc templates are designed to provide template methods for different types of JDBC operations. Each template method is responsible for controlling the overall process and allows you to override particular tasks of the process.

Spring JDBC framework is built on top of the core J2SE JDBC API, and it is the lowest level data access technology that spring supports. Other, higher level technologies are Ibatis, hibernate, JDO, and other O/R Mapping tools.

Configuring a DataSource in spring:

JDBC2.0 introduced a new interface, javax.sql.DataSource, that acts as a factory for connections. This is the preferred method for obtaining a connection as it allows the use of connection pooling and avoids hard coding the connection properties. Spring JDBC framework is built with this in mind and a DataSource is the only thing spring uses to get a connection to the database.

The javax.sql.DataSource is a standard interface defined by the jdbc specification. There are many datasource implementations provided by different vendors and projects. It is very easy to switch between different data source implementations because they implement the common DataSource interface. Spring also provides several convenient but less powerful data source implementations. The simplest one is DriverManagerDataSource, which opens a new connection every time it’s requested.

DriverManagerDataSource is not an effiecient data source implementation, as it opens a new connection for the client every time it's requested. Another data source implementation provided by spring is SingleConnectionDataSource. As its name indicates, this maintains only a single connection that is reused all the time and never closed. Obviously, it is not suitable in a multithreaded environment.

Spring own data source implementations are mainly used for testing purposes. The most commonly used DataSources are org.apache.commons.dbcp.BasicDataSource and org.apache.commons.dbcp.PoolingDataSource.

Exception Translation:

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

In addition to JDBC exceptions, Spring can also wrap Hibernate-specific exceptions, converting them from proprietary, checked exceptions (in the case of versions of Hibernate prior to Hibernate 3.0), to a set of focused runtime exceptions (the same is true for JDO and JPA exceptions). This allows one to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without having annoying boilerplate catch-and-throw blocks and exception declarations in one's DAOs. (One can still trap and handle exceptions anywhere one needs to though.) As mentioned above, JDBC exceptions (including database-specific dialects) are also converted to the same hierarchy, meaning that one can perform some operations with JDBC within a consistent programming model.

JdbcTemplate class:

This is the central class in the JDBC core package**.** It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving application code to provide SQL and extract results. This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the org.springframework.dao package.

At the core of Spring JDBC support is the JdbcTemplate class. The JdbcTemplate allows you to work directly with instances of the Statement, PreparedStatement, and CallableStatement classes. The JdbcTemplate helps manage the connection and translates any SQLException to a Spring data access exception.

Using the JdbcTemplate, you can perform call, execute, query, and update operations on the database. Each of these operations has a number of variations, which differ in the number of arguments and the return types of the JdbcTemplate methods. The JdbcTemplate makes heavy use of callback interfaces. The implementations perform the required functions. The JdbcTemplate performs the common steps and delegates to the callbacks at appropriate points.

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.

JdbcTemplate class are threadsafe once configured. This is important because it means that you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs .

JdbcTemplate is stateful, in that it maintains a reference to a DataSource, but this state is not conversational state.

JdbcTemplate is created in the setter for the DataSource.

The org.springframework.jdbc.core.JdbcTemplate class is the key class for accessing databases through JDBC in Spring. It provides a complete API for executing SQL statements on the database at run time. The following kinds of SQL operations are supported by JdbcTemplate:Querying (SELECT operations).

- Updating (INSERT, UPDATE, and DELETE operations).
- Other SQL operations (all other SQL operations).

SpringJDBC

JdbcTest.java

```
//JdbcTest.java
import org.springframework.jdbc.core.*;
import java.util.*;
public class jdbctest
{
	JdbcTemplate jt;
	public void setJt(JdbcTemplate jt)
	{
		this.jt = jt;
	}
	public void loadAll()
	{
		List l = jt.queryForList("select * from employee");
		Iterator it = l.iterator();
		while(it.hasNext())
		{
			Object o = it.next();
			System.out.println(o.toString());
```

```
		}
	}
}


Client.java
import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
public class Client
{
	public static void main(String args[])
	{
		Resource res = new ClassPathResource("test.xml");
		BeanFactory factory = new XmlBeanFactory(res);
		jdbctest ob =(jdbctest)factory.getBean("id3");
		ob.loadAll();
	}

}

/*
Output:

	log4j:WARN No appenders could be found for logger
(org.springframework.beans.factory.xml.XmlBeanDefinitionReader).
	log4j:WARN Please initialize the log4j system properly.
	{id=1, name=Rohin, age=25}
	{id=2, name=Chinmai, age=10}
	{id=3, name=Renu, age=35}
	{id=200, name=Shree, age=30}
	{id=201, name=Kumar, age=35}
	{id=200, name=Shree, age=30}
	{id=201, name=Kumar, age=35}
	{id=400, name=Raam, age=33}
	{id=401, name=Sar, age=31}
	{id=400, name=Raam, age=33}
	{id=401, name=Sar, age=31}
*/

test.xml
<?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="id1" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/sun"/>
   <property name="username" value="root"/>
```

```
   <property name="password" value="root"/>
 </bean>
 <bean id="id2" class="org.springframework.jdbc.core.JdbcTemplate">
  <constructor-arg>
        <ref bean="id1"/>
  </constructor-arg>
 </bean>
 <bean id="id3" class="jdbctest">
  <property name="jt">
     <ref bean="id2"/>
  </property>
 </bean>
</beans>
```

springJDBCProperties

```
//jdbctest.java
import org.springframework.jdbc.core.*;
import java.util.*;
public class jdbctest
{
        JdbcTemplate jt;
        public void setJt(JdbcTemplate jt)
        {
                this.jt = jt;
        }
        public void loadAll()
        {
                List l = jt.queryForList("select * from employee");
                Iterator it = l.iterator();
                while(it.hasNext())
                {
                        Object o = it.next();
                        System.out.println(o.toString());
                }
        }
}



import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;
import org.springframework.beans.factory.xml.*;
public class client
{
```

```
	public static void main(String args[])
	{
		Resource res = new ClassPathResource("test.xml");
		ConfigurableListableBeanFactory factory = new XmlBeanFactory(res);
		PropertyPlaceholderConfigurer ppc=new PropertyPlaceholderConfigurer();
		  ppc.setLocation(new ClassPathResource("app.properties"));
		  ppc.postProcessBeanFactory(factory);
		jdbctest ob =(jdbctest)factory.getBean("id3");
		ob.loadAll();
	}

}

/*
Output:

	log4j:WARN No appenders could be found for logger
(org.springframework.beans.factory.xml.XmlBeanDefinitionReader).
	log4j:WARN Please initialize the log4j system properly.
	{id=1, name=Rohin, age=25}
	{id=2, name=Chinmai, age=10}
	{id=3, name=Renu, age=35}
	{id=200, name=Shree, age=30}
	{id=201, name=Kumar, age=35}
	{id=200, name=Shree, age=30}
	{id=201, name=Kumar, age=35}
	{id=400, name=Raam, age=33}
	{id=401, name=Sar, age=31}
	{id=400, name=Raam, age=33}
	{id=401, name=Sar, age=31}
*/

app.properties
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc\:mysql\://localhost\:3306/sun
jdbc.user=root
jdbc.pass=root


test.xml

<?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="id1"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName" value="${jdbc.driver}" />
		<property name="url" value="${jdbc.url}" />
```

```
		<property name="username" value="${jdbc.user}" />
		<property name="password" value="${jdbc.pass}" />
	</bean>
	<bean id="id2" class="org.springframework.jdbc.core.JdbcTemplate">
		<constructor-arg>
			<ref bean="id1" />
		</constructor-arg>
	</bean>
	<bean id="id3" class="jdbctest">
		<property name="jt">
			<ref bean="id2" />
		</property>
	</bean>
</beans>
```

SpringJdbc-Jto
Employee.java

```
//DTO
public class Employee {
	private int id;
	private String name;
	private int age;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	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;
	}
```

```
}


MyDao.java
public interface MyDao {
	void remove(int eno);

	void create(Employee ob);

	void list(int eno);
}

//MyDaoImpl.java
import org.springframework.jdbc.core.*;
import org.springframework.jdbc.support.rowset.*;
import java.util.*;

public class MyDaoImpl implements MyDao {
	JdbcTemplate jt;

	public MyDaoImpl(JdbcTemplate jt) {
		this.jt = jt;
	}

	public void remove(int eno) {
		int k = jt.update("delete from employee where id=?",
				new Object[] { new Integer(eno) });
		System.out.println(k + " row deleted");
	}

	public void create(Employee ob) {
		int x1 = ob.getId();
		String str1 = ob.getName();
		int a1 = ob.getAge();
		Object args[] = { new Integer(x1), str1, a1 };
		int k = jt.update("insert into employee values(?,?,?)", args);
		System.out.println(k + " row inserted");
	}

	public void list(int eno) {
		List l = jt.queryForList("select * from employee where id=?",
				new Object[] { new Integer(eno) });
		Iterator it = l.iterator();
		while (it.hasNext()) {
			Object o = it.next();
			System.out.println(o.toString());
		}
		// for rowset
		SqlRowSet srs = jt.queryForRowSet("select * from employee");
```

```
                srs.afterLast();
                while (srs.previous()) {
                        System.out.println(srs.getInt(1) + "  " + srs.getString(2) + "  "
                                        + srs.getInt(3));
                }
        }
}
```

Client.java

```
import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;

public class Client {
        public static void main(String[] args) {
                Resource res = new ClassPathResource("demo.xml");
                BeanFactory factory = new XmlBeanFactory(res);
                MyDao md = (MyDao) factory.getBean("id3");

                md.remove(201);

                Employee e = new Employee();
                e.setId(99);
                e.setName("Chandra");
                e.setAge(24);
                md.create(e);

                md.list(20);
        }
}
```

demo.xml

```
<?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="id1" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/sun"/>
   <property name="username" value="root"/>
   <property name="password" value="root"/>
 </bean>

 <bean id="id2" class="org.springframework.jdbc.core.JdbcTemplate">
  <constructor-arg ref="id1"/>
 </bean>

 <bean id="id3" class="MyDaoImpl">
```

```
 <constructor-arg ref="id2"/>
 </bean>
</beans>
```

springJdbc-usingProcedures

Procedure in MySQL

1)To set delimiter as "$"
delimiter $

2) To create Procedure "pro1"

```
create procedure pro1(in eno int, out experience int)
begin
 SELECT (DATEDIFF(curdate(),hiredate))/365 into experience from emp where
empno=eno;
 select experience;
end
$
```

3)
To set delimiter as ";"

```
delimiter ;
```

4)To call the procedure

```
call pro1(2,@a);
```

Mybean.java

```
import org.springframework.jdbc.object.*;
import org.springframework.jdbc.core.*;
import javax.sql.*;
import java.sql.*;
import java.util.*;

public class Mybean extends StoredProcedure {
        public Mybean(DataSource ds) {
                super(ds, "pro2");
                declareParameter(new SqlParameter("no", Types.INTEGER));
                declareParameter(new SqlOutParameter("exp", Types.INTEGER));
        }

        public void executePro(int empno) {
                Map imap = new HashMap();
                imap.put("no", new Integer(empno));
                Map omap = execute(imap);
```

```
		Object o2 = omap.get("exp");
		Integer i = (Integer) o2;
		System.out.println(i.intValue());
		System.out.println("----------");
	}

}
```

Client.java

```
import org.springframework.context.*;
import org.springframework.context.support.*;

public class Client {
	public static void main(String args[]) {
		ApplicationContext context = new ClassPathXmlApplicationContext(
				"test.xml");
		Object o = context.getBean("mb");
		Mybean ob = (Mybean) o;
		ob.executePro(3);
		ob.executePro(2);
	}
}
```

test.xml

```
<?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="dataSource"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName" value="com.mysql.jdbc.Driver" />
		<property name="url" value="jdbc:mysql://localhost:3306/sun" />
		<property name="username" value="root" />
		<property name="password" value="root" />
	</bean>

	<bean id="mb" class="Mybean">
		<constructor-arg ref="dataSource" />
	</bean>
</beans>
```

RowMapper interface:

org.springframework.jdbc.core.RowMapper is an interface used by JdbcTemplate for mapping rows of a ResultSet on a per-row basis. Implementations of this interface perform the actual work of mapping each row to a result object.

This callback interface is used to map every row in the ResultSet to an Object. Again, the

JdbcTemplate takes care of iterating through the ResultSet; the implementation should not call ResultSet.next. Typically, the Object instances returned from this method will be returned as members of a List from the JdbcTemplate method that takes the RowMapper as its argument.

This interface has a single abstract method called mapRow() and implementer of this interaced should take care about overriding or implementing this method.

SpringJdbc-RowMapper

Employee.java

```
//DTO
public class Employee {
	private int id;
	private String name;
	private int age;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	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;
	}

}
```

EmployeeMapper.java

```
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.*;

public class EmployeeMapper implements RowMapper {
```

```
        public Object mapRow(ResultSet rs, int index) throws SQLException {
                Employee e = new Employee();
                e.setId(rs.getInt(1));
                e.setName(rs.getString(2));
                e.setAge(rs.getInt(3));
                return e;
        }
}
```

TestBean.java

```
import org.springframework.jdbc.core.*;
import java.util.*;

public class TestBean {
        JdbcTemplate jt;

        public void setJt(JdbcTemplate jt) {
                this.jt = jt;
        }

        public void getData() {
                List l = jt.query("select * from employee", new EmployeeMapper());
                Iterator it = l.iterator();
                while (it.hasNext()) {
                        Employee ob = (Employee) it.next();
                        System.out.println(ob.getId() + "  " + ob.getName() + "  "
                                        + ob.getAge());
                }
        }
}
```

Client.java

```
import org.springframework.context.*;
import org.springframework.context.support.*;

public class Client {
        public static void main(String args[]) {
                ApplicationContext context = new ClassPathXmlApplicationContext(
                                "test.xml");
                Object o = context.getBean("id3");
                TestBean tb = (TestBean) o;
                tb.getData();
        }
}
```

test.xml

```
<?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="id1"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName" value="com.mysql.jdbc.Driver" />
		<property name="url" value="jdbc:mysql://localhost:3306/sun" />
		<property name="username" value="root" />
		<property name="password" value="root" />
	</bean>

	<bean id="id2" class="org.springframework.jdbc.core.JdbcTemplate">
		<constructor-arg>
			<ref bean="id1" />
		</constructor-arg>
	</bean>
	<bean id="id3" class="TestBean">
		<property name="jt">
			<ref bean="id2" />
		</property>
	</bean>
</beans>
```

==========

Spring ORM Module

==========

spring-Hibernate-ORM

EventPojo.java

```
public class EventPojo {
	Integer eventid;
	String eventname;

	public EventPojo() {
	}

	public Integer getEventid() {
		return eventid;
	}

	public void setEventid(Integer eventid) {
		this.eventid = eventid;
	}
```

```
	public String getEventname() {
		return eventname;
	}

	public void setEventname(String eventname) {
		this.eventname = eventname;
	}
}

import org.springframework.orm.hibernate3.*;
import java.util.*;

public class EventDao {
	HibernateTemplate ht;

	public void setHt(HibernateTemplate ht) {
		this.ht = ht;
	}

	public void saveObject(Object o) {
		ht.save(o);
	}

	public void selectAll() {
		List l = ht.loadAll(EventPojo.class);
		Iterator it = l.iterator();
		while (it.hasNext()) {
			EventPojo ob = (EventPojo) it.next();
			System.out.println(ob.getEventid());
			System.out.println(ob.getEventname());
		}
	}
}
```

eventpojo.hbm.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
	<class name="EventPojo" table="event">
		<id name="eventid" column="eventId">
			<generator class="assigned" />
		</id>
		<property name="eventname" column="eventName" />
	</class>
</hibernate-mapping>
```

hibernate.cf.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 2.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
        <session-factory>
                <!-- Database connection settings -->
                <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
                <property name="connection.url">jdbc:mysql://localhost:3306/sun</property>
                <property name="connection.username">root</property>
                <property name="connection.password">root</property>

                <!-- JDBC connection pool (use the built-in) -->
                <property name="connection.pool_size">1</property>

                <!-- SQL dialect -->
                <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

                <!-- Enable Hibernate's automatic session context management -->
                <property name="current_session_context_class">thread</property>

                <!-- Disable the second-level cache -->
                <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

                <!-- Echo all executed SQL to stdout -->
                <property name="show_sql">true</property>

                <!-- Drop and re-create the database schema on startup -->
                <property name="hbm2ddl.auto">update</property>

                <!-- Mapping files -->
                <mapping resource="eventpojo.hbm.xml" />

        </session-factory>
</hibernate-configuration>


demo.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

        <bean id="ds"
                class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
```

```
		<property name="url" value="jdbc:mysql://localhost:3306/sun" />
		<property name="username" value="root" />
		<property name="password" value="root" />
	</bean>

	<bean id="lsfb"
		class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

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

		<property name="mappingResources">
			<list>
				<value>eventpojo.hbm.xml</value>
			</list>
		</property>

		<property name="hibernateProperties">
			<props>
				<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
				<prop key="show_sql"> true </prop>
			</props>
		</property>

	</bean>

	<bean id="ht" class="org.springframework.orm.hibernate3.HibernateTemplate">
		<property name="sessionFactory">
			<ref bean="lsfb" />
		</property>
		<!-- <constructor-arg ref="lsfb"></constructor-arg> -->

	</bean>

	<bean id="daoImpl" class="EventDao">
		<property name="ht" ref="ht"></property>

	</bean>
</beans>
```

Client.java

```
import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;

public class Client {
	public static void main(String[] args) {
		/*
		 * BeanFactory factory=new XmlBeanFactory(new
```

```
		 * ClassPathResource("demo.xml"));
		 */

		Resource res = new ClassPathResource("demo.xml");
		BeanFactory factory = new XmlBeanFactory(res);
		EventDao ed = (EventDao) factory.getBean("daoImpl");
		EventPojo ep = new EventPojo();
		ep.setEventid(new Integer(2010));
		ep.setEventname("abcd");
		ed.saveObject(ep);
		ed.selectAll();
		System.out.println("completed...");
	}
}

/*
 * Output: 2010 abcd completed...
 */
```

Spring-Hibernate-ORM With annotations

Person.java

```
package com.venus;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Person implements Serializable {
	@Id
	private int id;
	private String name;
	private int age;

	public Person() {
	}

	public Person(int id, String name, int age) {
		this.id = id;
		this.name = name;
		this.age = age;
	}

	public int getId() {
		return id;
	}
```

```
	public void setId(int id) {
		this.id = id;
	}

	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;
	}
}
```

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 2.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
	<session-factory>
		<!-- Database connection settings -->
		<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
		<property name="connection.url">jdbc:mysql://localhost:3306/sun</property>
		<property name="connection.username">root</property>
		<property name="connection.password">root</property>

		<!-- JDBC connection pool (use the built-in) -->
		<property name="connection.pool_size">10</property>

		<!-- SQL dialect -->
		<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

		<!-- Enable Hibernate's automatic session context management -->
		<property name="current_session_context_class">thread</property>

		<!-- Disable the second-level cache -->
		<property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
```

```
		<!-- Echo all executed SQL to stdout -->
		<property name="show_sql">true</property>

		<!-- Drop and re-create the database schema on startup -->
		<property name="hbm2ddl.auto">update</property>

		<!-- Mapping files -->
		<mapping class="com.venus.Person" />

	</session-factory>
</hibernate-configuration>
```

application-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:jaxws="http://cxf.apache.org/jaxws"
	xmlns:soap="http://cxf.apache.org/bindings/soap"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

	<bean id="asf"
		class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
		<property name="configLocation">
			<value>hibernate.cfg.xml</value>
		</property>
	</bean>
	<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
		<property name="sessionFactory">
			<ref bean="asf" />
		</property>
	</bean>
</beans>
```

Client.java

```
package com.venus;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.orm.hibernate3.HibernateTemplate;
import com.venus.Person;

public class Client {
```

```
        public static void main(String... args) {
                ApplicationContext context = new ClassPathXmlApplicationContext(
                                "application-context.xml");
                HibernateTemplate ht = (HibernateTemplate) context
                                .getBean("hibernateTemplate");
                Person p = new Person(1, "rakesh", 25);
                ht.save(p);
        }
}
```

==========

Spring AOP Module

==========

Service layer comprises of two concern 1) core concern 2) cross cutting concern
concern mean the piece of code that perform the well defined task.

Core concern : Programmatically implementation of business rule is BL. the code that implement rule is nothing but core concern.

Ex: Interest Calculation logic.

Cross cutting concern: Code that cut the core concern & effect the entire service layer are known as cross cutting concern.

Ex: Transaction logic, security logic.

OOP in service layer development is used to built the primary concern(core concern).
We can use oops for the implementation of cross cutting concern. It is static approach of integrating cross cutting concern with core concern. But it is tightly coupled as far as concern is concern.

→ DI is meant for decoupling business object(core concern). using DI cross cutting cannot decoupled with core concern.

→ AOP is used to decouple dynamically core concern with crosscutting concerns.

The service layer of the spring application is built

→ Core concern is modularized using objects(oop). Core concerns are clearly separated using oop.

→ Cross cutting concern are modularized using aspects(AOP). cross concerns clearly separated and dynamically integrated with core concerns using AOP.

Business layer in spring = objects + aspects.

==========

AOP terminalogy:

1) **Aspects:** it is a functionality or a feature that cross-cuts over objects. The addition of the functutionality makes the code to Unit Test difficul because of its dependencies and the availability of the various components it is referring ex: logging & Tx mgt are the aspects.

```
public void businessOperation(BusinessData data){
        //logging
         logger.info("Business method called");

         //Transaction Management Begin
         transaction.begin();

        //Do the original business operation here
        transaction.end();
}
```

2) **Join Point:** Join Points defines the various execution points where an aspect can be applied. Ex: consider the following code:

```
public void someBusinessOperation(BusinessData data)
{
//method start → Possible aspect code here like logging.
        try{
        //original BL here.
        }
        catch(Exception e){
        //Exception → Aspect code here when some exception is raised.
        }
        finally {
        //finally → Even possible to have aspect code at this point too.
        }
//Method End → Aspect code here in the end of a method.
}
```

→ From the above code we have various points like start of the method, End of the method, the exception block, the finally block where a particular piece of Aspect can be made to execute. such possible execution points in the application code for embedding aspects are called Join Points.

→It is necessary that an Aspect should be applied to all the possible join points.

→It is a single location in the code where an advice should be executed(such as field access, method invocation, constructor invocation, etc).

→Spring's built-in AOP only supports method invocation currently.

→Spring provides only method call as join point.

3) **Point cut:** Join points refer to the logical points wherein a particular aspect or a set of aspects can be applied. A Pointcut or a Pointcut definition will exactly tell on which Join Points the aspects will be

applied.

Some aspects are as follows:

```
aspect LoggingAspect{}
aspect TransactionManagementAspect{}
```

```
public void someBusinessOperation(BusinessData data)
{
//method start → Possible aspect code here loke logging.
    try{
        //original BL here.
    }
    catch(Exception e){
      //Exception → Aspect code here when some exception is raised.
    }
    finally{
    //finally → Even possible to have aspect code at this point too.
    }
//Method End → Aspect code here in the end of a method.
}
```

→ from the above code, the possible execution points, i.e. Join Points, are the start of the method, end of the method, exception block and the finally block. There are the possible points wherein any of the aspects, Logging Aspect or Transaction management Aspect can be applied.

Consider the following code:

```
pointcut method_start_end_pointcut() {
    //This point cut applies the aspects, logging and tx, before the    //beginning and
the end of the method.
}

pointcut catch_and_finally_pointcut() {
    //This point cut applies the aspects, logging and tx, in the catch    //block(Whenever
an exception raises) and the finally block.
}
```

→ It is possible to define a Point Cut that binds the Aspect to a particular Joint Point or some Set of Join Points.

→ A Point cut is a set of many joint points where an advice shoukld be executed. So if,in Spring

Advice: Advice is the code that implements the Aspect. In general an Aspect defines the functionality in a more abstract manner. But, it is this Advice that provides a concrete code Implementation for the Aspect.

Target: It is business object that comprises of business core concern to which Aspect is applied.

Weaving: The mechanism of dynamically association of cross cutting concern to Core concern is called Weaving.

Proxy: An object produced through weaving is nothing but a proxy.

Weaver: Code that produces proxies is nothing but weaver.

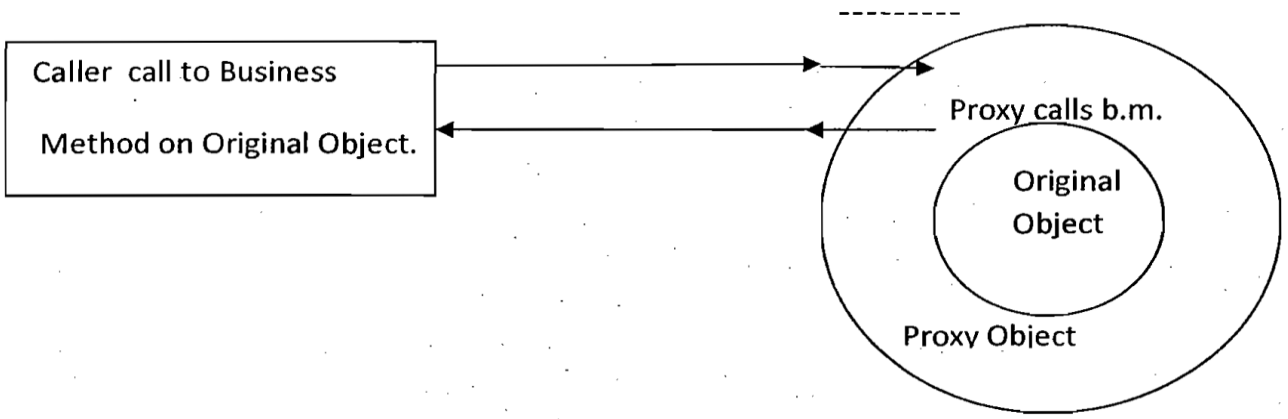
==========

AOP Working:

→ AOP works on the proxy designed pattern.

According to proxy design pattern the actual object(business object) is wrapped in to another object knows as proxy & substitute that object in place of actual object.

→ Proxy interrupt the call made by caller to original object. It will have chance to whether & when to pass on the call to original object & in mean time any additional code can be executed. Therefore proxy is best option for cross-cutting code.



There are two types of proxies →

1) **Static proxy** → Developing and maintaining proxy classes for each business method is static method.

2) **Dynamc proxy** → Without developing the proxy in advance as when required proxy is developed at runtime is known as dynamic proxy.

**Spring uses the dynamic proxy approach for AOP.

There are two ways to generate proxy generation.

1) **JDK approach:** If the business class is implementing any interface then JDK can create proxy for the business object.

2) **cglib approach:** If the business class is not implementing any interface then jdk cannot create prosy for the business object. so here we need help of cglib.

==========

→ **Spring AOP is framework**.

→ spring AOP is meant for crosscutting concern separation for spring bean registered in IOC container which is business component.

→ AOP & IOC goes hand in hand.

→ Spring uses dynamic proxy mechanism to provide AOP service spring beans.

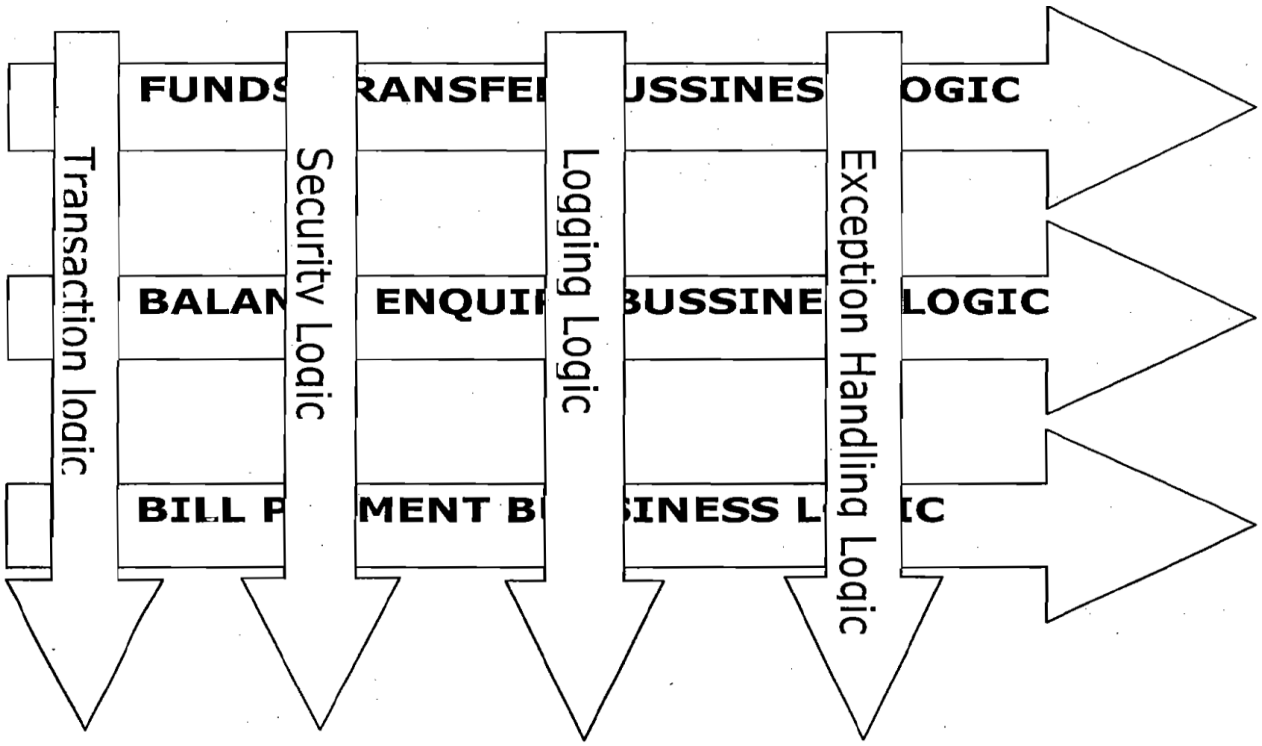
→ Spring framework use jdk, cglib for dynamic proxy creation.

→ spring 1.x AOP is known as classic spring AOP. It uses the spring interface to implement.

→ Spring 2.x AOP approach allows us to develop aspect as pojo.

Meta data are provided by 1) Annotation & aspect 2) xml element in wiring file.

Advice reffers to the actual implementation code for an Aspect. Spring support only Method Aspect. other one is Field Aspect.



==========

Different kinds of advices in spring AOP:

1) Before Advice 2) After Advice 3) Around 4) Throws advice

Configuration:

→ To configure these advices we have to depend on ProxyFactoryBean. This **Bean** is used to create **Proxy** objects for the implementation class along with the **Advice** implementatin

→ Note that the property "**proxyInterfaces**" contains the **Interface** name for which the proxy class has to be generated.

→ The "**interceptorNames**" property takes a list **Advices** to be applied to the dynamically generated proxy class.

→ Finally, the **implementation** class is given in the "**target**" property.

==========

1) **Before Advice :** It is used to intercept before the method execution starts.

→ In AOP, BeforeAdvice is represented in the form of **org.springframework.aop.MethodBeforeAdvice**.

Ex: A system should ,make security check on users before allowing them to accessing resources. In such a case, we can have a "Before Advice" that contains code which implements the User Authentication logic.

