SPRING FRAMEWORK 3.0

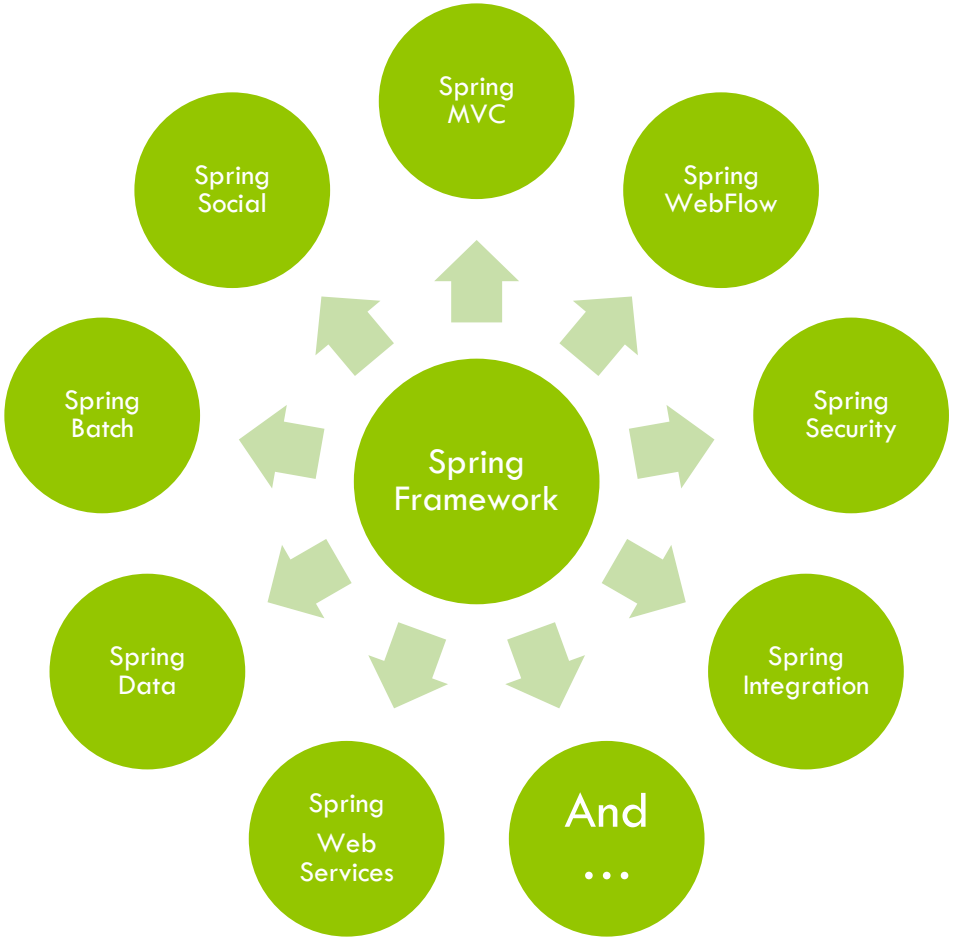
Dmitry Noskov

Spring Core

What is Spring?

Spring is the most popular application development framework for enterprise Java™. Millions of developers use Spring to create high performing, easily testable, reusable code without any lock-in.

The Spring projects



Overview

History

Goals

Spring modules

Spring triangle

Spring Framework history(1)

- the first version was written by Rod Johnson
- Expert One-on-One J2EE Design and Development
- first released in June 2003
- milestone releases in 2004 and 2005
- awards
  - Jolt productivity award
  - JAX Innovation award

Spring Framework history(2)

- Spring 3.0 (released Dec 2009)
  - Java 1.5+
  - REST support, SpEL, more annotations, JavaConfig
- Spring 2.5 (released Nov 2007)
  - Java 1.4+
  - XML namespaces, annotations
- Spring 2.0 (released Oct 2006)
  - Java 1.3+
  - AspectJ support, JPA

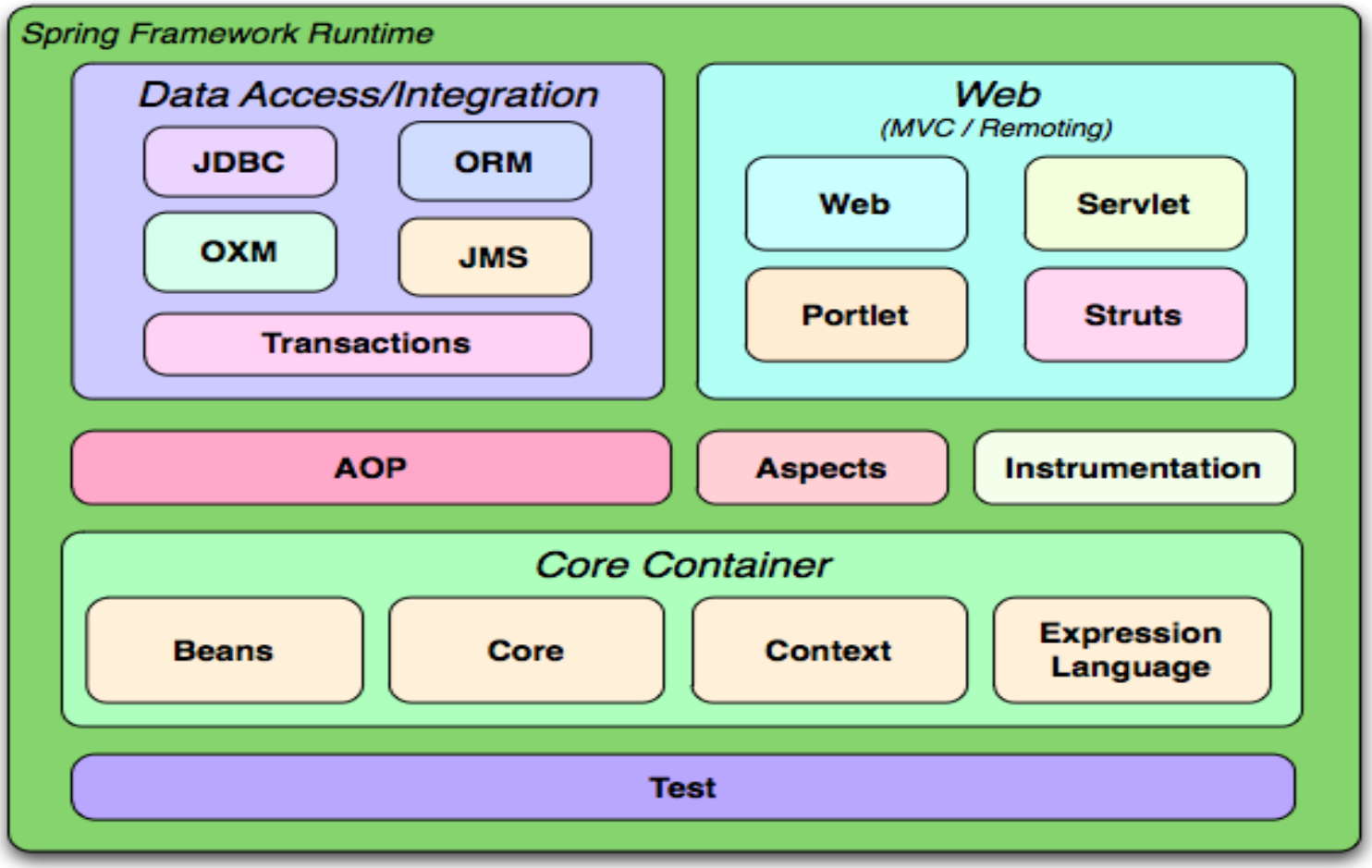
Goals

- make J2EE **easier** to use
- make the **common** tasks easier
- promote good programming **practice**
- you can focus on the **domain** problems

What is Spring Framework today?

- an **open source** application framework
- a **lightweight** solution for enterprise applications
- non-invasive (**POJO** based)
- is **modular**
- **extendible** for other frameworks
- de facto **standard** of Java Enterprise Application

Spring modules



Core container

- Core and Beans

provide the fundamental parts of the framework, including IoC and Dependency Injection features

- Context

it is a means to access objects in a framework-style manner that is similar to a JNDI registry

- Expression language

provides a powerful expression language for querying and manipulating an object graph at runtime

AOP, Aspect, Instrumentation

- AOP

provides an *AOP Alliance*-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated

- Aspect

provides integration with **AspectJ**

- Instrumentation

provides class instrumentation support and classloader implementations to be used in certain application servers

Data Access/Integration

- JDBC - provides a JDBC-abstraction layer
- ORM - provides integration layers for popular object-relational mapping APIs, including **JPA, JDO, Hibernate** and **iBatis**
- OXM - provides an abstraction layer that supports Object/XML mapping implementations for **JAXB, Castor, XMLBeans, JiBX** and **XStream.**
- JMS – contains features for **producing** and **consuming** messages.
- Transaction - supports **programmatic** and **declarative** transaction management

WEB

- Spring's WEB

provides basic web-oriented integration features

- WEB-Servlet

Spring's model-view-controller (**MVC**) implentation

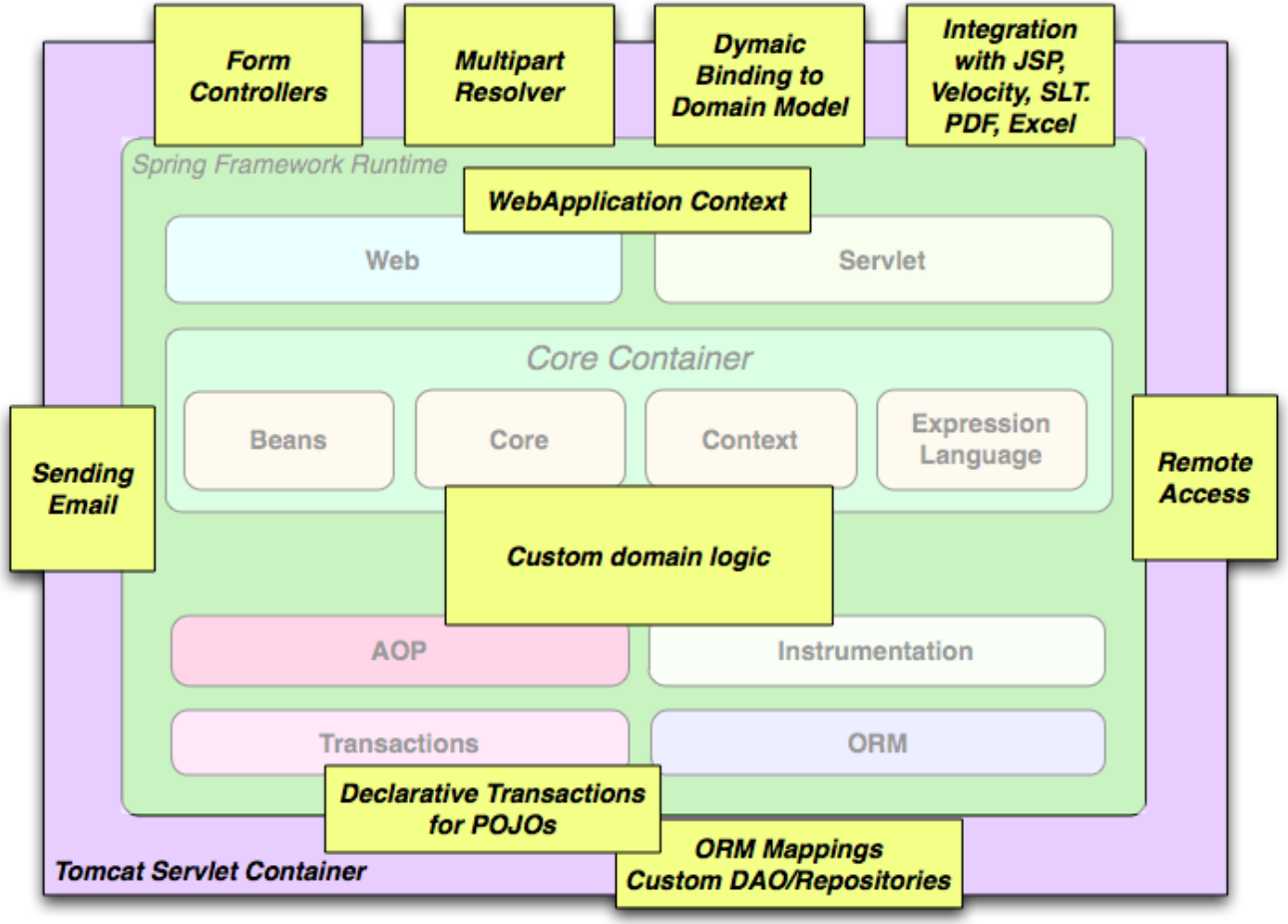
- WEB-Struts

contains the classes for integrating a classic **Struts** WEB tier within a Spring application

