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

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.

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.

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?

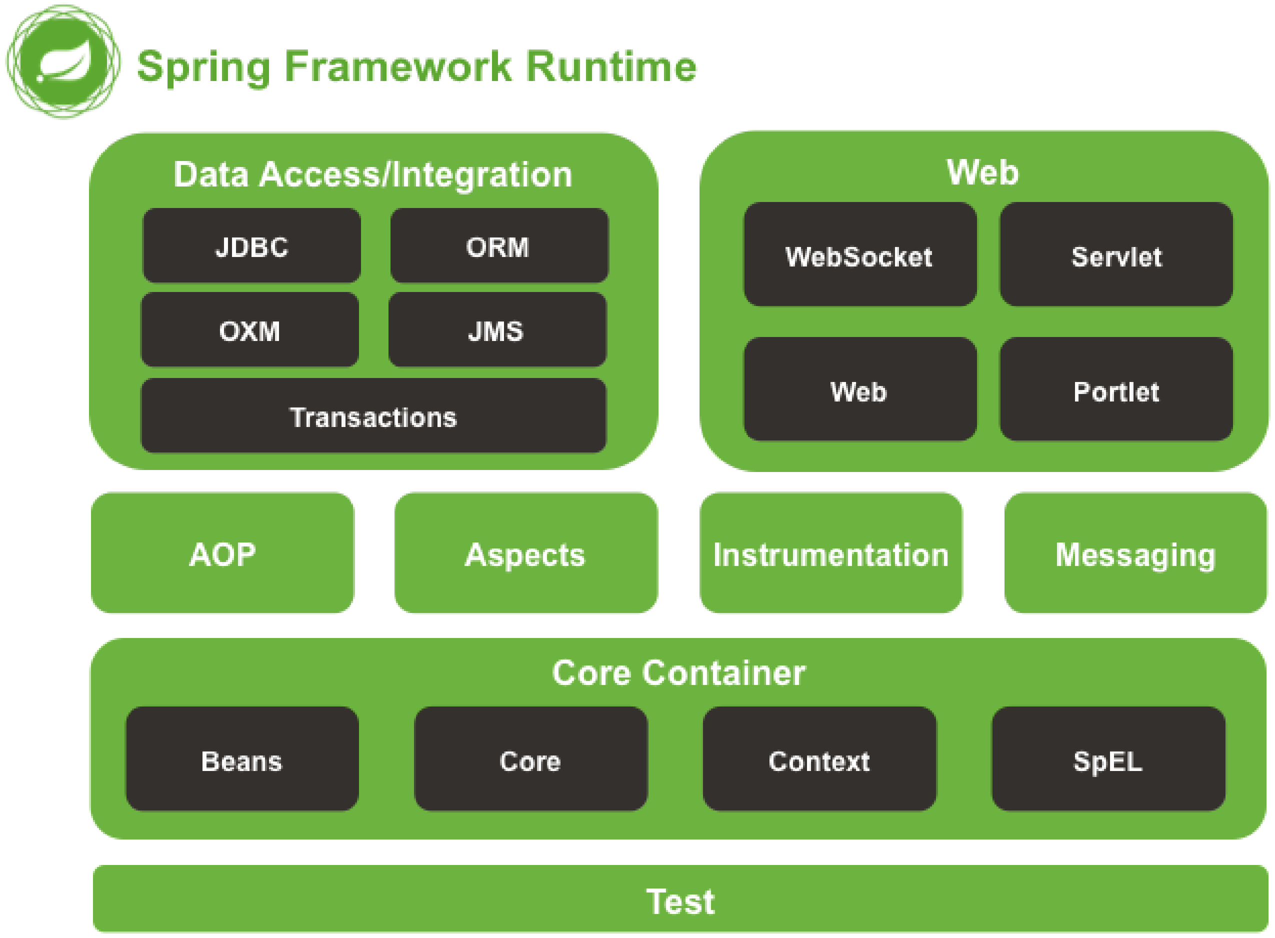
- Conversion of checked exceptions to unchecked
- Extremely modular and flexible
- Well designed
  - Easy to extend
  - Many reusable classes

