

SpringFramework & Spring Portfolio

Speaker

- 4 years Java professional experience
- 3 years Spring professional experience
- Development small, medium and big mission critical applications as well
- Strong relationship to SpringSource Community

Agenda

- Short introduction to Spring - Spring is not DI container.
- What's new in Spring 2.5
- Introduction to Spring Portfolio projects.

Vocabulary

- Spring
- Spring Framework
- Interface 21
- SpringSource
- Spring Portfolio

Spring from 1.000.000 feet

- Spring is a Lightweight Application Framework
- Spring can address all tiers of an application (in comparison to Struts, WebWork, EJB)

Spring From 100 000 feet

- Much more than IoC – solution to enterprise development
- Core components model (POJO programming level)
- Works on .NET

Core components model

- Services – enterprise and custom services
- Patterns (recipes) – degree of consistency (templates, observer and more)
- Integration(ecosystem) – vendor products, open source projects
- Portability (runs everywhere, nearly 😊)

History

- 2002/2003 Started as a framework developed around Rod Johnson's book Expert One-on-One J2EE Design and Development
- March 2004 Spring 1.0 released (Interface21)
- 2004/2005 Spring is emerging as a leading full-stack Java/J2EE application framework
- June 2006 Spring 2.0 released
- November 2007 SpringSource Spring Portfolio
- October 2007 Spring 2.5 released
- August 2008 Spring 3.0 RC1 ??

Motivation

J2ee problems:

- Complexity
- Checked exceptions
- Testability
- No OO promoting
- Entity beans
- Based on vision of distributed objects
- Monolithic J2EE server seen as runtime for everything

The Spring Framework Mission Statement

The authors of Spring believe that:

- 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 in Java. A framework shouldn't force you to catch exceptions you're unlikely to be able to recover from.
- Testability is essential, and a framework such as Spring should help make your code easier to test.

Spring Framework Mission Statement (continued)

The authors of Spring aim that:

- Spring should be a pleasure to use
- Your application code should not depend on Spring APIs
- Spring should not compete with good existing solutions, but should foster integration. (For example, JDO and Hibernate are great O/R mapping solutions. We don't need to develop another one.)

Spring Framework Key Abstractions

- Bean
- ApplicationContext

```
<bean id="timeMeasureInterceptor"  
class="foo.bar.TimeMeasureInterceptor">  
</bean>
```

```
<bean id="individualDAO,, class="foo.bar.IndividualDAO">  
<property name="sessionFactory" ref="sessionFactory" />  
<property name="doIDictionaryDAO" ref="doIDictionaryDAO" />  
</bean>
```

```
<bean id="individualManager"  
      class="foo.bar.IndividualManager">  
<property name="individualDAO" ref="individualDAO" />  
<property name="accountDAO" ref="accountDAO" />  
</bean>
```

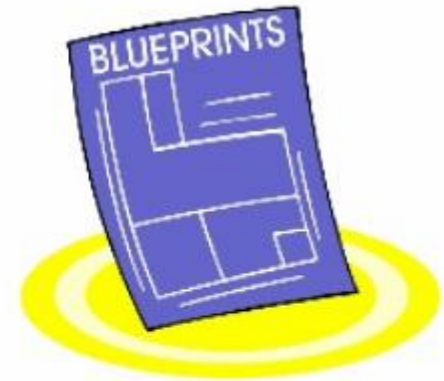
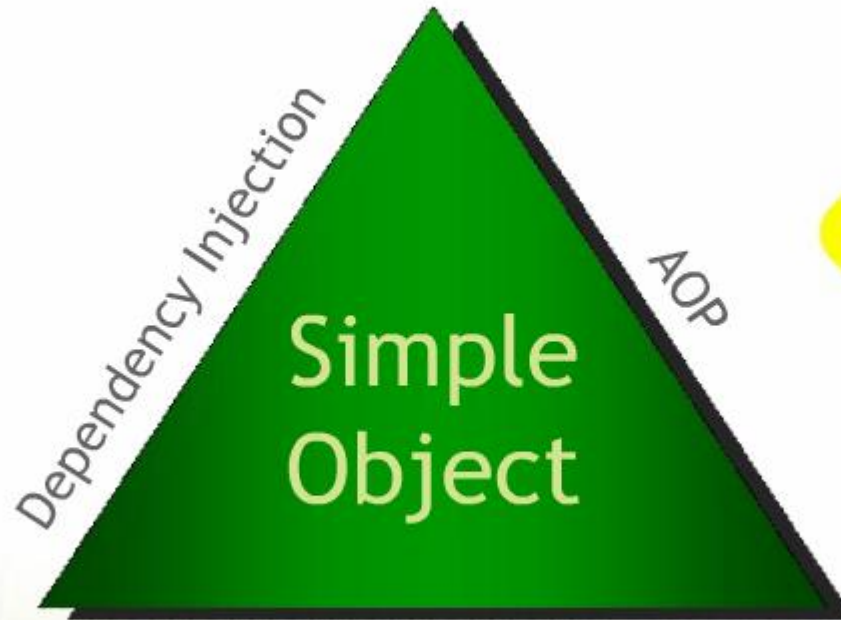
Spring Framework