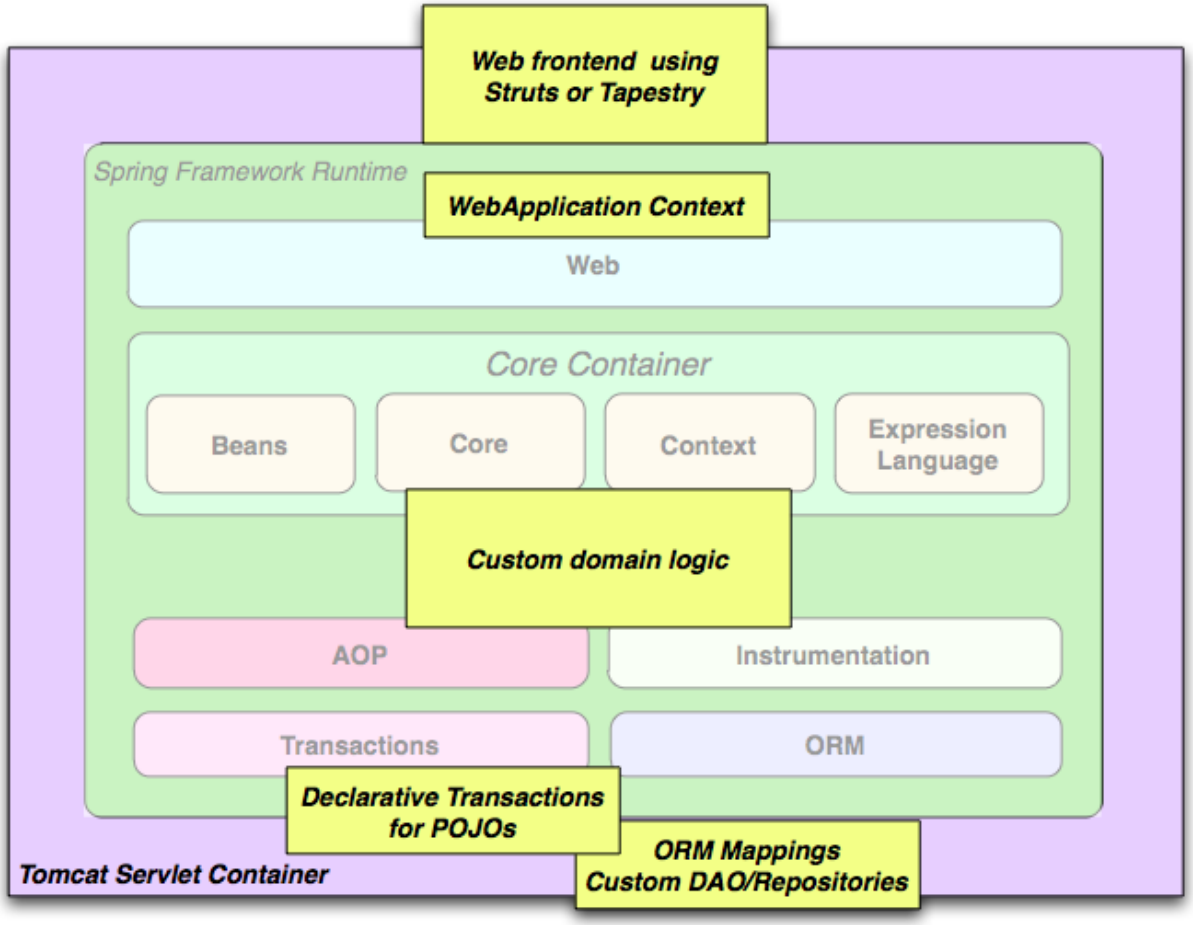
- WEB-Portlet

provides the **MVC** implementation to be used in a portlet environment

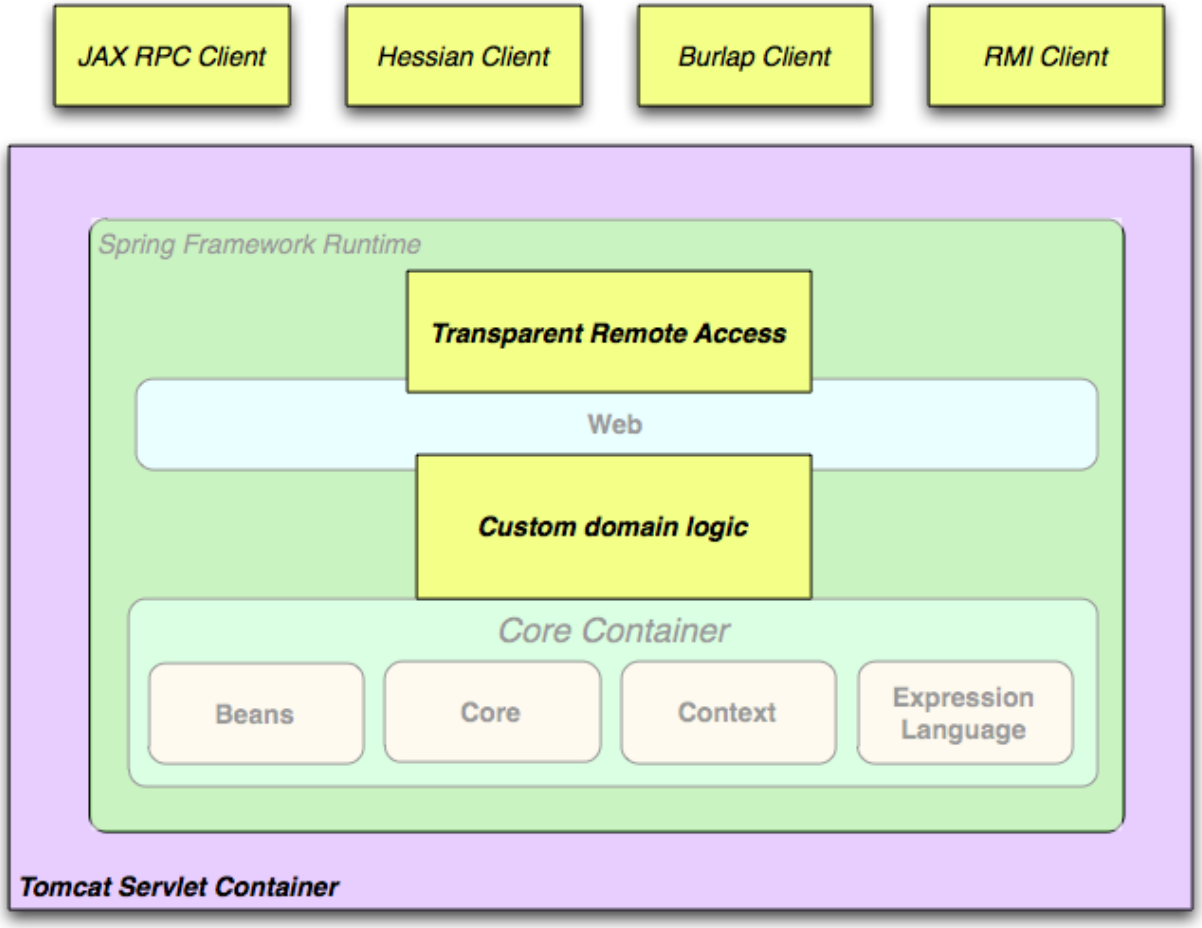
Full-fledged Spring web application



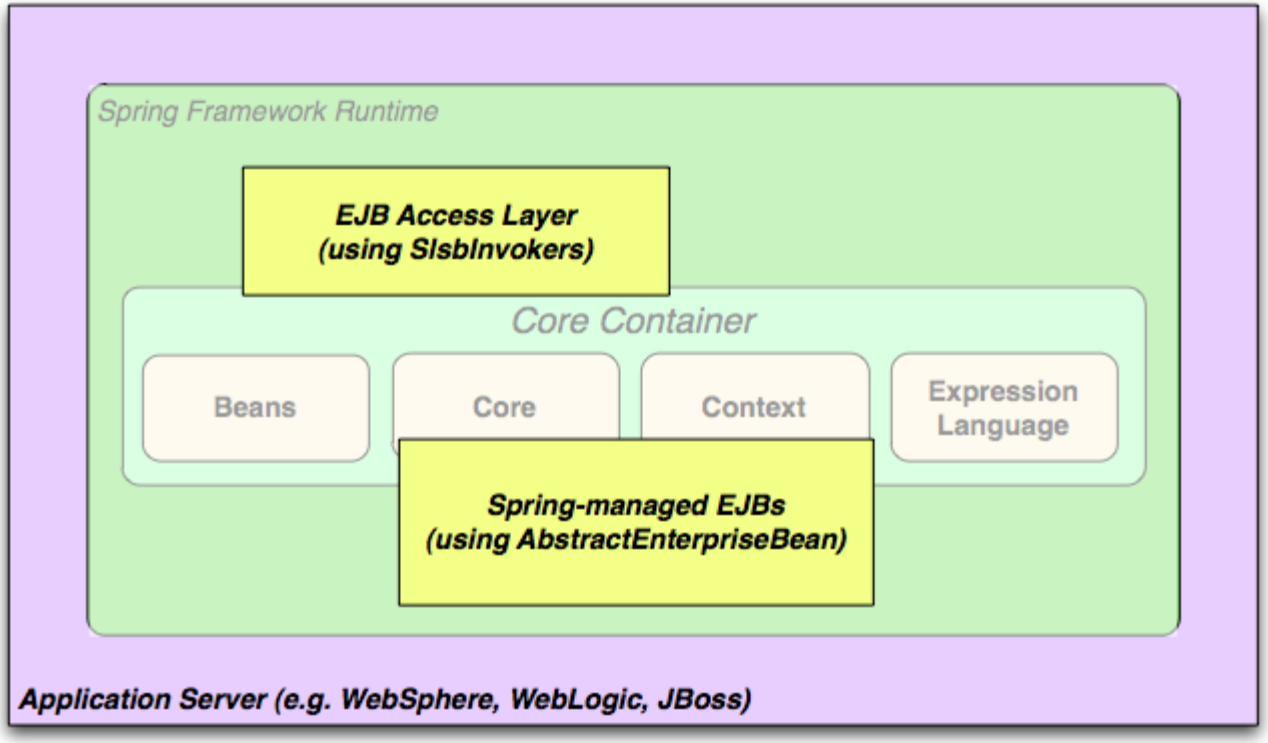
Using a third-party web framework



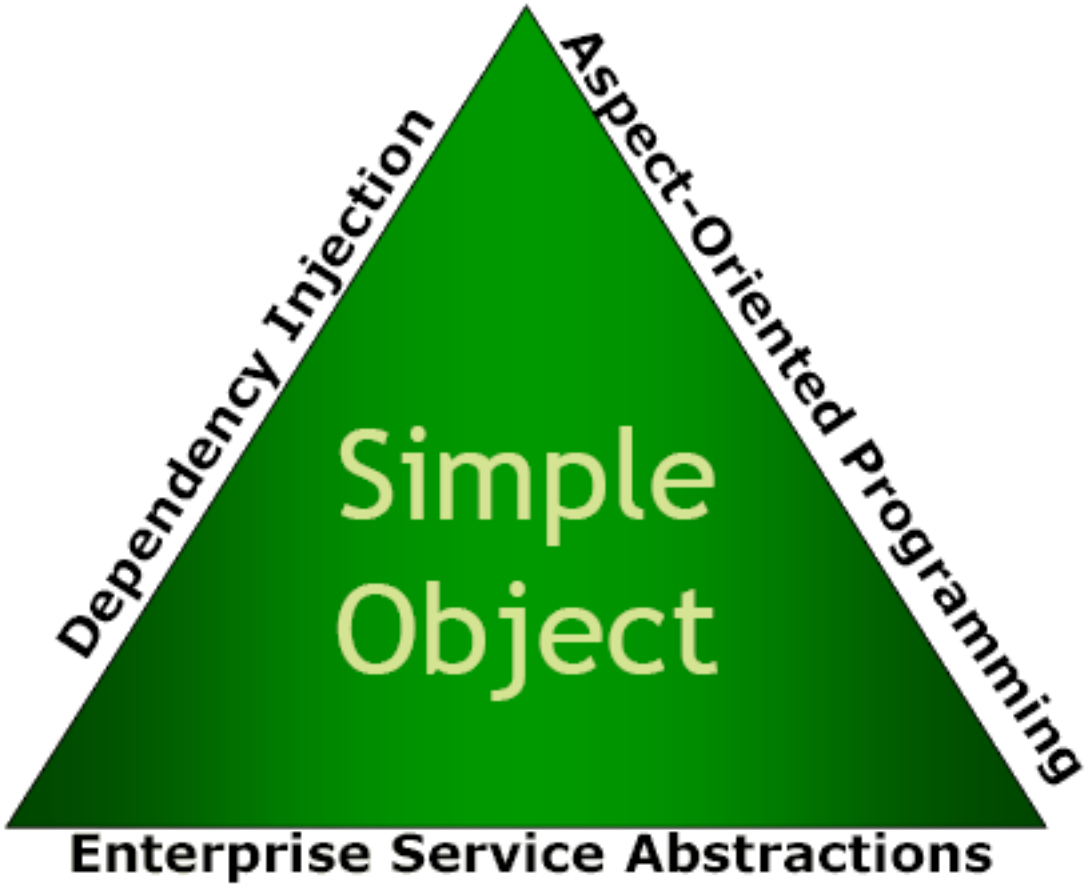
Remoting


JAX RPC Client
Hessian Client
Burlap Client
RMI Client
Spring Framework Runtime
Transparent Remote Access
Web
Custom domain logic
Core Container
Beans
Core
Context
Expression Language
Tomcat Servlet Container

EJB


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 Spring triangle



Spring IoC container

IoC pattern

Application lifecycle

Essence of Spring IoC container

Instantiation an ApplicationContext

What is IoC?

- is a **concept** in application development
- "don't call me, I'll call you"
- one form is **Dependency Injection** (DI)