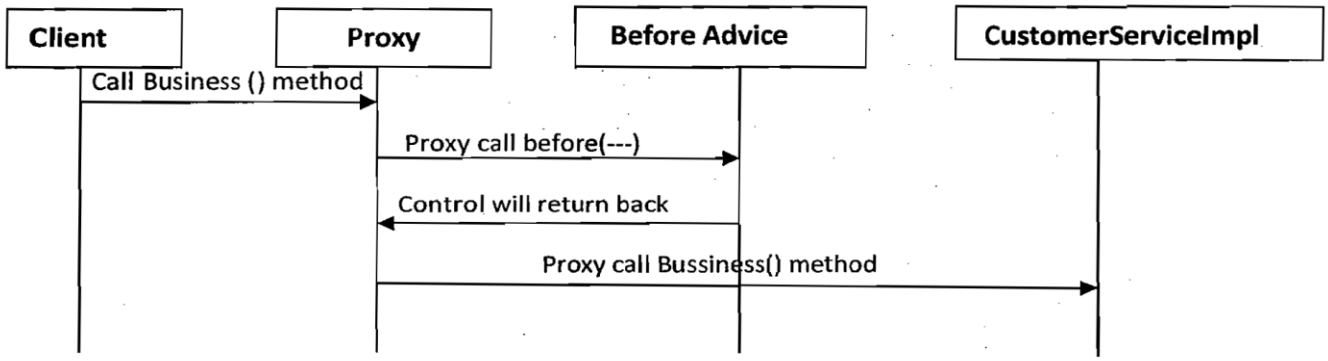
Steps required:

Once before advice is applied to a business bean whenever the caller the business method of the bean first of all before advice code is executed then business method is executed.

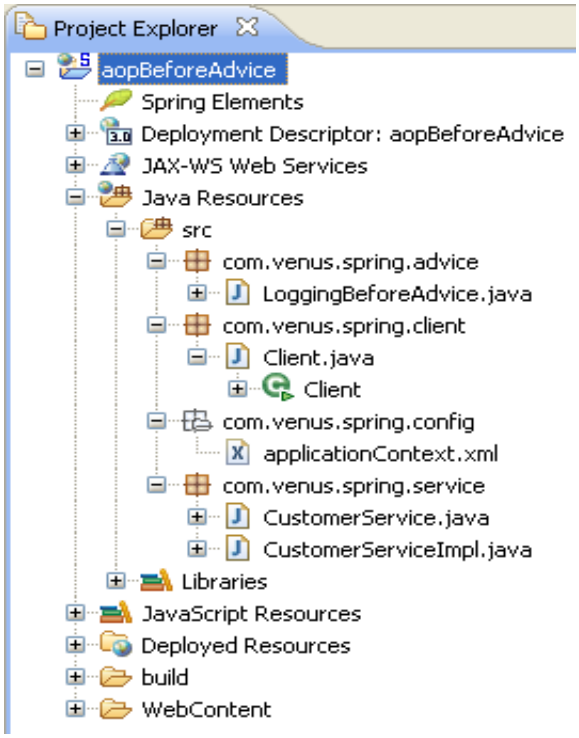
Step 1) Develop a business interface & its implementation class i.e. **Spring bean**.
Step 2) Develop a java class that implements **org.springframework.aop.MethodBeforeAdvice** interface and override **before()** method. In this implement the Crosscutting code.
Step 3) Get the proxy, and make a method call.

To apply BeforeAdvice we should implements MethodBeforeAdvice, thereby telling that **before(Method method, Object[] args, Object target)** method will be called before the execution of the method call. Note that the **java.lang.reflect.Method** method object represents target method to be invoked, Object[] args refers to the various args that are passed on to the method and target refers to the object which is calling the method.

Sequence Diagram of Before Advice



Example Program:



==========

CustomerService.java

```
package com.venus.spring.service;

public interface CustomerService {
        String printName();

        String printUrl();

        void printException();
```

```
}
```

==========

CustomerServiceImpl.java

```
package com.venus.spring.service;

public class CustomerServiceImpl implements CustomerService {
	private String name;
	private String url;

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

	public void setUrl(String url) {
		this.url = url;
	}

	@Override
	public String printName() {
		System.out.println("Business Method : printName()==>" + name);
		return name;
	}

	@Override
	public String printUrl() {
		System.out.println("Business Method: printUrl() ==>" + url);
		return url;
	}

	@Override
	public void printException() {
		throw new IllegalArgumentException("Custom Exception");
	}
}
```

==========

LoggingBeforeAdvice.java

```
package com.venus.spring.advice;

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

public class LoggingBeforeAdvice implements MethodBeforeAdvice {
	@Override
	public void before(Method method, Object[] args, Object target)
```

```
                        throws Throwable {
                    System.out.println("Before calling :: " + method.getName()
                                + " with arguments :: " + args.length + " on :: " + target);
            }
        }
```

==========

applicationContext.xml

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

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

        <bean id="lmba" class="com.venus.spring.advice.LoggingBeforeAdvice" />
        <bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
                <property name="name" value="Guru"></property>
                <property name="url" value="http://guru.edu"></property>
        </bean>

        <bean id="csProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <list>
                                <value> com.venus.spring.service.CustomerService</value>
                        </list>
                </property>
                <property name="interceptorNames">
                        <list>
                                <value>lmba</value>
                        </list>
                </property>

                <property name="target" ref="csImpl"></property>
        </bean>

</beans>
```

==========

Client.java

```
        package com.venus.spring.client;

        import org.springframework.context.ApplicationContext;
        import org.springframework.context.support.ClassPathXmlApplicationContext;
        import com.venus.spring.service.CustomerService;

        public class Client {
```

```
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/spring/config/applicationContext.xml");

	public static void main(String args[]) {
		CustomerService service = (CustomerService) context.getBean("csProxy");
		service.printName();
		System.out.println();
		service.printUrl();
		System.out.println();
		try {
			service.printException();
		} catch (Exception e) {
			System.out.println("Exception raised with message : "
					+ e.getMessage());
		}
	}
}

/*Output:

Dec 21, 2012 3:31:04 PM org.springframework.context.support.AbstractApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@179c285: startup date
[Fri Dec 21 15:31:04 IST 2012]; root of context hierarchy
Dec 21, 2012 3:31:04 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[com/venus/spring/config/applicationContext.xml]
Dec 21, 2012 3:31:04 PM
org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@bb7465: defining beans
[lmba,csImpl,csProxy]; root of factory hierarchy

Before calling :: printName with arguments :: 0 on ::
com.venus.spring.service.CustomerServiceImpl@1cbfe9d
Business Method : printName()==>Guru

Before calling :: printUrl with arguments :: 0 on ::
com.venus.spring.service.CustomerServiceImpl@1cbfe9d
Business Method: printUrl() ==>http://guru.edu

Before calling :: printException with arguments :: 0 on ::
com.venus.spring.service.CustomerServiceImpl@1cbfe9d
Exception raised with message : Custom Exception
 */
```

Required Jars

aopalliance.jar
commons-logging-1.0.4.jar
org.springframework.aop-3.0.1.RELEASE-A.jar
org.springframework.asm-3.0.1.RELEASE-A.jar
org.springframework.aspects-3.0.1.RELEASE-A.jar
org.springframework.beans-3.0.1.RELEASE-A.jar
org.springframework.context.support-3.0.1.RELEASE-A.jar
org.springframework.context-3.0.1.RELEASE-A.jar
org.springframework.core-3.0.1.RELEASE-A.jar
org.springframework.expression-3.0.1.RELEASE-A.jar
org.springframework.instrument.tomcat-3.0.1.RELEASE-A.jar
org.springframework.instrument-3.0.1.RELEASE-A.jar
org.springframework.jdbc-3.0.1.RELEASE-A.jar
org.springframework.jms-3.0.1.RELEASE-A.jar
org.springframework.orm-3.0.1.RELEASE-A.jar
org.springframework.oxm-3.0.1.RELEASE-A.jar
org.springframework.test-3.0.1.RELEASE-A.jar
org.springframework.transaction-3.0.1.RELEASE-A.jar
org.springframework.web.portlet-3.0.1.RELEASE-A.jar
org.springframework.web.servlet-3.0.1.RELEASE-A.jar
org.springframework.web.struts-3.0.1.RELEASE-A.jar
org.springframework.web-3.0.1.RELEASE-A.jar

==========

2) **After Advice:** It will be useful if some logic has to be executed before Returning the Control within a method execution. This advice is represented by the interface "org.springframework.aop.AfterReturningAdvice".

Ex: it is common in application to delete the session data and the various information pertaining to a user, after he has logged out from the application. These are ideal candidates for After Advice.

Steps to make use of After Advice in spring application:

→ Advice has to implement org.springframework.aop.AfterReturningAdvice.

→ afterReturning() method holds the cross cutting code.

→ Declaration of this method is

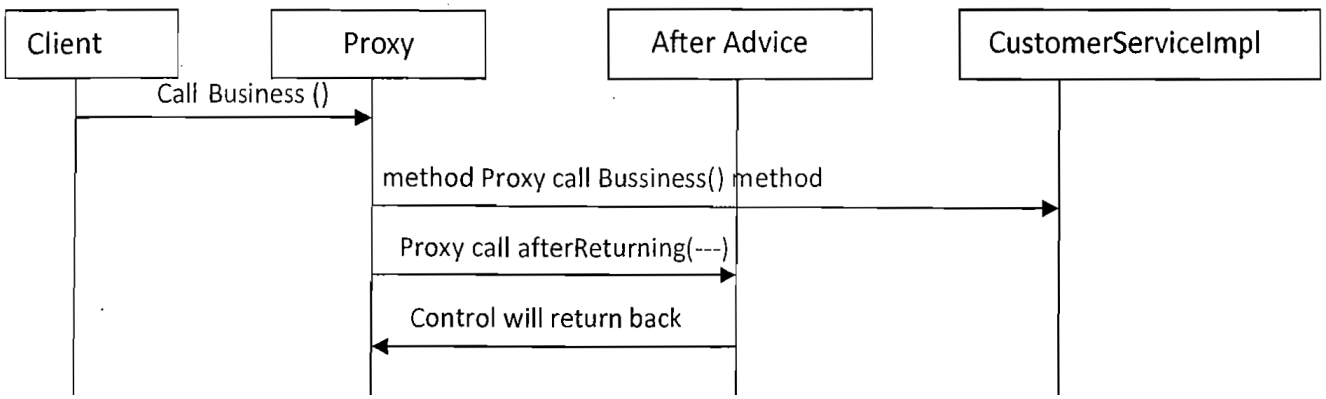
public void afterReturning(Object returningvalue, Method m, Object arg[], Object target);

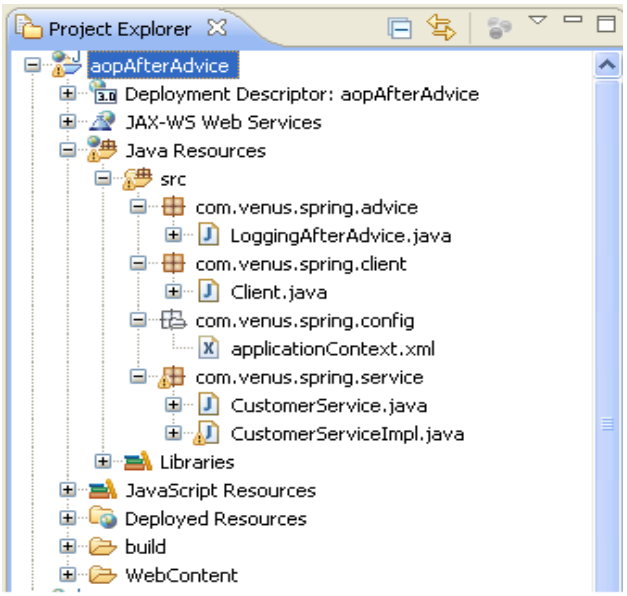
→ This advice is executed after the successful returning of Business method.

→ In this method returning value of business method is available but can't modified it.

Note that, afterReturning() method that will be called once the method returns normal execution. If some exception happens in the method execution then afterReturning() method will never be called.

Sequence Diagram of After Advice





CustomerService.java

```
package com.venus.spring.service;

public interface CustomerService {
        String printName(String name);

        String printUrl();
```

```
        void printException();
    }
```

CustomerServiceImpl.java

```
    package com.venus.spring.service;

    public class CustomerServiceImpl implements CustomerService {
        private String name;
        private String url;

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

        public void setUrl(String url) {
            this.url = url;
        }

        @Override
        public String printName(String name) {
            System.out.println("Business Method : printName()==>" + name);
            return name;
        }

        @Override
        public String printUrl() {
            System.out.println("Business Method: printUrl() ==>" + url);
            return url;
        }

        @Override
        public void printException() {
            throw new IllegalArgumentException("Custom Exception");
        }
    }
```

LoggingAfterAdvice.java

```
    package com.venus.spring.advice;

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

    public class LoggingAfterAdvice implements AfterReturningAdvice {
        @Override
        public void afterReturning(Object returnValue, Method method,
                Object[] args, Object target) throws Throwable {
```

```
                System.out.println("After calling :: " + method.getName() + " on "
                                + target.getClass() + " with arguments :: " + args.length
                                + "  giving return value :: " + returnValue);
                // After calling printName() on customerServiceImpl
                // with args 1, giving return value: null
        }
}
```

applicationContext.xml

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

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

        <bean id="lmba" class="com.venus.spring.advice.LoggingAfterAdvice" />
        <bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
                <!-- <property name="name" value="Guru"></property> -->
                <property name="url" value="http://guru.edu"></property>
        </bean>

        <bean id="csProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <list>
                                <value> com.venus.spring.service.CustomerService</value>
                        </list>
                </property>
                <property name="interceptorNames">
                        <list>
                                <value>lmba</value>
                        </list>
                </property>

                <property name="target" ref="csImpl"></property>
        </bean>

</beans>
```

Client.java

```
package com.venus.spring.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.venus.spring.service.CustomerService;
```

```
public class Client {
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/spring/config/applicationContext.xml");

	public static void main(String args[]) {
		CustomerService service = (CustomerService) context.getBean("csProxy");
		service.printName("Guru");
		System.out.println();
		service.printUrl();
		System.out.println();
		try {
			service.printException();
		} catch (Exception e) {
			System.out.println("Exception raised with message : "
					+ e.getMessage());
		}
	}
}

/*
 * preInstantiateSingletons INFO: Pre-instantiating singletons in
 * org.springframework.beans.factory.support.DefaultListableBeanFactory@bb7465:
 * defining beans [lmba,csImpl,csProxy]; root of factory hierarchy Before
 * calling :: printName with arguments :: 0 on ::
 * com.venus.spring.service.CustomerServiceImpl@1cbfe9d Business Method :
 * printName()==>Guru
 *
 * Before calling :: printUrl with arguments :: 0 on ::
 * com.venus.spring.service.CustomerServiceImpl@1cbfe9d Business Method:
 * printUrl() ==>http://guru.edu
 *
 * Before calling :: printException with arguments :: 0 on ::
 * com.venus.spring.service.CustomerServiceImpl@1cbfe9d Exception raised with
 * message : Custom Exception
 */
```

In our application, Client will calls business method, but actually calls the method on Proxy. Proxy will calls the After Advice method i.e., afterReturning() method then control will return back to Proxy and then Proxy will call the actual business method on CustomerServiceImpl.

Note: If business method raised Exception it is propagated to client according to the previous advice rules. Advice Exception can be raised which is propagated to client preventing value returning to client. afterReturning advice is used to verifies the process data of the business method.

After Returning advice is executed after the method invocation at the jointpoint has finished executing and has returned a value. Then after returning advice has access to the target of the method has already executed when after returning advice is invoked, it has no control over the method invocation at all.

==========

3) Around Advice: This Advice is very different from the other types of Advice that we have seen before, because of the fact that, this Advice provides finer control whether the target method has to be called or not. Considering the above advices, the return type of the method signature is always "void" meaning that, the Advice itself cannot change the return args of the method call. But, Around Advice can even change the return type, thereby returning a brand new object of other type if needed.

Implementing Around Advice:

→ It is most powerful advice which has the functionality of all three advice with additional capabilities.

→ Around advice method is executed before the business method and after business method. **Ex:** Transaction logic.

→ Around advice is used in scenario where the before business method execution some crosscutting code to be executed and agter the business method successfully or failure execution also crosscutting code has to implement.

"org.aopalliance.intercept.MethodInterceptor" interface.

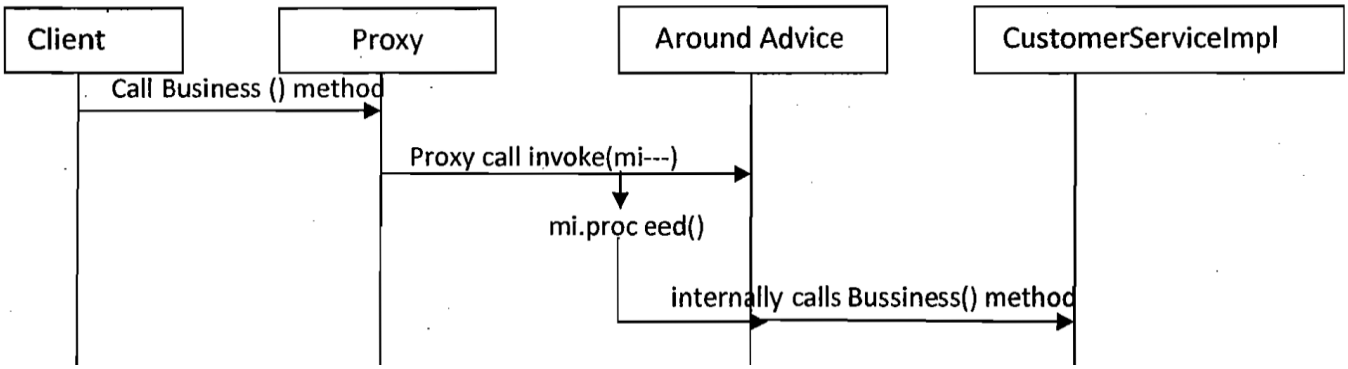
→ This interface has following abstract method has which advice class should implement

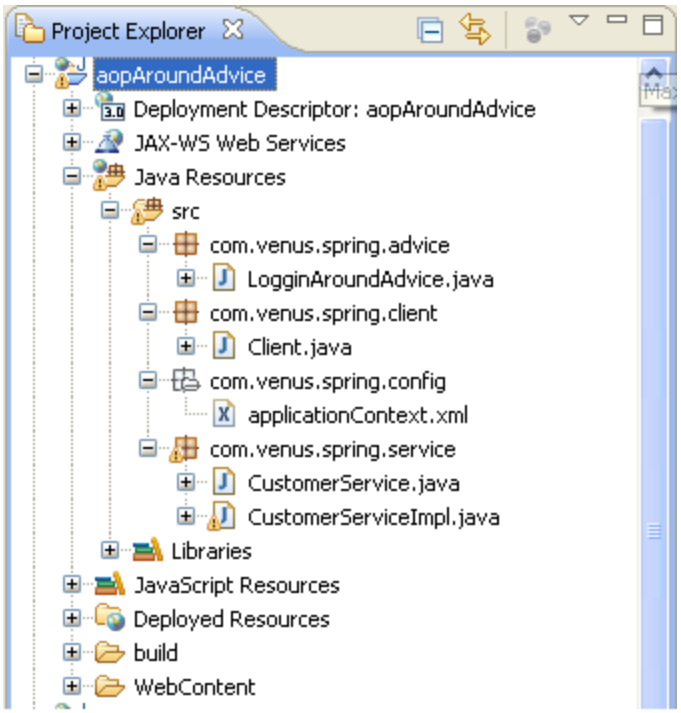
public Object invoke(method Invocation mi) throws throwable.

In this method the pre-processing & post processing crosscutting code is implemented.

Ex: System.out.println("Pre-processing crosscutting code");
Object o=mi.proceed();
System.out.println("post-processing crosscutting code");

Sequence Diagram of Around Advice






```
package com.venus.spring.service;

public interface CustomerService {
	void printName(String name);

	String printUrl();

	void printException();
}

package com.venus.spring.service;

public class CustomerServiceImpl implements CustomerService {
	private String name;
	private String url;



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

	public void setUrl(String url) {
```

```
		this.url = url;
	}

	@Override
	public void printName(String name) {
		System.out.println("Business Method : printName()==>" + name);
	}

	@Override
	public String printUrl() {
		System.out.println("Business Method: printUrl() ==>" + url);
		return url;
	}

	@Override
	public void printException() {
		throw new IllegalArgumentException("Custom Exception");
	}
}



package com.venus.spring.advice;

import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class LogginAroundAdvice implements MethodInterceptor {

	@Override
	public Object invoke(MethodInvocation invocation) throws Throwable {
		Method method = invocation.getMethod();
		Object[] args = invocation.getArguments();
		Object target = invocation.getThis();
		System.out.println(" Before calling :: " + method.getName()
				+ " with arguments :: " + args.length + " on :: "
				+ target.getClass());

		Object returnValue = null;

		try {
			returnValue = invocation.proceed();
		} catch (Exception e) {
			System.out.println(" Exception catched in advice : "
					+ e.getMessage());
		}

		System.out.println("After calling :: " + method.getName() + " on "
```

```
                        + target.getClass() + " with arguments :: " + args.length
                        + "  giving return value :: " + returnValue);

                return returnValue;

        }
}
```

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

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

        <bean id="laround" class="com.venus.spring.advice.LogginAroundAdvice" />
        <bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
                <!-- <property name="name" value="Guru"></property> -->
                <property name="url" value="http://guru.edu"></property>
        </bean>

        <bean id="csProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <list>
                                <value> com.venus.spring.service.CustomerService</value>
                        </list>
                </property>
                <property name="interceptorNames">
                        <list>
                                <value>laround</value>
                        </list>
                </property>

                <property name="target" ref="csImpl"></property>
        </bean>

</beans>
```

```
package com.venus.spring.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.venus.spring.service.CustomerService;

public class Client {
        private static ApplicationContext context = new ClassPathXmlApplicationContext(
```

```
                "com/venus/spring/config/applicationContext.xml");

        public static void main(String args[]) {
                CustomerService service = (CustomerService) context.getBean("csProxy");
                service.printName("Guru");
                System.out.println();
                service.printUrl();
                System.out.println();
                //service.printException();

        }
}
/*
Output:
 org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@e24e2a: startup date ;
root of context hierarchy
 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[com/venus/spring/config/applicationContext.xml]
 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
        INFO: Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@1125127: defining
beans [laround,csImpl,csProxy]; root of factory hierarchy
         Before calling :: printName with arguments :: 1 on :: class
com.venus.spring.service.CustomerServiceImpl
        Business Method : printName()==>Guru
        After calling :: printName on class com.venus.spring.service.CustomerServiceImpl with
arguments :: 1  giving return value :: null

         Before calling :: printUrl with arguments :: 0 on :: class
com.venus.spring.service.CustomerServiceImpl
        Business Method: printUrl() ==>http://guru.edu
        After calling :: printUrl on class com.venus.spring.service.CustomerServiceImpl with
arguments :: 0  giving return value :: http://guru.edu

*/
```

Here we will call the business method, but actually it call the method on proxy, then proxy will call invoke() method, invoke() method will call proceed() method then internally it will call business method. So it has the control over the business method invocation.

It is important to make a call to *MethodInvocation.proceed()* because if we didn't call the proceed() method then it never calls the business method.

Around Advice is allowed to execute before and after the method invocation, and we can control the point at which the method invocation is allowed to proceed. We can provide our own implementation logic according to our requirement.

The additional things in around advice :

→ Before advice can't prevent to business method() call from being invoked under normal condition. Around advice can do it by returning from invoke() with calling proceed().

→ After returning advice cannot changes the return value. Around Advice can.

→ Throws advice even handles the exception it cannot prevent the Exception begin propagated to client. Around Advice can propagated some value to client.
Instate of exception even though exception is raised.

==========

4) **Throws Advice:** When some kind of exception happens during the execution of a method, then to handle the exception properly, Throws Advice can be used through the means of "org.springframework.aop.ThrowsAdvice" interface. This interface is a marker interface meaning that it doesn't have any method within it. The method signature inside the Throws Advice can take any of the following form.

```
public void afterThrowing(Exception ex);
public void afterThrowing(Method method, Object[] args, Object target, Exception exception);
```

Ex: In a File copy program, if some kind of exception happens in the mid-way then the newly created target file has to be deleted as the partial content in the file doesn't carry any sensible meaning. It can be easily achieved through the means of Throws Advice.

Implementing Throws Advice:

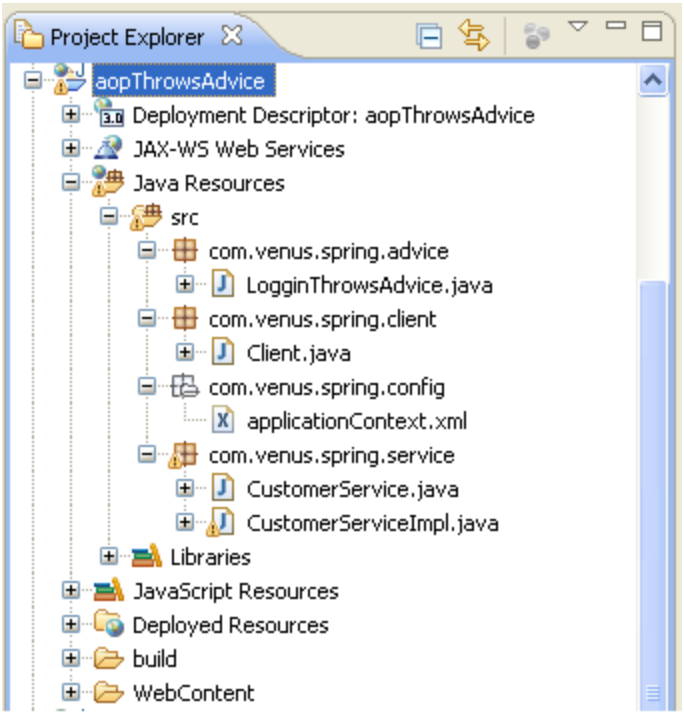
→ The advice is executed only when exception is raised business method of advice. Advice has to implement org.springframweork.aop.throwsAdvice.

→ It is just marker interface, it don't have any method.

→ It is also called null interface.

→ Exception handling code is considered as crosscutting code & can be done for entire service layer in Throwsadvice with different kinds of exception parameter.

Note: Here we have to override the method which take four args.
public void afterThrowing(Method method, Object[] args, Object target, Throwable t);




```
package com.venus.spring.service;

public interface CustomerService {
	String printName(String name);

	String printUrl();

	void printException();
}
```

```
package com.venus.spring.service;

public class CustomerServiceImpl implements CustomerService {
	private String name;
	private String url;

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

	public void setUrl(String url) {
		this.url = url;
	}
```

```
	@Override
	public String printName(String name) {
		System.out.println("Business Method : printName()==>" + name);
		return name;
	}

	@Override
	public String printUrl() {
		System.out.println("Business Method: printUrl() ==>" + url);
		return url;
	}

	@Override
	public void printException() {
		throw new IllegalArgumentException("Custom Exception");
	}
}
```

```
package com.venus.spring.advice;

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

public class LogginThrowsAdvice implements ThrowsAdvice {
	// public void afterThrowing(Exception ex)

	public void afterThrowing(Method method, Object[] args, Object target,
			Exception exception) {
		System.out.println("After throwing Exception by method :: "
				+ method.getName() + " with arguments :: " + args.length
				+ " on " + target.getClass() + "  with exception message :: "
				+ exception.getMessage());
	}
}
```

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

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

	<bean id="lta" class="com.venus.spring.advice.LogginThrowsAdvice" />
	<bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
		<!-- <property name="name" value="Guru"></property> -->
```

```
		<property name="url" value="http://guru.edu"></property>
	</bean>

	<bean id="csProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="proxyInterfaces">
			<list>
				<value> com.venus.spring.service.CustomerService</value>
			</list>
		</property>
		<property name="interceptorNames">
			<list>
				<value>lta</value>
			</list>
		</property>

		<property name="target" ref="csImpl"></property>
	</bean>

</beans>
```

```
package com.venus.spring.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.venus.spring.service.CustomerService;

public class Client {
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/spring/config/applicationContext.xml");

	public static void main(String args[]) {
		CustomerService service = (CustomerService) context.getBean("csProxy");
		service.printName("Guru");
		System.out.println();
		service.printUrl();
		System.out.println();
		try {
			service.printException();
		} catch (Exception e) {
			System.out.println("Exception raised with message : "
					+ e.getMessage());
		}
	}
}

/*
 * preInstantiateSingletons INFO: Pre-instantiating singletons in
 * org.springframework.beans.factory.support.DefaultListableBeanFactory@18dfef8:
```

```
 * defining beans [lta,csImpl,csProxy]; root of factory hierarchy Business
 * Method : printName()==>Guru
 *
 * Business Method: printUrl() ==>http://guru.edu
 *
 * After throwing Exception by method :: printException with arguments :: 0 on
 * class com.venus.spring.service.CustomerServiceImpl with exception message ::
 * Custom Exception Exception raised with message : Custom Exception
 */
```

Throws Advice is executed after a method invocation returns but only if that invocation threw an exception. It is possible for throws advice to catch only specific exceptions, and if we choose to do so, we can access the method that threw the exception, the args passed into the invocation, and the target of the invocation.

==========

To deal with proxy beans:

--> Per bean configure the ProxyfactoryBean with different ID on configuration file. While calling the getBean() specify that id to get proxy for corresponding bean.

--> We have to follow naming convention to give an ID to the proxy because we configure multiple proxy objects. So to identify particular proxy we have to follow the naming convention.