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 Modules

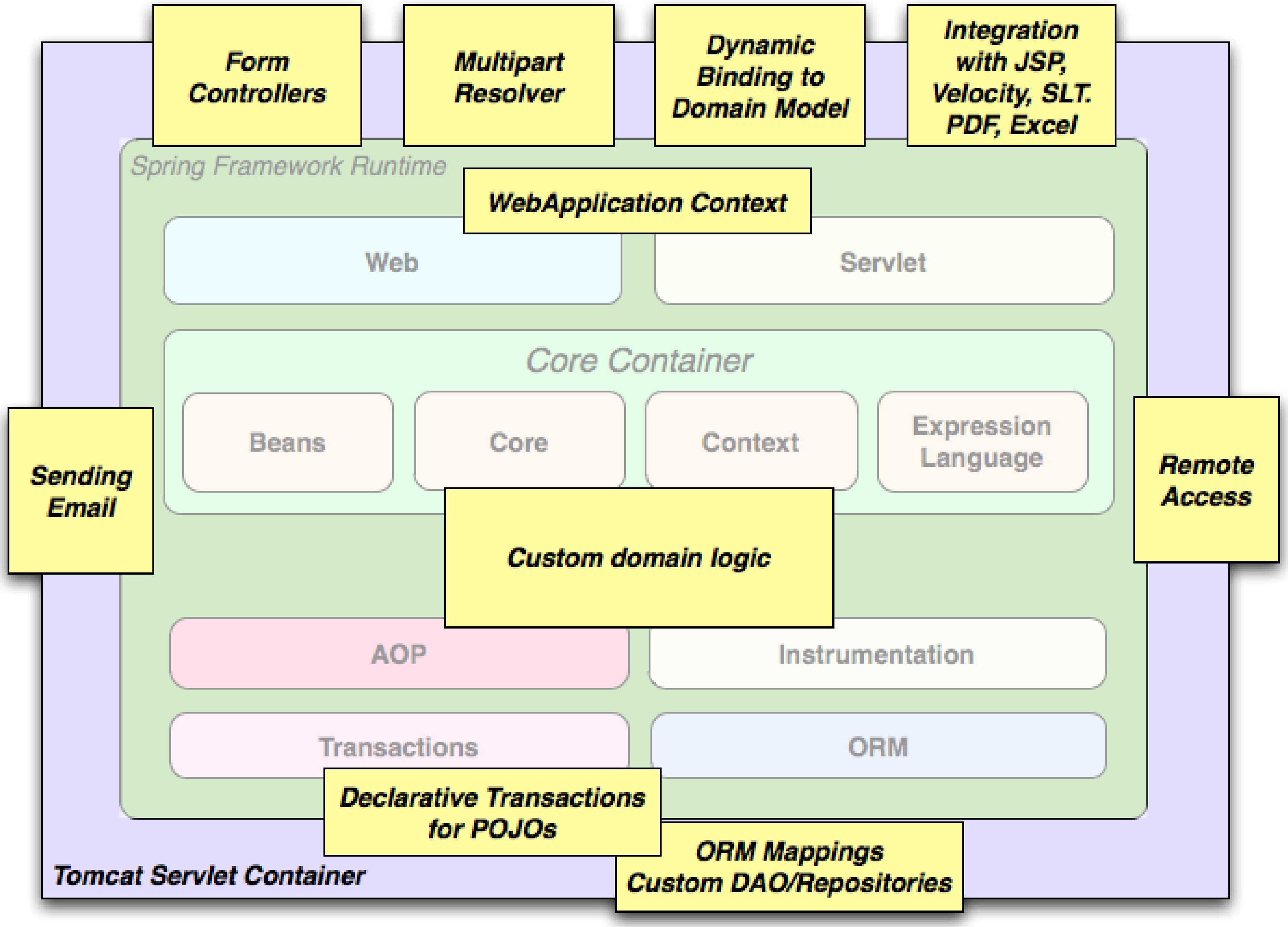


