Spring Core

Specification

Provides API , standards, recommended practices, codes and technical publications, reports and studies.

JCP - Java Community Process
JSR - Java Specification Request

JSRs directly relate to one or more of the Java platforms.

There are 3 collections of standards that comprise the three Java editions:

Standard, Enterprise and Micro

Java EE (47 JSRs)
The Java Enterprise Edition offers APIs and tools for developing multitier enterprise applications.

Java SE (48 JSRs)
The Java Standard Edition offers APIs and tools for developing desktop and server-side enterprise applications.

Java ME (85 JSRs)
Java ME technology, Java Micro Edition, designed for embedded systems (mobile devices)

JSR 168,286,301 - Portlet Applications

JSR 127,254,314 - JSF

JSR 220 - Ejb3.0 & Jpa   JSR 318 – EJB 3.1

JSR 340: Java Servlet 3.1 Specification

JSR 250 - Common Annotaions for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws

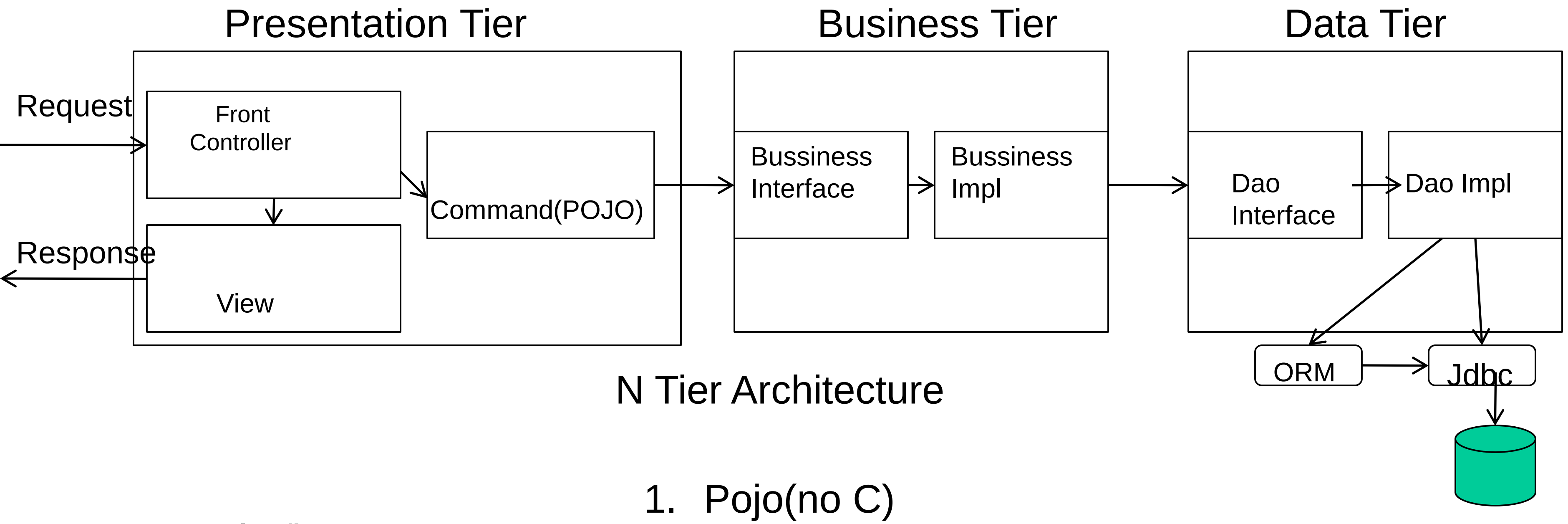
JSR 311 - Jax-RS

JSR 299 - Context & DI

JSR 330 – DI

JSR-303 - Bean Validations

JSR208,312 – JBI (Java Business Integration)



N Tier Architecture

1. Servlet/jsp
2. MVC
   Struts
   JSF
   Flex
   Gwt
   Spring MVC
   ...

> Any MVC + Spring

1. Pojo(no C)
2. Ejb 2.x(HW C)
   -Session Bean
   -Mdb
3. Pojo + LW Containe
   - Spring
   - Microcontainer
   - Xwork
4. Ejb3.0

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
   - Hibernate
   - Kodo
   - Toplink
   - MyBatis
5. JPA

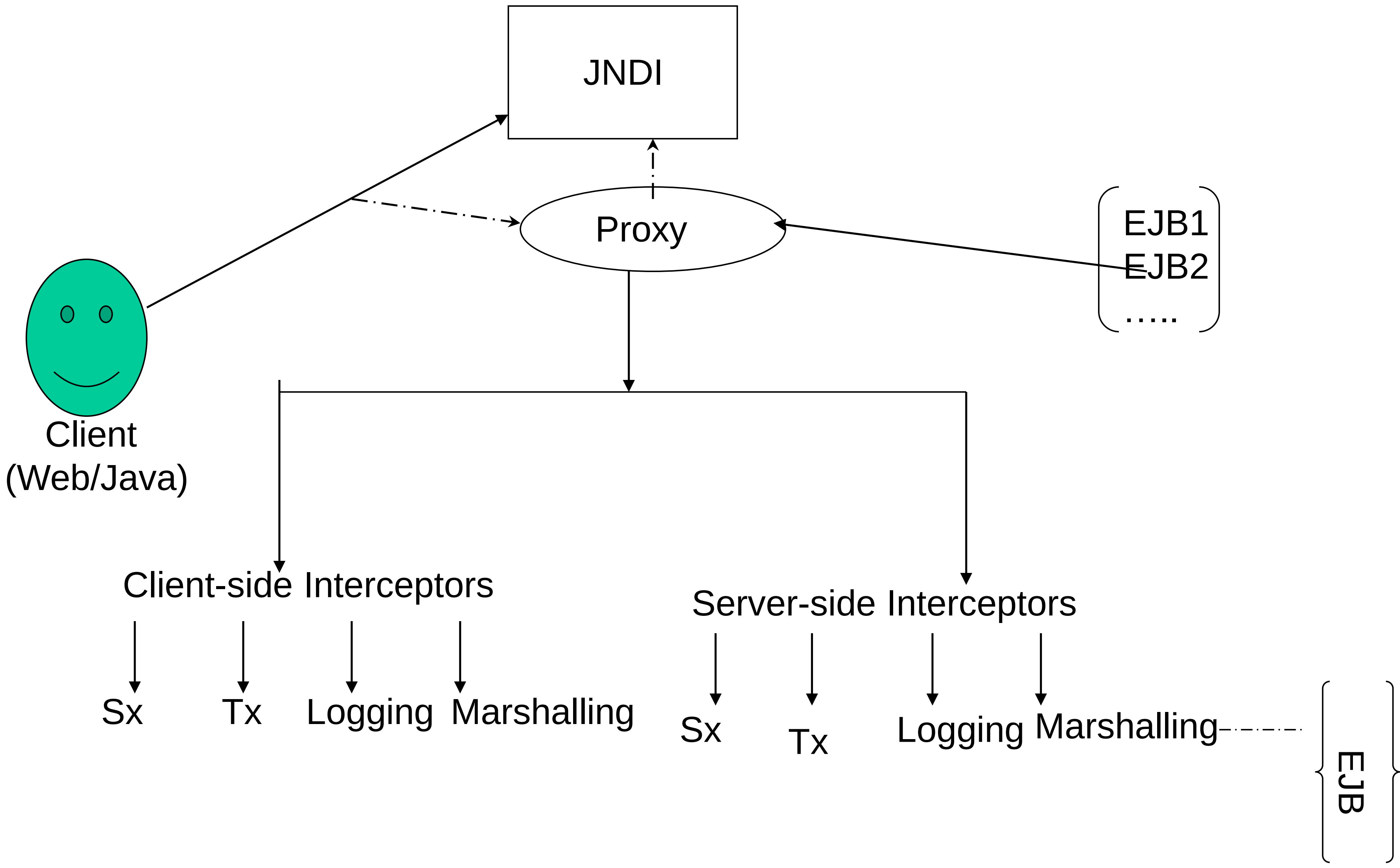
+ Spring Templates

POJO Vs EJB

POJO	EJB
1. Bean-managed Life cycle Management	1. Container-mangaged Lifecycle
2. No Remoting	2. Can be Remoted
3. Bean-managed Services (Tx,Sx etc.,)	3. Declarative Services (Tx,Sx etc.,)
4. No container	4. Application server provides EJB Container
5. Simple to code	5. Complex coding – Home interface, Remote Interface, Deployment Descriptors, API specific interfaces/ classes

1. Ejb Container is heavy-weight, there are different types of EJB’s and too many declarative services

2. Spring, Picocontainer, MircroContainers, Xwork, Avalon etc., provides light-weight container to POJO’s


JNDI
Proxy
EJB1
EJB2
…..
Client
(Web/Java)
Client-side Interceptors
Sx
Tx
Logging
Marshalling
Server-side Interceptors
Sx
Tx
Logging
Marshalling
EJB

IOC is used to decouple common task from implementation.

Six basic techniques to implement Inversion of Control.

These are:

1. using a factory pattern
2. using a service locator pattern
3. using a constructor injection
4. using a setter injection
5. using an interface injection
6. using a contextualized lookup

Constructor, setter, and interface injection are all
 aspects of Dependency injection.

constructor injection to enforce required dependencies

Setter injection would be used for optional parameters.

SPRING BOOT

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that can you can "just run". Most Spring Boot applications need very little Spring configuration. Features

- Create stand-alone Spring applications

- Embed Tomcat or Jetty directly (no need to deploy WAR files)

- Provide opinionated 'starter' POMs to simplify your Maven configuration

- Automatically configure Spring whenever possible

- Provide production-ready features such as metrics, health checks and externalized configuration