To choose an Advice type:

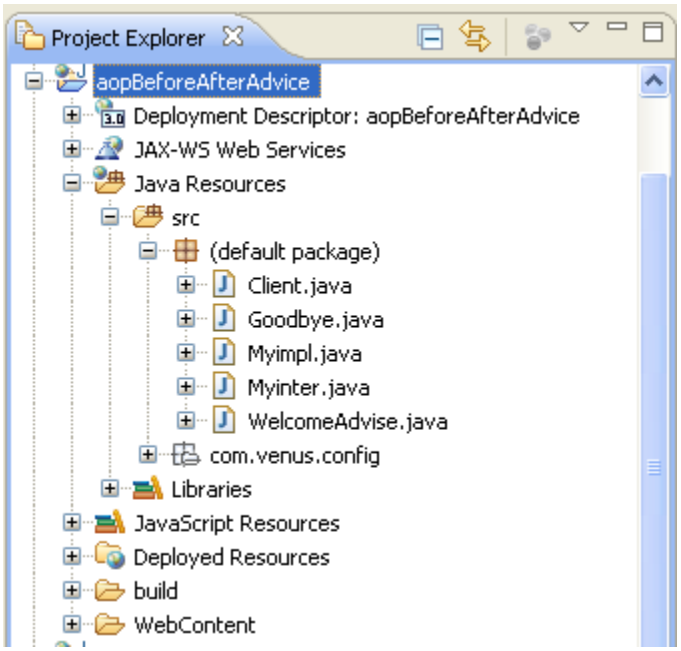
1) Before Advice: → Security logic

2) After Advice: → logging verification process data

3) Around Advice: → Transaction management (performance comprises)

4) Throws Advice: → Centralization exception

AopBeforeAfterAdvice Program:




```
public interface Myinter {
	void method1();

	void xyz();
}

public class Myimpl implements Myinter {
	public void method1() {
		System.out.println("i am method1() B.L");

	}

	public void xyz() {
		System.out.println("i am  xyz()");
	}
}

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

public class Goodbye implements AfterReturningAdvice {
	public void afterReturning(Object rvalue, Method m, Object[] args,
			Object target) throws Throwable {
		System.out.println("good bye");
	}
}
```

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

public class WelcomeAdvise implements MethodBeforeAdvice {
	public void before(Method m, Object[] args, Object target) throws Throwable {
		System.out.println("welcome to  " + m.getName());
	}
}
```

```
<?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="id1" class="Myimpl" />
	<bean id="id2" class="WelcomeAdvise" />
	<bean id="id3" class="Goodbye" />
	<bean id="id4" class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="proxyInterfaces">
			<value>Myinter</value>
		</property>
		<property name="interceptorNames">
			<list>
				<value>id2</value>
				<value>id3</value>
			</list>
		</property>
		<property name="target">
			<ref bean="id1" />
		</property>
	</bean>
</beans>
```

```
import org.springframework.context.*;
import org.springframework.context.support.*;
public class Client
{
	public static void main(String[] args)
	{
		//ApplicationContext ctx=new
FileSystemXmlApplicationContext("src/com/venus/config/spconfig.xml");

		ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/venus/config/spconfig.xml");

		//Myinter mi=(Myinter)ctx.getBean("id4");
```

```
                Myinter mi=(Myinter)context.getBean("id4");

                mi.method1();
                mi.xyz();
        }
}

/*
 Output:

org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@e24e2a: startup date;
root of context hierarchy
 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [com/venus/config/spconfig.xml]
 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@f84386: defining beans
[id1,id2,id3,id4]; root of factory hierarchy
welcome to  method1
i am method1() B.L
good bye
welcome to  xyz
i am  xyz()
good bye

*/
```

==========

Pointcut in AOP

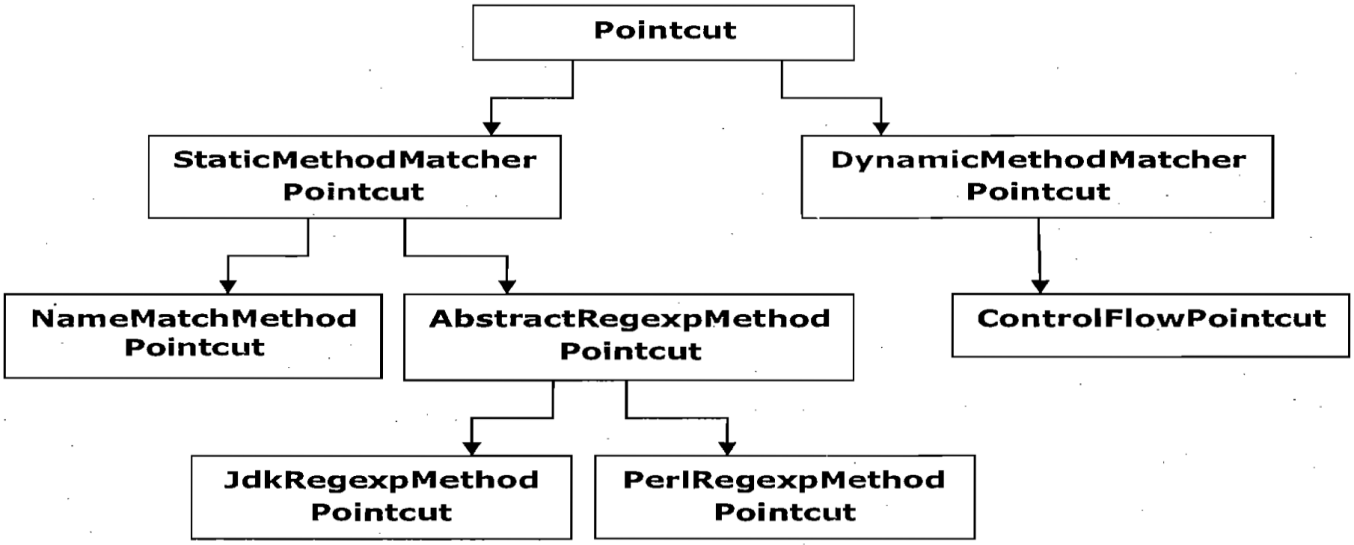
==========

Point Cuts define where exactly the Advices have to be applied in various Join Points. Generally they act as Filters for the application of various Advices into the real implementation.

Spring defines two types of PointCuts namely the **Static** and the **Dynamic** PointCuts.

--> Static point cut verifies whether that join point has to be adviced or not, only once the result is catched & reused.

--> Dynamic point cut verifies every time as it has to decide the join point based on the arg passed to method call.



PointCut is an interface, represented by "org.springframework.aop.Pointcut"

```
public interface Pointcut {
        ClassFilter getClassFilter();

        MathodMatcher getMethodMatcher();
}
```

The getClassFilter() method returns a ClassFilter object which determines whether the classObject arg passed to the matches() method should be considered for giving Advices.

The typical implementation of the ClassFilter interface is as follows:

```
public class MyClassFilter implements ClassFilter {
        public boolean matches(Class classObject) {
        //check whether the class objects should be adviced based on their name.
                        if (shouldBeAdviced(className){
                                return true;
                        }
                        return false;
        }
}
```

The next interface in consideration is the MethodMatcher which will filter whether various methods within the class should be given Advices. Ex:

MyMethodMatcher.java

```
public class MyMethodMatcher implements MethodMatcher {
        public boolean matches(Method m, class targetClass){
```

```
                String methodName=m.getName();
                if(methodName.startsWith("get")){
                        return true;
                }
                return false;
        }
        public boolean isRuntime() {
                return false;
        }
        public booleanMatches(Method m, Class target, Object[] args) {
        //This method wont be called in our case. so, just return false.
                return false;
        }
}
```

In the above code we have 3 methods defined inside in MethodMatcher interface. The isRuntime() method should return true when we want to go for Dynamic Point cut Inclusion by depending on the values of the args, which usually happens at run-time.

The implementation of the 2 args matches() method essentially says that we want only the getter methods to be advised.

==========

StaticMethodMatherPointcut: It has two types of Pointcuts:

→ NameMatchMethodPointcut

→ RegularExpressionPointcut

Pointcut allows us to intercept a method by it's method name. Inaddition, a "Pointcut" must be associated with an "Advisor".

In Spring AOP, 3 very common technical terms are→ **Advice**, **Pointcut**, **Advisor**

Advice → Indicate the action to take either before or after the method execution.

Pointcut → Indicate which method should be intercept, by method name or regular expression name.

Advisor → Group "Advice" and "Pointcut" into a single unit, and pass it to a proxy factory object.

==========

NameMatchPointCut

We can intercept a method via "**pointcut**" and "**advisor**". Create a **NameMatchMethodPointcut** pointcut bean, and put the method name we want to intercept in the "mappedName" property value.

→ org.springframework.aop.support.NameMatchMethodPointcut has to configure in bean configuration file. It has java.util.List type variable "mapperNames". Through setter injection we need to specify the method names of the target object which we want them to advice.

→ This pointcut is bean uses their names to decide the selection of join point.

→ This pointcut should inform to proxy.

→ We cannot inform only point cut to ProxyFactoryBean. We need to use built in advisor for this purpose.

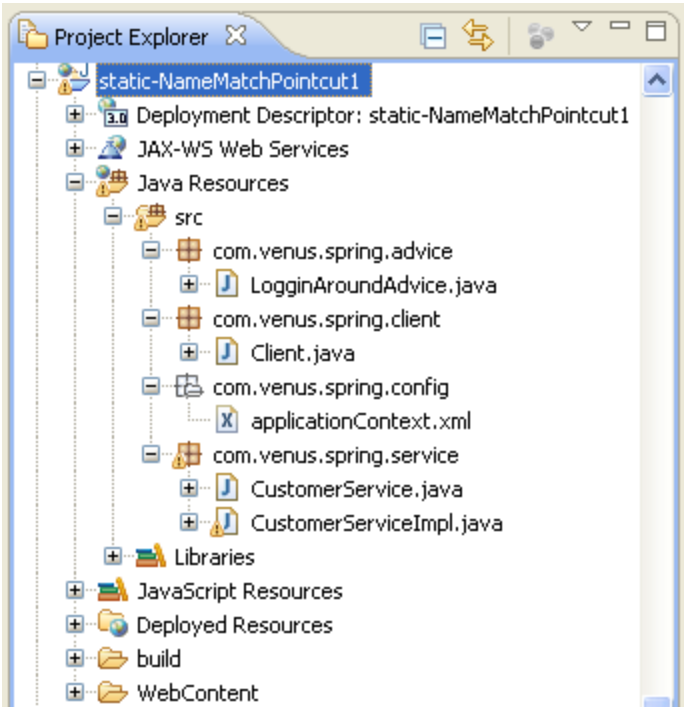
→ While configuration the ProxyFactoryBean we need to specify the advisor instead of advice.

We can use NameMatchMethodPointcutAdvisor to combine both pointcut and advisor into a single bean.

Here the name of the methods that are too be given advices can be directly mentioned in the configuration file. "*" represents that all the methods in the class should be given Advice.

The Expression print* tells that all method names starting with the method name print will be given Advices. If we want all the methods in our Class to be advised, then the "value" tag should be given "*" meaning all the methods in the Class.

Program:




```
package com.venus.spring.service;
```

```
public interface CustomerService {
	String printName(String name);

	String printUrl();

	void printException();
}

package com.venus.spring.service;

public class CustomerServiceImpl implements CustomerService {
	private String name;
	private String url;

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

	public void setUrl(String url) {
		this.url = url;
	}

	@Override
	public String printName(String name) {
		System.out.println("Business Method : printName()==>" + name);
		return name;
	}

	@Override
	public String printUrl() {
		System.out.println("Business Method: printUrl() ==>" + url);
		return url;
	}

	@Override
	public void printException() {
		throw new IllegalArgumentException("Custom Exception");
	}
}


package com.venus.spring.advice;

import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class LogginAroundAdvice implements MethodInterceptor {
```

```
	@Override
	public Object invoke(MethodInvocation invocation) throws Throwable {
		Method method = invocation.getMethod();
		Object[] args = invocation.getArguments();
		Object target = invocation.getThis();
		System.out.println(" Before calling :: " + method.getName()
				+ " with arguments :: " + args.length + " on :: "
				+ target.getClass());

		Object returnValue = null;

		try {
			returnValue = invocation.proceed();
		} catch (Exception e) {
			System.out.println(" Exception catched in advice : "
					+ e.getMessage());
		}

		System.out.println("After calling :: " + method.getName() + " on "
				+ target.getClass() + " with arguments :: " + args.length
				+ "  giving return value :: " + returnValue);

		return returnValue;

	}
}
```

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

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

	<bean id="laround" class="com.venus.spring.advice.LogginAroundAdvice" />
	<bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
		<!-- <property name="name" value="Guru"></property> -->
		<property name="url" value="http://guru.edu"></property>
	</bean>

	<bean id="laroundNameUrlAdvisor"
		class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
		<property name="advice" ref="laround"></property>
		<property name="mappedNames">
			<list>
				<value>printName</value>
				<value>printUrl</value>
```

```
                        </list>

                </property>
        </bean>

        <bean id="csProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <list>
                                <value> com.venus.spring.service.CustomerService</value>
                        </list>
                </property>
                <property name="interceptorNames">
                        <list>
                                <value>laroundNameUrlAdvisor</value>
                        </list>
                </property>

                <property name="target" ref="csImpl"></property>
        </bean>

</beans>
```

```
package com.venus.spring.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.venus.spring.service.CustomerService;

public class Client {
        private static ApplicationContext context = new ClassPathXmlApplicationContext(
                        "com/venus/spring/config/applicationContext.xml");

        public static void main(String args[]) {
                CustomerService service = (CustomerService) context.getBean("csProxy");
                service.printName("Guru");
                System.out.println();
                service.printUrl();
                System.out.println();
                try {
                        service.printException();
                } catch (Exception e) {
                        System.out.println("Exception raised with message : "
                                        + e.getMessage());
                }
        }
}

/*
Before calling :: printName with arguments :: 1 on :: class
```

```
com.venus.spring.service.CustomerServiceImpl
Business Method : printName()==>Guru
After calling :: printName on class com.venus.spring.service.CustomerServiceImpl with
arguments :: 1  giving return value :: Guru

Before calling :: printUrl with arguments :: 0 on :: class
com.venus.spring.service.CustomerServiceImpl
Business Method: printUrl() ==>http://guru.edu
After calling :: printUrl on class com.venus.spring.service.CustomerServiceImpl with arguments
:: 0  giving return value :: http://guru.edu

Exception raised with message : Custom Exception
*/
```

==========

RegularExpression Pointcut:

Beside the matching method by name, we can also match the method's name by using regular expression pointcut.

This pointcut is used to verify join point based on pattern of method name instead of name.

This kind is used if we want to match the name of the methods in the Class based on Regular Expression.

We have two types of regular expressions:

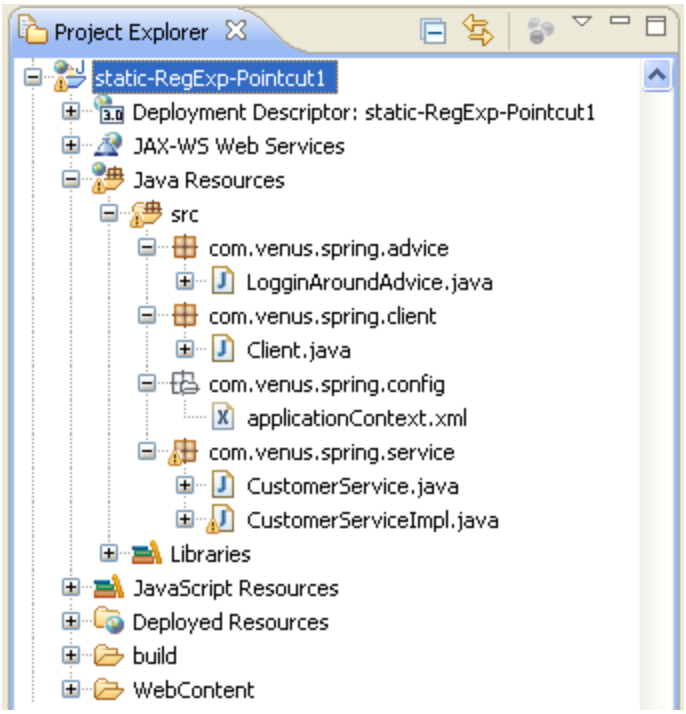
→ **PerlRegularExpression** → represented by
org.springframework.aop.support.Perl5RegexpMethodPointcut

→ **JdkRegularExpression** → represented by
org.springframework.aop.support.JdkRegexpMethodPointcut

Here we are using JdkRegularExpression.

The Expression *.Exception*. tells that method names ending with the method name Exception will be given Advices.

Suppose we wish that only the methods printUrl() and printException() should be given Advice by some aspect. The following is the example program:




```
package com.venus.spring.service;

public interface CustomerService {
	String printName(String name);

	String printUrl();

	void printException();
}


package com.venus.spring.service;

public class CustomerServiceImpl implements CustomerService {
	private String name;
	private String url;

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

	public void setUrl(String url) {
		this.url = url;
	}
```

```
	@Override
	public String printName(String name) {
		System.out.println("Business Method : printName()==>" + name);
		return name;
	}

	@Override
	public String printUrl() {
		System.out.println("Business Method: printUrl() ==>" + url);
		return url;
	}

	@Override
	public void printException() {
		throw new IllegalArgumentException("Custom Exception");
	}
}


package com.venus.spring.advice;

import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class LogginAroundAdvice implements MethodInterceptor {

	@Override
	public Object invoke(MethodInvocation invocation) throws Throwable {
		Method method = invocation.getMethod();
		Object[] args = invocation.getArguments();
		Object target = invocation.getThis();
		System.out.println(" Before calling :: " + method.getName()
				+ " with arguments :: " + args.length + " on :: "
				+ target.getClass());

		Object returnValue = null;

		try {
			returnValue = invocation.proceed();
		} catch (Exception e) {
			System.out.println(" Exception catched in advice : "
					+ e.getMessage());
		}

		System.out.println("After calling :: " + method.getName() + " on "
				+ target.getClass() + " with arguments :: " + args.length
				+ "  giving return value :: " + returnValue);
```

```
                return returnValue;

        }
}
```

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

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

        <bean id="laround" class="com.venus.spring.advice.LogginAroundAdvice" />
        <bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
                <!-- <property name="name" value="Guru"></property> -->
                <property name="url" value="http://guru.edu"></property>
        </bean>

        <bean id="laroundExceptionUrlAdvisor"
                class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
                <property name="advice" ref="laround"></property>
                <property name="patterns">
                        <list>
                                <value>.*Exception.*</value>
                                <value>.*Url.*</value>
                        </list>
                </property>
        </bean>

        <bean id="csProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <list>
                                <value> com.venus.spring.service.CustomerService</value>
                        </list>
                </property>
                <property name="interceptorNames">
                        <list>
                                <value>laroundExceptionUrlAdvisor</value>
                        </list>
                </property>

                <property name="target" ref="csImpl"></property>
        </bean>

</beans>
```

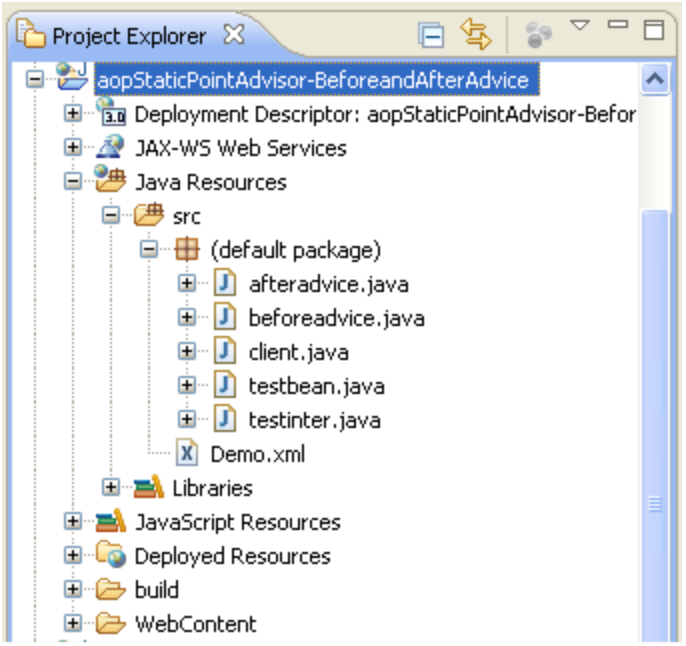
```
package com.venus.spring.client;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.venus.spring.service.CustomerService;

public class Client {
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/spring/config/applicationContext.xml");

	public static void main(String args[]) {
		CustomerService service = (CustomerService) context.getBean("csProxy");
		service.printName("Guru");
		System.out.println();
		service.printUrl();
		System.out.println();
		try {
			service.printException();
		} catch (Exception e) {
			System.out.println("Exception raised with message : "
					+ e.getMessage());
		}
	}
}

/*
 * INFO: Pre-instantiating singletons in
 * org.springframework.beans.factory.support.DefaultListableBeanFactory@18dfef8:
 * defining beans [laround,csImpl,laroundExceptionUrlAdvisor,csProxy]; root of
 * factory hierarchy Business Method : printName()==>Guru
 *
 * Before calling :: printUrl with arguments :: 0 on :: class
 * com.venus.spring.service.CustomerServiceImpl Business Method: printUrl()
 * ==>http://guru.edu After calling :: printUrl on class
 * com.venus.spring.service.CustomerServiceImpl with arguments :: 0 giving
 * return value :: http://guru.edu
 *
 * Before calling :: printException with arguments :: 0 on :: class
 * com.venus.spring.service.CustomerServiceImpl Exception catched in advice :
 * Custom Exception After calling :: printException on class
 * com.venus.spring.service.CustomerServiceImpl with arguments :: 0 giving
 * return value :: null
 */
```

It will only match the method which has "Url" within the method name. It's also quite useful for the DAO transaction management, where we can declare ".*DAO.* to include all our DAO classes.

==========

AOP StaticPointAdvisor-BeforeandAfterAdvice Program:




```
public interface testinter {
	void sayHello();

	void sayBye();

	void getData();

	void getMyData();
}


public class testbean implements testinter {
	public void getData() {
		System.out.println("it is getData()");

	}

	public void getMyData() {
		System.out.println("it is getMyData()");
	}

	public void sayHello() {
		System.out.println("i am sayHello()");
	}
```

```
	public void sayBye() {
		System.out.println("i am sayBye()");
	}
}

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class afteradvice implements AfterReturningAdvice {
	public void afterReturning(Object rvalue, Method m, Object[] args,
			Object target) throws Throwable {
		System.out.println("i am after advice to " + m.getName() + " method");
	}
}

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class beforeadvice implements MethodBeforeAdvice {
	public void before(Method m, Object[] args, Object o) throws Throwable {
		System.out.println("i am before advice to " + m.getName() + " method");
	}
}

<?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="id1" class="testbean" />
	<bean id="id2" class="beforeadvice" />
	<bean id="id3" class="afteradvice" />

	<bean id="advisor1"
		class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
		<property name="mappedName">
			<value>say*</value>
		</property>
		<property name="advice">
			<ref bean="id2" />
		</property>
	</bean>

	<bean id="advisor2"
		class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
		<property name="pattern">
			<value>.*get.+Data</value>
		</property>
```

```
		<property name="advice">
			<ref bean="id3" />
		</property>
	</bean>

	<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="proxyInterfaces">
			<value>testinter</value>
		</property>
		<property name="interceptorNames">
			<list>
				<value>advisor1</value>
				<value>advisor2</value>
			</list>
		</property>
		<property name="target">
			<ref bean="id1" />
		</property>
	</bean>
</beans>
```

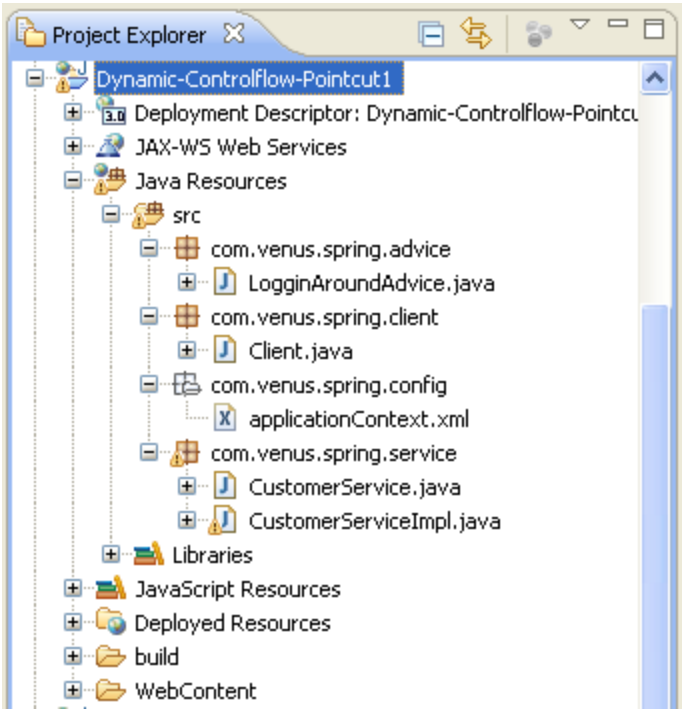
```
import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;

public class client {
	public static void main(String args[]) {
		Resource res = new ClassPathResource("Demo.xml");
		BeanFactory factory = new XmlBeanFactory(res);
		testinter tb = (testinter) factory.getBean("proxy");
		tb.getData();
		System.out.println("----------");
		tb.sayHello();
		System.out.println("----------");
		tb.getMyData();
	}
}

/*
 * Output:
 *
 * Apr 8, 2013 3:18:32 PM
 * org.springframework.beans.factory.xml.XmlBeanDefinitionReader
 * loadBeanDefinitions INFO: Loading XML bean definitions from class path
 * resource [Demo.xml] it is getData() ---------- i am before advice to sayHello
 * method i am sayHello() ---------- it is getMyData() i am after advice to
 * getMyData method
```

```
*/
```

==========

Dynamic ControlFlow Pointcut

This point cur is used to verify from which context business method call is made. If method call is made in specified flow it will be advice otherwise it won't.




```
package com.venus.spring.service;

public interface CustomerService {
	String printName(String name);

	String printUrl();

	void printException();
}


package com.venus.spring.service;
```

```
public class CustomerServiceImpl implements CustomerService {
	private String name;
	private String url;

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

	public void setUrl(String url) {
		this.url = url;
	}

	@Override
	public String printName(String name) {
		System.out.println("Business Method : printName()==>" + name);
		return name;
	}

	@Override
	public String printUrl() {
		System.out.println("Business Method: printUrl() ==>" + url);
		return url;
	}

	@Override
	public void printException() {
		throw new IllegalArgumentException("Custom Exception");
	}
}


package com.venus.spring.advice;

import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class LogginAroundAdvice implements MethodInterceptor {

	@Override
	public Object invoke(MethodInvocation invocation) throws Throwable {
		Method method = invocation.getMethod();
		Object[] args = invocation.getArguments();
		Object target = invocation.getThis();
		System.out.println(" Before calling :: " + method.getName()
				+ " with arguments :: " + args.length + " on :: "
				+ target.getClass());
```

```
		Object returnValue = null;

		try {
			returnValue = invocation.proceed();
		} catch (Exception e) {
			System.out.println(" Exception catched in advice : "
					+ e.getMessage());
		}

		System.out.println("After calling :: " + method.getName() + " on "
				+ target.getClass() + " with arguments :: " + args.length
				+ "  giving return value :: " + returnValue);

		return returnValue;

	}
}
```

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

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

	<bean id="laround" class="com.venus.spring.advice.LogginAroundAdvice" />
	<bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
		<!-- <property name="name" value="Guru"></property> -->
		<property name="url" value="http://guru.edu"></property>
	</bean>

	<bean id="clientMainPc" class="org.springframework.aop.support.ControlFlowPointcut">
		<constructor-arg type="java.lang.Class" index="0"
			value="com.venus.spring.client.Client" />
		<constructor-arg type="java.lang.String" index="1"
			value="main" />
	</bean>

	<bean id="laroundClientMainAdvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor">
		<property name="advice" ref="laround"></property>
		<property name="pointcut" ref="clientMainPc"></property>
	</bean>

	<bean id="csProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="proxyInterfaces">
			<list>
				<value> com.venus.spring.service.CustomerService</value>
			</list>
```

```
		</property>
		<property name="interceptorNames">
			<list>
				<value>laroundClientMainAdvisor</value>
			</list>
		</property>

		<property name="target" ref="csImpl"></property>
	</bean>

</beans>
```

```
package com.venus.spring.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.venus.spring.service.CustomerService;

public class Client {
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/spring/config/applicationContext.xml");

	public static void main(String args[]) {
		CustomerService service = (CustomerService) context.getBean("csProxy");
		service.printName("Guru");
		System.out.println();
		service.printUrl();
		System.out.println();
		try {
			service.printException();
		} catch (Exception e) {
			System.out.println("Exception raised with message : "
					+ e.getMessage());
		}
	}
}
```

```
/*
 Before calling :: printName with arguments :: 1 on :: class
com.venus.spring.service.CustomerServiceImpl
Business Method : printName()==>Guru
After calling :: printName on class com.venus.spring.service.CustomerServiceImpl with
arguments :: 1  giving return value :: Guru

 Before calling :: printUrl with arguments :: 0 on :: class
com.venus.spring.service.CustomerServiceImpl
Business Method: printUrl() ==>http://guru.edu
```

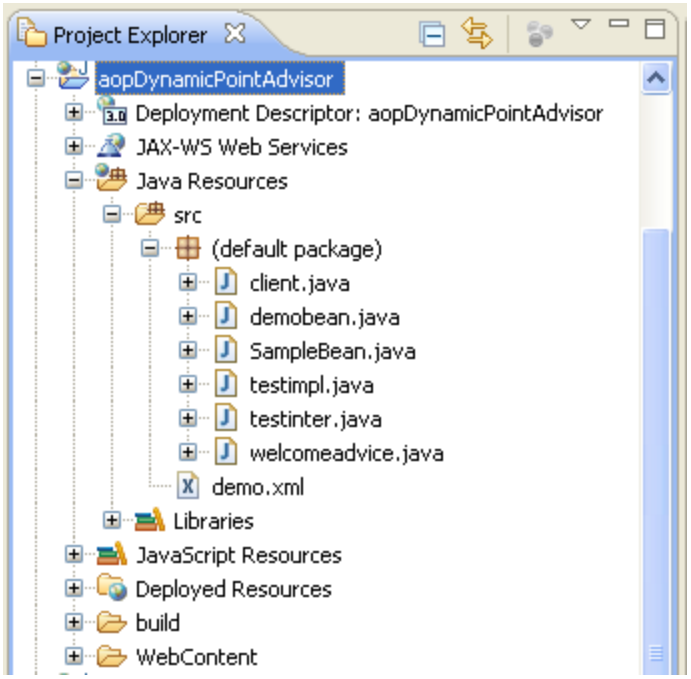
After calling :: printUrl on class com.venus.spring.service.CustomerServiceImpl with arguments :: 0 giving return value :: http://guru.edu

Before calling :: printException with arguments :: 0 on :: class com.venus.spring.service.CustomerServiceImpl
Exception catched in advice : Custom Exception
After calling :: printException on class com.venus.spring.service.CustomerServiceImpl with arguments :: 0 giving return value :: null

*/

==========

AOP DynamicPointAdvisor Program:




```
public interface testinter {
	void sayHello();
}


public class testimpl implements testinter {
	public void sayHello() {
		System.out.println("sayHello.....");
	}
}
```

```
public class SampleBean {
	public void abc(testinter ti) {
		ti.sayHello();
	}

}

public class demobean {
	public void x(testinter ti) {
		ti.sayHello();
	}

	public void y(testinter ti) {
		ti.sayHello();
	}
}


import org.springframework.aop.*;
import java.lang.reflect.*;

public class welcomeadvice implements MethodBeforeAdvice {
	public void before(Method m, Object args[], Object target) throws Exception {
		System.out.println("Now advice before advice injected");
	}
}


<?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="sb" class="SampleBean" />
	<bean id="id1" class="testimpl" />
	<bean id="id2" class="welcomeadvice" />
	<bean id="id3" class="demobean" />
	<bean id="mypointcut" class="org.springframework.aop.support.ControlFlowPointcut">
		<constructor-arg>
			<value type="java.lang.Class">demobean</value>
		</constructor-arg>
		<constructor-arg>
			<value>y</value>
		</constructor-arg>
	</bean>
	<bean id="advisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
		<property name="advice" ref="id2" />
		<property name="pointcut" ref="mypointcut" />
	</bean>
```

```
        <bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <value>testinter</value>
                </property>
                <property name="interceptorNames">
                        <list>
                                <value>advisor</value>
                        </list>
                </property>
                <property name="target">
                        <ref bean="id1" />
                </property>
        </bean>
</beans>



import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
public class client
{
        public static void main(String args[ ])
        {
                Resource res = new ClassPathResource("demo.xml");
                BeanFactory factory = new XmlBeanFactory(res);
                testinter t =(testinter)factory.getBean("proxy");
                demobean db =(demobean)factory.getBean("id3");
                db.x(t);
                System.out.println("--------------------");
                db.y(t);
                System.out.println("--------------");
                t.sayHello();
                SampleBean  ob =(SampleBean)factory.getBean("sb");
                ob.abc(t);
        }
}

/*
Output:

        Apr 8, 2013 3:23:37 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
        INFO: Loading XML bean definitions from class path resource [demo.xml]
        sayHello......
        --------------------
        Now advice before advice injected
        sayHello......
        --------------
```

```
        sayHello.....
        sayHello.....
*/
```

==========

Schema based AOP support

Spring2.0 also offers support for defining aspects using the new "**aop**" namespace tags. To use the aop namespace tags we need to import the spring-aop schema.