Usage Scenarios

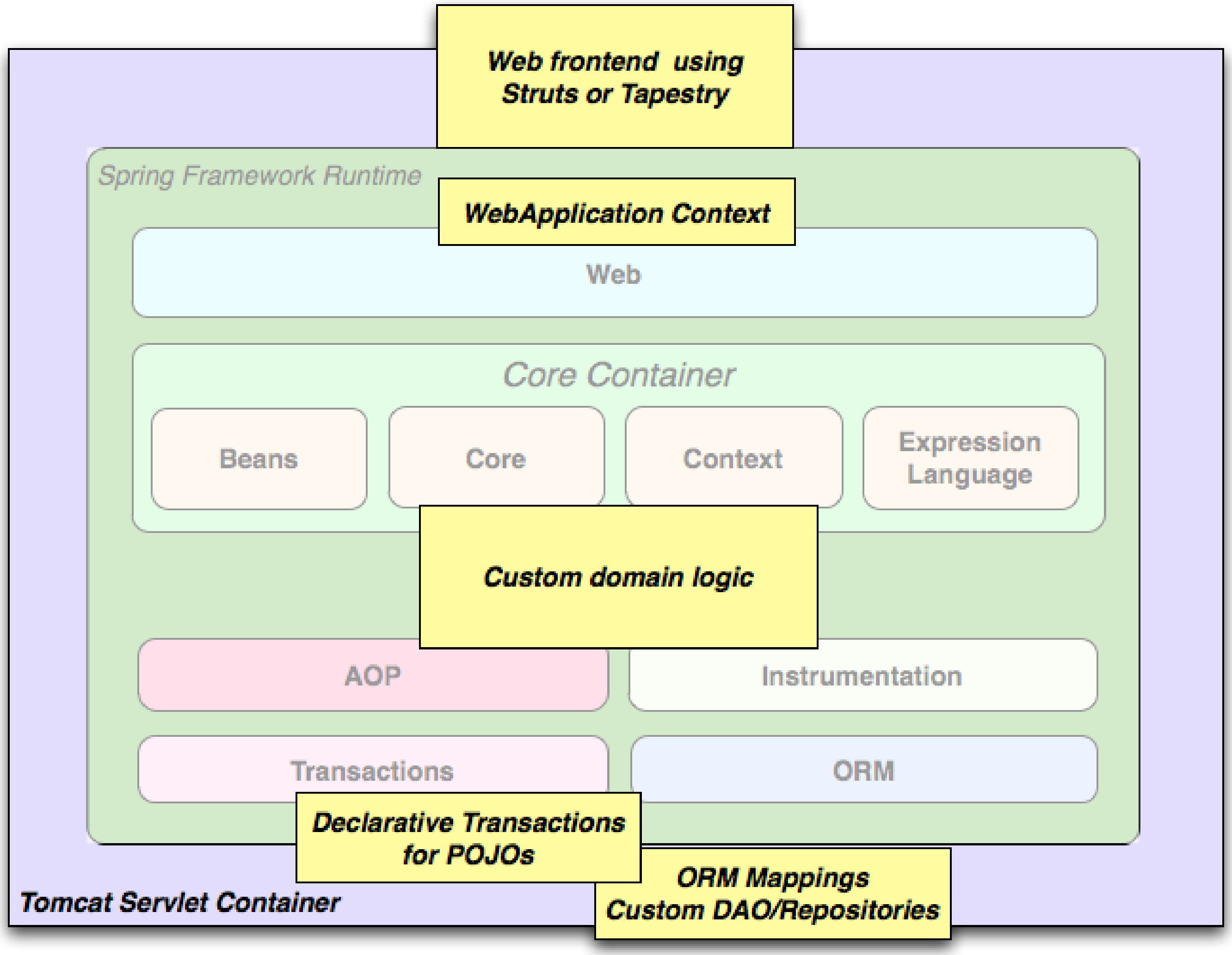
Usage Scenarios

- 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