Dependency Injection

- is a **process** whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method
- exist in two major variants

    1) **constructor** injection

    2) **setter** injection

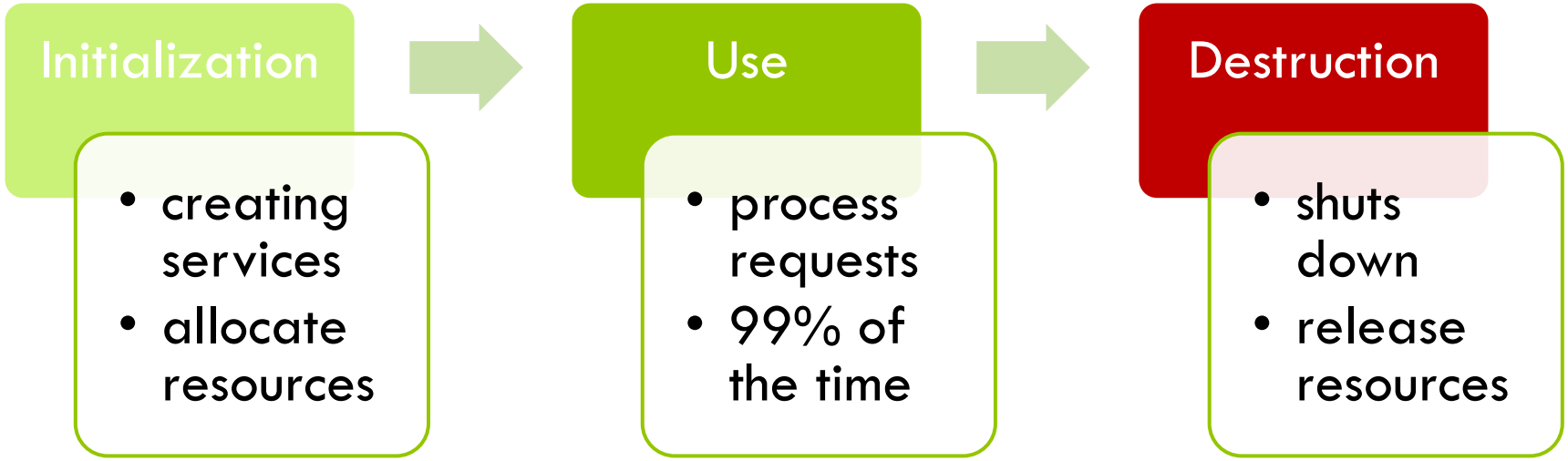
IoC vs DI vs Factory

- **DI** it is specific type of **IoC**

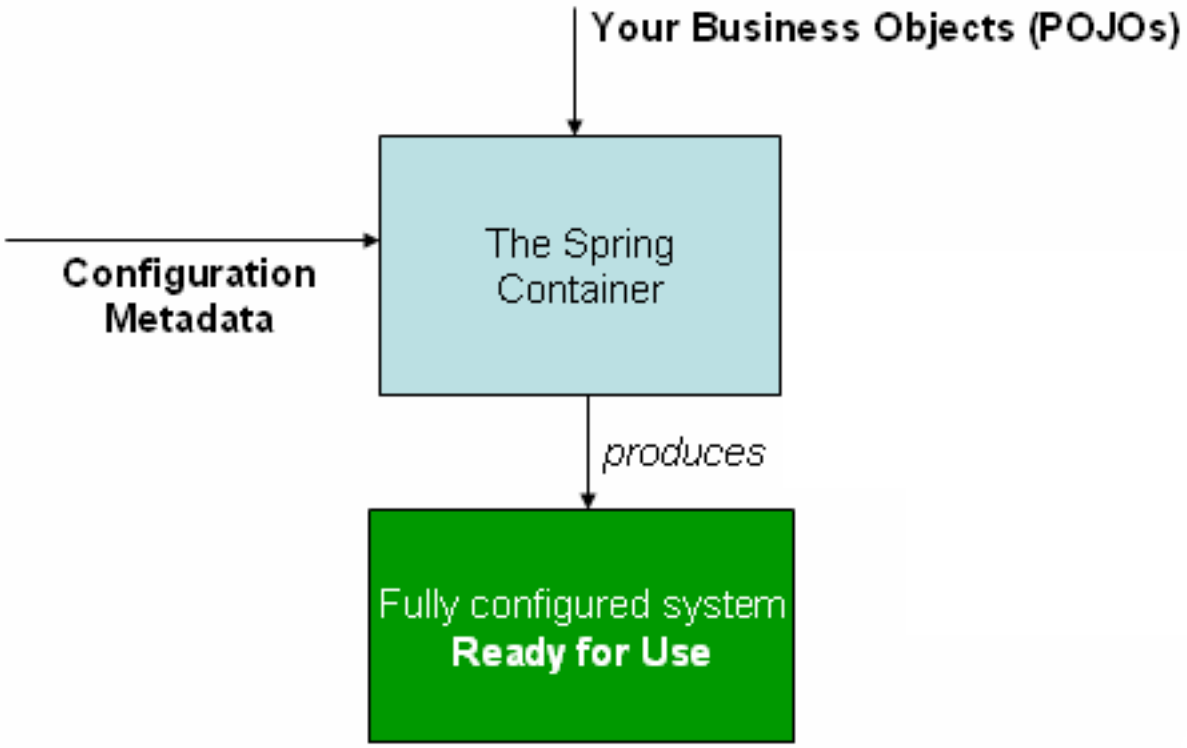
- The **Factory** pattern's main concerns is **creating**
- The **DI's** main concern is how things are **connected together**

- **DI** related to **Factory** and **Strategy** patterns, but mostly resembles to **Build Pattern**

Application lifecycle



Essence of IoC container

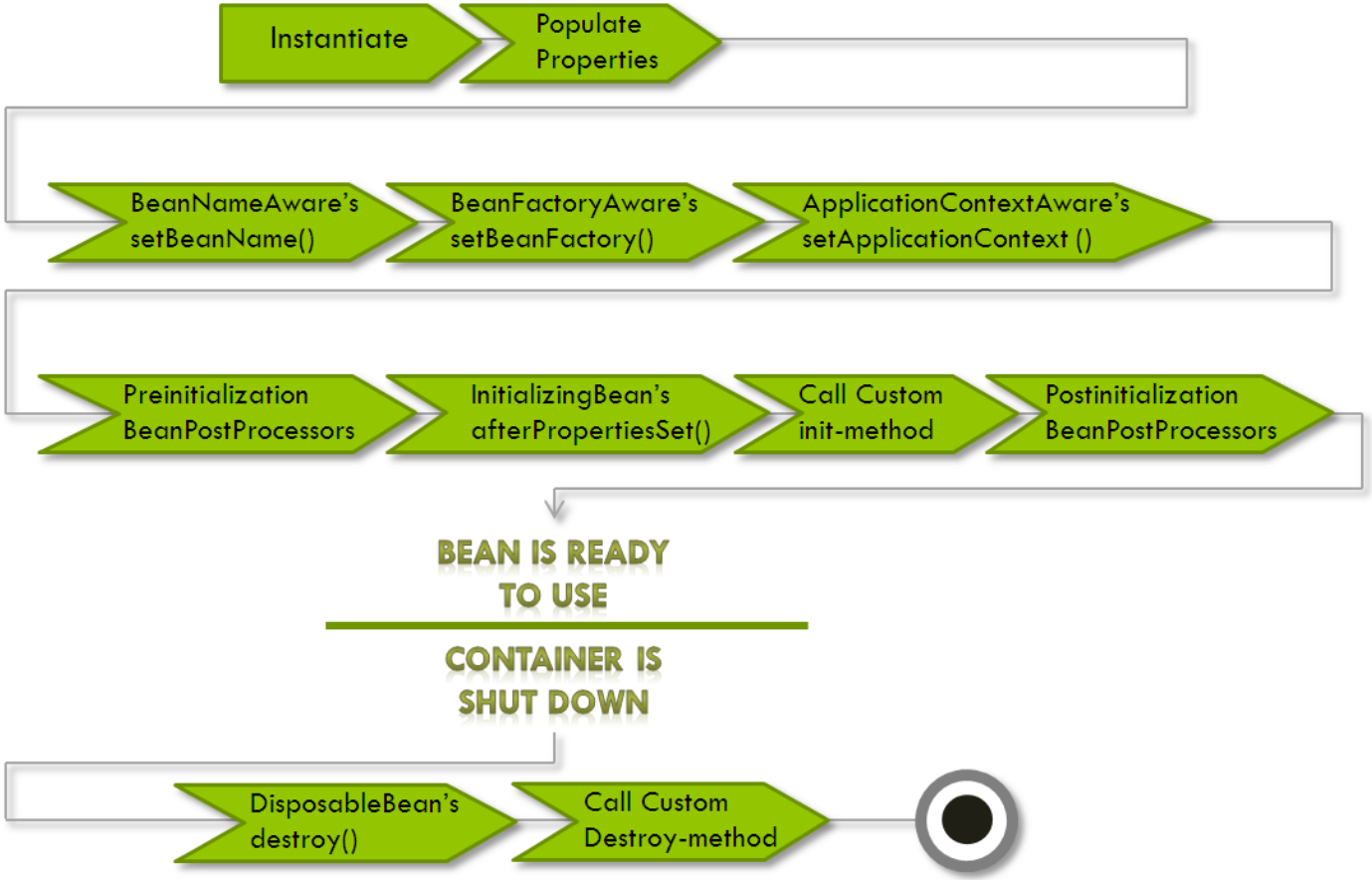


Terms

- ApplicationContext
  - **represents** the Spring IoC container
- bean
  - is an object that **managed** by Spring IoC container
- BeanDefinition
  - **describe** a bean instance