Within our Spring configutations, all aspect and advisor elements must be placed within an <aop:config> element (we can have morethan one <aop:congig> element in an application context configuration). It can contain pointcut, advisor and aspect elements.

Declaring an aspect: Using the schema support, an aspect is simply a regular Java object defined as a bean in spring application context. The state and behaviour is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.

An aspect is declared using the <aop:aspect> element.

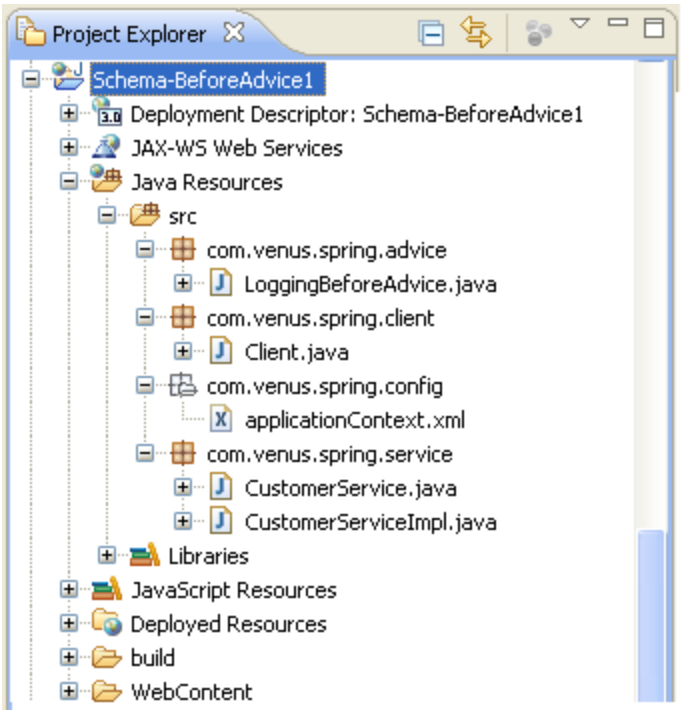
Declaring a pointcut: A named point cut can be declared inside an <aop:config> element, enabling the pointcut definition to be shared across several aspects and advisors.

==========

Before Advice based on Schema based AOP :

Before advice runs before a matched method execution. It is declared inside an <aop:aspect> using the <aop:before> element.

The method attribute identifies a method that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before data access operation is executed (a method execution join point matched by the pointcut expression), the method on the aspect bean will be invoked.




```
package com.venus.spring.service;

public interface CustomerService {
	String printName();

	String printUrl();

	void printException();
}


package com.venus.spring.service;

public class CustomerServiceImpl implements CustomerService {
	private String name;
	private String url;

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

	public void setUrl(String url) {
```

```
		this.url = url;
	}

	@Override
	public String printName() {
		System.out.println("Business Method : printName()==>" + name);
		return name;
	}

	@Override
	public String printUrl() {
		System.out.println("Business Method: printUrl() ==>" + url);
		return url;
	}

	@Override
	public void printException() {
		throw new IllegalArgumentException("Custom Exception");
	}
}


package com.venus.spring.advice;

public class LoggingBeforeAdvice {
	public void myBefore() {
		System.out.println("LoggingBeforeAdvice.before");
	}
}


<?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"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

	<bean id="csImpl" class="com.venus.spring.service.CustomerServiceImpl">
		<!-- <property name="name" value="Guru"></property> <property name="url"
			value="http://guru.edu"></property> -->
	</bean>
	<bean id="lba" class="com.venus.spring.advice.LoggingBeforeAdvice" />


	<aop:config>
```

```
		<!-- <aop:pointcut id="csPc"
expression="within(com.venus.spring.service.CustomerServiceImpl)"/>
			<aop:aspect ref="lba"> <aop:before method="myBefore" pointcut-
ref="csPc"/>
			</aop:aspect> -->
		<aop:aspect ref="lba">
			<aop:before method="myBefore"
				pointcut="within(com.venus.spring.service.CustomerServiceImpl)"
/>
		</aop:aspect>
	</aop:config>

</beans>
```

```
package com.venus.spring.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.venus.spring.service.CustomerService;

public class Client {
	private static ApplicationContext context = new ClassPathXmlApplicationContext(
			"com/venus/spring/config/applicationContext.xml");

	public static void main(String args[]) {
		CustomerService service = (CustomerService) context.getBean("csImpl");
		service.printName();
		System.out.println();
		service.printUrl();
		System.out.println();
		try {
			service.printException();
		} catch (Exception e) {
			System.out.println("Exception raised with message : "
					+ e.getMessage());
		}
	}
}

/*
 * org.springframework.context.support.AbstractApplicationContext prepareRefresh
 * INFO: Refreshing
 * org.springframework.context.support.ClassPathXmlApplicationContext@179c285:
 * startup date ; root of context hierarchy ,
 * org.springframework.beans.factory.xml.XmlBeanDefinitionReader
 * loadBeanDefinitions INFO: Loading XML bean definitions from class path
```

```
* resource [com/venus/spring/config/applicationContext.xml]
* PM org.springframework.beans.factory.support.DefaultListableBeanFactory
* preInstantiateSingletons INFO: Pre-instantiating singletons in
* org.springframework.beans.factory.support.DefaultListableBeanFactory@bb7465:
* defining beans [lmba,csImpl,csProxy]; root of factory hierarchy Before
* calling :: printName with arguments :: 0 on ::
* com.venus.spring.service.CustomerServiceImpl@1cbfe9d Business Method :
* printName()==>Guru
*
* Before calling :: printUrl with arguments :: 0 on ::
* com.venus.spring.service.CustomerServiceImpl@1cbfe9d Business Method:
* printUrl() ==>http://guru.edu
*
* Before calling :: printException with arguments :: 0 on ::
* com.venus.spring.service.CustomerServiceImpl@1cbfe9d Exception raised with
* message : Custom Exception
*/
```

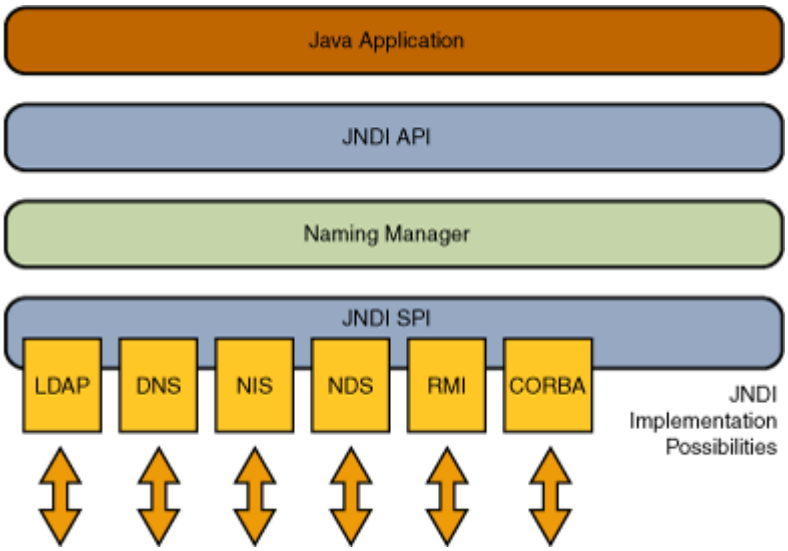
JNDI

JNDI

The Java Naming and Directory Interface™ (JNDI) is an application programming interface (API) that providesnaming and directory functionality to applications written using the Java™ programming language. It is defined to be independent of any specific directory service implementation. Thus a variety of directories -new, emerging, and already deployed can be accessed in a common way.

Architecture

The JNDI architecture consists of an API and a service provider interface (SPI). Java applications use the JNDI API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be plugged in transparently, thereby allowing the Java application using the JNDI API to access their services. See the following figure:



Packaging

JNDI is included in the Java SE Platform. To use the JNDI, you must have the JNDI classes and one or more service providers. The JDK includes service providers for the following naming/directory services:

- Lightweight Directory Access Protocol (LDAP)
- Common Object Request Broker Architecture (CORBA) Common Object Services (COS) name service
- Java Remote Method Invocation (RMI) Registry
- Domain Name Service (DNS)

The JNDI is divided into five packages:

- javax.naming
- javax.naming.directory
- javax.naming.ldap
- javax.naming.event
- javax.naming.spi

The next part of the lesson has a brief description of the JNDI packages.

Naming Package

The javax.naming package contains classes and interfaces for accessing naming services.

Context

The javax.naming package defines a Context interface, which is the core interface for looking up, binding/unbinding, renaming objects and creating and destroying subcontexts.

Lookup

The most commonly used operation is lookup(). You supply lookup() the name of the object you want to look up, and it returns the object bound to that name.

Bindings

listBindings() returns an enumeration of name-to-object bindings. A binding is a tuple containing the name of the bound object, the name of the object's class, and the object itself.

List

list() is similar to listBindings(), except that it returns an enumeration of names containing an

object's name and the name of the object's class. list() is useful for applications such as browsers that want to discover information about the objects bound within a context but that don't need all of the actual objects. Although listBindings() provides all of the same information, it is potentially a much more expensive operation.

Name

Name is an interface that represents a generic name--an ordered sequence of zero or more components. The Naming Systems use this interface to define the names that follow its conventions.

References

Objects are stored in naming and directory services in different ways. A reference might be a very compact representation of an object.

The JNDI defines the Reference class to represent reference. A reference contains information on how to construct a copy of the object. The JNDI will attempt to turn references looked up from the directory into the Java objects that they represent so that JNDI clients have the illusion that what is stored in the directory are Java objects.

The Initial Context

In the JNDI, all naming and directory operations are performed relative to a context. There are no absolute roots. Therefore the JNDI defines an InitialContext, which provides a starting point for naming and directory operations. Once you have an initial context, you can use it to look up other contexts and objects.

Exceptions

The JNDI defines a class hierarchy for exceptions that can be thrown in the course of performing naming and directory operations. The root of this class hierarchy is NamingException. Programs interested in dealing with a particular exception can catch the corresponding subclass of the exception. Otherwise, they should catch NamingException.

Directory and LDAP Packages

Directory Package

The javax.naming.directory package extends the javax.naming package to provide functionality for accessing directory services in addition to naming services. This package allows applications to retrieve associated with objects stored in the directory and to search for objects using specified attributes.

The Directory Context

The DirContext interface represents a directory context. DirContext also behaves as a naming context by extending the Context interface. This means that any directory object can also provide a naming context. It defines methods for examining and updating attributes associated with a directory entry.

Attributes

You use getAttributes() method to retrieve the attributes associated with a directory entry (for which you supply the name). Attributes are modified using modifyAttributes() method. You can add, replace, or remove attributes and/or attribute values using this operation.

Searches

DirContext contains methods for performing content based searching of the directory. In the simplest and most common form of usage, the application specifies a set of attributes possibly with specific values to match and submits this attribute set to the search() method. Other overloaded forms of search() support more sophisticated search filters.

LDAP Package

The javax.naming.ldap package contains classes and interfaces for using features that are specific to the LDAP v3that are not already covered by the more generic javax.naming.directory package. In fact, most JNDI applications that use the LDAP will find the javax.naming.directory package sufficient and will not need to use the javax.naming.ldap package at all. This package is primarily for those applications that need to use "extended" operations, controls, or unsolicited notifications.

"Extended" Operation

In addition to specifying well defined operations such as search and modify, the LDAP v3 (RFC 2251) specifies a way to transmit yet-to-be defined operations between the LDAP client and the server. These operations are called "extended" operations. An "extended" operation may be defined by a standards organization such as the Internet Engineering Task Force (IETF) or by a vendor.

Controls

The LDAP v3 allows any request or response to be augmented by yet-to-be defined modifiers, called controls . A control sent with a request is a request control and a control sent with a response is a response control . A control may be defined by a standards organization such as the IETF or by a vendor. Request controls and response controls are not necessarily paired, that is, there need not be a response control for each request control sent, and vice versa.

Unsolicited Notifications

In addition to the normal request/response style of interaction between the client and server, the LDAP v3 also specifies unsolicited notifications--messages that are sent from the server to the client asynchronously and not in response to any client request.

The LDAP Context

The LdapContext interface represents a context for performing "extended" operations, sending request controls, and receiving response controls.

Event and Service Provider Packages

Event Package

The javax.naming.event package contains classes and interfaces for supporting event notification in naming and directory services. Event notification is described in detail in the Event Notification trail.

Events

A NamingEvent represents an event that is generated by a naming/directory service. The event contains a typethat identifies the type of event. For example, event types are categorized into those that affect the namespace, such as "object added," and those that do not, such as "object changed."

Listeners

A NamingListener is an object that listens for NamingEvents. Each category of event type has a corresponding type of NamingListener. For example,
a NamespaceChangeListener represents a listener interested in namespace change events and an ObjectChangeListener represents a listener interested in object change events.

To receive event notifications, a listener must be registered with either an EventContext or an EventDirContext. Once registered, the listener will receive event notifications when the corresponding changes occur in the naming/directory service.

Service Provider Package

The javax.naming.spi package provides the means by which developers of different naming/directory service providers can develop and hook up their implementations so that the corresponding services are accessible from applications that use the JNDI.

Plug-In Architecture

The javax.naming.spi package allows different implementations to be plugged in dynamically. These implementations include those for the initial context and for contexts that can be reached from the initial context.

Java Object Support

The javax.naming.spi package supports implementors of lookup and related methods to return Java objects that are natural and intuitive for the Java programmer. For example, if you look up a printer name from the directory, then you likely would expect to get back a printer object on which to operate. This support is provided in the form of object factories.
This package also provides support for doing the reverse. That is, implementors of Context.bind() and related methods can accept Java objects and store the objects in a format acceptable to the underlying naming/directory service. This support is provided in the form of state factories.

Multiple Naming Systems (Federation)

JNDI operations allow applications to supply names that span multiple naming systems. In the process of completing an operation, one service provider might need to interact with another service provider, for example to pass on the operation to be continued in the next naming system. This package provides support for different providers to cooperate to complete JNDI operations.

Java Platform Software

JNDI is included in the Java SE Platform.

To run the applets, you can use any Java-compatible Web browser, such as Firefox, or Internet Explorer v5 or later. To ensure that your applets take full advantage of the latest features of the Java platform software, you can use the Java Plug-in with your Web browser.

Service Provider Software

The JNDI API is a generic API for accessing any naming or directory service. Actual access to a naming or directory service is enabled by plugging in a service provider under the JNDI.

A service provider is software that maps the JNDI API to actual calls to the naming or directory server. Typically, the roles of the service provider and that of the naming/directory server differ. In the terminology of client/server software, the JNDI and the service provider are the client (called the JNDI client) and the naming/directory server is the server.

Clients and servers may interact in many ways. In one common way, they use a network protocol so that the client and server can exist autonomously in a networked environment. The server typically supports many different clients, not only JNDI clients, provided that the clients conform to the specified protocol. The JNDI does not dictate any particular style of interaction between JNDI clients and servers. For example, at one extreme the client and server could be the same entity.

You need to obtain the classes for the service providers that you will be using. For example, if you plan to use the JNDI to access an LDAP directory server, then you would need software for an LDAP service provider.

The JDK comes with service providers for:

- Light Weight Directory Protocol (LDAP)
- CORBA Common Object Services naming (COS naming)
- RMI registry

•Domain Name Service (DNS)

This tutorial uses only the LDAP Service provider. When using the LDAP service provider, you need either to set up your own server or to have access to an existing server, as explained next.

Naming and Directory Server Software

Once you have obtained the service provider software, you then need to set up or have access to a corresponding naming/directory server. Setting up a naming/directory server is typically the job of a network system administrator. Different vendors have different installation procedures for their naming/directory servers. Some require special machine privileges before the server can be installed. You should consult the naming/directory server software's installation instructions.

```
import javax.naming.*; //jndi api
import java.util.*;

public class JndiConnectionTest {
	public static void main(String args[]) throws Exception {
		// Prepare Jndi properties
		Hashtable<String, String> ht = new Hashtable<String, String>();
		ht.put(Context.INITIAL_CONTEXT_FACTORY,
				"weblogic.jndi.WLInitialContextFactory");
		/*
		 * ht.put(Context.PROVIDER_URL, "t3://localhost:7001 john mypasswd");
		 */
		ht.put(Context.PROVIDER_URL, "t3://localhost:7001 weblogic weblogic");

		// create initial context object
		InitialContext ic = new InitialContext(ht);
		if (ic == null)
			System.out.println("Connection is not established");
		else
			System.out.println("Connection is established");
	}
}
```

JNDI:

==> TO provide global visibility to java objects and their references we keep them in registry s/w.
Every web server/application server gives an registry s/w.
web logic gives web logic registry
tomcat gives tomcat registry
GlassFist gives GlassFish registry
JBoss gives Jnp registry and etc....

==> java applications use jdbc api (javax.sql, java.sql pckages) to interact with DB s/w
==> java applications use jndi api (javax.naming and its sub packages) to interact with registry.
==> jdbc api, jndi api are part of jse module(jdk s/w).
==> there are two types of registry s/ws:
	1) Naming Registry s/w: ==> maintains information as key values pairs
		==> we must know keys to get the values.

ex: Telephone book (real life)
Rmi registry
COS registry
and etc...

2) Directory Registry: ==> Maintains information as key-value pairs and values can extract properties/attribute.

==> We can use either keys or extra properties to get the values.

Ex: yellow pages(category wise it give) ==> real life
windows filesystem ==> reallife
Weblogic registry
GlassFish registry
DNS registry

==> we can perform insert, update, delete and select operattions on db table using jdbc.
==> we can perform bind, unbind, rebind, lookup and list operations on Jndi registry by using jndi code.

==> bind ==> keeping object with nickname/aliasname/jndi name in registry
==> unbind ==> removing object from registry
==> rebind ==> replacing existing object with new object
==> lookup==> gathering object from registry with nickname/alias name
==> list ==> gathering all binding from registry(nicknames, objects)

==> jdbc connectin object represents connectivity between java application and db s/w. to create this on object we need jdbc properties(Driver class name, url, db user, pwd) and these jdbc properties will change based on the jdbc driver, db s/w we use.

==> Jdbc driver is the bridge between java application and db software.
diagram page 97

==> Initialcontext obj represents connectivity between java application and registry s/w.
==> Naming manager is the bridge between java application and registry s/w.
diagram page 97
==> to create this InitialContext object we need jndi properties they are
1) InitialContextFactory class
2) Provider url
These jndi properties will change based on the registry s/w we use.

==> Jndi properties of weblogic Registry:
InitialContext class name : weblogic.jndi.WLInitialContextFactory(Weblogic.jar)
Provider Url : t3://<hostname/IPaddress>:<portnumber>
Ex : t3://localhost:7001
==> javax.naming.InitialContext class gives the following method to perform jndi operations
bind(-,-), unbind(-,-), rebind(-,-), lookup(-), list(-).
diagram pageno:99

==> Each NameClassPair class object represents one binding(nickname, obj).
javax.naming.NamingEnumeration(I) is sub interface of java.util.Enumeration(I).

```
      //JndiList.java
import javax.naming.*;
import java.util.*;

public class JndiList {
      public static void main(String args[]) throws Exception {
            // prepare Jndi properties
            Hashtable<String, String> ht = new Hashtable<String, String>();
            ht.put(Context.INITIAL_CONTEXT_FACTORY,
                        "weblogic.jndi.WLInitialContextFactory");
            ht.put(Context.PROVIDER_URL, "t3://localhost:7001");
            ht.put(Context.SECURITY_PRINCIPAL, "weblogic");
            ht.put(Context.SECURITY_CREDENTIALS, "weblogic");
            // create initial context object
            InitialContext ic = new InitialContext(ht);

            // performing list operation
            // NamingEnumeration ne=ic.list("");// gives all the binding root
            // context "" ==> root context
            NamingEnumeration ne = ic.list("javax/jms");// gives all bindings of
                                                                        //
javax/jms Context
            while (ne.hasMore()) {
                  NameClassPair ncp = (NameClassPair) ne.next();// gives one binding

      // at a time
                  System.out.println(ncp.getName() + " <=======> "
                              + ncp.getClassName());

            }// while
      }// main
}// class

/*
 * output: QueueConnectionFactory <=======>
 * weblogic.rmi.cluster.ClusterableRemoteObject TopicConnectionFactory <=======>
 * weblogic.rmi.cluster.ClusterableRemoteObject
 */
```

==>**Spring Jndi:**

==> Spring Jndi provides abstraction layer on plain jandi programming by giving **org.springframework.jndi.JndiTemplate** class

==> This JndiTemplate class internally uses plain Jndi and simplify the processing of performing jndi operations on Registry s/w

==> Plain jndi programming:

==> prepare Jndi properties

==> create IntialContext object

==> perform Jndi operations

==> take care of Exception handling

==> close InitialContext object

==> spring jndi programming

==> get JndiTemplate class object

==> perform Jndi operations

Note: the remaining common operations will be taken care by JndiTemplate class.

Note: Spring jndi allows to pass Jndi properties fro SpringConfiguration file gives flexibility of modification.

==> spring Jndi internally uses plain Jndi, but it never makes programmer to bother about plain Jndi(This is nothing but getting abstraction layer). Client application of Spring application is java class. so that it can be taken as Spring bean and its progperies can be configured for Dependency Injection.

==> JndiTemplate clas gives bind(),, unbind(), rebind() and lookup() methods, but not giving list().

==> Spring technologies provides abstraction layer on plain technologies.

In this process if SpringAPI fails to support certain operation then we can use Spring supplied callBack interfaces to implement that operation by using plain technology API.

==> The interface whose methods implemented in implementation class called by underlaying container automatically is called "CallBack" interface.

Ex: The Spring Jndi supplied JndiTemplate class does provide methods to perform list operation on registry. This can be acheived by using JndiCallback interface. JndiCallback(I) gives doInContext(-) method. In this method we can write plain Jndi code to perform list operation.

//SprintJndiList.java

```
import javax.naming.*; //plain JndI to work with callback(I)
import org.springframework.context.support.*;
import org.springframework.jndi.*;
import java.util.*;
public class SprintJndiList
{
static JndiTemplate template;
//setter method for setter injection
public setTemplate(JndiTemplate template)
{
this.template=template;
}
public static void main(String args[]) throws Exception
{
/*  //prepare Jndi properties
    Hashtable<String, String> ht=new Hashtable<String, String>();
    ht.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
    ht.put(Context.PROVIDER_URL,"t3://localhost:7001");
    ht.put(Context.SECURITY_PRINCIPAL,"weblogic");
    ht.put(Context.SECURITY_CREDENTIALS,"weblogic");*/
    FileSystemXmlApplicaionContext ctx=new FileSystemXmlApplicaion("Democfg.xml");
    //implement list operation logic
    template.execute(new JndiCallback(){
    public Object doInContect(Context ctx) throws NamingException
    {
    //use plain Jndi api to implement list operation logic
    NamingEnumeration ne=ctx.list("");
        while(ne.hasMore())
        {
```

```
            NameClassPair ncp = (NameClassPair)ne.next();//gives one binding at a time
            System.out.println(ncp.getName()+" <=======> "+ncp.getClassName());

            }//while
        }     //doInContext()
        }//Inner class
        );

        }//main
        }//class
```

JMS (Java Messaging Service):

Messaging:

→ Messaging is a method of communication between "software components or applicaion". a messaging system is a peer-to-peer facility: A messaging client can send messages to and receive messages from any other client. Each client conencts to a messaging agent that provides facilities for creating, sending, receiving and reading messages.

→Messaging enables distributed communication that is loosely coupled. A component sends a message to adestination, and the recipent can retrieve the message from the destination. However the sender and the receiver "do not have to available at the same time in order to communicate". Infact the sender does not need to know anything about the receiver, nor does the receiver need to know anything about the sender. The sender and receiver need to know only what message format and what destination to use.

In this respect, messaging differs from tightly coupled technologies such as Remote Method Invocation(RMI), which require an application to know a remove applicaion's methods.

→ Messaging also differs from e-mail, which is a method of communication between people or between software applicaions and people."Messaging is used for communication between software applications or software components".

JMS API:

The Java Message Service is a java API that allows applicaions to create, send , receive and read messages.

→ It defines a common set of interfaces and associated semantics that llow programs written in the java programming language to communicate with other messaging implementations.

→ The JMS API enables communication that is not only loosely coupled but also **Asynchronous.**→ A JMs provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.

→ **Reliable:** the JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are avai;ane for applications that ca afford to miss messages or to receive duplicate messages.

JMS messaging models

JMS supports two messaging models: point-to-point (PTP) and publish/subscribe. The terms "message producer" and "message consumer" describe clients that send and receive messages in either of the models, although each model has its own specific terms for producers and consumers.

Point-to-point messaging

The point-to-point messaging model is based on message Queues. A QueueSender (producer) sends a message to a specified Queue. A QueueReceiver (consumer) receives messages from the Queue. A Queue can have multiple QueueSenders and QueueReceivers, but an individual message can only be delivered to one QueueReceiver. If multiple QueueReceivers are listening for messages on a Queue, WebLogic JMS determines which will receive the next message. If no QueueReceivers are listening on the Queue, messages remain in the Queue until a QueueReceiver attaches to the Queue.

Publish/subscribe messaging

The publish/subscribe messaging model is organized around Topics. TopicPublishers (producers) send messages to a Topic. TopicSubscribers (consumers) retrieve messages from a Topic. Unlike the point-to-point model, many TopicSubscribers can receive the same message.

The PTP and pub/sub messaging models are very similar. They are implemented with classes that extend common root classes. For example, the PTP **javax.jms.Queue** class and the pub/sub **javax.jms.Topic** class each extend a common j**avax.jms.Destination** class.

Messages can be NON-PERSISTENT or PERSISTENT. NON-PERSISTENT messages are not stored in a database, and may be lost during a failure. JMS specifies that a NON-PERSISTENT message will be delivered *at most* once. A PERSISTENT message is not considered sent until it has been stored in the database. JMS guarantees that PERSISTENT messages are delivered at least once. WebLogic JMS writes PERSISTENTmessages to database via a JDBC connection pool assigned to JMS in the weblogic.properties file.

JMS objects

You use the javax.jms client APIs in your JMS client applications. You retrieve the initial JMS object, a ConnectionFactory, using WebLogic JNDI. The Topics and Queues your application uses are also retrieved through JNDI. An administrator defines ConnectionFactories, Topics, and Queues in the weblogic.properties file, and the WebLogic Server adds them to the JNDI space during startup. WebLogic JMS provides default connection factories, so it is not necessary to define connection factories for most applications.

Connections

A **Connection** represents an open connection to the messaging system. It creates the associated server-side and client-side objects that manage the state of the JMS connection. It also creates Sessions, objects that manage the active exchange of messages between the client and the messaging system.

Connections are created by a **ConnectionFactory** obtained with a JNDI lookup. The ConnectionFactory and Connection classes for the two JMS messaging models extend ConnectionFactory and Connection:

Point-to-point:

javax.jms.QueueConnectionFactory
javax.jms.QueueConnection

Publish/subscribe:

javax.jms.TopicConnectionFactory
javax.jms.TopicConnection

In the WebLogic Server, JMS traffic is multiplexed with the other WebLogic services on the client connection to the server. No additional TCP/IP connections are created for JMS. Servlets and other server-side objects may also obtain JMS Connections.

A Connection may contain a client identifier that is used for clients with durable subscriptions. Durable subscriptions are a feature unique to the pub/sub messaging model, so client IDs are only of consequence with TopicConnections; QueueConnections also contain Client IDs, but JMS does not use them. The client ID can be supplied in two ways:

- The preferred method, according to the JMS specification, is to configure the ConnectionFactory with the client ID. For WebLogic JMS, this means adding a separate TopicConnectionFactory definition to the weblogic.properties file for each client ID. Clients then look up their own TopicConnectionFactory in JNDI and use them to create Connections containing their own client Ids.

Sessions

A JMS Session is a context for producing and consuming messages. A Session creates MessageProducers and MessageConsumers, the objects that transmit and receive messages. A Session can also create a temporary Queue or Topic, which exists only for the duration of the session. Sessions are created by the JMS Connection. QueueConnections create QueueSessions and TopicConnections create TopicSessions.

Sessions can be transacted. A transacted Session has one active transaction at a time. Messages sent or received during a transaction are treated as an atomic unit. If the client rolls back the transaction, the messages it received during the transaction are not acknowledged and messages it sent are discarded. When a client commits a transaction, all of the messages received during the transaction are acknowledged by the messaging system and messages it sent are accepted for delivery. JMS can participate in distributed transactions with other Java services that use the Java Transaction Service (JTS), such as EJB

In non-transacted Sessions, the client creating the Session selects one of three acknowledgement modes. The most efficient in terms of resource usage is **DUPS_OK_ACKNOWLEDGE**, which allows the session to "**lazily**" acknowledge receipt on behalf of the client. This mode can result in duplicate messages if there is a failure and you should avoid it if your application is not able to handle duplicates. If you create a Session with **AUTO_ACKNOWLEDGE**, the Session acknowledges receipt of a message when the client method that receives the message has returned from processing it. A Session created with **CLIENT_ACKNOWLEDGE** relies on the client to call an acknowledge method on a received message. Once this method is called, the Session acknowledges all messages received since the last acknowledgement. This allows the client to receive and process a batch of messages and then acknowledge them all with one call.

Destinations -- Queues and Topics

JMS Queues and Topics extend **javax.jms.Destination**. Queues manage messages for the point-to-point messaging model and Topics manage messages for the publish/subscribe model.

Client programs retrieve Queues and Topics with JNDI. Like **ConnectionFactories**, **Queues** and **Topics** can be configured in the **weblogic.properties** file, in the WebLogic Console, or programmatically by using JMS interfaces. As part of their configuration in theweblogic.properties file and in the WebLogic Console, Queues and Topics are automatically bound to JNDI names. If configured programmatically, the message destinations must be bound explicitly using the appropriate JNDI interface.

Clients can also create temporary Queues and Topics that exist only for the duration of the JMS connection in which they are created.

On the client side, Queue and Topic objects are handles to the object on the server. Their only methods simply return their names. To access them for messaging, you create message producers and consumers that attach to them.

MessageProducers and MessageConsumers

The Session object creates message senders and receivers attached to Queues and Topics. The message sender/receiver objects are subclassed from the MessageProducer and MessageConsumer classes. A MessageProducer sends messages to a Queue or Topic. A MessageConsumer receives messages from a Queue or Topic. The point-to-point and pub/sub messaging models have different names for these objects, as this table shows:

Base class	Point-to-point	Publish/subscribe
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

The **QueueReceiver** and **TopicSubscriber objects** can be created with a message selector that specifies which messages are to be delivered to the client. The **selector** is a String patterned after the **WHERE clause** of an **SQL SELECT statement**. The selector filters messages based on values of properties that are set in the message. Selectors are discussed in Using message selectors.

The client can request that messages be delivered synchronously or asynchronously. If synchronous, the client calls a receive() method on the QueueReceiver or TopicSubscriber and blocks until a message arrives. If asynchronous, the client provides an implementation of the JMS MessageListener interface. When a message arrives, the onMessage() method of that implementation is called to receive the message.

With the point-to-point model, clients have the option of browsing queues by creating a QueueBrowser object in their QueueSession. This object produces an enumeration of the messages in the queue at the time the QueueBrowser is created -- a snapshot. The client can view the messages in the queue, but the messages are not considered "read" or removed from the queue.

Messages

A JMS message (javax.jms.Message) contains a set of standard header fields, application-specific properties, and a message body.

Message header fields

JMS Messages contain a standard set of headers that are always transmitted with the message. They are available to message consumers that receive messages and some fields can be set by the message producers that send messages.

