Spring Framework Basics

Topics

- What is Spring framework?
- Why Spring framework?
- Spring framework architecture
- Usage scenario
- Dependency Injection (DI)
  - BeanFactory
  - Autowiring
  - ApplicationContext

Introduction to Spring Framework

Goal Of Spring Framework

- The Spring Framework Mission Statement
  - J2EE should be easier to use
  - It's best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero
  - JavaBeans offer a great way of configuring applications
  - OO design is more important than any implementation technology, such as J2EE
  - Checked exceptions are overused. A framework should not force to catch

What is Spring Framework? (1)

Light-weight yet comprehensive framework for building Java SE and Java EE applications

What is Spring Framework? (2)

- Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.
- Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.
- Examples of how you, as an application developer, can use the Spring platform advantage:
  - Make a Java method execute in a database transaction without having to deal with transaction APIs.
  - Make a local Java method a remote procedure without having to deal with remote APIs.
  - Make a local Java method a management operation without having to deal with JMX APIs.
  - Make a local Java method a message handler without having to deal with JMS APIs.

Overview Of Spring Framework

- The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications.
- Spring is modular, allowing you to use only those parts that you need, without having to bring in the rest.
  - You can use the **IoC** container, with Struts on top, but you can also use only the Hibernate integration code or the JDBC abstraction layer.
- The Spring Framework supports declarative transaction management, remote access to your logic through RMI or web services, and various options for persisting your data.
- It offers a full-featured **MVC** framework, and enables you to integrate **AOP** transparently into your software.

Overview Of Spring Framework

- Spring is designed to be non-intrusive, meaning that your domain logic code generally has no dependencies on the framework itself.
- In your integration layer (such as the data access layer), some dependencies on the data access technology and the Spring libraries will exist.
- However, it should be easy to isolate these dependencies from the rest of your code base

Key Features (1)

- JavaBeans-based configuration management, applying Inversion-of-Control principles, specifically using the Dependency Injection technique
  - This aims to reduce dependencies of components on specific implementations of other components.
- A core bean factory, which is usable globally
- Generic abstraction layer for database transaction management

Key Features (2)

- Built-in generic strategies for JTA and a single JDBC DataSource
  - This removes the dependency on a Java EE environment for transaction support.
- Integration with persistence frameworks Hibernate, JDO and iBATIS.
- MVC web application framework, built on core Spring functionality, supporting many technologies for generating views, including JSP, FreeMarker, Velocity, Tiles, iText, and POI.

Key Features (3)

- Extensive aspect-oriented programming (AOP) framework to provide services such as transaction management
  - As with the Inversion-of-Control parts of the system , this aims to improve the modularity of systems created using the framework.

Why Use Spring Framework?

Why Use Spring?

- Wiring of components through Dependency Injection
  - Promotes de-coupling among the parts that make the application
- Design to interfaces
  - Insulates a user of a functionality from implementation details
- Test-Driven Development (TDD)
  - POJO classes can be tested without being tied up with the framework

Why Use Spring?

- Declarative programming through AOP
  - Easily configured aspects, esp. transaction support
- Simplify use of popular technologies
  - Abstractions insulate application from specifics, eliminate redundant code
    - Handle common error conditions
    - Underlying technology specifics still accessible

Why Use Spring?

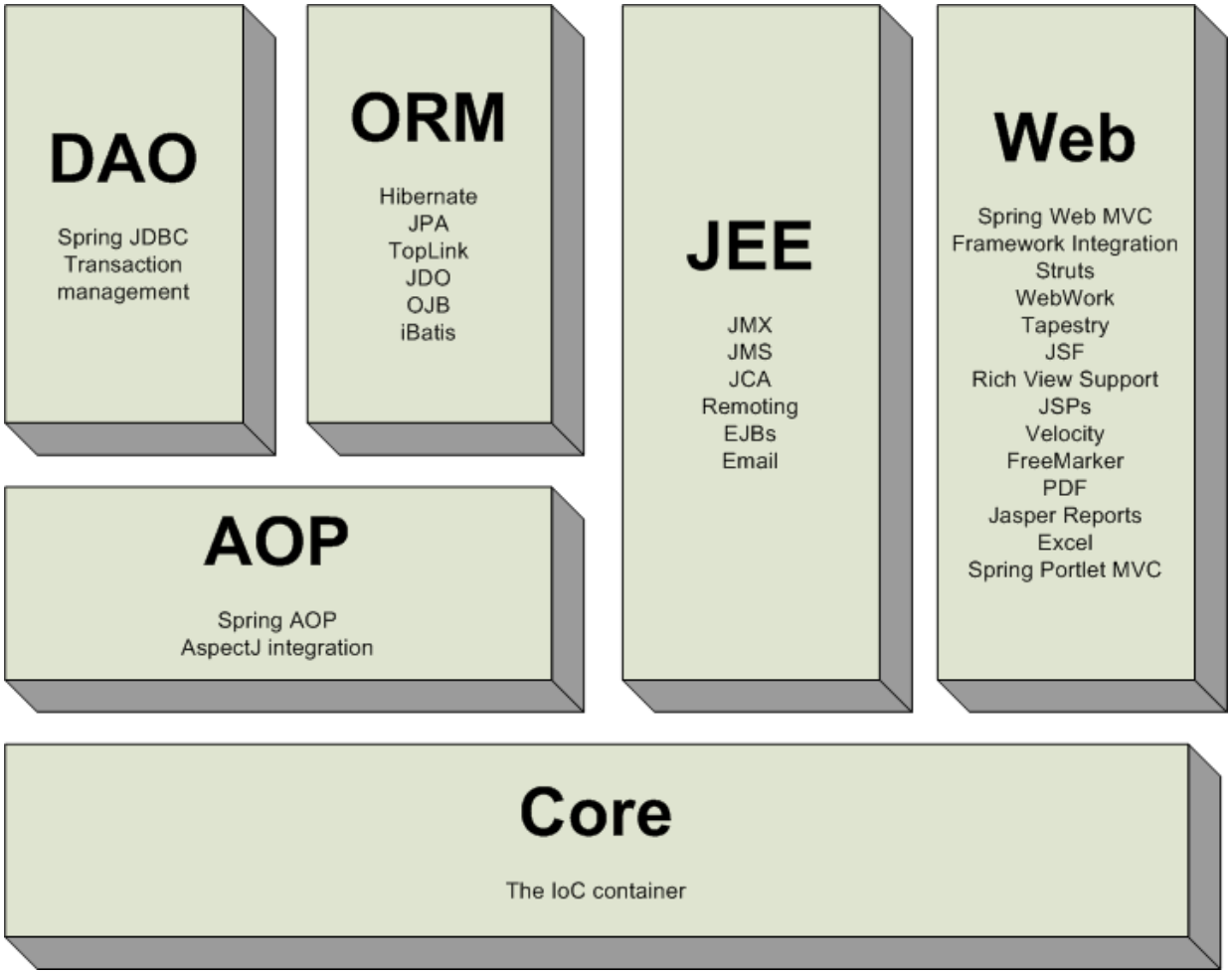
- Conversion of checked exceptions to unchecked
- Extremely modular and flexible
- Well designed
  - Easy to extend
  - Many reusable classes