At it's core, Spring provides:

- An Inversion of Control Container
- An AOP Framework
- A Service Abstraction Layer

Spring Framework

Enabling Technologies



Portable Service Abstractions (PSA)

Inversion of Control(IoC)

- The “Hollywood Principle”: Don’t call me, I’ll call you
- Eliminates lookup code from within your application
- Allows for pluggability and hot swapping
- Promotes good OO design
- Enables reuse of existing code
- Makes your application extremely testable

DI

- Injection of configuration/collaborators
- Various configuration options:
 - Spring Java Config
 - XML
 - Annotations
- IoC in various environments:
 - standalone/Junit
 - Web
 - EJB
 - JCA
 - OSGI

DI

- Bean scopes:
 - Proxy
 - Singleton
 - Http session
 - Global Session
 - Http request
 - Custom scope

Bean instantiation methods

- Constructor
- Static factory Method
- Instance factory method
- FactoryBean

Constructor

```
public class MyBean1 {  
    private String prop;  
    private MyBean2 collaborator;  
    public MyBean1( String prop, MyBean2 collaborator ) {  
        this.prop = prop;  
        this.collaborator = collaborator;  
    }  
}
```

```
<bean id="myBean1" class="foo.bar.model.MyBean1">  
    <constructor-arg value="foo"/>  
    <constructor-arg ref="myBean2"/>  
</bean>
```

Static factory Method

```
public class Factory {  
    public static FactoryProduct createProduct(String dir){  
  
        return new FactoryProduct();  
    }  
}
```

```
<bean id="factory" class="foo.bar.model.Factory" factory-  
method="createProduct" scope="singleton">  
    <constructor-arg value="methodParam"/>  
    <property name="nazwa" value="sth"></property>  
</bean>
```

Instance factory method

```
public class MyBean1 {
    private String prop;
    private MyBean2 collaborator;
    public MyBean1( String prop, MyBean2 collaborator ) {
        this.prop = prop;
        this.collaborator = collaborator;
    }
    public long calculate(){
        return prop.length() * collaborator.getAge();
    }
}
```

```
<bean id="someThing" factory-bean="myBean1" factory-
method="calculate" scope="singleton">
</bean>
<bean id="myBean1" class="foo.bar.model.MyBean1">
    <constructor-arg value="foo"/>
    <constructor-arg ref="myBean2"/>
</bean>
```

FactoryBean

```
public class Factory2 implements FactoryBean {  
    public Object getObject() throws Exception {  
        return Calendar.getInstance();  
    }  
    public Class getObjectType() {  
        return Calendar.class;  
    }  
    public boolean isSingleton() {  
        return true;  
    }  
}
```

```
<bean id="myBean1" class="foo.bar.model.Factory2">  
</bean>
```

DI

- Dependencies resolving:
 - Injection of straight values
 - Injection of collections
 - Injection of null
 - Injection of collaborators

Injection of Straight Values

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-  
method="close">  
  <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>  
  <property name="password">  
    <value>masterkaoli</value>  
  </property>  
  <property name="maxWait" value="12" />  
</bean>
```

Injection of collections

```
<bean id="sessionFactory"  
  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">  
  <property name="dataSource" ref="dataSourcePool" />  
  <property name="hibernateProperties">  
    <props>  
      <prop key="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</prop>  
      <prop key="hibernate.show_sql">>true</prop>  
      <prop key="hibernate.format_sql">>true</prop>  
      <prop key="hibernate.use_sql_comments">>true</prop>  
      <prop key="hibernate.generate_statistics">>false</prop>  
      <prop  
key="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</prop>  
    </props>  
  </property>  
  <property name="eventListeners">  
    <map>  
      <entry key="merge">  
        <bean  
class="org.springframework.orm.hibernate3.support.IdTransferringMergeEventListener"  
/></entry>  
    </map>  
  </property>  
</bean>
```

Injection on collaborators