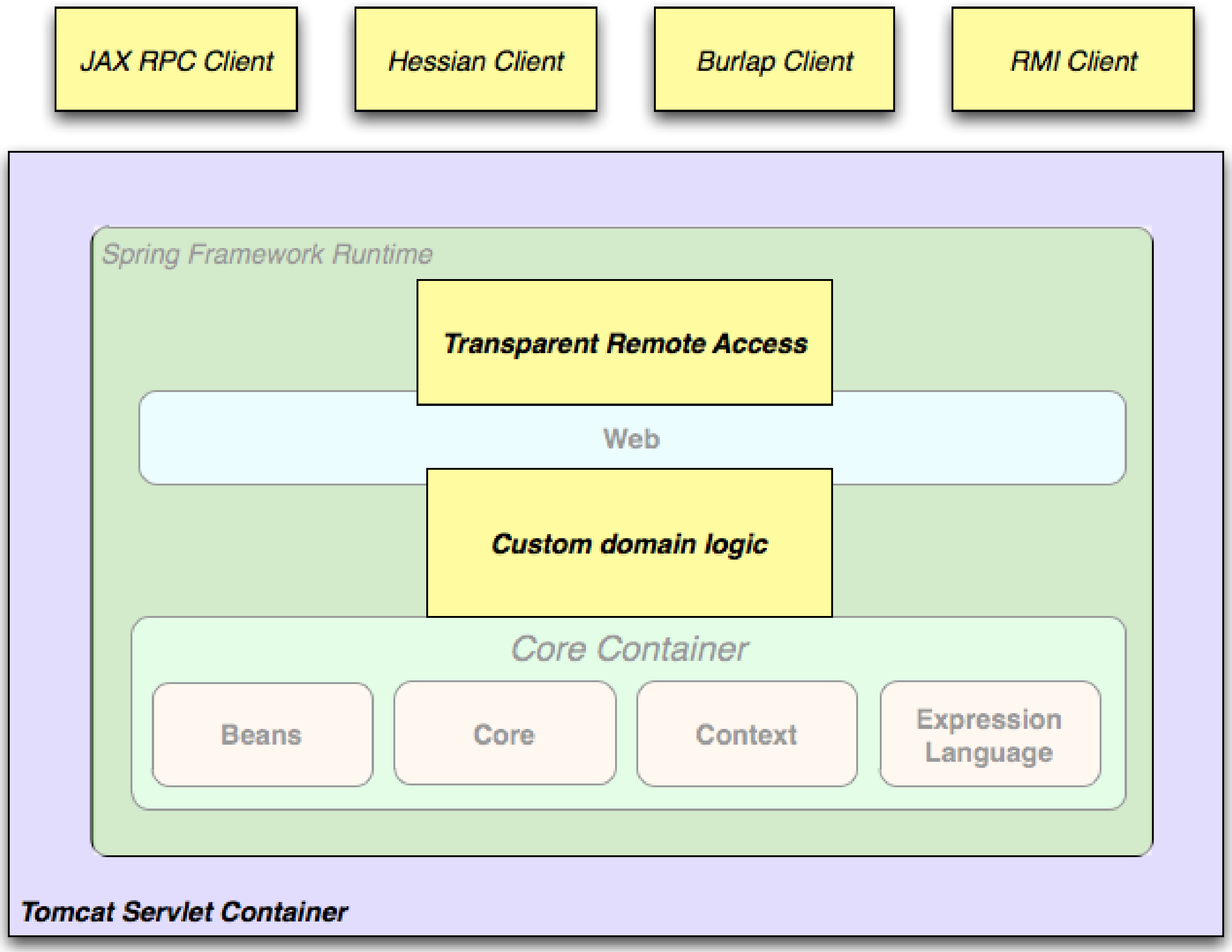
Typical Full-fledged Spring Web Application