- Absolutely no code generation and no requirement for XML configuration

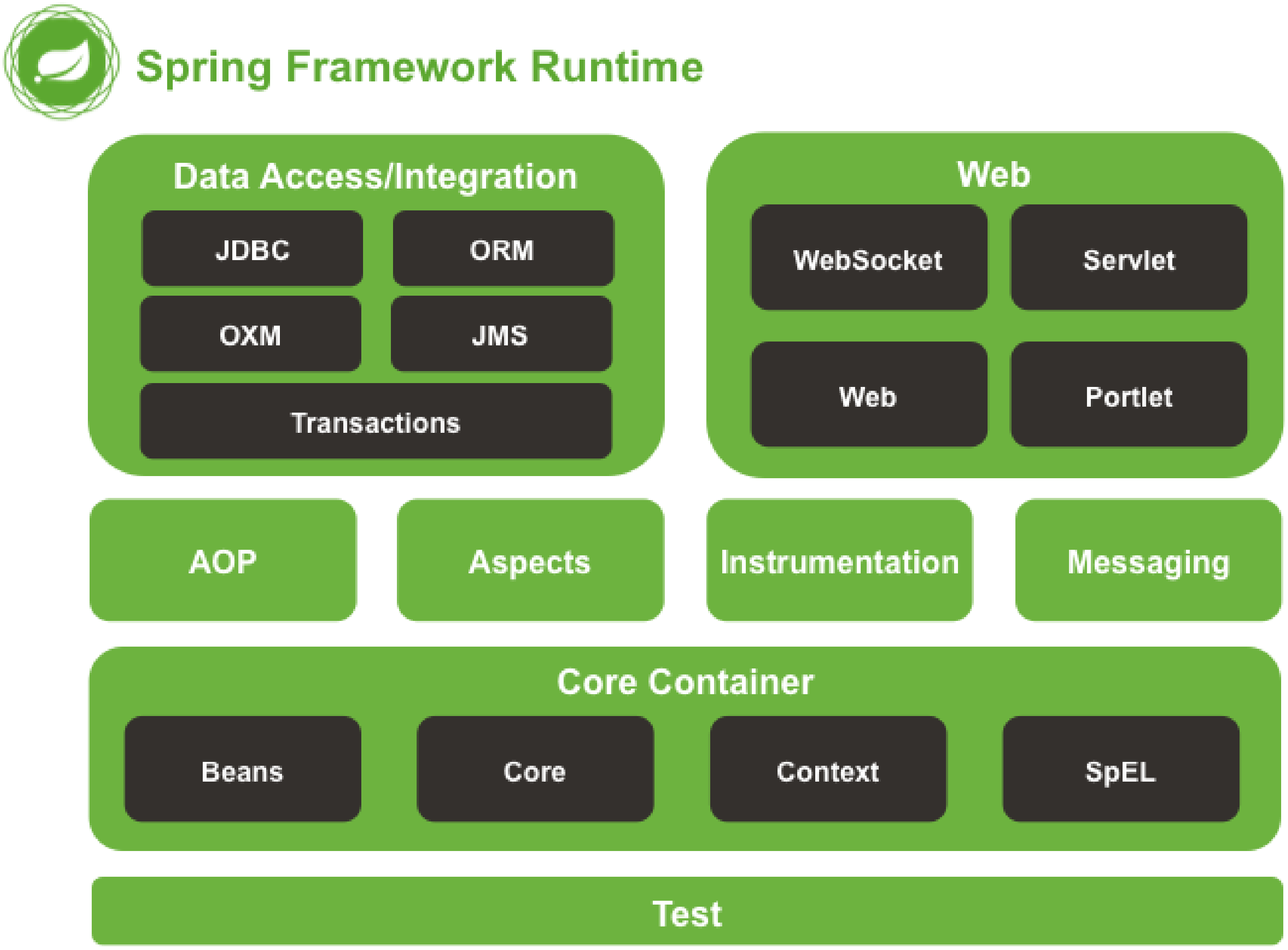
SPRING BATCH

Spring Batch provides reusable functions that are essential in processing :

large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management.

SPRING XD

Spring XD is a unified, distributed, and extensible system for data ingestion, real time analytics, batch processing, and data export. The project’s goal is to simplify the development of big data applications.


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
WebSocket
Servlet
Web
Portlet
AOP
Aspects
Instrumentation
Messaging
Core Container
Beans
Core
Context
SpEL
Test

Core Container

The Core Container consists of the spring-core, spring-beans, spring-context, springcontext-support, and spring-expression (Spring Expression Language) modules.

The spring-core and spring-beans modules provide the fundamental parts of the framework,including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The Context (spring-context) module builds on the solid base provided by the Core and Beans modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry.

The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event propagation, resource loading, and the transparent creation of contexts by, for example, a Servlet container.

The spring-expression module provides a powerful Expression Language for querying and manipulating an object graph at runtime.

AOP, Aspects and Instrumentation

These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.

The aspects module provides support to integration with AspectJ.

The instrumentation module provides support to class instrumentation and classloader implementations.

Web

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

Spring's Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring's remoting support.

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

Spring Framework Versions

The Spring Framework was first released in 2004; since then there have been significant major revisions:

Spring 2.0 provided XML namespaces and AspectJ support;

Spring 2.5 provided annotation-driven configuration;

Spring 3.0 introduced a strong Java 5+ foundation across the framework codebase, and features such as the Java-based @Configuration model.

Version 4.0 is the latest major release of the Spring Framework and the first to fully support Java 8 features (minimum requirement is Java SE 6)

New Features in Spring 2.5

 Annotation-driven dependency injection through @Autowired annotation and fine-grained auto wiring control with @Qualifier.

Support for JSR-250 annotations, including @Resource for dependency injection of a named resource, as well as @PostConstruct and @PreDestroy for life-cycle methods.

Auto-detection of Spring components that are annotated with @Component.

Annotation-driven Spring MVC programming model that greatly simplifies Spring web development such as @Controller And Service layer with @Service. **@Repository with DAO layer was introduced in version 2.0*

Full Java 6 and Java EE 5 support, including JDBC 4.0, JTA 1.1, Java Mail 1.4 and JAX-WS2.0.

A new bean name pointcut expression for weaving aspects into Spring beans by their name.Built-in support for AspectJ load-time weaving.

New XML configuration namespaces, including context namespace for configuring application context details and a jms namespace for configuring message-driven beans.

What’s new In Spring 3.0?

- Spring 3.0 makes the entire spring code base to take advantage of the Java 5.0 technology. The notable Java 5 features like Generics, Varargs, Annotations and other improvements has been extensively implemented with the Spring 3.0.

- Full-scale REST support in Spring MVC, including Spring MVC controllers that respond to REST-style URLs with XML, JSON, RSS or any other appropriate response.

A new expression language that brings Spring dependency injection to a new level by enabling injection of values from a verity of sources, including beans and system properties.

New annotations for Spring MVC, including @CookieValue and @Request-Header, to pull values from cookies and request headers, respectively.

A new XML namespace for easing configuration of Spring MVC.

Support for declarative validation with JSR-303 (Bean Validation) annotations(javax.validation.*) : @Size, @NotNull, @Min(2)

Support for the new JSR-330 dependency injection specification. Annotation-oriented declaration of asynchronous and scheduled methods(javax.inject.*) : @Inject, @Scope, @Named

A new annotation-based configuration model that allows for nearly XML-free Spring configuration (@Configuration)

The Object-to-XML (OXM) mapping functionality from the Spring Web services project has been moved into the core Spring framework.

What’s new In Spring 4.x?

Supports Java 8 with expression of callbacks using lambdas, JSR 310 Date and Time API, and parameter name discovery.

Java EE 7 support includes JMS 2.0, JTA 1.2, JPA 2.1, Bean Validation 1.1, and JSR-236 Concurrency utilities.

Spring 4 also features improved REST support with a new AsyncRestTemplate and HTML5/WebSocket integration.

Lambda expression provides a way to represent one method interface using an expression

Old

```
– button.addActionListener(
new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
doSomethingWith(e);
}
});
```

• **New**

```
– button.addActionListener(e -> doSomethingWith(e));
```

Lambda Conventions

- **Many common Spring APIs are candidates for lambdas**
  - through naturally following the lambda interface conventions
  - formerly "single abstract method" types, now "functional interfaces"

- **JdbcTemplate**
  - RowMapper:
    Object mapRow(ResultSet rs, int rowNum) throws SQLException

- **JmsTemplate**
  - MessageCreator:
    Message createMessage(Session session) throws JMSException

- **TransactionTemplate, TaskExecutor, etc**

Instead of writing code like this:

```
jdbcTemplate.query("SELECT * from products",
new RowMapper<Product>(){
  @Override
  public Product mapRow(ResultSet rs, int rowNum) throws
SQLException {
    Integer id = rs.getInt("id");
    String description = rs.getString("description");
...
```

we write the code like this:

```
jdbcTemplate.query("SELECT * from queries.products",
(rs, rowNum) -> {
    Integer id = rs.getInt("id");
    String description = rs.getString("description");
    Integer quantity = rs.getInt("quantity");
```

Beans, BeanFactory and the ApplicationContext

Two of the most fundamental and important packages in Spring are the

org.springframework.beans and
org.springframework.context

Code in these packages provides the basis for Spring's *Inversion of Control* (alternately called *Dependency Injection*) features.

The BeanFactory provides an advanced configuration mechanism capable of managing beans (objects) of any nature, using potentially any kind of storage facility.

The ApplicationContext builds on top of the BeanFactory (it's a subclass) and adds other functionality such as

• easier integration with Springs AOP features,
• message resource handling (for use in internationalization),
• event propagation,
• declarative mechanisms to create the ApplicationContext and optional parent contexts, and application-layer specific contexts such as the WebApplicationContext, among other enhancements.

Unsure, which one to use BeanFactory or an ApplicationContext ?

when building most applications in a J2EE-environment, *the best option is to use the ApplicationContext*, since it offers all the features of the BeanFactory and adds on to it in terms of features, while also allowing a more declarative approach to use of some functionality

when you might prefer to use the BeanFactory is when memory usage is the greatest concern (such as in an applet where every last kilobyte counts), and you don't need all the features of the ApplicationContext.

BeanFactory and BeanDefinitions - the basics:

The BeanFactory

The BeanFactory is the actual *container* which instantiates, configures, and manages a number of beans.

These beans typically collaborate with one another, and thus have dependencies between themselves.

These dependencies are reflected in the configuration data used by the BeanFactory

A BeanFactory is represented by the interface

org.springframework.beans.factory.BeanFactory,

for which there are multiple implementations.

The most commonly used simple BeanFactory implementation

is org.springframework.beans.factory.xml.XmlBeanFactory.

```
Resource res = new FileSystemResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

or

```
ClassPathResource res = new ClassPathResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

or

```
ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml", "applicationContext-part2.xml"});
// of course, an ApplicationContext is just a BeanFactory
BeanFactory factory = (BeanFactory) appContext;
```

For many usage scenarios, user code will not have to instantiate the BeanFactory or ApplicationContext, since Spring Framework code will do it.