JMSDestination

The JMSDestination field contains the destination (Queue or Topic name) where the message is to be delivered. The value of the field is set when the message is sent via the client's message producer. Any previous value is ignored.

JMSDeliveryMode

The JMSDeliveryMode field contains either NON_PERSISTENT or PERSISTENT. NON_PERSISTENT messages are not stored in the JMS database, so they can be lost following a failure. A PERSISTENT message is stored in the JMS database when the message is sent. The send operation is not considered successful until the message has been written to the database. A NON_PERSISTENT message is guaranteed to be delivered at most once. A PERSISTENT message is guaranteed to be delivered once, and only once.

JMSTimestamp

The JMSTimestamp field contains the time the message was transferred from the client's MessageProducer to WebLogic JMS. The value stored in the field is a Java millis time value. The timestamp is written in the message when WebLogic JMS accepts the message for delivery, which may be later than when the client sends the message.

JMSCorrelationID

The JMSCorrelationID field can be used to link messages. The field can hold a WebLogic JMS message ID, an application-specific string, or a byte array. A common use for this field is to set up requests and responses. When a client sends a message, it remembers its message ID. When a receiving client receives the message, it copies the message ID into the JMSCorrelationID field of a response message that it sends to the first client. All JMS message IDs start with an ID: prefix. If the JMSCorrelationID is used for some other application-specific string, it *must not* begin with the ID: prefix.

The byte[] JMSCorrelationID is available for external JMS providers and is not supported by WebLogic JMS. CallingsetJMSCorrelationIDAsBytes() throws a java.lang.UnsupportedOperationException.

JMSReplyTo

A client stores a Queue or Topic name in the JMSReplyTo field before sending a message. It is up to the receiving client to decide whether to reply to the message. The JMSReplyTo field can also be NULL, which may have a semantic meaning to the receiving client, such as a notification event.

JMSRedelivered

The JMSRedelivered field is only of interest to a receiving client. If set, then JMS may have delivered the Message earlier. It can mean that the client has already received the message, but did not acknowledge it, or that the session's recover() method was called to restart the session

beginning after the last acknowledged message. See the section on Sessions for more information.

JMSType
The JMSType field contains the message type identifier set by the sending client. The JMS specification allows some flexibility with this field in order to accommodate diverse JMS providers. Some messaging systems allow applications to formally define message types within the messaging system. For those systems, the JMSType field could contain a message type ID that provides access to the stored type definitions. WebLogic JMS does not restrict the use of this field.

JMSExpiration
The client can set a time-to-live value on a message when it is sent. WebLogic JMS calculates the JMSExpiration value as the sum of the client's time-to-live and the current GMT. If the client specifies time-to-live as 0, JMSExpiration is set to 0, which means the message never expires.

WebLogic JMS periodically scans the JMS database and deletes any persistent messages that have expired, including messages waiting to be delivered to durable subscribers.

JMSPriority
JMS defines ten priority levels, 0 to 9. The lowest priority is 0. The sending client sets the priority level in the JMSPriority field before sending the message. Levels 0-4 are to be considered gradations of *normal* priority and level 5-9 are to be considered gradations of*expedited* priority. JMS does not require that JMS providers honor message priority, but only to "do their best." Currently, WebLogic JMS does not honor message priority. However, a simple priority mechanism can be implemented using message property fields.

Message property fields

The property fields section of a Message contains additional header fields added by the sending client. The properties are standard Java name/value pairs. The values can be boolean, byte, short, int, long, float, double, and String data types.

Although, message property fields may be used for application-specific purposes, JMS provides them primarily for use in message selectors. Message selectors allow clients to choose the messages they want to receive by providing a simple query string. The sending client can set message property fields to describe or classify a message in a standardized way so that interested receivers can choose it with their message selector. Since message selectors cannot reference the contents (body) of a message, some fields from the message contents may be duplicated as message property fields.

Message body

The Message body contains the message content. JMS defines six message types:

javax.jms.Message
Contains only message headers and properties, but no message body. All other message types extend Message.

javax.jms.BytesMessage
Contains a stream of bytes, which must be understood by the sender and receiver. The access

methods for this message type are stream-oriented readers and writers based on the java.io.DataInputStream and java.io.DataOutputStream.

javax.jms.StreamMessage
Similar to a BytesMessage, except that only Java primitive types are written to or read from the stream.

javax.jms.ObjectMessage
Holds a single serializable Java object.

javax.jms.MapMessage
Holds a set of name/value pairs where names are Strings and values are Java primitive types. The pairs can be read sequentially or randomly by specifying a name.

javax.jms.TextMessage
Holds a single String. In addition to applications where it is natural to represent messages as text, the TextMessage can carry more complex data represented with XML.

Import statements

WebLogic JMS uses the JavaSoft JMS API, javax.jms. You also need JNDI with JMS, and if you use transactions you need the JTS classes. Here are the basic imports for JMS applications:

```
import java.util.*;
import javax.naming.*;
import javax.jms.*;
import javax.transaction.*;
```

WebLogic JMS provides one public API for use with server session pools, an optional application server facility described in the JMS specification. If you implement a server session pool application, add this package to your import list:

```
import weblogic.jms.ServerSessionPoolFactory.*
```

Concurrent use of JMS objects

JMS Connections, ConnectionFactories, Topics, and Queues support concurrent use. Other JMS objects -- including QueueSenders and QueueReceivers, TopicPublishers and TopicSubscribers, and Sessions -- can be accessed by only one thread at a time.

Using transactions with JMS

To support transactions, JMS uses Java Transaction Service (JTS), which is implemented in the WebLogic Server. There are two ways to use transactions with JMS. If you are using only JMS in your transactions, you can create a transacted JMS session. If you are mixing other operations, such as EJB, with JMS operations, you should use a JTS UserTransaction in a non-transacted JMS session.

Transacted JMS sessions
Transactions in JMS transacted sessions are chained -- whenever you commit or roll back a

transaction, another transaction automatically begins. The JMS Connection has a TransactionTimeout value that specifies the number of seconds a transaction can run before it is timed out. When a transaction times out, it is rolled back. The default TransactionTimeout is 3600 seconds (one hour). If you have long-running transactions you might need to increase this value to allow your transactions to complete.

JTS UserTransactions

If you are using JMS and EJBs, or have already started a user transaction, WebLogic JMS respects the existing UserTransaction. If you create a JMS transacted session when you already have a UserTransaction, an exception is thrown when you call any transaction method on the JMS Session object.

To combine JMS and EJB operations in a transaction, you can start a transaction from an EJB or by getting ajavax.transaction.UserTransaction with a JNDI lookup. Here is some code to show how to set up a client for mixed EJB and JMS operations in a transaction:

```
public final static String JNDI_FACTORY=
          "weblogic.jndi.WLInitialContextFactory";
public final static String JTS_USER_XACT=
          "javax.transaction.UserTransaction";
Context ctx = new InitialContext();
UserTransaction xact = ctx.lookup(JTS_USER_XACT);
// start a transaction
xact.begin();
//
// perform some JMS and EJB operations
//
// commit the transaction
xact.commit()
```

Setting up a JMS client

JMS clients create several objects to begin using WebLogic JMS. Here are the general steps for setting up a JMS client:

1.Look up a JMS ConnectionFactory in JNDI.
2.Look up Queues and Topics in JNDI.
3.Use the ConnectionFactory to create a Connection.
4.Use the Connection to create a Session.
5.Use the Session and the Queues or Topics to create message producers and consumers.
6.Start the connection.

The **init()** method is as follows:

```
public final static String JNDI_FACTORY=
          "weblogic.jndi.WLInitialContextFactory";
public final static String JMS_FACTORY=
          "javax.jms.QueueConnectionFactory";
public final static String QUEUE="javax.jms.exampleQueue";
```

```
private QueueConnectionFactory qconFactory;
private QueueConnection qcon;
private QueueSession qsession;
private QueueSender qsender;
private Queue queue;
private TextMessage msg;

/**
 * Create all the necessary objects for sending
 * messages to a JMS queue.
 */
public void init(Context ctx, String queueName)
    throws NamingException, JMSException
{
  // The ctx object is a JNDI initial context passed
  // in by the main() method.

  // Look up the ConnectionFactory
  qconFactory = (QueueConnectionFactory)
              ctx.lookup(JMS_FACTORY);

  // Get a QueueConnection from the
  // QueueConnectionFactory
  qcon = qconFactory.createQueueConnection();

  // Get a QueueSession that is not transacted
  // and acknowledges automatically
  qsession = qcon.createQueueSession(false,
              Session.AUTO_ACKNOWLEDGE);

  // Look up the Queue
  queue = (Queue) ctx.lookup(queueName);

  // Create a message producer (QueueSender)
  // for the Queue we looked up.
  qsender = qsession.createSender(queue);

  // Create a message object
  msg = qsession.createTextMessage();

  // Start the connection
  qcon.start();
}
```

You use the same process for setting up a Topic session, substituting the point-to-point object names with the publish/subscribe names: **TopicConnectionFactory**, **TopicConnection**, **Topic**, and **TopicSession**.

Sending messages

To send a message, you first create the message, fill it in with the information you want to send, and then transmit it to its destination through the message producer. JMS provides six different message types, which are described in the section Message types earlier in this document.

You create a message by calling a method on the Session object. Here is the code to create a TextMessage, set the text of the message, and send the message off to a Queue:

```
msg = qsession.createTextMessage();
msg.setText("Welcome to WebLogic JMS");
qsender.send(msg);
```

The body of a TextMessage holds a single String. The send() message in the example puts the String into the message and calls the QueueSender's **send()** method:

```
/**
 * Send a message to a JMS queue.
 */
public void send(String message)
    throws JMSException
{
  msg.setText(message);
  qsender.send(msg);
}
```

Receiving messages

In the QueueSend example, messages originate from the client's QueueSender. A QueueReceiver object to receive the messages the QueueSend produces. The code for setting up the receiver is very similar to QueueSend. If your application receives messages, you choose whether to receive them synchronously or asynchronously. The QueueReceive example receives messages asynchronously by implementing the javax.jms.MessageListener interface, which includes an onMessage() method. Here is the code from the init() method of the examples.jms.QueueReceive example that establishes the message listener:

```
qreceiver = qsession.createReceiver(queue);
qreceiver.setMessageListener(this);
```

The first line creates the QueueReceiver object on the Queue. The second line sets the object that implements the MessageListener interface, in this case examples.jms.QueueReceive itself. When a message is delivered to the QueueSession, it is passed to theexamples.jms.QueueReceive.onMessage() method.

The **onMessage()** method processes messages received through the QueueReceiver. In the QueueReceive , it checks that the message is a TextMessage and if so, prints the text of the message. If it receives a different message type, it uses the message's **toString()**method to display the message contents. In a message receiver, it is good practice to check that the the received message is of the type the message handler method expects. Here is the implementation of the **onMethod()** interface from the QueueReceive :

```
public void onMessage(Message msg)
```

```
 {
  try {
    String msgText;
    if (msg instanceof TextMessage) {
      msgText = ((TextMessage)msg).getText();
    } else { // If it is not a TextMessage...
      msgText = msg.toString();
    }

    System.out.println("Message Received: "+ msgText );

    if (msgText.equalsIgnoreCase("quit")) {
      synchronized(this) {
        quit = true;
        this.notifyAll(); // Notify main thread to quit
      }
    }
  } catch (JMSException jmse) {
    jmse.printStackTrace();
  }
 }
```

To receive messages synchronously, the example would call **qreceiver.receive()** for each message instead of setting a listener with **qreceiver.setMessageListener()**. The **receive()** method blocks and waits for a message.

To work with the following programs first we have to configure the JMS server in web logic : the procedure is as follows:

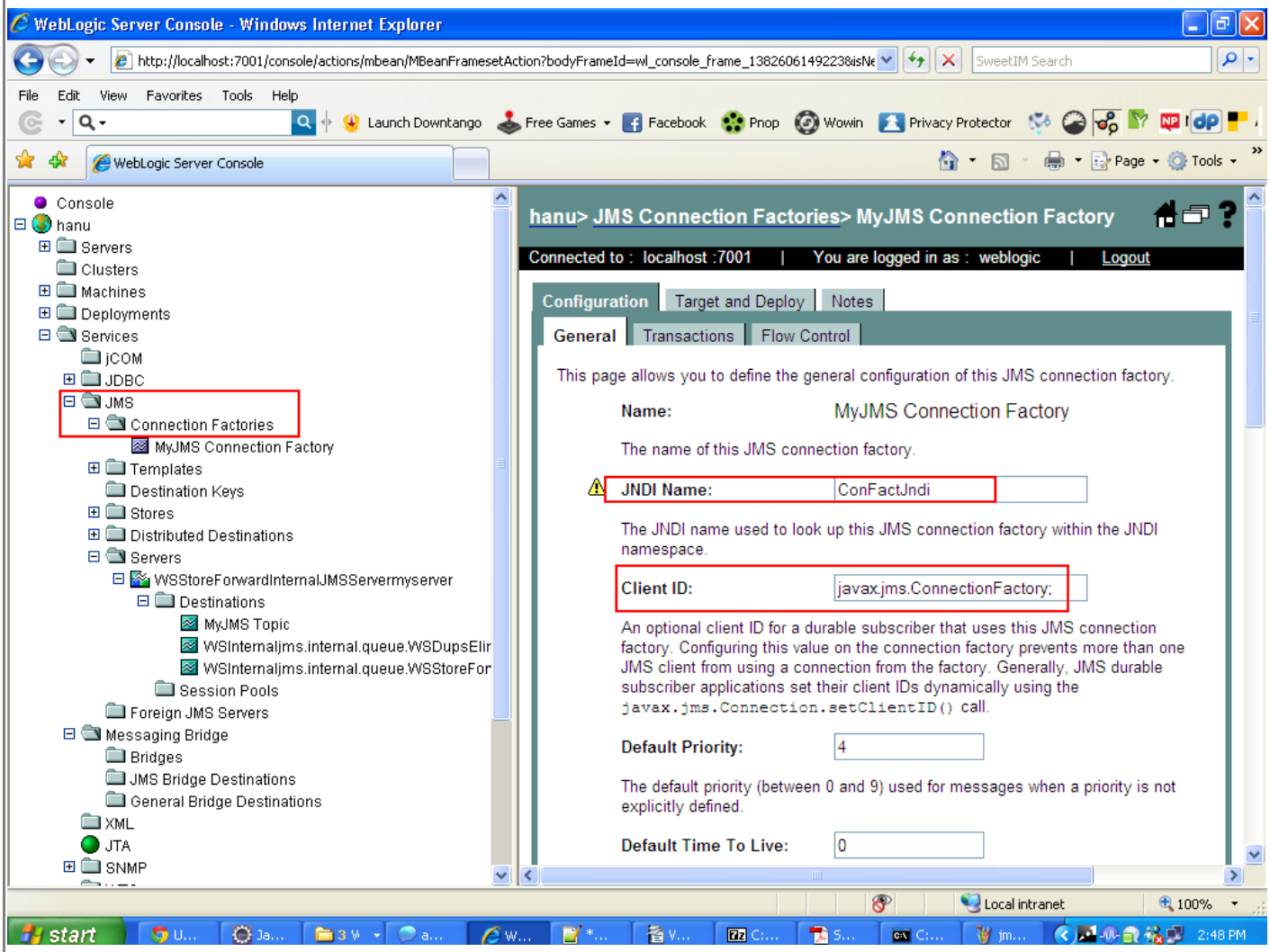
==> Start the web logic server (startWeblogic)

==> Run the following program to test the connection is established or not:

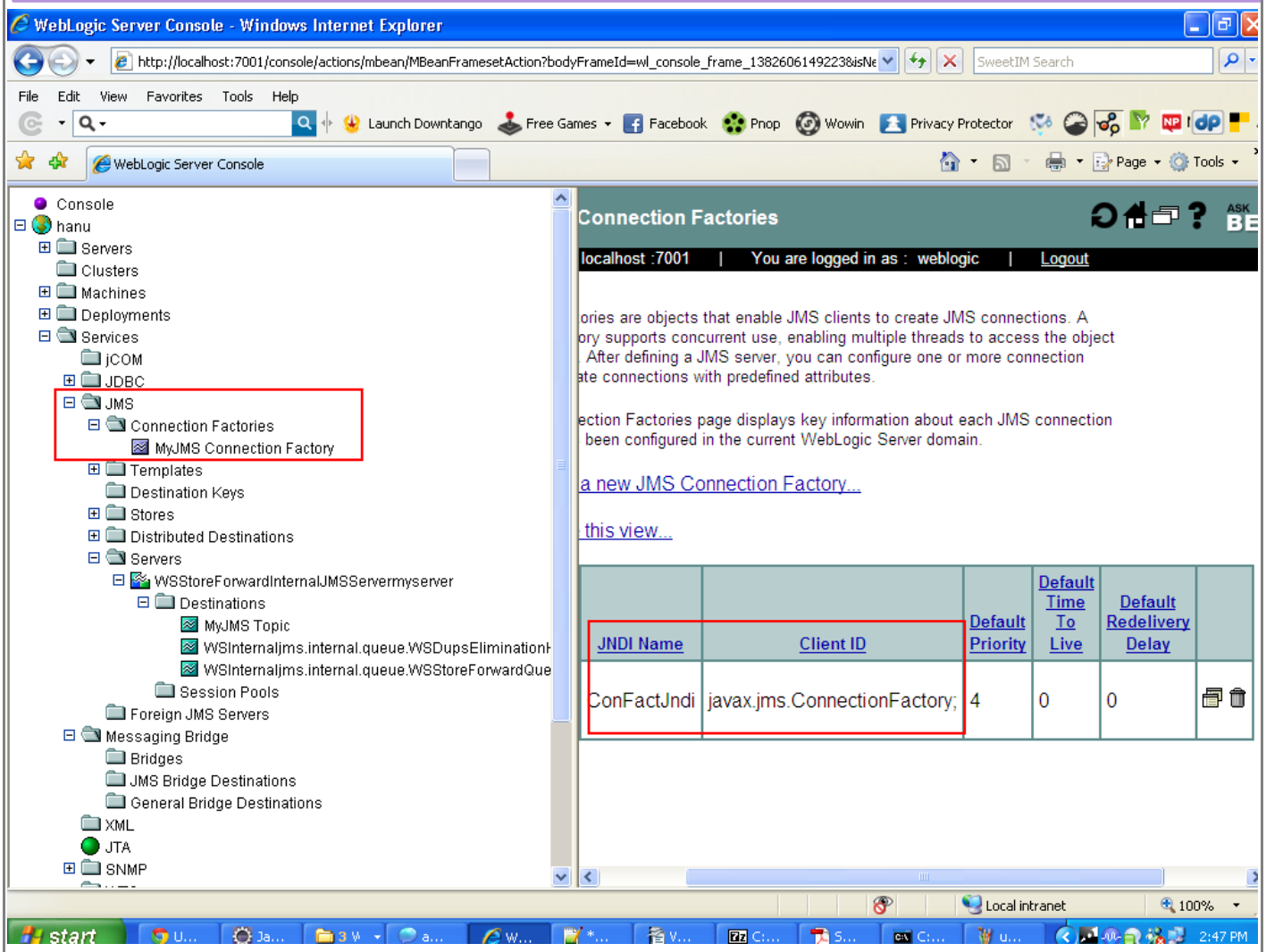
```
import javax.naming.*; //jndi api
import java.util.*;

public class JndiConnectionTest {
      public static void main(String args[]) throws Exception {
            // Prepare Jndi properties
            Hashtable<String, String> ht = new Hashtable<String, String>();
            ht.put(Context.INITIAL_CONTEXT_FACTORY,
                        "weblogic.jndi.WLInitialContextFactory");
            /*
             * ht.put(Context.PROVIDER_URL, "t3://localhost:7001 john mypasswd");
             */
            ht.put(Context.PROVIDER_URL, "t3://localhost:7001 weblogic weblogic");

            // create initial context object
            InitialContext ic = new InitialContext(ht);
            if (ic == null)
                  System.out.println("Connection is not established");
```

```
            else
                  System.out.println("Connection is established");
      }
}
```

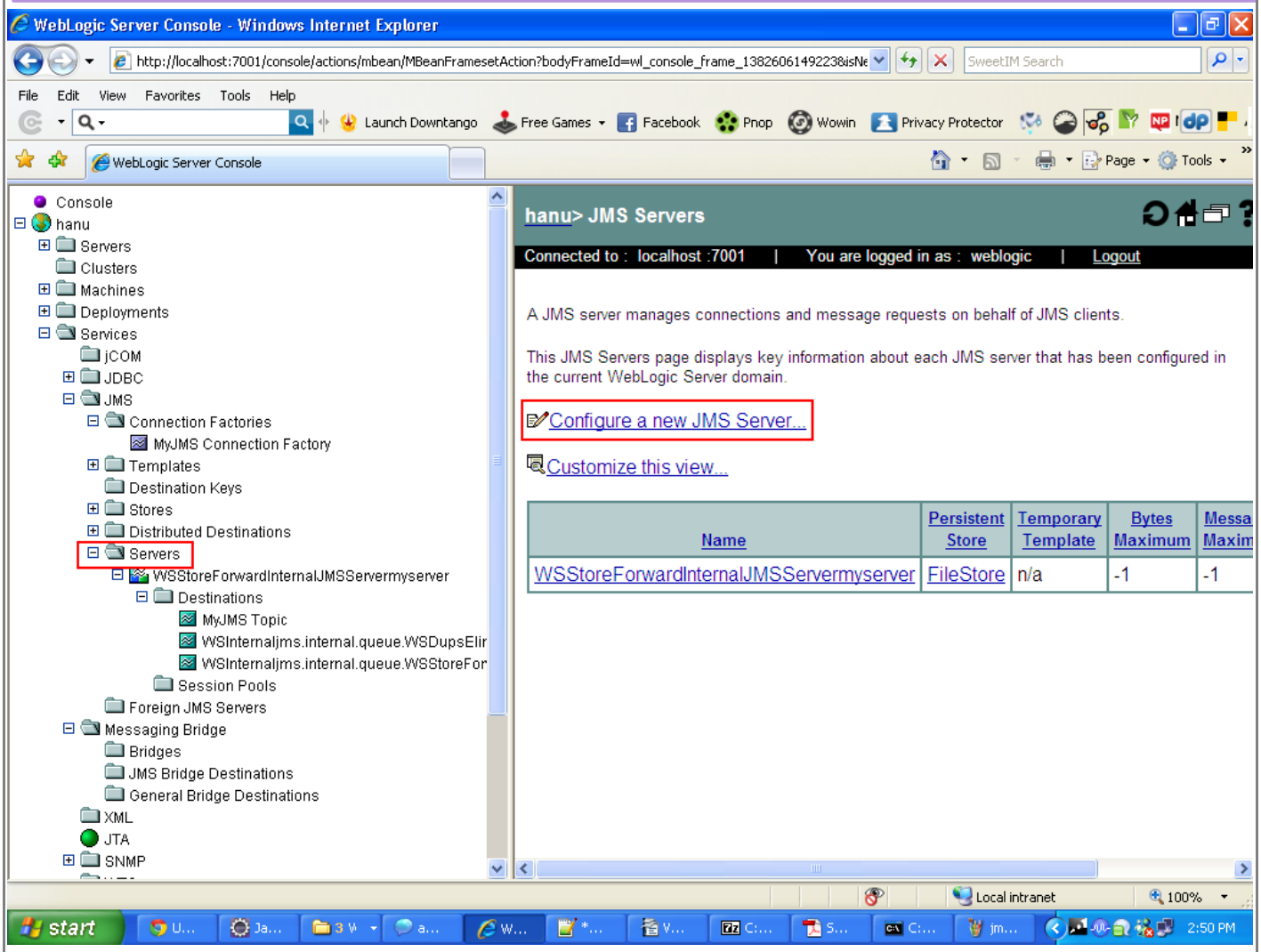
1)



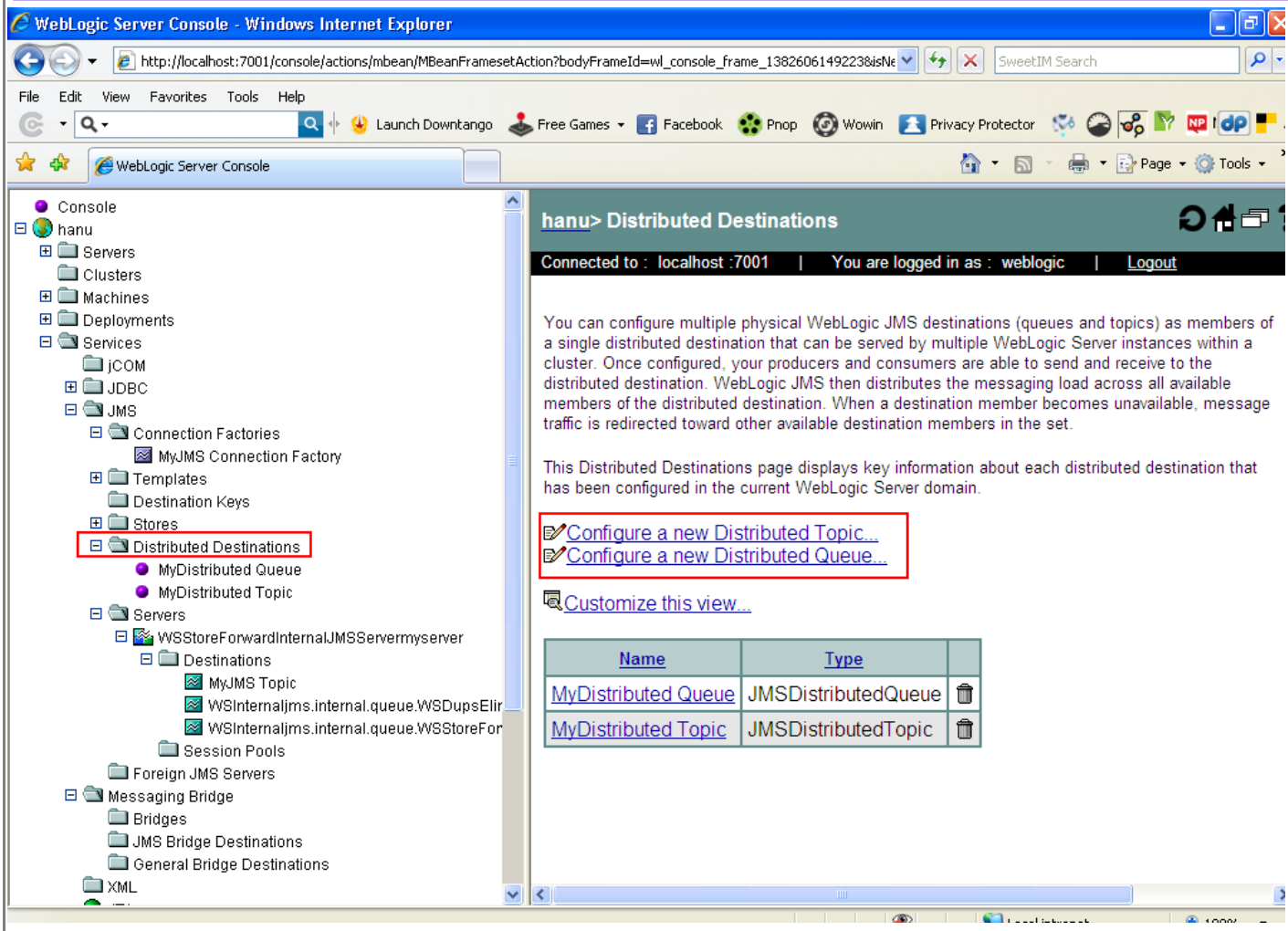
2)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsEliminationH
WSInternaljms.internal.queue.WSStoreForwardQue
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
JTA
SNMP
Connection Factories
localhost :7001 | You are logged in as : weblogic | Logout
ories are objects that enable JMS clients to create JMS connections. A
ory supports concurrent use, enabling multiple threads to access the object
After defining a JMS server, you can configure one or more connection
ate connections with predefined attributes.
ection Factories page displays key information about each JMS connection
been configured in the current WebLogic Server domain.
a new JMS Connection Factory...
this view...
JNDI Name
Client ID
Default Priority
Default Time To Live
Default Redelivery Delay
ConFactJndi
javax.jms.ConnectionFactory;
4
0
0
Local intranet
100%
start
2:47 PM

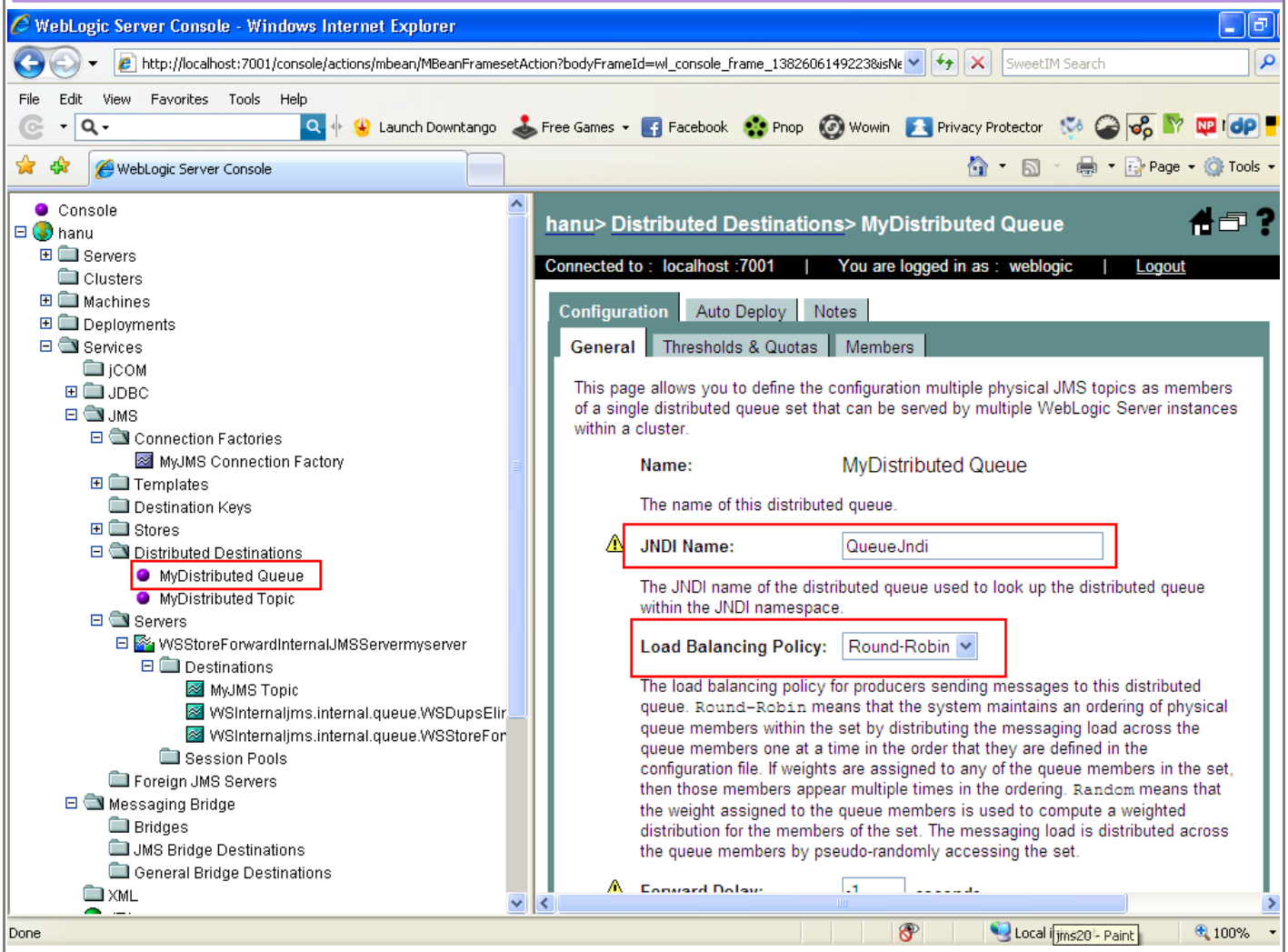
3)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_13826061492238isNe
File Edit View Favorites Tools Help
WebLogic Server Console
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
JTA
SNMP
hanu> JMS Servers
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
A JMS server manages connections and message requests on behalf of JMS clients.
This JMS Servers page displays key information about each JMS server that has been configured in the current WebLogic Server domain.
Configure a new JMS Server...
Customize this view...
Name
Persistent Store
Temporary Template
Bytes Maximum
WSStoreForwardInternalJMSServermyserver
FileStore
n/a
-1
-1
Local intranet
100%
start
2:50 PM