Context lifecycle

Bean lifecycle



Creating application context

- environments
  - standalone
  - WEB
  - JUnit
  - EJB
- special prefixes
  - classpath
  - file system
  - relative path

Instantiation for standalone

- ClassPathApplicationContext
- FileSystemApplicationContext

```
ApplicationContext context =
  new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "repositories.xml"});

// retrieve configured instance
CurrencyService currencyService
       = context.getBean("currency", CurrencyService.class);

AccountService accountService
       = context.getBean(AccountService.class);
```

Instantiation for WEB applications

Servlet 2.4+

```
<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>
```

Servlet 2.3

```
<servlet>
 <servlet-name>
  context
 </servlet-name>
 <servlet-class>
org.springframework.web.context.ContextLoaderServlet
 </servlet-class>
 <load-on-startup>1
 </load-on-startup>
</servlet>
```

Bean Scope

Simple

Runtime

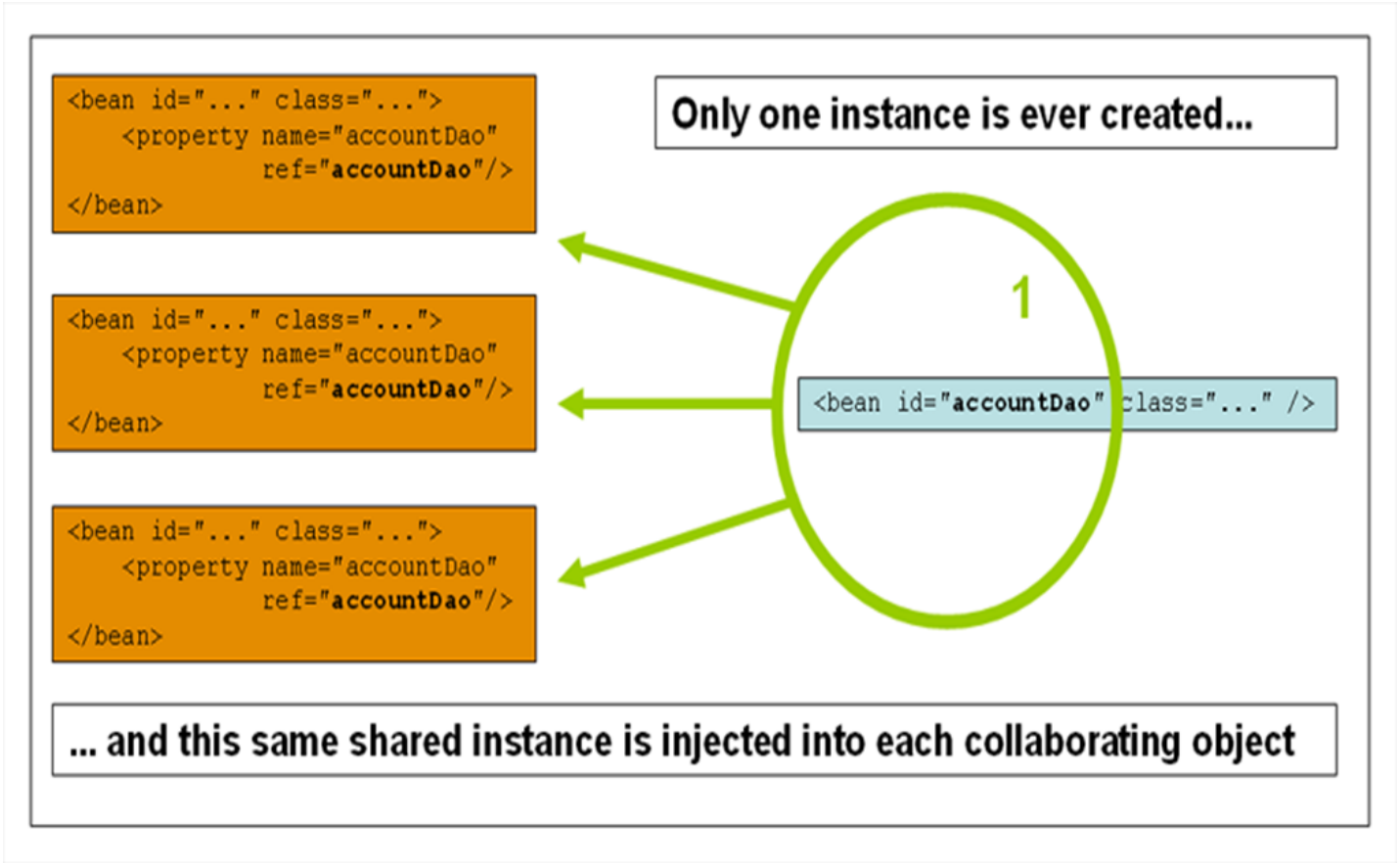
WEB

CGLIB / JDK proxies

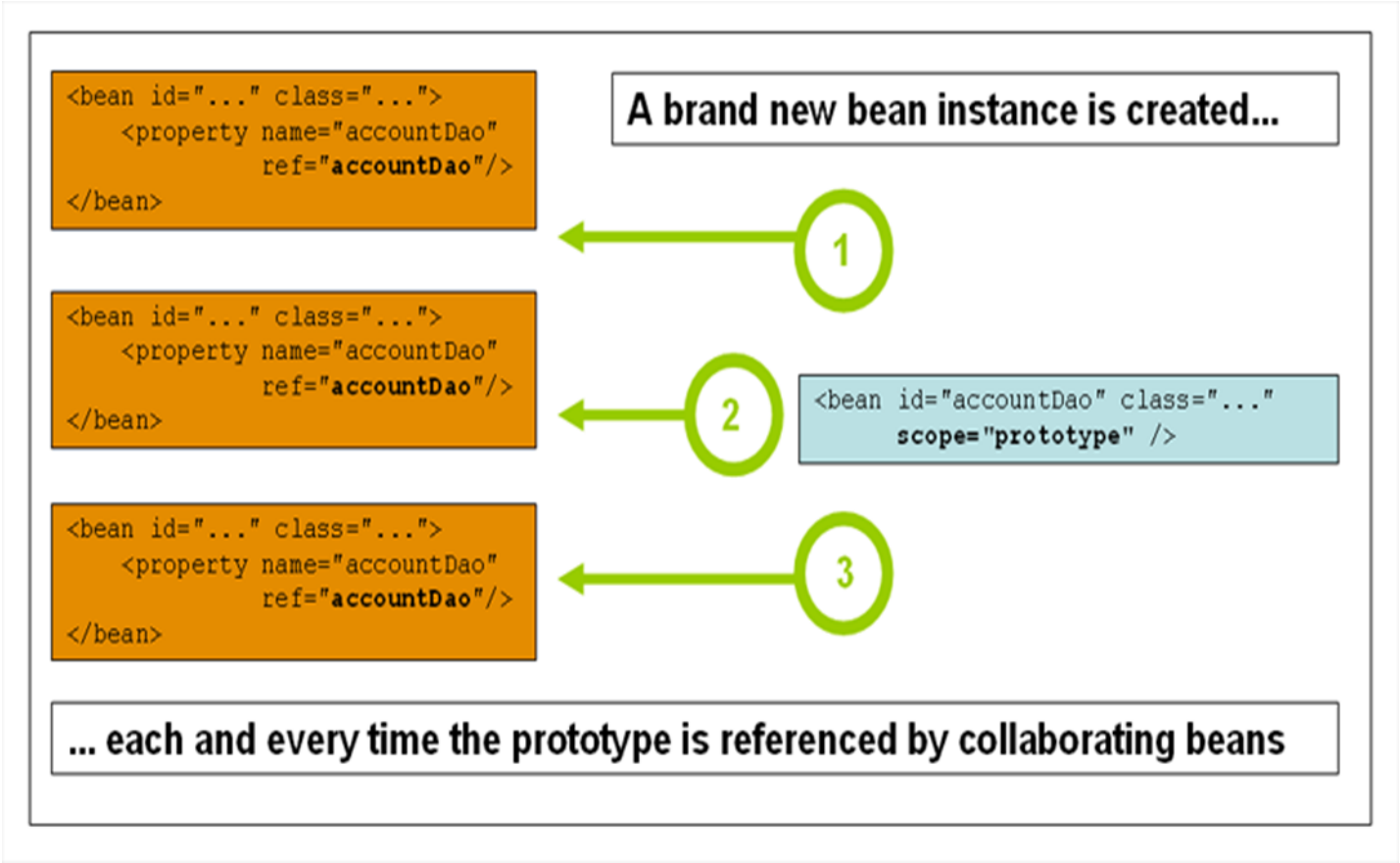
Spring Bean Scopes

- simple
  - singleton
  - prototype
- runtime
  - thread
  - custom implementation
- web-aware scopes (available only for web-aware ApplicationContext)
  - request
  - session
  - global session

The singleton scope



The prototype scope



Custom scope

```
public interface org.springframework.beans.factory.config.Scope {
  //Return the object with the given name from the underlying scope
  Object get(String name, ObjectFactory<?> objectFactory);

  //Remove the object with the given name from the underlying scope.
  Object remove(String name);

  //Register a callback to be executed on destruction of the specified object
  void registerDestructionCallback(String name, Runnable callback);

  //Resolve the contextual object for the given key, if any.
  Object resolveContextualObject(String key);

  //Return the conversation ID for the current underlying scope, if any.
  String getConversationId();
}
```

Using runtime scope

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="thread">
        <bean class="org.springframework.context.support.SimpleThreadScope"/>
      </entry>
    </map>
  </property>
</bean>

<bean id="threadService" class="example.ThreadServiceImpl" scope="thread">
  <property name="priority" value="0"/>
  <aop:scoped-proxy/>
</bean>

<bean id="threadMonitor" class="example.ThreadMonitorImpl">
  <property name="threadService" ref="thredService"/>
</bean>
```

Scoped bean as dependencies

Using CGLIB library (by default)

```
<bean id="preferences" class="UserPreferences" scope="session">
  <aop:scoped-proxy/>
</bean>
<bean id="userManager" class="example.UserManager">
  <property name="userPreferences" ref="preferences"/>
</bean>
```

Using JDK interface based proxies means

```
<bean id="preferences" class="UserPreferences" scope="session">
  <aop:scoped-proxy proxy-target-class="false"/>
</bean>
<bean id="userManager" class="example.UserManager">
  <property name="userPreferences" ref="preferences"/>
</bean>
```

WEB-aware scope configuration

Servlet 2.4+

```
<web-app>
<listener>
 <listener-class>
org.springframework.web.context.request.
RequestContextListener
 </listener-class>
</listener>
</web-app>
```

Servlet 2.3