Why Use Spring?

- Integration with other technologies
  - EJB for J2EE
  - Hibernate, iBates, JDBC (for data access)
  - Velocity (for presentation)
  - Struts and WebWork (For web)

Spring Framework Architecture

Spring Framework



Core Package

- Core package is the most fundamental part of the framework and provides the IoC and Dependency Injection features
- The basic concept here is the BeanFactory, which provides a sophisticated implementation of the factory pattern which removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic

DAO Package

- The DAO package provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes
- The JDBC package provides a way to do programmatic as well as declarative transaction management, not only for classes implementing special interfaces, but for all your POJOs (plain old Java objects)

ORM Package

- The ORM package provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- Using the ORM package you can use all those O/R-mappers in combination with all the other features Spring offers, such as the simple declarative transaction management feature mentioned previously

AOP Package

- Spring's AOP package provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated
- Using source-level metadata functionality you can also incorporate all kinds of behavioral information into your code

MVC Package

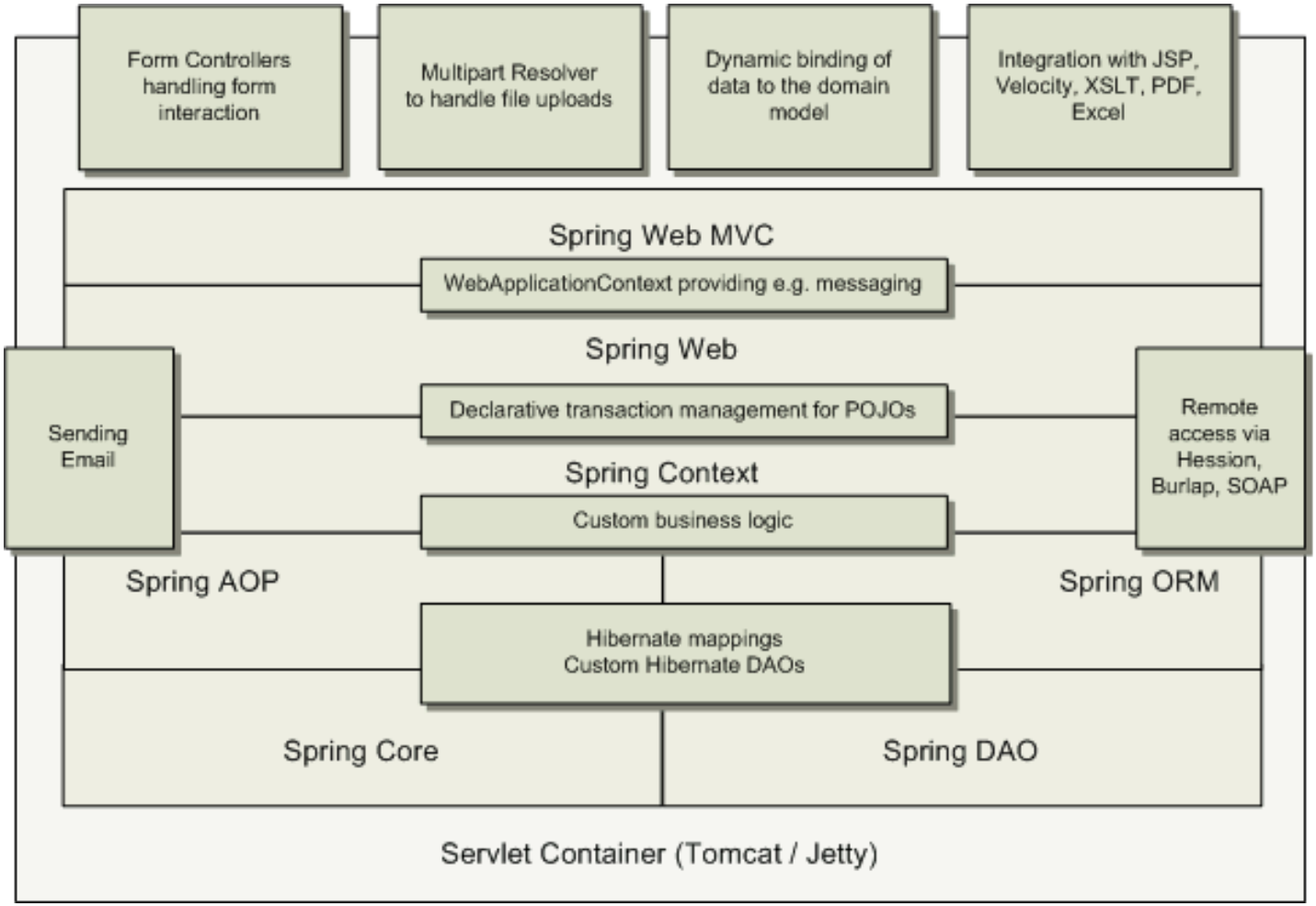
- Spring's MVC package provides a Model-View-Controller (MVC) implementation for webapplications
- Spring's MVC framework is not just any old implementation; it provides a clean separation between domain model code and web forms, and allows you to use all the other features of the Spring Framework.