For example, the web layer provides support code to load a Spring ApplicationContext automatically as part of the normal startup process of a J2EE web-app(web.xml)

```
<context-param>
 <param-name>contextConfigLocation</param-name>
<param-value>
/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml
</param-value> </context-param>
 <listener>
<listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

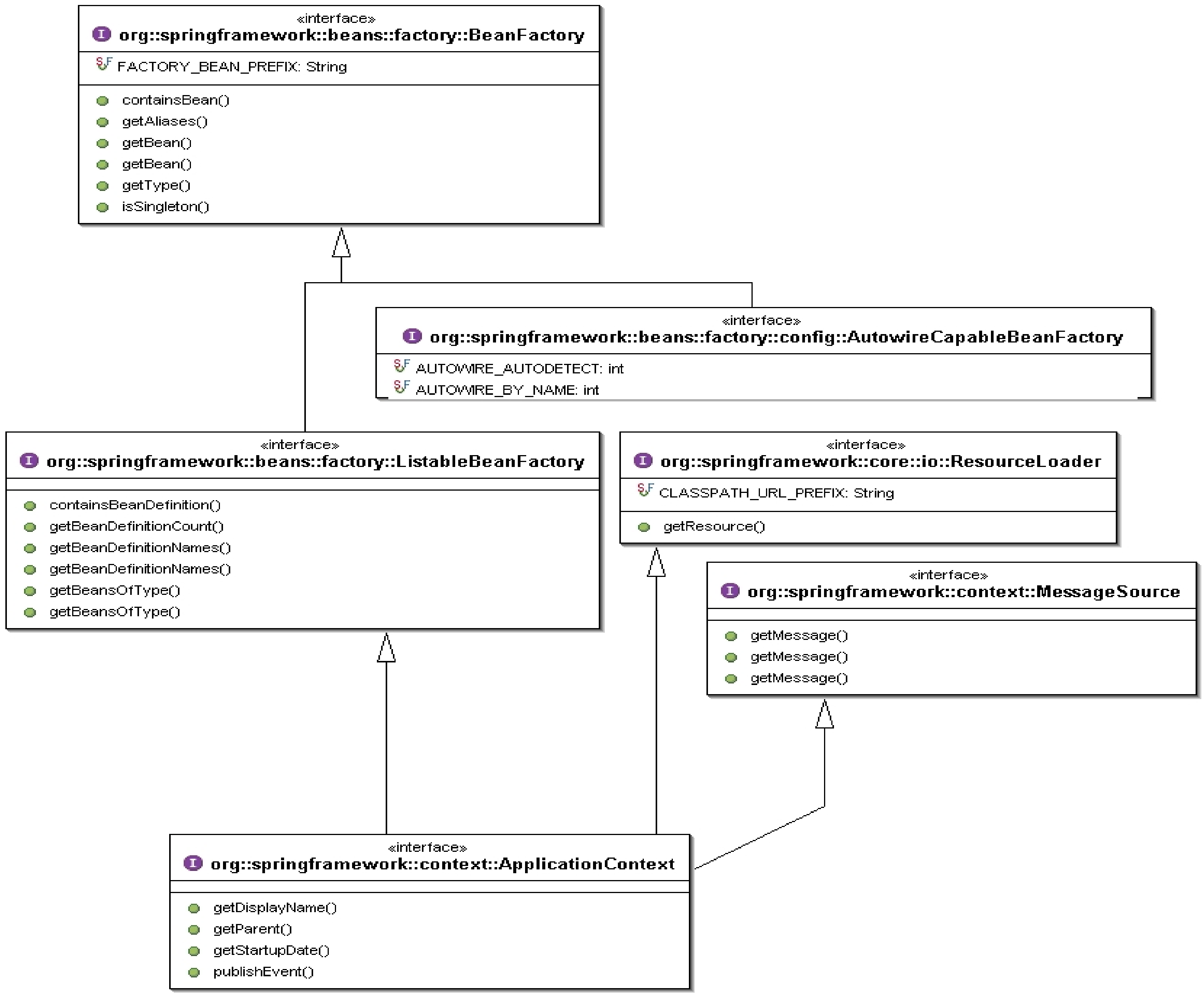
A BeanFactory configuration consists of, at its most basic level, definitions of one or more beans that the BeanFactory must manage.

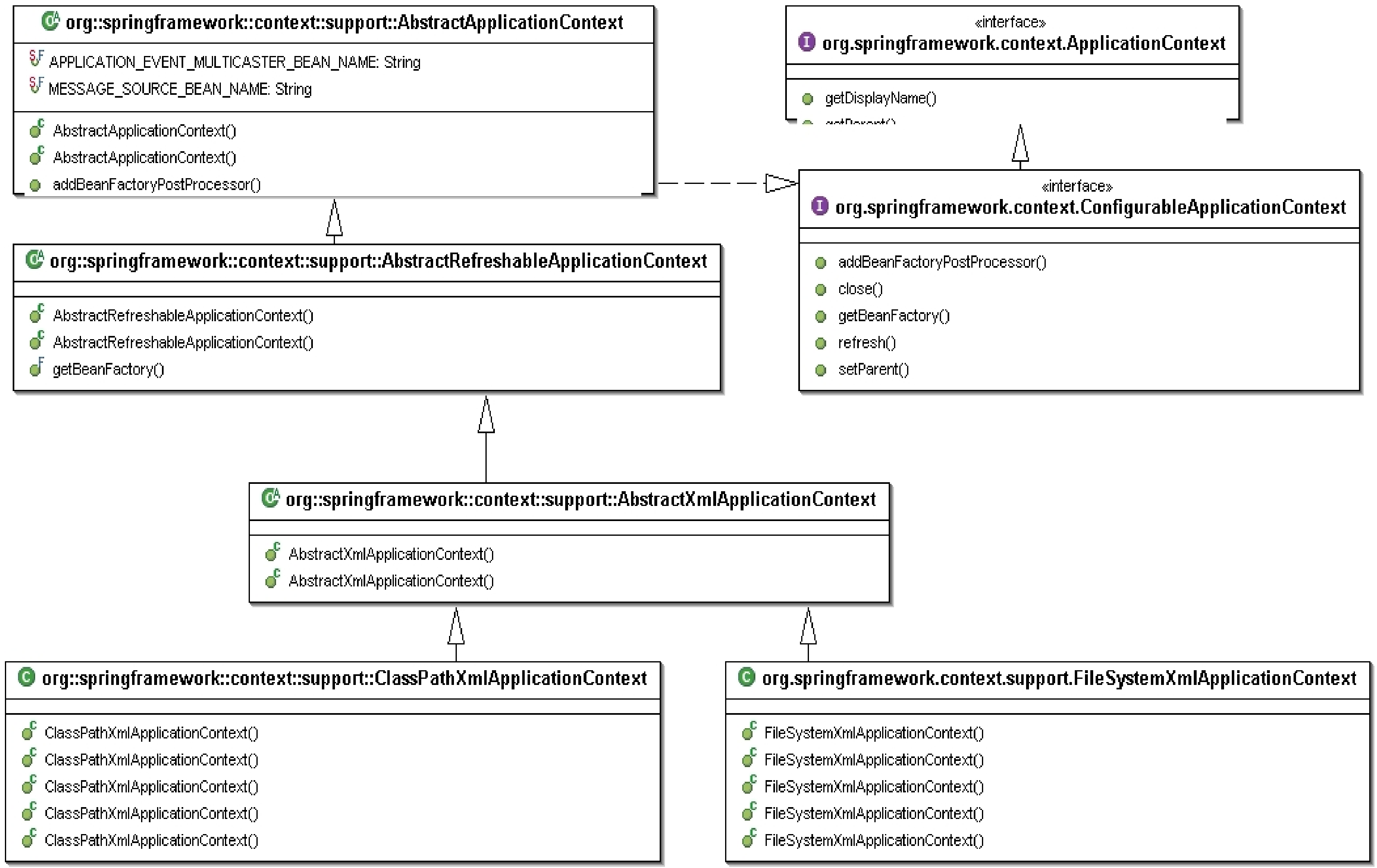
In an XmlBeanFactory, these are configured as one or more bean elements inside a top-level beans element.

```
<?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="..." class="...">
    ...
  </bean>
  <bean id="..." class="...">
    ...
  </bean>
  ...

</beans>
```


«interface»
org::springframework::beans::factory::BeanFactory
FACTORY_BEAN_PREFIX: String
containsBean()
getAliases()
getBean()
getBean()
getType()
isSingleton()
«interface»
org::springframework::beans::factory::config::AutowireCapableBeanFactory
AUTOWIRE_AUTODETECT: int
AUTOWIRE_BY_NAME: int
«interface»
org::springframework::beans::factory::ListableBeanFactory
containsBeanDefinition()
getBeanDefinitionCount()
getBeanDefinitionNames()
getBeanDefinitionNames()
getBeansOfType()
getBeansOfType()
«interface»
org::springframework::core::io::ResourceLoader
CLASSPATH_URL_PREFIX: String
getResource()
«interface»
org::springframework::context::MessageSource
getMessage()
getMessage()
getMessage()
«interface»
org::springframework::context::ApplicationContext
getDisplayName()
getParent()
getStartupDate()
publishEvent()


org::springframework::context::support::AbstractApplicationContext
APPLICATION_EVENT_MULTICASTER_BEAN_NAME: String
MESSAGE_SOURCE_BEAN_NAME: String
AbstractApplicationContext()
AbstractApplicationContext()
addBeanFactoryPostProcessor()
«interface»
org.springframework.context.ApplicationContext
getDisplayName()
«interface»
org.springframework.context.ConfigurableApplicationContext
addBeanFactoryPostProcessor()
close()
getBeanFactory()
refresh()
setParent()
org::springframework::context::support::AbstractRefreshableApplicationContext
AbstractRefreshableApplicationContext()
AbstractRefreshableApplicationContext()
getBeanFactory()
org::springframework::context::support::AbstractXmlApplicationContext
AbstractXmlApplicationContext()
AbstractXmlApplicationContext()
org::springframework::context::support::ClassPathXmlApplicationContext
ClassPathXmlApplicationContext()
ClassPathXmlApplicationContext()
ClassPathXmlApplicationContext()
ClassPathXmlApplicationContext()
ClassPathXmlApplicationContext()
org.springframework.context.support.FileSystemXmlApplicationContext
FileSystemXmlApplicationContext()
FileSystemXmlApplicationContext()
FileSystemXmlApplicationContext()
FileSystemXmlApplicationContext()
FileSystemXmlApplicationContext()

Different Bean Factories :

1. BeanFactory : The basic interface used to access all bean factories.

The simple getBean(String name) method allows you
to get a bean from the container by name.

The getBean(String name, Class requiredType) allows you
to specify the required class of the returned bean, throwing
exception if it doesn’t exist.

Query the bean factory to see if a bean exists.

Query to find out the type of a bean.

Query to find out there are any aliases for a bean in the factory.

Find out if a bean is configured as a singleton/prototype.

2. HierarchicalBeanFactory : Most bean factories can be created as part of hierarchy, such that if you ask a factory for a bean, and it doesn’t exist in that particular factory, a parent factory is also asked for the bean, and that parent factory can ask a parent factory of its own, and so on.

   The main advantage of hierarchical layout is to match the actual architectural layers or modules in an application.

   While getting a bean from a factory that is part of a hierarchy is transparent, the HierarchicalBeanFactory interface exists, so that you may ask a factory for its parent.

3. ListableBeanFactory : The methods in this interface allows listing or enumeration of beans in a bean factory, including the names of all the beans, the names of all the beans of a certain type, and the number of beans in the factory.

Allow you to get a Map instance containing all beans of a certain type in the factory.

Note : While methods from the BeanFactory interface automatically take part in any hierarchy that a factory may be part of, the ListableBeanFactory interface applies strictly to one bean factory.

4. AutoWireCapableBeanFactory : This interface allows you, via the autowireBeanProperties() and applyBeanPropertyValues() methods, to have the factory configure an existing external object and supply its dependencies.

The method autowire(), allows you to specify a classname to the factory, have the factory instantiate that class, use reflection to discover all the dependencies of the class, and inject those dependencies into the bean, returning the fully configured object to you.

5. ConfigurableBeanFactory : This interface allows for additional configuration options on a basic bean factory, to be applied during the initialization stage.

The bean class

The class attribute is normally mandatory and is used for two purposes:

> The BeanFactory itself directly creates the bean by calling its constructor (equivalent to java code calling new), the class attribute specifies the class of the bean to be constructed.

> The BeanFactory calls a static, so-called factory method on a class to create the bean, the class attribute specified the actual class containing the static factory method.

Bean creation via constructor :

```
<bean id="exampleBean”
   class="examples.ExampleBean"/>

 <bean name="anotherExample”
    class="examples.ExampleBeanTwo"/>
```

Bean creation via static factory method:

Following is an example of a bean definition which specifies that the bean is to be created by calling a factory-method.

Note that the definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, createInstance must be a *static* method.