```
<web-app>
 <filter>
  <filter-name>filter</filter-name>
  <filter-class>
  …web.filter.RequestContextFilter
  </filter-class>
 </filter>
 <filter-mapping>
  <filter-name>filter</filter-name>
  <url-pattern>/*</url-pattern>
 </filter-mapping>
</web-app>
```

XML-based configuration

Configuration

Main features

Bean lifecycle

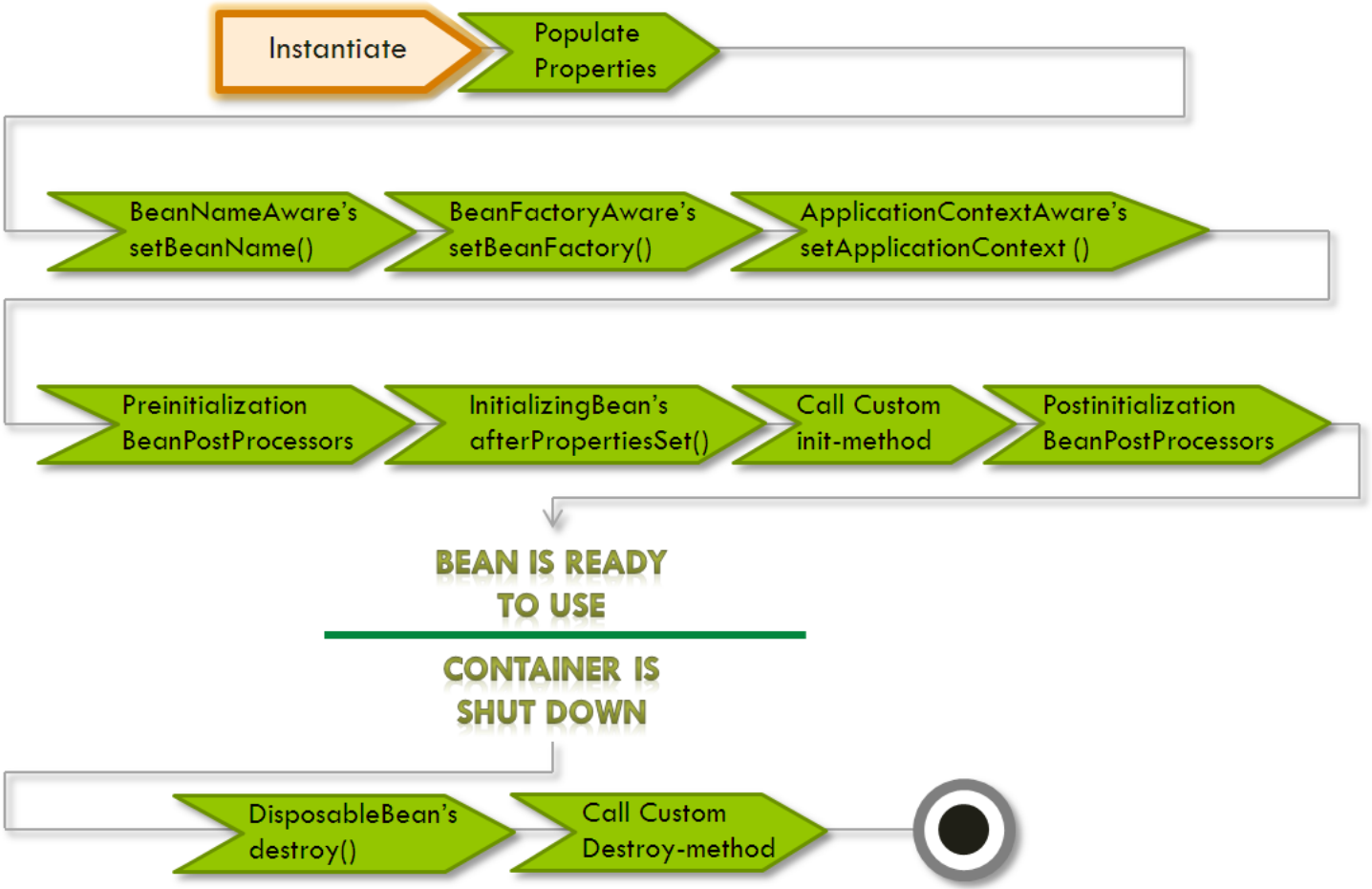
Additional features

Namespaces

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <bean id="" class=""></bean>

  <bean id="" class=""/>
</beans>
```

BeanFactoryPostProcessor

```
public interface BeanFactoryPostProcessor {

    /**
     * Modify the application context's internal bean factory after its standard
     * initialization. All bean definitions will have been loaded, but no beans
     * will have been instantiated yet. This allows for overriding or adding
     * properties even to eager-initializing beans.
     * @param beanFactory the bean factory used by the application context
     * @throws org.springframework.beans.BeansException in case of errors
     */
    void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory);
}
```



Instantiating bean

Naming beans

```
<beans>
  <bean id="beanId" class="org.exhanger.api.OrderServiceImpl"/>

  <bean name="name/" class="org.exhanger.api.OrderServiceImpl"/>

  <bean name="name1,name2;name3 name4" class="...">
  </bean>

  <alias name="fromName" alias="toName"/>
</beans>
```

Instantiating bean

- with a **constructor**
- with a static **factory method**
- using an **instance** factory method
- with the **FactoryBean**

Instantiating with a constructor

- simple class

```
<bean id="exampleBean" class="examples.ExampleBean"/>
public class ExampleBean {
  public ExampleBean() {}
}
```

- inner class

```
<bean id="innerBean" class="examples.ExampleBean$InnerBean"/>
public class ExampleBean {
  public static class InnerBean {
  }
}
```

Instantiating with a static method

```
<bean id="clientService" class="examples.ClientService"
      factory-method="createInstance"/>

public class ClientService {
  private static ClientService clientService = new ClientService();

  private ClientService() {}

  public static ClientService createInstance() {
    return clientService;
  }
}
```

Using an instance factory method

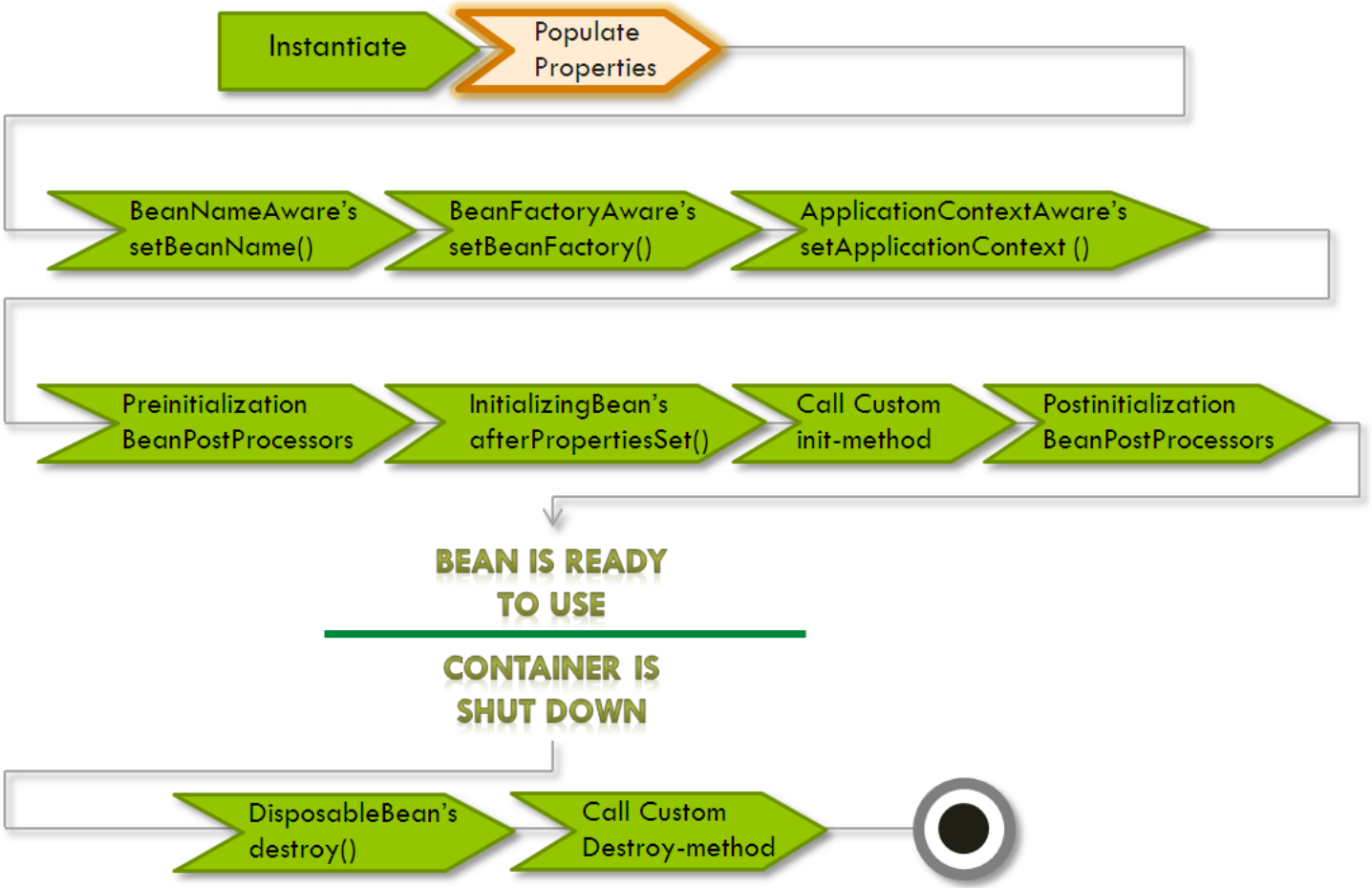
```
<!-- the factory bean-->
<bean id="serviceLocator" class="example.ServiceLocator"/>
<!-- the bean to be created via the factory bean -->
<bean id="clientService"
      factory-bean="serviceLocator"
      factory-method="createClientServiceInstance"/>
```

```
public class ServiceLocator {
  private static ClientService clientService = new ClientService();

  public ClientService createClientServiceInstance() {
    return clientService;
  }
}
```

Instantiating with the FactoryBean

```
public interface FactoryBean<T> {
  /**Return an instance of the object managed by this factory.*/
  T getObject() throws Exception;

  /** Return the type of object that this FactoryBean creates.*/
  Class<?> getObjectType();

  /**Is the object managed by this factory a singleton? */
  boolean isSingleton();
}
```

```
<bean id="bean" class="FactoryBean"/>
```



Dependency injection

Constructor-based DI

- is accomplished by the container invoking a **constructor** with a number of arguments, each representing a dependency
- calling a static factory **method** with specific arguments to construct the bean is nearly equivalent

Constructor argument resolution

- argument resolution

```
<bean id="b1" class="org.examples.ClientServiceBean">
  <constructor-arg ref="b2"/>
  <constructor-arg ref="b3"/>
</bean>
```

- argument type matching

```
<bean id="b2" class="ClientService" factory-method="getInstance">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

- argument index

```
<bean id="b3" factory-bean="factory" factory-method="getInstance"/>
  <constructor-arg index="0" value="75"/>
  <constructor-arg index="1" value="42"/>
</bean>
```

Setter-based DI

- is accomplished by the container calling **setter methods** on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate bean

Constructor vs setter

- constructor
  - mandatory dependencies
  - immutability
- setter
  - optional dependencies and default values
  - obvious names
  - auto inheritance

Straight values

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource" >

  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url">
      <value>jdbc:mysql://localhost:3306/mydb</value>
  </property>
  <property name="username" value="root"/>
  <property name="password" value="masterkaoli"/>
</bean>
```

Null and empty values

```
<!-- is equivalent to the following code: bean.setEmail("").-->
<bean class="Bean">
  <property name="email" value=""/>
</bean>
```

```
<!-- The <null/> element handles null values. -->
<bean class="Bean">
  <property name="email"><null/></property>
</bean>
```

properties

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations" value="classpath:jdbc.properties"/>
</bean>

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

idref

bean

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

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

local

```
<property name="targetName">
<!--a bean with id 'theTargetBean' must exist -->
  <idref local="theTargetBean"/>
</property>
```

Reference to other bean

- bean

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

- local

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

- parent

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService"/>

<!-- in the child (descendant) context -->
<bean id="accountService" class="org.springframework.ProxyFactoryBean">
  <property name="target">
    <ref parent="accountService"/>
  </property>
</bean>
```

Inner bean

```
<bean id="orderService" class="OrderServiceImpl">
  <property name="repository" ref="orderRepository"/>
  <property name="builder">
    <bean class="OrderBuilderImpl">
      <constructor-arg ref="accountService"/>
      <constructor-arg ref="currencyService"/>
    </bean>
  </property>
</bean>
```

Collections(1)