Usage Scenarios

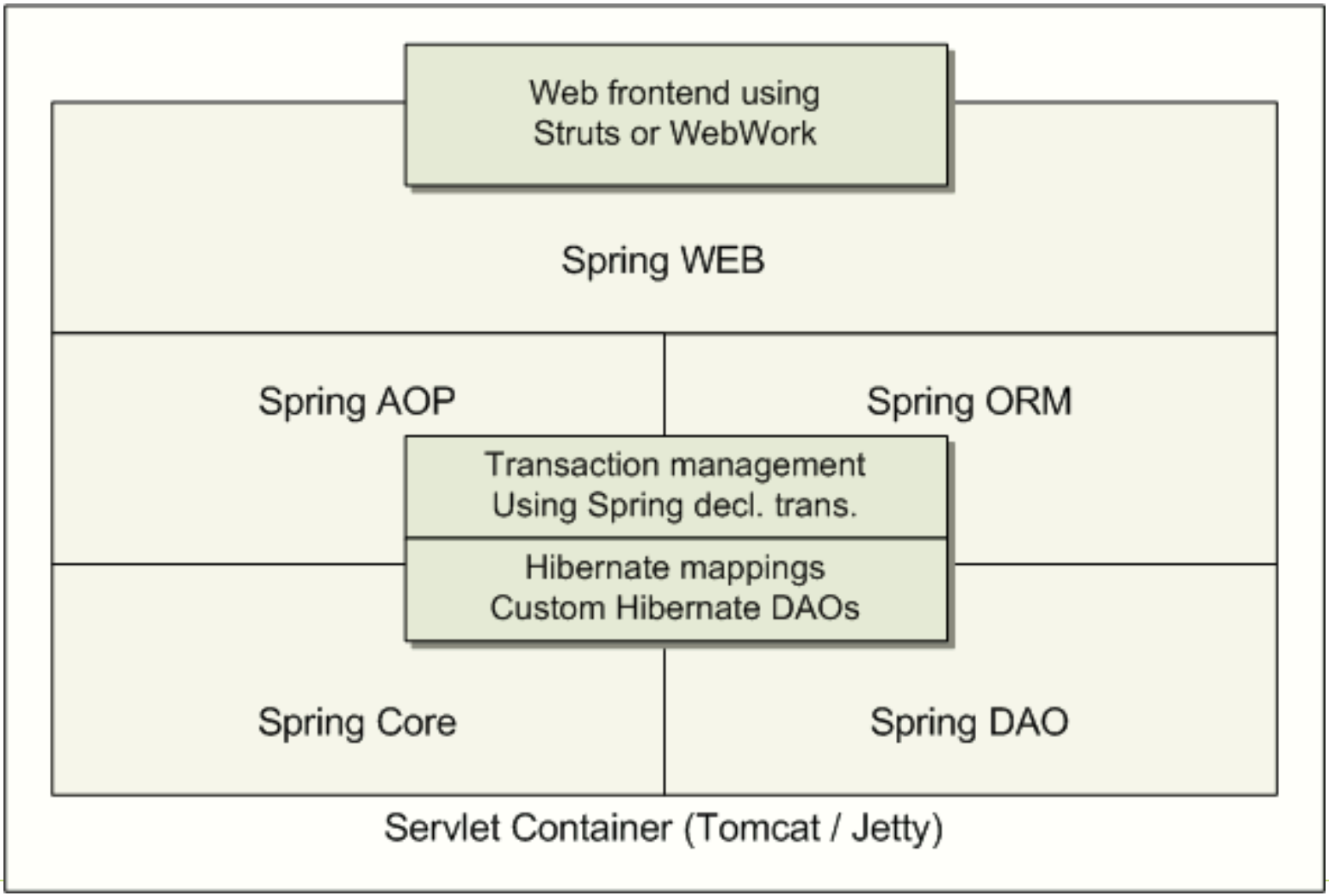
Usage Scenarios

- 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 