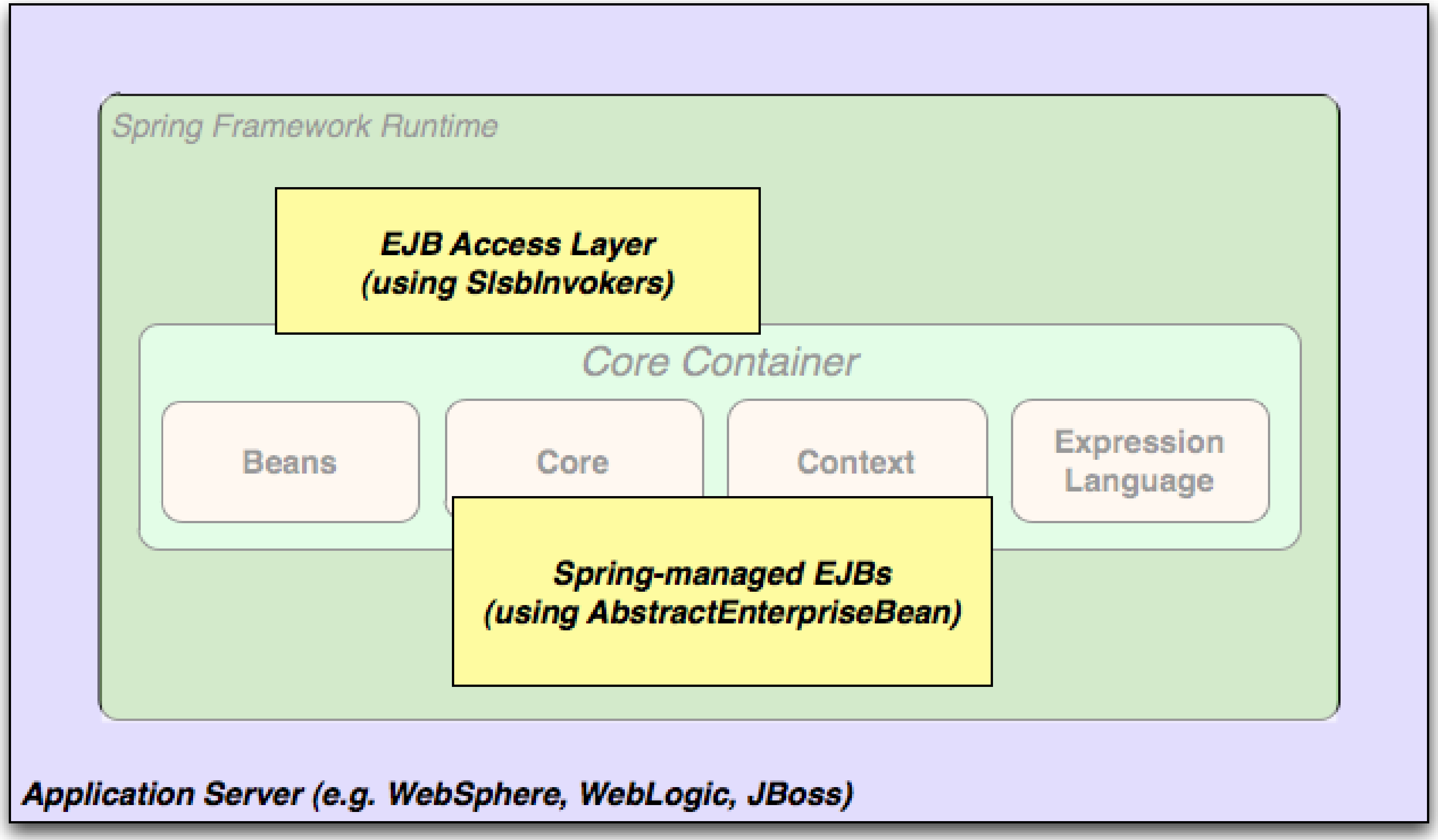
Spring Middle-tier Using 3rd party Web Framework



Remoting Usage Scenario



EJBs – Wrapping Existing POJOs


Spring Framework Runtime
EJB Access Layer (using SlsbInvokers)
Core Container
Beans
Core
Context
Expression Language
Spring-managed EJBs (using AbstractEnterpriseBean)
Application Server (e.g. WebSphere, WebLogic, JBoss)

The IOC Container and Dependency Injection

Dependency Injection (DI): Basic concept

Spring Dependency Injection

- A kind of Inversion of Control (IoC)
- “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

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  bean="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

- 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
5. Global 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

Inner Beans

- A <bean/> element inside the <property/> or <constructor-arg/> elements is used to define a so-called *inner bean*.

```
<bean id="outer" class="...">

  <property name="target">
    <bean class="com.example.Person">
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

- Inner beans are always anonymous and they are always scoped as prototypes.
- Please also note that it is not possible to inject inner beans into collaborating beans other than the enclosing bean.

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.ResourceBundleMessageSour
    ce">
<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