```
<!-- setAdminEmails(java.util.Properties) call -->
<property name="adminEmails">
  <props>
    <prop key="administrator">administrator@gmail.com</prop>
  </props>
</property>
<!-- setSomeList(java.util.List) call -->
<property name="someList">
  <list>
    <value>a list element followed by a reference</value>
    <ref bean="myDataSource" />
  </list>
</property>
```

Collections(2)

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

p namespace

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

  <bean id="orderService" class="OrderServiceImpl"
        p:repository-ref="orderRepository"
        p:locaton="BY"/>

</beans>
```

util namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/util
     http://www.springframework.org/schema/util/spring-util-3.0.xsd">

</beans>
```

util in action

```
<property name="isolation">
  <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
</property>

<bean id="testBean" class="org.springframework.beans.TestBean">
  <property name="age" value="10"/>
</bean>
<util:property-path id="name" path="testBean.age"/>

<util:properties id="" location="classpath:jdbc.properties"/>

<util:list id="emails" list-class=""><value/><value/></util:list>

<util:map id="emails" map-class=""><entry key="" value=""/></util:map>

<util:set id="emails" set-class=""><value></value></util:set>
```

Additional features

- bean definition inheritance
- importing configuration files
- autowiring
- lazy initialization
- dependency checking

Bean definition inheritance

```
<!—attribute class is optional →
<bean id="parent" abstract="true" class="AbstractService">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="service" class="ServiceImpl" parent="parent">
     <property name="name" value="override it"/>
</bean>
```

Bean definition inheritance(2)

```
<bean id="parent" abstract="true">
  <property name="name" value="parent"/>
</bean>

<bean id="default" class="DefaultServiceImpl" parent="parent">
  <property name="name" value="default"/>
  <property name="value" value="22"/>
</bean>

<bean id="custom" class="CustomServiceImpl" parent="default">
     <property name="name" value="custom"/>
</bean>
```

Importing configuration files

- xml-context-config.xml

```
<beans>
  <import resource="classpath:xml-repository-config.xml"/>
  <import resource="classpath:xml-service-config.xml"/>
</beans>
```

- xml-service-config.xml

```
<beans>
  <bean id="currencyRepository" class="CurrencyMapRepository"/>
</beans>
```

- xml-repository-config.xml

```
<bean id="currencyService" class="CurrencyServiceImpl">
  <constructor-arg ref="currencyRepository"/>
</bean>
```

Using depends-on

- can explicitly force one or more beans to be initialized **before** the bean using this element is initialized

```
<bean id="beanOne" class="ExampleBean"
      depends-on="costService,accountDao">
  <property name="deviceManager" ref="deviceManager" />
</bean>

<bean id="costService" class="example.service.CostService" />
<bean id="accountDao" class="example.jdbc.JdbcAccountDao" />
```

Lazy-initialized beans

- IoC container create a bean instance when it is first requested

```
<bean id="lazy" class="ExpensiveToCreateBean" lazy-init="true"/>
```

- control lazy-initialization at the container level

```
<beans default-lazy-init="true">
  <!-- no beans will be pre-instantiated... -->
</beans>
```

- disadvantages
  - errors in the configuration or surrounding
  - environment are detected some time later

Autowiring

- advantages
  - can significantly **reduce** the need to **specify** properties or constructor arguments
  - can **update** a configuration as your objects evolve
- disadvantages
  - cannot autowire **simple** properties (primitives)
  - autowiring is **less exact** than explicit wiring
  - wiring information may not be available to tools that may generate documentation from a Spring container

Autowiring modes

- no (default)
- byName

```
<bean id="messageSource" class="MessageSourceImpl" />
public void setMessageSource(MessageSource messageSource)
```

- byType

```
<bean id="messageSource" class="MessageSourceImpl" />
public void setMessageSource(MessageSource messageSource)
```

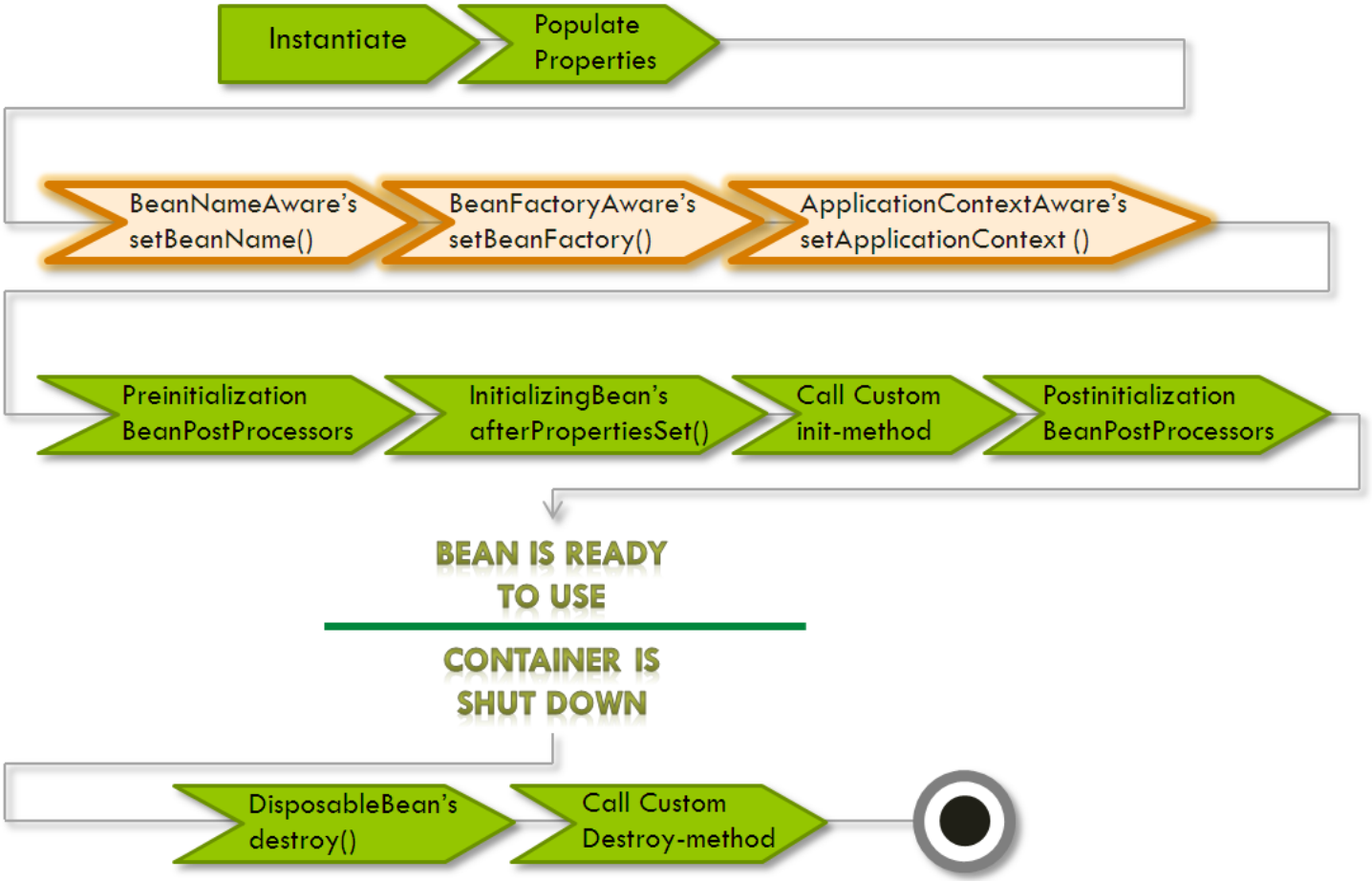
- constructor
- autodetect

Dependency checking

- none (default)
- simple

  checking for primitive types and collections
- object

  checking for collaborators only
- all



Spring Aware Interfaces

Spring **aware** interfaces

- BeanNameAware
- BeanFactoryAware
- ApplicationContextAware

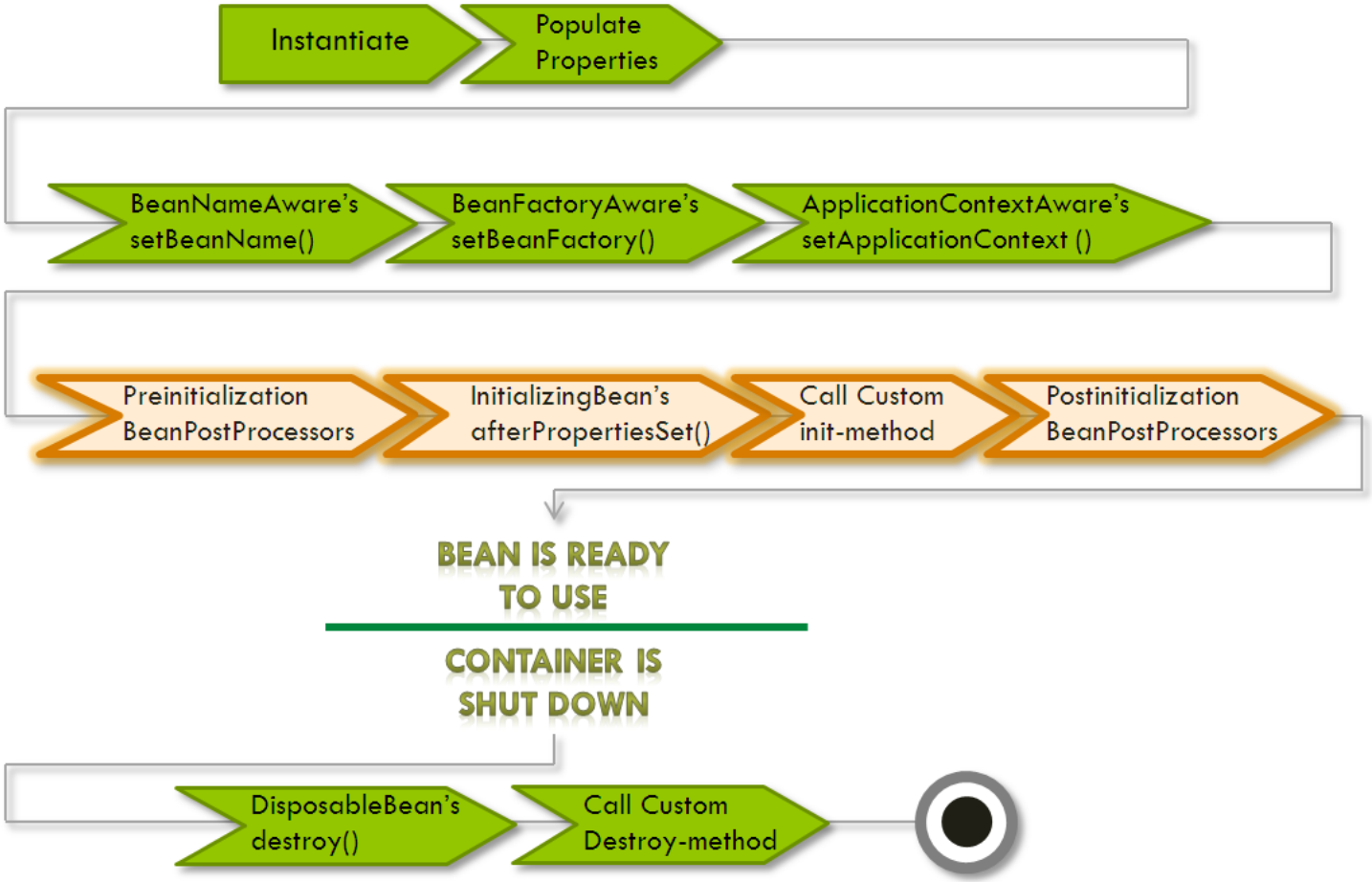
- ApplicationEventPublisherAware
- ServletConfigAware
- ServletContextAware
- MessageSourceAware

Application events