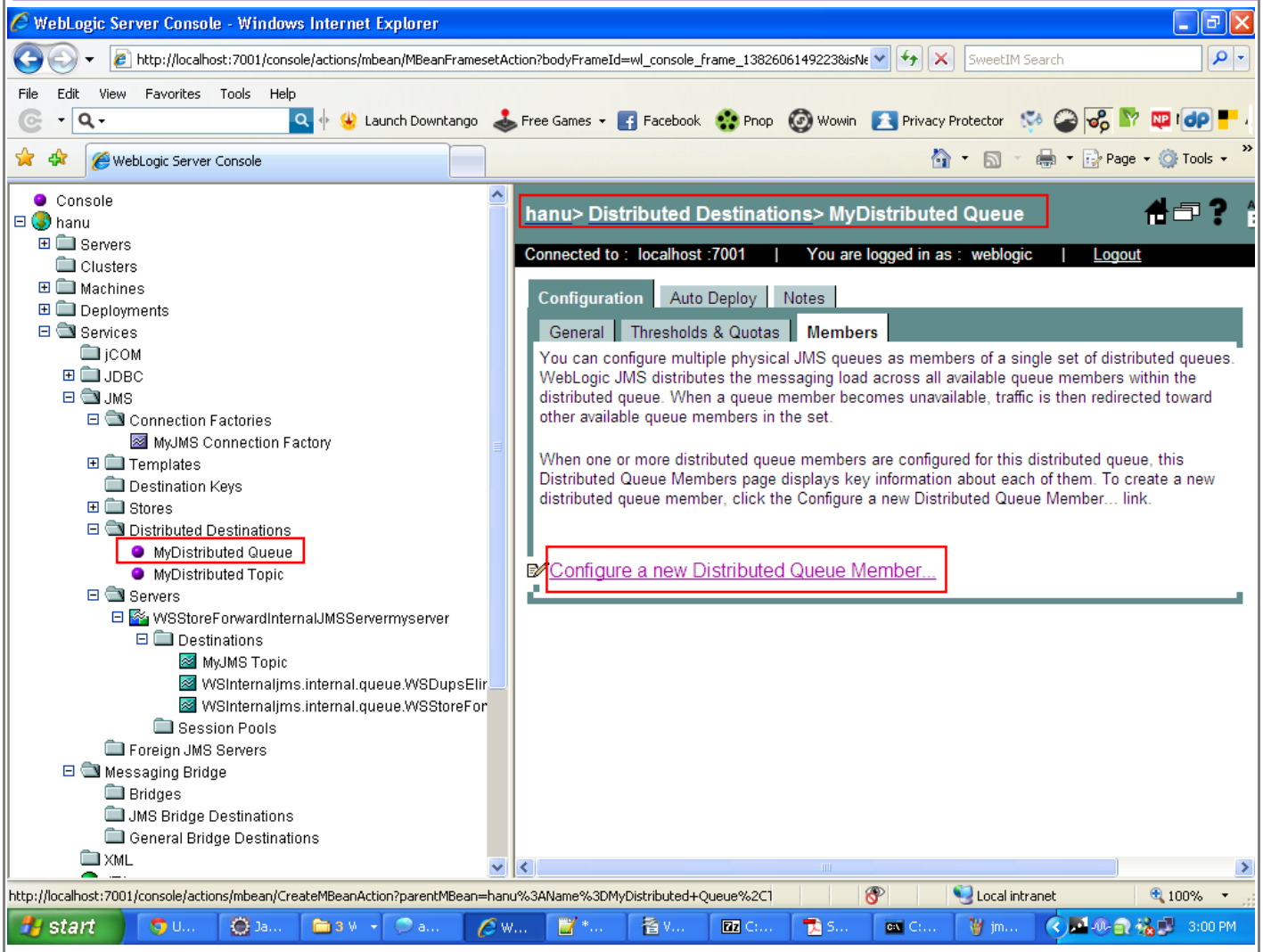
4)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
You can configure multiple physical WebLogic JMS destinations (queues and topics) as members of a single distributed destination that can be served by multiple WebLogic Server instances within a cluster. Once configured, your producers and consumers are able to send and receive to the distributed destination. WebLogic JMS then distributes the messaging load across all available members of the distributed destination. When a destination member becomes unavailable, message traffic is redirected toward other available destination members in the set.
This Distributed Destinations page displays key information about each distributed destination that has been configured in the current WebLogic Server domain.
Configure a new Distributed Topic...
Configure a new Distributed Queue...
Customize this view...
Name
Type
MyDistributed Queue
JMSDistributedQueue
MyDistributed Topic
JMSDistributedTopic

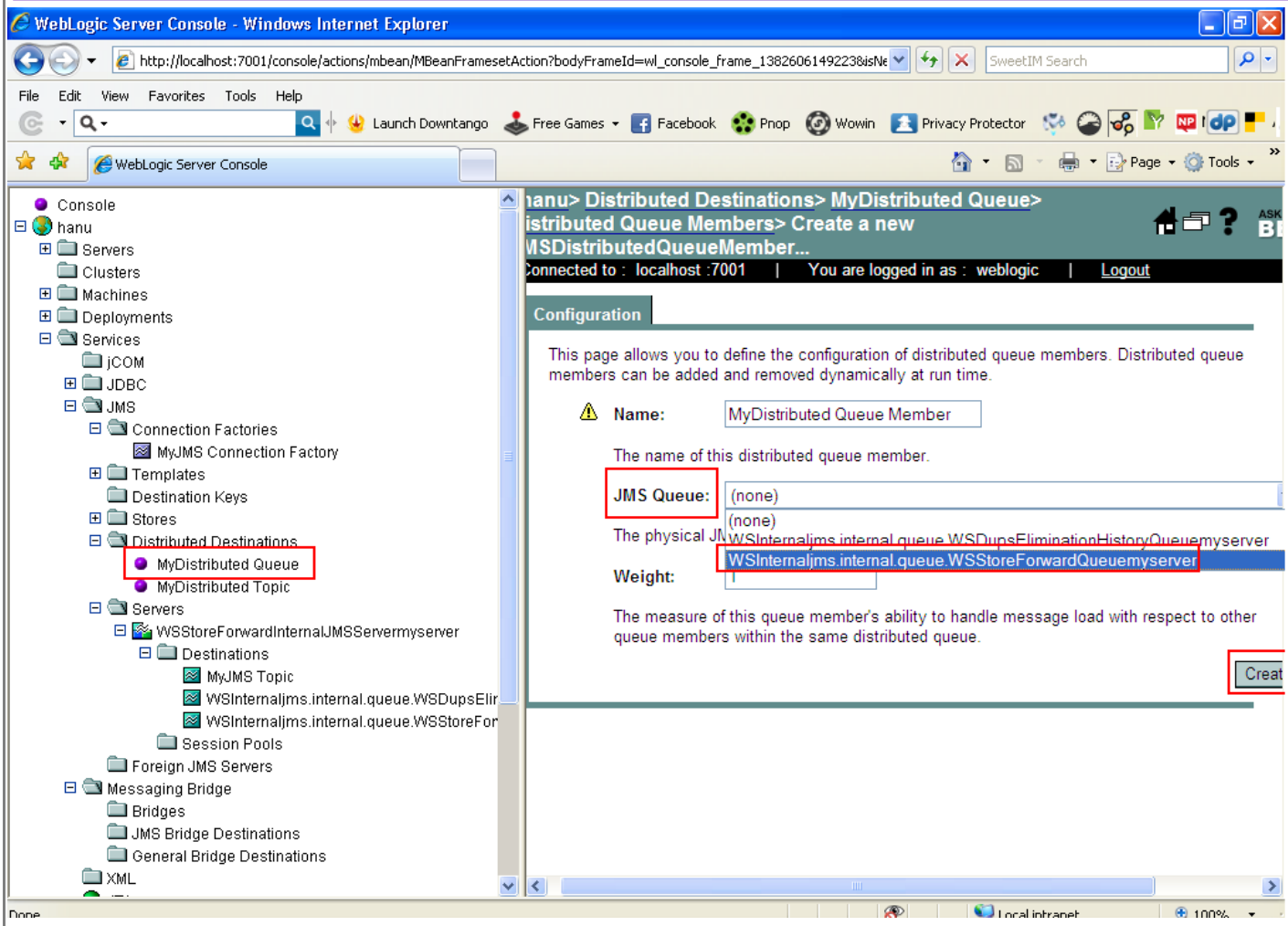
5)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
WebLogic Server Console
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations> MyDistributed Queue
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
Auto Deploy
Notes
General
Thresholds & Quotas
Members
This page allows you to define the configuration multiple physical JMS topics as members of a single distributed queue set that can be served by multiple WebLogic Server instances within a cluster.
Name: MyDistributed Queue
The name of this distributed queue.
JNDI Name: QueueJndi
The JNDI name of the distributed queue used to look up the distributed queue within the JNDI namespace.
Load Balancing Policy: Round-Robin
The load balancing policy for producers sending messages to this distributed queue. Round-Robin means that the system maintains an ordering of physical queue members within the set by distributing the messaging load across the queue members one at a time in the order that they are defined in the configuration file. If weights are assigned to any of the queue members in the set, then those members appear multiple times in the ordering. Random means that the weight assigned to the queue members is used to compute a weighted distribution for the members of the set. The messaging load is distributed across the queue members by pseudo-randomly accessing the set.
Done
100%

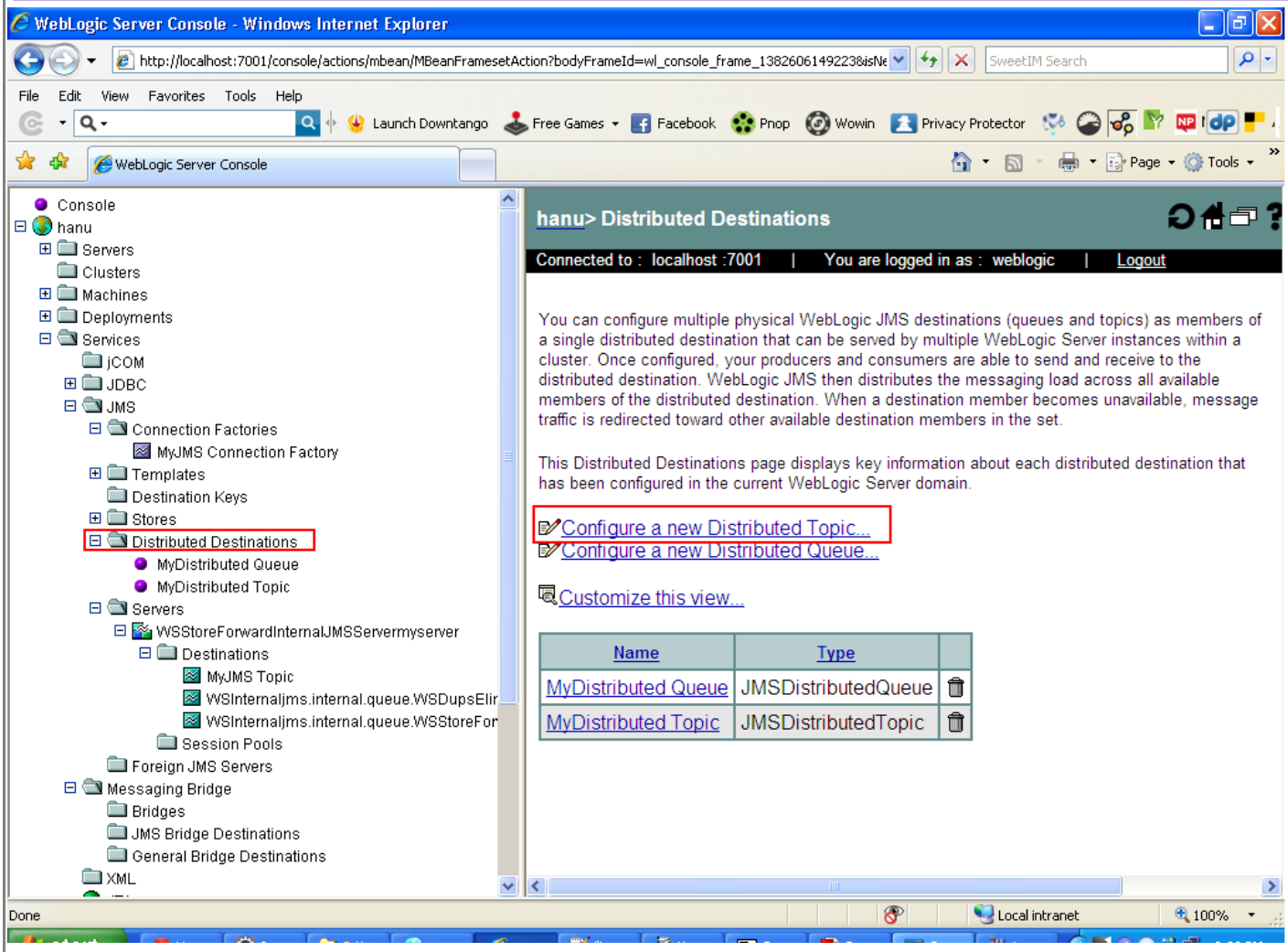
6)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
WebLogic Server Console
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations> MyDistributed Queue
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
Auto Deploy
Notes
General
Thresholds & Quotas
Members
You can configure multiple physical JMS queues as members of a single set of distributed queues. WebLogic JMS distributes the messaging load across all available queue members within the distributed queue. When a queue member becomes unavailable, traffic is then redirected toward other available queue members in the set.
When one or more distributed queue members are configured for this distributed queue, this Distributed Queue Members page displays key information about each of them. To create a new distributed queue member, click the Configure a new Distributed Queue Member... link.
Configure a new Distributed Queue Member...
Local intranet
100%
start
3:00 PM

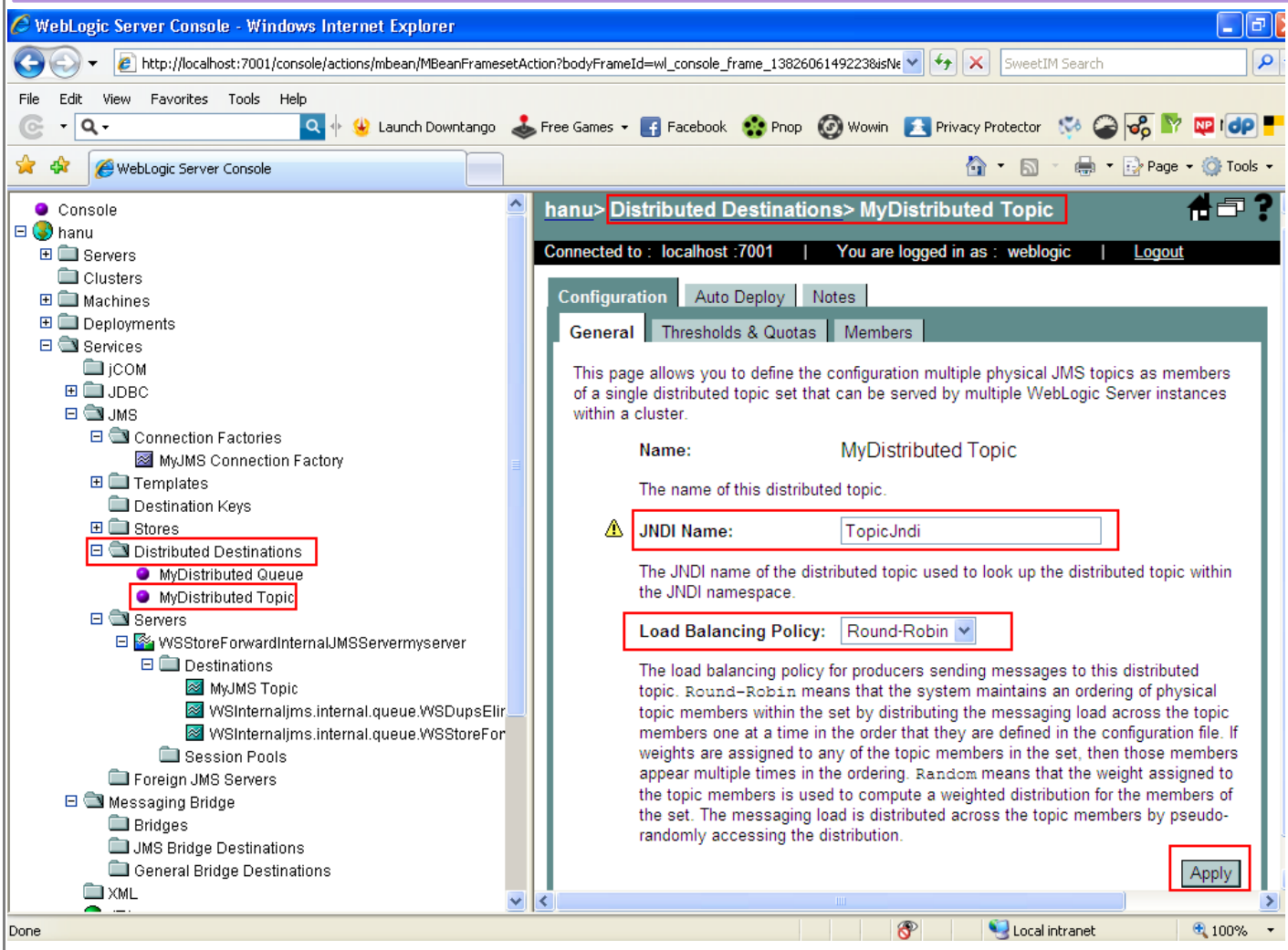
7)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations> MyDistributed Queue>
istributed Queue Members> Create a new
MSDistributedQueueMember...
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
This page allows you to define the configuration of distributed queue members. Distributed queue members can be added and removed dynamically at run time.
Name:
MyDistributed Queue Member
The name of this distributed queue member.
JMS Queue:
(none)
(none)
WSInternaljms.internal.queue.WSStoreForwardQueuemyserver
Weight:
The measure of this queue member's ability to handle message load with respect to other queue members within the same distributed queue.
Creat

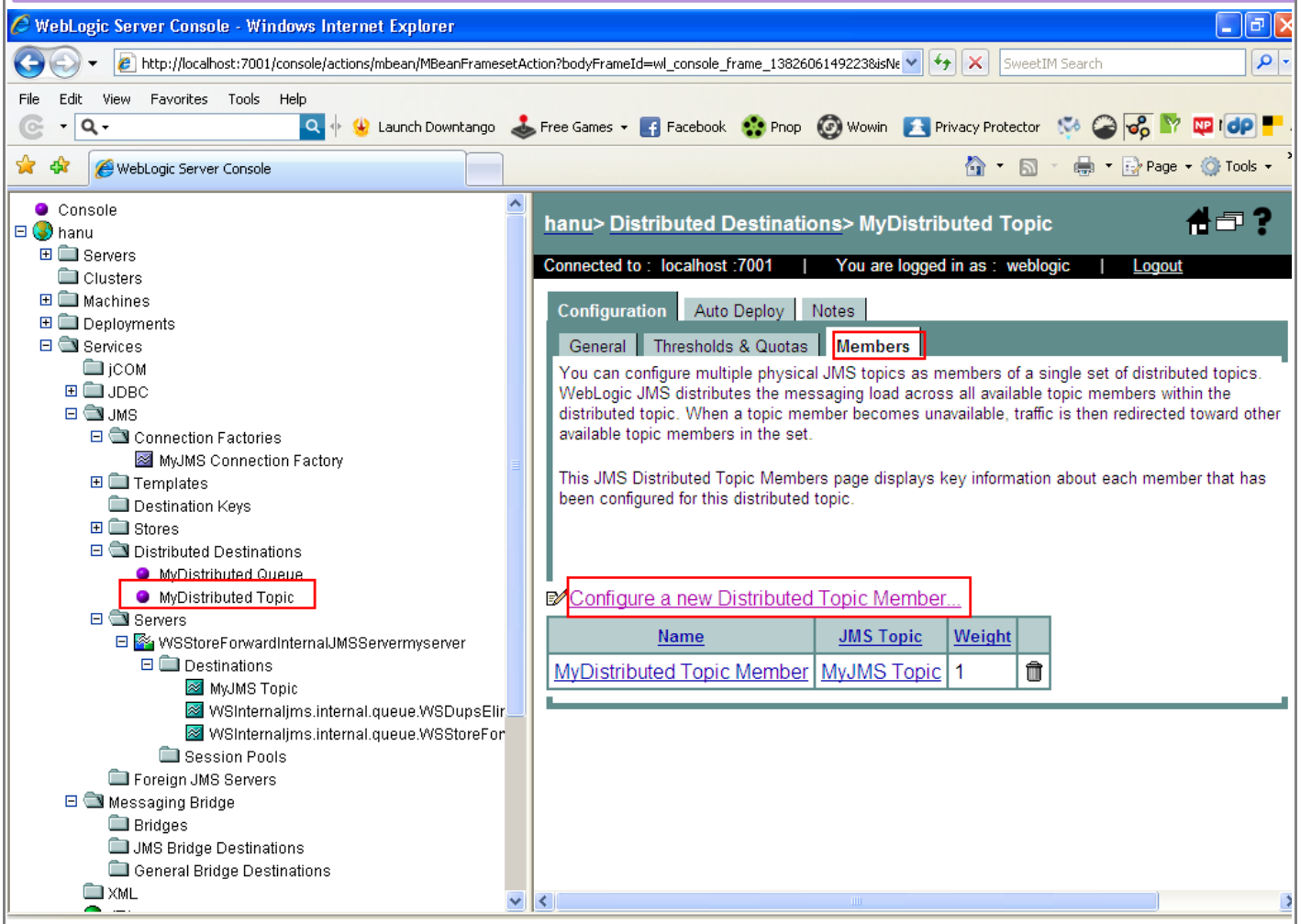
8)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
You can configure multiple physical WebLogic JMS destinations (queues and topics) as members of a single distributed destination that can be served by multiple WebLogic Server instances within a cluster. Once configured, your producers and consumers are able to send and receive to the distributed destination. WebLogic JMS then distributes the messaging load across all available members of the distributed destination. When a destination member becomes unavailable, message traffic is redirected toward other available destination members in the set.
This Distributed Destinations page displays key information about each distributed destination that has been configured in the current WebLogic Server domain.
Configure a new Distributed Topic...
Configure a new Distributed Queue...
Customize this view...
Name
Type
MyDistributed Queue
JMSDistributedQueue
MyDistributed Topic
JMSDistributedTopic
Done
Local intranet
100%

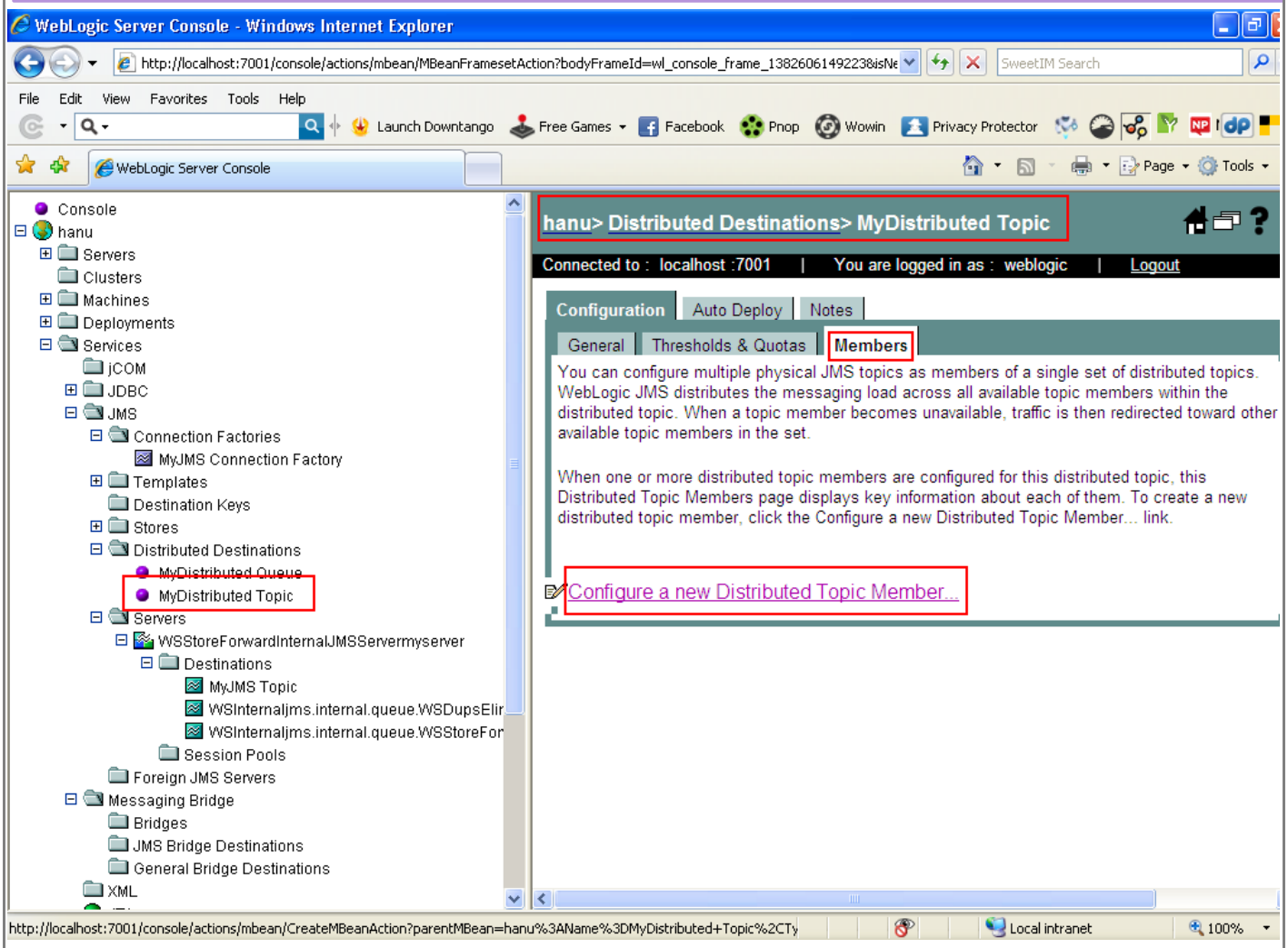
9)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations> MyDistributed Topic
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
Auto Deploy
Notes
General
Thresholds & Quotas
Members
This page allows you to define the configuration multiple physical JMS topics as members of a single distributed topic set that can be served by multiple WebLogic Server instances within a cluster.
Name:
MyDistributed Topic
The name of this distributed topic.
JNDI Name:
TopicJndi
The JNDI name of the distributed topic used to look up the distributed topic within the JNDI namespace.
Load Balancing Policy:
Round-Robin
The load balancing policy for producers sending messages to this distributed topic. Round-Robin means that the system maintains an ordering of physical topic members within the set by distributing the messaging load across the topic members one at a time in the order that they are defined in the configuration file. If weights are assigned to any of the topic members in the set, then those members appear multiple times in the ordering. Random means that the weight assigned to the topic members is used to compute a weighted distribution for the members of the set. The messaging load is distributed across the topic members by pseudo-randomly accessing the distribution.
Apply
Done
Local intranet
100%

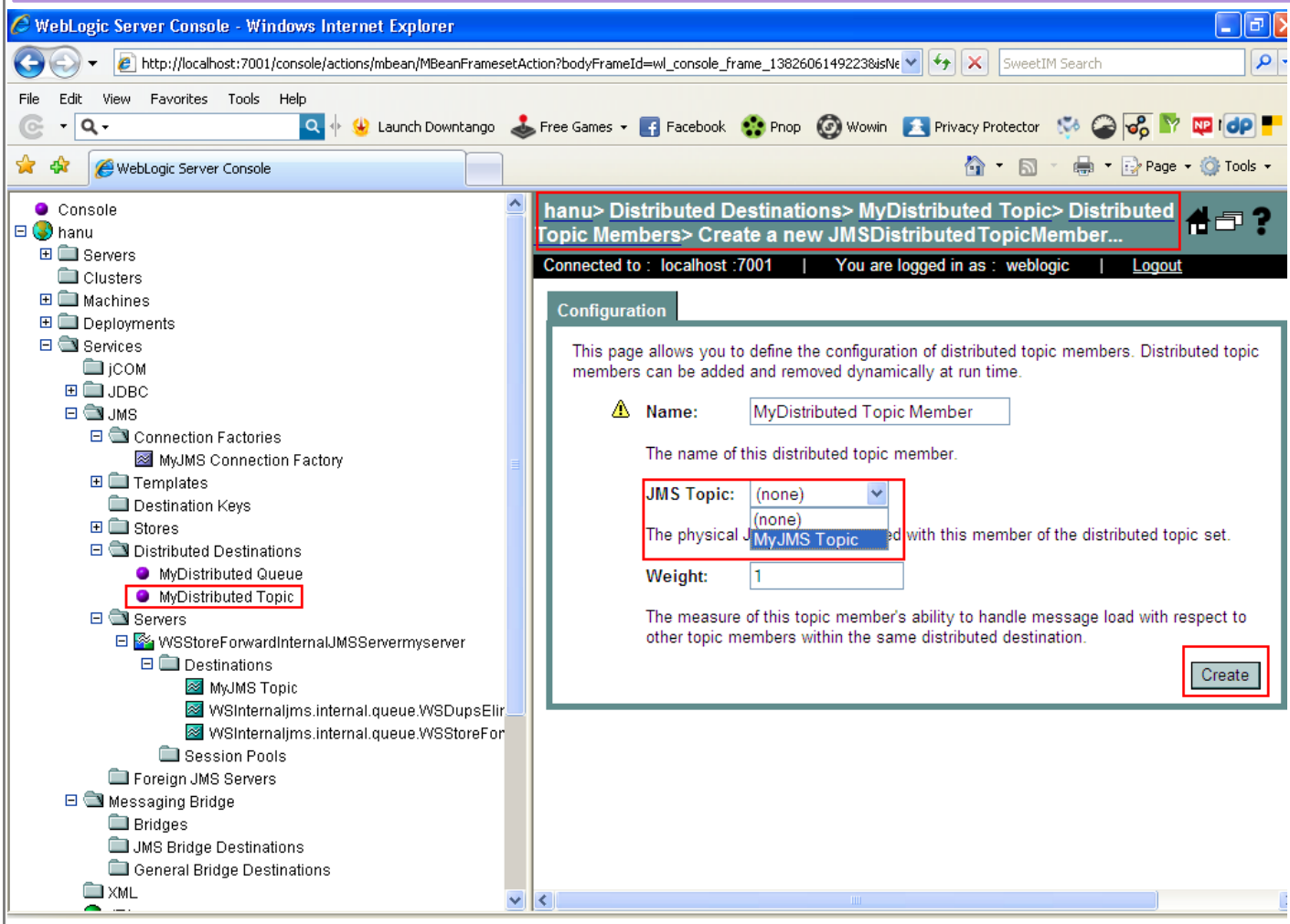
10)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations> MyDistributed Topic
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
Auto Deploy
Notes
General
Thresholds & Quotas
Members
You can configure multiple physical JMS topics as members of a single set of distributed topics. WebLogic JMS distributes the messaging load across all available topic members within the distributed topic. When a topic member becomes unavailable, traffic is then redirected toward other available topic members in the set.
This JMS Distributed Topic Members page displays key information about each member that has been configured for this distributed topic.
Configure a new Distributed Topic Member...
Name
JMS Topic
Weight
MyDistributed Topic Member
MyJMS Topic
1