integration

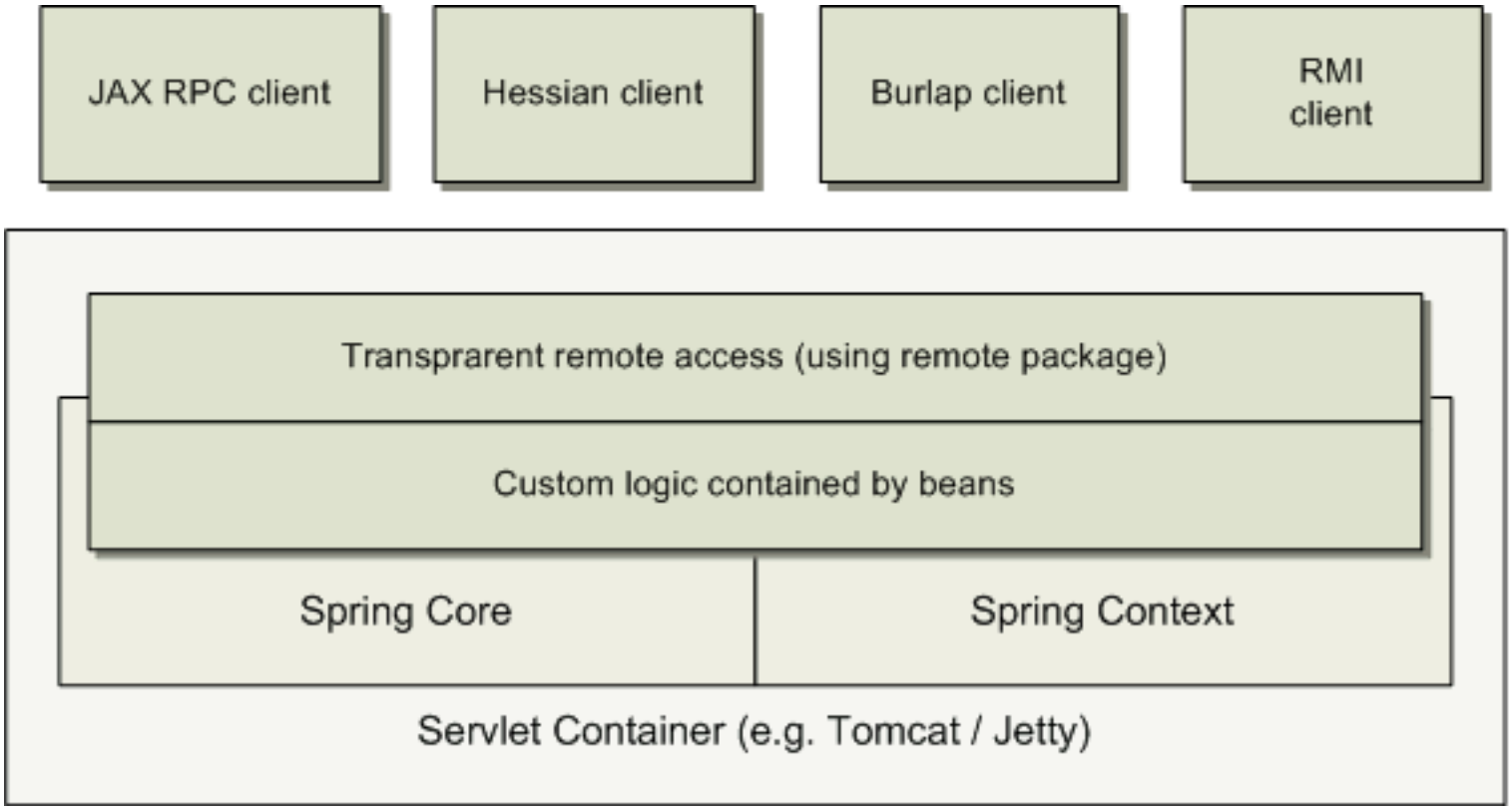
Typical Full-fledged Spring Web Application