```
<bean id="sessionFactory"  
  class="org.springframework.orm.hibernate3.annotation.AnnotationS  
  essionFactoryBean">  
  <property name="dataSource" ref="dataSourcePool" />  
  <property name="lobHandler" ref="lobHandler"/>  
</bean>  
  
<bean id="lobHandler"  
  class="org.springframework.jdbc.support.lob.OracleLobHandler" >  
  <property name="nativeJdbcExtractor">  
    <bean id="nativeJdbcExtractor"  
      class="org.springframework.jdbc.support.nativejdbc.WebSpher  
      eNativeJdbcExtractor"/>  
  </property>  
</bean>
```

DI

- Injecting dependencies:
 - Constructor Injection
 - Setter Injection
 - Method Injection
 - Arbitrary method replacement

Constructor & Setter Injection

```
public class ConstructorInjectionBean {
    private String property;
    private IWorker worker;
    public ConstructorInjectionBean( String property, IWorker worker ) {
        this.property = property;
        this.worker = worker;
    }
    public void doIt(){
        worker.doSth();
    }
}
```

```
public class DummyImpl implements IWorker {
    private String foo;
    public String getFoo() {
        return foo;
    }
    public void setFoo( String foo ) {
        this.foo = foo;
    }
    public void doSth() {
    }
}
```

```
<bean id="worker" class="foo.bar.model.DummyImpl">
    <property name="foo" value="foo"/>
</bean>
<bean id="constructorInjectionBean" class="foo.bar.model.ConstructorInjectionBean">
    <constructor-arg value="foo"/>
    <constructor-arg ref="worker"/>
</bean>
```

Method Injection

```
public abstract class Controller {  
    public void process(){  
        ICommand command = getCommand();  
        command.doSth1();  
        command.doSth2();  
    }  
    public abstract ICommand getCommand();  
}
```

```
public class Command implements ICommand{  
    public void doSth1() {  
    }  
    public void doSth2() {  
    }  
}
```

```
<bean id="command" class="foo.bar.Command" scope="prototype"/>  
  
<bean id="controller" class="foo.bar.Controller">  
    <lookup-method bean="command" name="getCommand"/>  
</bean>
```

Arbitrary method replacement

```
public class BeanWithMethodToReplace {  
    public Long toReplace(String sth){  
        return new Long(sth);  
    }  
    public void sth(){  
    }  
}
```

```
public class MethodReplacer implements  
    org.springframework.beans.factory.support.MethodReplacer {  
    public Object reimplement( Object obj, Method method, Object[] args ) throws Throwable {  
        String input = (String) args[0];  
        return new Long(input) *2;  
    }  
}
```

```
<bean id="beanWithMethodToReplace" class="foo.bar.BeanWithMethodToReplace">  
    <replaced-method name="toReplace" replacer="replacer"/>  
</bean>  
  