```
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance"/>
```

Bean creation via instance factory method :

Quite similar to using a static factory method to create a bean, is the use of an instance (non-static) factory method, where a factory method of an existing bean from the factory is called to create the new bean.

To use this mechanism, the class attribute must be left empty, and the factory-bean attribute must specify the name of a bean in the current or an ancestor bean factory which contains the factory method.

The factory method itself should still be set via the factory-method attribute.

Following is an example:

```
<!-- The factory bean, which contains a method called
    createInstance -->
<bean id="myFactoryBean"
     class="...">
  ...
</bean>
<!-- The bean to be created via the factory bean -->
<bean id="exampleBean"
     factory-bean="myFactoryBean"
     factory-method="createInstance"/>
```

The bean identifiers (id and name)

Every bean has one or more ids (also called identifiers, or names).

These ids must be unique within the BeanFactory or ApplicationContext the bean is hosted in.

A bean will almost always have only one id, but if a bean has more than one id, the extra ones can essentially be considered aliases.

In an XmlBeanFactory (including ApplicationContext variants), you use the id or name attributes to specify the bean id(s), and at least one id must be specified in one or both of these attributes.

The id attribute allows you to specify one id, and as it is marked in the XML DTD (definition document) as a real XML element ID attribute, the parser is able to do some extra validation when other elements point back to this one.

As such, it is the preferred way to specify a bean id. However, the XML spec does limit the characters which are legal in XML IDs.

This is usually not really a constraint, but if you have a need to use one of these characters, or want to introduce other aliases to the bean, you may also or instead specify one or more bean ids (separated by a comma (,) or semicolon (;) via the name attribute.

To singleton or not to singleton

Beans are defined to be deployed in one of two modes:
singleton or non-singleton(prototype)

When a bean is a singleton, only one *shared* instance of the bean will be managed and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned.

The non-singleton, prototype mode of a bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is done.

Beans are deployed in singleton mode by default:

```
<bean id="exampleBean"
      class="examples.ExampleBean"
        singleton="false"/>

<bean name="yetAnotherExample"
      class="examples.ExampleBeanTwo"
        singleton="true"/>
```

Setting bean properties and collaborators :

Inversion of Control/Dependency Injection exists in two major variants:

1. Setter-based dependency injection

2. Constructor-based dependency injection

setter-based dependency injection is realized by calling setters on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Beans defined in the BeanFactory that use setter-based dependency injection are *true JavaBeans*.

Spring generally advocates usage of setter-based dependency injection, since a large number of constructor arguments can get unwieldy, especially when some properties are optional.

constructor-based dependency injection is realized by invoking a constructor with a number of arguments, each representing a collaborator or property.

Additionally, calling a static factory method with specific arguments, to construct the bean, can be considered almost equivalent, and the rest of this text will consider arguments to a constructor and arguments to a static factory method similarly.

Although Spring generally advocates usage of setter-based dependency injection for most situations, it does fully support the constructor-based approach as well, since you may wish to use it with pre-existing beans which provide only multi-argument constructors, and no setters. Additionally, for simpler beans, some people prefer the constructor approach as a means of ensuring beans cannot be constructed in an invalid state.

Ex : 1 Setter-based injection:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>
  <property name="beanTwo"><ref bean="yetAnotherBean"/></property>
  <property name="integerProperty"><value>1</value></property>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }

    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

2. Constructor-based injection:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
  <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>
  <constructor-arg type="int"><value>1</value></constructor-arg>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public ExampleBean(AnotherBean anotherBean,
                    YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

Ex : 3 Now consider a variant of this where instead of using a constructor,

Spring is told to call a static factory method to return an instance of the object.:

```
<bean id="exampleBean" class="examples.ExampleBean"
     factory-method="createInstance">
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
  <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>
  <constructor-arg><value>1</value></constructor-arg>
</bean>



<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    ...

    // a private constructor
    private ExampleBean(...) {
      ...
    }

    // a static factory method
    // the arguments to this method can be considered the dependencies
      of the bean that
    // is returned, regardless of how those arguments are actually used.

    public static ExampleBean createInstance(
            AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        ExampleBean eb = new ExampleBean(...);
        // some other operations
        ...
        return eb;
    }
}
```

Constructor Argument Type Matching

The above scenario *can* use type matching with simple types by explicitly specifying the type of the constructor argument using the type attribute.

For example:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg type="int"><value>7500000</value></constructor-arg>
  <constructor-arg type="java.lang.String"><value>42</value></constructor-arg>
</bean>
```

Constructor Argument Index

Constructor arguments can have their index specified explicitly by use of the index attribute.

For example:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg index="0"><value>7500000</value></constructor-arg>
  <constructor-arg index="1"><value>42</value></constructor-arg>
</bean>
```

The value element:

The value element specifies a property or constructor argument as a

human-readable string representation.

JavaBeans PropertyEditors are used to convert these string values from

a java.lang.String to the actual property or argument type.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/mydb</value>
  </property>
  <property name="username">
    <value>root</value>
  </property>
</bean>
```

The collection elements

The list, set, map, and props elements allow properties and arguments of Java type List, Set, Map, and Properties, respectively, to be defined and set.

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setPeople(java.util.Properties) call -->
  <property name="people">
    <props>
      <prop key="HarryPotter">The magic property</prop>
      <prop key="JerrySeinfeld">The funny property</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource"/>
    </list>
  </property>
```

```
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
<map> <entry>
 <key><value>yup an entry</value></key>
<value>just some string</value>
</entry> <entry>
<key><value>yup a ref</value></key>

<ref bean="myDataSource"/> </entry>
 </map>
 </property> <!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
 <set> <value>just some string</value> <ref bean="myDataSource"/> </set>
 </property>
 </bean>
```

Inner bean definitions via nested bean elements

A bean element inside the property element is used to define a bean value inline, instead of referring to a bean defined elsewhere in the BeanFactory.

The inline bean definition does not need to have any id defined.

```
<bean id="outer" class="...">
  <!-- Instead of using a reference to target, just use an inner bean -->
  <property name="target">
    <bean class="com.mycompany.PersonImpl">
      <property name="name"><value>Tony</value></property>
      <property name="age"><value>51</value></property>
    </bean>
  </property>
</bean>
```

The idref element

An idref element is simply a shorthand and error-proof way to set a property to the String *id* or *name* of another bean in the container.

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
  <property name="targetName">
    <idref bean="theTargetBean"/>
  </property>
</bean>
```

This is exactly equivalent at runtime to the following fragment:

```
<bean id="theTargetBean" class="...">
</bean>

<bean id="theClientBean" class="...">
  <property name="targetName">
    <value>theTargetBean</value>
  </property>
</bean>
```

The ref element :

Specifying the target bean by using the bean attribute of the ref tag is the most general form, and will allow creating a reference to any bean in the same BeanFactory/ApplicationContext (whether or not in the same XML file), or parent BeanFactory/ApplicationContext.

The value of the bean attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean.

```
<ref bean="someBean"/>
```

Specifying the target bean by using the local attribute leverages the ability of the XML parser to validate XML id references within the same file.

The value of the local attribute must be the same as the id attribute of the target bean.

The XML parser will issue an error if no matching element is found in the same file.

```
<ref local="someBean"/>
```

Specifying the target bean by using the parent attribute allows a reference
to be created to a bean which is in a parent BeanFactory (or ApplicationContext)
of the current BeanFactory (or ApplicationContext).

The value of the parent attribute may be the same as either the id attribute of
the target bean, or one of the values in the name attribute of the target bean,
and the target bean must be in a parent BeanFactory or ApplicationContext to
the current one.

The main use of this bean reference variant is when there is a need to wrap
an existing bean in a parent context with some sort of proxy (which may have t
he same name as the parent), and needs the original object so it may wrap it.

```
<ref parent="someBean"/>
```

Method Injection

For most users, the majority of the beans in the container will be singletons. When a singleton bean needs to collaborate with (use) another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, the typical and common approach of handling this dependency by defining one bean to be a property of the other, is quite adequate.

There is however a problem when the bean lifecycles are different.

Consider a singleton bean A which needs to use a non-singleton (prototype) bean B, perhaps on each method invocation on A. The container will only create the singleton bean A once, and thus only get the opportunity to set its properties once. There is no opportunity for the container to provide bean A with a new instance of bean B every time one is needed.

One solution to this problem is to forgo some inversion of control. Bean A can be aware of the container by implementing BeanFactoryAware, and use programmatic means to ask the container via a getBean("B") call for (a new) bean B every time it needs it.

This is generally not a desirable solution since the bean code is then aware of and coupled to Spring.

Note: Method Injection, an advanced feature of the BeanFactory, allows

this use case to be

Lookup method Injection

Lookup method injection refers to the ability of the container to override abstract or concrete methods on managed beans in the container, to return the result of looking up another named bean in the container.

The lookup will typically be of a non-singleton bean as per the scenari
o described above (although it can also be a singleton). Spring implements this through a dynamically generated subclass overriding the method, using bytecode generation via the CGLIB library.

In the client class containing the method to be injected, the method definition must be an abstract (or concrete) definition in this form:

```
protected abstract SingleShotHelper createSingleShotHelper();
```

If the method is not abstract, Spring will simply override the existing implementation. In the XmlBeanFactory case, you instruct Spring to inject/override this method to return a particular bean from the container, by using the lookup-method element inside the bean definition. For example:

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="singleShotHelper class="..." singleton="false">
</bean>

<!-- myBean uses singleShotHelper -->
<bean id="myBean" class="...">
  <lookup-method name="createSingleShotHelper" bean="singleShotHelper"/>
  <property>
    ...
  </property>
</bean>
```

Using depends-on:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager">
  <property name="manager"><ref local="manager"/></property>
</bean>

<bean id="manager" class="ManagerBean"/>
```

AUTOWIRING COLLABORATIONS

Autowiring modes

Mode Explanation

1. no: No autowiring at all. Bean references must be defined via a ref element. This is the default, and changing this is discouraged for larger deployments, since explicitly specifyin collaborators gives greater control and clarity.

2. byName: Autowiring by property name.
This option will inspect the BeanFactory and look for a bean named exactly the same as the property which needs to be autowired.

For example, if you have a bean definition which is set to autowire by name, and it contains a *master* property (that is, it has a *setMaster*(...) method), Spring will look for a bean definition named master, and use it to set the property.

3. byType: Allows a property to be autowired if there is exactly one bean of the property type in the BeanFactory.
   If there is more than one, a fatal exception is thrown, and this indicates that you may not use *byType* autowiring for that bean.