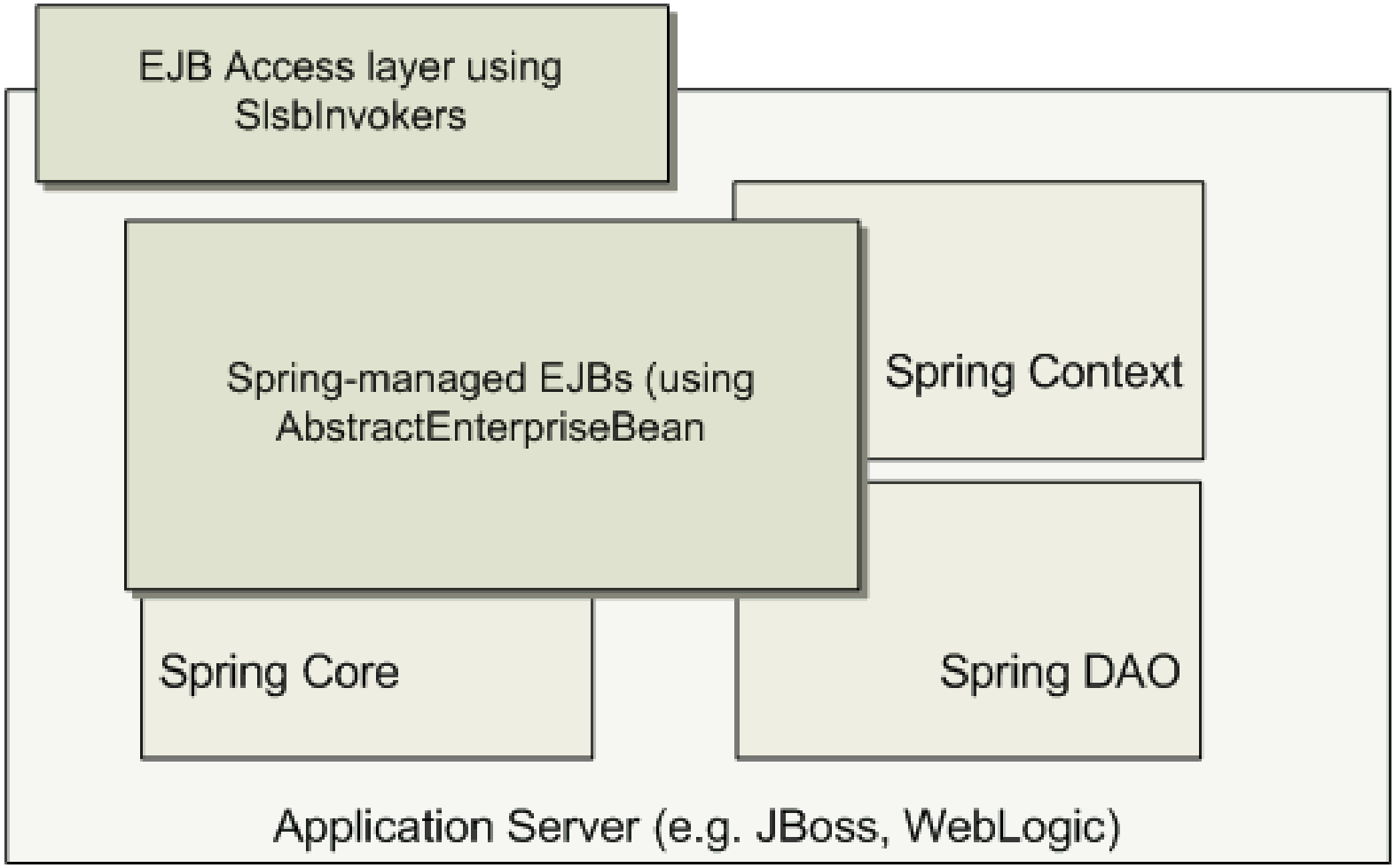
Spring Middle-tier Using 3rd party Web Framework



Remoting Usage Scenario



EJBs – Wrapping Existing POJOs



The IOC Container and Dependency Injection

Dependency Injection and IOC Container

- Java applications -- a loose term that runs the gamut from constrained applets to n-tier server-side enterprise applications -- typically consist of objects that collaborate to form the application proper.
- Thus the objects in an application have *dependencies* on each other
- Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers

Dependency Injection and IOC Container

- Architects and Developers can use design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* to compose the various classes and object instances that make up an application
- Patterns are formalized best practices that *you must implement yourself* in your application.
- The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use

Dependency Injection (DI): Basic concept

Spring Dependency Injection

- A kind of Inversion of Control (IoC)
- "Hollywood Principle"
  - Don't call me, I'll call you
- "Container" resolves (injects) dependencies of components by setting implementation object (push)
  - As opposed to component instantiating or Service Locator pattern where component locates implementation (pull)
- Martin Fowler calls Dependency Injection

Benefits of Dependency Injection

- Flexible
  - Avoid adding lookup code in business logic
- Testable
  - No need to depend on external resources or containers for testing
- Maintainable
  - Allows reuse in different application environments by changing configuration files instead of code
  - Promotes a consistent approach across all applications and teams

Two Dependency Injection Variants

- Constructor dependency Injection
  - Dependencies are provided through the constructors of the component
- Setter dependency injection
  - Dependencies are provided through the JavaBean style setter methods of the component
  - More popular than Constructor dependency injection

Constructor Dependency Injection

```
public class ConstructorInjection {

    private Dependency dep;

    public ConstructorInjection(Dependency dep) {

                this.dep = dep;

        }

    }
```

Setter Dependency Injection

```
public class SetterInjection {

    private Dependency dep;

    public void setMyDependency(Dependency dep) {

            this.dep = dep;

    }
}
```

Dependency Injection (DI): DI Support in Spring

Beans and Containers

Beans

- 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
  - there is nothing special about a bean (it is in all other respects one of probably many objects in your application).
- These beans, and the *dependencies* between them, are reflected in the *configuration metadata* used by a container

BeanFactory – The Container

- 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 beans.
- The BeanFactory interface is the central IoC container interface in Spring.
- Its responsibilities include instantiating or sourcing application objects, configuring such objects, and assembling the dependencies between these objects.

BeanFactory – Implementaions

- There are a number of implementations of the BeanFactory interface that come supplied straight out-of-the-box with Spring.
- The most commonly used BeanFactory implementation is the XmlBeanFactory class.
  - Convenience extension of DefaultListableBeanFactory
    - that reads bean definitions from an XML document
- The XmlBeanFactory takes this XML *configuration metadata* and uses it to create a fully configured system or application.