```
public class SomeService implements ApplicationContextAware {
    public void setApplicationContext(ApplicationContext ctx) {}

    public void anyMethod(String value) {
       context.publishEvent(new AnyEvent(value));
    }
}

public class Listener implements ApplicationListener<AnyEvent> {
    public void onApplicationEvent(AnyEvent event) {
    }
}
```

Internationalization

```
<bean class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list><value>label</value><value>exceptions</value></list>
  </property>
</bean>

public interface MessageSource {
  /** Try to resolve the message. Return default if no message was found.*/
  String getMessage(String code, Object[] args, String default, Locale locale);

  /** Try to resolve the message. Treat an error if the message can't be found.*/
  String getMessage(String code, Object[] args, Locale locale);

  /** Try to resolve the message using MessageSourceResolvable */
  String getMessage(MessageSourceResolvable resorvable, Locale locale);
}
```



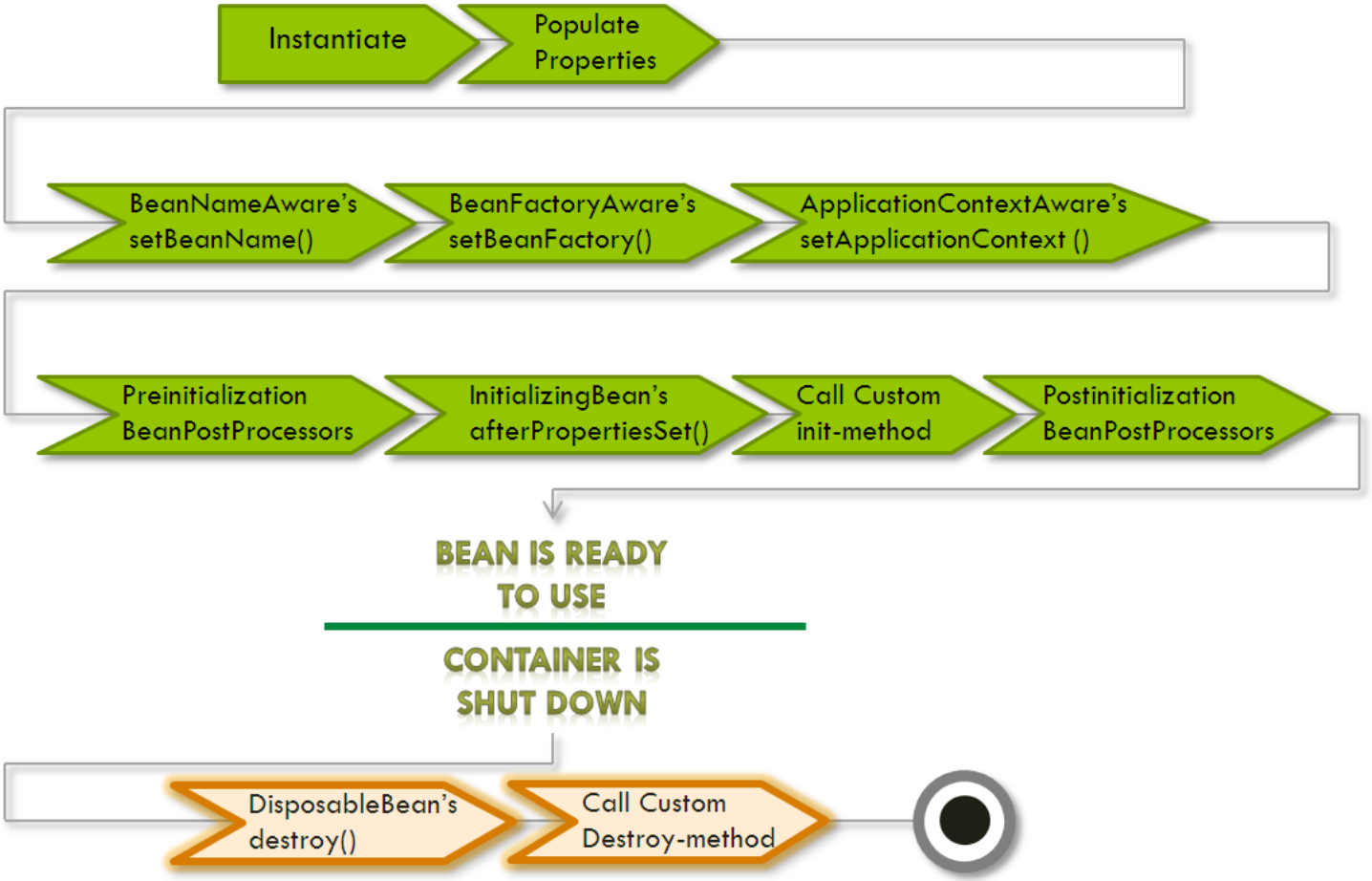
Initialization

BeanPostProcessor

```
/**
* ApplicationContexts can autodetect BeanPostProcessor beans in their
* bean definitions and apply them to any beans subsequently created. */
public interface BeanPostProcessor {
/**
* Apply this BeanPostProcessor to the given new bean instance before any bean
* initialization callbacks (like afterPropertiesSet or a custom init-method)*/
Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;

/**
* Apply this BeanPostProcessor to the given new bean instance after any bean
* initialization callbacks (like afterPropertiesSet or a custom init-method).
*/
Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;

}
```

Initialization callbacks

```
<bean id="bean1" class="exampl.ExampleBean1" init-method="init"/>
public class ExampleBean1 {
  public void init() {}
}

<bean id="bean2" class="examples.ExampleBean2"/>
public class ExampleBean2 implements InitializingBean {
  public void afterPropertiesSet() {}
}

public class ExampleBean2 {
  @PostConstruct
  public void initialize() {}
}
```



Destroy Spring Beans

Destruction callbacks

```
<bean id="bean1" class="example.Bean1" destroy-method="cleanup"/>
public class Bean1 {
  public void cleanup() {}
}

<bean id="bean2" class="example.Bean2"/>
public class Bean2 implements DisposableBean {
  public void destroy() {}
}

public class Bean2 {
  @PreDestroy
  public void destroy() {}
}
```

Combining lifecycle mechanisms

- initialization methods, are called as follows:
  - methods annotated with `@PostConstruct`
  - afterPropertiesSet as defined by **InitializingBean**
  - **custom** configured init method
- destroy methods are called in the same order:
  - method annotated with `@PreDestroy`
  - destroy() as defined by the **DisposableBean**
  - **custom** configured destroy method

Annotation-based configuration

Basic annotations

Spring stereotypes

JSR-250

JSR-330

Annotation based configuration

- an **alternative** to XML setups
- annotation injections is performed **before** XML

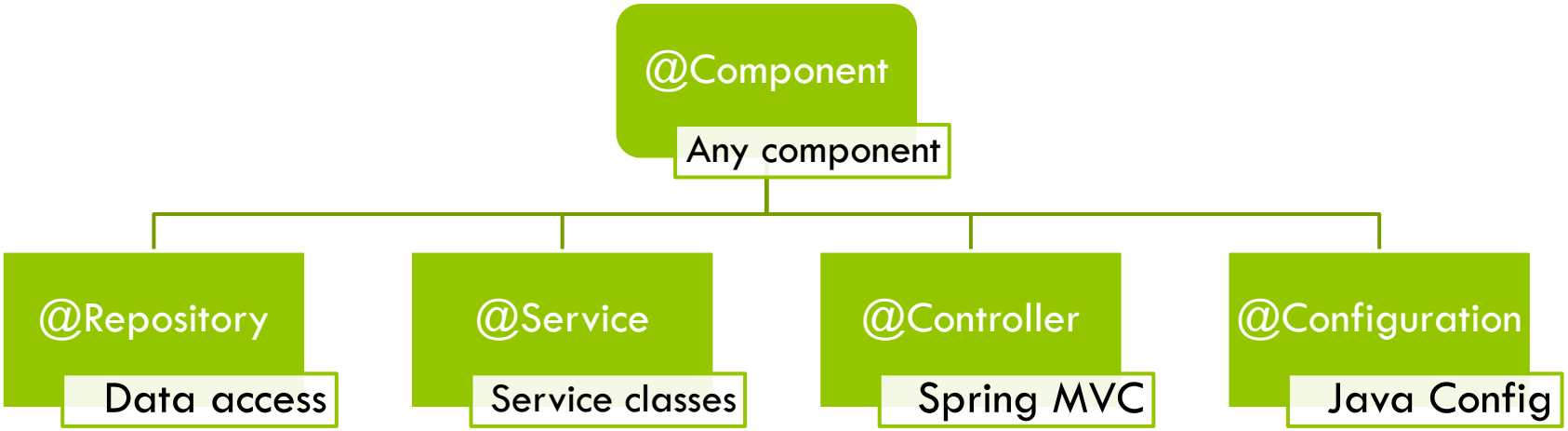
Basic annotations(1)

- Spring Annotations
  - `@Autowired`
  - `@Qualifier`
  - `@Required`
  - `@Value`
- JSR 250
  - `@Resource`
  - `@PostConstruct`
  - `@PreDestroy`

Basic annotations(2)

- context
  - `@Scope`
  - `@Bean`
  - `@DependsOn`
  - `@Lazy`
- transactional
  - `@Transactional`

Stereotypical



Basic configuration

```
<!-- looks for annotations on beans -->
<context:annotation-config/>


<!- scan stereotyped classes and register BeanDefinition -->
<context:component-scan
    base-package="org.exhanger.repository.map,org.exhanger.api">
```

Basic configuration(2)

```
<context:component-scan
    base-package="org.exhanger"
    name-generator="my.NameGeneratorImpl"
    resource-pattern="**/*.class"
    scope-resolver="my.ScopeResolverImpl"
    scoped-proxy="no|interfaces|targetClass">

  <context:include-filter type="annotation"
      expression="org.springframework.stereotype.Service"/>
  <context:exclude-filter type="regex"
      expression=".*MapRepository"/>

</context:component-scan>
```

@Value

```
@Component("authenticationProvider")
public class EjbAuthenticationProvider {

    /** The provider api connection url. */
    @Value("${provider.api.url}")
    private String apiUrl;

    /** The provider api connection username. */
    private @Value("${provider.api.username}") String apiUsername;

    /** The provider api connection password. */
    @Value("${provider.api.password}")
    private String apiPassword;
}
```

@Autowired(1)

```
@Service("accountService")
public class AccountServiceImpl {
  @Autowired
  private AccountRepository repository;

  @Autowired
  public AccountServiceImpl(AccountRepository repository) {}

  @Autowired(required = false)
  public void setRepository(AccountRepository repository) {}

  @Autowired
  public void populate(Repository r, OrderService s) {}
}
```

@Autowired(2)

```
@Service
public class AccountServiceImpl {

  @Autowired
  private AccountRepository[] repositories;

  @Autowired
  public AccountServiceImpl(Set<AccountRepository> set) {}

  @Autowired(required = false)
  public void setRepositories(Map<String,AccountRepository> map){
  }
}
```

@Required(1)

```
public class AccountServiceImpl {
  private AccountRepository repository;

  @Required
  public void setRepository(AccountRepository repository) {}
}
```

```
<bean name="accountService" class="AccountServiceImpl">
  <property name="repository" ref="accountRepository"/>
</bean>
```

@Required(2)

```
public class AccountServiceImpl {
  private AccountRepository repository;

  @Mandatory
  public void setRepository (AccountRepository repository) {}
}
```

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor">
  <property name="requiredAnnotationType" value="my.Mandatory"/>
</bean>
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Mandatory {}
```

@Qualifier

```
public class ReportServiceImpl {
  @Autowired
  @Qualifier("main")
  private DataSource mainDataSource;

  @Autowired
  @Qualifier("freeDS")
  private DataSource freeDataSource;
}
```

```
<beans>
  <bean class="org.apache.commons.dbcp.BasicDataSource">
    <qualifier value="main"/>
  </bean>
  <bean id="freeDS" class="org.apache.commons.dbcp.BasicDataSource"/>
</beans>
```