4. constructor: This is analogous to *byType*, but applies to constructor arguments.
If there isn't exactly one bean of the constructor argument type in the bean factory, a fatal error is raised.

5. autodetect: Chooses *constructor* or *byType* through introspection of the bean class. If a default constructor is found, byType gets applied.

Checking for dependencies

Spring has the ability to try to check for the existence of unresolved dependencies of a bean deployed into the BeanFactory.

These are JavaBeans properties of the bean, which do not have actual values set for them in the bean definition, or alternately provided automatically by the autowiring feature.

This feature is sometimes useful when you want to ensure that all properties are set on a bean.

Mode	**Explanation**
none	No dependency checking. Properties of the bean which have no value specified for them are simply not set.
simple	Dependency checking is performed for primitive types and collections (everything except collaborators, i.e. other beans)
object	Dependency checking is performed for collaborators
all	Dependency checking is done for collaborators, primitive types and collections

Bean’s life cycle :

In traditional Java application, the life cycle of a bean if fairly simple.
-‘new’ keyword is used to instantiate the bean and it’s ready to use.

In contrast, the life cycle of a bean within a Spring container is a bit more elaborate.

We can customize the bean how we want to create?

A bean factory performs several steps before a bean is ready to use :

1. The container finds the bean's definition and instantiate the bean.
2. Using dependency injection, Spring populates all of the operation as specified in the bean definition.
3. If the bean implements the BeannameAware interface, the factory calls setBeanName() passing the bean's ID.
4. If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.
5. If there are any BeanPostProcessors associated with the bean, their ProcessBeforeInitialization() methods will be called.
6. If an init-method is specified for the bean, it will be called.
7. Finally, if there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.

Life cycle of BeanFactory

1. Instantiate
2. Populate properties
3. BeanNameAware’s setBeanName()
4. BeanFactoryAware’s setBeanFactory()
5. Pre-initialization Bean Post processors
6. InitializingBean’s   afterPropertiesSet()
7. Call custom init-method
8. Post-initialization BeanPostProcessors
9. Now Bean is ready to use

Container is shutdown

1. DisposableBean’s destroy
2. Call custom destroy-method

Bean is removed from the bean factory in two ways:

1. If the bean implements the DisposableBean interface, the destroy() method is called
2. If a custom destroy-method is specified, it will be called

Life cycle of a bean within Spring application context

1. Instantiate
2. Populate properties(DI)
3. BeanNameAware's setBeanName()
4. **ApplicationContextAware's setApplicationContext ()**
5. Pre-initialization Bean Post processors
6. Any method with @PostConstruct
7. InitializingBean's   afterPropertiesSet()
8. Call custom init-method
9. Post-initialization BeanPostProcessors
10. Now Bean is ready to use

Container is shutdown

1. Any method with @PreDestroy
2. DisposableBean's destroy
3. Call custom destroy-method

Bean is removed from the bean factory in two ways:

1. If the bean implements the DisposableBean interface, the destroy() method is called
2. If a custom destroy-method is specified, it will be called

At this point, the bean is ready to be used by an application and will remain in the bean factory until it is no longer needed.

Bean is removed from the bean factory in two ways:

1. If the bean implements the DisposableBean interface, the destroy() method is called.

2. If a custom destroy-method is specified, it will be called.

##

If the bean implements the ApplicationContextAware interface, the setApplicationContext() method is called.

Customizing the nature of a bean

Lifecycle interfaces

Spring provides several marker interfaces to change the behavior of your bean in the BeanFactory.

They include InitializingBean and DisposableBean. Implementing these interfaces will result in the BeanFactory calling afterPropertiesSet() for the former and destroy() for the latter to allow the bean to perform certain actions upon initialization and destruction.

Internally, Spring uses BeanPostProcessors to process any marker interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring doesn't offer out-of-the-box, you can implement a BeanPostProcessor yourself. More information about this can be found in

InitializingBean / init-method

Implementing the org.springframework.beans.factory.InitializingBean allows a bean to perform initialization work after all necessary properties on the bean are set by the BeanFactory.

The InitializingBean interface specifies exactly one method:

```
void afterPropertiesSet() throws Exception;
```

Note: generally, the use of the InitializingBean marker interface can be avoided (and is discouraged since it unnecessarily couples the code to Spring). A bean definition provides support for a generic initialization method to be specified.

In the case of the XmlBeanFactory, this is done via the init-method attribute.

For example, the following definition:

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

public class ExampleBean {
    public void init() {
        // do some initialization work
    }
}
```

Is exactly the same as:

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

but does not couple the code to Spring.

DisposableBean / destroy-method

Implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to get a callback when the BeanFactory containing it is destroyed. The DisposableBean interface specifies one method:

```
void destroy() throws Exception;
```

Note: generally, the use of the DisposableBean marker interface can be avoided (and is discouraged since it unnecessarily couples the code to Spring).

A bean definition provides support for a generic destroy method to be specified.

In the case of the XmlBeanFactory, this is done via the destroy-method attribute.

For example, the following definition:

```
<bean id="exampleInitBean" class="examples.ExampleBean"
                       destroy-method="cleanup"/>

public class ExampleBean {
   public void cleanup() {
      // do some destruction work (like closing connection)
   }
}
```

Is exactly the same as:

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {
    public void destroy() {
        // do some destruction work
    }
}
```

but does not couple the code to Spring.

Knowing who you are -- BeanFactoryAware

A class which implements the org.springframework.beans.factory. BeanFactoryAware interface is provided with a reference to the BeanFactory that created it, when it is created by that BeanFactory.

```
public interface BeanFactoryAware {

void setBeanFactory(BeanFactory beanFactory) throws BeansException;

}
```

This allows beans to manipulate the BeanFactory that created them programmatically, through the org.springframework.beans.factory.BeanFactory interface

BeanNameAware

If a bean implements the org.springframework.beans.factory.BeanNameAware interface and is deployed in a BeanFactory, the BeanFactory will call the bean through this interface to inform the bean of the *id* it was deployed under.

The callback will be Invoked after population of normal bean properties but before an init callback like InitializingBean's *afterPropertiesSet* or a custom init-method.

Interacting with the BeanFactory

A BeanFactory is essentially nothing more than the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies.

The BeanFactory enables you to read bean definitions and access them using the bean factory. When using just the BeanFactory you would create one and read in some bean definitions in the XML format as follows:

```
InputStream is = new FileInputStream("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
```

The BeanFactory interface has only five methods for clients to call:

1. boolean containsBean(String): returns true if the BeanFactory contains a bean definition or bean instance that matches the given name

2. Object getBean(String): returns an instance of the bean registered under the given name..

3. Object getBean(String,Class): returns a bean, registered under the given name.

4. boolean isSingleton(String): determines whether or not the bean definition or bean instance registered under the given name is a singleton or a prototype.

5. String[] getAliases(String): Return the aliases for the given bean name, if any were defined in the bean definition

Obtaining a FactoryBean, not its product

Sometimes there is a need to ask a BeanFactory for an actual FactoryBean instance itself, not the bean it produces.

This may be done by prepending the bean id with & when calling the getBean method of BeanFactory (including ApplicationContext).

So for a given FactoryBean with an id myBean, invoking getBean("myBean") on the BeanFactory will return the product of the FactoryBean, but invoking getBean("&myBean") will return the FactoryBean instance itself.

Customizing bean factories with BeanFactoryPostProcessors

A bean factory post-processor is a java class which implements the org.springframework.beans.factory.config.BeanFactoryPostProcessor interface.

It is executed manually (in the case of the BeanFactory) or automatically (in the case of the ApplicationContext) to apply changes of some sort to an entire BeanFactory, after it has been constructed.

Spring includes a number of pre-existing bean factory post-processors, such as PropertyResourceConfigurer and PropertyPlaceHolderConfigurer, and **BeanNameAutoProxyCreator, very useful for wrapping other beans transactionally or with any other kind of proxy.**

The BeanFactoryPostProcessor can be used to add custom editors.

The PropertyPlaceholderConfigurer

The PropertyPlaceholderConfigurer, implemented as a bean factory post-processor, is used to externalize some property values from a BeanFactory definition, into another separate file in Java Properties format.

This is useful to allow the person deploying an application to customize some key properties (for example database URLs, usernames and passwords), without the complexity or risk of modifying the main XML definition file or files for the BeanFactory.

Consider a fragment from a BeanFactory definition, where a DataSource with placeholder values is defined:

In the example below, a datasource is defined, and we will configure s ome properties from an external Properties file. At runtime, we will apply a PropertyPlaceholderConfigurer to the BeanFactory which will replace some properties of the datasource:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
                                 destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
```

The actual values come from another file in Properties format:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

To use this with a BeanFactory, the bean factory post-processor is manually executed on it:

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource
                                                ("beans.xml"));
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));
cfg.postProcessBeanFactory(factory);
```

Note : ApplicationContexts are able to automatically recognize and apply beans deployed in them which implement BeanFactoryPostProcessor.

Registering additional custom PropertyEditors

When setting bean properties as a string value, a BeanFactory ultimately uses standard JavaBeans PropertyEditors to convert these Strings to the complex type of the property. Spring pre-registers a number of custom PropertyEditors (for example, to convert a classname expressed as a string into a real Class object).

If there is a need to register other custom PropertyEditors, there are several mechanismsavailable.

The most manual approach, which is not normally convenient or recommended, is to simply use the registerCustomEditor() method of the ConfigurableBeanFactory interface, assuming you have a BeanFactory reference.

Introduction to the ApplicationContext

While the beans package provides basic functionality for managing and manipulating beans, often in a programmatic way, the context package adds ApplicationContext, which enhances BeanFactory functionality in a more *framework-oriented style*.

Many users will use ApplicationContext in a completely declarative fashion, not even having to create it manually, but instead relying on support classes such as ContextLoader to automatically start an ApplicationContext as part of the normal startup process of a J2EE web-app.

Of course, it is still possible to programmatically create an ApplicationContext.

The basis for the context package is the ApplicationContext interface, located in the org.springframework.context package.

Deriving from the BeanFactory interface, it provides all the functionality of BeanFactory.

To allow working in a more framework-oriented fashion, using layering and hierarchical contexts, the context package also provides the following:

1.MessageSource, providing access to messages in, i18n-style

2. *Access to resources*, such as URLs and files

3. *Event propagation* to beans implementing the ApplicationListener interface

4. *Loading of multiple (hierarchical) contexts*, allowing each to be focused

5. on one particular layer, for example the web layer of an application

ApplicationContextAware marker interface

All marker interfaces available with BeanFactories still work.
The ApplicationContext does add one extra marker interface which
beans may implement, org.springframework.context.ApplicationContextAware.

A bean which implements this interface and is deployed into the context will
be called back on creation of the bean, using the interface's
setApplicationContext() method, and provided with a reference to the context,
which may be stored for later interaction with the context.

Creating an ApplicationContext from a web application

As opposed to the BeanFactory, which will often be created programmatically, ApplicationContexts can be created declaratively using for example a ContextLoader.