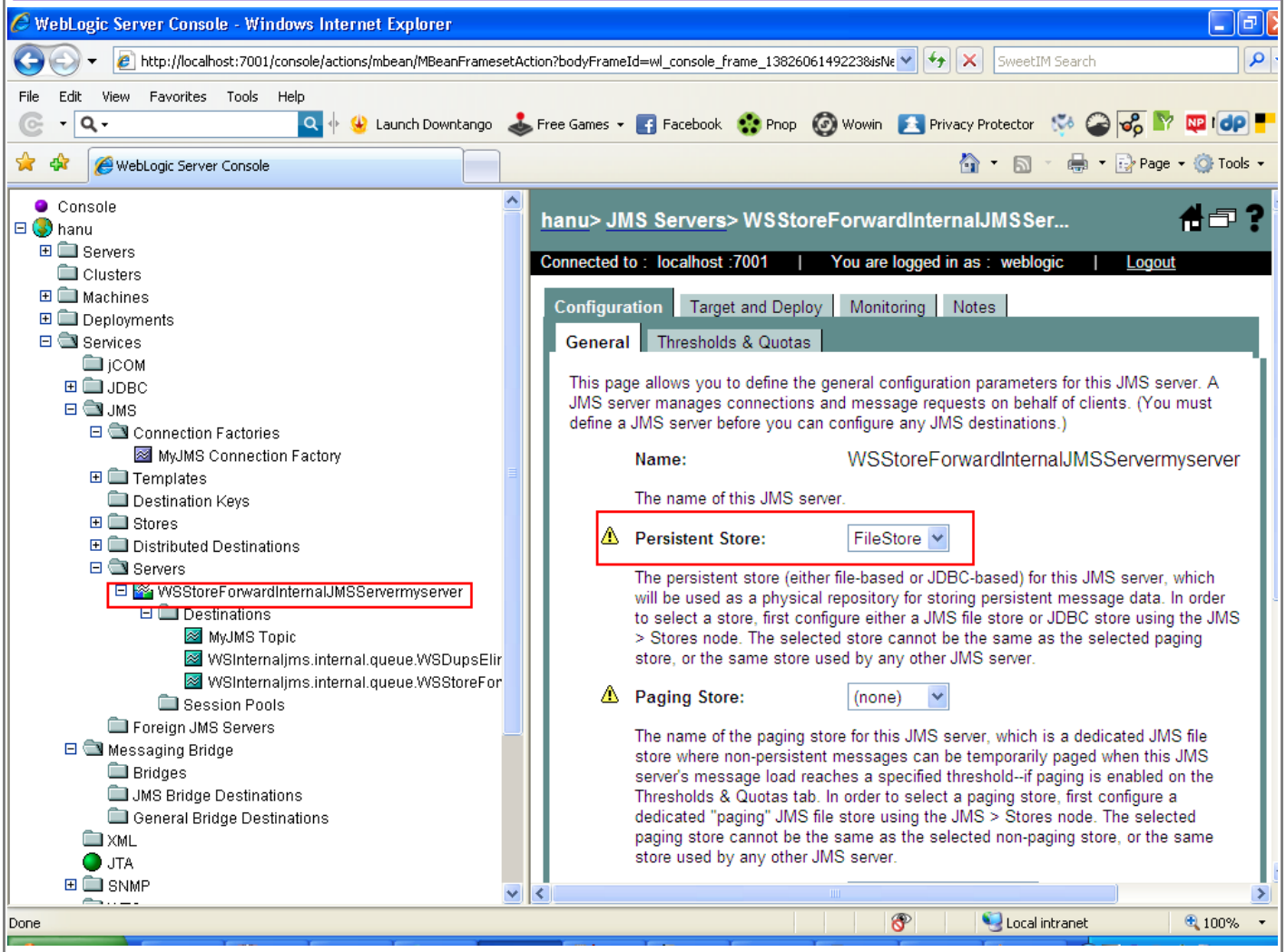
11)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations> MyDistributed Topic
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
Auto Deploy
Notes
General
Thresholds & Quotas
Members
You can configure multiple physical JMS topics as members of a single set of distributed topics. WebLogic JMS distributes the messaging load across all available topic members within the distributed topic. When a topic member becomes unavailable, traffic is then redirected toward other available topic members in the set.
When one or more distributed topic members are configured for this distributed topic, this Distributed Topic Members page displays key information about each of them. To create a new distributed topic member, click the Configure a new Distributed Topic Member... link.
Configure a new Distributed Topic Member...
http://localhost:7001/console/actions/mbean/CreateMBeanAction?parentMBean=hanu%3AName%3DMyDistributed+Topic%2CTy
Local intranet
100%

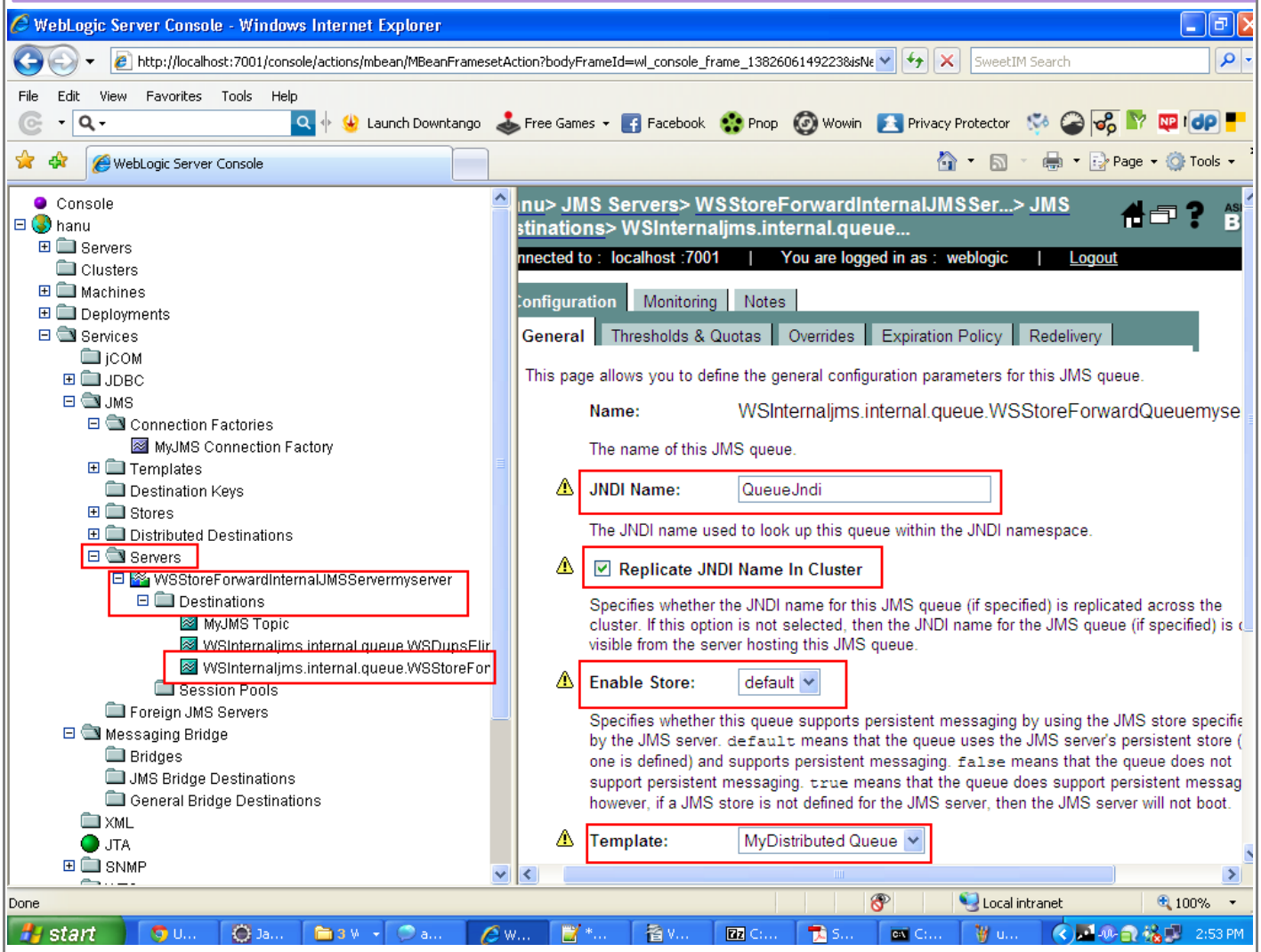
12)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
hanu> Distributed Destinations> MyDistributed Topic> Distributed Topic Members> Create a new JMSDistributedTopicMember...
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
This page allows you to define the configuration of distributed topic members. Distributed topic members can be added and removed dynamically at run time.
Name:
MyDistributed Topic Member
The name of this distributed topic member.
JMS Topic:
(none)
(none)
MyJMS Topic
with this member of the distributed topic set.
Weight:
1
The measure of this topic member's ability to handle message load with respect to other topic members within the same distributed destination.
Create

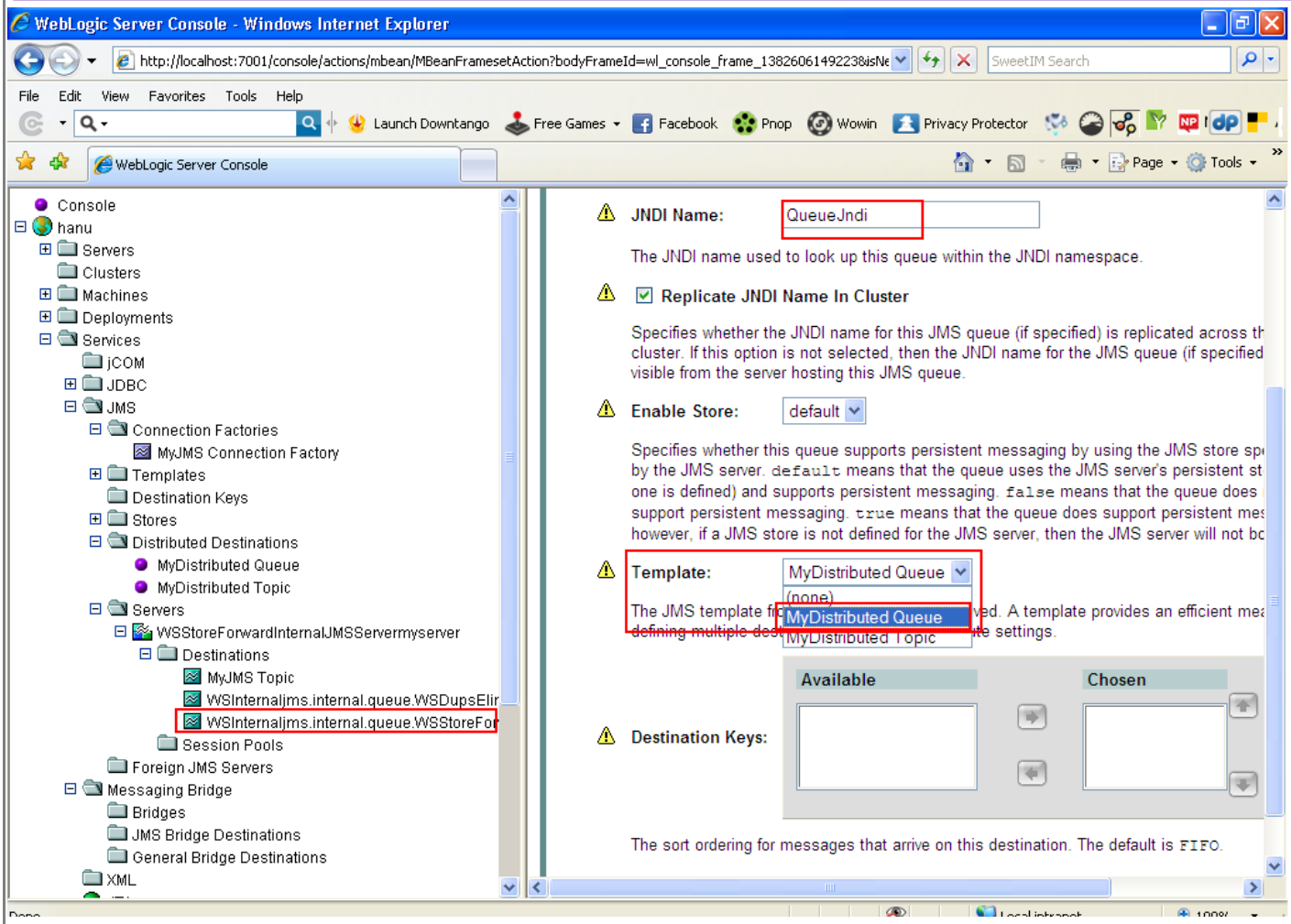
13)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
JTA
SNMP
hanu> JMS Servers> WSStoreForwardInternalJMSSer...
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
Target and Deploy
Monitoring
Notes
General
Thresholds & Quotas
This page allows you to define the general configuration parameters for this JMS server. A JMS server manages connections and message requests on behalf of clients. (You must define a JMS server before you can configure any JMS destinations.)
Name:
WSStoreForwardInternalJMSServermyserver
The name of this JMS server.
Persistent Store:
FileStore
The persistent store (either file-based or JDBC-based) for this JMS server, which will be used as a physical repository for storing persistent message data. In order to select a store, first configure either a JMS file store or JDBC store using the JMS > Stores node. The selected store cannot be the same as the selected paging store, or the same store used by any other JMS server.
Paging Store:
(none)
The name of the paging store for this JMS server, which is a dedicated JMS file store where non-persistent messages can be temporarily paged when this JMS server's message load reaches a specified threshold--if paging is enabled on the Thresholds & Quotas tab. In order to select a paging store, first configure a dedicated "paging" JMS file store using the JMS > Stores node. The selected paging store cannot be the same as the selected non-paging store, or the same store used by any other JMS server.
Done
Local intranet
100%

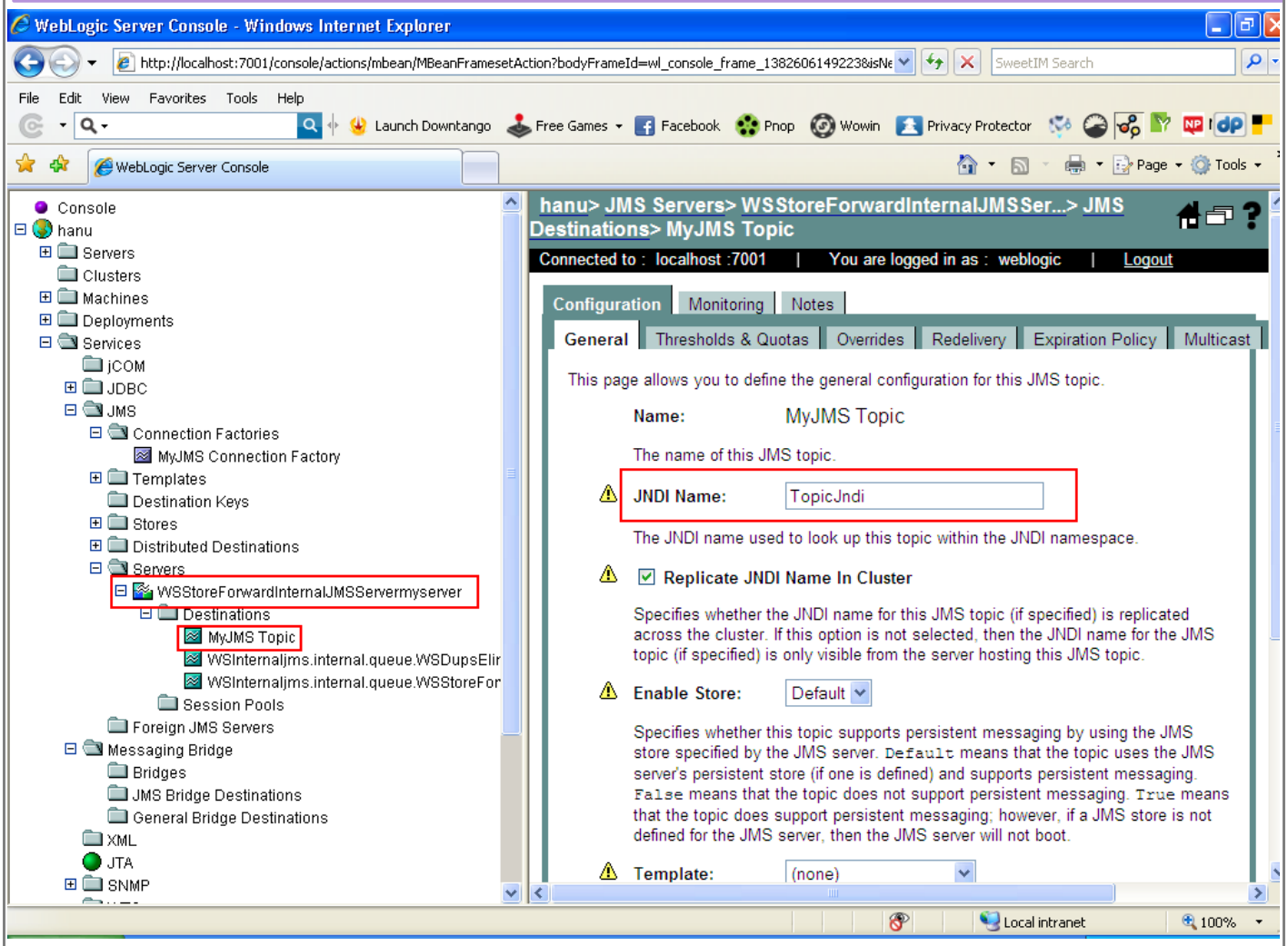
14)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
WebLogic Server Console
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
JTA
SNMP
nu> JMS Servers> WSStoreForwardInternalJMSSer...> JMS
stinations> WSInternaljms.internal.queue...
nnected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration Monitoring Notes
General Thresholds & Quotas Overrides Expiration Policy Redelivery
This page allows you to define the general configuration parameters for this JMS queue.
Name: WSInternaljms.internal.queue.WSStoreForwardQueuemyse
The name of this JMS queue.
JNDI Name: QueueJndi
The JNDI name used to look up this queue within the JNDI namespace.
Replicate JNDI Name In Cluster
Specifies whether the JNDI name for this JMS queue (if specified) is replicated across the cluster. If this option is not selected, then the JNDI name for the JMS queue (if specified) is
visible from the server hosting this JMS queue.
Enable Store: default
Specifies whether this queue supports persistent messaging by using the JMS store specifie
by the JMS server. default means that the queue uses the JMS server's persistent store (
one is defined) and supports persistent messaging. false means that the queue does not
support persistent messaging. true means that the queue does support persistent messag
however, if a JMS store is not defined for the JMS server, then the JMS server will not boot.
Template: MyDistributed Queue
Done
Local intranet
100%
start
2:53 PM

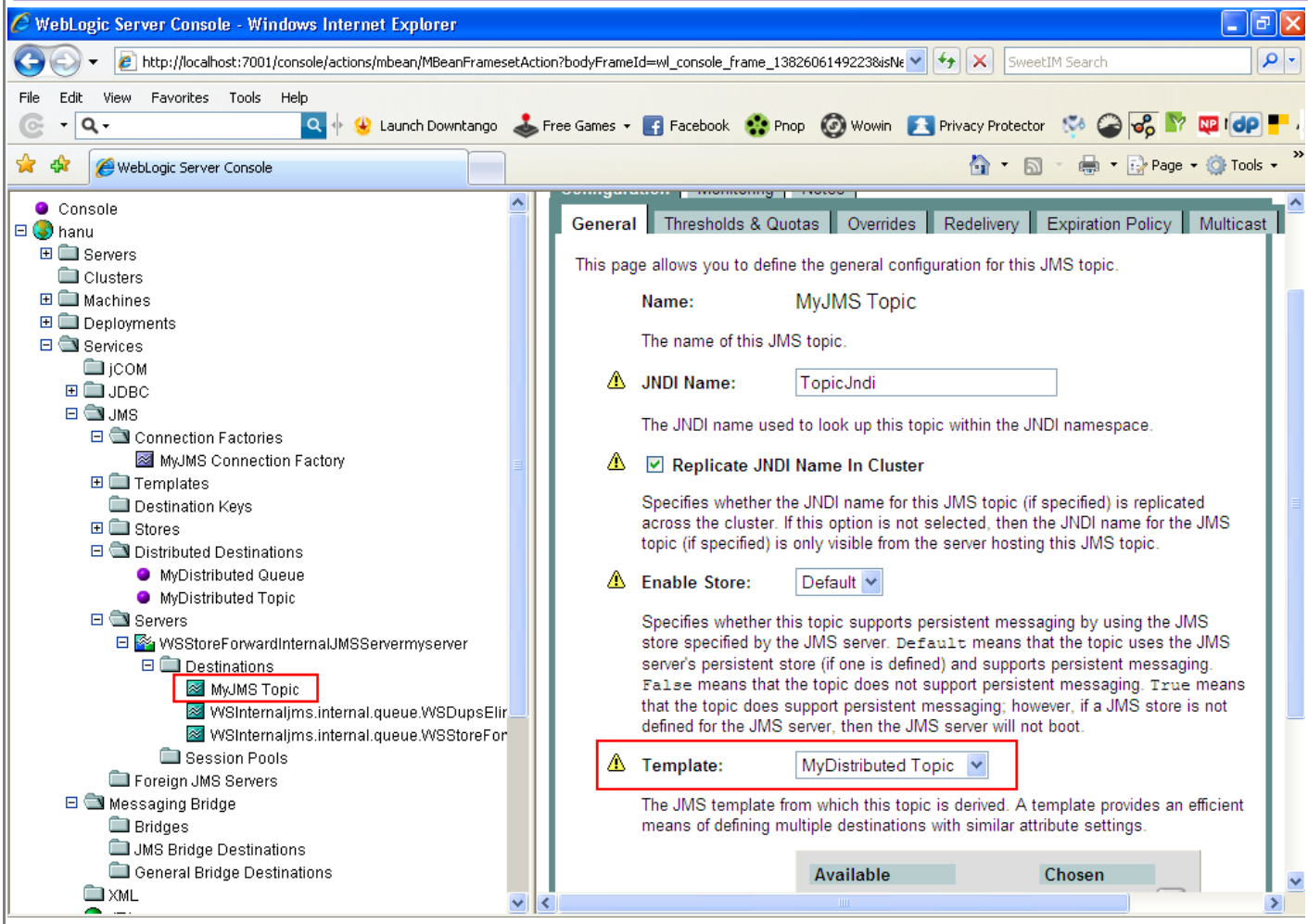
15)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
MyDistributed Queue
MyDistributed Topic
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
JNDI Name:
QueueJndi
The JNDI name used to look up this queue within the JNDI namespace.
Replicate JNDI Name In Cluster
Specifies whether the JNDI name for this JMS queue (if specified) is replicated across th
cluster. If this option is not selected, then the JNDI name for the JMS queue (if specified
visible from the server hosting this JMS queue.
Enable Store:
default
Specifies whether this queue supports persistent messaging by using the JMS store sp
by the JMS server. default means that the queue uses the JMS server's persistent st
one is defined) and supports persistent messaging. false means that the queue does
support persistent messaging. true means that the queue does support persistent mes
however, if a JMS store is not defined for the JMS server, then the JMS server will not bc
Template:
MyDistributed Queue
(none)
MyDistributed Queue
MyDistributed Topic
Available
Chosen
Destination Keys:
The sort ordering for messages that arrive on this destination. The default is FIFO.

16)


WebLogic Server Console - Windows Internet Explorer
http://localhost:7001/console/actions/mbean/MBeanFramesetAction?bodyFrameId=wl_console_frame_1382606149223&isNe
File Edit View Favorites Tools Help
Launch Downtango
Free Games
Facebook
Pnop
Wowin
Privacy Protector
WebLogic Server Console
Page
Tools
Console
hanu
Servers
Clusters
Machines
Deployments
Services
jCOM
JDBC
JMS
Connection Factories
MyJMS Connection Factory
Templates
Destination Keys
Stores
Distributed Destinations
Servers
WSStoreForwardInternalJMSServermyserver
Destinations
MyJMS Topic
WSInternaljms.internal.queue.WSDupsElir
WSInternaljms.internal.queue.WSStoreFor
Session Pools
Foreign JMS Servers
Messaging Bridge
Bridges
JMS Bridge Destinations
General Bridge Destinations
XML
JTA
SNMP
hanu> JMS Servers> WSStoreForwardInternalJMSSer...> JMS Destinations> MyJMS Topic
Connected to : localhost :7001 | You are logged in as : weblogic | Logout
Configuration
Monitoring
Notes
General
Thresholds & Quotas
Overrides
Redelivery
Expiration Policy
Multicast
This page allows you to define the general configuration for this JMS topic.
Name: MyJMS Topic
The name of this JMS topic.
JNDI Name: TopicJndi
The JNDI name used to look up this topic within the JNDI namespace.
Replicate JNDI Name In Cluster
Specifies whether the JNDI name for this JMS topic (if specified) is replicated across the cluster. If this option is not selected, then the JNDI name for the JMS topic (if specified) is only visible from the server hosting this JMS topic.
Enable Store: Default
Specifies whether this topic supports persistent messaging by using the JMS store specified by the JMS server. Default means that the topic uses the JMS server's persistent store (if one is defined) and supports persistent messaging. False means that the topic does not support persistent messaging. True means that the topic does support persistent messaging; however, if a JMS store is not defined for the JMS server, then the JMS server will not boot.
Template: (none)
Local intranet
100%

17)



P2P related Program:

MyQueueSender.java

```
import javax.jms.*;
import javax.naming.*;

import java.util.Properties;

public class MyQueueSender {
      public static void main(String[] args) throws Exception {
            Properties prop = new Properties();
            prop.put(Context.INITIAL_CONTEXT_FACTORY,
                        "weblogic.jndi.WLInitialContextFactory");
            prop.put(Context.PROVIDER_URL, "t3://localhost:7001");
            // InitialContext ic = new InitialContext(prop);
            Context ctx = new InitialContext(prop);
```

```
            QueueConnectionFactory qcf = (QueueConnectionFactory) ctx
                        .lookup("ConFactJndi");
            QueueConnection qconn = qcf.createQueueConnection();
            QueueSession qsession = qconn.createQueueSession(false,
                        Session.AUTO_ACKNOWLEDGE);
            Queue queue = (Queue) ctx.lookup("QueueJndi");
            QueueSender qsender = qsession.createSender(queue);
            TextMessage message;
            String text;
            int k = 1;
            for (int i = 1; i <= 4; i++) {
                  if (i == 1)
                        text = "Hotmail";
                  else if (i == 2)
                        text = "Amazon";
                  else if (i == 3)
                        text = "Gmail";
                  else
                        text = "Yahoo";
                  for (int j = 1; j <= 3; j++) {
                        message = qsession.createTextMessage();
                        message.setStringProperty("Name", text);
                        message.setText("This is Item " + k + "......" + text);
                        System.out.println("Sending " + k + "\t" + text);
                        qsender.send(message);
                        k++;
                  }
            }
            qsession.close();
            qconn.close();
      }
}
/*
 * Output: Sending 1 Hotmail Sending 2 Hotmail Sending 3 Hotmail Sending 4
 * Amazon Sending 5 Amazon Sending 6 Amazon Sending 7 Gmail Sending 8 Gmail
 * Sending 9 Gmail Sending 10 Yahoo Sending 11 Yahoo Sending 12 Yahoo
 */
```

MyQueueReceiver.java

```
import javax.jms.*;
import javax.naming.*;

import java.util.Properties;

public class MyQueueReceiver implements MessageListener {
      public static void main(String[] args) throws Exception {
            Properties prop = new Properties();
            prop.put(Context.INITIAL_CONTEXT_FACTORY,
                        "weblogic.jndi.WLInitialContextFactory");
            prop.put(Context.PROVIDER_URL, "t3://localhost:7001");
```

```
            Context ctx = new InitialContext(prop);
            QueueConnectionFactory qcf = (QueueConnectionFactory) ctx
                        .lookup("ConFactJndi");
            System.out.println(qcf);
            QueueConnection qconn = qcf.createQueueConnection();
            QueueSession qsession = qconn.createQueueSession(false,
                        Session.AUTO_ACKNOWLEDGE);
            Queue queue = (Queue) ctx.lookup("QueueJndi");
            System.out.println(queue);

            String selector = "Name In('Gmail','Hotmail')";
            QueueReceiver qreceiver = qsession.createReceiver(queue, selector);
            MyQueueReceiver mq = new MyQueueReceiver();
            qreceiver.setMessageListener(mq);
            qconn.start();
      }

      public void onMessage(Message msg) {
            try {
                  TextMessage tmsg = (TextMessage) msg;
                  System.out.println("\n\n\t\tReceived message is:" + tmsg.getText());
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }

}
/*
 * Output: weblogic.jms.client.JMSXAConnectionFactory@1dfc547
 * WSInternaljms.internal.queue.WSStoreForwardQueuemyserver Received message
 * is:This is Item 2......Hotmail
 */
```

Publish/Subcribe Program:

TopicDemo.java

```
import javax.jms.*;
import javax.naming.*;

import java.util.Properties;

public class TopicDemo {
      public static void main(String[] args) throws Exception {
            Properties prop = new Properties();
            prop.put(Context.INITIAL_CONTEXT_FACTORY,
                        "weblogic.jndi.WLInitialContextFactory");
            prop.put(Context.PROVIDER_URL, "t3://localhost:7001");

            Context ctx = new InitialContext(prop);
            TopicDemo td = new TopicDemo();
            td.doPub(ctx);
            ctx.close();
```

```
	}

	public void doPub(Context ctx) throws JMSException, NamingException {
		Topic topic = (Topic) ctx.lookup("TopicJndi");
		System.out.println(topic);

		TopicConnectionFactory tcf = (TopicConnectionFactory) ctx
				.lookup("ConFactJndi");
		System.out.println(tcf);
		TopicConnection tc = tcf.createTopicConnection();
		TopicSession ts = tc.createTopicSession(false,
				TopicSession.AUTO_ACKNOWLEDGE);

		TopicPublisher tp = ts.createPublisher(topic);

		TextMessage msg;
		msg = ts.createTextMessage("This is from GURU.....!");
		System.out.println("Publishing message on the Destination(Topic)");
		tp.publish(msg);
		tc.close();
	}

}
```

SubscriberDemo.java

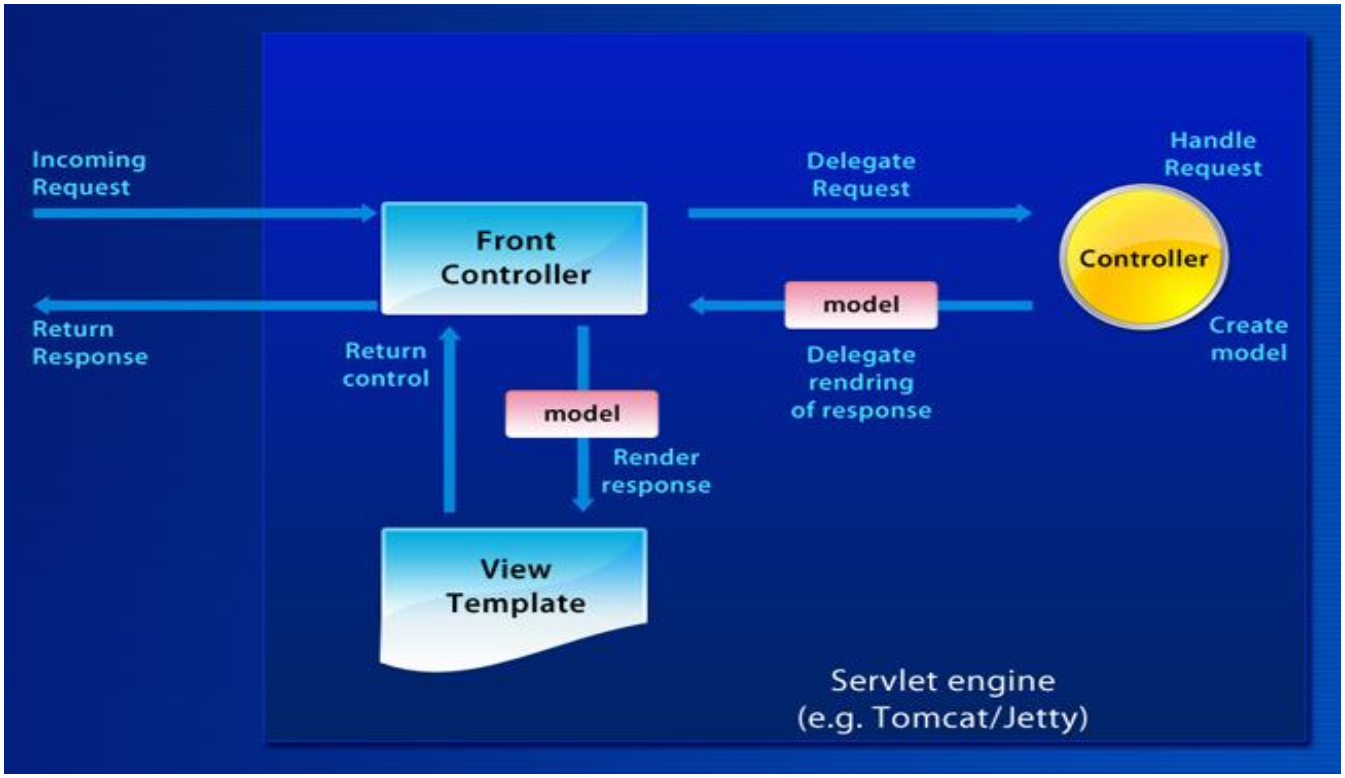
```
import javax.jms.*;
import javax.naming.*;

import java.util.Properties;

public class SubscriberDemo {
	public static void main(String[] args) throws Exception {
		Properties prop = new Properties();
		prop.put(Context.INITIAL_CONTEXT_FACTORY,
				"weblogic.jndi.WLInitialContextFactory");
		prop.put(Context.PROVIDER_URL, "t3://localhost:7001");

		Context ctx = new InitialContext(prop);
		SubscriberDemo td = new SubscriberDemo();
		td.doSub(ctx);
		ctx.close();
	}

	public void doSub(Context ctx) throws JMSException, NamingException {
		Topic topic = (Topic) ctx.lookup("TopicJndi");
		System.out.println(topic);

		TopicConnectionFactory tcf = (TopicConnectionFactory) ctx
				.lookup("ConFactJndi");
		System.out.println(tcf);
		TopicConnection tc = tcf.createTopicConnection();
```

```
            TopicSession ts = tc.createTopicSession(false,
                        TopicSession.AUTO_ACKNOWLEDGE);

            TopicSubscriber tsub = ts.createSubscriber(topic);
            tc.start();

            TextMessage msg;
            msg = (TextMessage) tsub.receive();
            System.out.println("Received message is :..." + msg.getText());
            tc.close();
      }

}
```

==========

Spring MVC Module

==========



Following events happens when DispatcherServlet receives are request:

1.The DispatcherServlet configured in web.xml file receives the request.