@Resource

```
public class AccountServiceImpl {
  @Resource(name = "orderService")
  private OrderService orderService;

  @Resource(name = "orderService")
  public void setOrderService(OrderService orderService) {}

  @Resource
  private AuditService auditService;

  @Resource
  public void setAuditService(AuditService auditService) {}
}
```

@Scope

```
@Service
@Scope("prototype")
public class AccountServiceImpl implements AccountService {
}
```

```
@Service
@Scope(value = BeanDefinition.SCOPE_PROTOTYPE,
       proxyMode = ScopedProxyMode.TARGET_CLASS)
public class AccountServiceImpl implements AccountService {
}
```

@Lazy & @DependsOn

```
@Lazy
@Service
@DependsOn({"orderService", "currencyService"})
public class AccountServiceImpl implements AccountService {
}
```

Factory method component

```
@Component
public class CurrencyRepositoryFactory {

  @Bean
  @Lazy
  @Scope("prototype")
  @Qualifier("public")
  public CurrencyRepository getCurrencyRepository() {
    return new CurrencyMapRepository();
  }
}
```

JSR 330

```
@Named
@Singleton
public class AccountServiceImpl implements AccountService {

  @Inject
  private AccountRepository repository;

  @Inject
  public AccountServiceImpl(@Named("default") AccountRepository r) {}

  @Inject
  public void setAccountRepository(AccountRepository repository) {}

}
```

@Autowired vs @Inject

Spring	JSR-330	JSR-330 restrictions
`@Autowired`	`@Inject`	has no 'required' attribute
`@Component`	`@Named`	
`@Scope`	`@Scope`	only for meta-annotations and injection points
`@Scope`	`@Singleton`	default scope is like 'prototype'
`@Qualifier`	`@Named`	
`@Value`	X	
`@Required`	X	
`@Lazy`	X	

Java-based configuration

Configuration

Basic annotations

Instantiating for standalone

- simple

```
new AnnotationConfigApplicationContext(ExchangerConfig.class);
```

- programmaticaly

```
context = new AnnotationConfigApplicationContext();
context.register(ExchangerConfig.class);
context.refresh();
```

- scanning

```
context = new AnnotationConfigApplicationContext();
context.scan("org.exchanger.config");
context.refresh();
```

Instantiating bean

```
@Configuration
public class ExchangerRepositoryConfg {

  @Bean(name = "accountRepository", initMethod = "init")
  public AccountRepository accountRepository() {
    return new AccountMapRepository();
  }

  @Bean
  public AuditRepository auditRepository() {
    return new AuditMapRepository();
  }
}
```

@Import

```
@Configuration
@Import({RepositoryConfig.class, ServiceConfig.class})
public class ExchangerConfig {
}
```

```
@Configuration
public class RepositoryConfg {}
```

```
@Configuration
public class ServiceConfig {}
```

@ImportResources

```
@Configuration
@ImportResource("classpath:jdbc.properties")
public class ExchangerConfig {

  @Value("jdbc.url")
  private String jdbcUrl;

  @Bean
  public DataSourse dataSource() {
    return new SimpleDataSource(jdbcUrl);
  }
}
```

Dependency injection

```
@Configuration
public class ExchangerServiceConfig {

  @Autowired
  private CurrencyRepository currencyRepository;

  public @Bean CurrencyService currencyService() {
    return new CurrencyServiceImpl(currencyRepository);
  }

  @Bean(name = {"orderBuilder", "builder"})
  public OrderBuilder orderBuilder() {
    return new OrderBuilder(currencyService(), accountService());
  }
}
```

Summary

Benefits of IoC

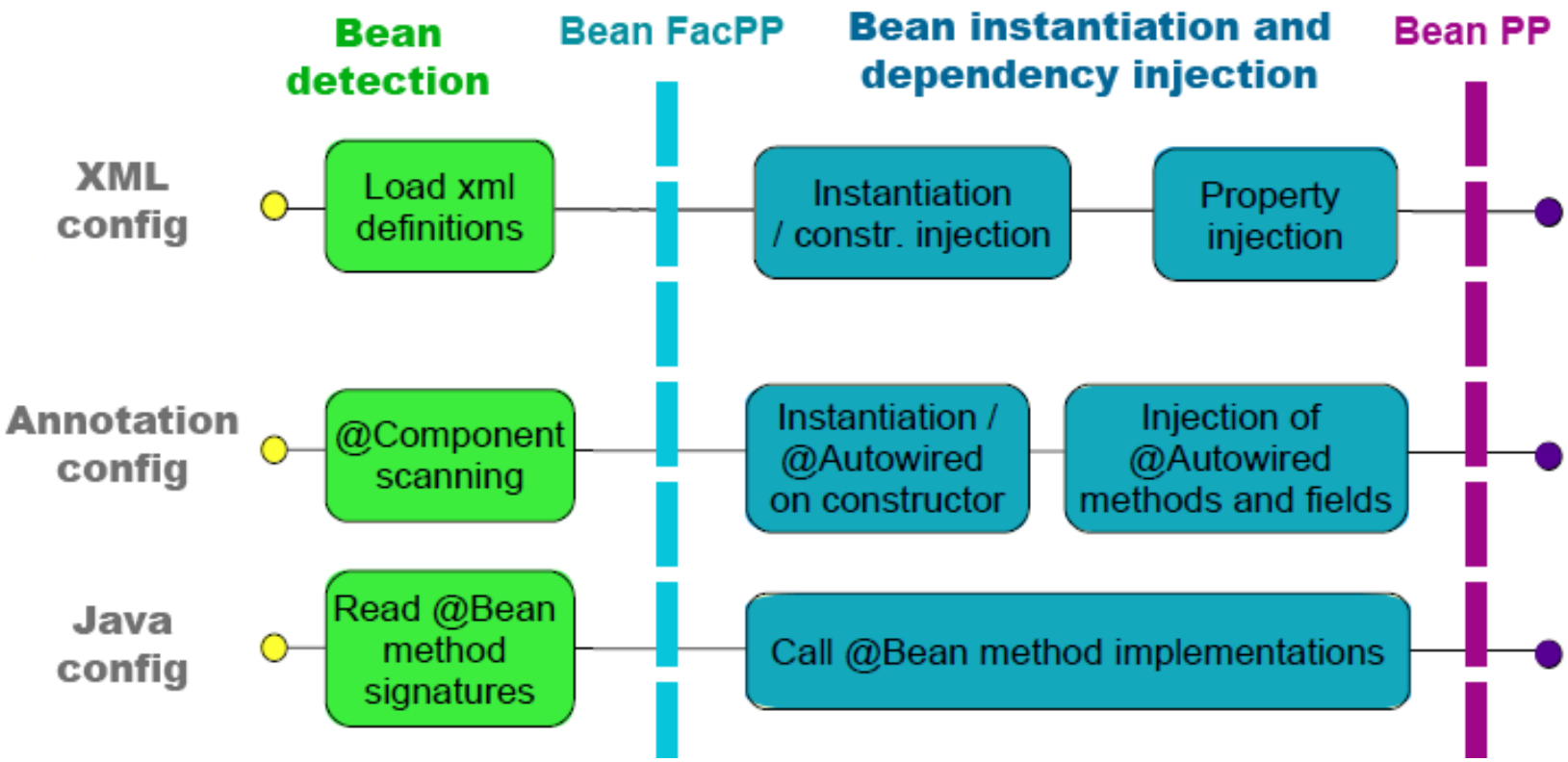
Context lifecycle

Mix and match

Benefits of IoC

- **minimizes** the amount of code
- make application more **testable**
- promote programming to **interfaces**
- loose **coupling** with minimal effort
- support **eager** instantiation and **lazy** loading
- provide control over object **lifecycle**

Context lifecycle



Approach to configuration

- XML
  - infrastructure beans
- annotations
  - working beans
- java
  - an alternative to the FactoryBean

You can mix and match

- dependency injection
  - constructor-based and setter-based
  - you can mix and match
- configuration metadata
  - XML, annotations, Java
  - you can mix and match
- annotations
  - own, JSR-250, JSR-330
  - you can mix and match

Links

- site

  http://www.springsource.org/

- reference

  http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/

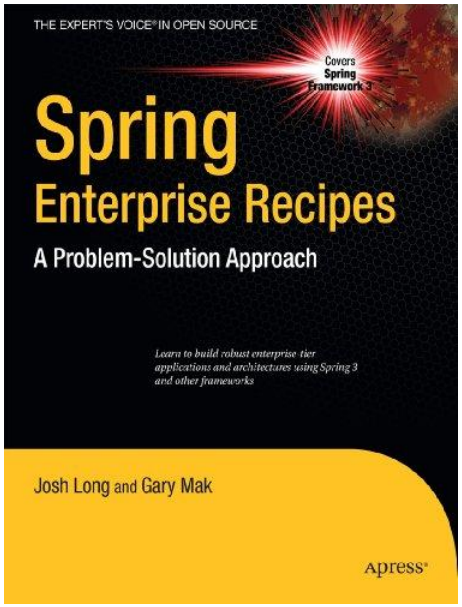
- blog

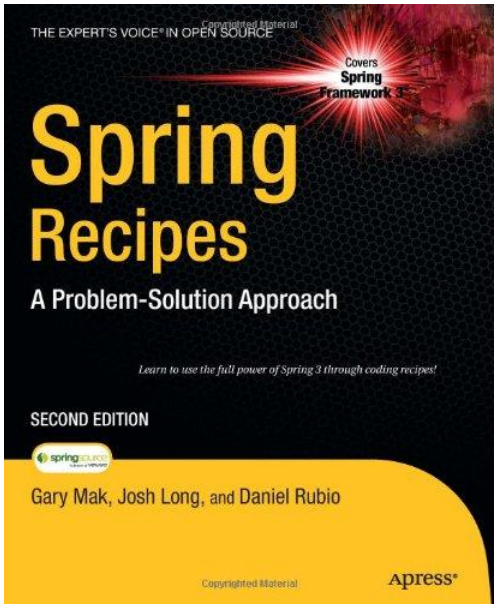
  http://blog.springsource.com/category/spring/

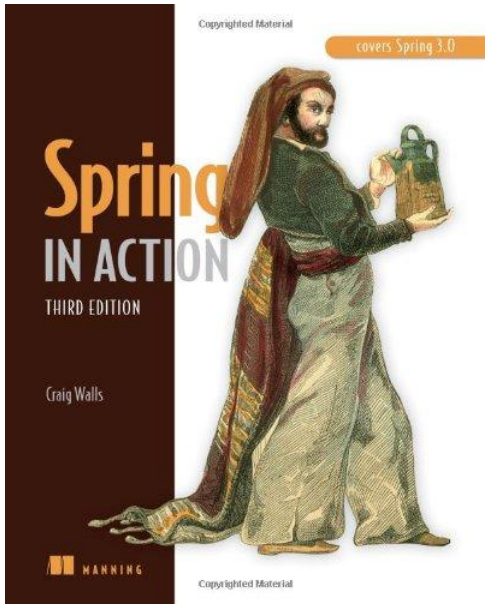
- forum

  http://forum.springsource.org/forumdisplay.php?f=26

Books


THE EXPERT'S VOICE® IN OPEN SOURCE
Covers Spring Framework 3
Spring
Enterprise Recipes
A Problem-Solution Approach
Learn to build robust enterprise tier applications and architectures using Spring 3 and other frameworks
Josh Long and Gary Mak
Apress


THE EXPERT'S VOICE® IN OPEN SOURCE
Covers Spring Framework 3
Spring
Recipes
A Problem-Solution Approach
Learn to use the full power of Spring 3 through coding recipes!
SECOND EDITION
Gary Mak, Josh Long, and Daniel Rubio
Apress


covers Spring 3.0
Spring
IN ACTION
THIRD EDITION
Craig Walls
MANNING

Questions

The end

noskov.d@Gmail.com

my LinkedIn profile

http://www.linkedin.com/in/noskovd

slideshare

http://www.slideshare.net/analizator/presentations