<bean id="replacer" class="foo.bar.MethodReplacer"/>
```

AOP

- Injection of behaviour
- OOP and AOP programming units
- AOP and OOP are not competitors

Crosscutting concern

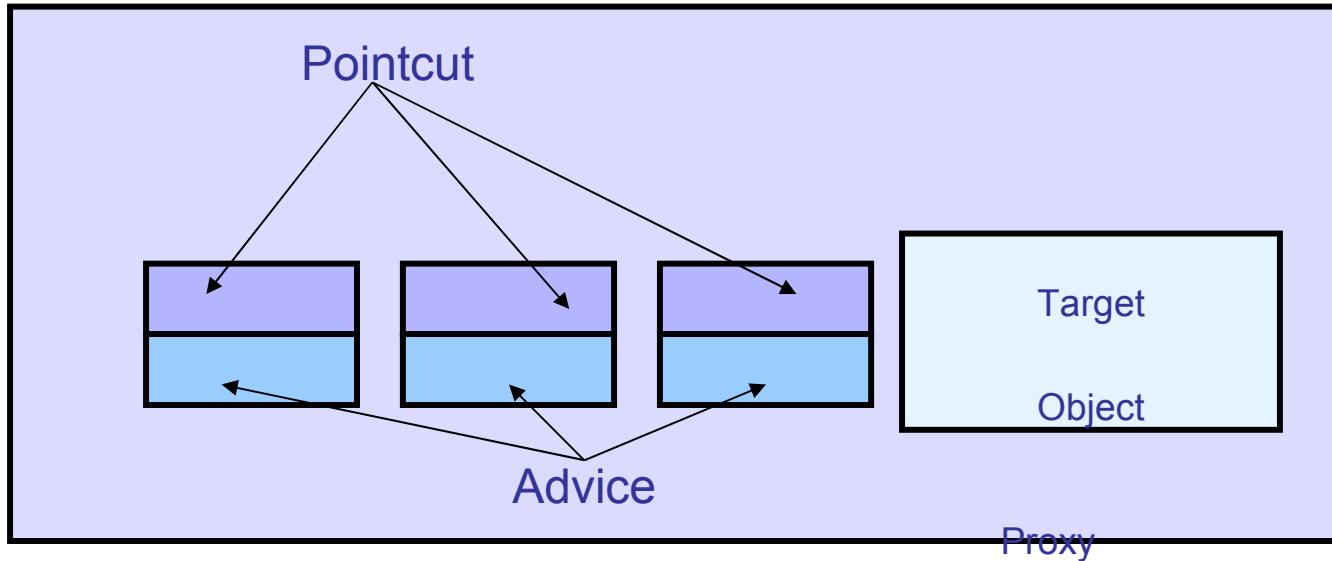
- Functionality whose implementation spans multiple modules
- Symptoms:
 - Code tangling
 - Code scattering
- Examples:
 - Declarative transaction management
 - Custom security requirements
 - Application or company-specific aspects
 - Caching
 - Process control
 - Monitoring

Core AOP concepts

- Join point
 - An identifiable point in the execution of a program.
 - Central, distinguishing concept in AOP
- Pointcut