BeanFactory – The Container

- BeanFactory object is responsible for managing beans and their dependencies
- Your application interacts with Spring's DI container through BeanFactory interface
  - BeanFactory object has to be created by the application typically XmlBeanFactory
  - BeanFactory object, when it gets created, read bean configuration file and performs the wiring
  - Once created, the application can access the beans via BeanFactory interface

Reading XML Configuration File via *XmlBeanFactory* class

```
public class XmlConfigWithBeanFactory {

public static void main(String[] args) {

XmlBeanFactory factory =

new XmlBeanFactory(new
    FileSystemResource("beans.xml"));

SomeBeanInterface b =

    (SomeBeanInterface)
    factory.getBean("nameOftheBean");

    }

}
```

Bean Configuration File

- Each bean is defined using *<bean>* tag under the root of the <beans> tag
- The *id* attribute is used to give the bean its default name
- The *class* attribute specifies the type of the bean

Bean Configuration File Example

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="renderer" class="StandardOutMessageRenderer">
        <property name="messageProvider">
                <ref local="provider"/>
        </property>
    </bean>
    <bean id="provider" class="HelloWorldMessageProvider"/>
</beans>
```

Wiring a Bean

Beans

- The term “bean” is used to refer any component managed by the BeanFactory
- The “beans” are in the form of JavaBeans (in most cases)
  - no arg constructor
  - getter and setter methods for the properties
- Beans are singletons by default
- Properties the beans may be simple values or references to other beans
- Beans can have multiple names

What is Wiring?

- The act of creating associations between application components is referred to as wiring
- There are many ways to wire a bean but common approach is via XML

Wiring example

```
<?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="greetBean“       class="GreetingServiceImpl">
      <property name="greeting">
      <value>Hello friends of Spring</value>
      </property>
    </bean>
 </beans>
```

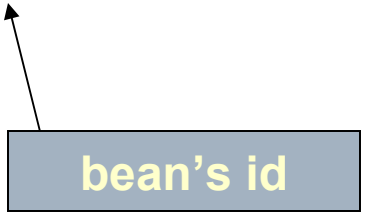
Wiring the beans

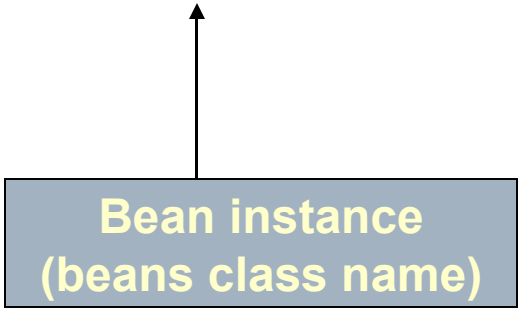
- Spring beans can be driven from any configuration
  - Properties files
  - Relational database
  - an LDAP
- Preferred Choice for configuration is XML
- Several Spring containers support wiring through xml
  - XmlBeanFactory
  - ClasspathXmlApplicationContext
  - FileSystemApplicationContext
  - XmlWebApplicationContext

Wiring the beans

```
<?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="mybean"       class="com.jp.TestBean">

   </bean>

 </beans>
```





Wiring the beans

- Prototype and Singleton beans
  - all spring beans are singleton
  - but prototype beans can also be defined

```
<bean id =“myBean” class=“com.jp.TestBean”
                singleton=“false”/>
```

  - singleton = “false” returns a prototype bean
  - singleton = “true” returns a singleton bean
  - default value for “singleton” is “true”

- In Spring Framework Version 2.x the configuration for scope is

```
<bean id =“myBean” class=“com.jp.TestBean”
                                        scope=“singletone”/>
```

  Scope attribute has values:

  1. Singleton
  2. Prototype
  3. Request
  4. session

Wiring the beans

- Initialization and Destruction
  - beans can be initialized and destroyed by calling bean specific methods
    - init-method : calls bean specific initialization method
    - destroy-method : calls bean specific cleanup method

Wiring the beans

- **Initialization and Destruction (example)**

```
public class MyConnectionPool {
    public void initialize(){
        //initialize a connection;
    }
    public void cleanup() {
    //release connection;
    }
}
```

- **configuration:**

```
<bean id="myBean" class ="com.jp.MyConnectionPool"
    init-method="initialize" destroy-method="cleanup" />
```

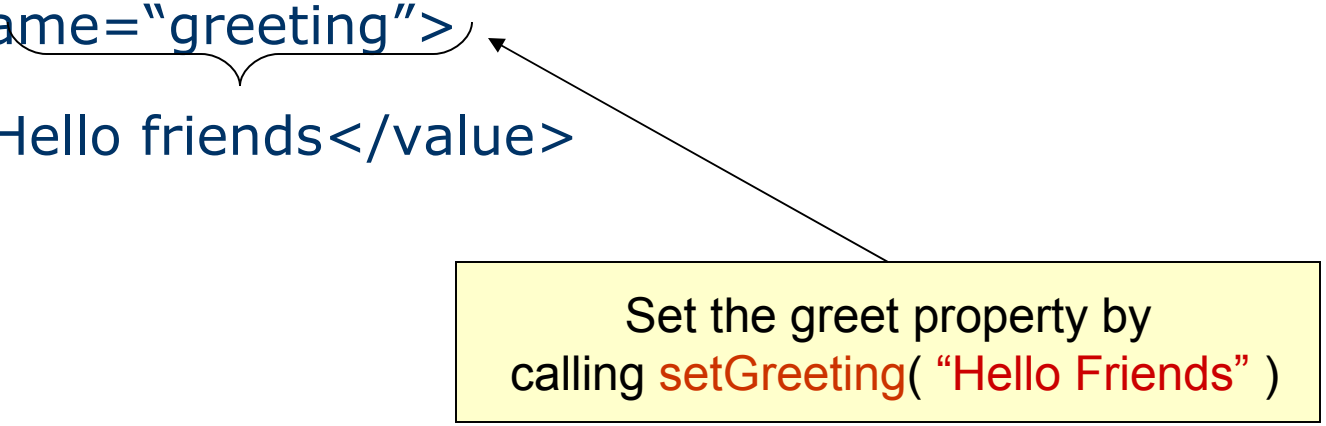
Spring Dependency Injection *Revisited*

- Two types of Dependency Injection
  - setter injection
    - dependency injected via setter methods
  - constructor injection
    - dependency injected via constructor

Spring Dependency Injection

- Setter Injection