Of course you can also create ApplicationContexts programmatically using one of the ApplicationContext implementations.

First, let's examine the ContextLoader and its implementations.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml
                      </param-value>
</context-param>

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

<!-- OR USE THE CONTEXTLOADERSERVLET INSTEAD OF THE LISTENER
<servlet>
  <servlet-name>context</servlet-name>
  <servlet-class>org.springframework.web.context.ContextLoaderServlet
                       </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
-->
```

Abstracting Access to Low-Level Resources

Java's standard java.net.URL interface and istandard handlers for vairous URL prefixes are unfortunately not quite adequate enough for all access to low-level resources.

There is for example no standardized URL implementation which may be used to access a resource that needs to be obtained from somewhere on the classpath, or relative to a ServletContext, for example.

While it is possible to register new handlers for specialized URL prefixes (similar to existing handlers for prefixes such as http:), this is generally quite complicated, and the URL interface still lacks some desireable functionality, such as a method to check the existence of the resource being pointed to.

The Resource interface

Spring's Resource interface is meant to be a more capable interface for abstracting access to low-level resources.

```
public interface Resource extends InputStreamSource {
    boolean exists();
    boolean isOpen();
    URL getURL() throws IOException;
    File getFile() throws IOException;
    Resource createRelative(String relativePath) throws IOException;
    String getFilename();
    String getDescription();
}

public interface InputStreamSource {

    InputStream getInputStream() throws IOException;

}
```

Built-in Resource implementations

There are a number of built-in Resource implementations

1. **UrlResource**

   This wraps a java.net.URL, and may be used to access any object that is normally accessible via a URL, such as files, an http target, an ftp target, etc.

2. **ClassPathResource**

   This class represents a resource which should be obtained from the classpath.

   This uses either the thread context class loader, a given class loader, or a given class for loading resources

3. FileSystemResource

This is a Resource implementation for java.io.File handles.

It obviously supports resolution as a File, and as a URL.

4 ServletContextResource

This is a Resource implementation for ServletContext resources,

interpreting relative paths within the web application root directory.

5. **InputStreamResource**

A Resource implementation for a given InputStream. This should only be used if no specific Resource implementation is applicable. Prefer ByteArrayResource or any of the file-based Resource implementations

6. **ByteArrayResource**

This is a Resource implementation for a given byte array. It creates ByteArrayInputStreams for the given byte array.

The ResourceLoader Interface

The ResourceLoader interface is meant to be implemented by objects that can return (i.e load) Resources.

```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

All application contexts implement ResourceLoader therefore all application contexts may be used to obtain Resources.

When you call getResource() on a specific application context, and the location path specified doesn't have a specific prefix, you will get back a Resource type that is appropriate to that particular application context.

```
Resource template = ctx.getResource
                    ("classpath:some/resource/path/myTemplate.txt);


Resource template = ctx.getResource

                    ("file:/some/resource/path/myTemplate.txt);


Resource template = ctx.getResource
              ("http://myhost.com/resource/path/myTemplate.txt);
```

Application contexts and Resource paths

1. Constructing application contexts

For example, if you create a ClassPathXmlApplicationContext as follows:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext
                            ("conf/appContext.xml");
```

then the definition will be loaded from the classpath, as a ClassPathResource will be used.

But if you create a FilleSystemXmlApplicationContext as follows:

```
ApplicationContext ctx =
    new FileSystemClassPathXmlApplicationContext("conf/appContext.xml");
```

then the definition will be loaded from a filesystem location, in this case relative to the current working directory.

Note that the use of the special classpath prefix or a standard URL prefix on the location path will override the default type of Resource created to load the definition.

So this FileSystemXmlApplicationContext

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

will actually load its definition from the classpath. However, it's still a FileSystemXmlApplicationContext.

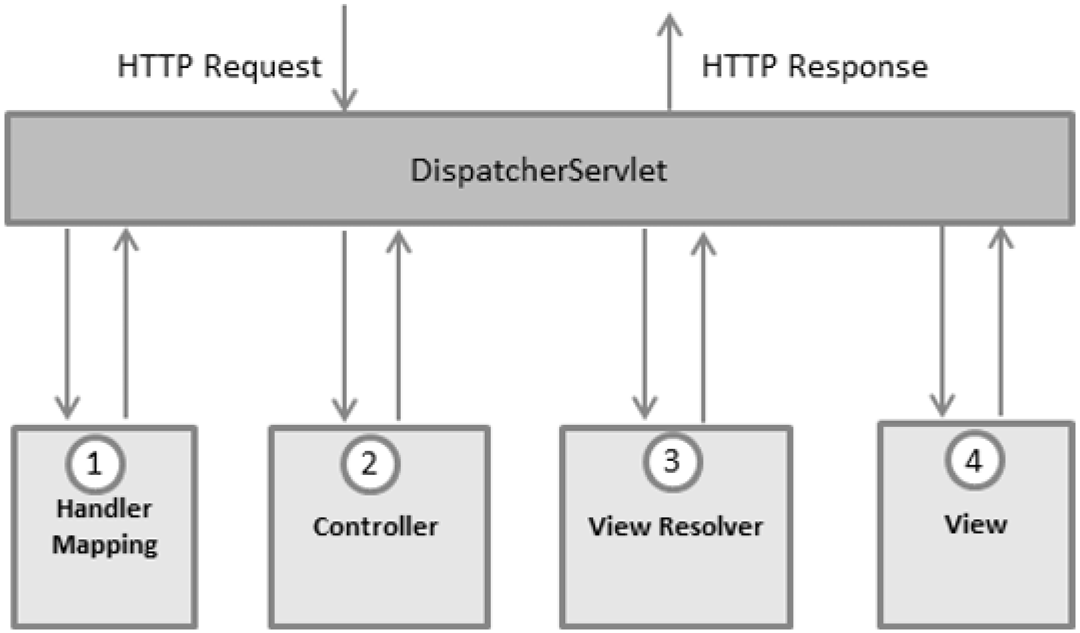
The classpath*: prefix

When constructing an XML-based application context, a location string may use the special classpath*: prefix:

```
ApplicationContext ctx =
   new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

This special prefix specifies that all classpath resources that match the gven name should be obtained (internally, this essentially happens via a ClassLoader.getResources(...) call), and then merged to form the final application context definition.

Spring - MVC Framework



@Controller stereotype

This is the simplest way for creating a controller class to handle one or multiple requests. Just by annotating a class with the @Controller stereotype, for example:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

    @RequestMapping("/")
    public String visitHome() {

        // do something before returning view name

        return "home";
    }
}
```

NOTE: the @Controller stereotype can only be used with annotation-driven is enabled in the Spring’s configuration file:

```
<annotation-driven />
```

When annotation-driven is enabled, Spring container automatically scans for classes under the package specified in the following statement:

```
<context:component-scan base-package=“org.sample" />
```

multi-actions controller class that is able to serve multiple different requests. For example:

```
@Controller
public class MultiActionController {

    @RequestMapping("/listUsers")
    public ModelAndView listUsers() {

    }

    @RequestMapping("/saveUser")
    public ModelAndView saveUser(User user) {

    }

    @RequestMapping("/deleteUser")
    public ModelAndView deleteUser(User user) {

    }
}
```

Implementing the Controller Interface

Another (and maybe classic) way of creating a controller in Spring MVC is having a class implemented the Controller interface. For example:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class MainController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        System.out.println("Welcome main");
        return new ModelAndView("main");
    }
}
```

The implementing class must override the handleRequest() method which will be invoked by the Spring dispatcher servlet when a matching request comes in.

The request URL pattern handled by this controller is defined in the Spring’s context configuration file as follows:

```
<bean name="/main" class="net.codejava.spring.MainController" />
```

Extending the AbstractController Class

Having your controller class extended the AbstractController class if you want to easily control the supported HTTP methods, session and content caching. Consider the following example:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class BigController extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        System.out.println("You're big!");
        return new ModelAndView("big");
    }
}
```

This creates a single-action controller with configurations regarding supported methods, session and caching can be specified in the bean declaration of the controller. For example:

```
<bean name="/info" class="InfoController">
   <property name="supportedMethods" value="POST"/>
</bean>
```

```
public abstract class AbstractController
extends WebContentGenerator
implements Controller
```

AbstractController Workflow:

1. handleRequest() will be called by the DispatcherServlet
2. Inspection of supported methods (ServletException if request method is not support)
3. If session is required, try to get it (ServletException if not found)
4. Set caching headers if needed according to the cacheSeconds property

5. Call abstract method handleRequestInternal() (optionally synchronizing around the call on the HttpSession), which should be implemented by extending classes to provide actual functionality to return ModelAndView objects.

Specifying URL Mapping for Handler Method

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/hello")
public class SingleActionController {

    @RequestMapping(method = RequestMethod.GET)
    public String sayHello() {
        return "hello";
    }
}
```

Specifying HTTP Request Methods for Handler Method

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class LoginController {

    @RequestMapping(value = "/login", method = RequestMethod.GET)
    public String viewLogin() {
        return "LoginForm";
    }

    @RequestMapping(value = "/login", method = RequestMethod.POST)
    public String doLogin() {
        return "Home";
    }
}
```

Mapping Request Parameters to Handler Method

We can retrieve request parameters as regular parameters of the handler method by using the @RequestParam annotation. This is a good way to decouple the controller from the HttpServletRequest interface of Servlet API.

URL as follows:

http://localhost:8080/spring/login?username=scott&password=tiger

```
@RequestMapping(value = "/login", method =
RequestMethod.POST)
public String doLogin(@RequestParam String username,
                    @RequestParam String password) {

}
```

Type conversion is also done automatically. For example, if you declare a parameter of type integer as follows:

```
@RequestParam int securityNumber
```

Then Spring will automatically convert value of the request parameter (String) to the specified type (integer) in the handler method.

In case *the parameter name is different* than the variable name. We can specify actual name of the parameter as follows:

```
@RequestParam("SSN") int securityNumber
```

The @RequestParam annotation also has additional 2 attributes which might be useful in some cases. The required attribute specifies whether the parameter is mandatory or not. For example:

```
@RequestParam(required = false) String country
```

That means the parameter country is optional, hence can be missing from the request. In the above example, the variable country will be null if there is no such parameter present in the request.

Another attribute is defaultValue, which can be used as a fallback value when the request parameter is empty. For example:

```
@RequestParam(defaultValue = "18") int age
```

Returning Model And View

In the following example, the handler method returns a String represents a view named “LoginForm”. No model is returned.

```
@RequestMapping(value = "/login", method = RequestMethod.GET)
public String viewLogin() {
    return "LoginForm";
}
```

But if we want to send additional data to the view, we must return a ModelAndView object. Consider the following handler method:

```
@RequestMapping("/listUsers")
public ModelAndView listUsers() {

    List<User> listUser = new ArrayList<>();
    // get user list from DAO...