 - Program construct that selects join points and collects context at those points.
- Advice
 - Code to be executed at a join point that has been selected by a pointcut (“what to do at a joinpoint”)
 - *Before advice*
 - *After returning advice*
 - *After throwing advice*
 - *After (finally) advice*
 - *Around advice*
- Introduction
 - Additional data or method to existing types, implementing new interfaces

Spring AOP

- A proxy-based AOP framework
 - JDK proxies
 - CGLIB proxies



Advantages of proxy-based approach

- Requires no special compiler
- Allow per-object interceptors
 - Not per-class
- Allows easing into AOP
- Easy to modify applicable interceptors dynamically

Disadvantage of proxy-based approach

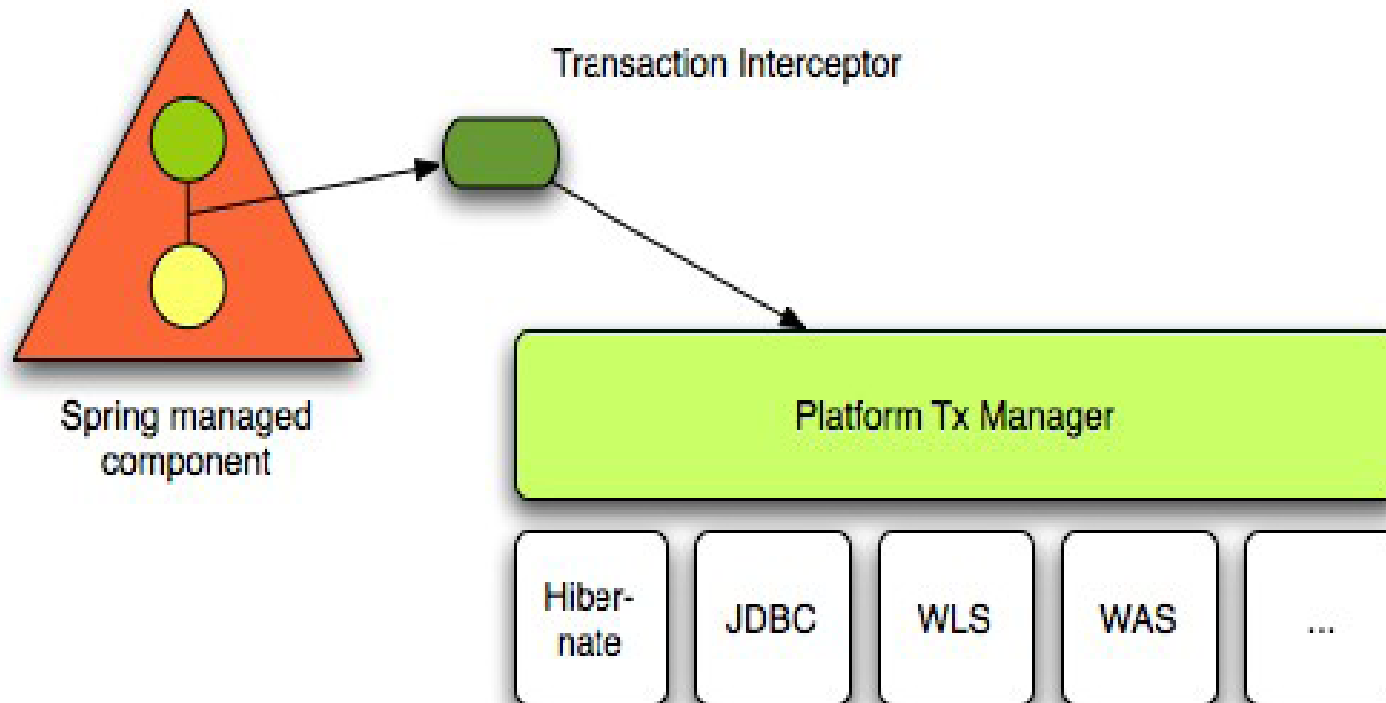
- Limitation of method-only interception
 - No field-access or object creation
- Explicit creation of proxy required
 - Can't use 'new' to create objects
- Calls to 'self' don't go through interceptors
- Problems with final method/classes

Portable Service Abstractions

- Enterprise services layer built on top of kernel
- Services that address problems of enterprise applications
- Out-of-box enterprise services and custom services

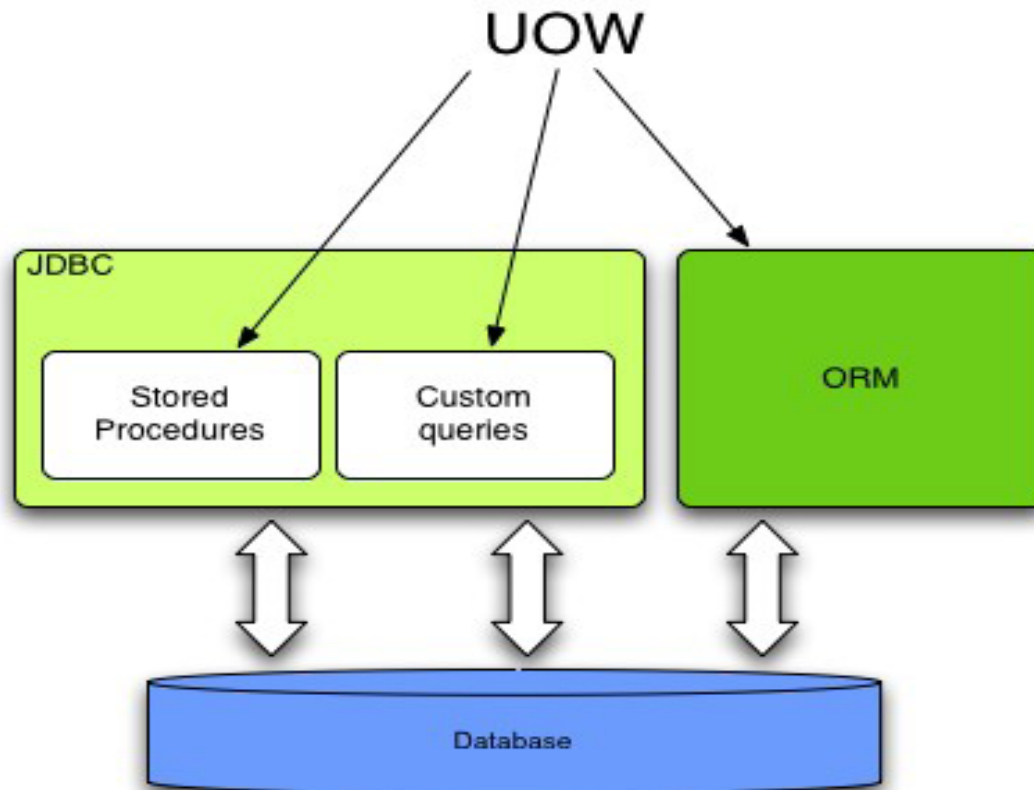
Enterprise Services Layer

- Transaction Management