```
<bean id="test" class="com.jp.TestBean">
<property name="greeting">
    <value>Hello friends</value>
</property>
</bean>
```



Spring Dependency Injection

- Referencing other beans

```
<beans>
    <bean id="test" class ="com.jp.TestBean">
    <property name="greeting">
            <ref bean="greetBean"/>
    </property>

<bean id="greetBean" class ="com.jp.GreetBean" />
<beans>
```

Spring Dependency Injection

- Constructor Injection

```
<bean id="test" class="com.jp.testBean">
<constructor-arg>
    <value>Hello friends</value>
</constructor-arg>
</bean>
```

```
<bean id="test" class="com.jp.testBean">
<constructor-arg>
    <ref bean="greetBean"/>
</constructor-arg>
</bean>
```

constructs a **TestBean** object through its constructor

Wiring Collections

- Spring supports Many types of Collections as bean properties
- Supported types are:

XML	Types
<list>	java.util.List, arrays
<set>	java.util.Set
<map>	java.util.Map
<props>	java.util.Properties

Wring Lists and Arrays

```
<property name="testList">
    <list>
        <value>value1</value>
        <value>value2</value>
    </list>
</property>
```

```
<bean id="exampleSessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource"><ref local="exampleDataSource"/>
        <property name="hibernateProperties">
                <ref bean="exampleHibernateProperties" />
        </property>
        <property name="mappingResources">
                <list>
                 <value>Customer.hbm.xml</value>
                <value>Account.hbm.xml</value>
                </list>
        </property>
    </bean>
```

Wring Set and Map, Properties

```
<property name="testSet">
      <set>
      <value>value1</value>
      <value>value2</value>
      </set>
</property>
```

```
<property name="barMap">
<map>
            <entry key="key1">
                <value>value1</value>
            </entry>
             <entry key="key1">
             <value>value1</value>
            </entry>
</map>
</property>
```

```
<property name="barProperty">
            <props>
                        <prop key="key1">value1</prop>
                        <prop key="key2">value2</prop>
                        <prop key="key3">value3</prop>
            </props>
</property>
```

Dependency Injection: Autowiring

Auto Wiring

- So far we wired beans explicitly using <property> tag
- Spring can also do Wiring automatically

```
<bean id="foo" class="com.jp.spring.Foo“
        autowire= "autowire type"/>
```

Autowiring Properties

- Beans may be auto-wired (rather than using <ref>)
  - Per-bean attribute *autowire*
  - *Explicit settings override*
- *autowire="byName"*
  - Bean identifier matches property name
- *autowire="byType"*
  - Type matches other defined bean
- *autowire="constructor"*
  - Match constructor argument types
- *autowire="autodetect"*
  - Attempt by constructor, otherwise "type"
- Autowire="no"
  - no autowire is allowed

Bean Naming

- Each bean must have at least one name that is unique within the containing BeanFactory
- Name resolution procedure
  - If a <bean> tag has an id attribute, the value of the id attribute is used as the name
  - If there is no id attribute, Spring looks for name attribute
  - If neither id nor name attribute are defined, Spring use the class name as the name
- A bean can have multiple names
  - Specify comma or semicolon-separated list of names in the name attribute

Bean Naming Example

```
<bean id="mybeanid" class="mypackage.MyClass"/>

<bean name="mybeanname" class="mypackage.MyClass"/>

<bean class="mypackage.MyClass"/>

<bean id="mybeanid" name="name1,name2,name3"
class="mypackage.MyClass"/>
```

ApplicationContext

What is ApplicationContext?

- Extension of BeanFactory
  - It provides all the same functionality and more
  - Reduces the amount of code you need
  - In a more framework-oriented style
- Add new features over BeanFactory
  - Resource management and access
  - Additional life-cycle interfaces
  - Improved automatic configuration of infrastructure components
  - Event publication
  - Internationalization

Important Application Contexts

- ClassPathXmlApplicationContext
- WebApplicationContext
- FileSystemApplicationContext

When to Use ApplicationContext?

- Use **ApplicationContext** over ***BeanFactory*** to take advantage of its extended functionality
  - Except for a few limited situations such as perhaps in an Applet, where memory consumption might be critical, and a few extra kilobytes might make a difference

Using MessageSource

- The ApplicationContext interface extends an interface called MessageSource, and therefore provides messaging (i18n or internationalization)functionality

```
<beans>
<bean id="messageSource”
class="org.springframework.context.support.ResourceBundleMessageSource">
<property name="basenames">
    <list>
        <value>format</value>
        <value>exceptions</value>
        <value>windows</value>
    </list>
</property>
</bean>
</beans>
```

Propagating Events

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

Three Built-in Events

- ContextRefreshEvent
  - ApplicationContext is initialized or refreshed
- ContextClosedEvent
  - ApplicationContext is closed
- RequestHandleEvent
  - A web-specific event telling all beans that a HTTP request has been serviced

Example: Event Handling

Configuration