    ModelAndView modelView = new ModelAndView("UserList");
    modelView.addObject("listUser", listUser);

    return modelView;
}
```

<mvc:annotation-driven />

registers a RequestMappingHandlerMapping, a RequestMappingHandlerAdapter, and an ExceptionHandlerExceptionResolver (among others) in support of processing requests with annotated controller methods using annotations such as @RequestMapping , @ExceptionHandler, and others.

Support for validating @Controller inputs with @Valid, if a JSR-303 Provider is present on the classpath.

Support for formatting Number fields using the @NumberFormat annotation through the ConversionService.

Support for formatting Date, Calendar, Long, and Joda Time fields using the @DateTimeFormat annotation.

PropertyEditors

PropertyEditors to effect the conversion between an Object and a String.

examples where property editing is used in Spring:

- setting properties on beans is done using PropertyEditors. When mentioning java.lang.String as the value of a property of some bean you're declaring in XML file, Spring will (if the setter of the corresponding property has a Class-parameter) use the ClassEditor to try to resolve the parameter to a Class object.

- parsing HTTP request parameters in Spring's MVC framework is done using all kinds of PropertyEditors that you can manually bind in all subclasses of the CommandController.

<context:annotation-config>

XML element in application context implicitly registered post-processors include:

CommonAnnotationBeanPostProcessor :
@PostConstruct, @PreDestroy, @Resource

AutowiredAnnotationBeanPostProcessor :
@Autowired, @Value, @Inject, @Qualifier, etc

RequiredAnnotationBeanPostProcessor : @Required annotation

PersistenceAnnotationBeanPostProcessor :@PersistenceUnit and @PersistenceContext annotations

<context:component-scan>

The main function of <context:component-scan> tag is to register the beans to the context and also scans the annotations in the beans and activate them.

In short what we can say is that <context:component-scan> does what <context:annotation-config> does as well as registers the beans to the context

<context:component-scan>=<context:annotation-config>+Bean Registration

Standard and Custom Events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. If a bean that implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified. Essentially, this is the standard Observer design pattern.

Spring provides the following standard events:

ContextRefreshedEvent
ContextStartedEvent
ContextStoppedEvent
ContextClosedEvent
RequestHandledEvent

```
<beans>
<import resource="services.xml"/>
<import resource="resources/messageSource.xml"/>
<import resource="/resources/themeSource.xml"/>
<bean id="bean1" class="..."/>
<bean id="bean2" class="..."/>
</beans>
```

Alfresco is an open source Enterprise Content Management (ECM) system that manages all the content within an enterprise and provides the services and controls that manage this content.

At the core of the Alfresco system is a repository supported by a server that persists content, metadata, associations, and full text indexes.

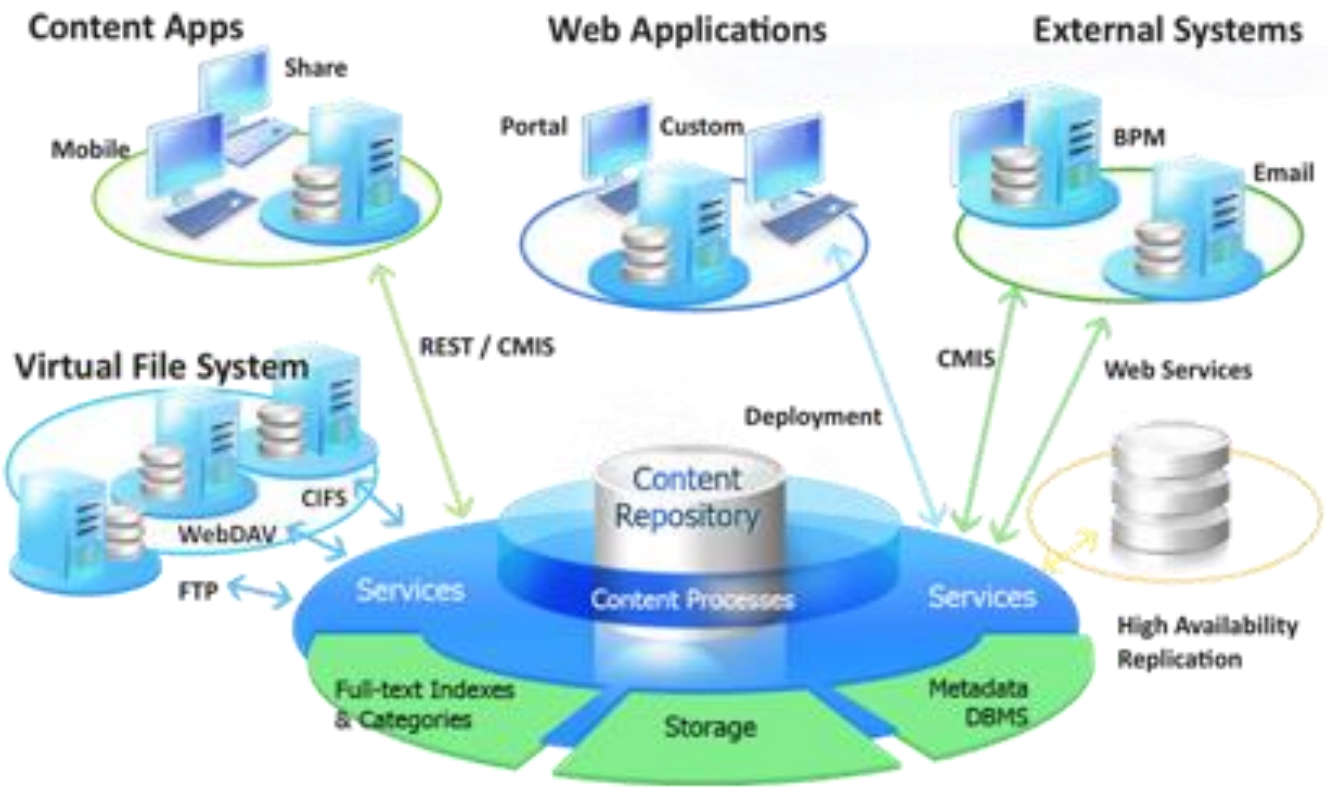
Alfresco provides Programming interfaces support multiple languages and protocols upon which developers can create custom applications and solutions.

Out-of-the-box applications provide standard solutions such as document management, records management, and web content management.

Alfresco is an entirely Java application, the Alfresco system runs on virtually any system that can run Java Enterprise Edition.

At the core is the Spring platform, providing the ability to modularize functionality, such as versioning, security, and rules.

Alfresco uses scripting to simplify adding new functionality and developing new programming interfaces. This portion of the architecture is known as web scripts and can be used for both data and presentation services. The lightweight architecture is easy to download, install, and deploy.


Content Apps
Share
Mobile
Web Applications
Portal
Custom
External Systems
BPM
Email
Virtual File System
CIFS
WebDAV
FTP
REST / CMIS
Deployment
CMIS
Web Services
Content Repository
Content Processes
Services
Services
High Availability Replication
Full-text Indexes & Categories
Storage
Metadata DBMS

web scripts provide a unique way to programmatically interact with the Alfresco content application server.

Unlike other interfaces exposed by Alfresco, web scripts offer a RESTful API for the content residing in the content repository.

The REST (Representational State Transfer) web architecture is based on HTTP requests and responses, URIs (Uniform Resource Identifiers), and document types.

Web scripts let you implement your own RESTful API without tooling or Java knowledge, requiring only a text editor or the Alfresco Explorer web client.

By focusing on the RESTful architectural style, web scripts let you build custom URI-identified and HTTP accessible content management web services backed by the Alfresco content application server.

web script description document for the Hello World example(hello.get.desc.xml)

```
<webscript>
  <shortname>Hello</shortname>
  <description>Polite greeting</description>
  <url>/hello</url>
</webscript>
```

web script response template to render the Hello World greeting
(hello.get.html.ftl)