```java
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;
                }
        }
}
```

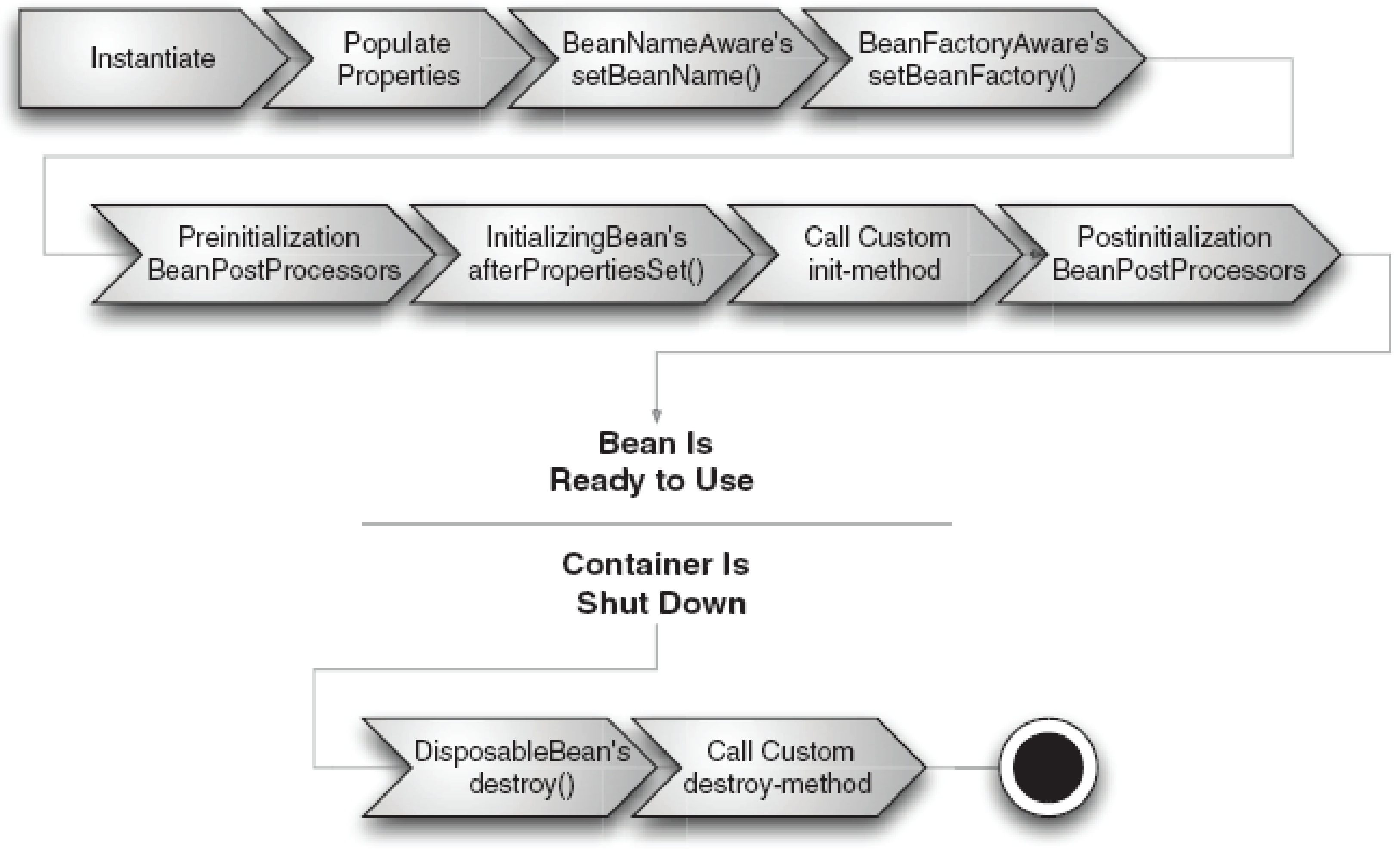
Example: Event Handling

- Notifier class

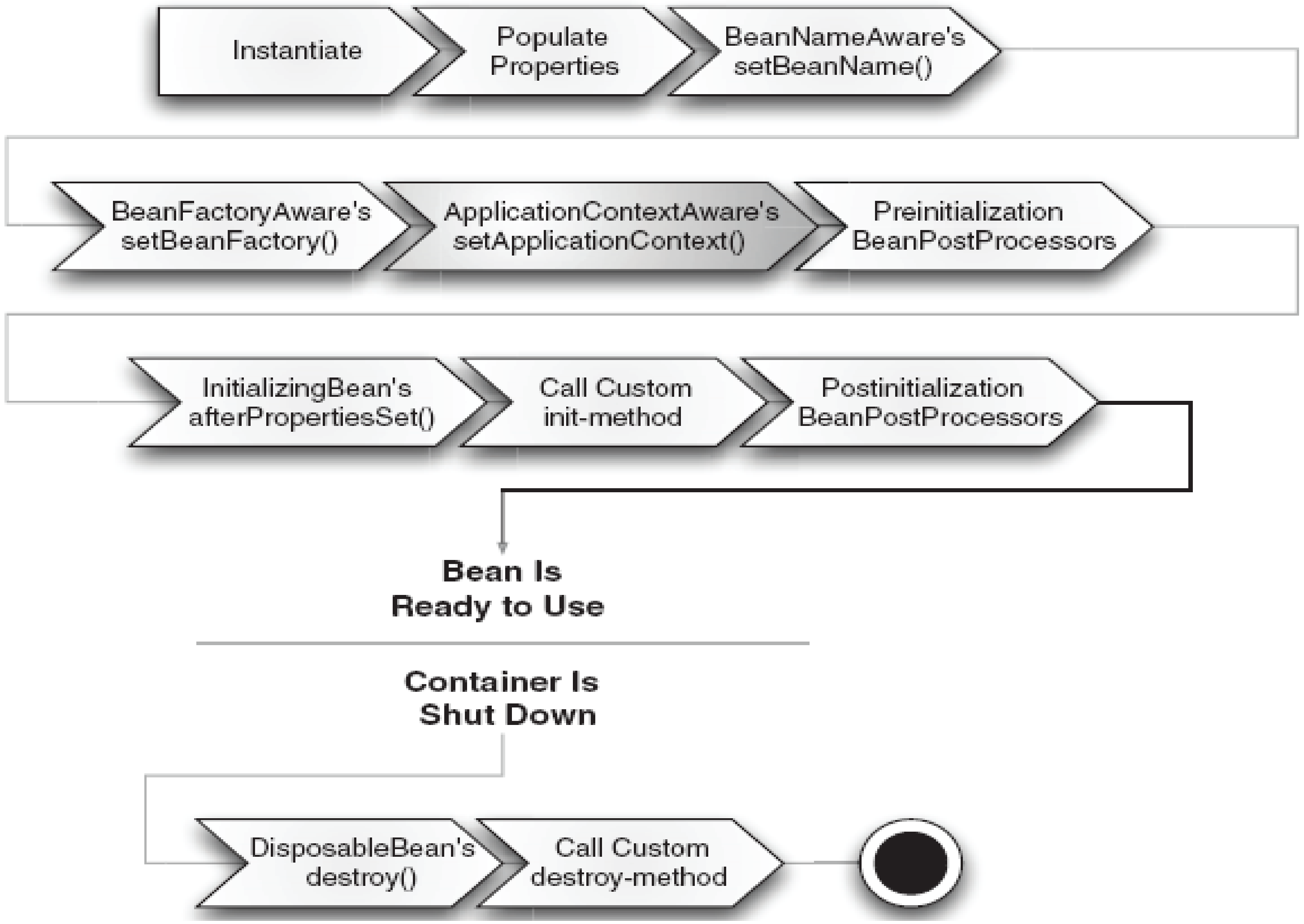
```
public class BlackListNotifier implement ApplicationListener {
/** notification address */
private String notificationAddress;
      public void setNotificationAddress(String notificationAddress) {
            this.notificationAddress = notificationAddress;
      }
public void onApplicationEvent(ApplicationEvent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

Life Cycle of a Bean

Life cycle of a bean in BeanFactory Container


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
Bean Is Ready to Use
Container Is Shut Down
DisposableBean's destroy()
Call Custom destroy-method

The lifecycle of a bean within a Spring application context


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
Bean Is Ready to Use
Container Is Shut Down
DisposableBean's destroy()
Call Custom destroy-method