```
<bean id="emailer" class="example.EmailBean">
<property name="blackList">
<list>
<value>black@list.org</value>
<value>white@list.org</value>
<value>john@doe.org</value>
</list>
</property>
</bean>
<bean id="blackListListener" class="example.BlackListNotifier">
<property name="notificationAddress" value="spam@list.org"/>
</bean>
```

Example: Event Handling

```
public class BlackListEvent extends ApplicationEvent{
String address;
String text;

// getters and setters


public BlackListEvent(String address, String text) {
super( address);  //Required !!!
this.address = address;
this.text = text;
}
}
```

Example: Event Handling

- Bean class

```
public class EmailBean implements ApplicationContextAware {
/** the blacklist */
private List blackList;
public void setBlackList(List blackList) {
this.blackList = blackList;
}
        public void setApplicationContext(ApplicationContext ctx) {this.ctx = ctx;}
        public void sendEmail(String address, String text) {
                if (blackList.contains(address)) {
                BlackListEvent evt = new BlackListEvent(address, text);
                ctx.publishEvent(evt);
                return;
                }
        }
}
```

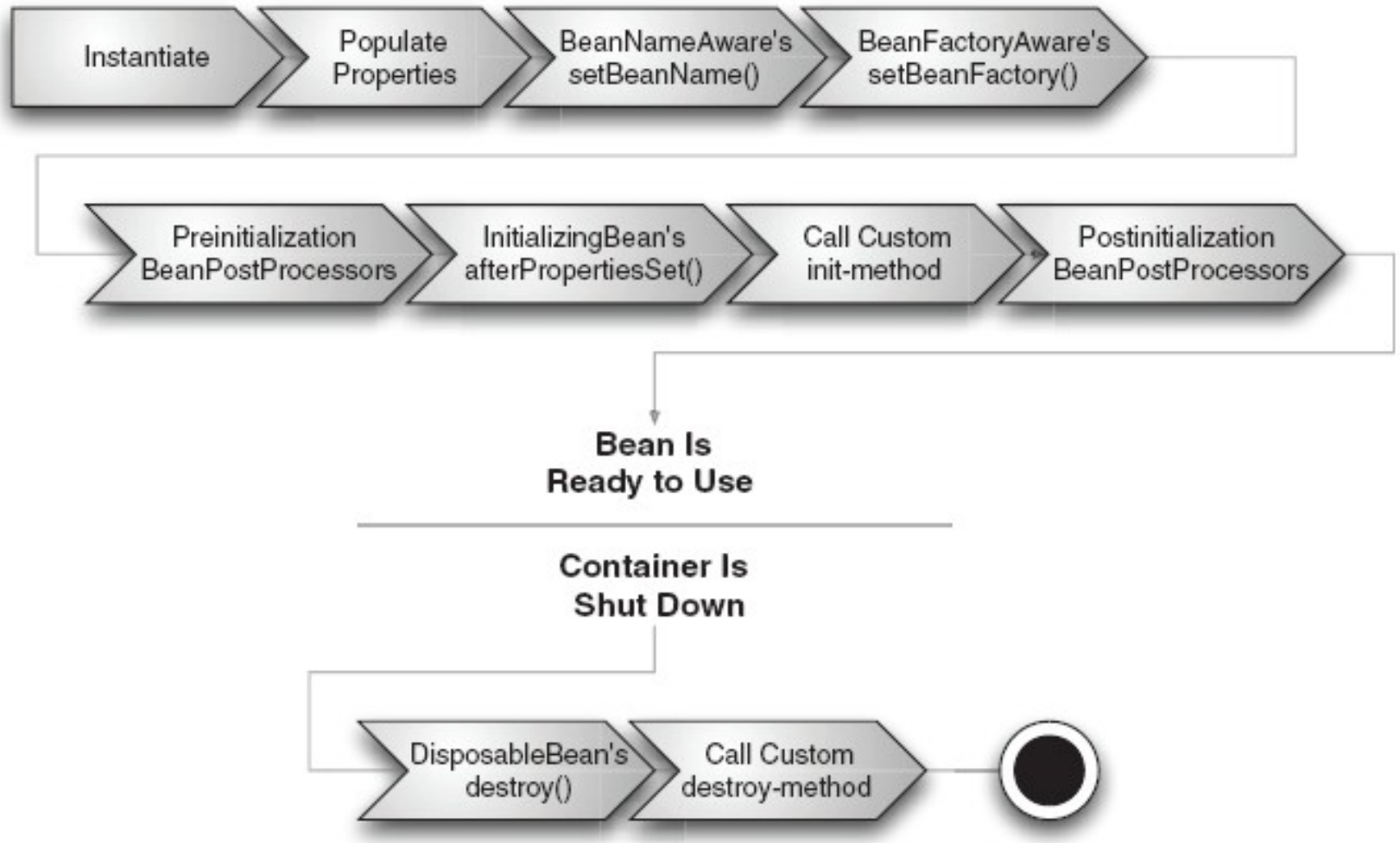
Example: Event Handling

- Notifier class

```
public class BlackListNotifier implement ApplicationListener {
/** notification address */
private String notificationAddress;
      public void setNotificationAddress(String notificationAddress) {
            this.notificationAddress = notificationAddress;
      }
public void onApplicationEvent(ApplicationEvent evt) {
            if (evt instanceof BlackListEvent) {
                        // notify appropriate person
            }
      }
}
```

Life Cycle of a Bean

Life cycle of a bean in BeanFactory Container



The lifecycle of a bean within a Spring application context