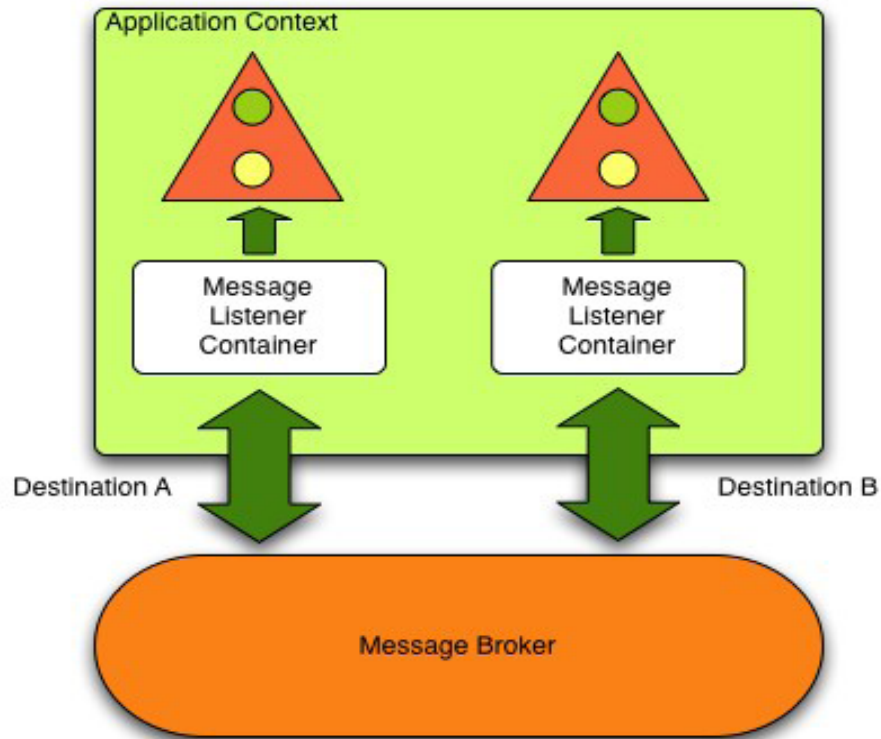
Enterprise Services Layer

- Data Access



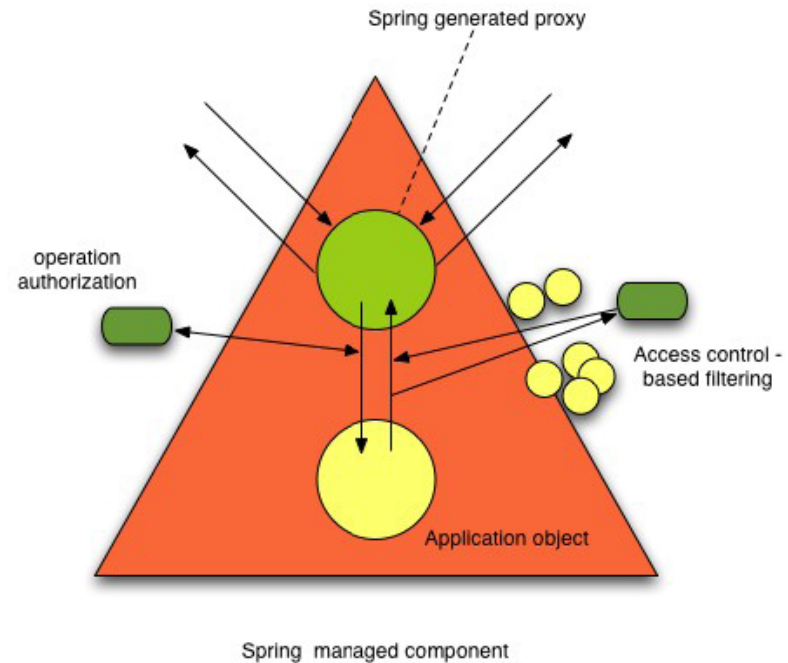
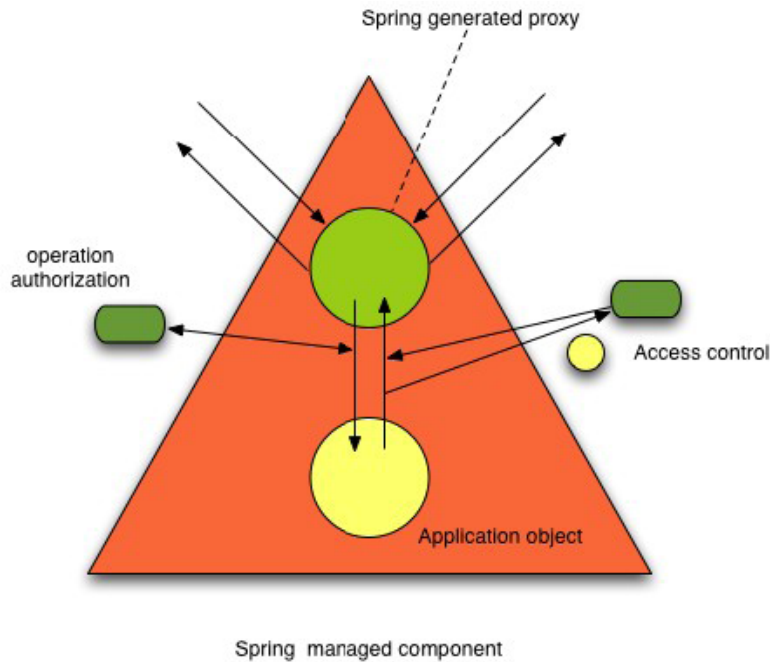
Enterprise Services Layer

- Messaging



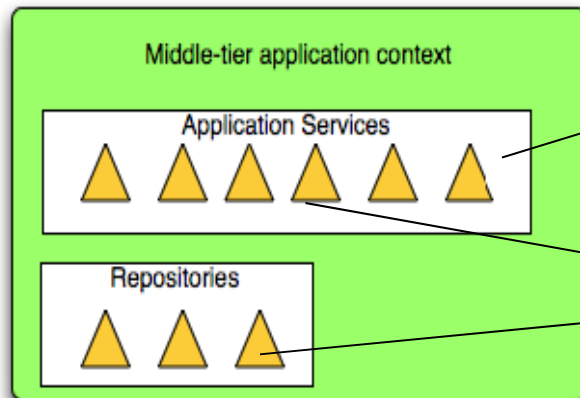
Enterprise Services Layer

- Security Management



Enterprise Services Layer

- JMX integration

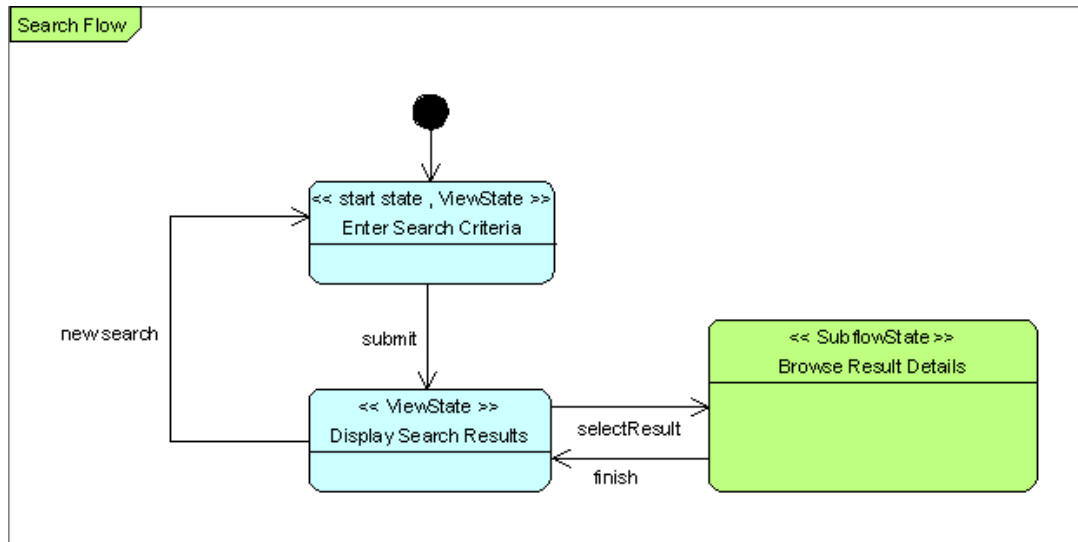


The screenshot shows the "J2SE 1.5 Monitoring & Management Console: localhost:0 (Monitoring Self)". The interface includes a "Connection" header and several tabs: "Summary", "Memory", "Threads", "Classes", "MBeans", and "VM". The "MBeans" tab is