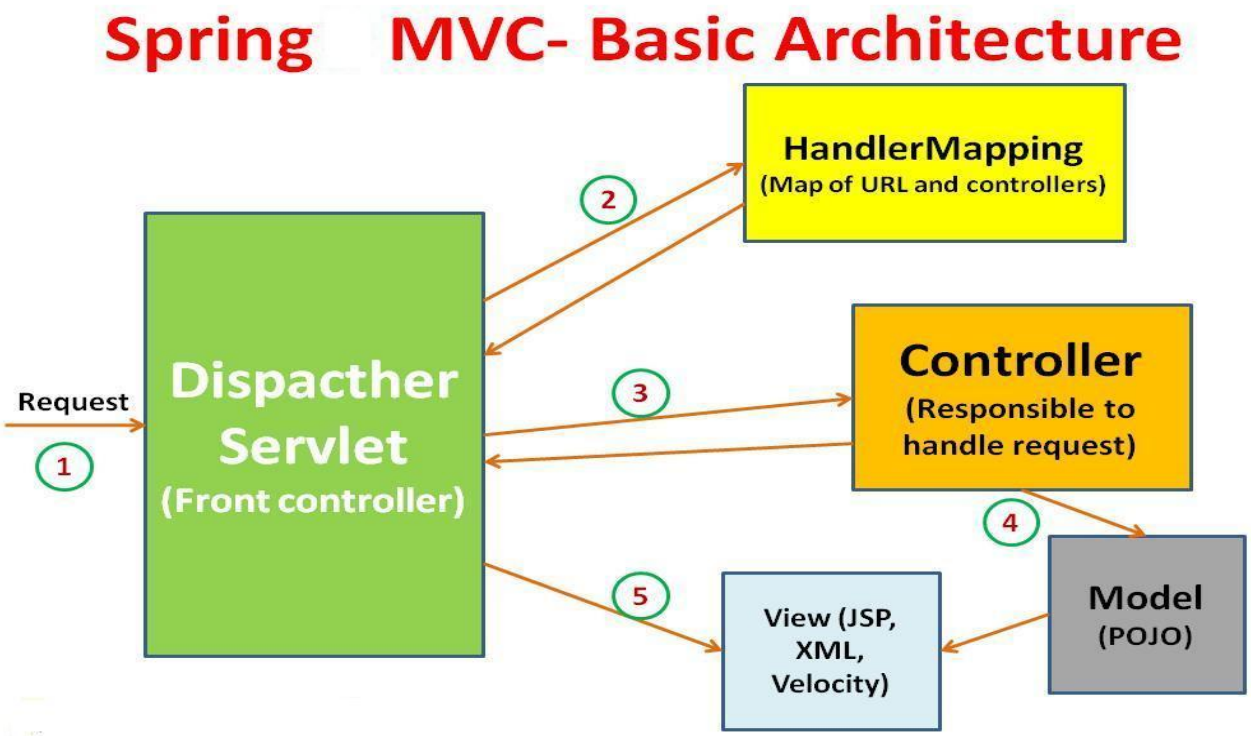
2.The DispatcherServlet finds the appropriate Controller with the help of HandlerMapping and then invokes associated Controller.

3.Then the Controller executes the logic business logic (if written by the programmer) and then returns ModeAndView object to the DispatcherServlet.

4.The DispatcherServlet determines the view from the ModelAndView object.

5.Then the DispatcherServlet passes the model object to the View.

6.The View is rendered and the Dispatcher Servlet sends the output to the Servlet container. Finally Servlet Container sends the result back to the user.



Front Controller has a very important role to play in the **MVC Frameworks** work flow. The front controller component in a MVC framework is responsible to capture the requests landing at the application and route them into the control of the MVC Framework. In **Spring 3 MVC** the **DispatcherServlet acts as the Front Controller** and is responsible to caputure the request and route it into the frameworks control.

Apart from capturing the requests, the **DispatcherServlet** is also responsible to initialize the frameworks components which are used to process the request at various stages.

Request handling steps in Spring MVC

1.Client access some URL on the server.

2.The Spring Front Controller (DispatcherServlet) intercepts the Request. After receiving the request it finds the appropriate Handler Mappings.

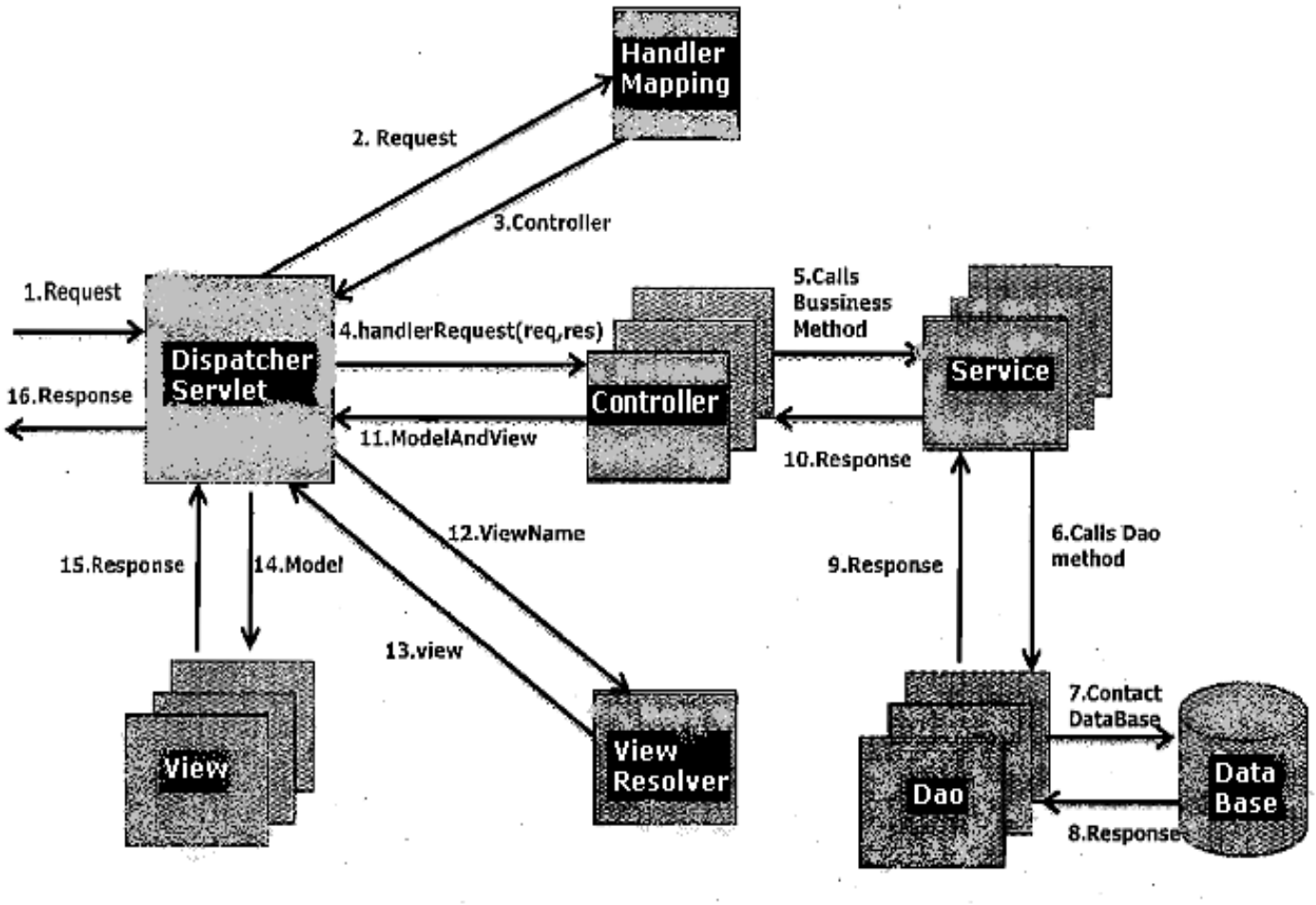
3.The Handle Mappings maps the client request to appropriate Controller. In this process framework reads the configuration information from the configuration file or from the annotated controller list.Then DispatcherServlet dispatch the request to the appropriate Controller. The

Handler Adapters involves in this process.

4.Then the Controller processes the Client Request, it executes the logic defined in the Controller method and finally returns the ModelAndView object back to the Front Controller.

5.Based on the values in the ModelAndView Controller resolves the actual view, which can be JSP, Velocity, FreeMaker, Jasper or any other configured view resolver.

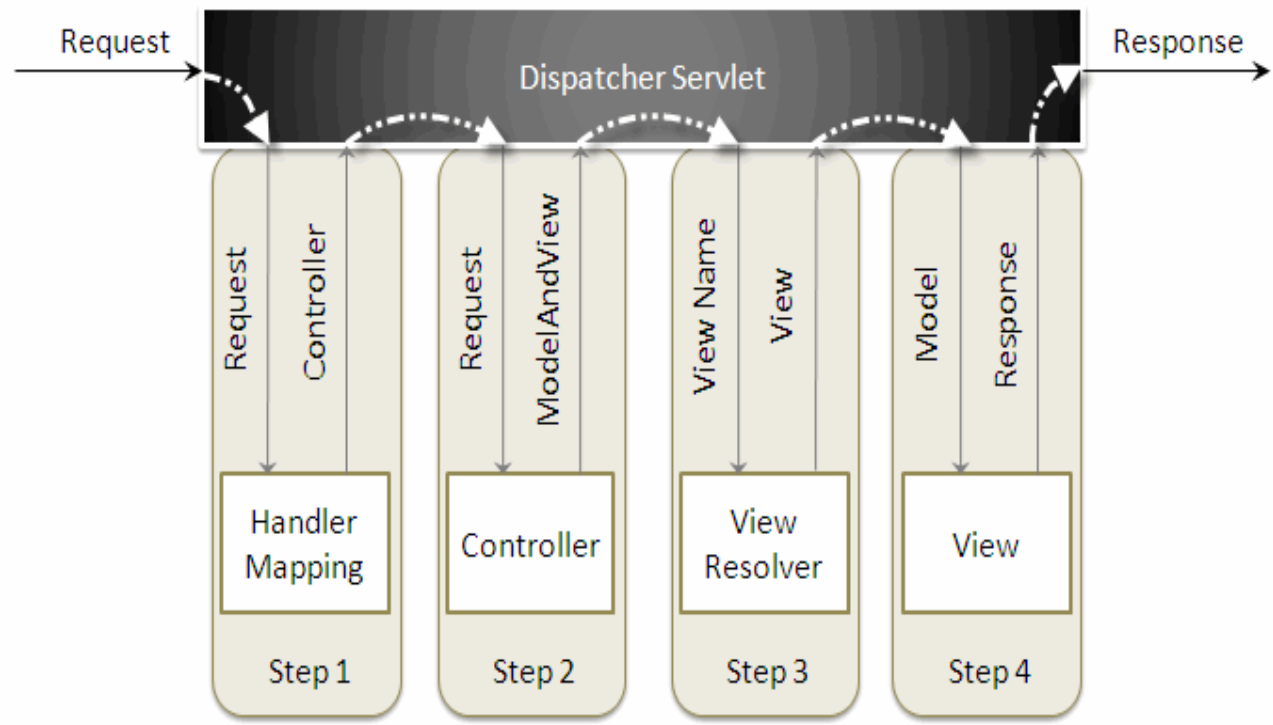
6.Then the selected view is rendered and output is generated in the form of HttpServletResponse. Finally Controller sends the response to the Servlet container, which sends the output to the user.

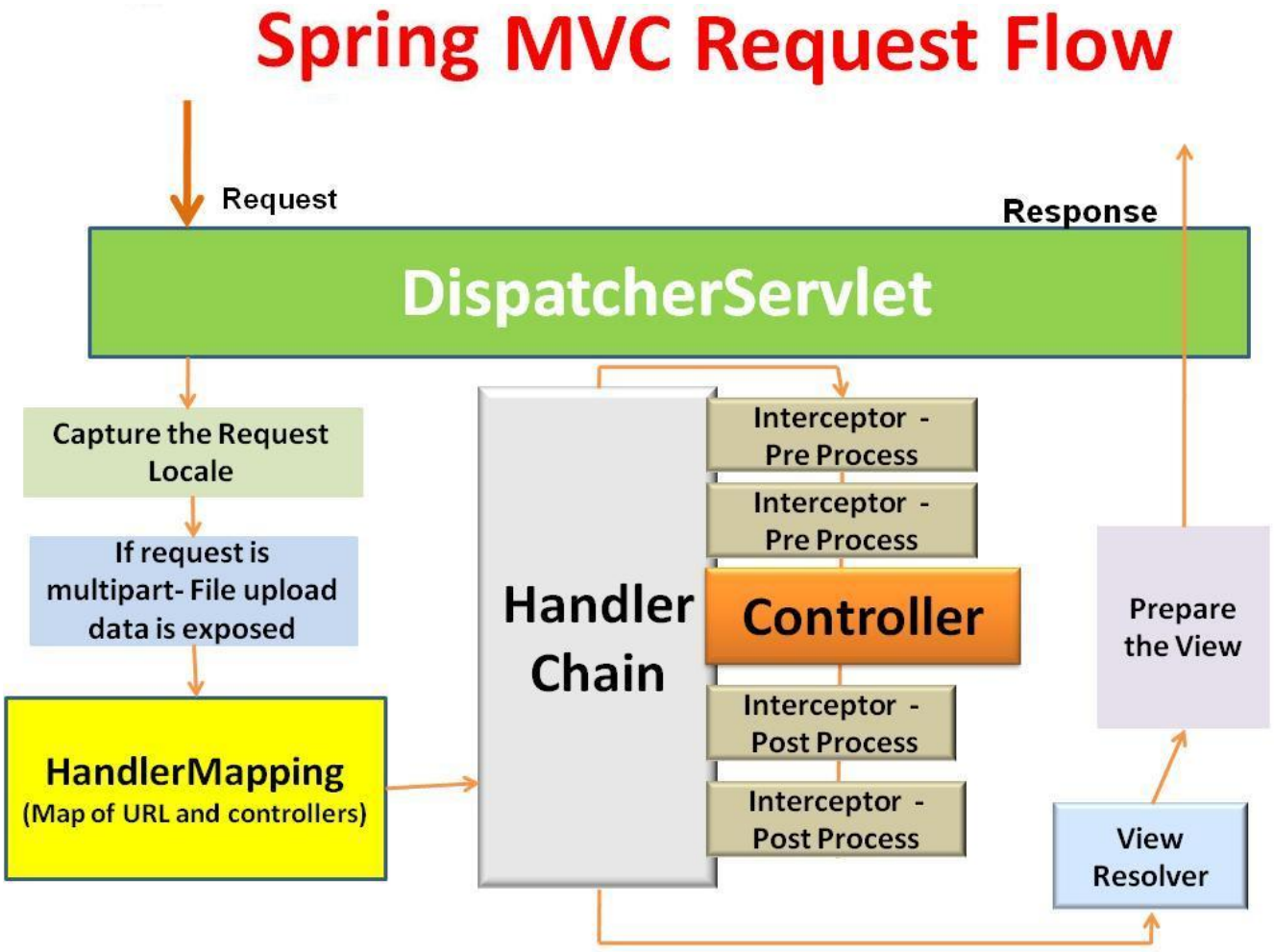


Workflow:

1. The Client requests for a **Resource** in the Web Application.
2. The **Spring Front Controller** i.e. **DispatcherServlet** made a request to **HandlerMapping** to identify the particular controller for the given url.
3. **HandlerMapping** identifies the controller for the given request and sends to the DispatcherServlet.
4. **DispatcherServlet** will call the **handleRequest(req,res)** method on **Controller**. **Controller** is developed by writing a simple java class which implements **Controller** interface or extends its adpater class.
5. **Controller** will call the business method according to business requirement.
6. **Service** class will calls the **Dao** class method for the business data.
7. **Dao** interact with the **DB** to get the database data.
8. **Database** gives the result.
9. **Dao** returns the same data to service.
10. **Dao** given data will be processed according to the business requirements, and returns the results to Controller.

11. The **Controller** returns the **Model and the View** in the form of **ModelAndView** object back to the **Front Controller**.
12. The **Front Controller** i.e. **DispatcherServlet** then tries to resolve the actual View (which may be Jsp, Velocity or Free Marker) by consulting the **View Resolver object**.
13. **ViewResolver** selected View is rendered back to the **DispatcherServlet**.
14. **DispatcherServlet** consult the particular **View** with the **Model**.
15. **View** executes and returns HTML output to the **DispatcherServlet**.
16. **DispatcherServlet** will sends the output to the Browser.

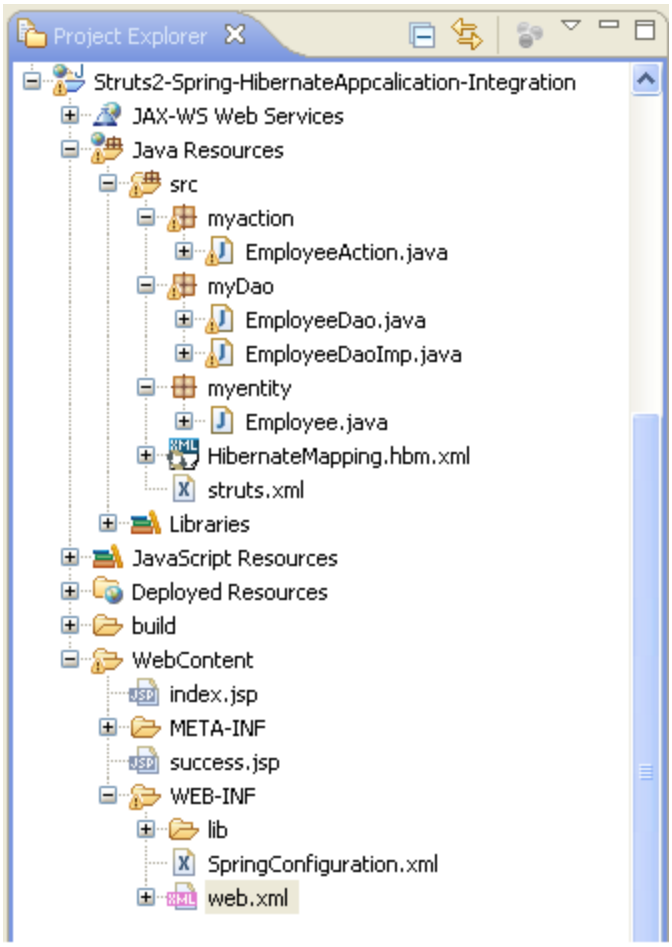




1.**DispatcherServlet** configured in **web.xml** file receives the request for a URL in the application.
2.The **Locale Resolver** component will look for the the Locale information in the request header or session or Cookie as the configuration. The Locale is used to pick the resource files based on the language of the user. This Locale Resolver plays a key role in internationalization of the application.
3.The **Theme Resolver** is bound to the request to make the views determine which theme/CSS needs to be applied.
4.The **Multipart Resolver** component is invoked to check if the request is for a file upload and then wraps the request to facilitate the file upload functionality.
5.The **Handler mapping** component is invoked on the request to get the respective controller which is responsible to handle this request.
6.The **DispatcherServlet** then invokes the HandlerChain which will execute the following:
   1.Checks if there are any interceptors mapped and invokes the Pre Processing logic.
   2.The controllers handler method will be invoked where the request is processed and the result is returned.

3.The mapped interceptors post processing logic will be invoked
7.The **DispactherServlet** based on the result returned by the Controllers handlers method, the ResutlToViewNameTranslator component is invoked to generate the view name.
8.The **view resolver** will then decide on what view needs to be rendered (JSP/XML/PDF/VELOCITY etc.,) and then the result will be dispatched to the client.

Struts2-Spring-HibernateAppcalication-Integration



Employee.java

```
package myentity;
```

```
public class Employee {

	private Long empid;
	private String empname;
	private int age;

	public Long getEmpid() {
		return empid;
	}

	public void setEmpid(Long empid) {
		this.empid = empid;
	}

	public String getEmpname() {
		return empname;
	}

	public void setEmpname(String empname) {
		this.empname = empname;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}
}
```

EmployeeDao.java

```
package myDao;

import java.util.List;

interface EmployeeDao {

	List showEmployuee();
}
```

EmployeeDaoImpl.java

```
package myDao;

import java.util.List;

import org.hibernate.Session;
```

```
import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class EmployeeDaoImp implements EmployeeDao {

	private SessionFactory sessionfactory;

	public void setSessionfactory(SessionFactory sessionfactory) {
		this.sessionfactory = sessionfactory;
	}

	@Override
	public List showEmployuee() {

		Session session = null;

		session = sessionfactory.openSession();
		List emp = null;
		try {
			emp = session.createQuery("from Employee").list();
		} catch (Exception e) {
			e.printStackTrace();
		}
		return emp;
	}

}
```

EmployeeAction.java

```
package myaction;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
import myDao.*;
import myentity.Employee;

public class EmployeeAction extends ActionSupport implements ModelDriven {

	private Employee emp;
	private List emplist = new ArrayList();
	private EmployeeDaoImp empdao;

	public void setEmpdao(EmployeeDaoImp empdao) {
		this.empdao = empdao;
```

```
	}

	@Override
	public Employee getModel() {
		return emp;
	}

	@Override
	public String execute() throws Exception {
		emplist = empdao.showEmployuee();
		return SUCCESS;

	}

	public Employee getEmp() {
		return emp;
	}

	public void setEmp(Employee emp) {
		this.emp = emp;
	}

	public List getEmplist() {
		return emplist;
	}

	public void setEmplist(List emplist) {
		this.emplist = emplist;
	}

}
```

HibernateMapping.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
	<class name="myentity.Employee" table="EMPLOYEE">
		<id name="empid">
			<column name="id" />
			<generator class="assigned" />
		</id>
		<property name="empname">
			<column name="name" />
		</property>
		<property name="age">
			<column name="age" />
		</property>
```

```
        </class>
    </hibernate-mapping>
```

struts.xml

```
    <!DOCTYPE struts PUBLIC "-//Apache Software
    Foundation//DTD Struts
    Configuration 2.0//EN" "http://struts.apache.
    org/dtds/struts-2.0.dtd">

    <struts>
        <package name="default" extends="struts-default">

            <action name="showEmp" class="myaction.EmployeeAction">
                <result name="success">/success.jsp</result>
            </action>
        </package>
    </struts>
```

SpringConfiguration.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:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

        <bean id="springdatasource"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <property name="driverClassName" value="com.mysql.jdbc.Driver" />
            <property name="url" value="jdbc:mysql://localhost:3306/sun" />
            <property name="username" value="root" />
            <property name="password" value="root" />
        </bean>

        <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
            <property name="dataSource" ref="springdatasource" />
            <property name="mappingResources">
                <list>
                    <value>HibernateMapping.hbm.xml</value>
                </list>
            </property>
            <property name="hibernateProperties">
                <value>hibernate.dialect=org.hibernate.dialect.MySQLDialect</value>
            </property>
        </bean>
```

```
        <bean id="empdao" class="myDao.EmployeeDaoImp">
            <property name="sessionfactory" ref="sessionFactory" />
        </bean>

        <bean id="empAction" class="myaction.EmployeeAction">
            <property name="empdao" ref="empdao" />
        </bean>

    </beans>
```

index.jsp

```
    <%@taglib uri="/struts-tags" prefix="s"%>
    <html>

    <body>
        <h3>Welcome to Employee Information Page</h3>
        <br>

        <s:form action="showEmp">
            <s:submit value="Show Records" />
        </s:form>


    </body>
    </html>
```

success.jsp

```
    <%@taglib uri="/struts-tags" prefix="s"%>
    <html>

    <body>
        <h3>Welcome to Employee Information Page</h3>
        <br>
        <table>
            <tr>
                <th>EMP_ID</th>
                <th>EMP_NAME</th>
                <th>EMP_AGE</th>
            </tr>
            <s:iterator value="emplist">
                <tr>
                    <td><s:property value="empid" /></td>
                    <td><s:property value="empname" /></td>
                    <td><s:property value="age" /></td>
                </tr>
            </s:iterator>
```

```
        </table>

    </body>
    </html>
```

web.xml

```
    <?xml version="1.0" encoding="UTF-8"?>
    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
            xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
            id="WebApp_ID" version="2.5">
      <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/SpringConfiguration.xml</param-value>
      </context-param>
      <filter>
        <filter-name>Struts2Filter</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
      </filter>
      <filter-mapping>
        <filter-name>Struts2Filter</filter-name>
        <url-pattern>/*</url-pattern>
      </filter-mapping>
      <listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
      </listener>
      <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
      </welcome-file-list>
    </web-app>
```

Required Jars

antlr-2.7.6.jar
c3p0-0.9.1.jar
commons-beanutils-1.7.0.jar
commons-chain-1.2.jar
commons-collections-3.1.jar
commons-collections-3.2.jar
commons-digester-2.0.jar
commons-fileupload-1.2.1.jar
commons-io-1.3.2.jar
commons-lang-2.3.jar

commons-logging-1.0.4.jar
commons-logging-api-1.1.jar
commons-validator-1.3.1.jar
dom4j-1.6.1.jar
ehcache-1.2.3.jar
freemarker-2.3.16.jar
hibernate3.jar
hibernate-cglib-repack-2.1_3.jar
hibernate-testing.jar
javassist-3.3.ga.jar
javassist-3.4.GA.jar
jboss-cache-1.4.1.GA.jar
jbosscache-core-2.1.1.GA.jar
jta-1.1.jar
mysql-connector-java-5.1.13-bin.jar
ognl-3.0.jar
org.springframework.aop-3.0.0.M4.jar
org.springframework.asm-3.0.0.M4.jar
org.springframework.aspects-3.0.0.M4.jar
org.springframework.beans-3.0.0.M4.jar
org.springframework.context.support-3.0.0.M4.jar
org.springframework.context-3.0.0.M4.jar
org.springframework.core-3.0.0.M4.jar
org.springframework.expression-3.0.0.M4.jar
org.springframework.instrument.classloading-3.0.0.M4.jar
org.springframework.instrument-3.0.0.M4.jar
org.springframework.integration-tests-3.0.0.M4.jar
org.springframework.jdbc-3.0.0.M4.jar
org.springframework.jms-3.0.0.M4.jar
org.springframework.orm-3.0.0.M4.jar
org.springframework.oxm-3.0.0.M4.jar
org.springframework.test-3.0.0.M4.jar
org.springframework.transaction-3.0.0.M4.jar
org.springframework.web.portlet-3.0.0.M4.jar
org.springframework.web.servlet-3.0.0.M4.jar
org.springframework.web-3.0.0.M4.jar
oscache-2.1.jar
proxool-0.8.3.jar
slf4j-api-1.4.2.jar
slf4j-simple-1.4.2.jar
spring-beans-2.5.6.jar
spring-context-2.5.6.jar
spring-core-2.5.6.jar
spring-web-2.5.6.jar
sslext-1.2-0.jar
struts2-core-2.2.1.jar
struts2-spring-plugin-2.2.1.1.jar
struts2-tiles-plugin-2.2.1.1.jar
swarmcache-1.0RC2.jar

tiles-api-2.0.6.jar
tiles-core-2.0.6.jar
tiles-jsp-2.0.6.jar
xwork-core-2.2.1.jar

==========

||Jai Hind||