```
<html>
<body>
  <p>Hello world!</p>
</body>
</html>
```

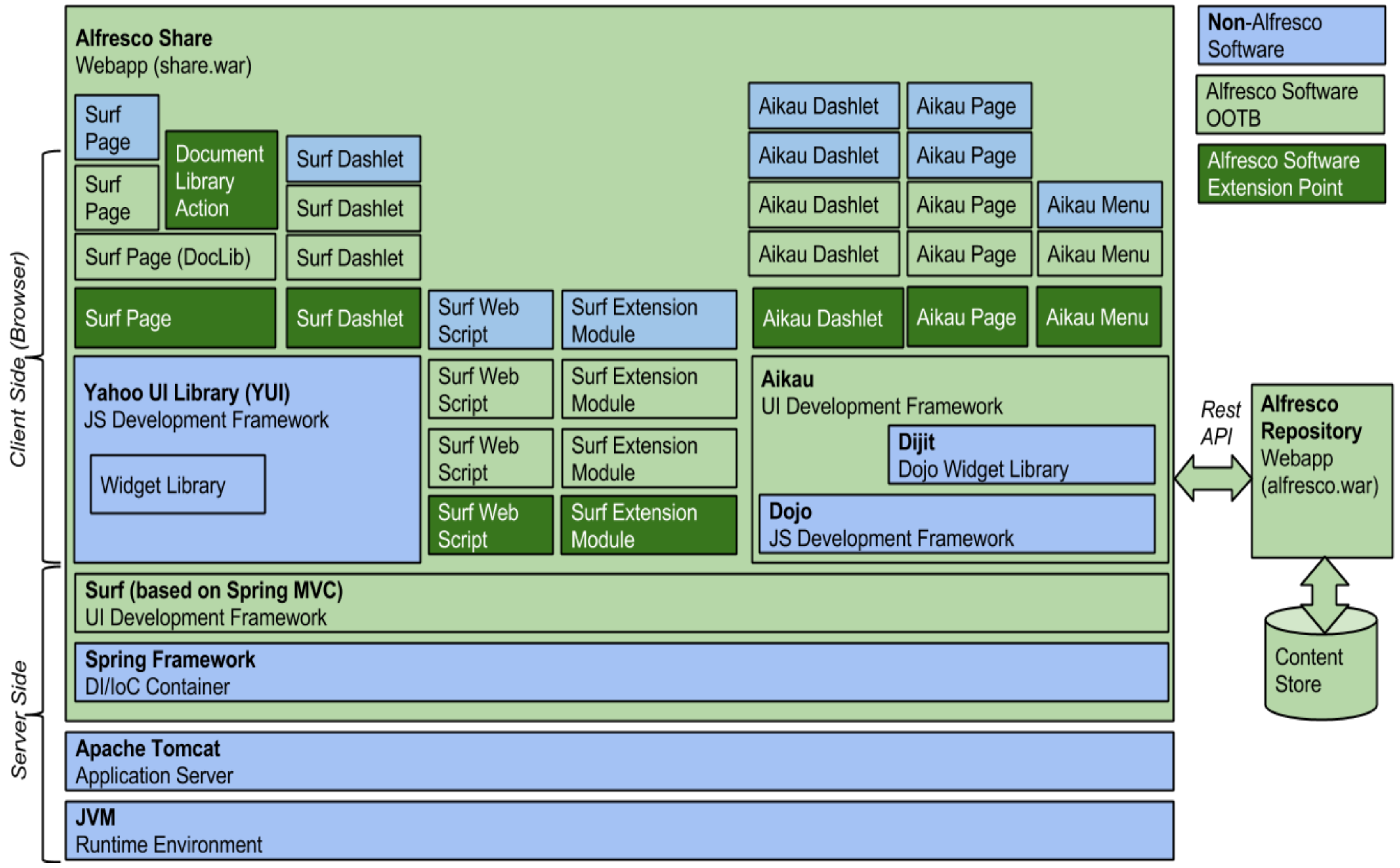
Type http://localhost:8080/alfresco/service/hello in the web browser to test the new web script.

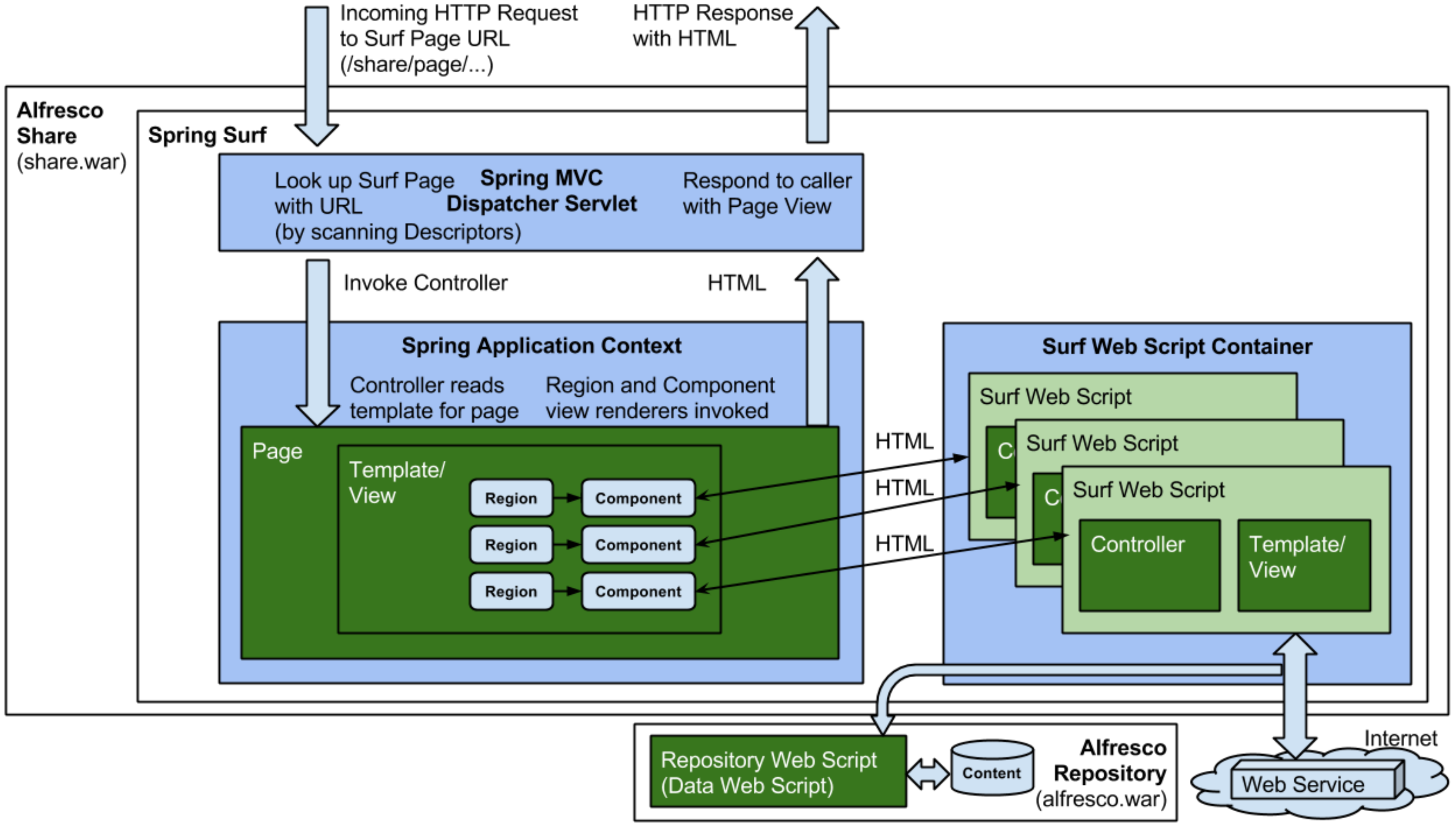
A Hello World message is displayed, indicating the web script is working.

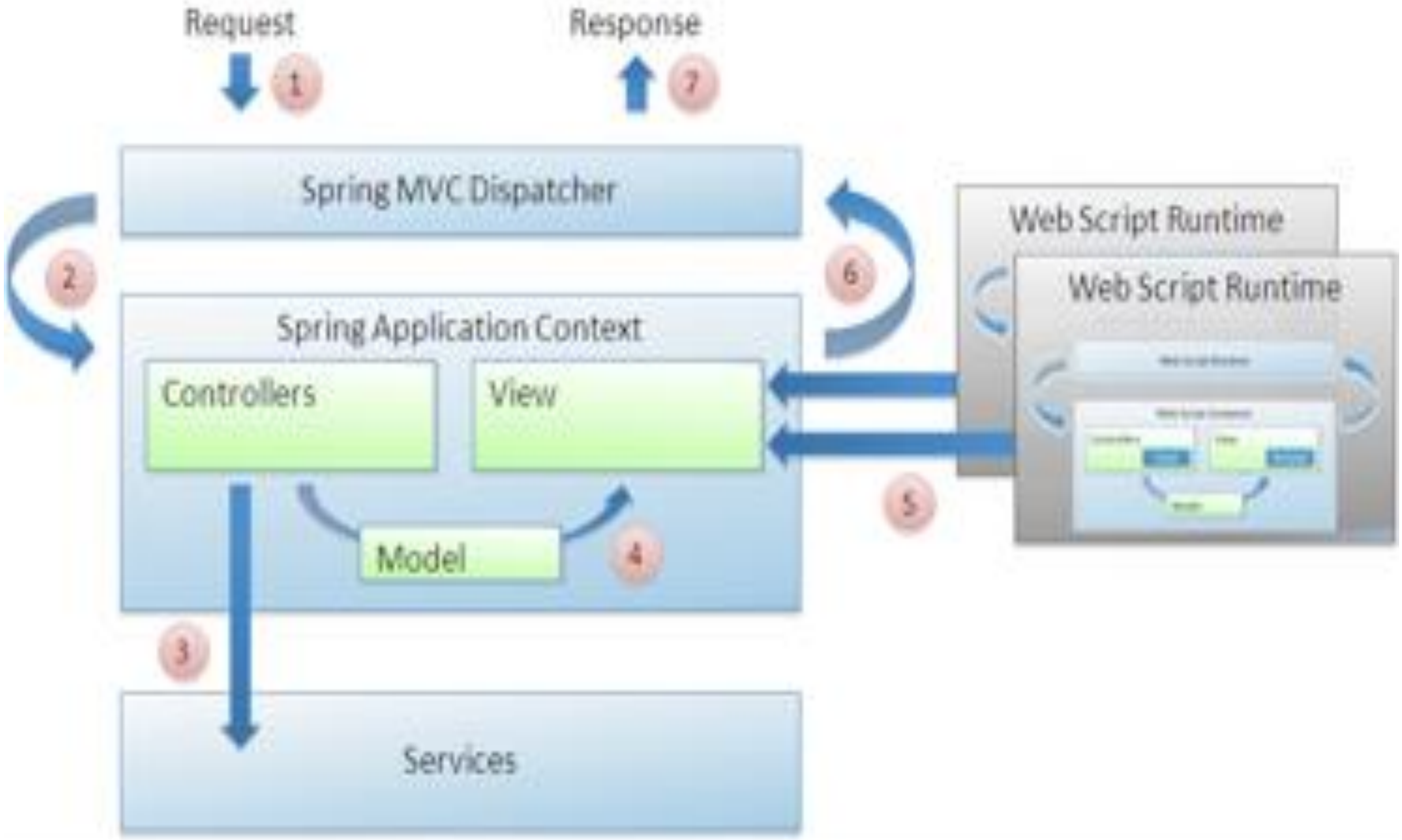
Surf lets us to build user interfaces for web applications using server-side scripts and templates without Java coding, recompilation, or server restarts.

Surf follows a content-driven approach, where scripts and templates are simple files on disk so that you can make changes to a live site in a text editor.

Surf is a Spring framework extension for building new Spring framework applications or plugging into existing Spring web MVC (Model, View, Controller) applications.


Alfresco Share
Webapp (share.war)
Surf Page
Surf Page
Document Library Action
Surf Dashlet
Surf Dashlet
Surf Page (DocLib)
Surf Dashlet
Surf Page
Surf Dashlet
Surf Web Script
Surf Extension Module
Aikau Dashlet
Aikau Page
Aikau Dashlet
Aikau Page
Aikau Dashlet
Aikau Page
Aikau Menu
Aikau Dashlet
Aikau Page
Aikau Menu
Aikau Dashlet
Aikau Page
Aikau Menu
Yahoo UI Library (YUI)
JS Development Framework
Widget Library
Surf Web Script
Surf Extension Module
Surf Web Script
Surf Extension Module
Surf Web Script
Surf Extension Module
Aikau
UI Development Framework
Dijit
Dojo Widget Library
Dojo
JS Development Framework
Surf (based on Spring MVC)
UI Development Framework
Spring Framework
DI/IoC Container
Apache Tomcat
Application Server
JVM
Runtime Environment
Client Side (Browser)
Server Side
Non-Alfresco Software
Alfresco Software OOTB
Alfresco Software Extension Point
Rest API
Alfresco Repository
Webapp (alfresco.war)
Content Store


Incoming HTTP Request
to Surf Page URL
(/share/page/...)
HTTP Response
with HTML
Alfresco
Share
(share.war)
Spring Surf
Look up Surf Page
with URL
(by scanning Descriptors)
Spring MVC
Dispatcher Servlet
Respond to caller
with Page View
Invoke Controller
HTML
Spring Application Context
Controller reads
template for page
Region and Component
view renderers invoked
Page
Template/
View
Region
Component
Region
Component
Region
Component
HTML
HTML
HTML
Surf Web Script Container
Surf Web Script
C
Surf Web Script
C
Surf Web Script
Controller
Template/
View
Repository Web Script
(Data Web Script)
Content
Alfresco
Repository
(alfresco.war)
Internet
Web Service


Request
1
Response
7
Spring MVC Dispatcher
2
Spring Application Context
Controllers
View
Model
4
3
Services
6
Web Script Runtime
Web Script Runtime
5

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum stuff.

Most Spring Boot applications need very little Spring configuration.
Features:

- ✓ Create stand-alone Spring applications
- ✓ Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- ✓ Provide opinionated 'starter' POMs to simplify your Maven configuration
- ✓ Automatically configure Spring whenever possible
- ✓ Provide production-ready features such as metrics, health checks and externalized configuration
- ✓ Absolutely no code generation and no requirement for XML configuration