selected, displaying a tree view of MBean instances. The tree is expanded to show the "Memory" section, with "Eden Space" selected. The right-hand pane shows the "Attributes" tab for the selected MBean, displaying a table of attributes and their values.

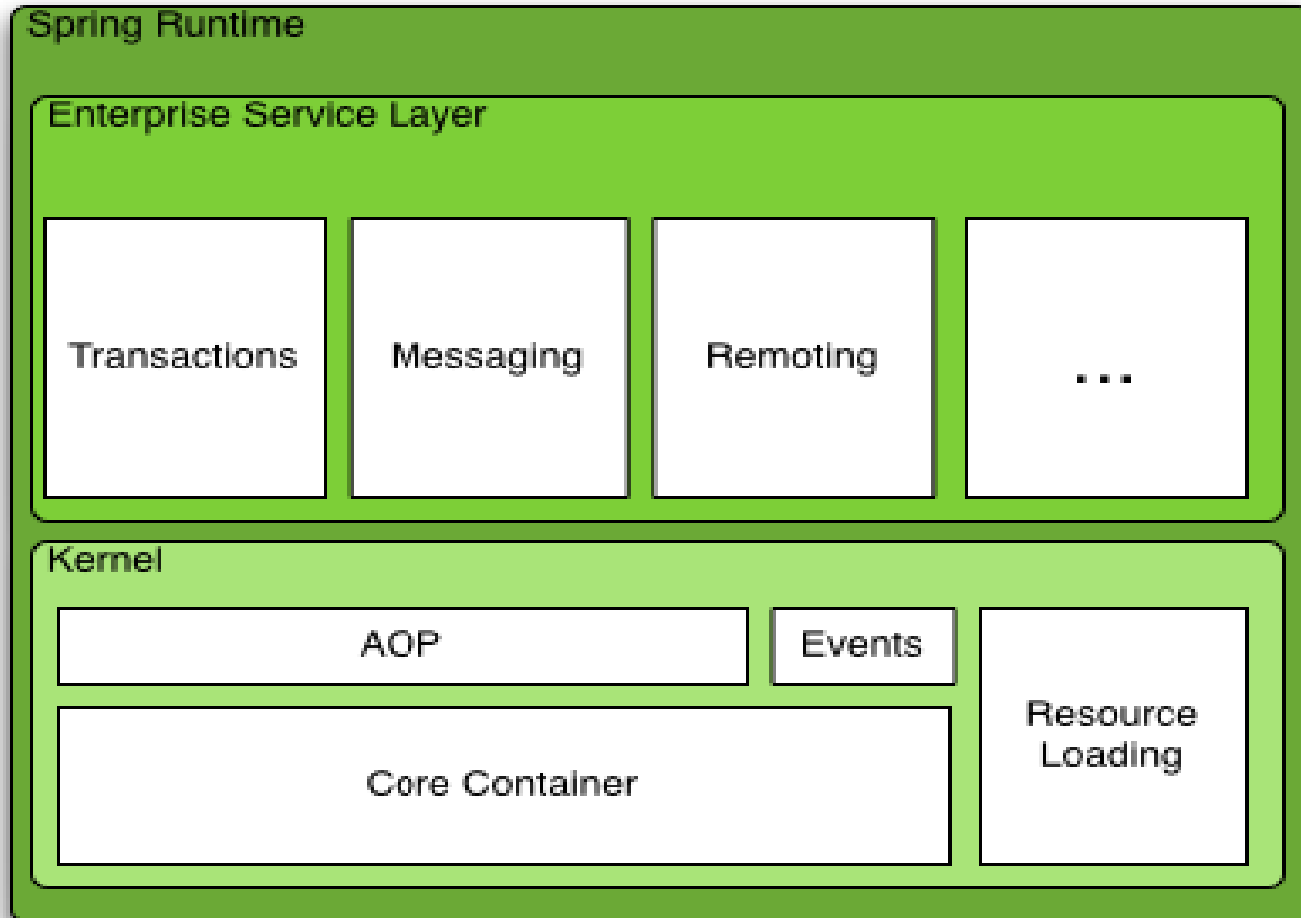
| Name | Value |
|--------------------------------------|---------------------------------|
| CollectionUsage | javax.management.openmbean.C... |
| CollectionUsageThreshold | 0 |
| CollectionUsageThresholdCo... | 0 |
| CollectionUsageThresholdEx... | true |
| CollectionUsageThresholdSu... | true |
| MemoryManagerNames | java.lang.String[2] |
| Name | Eden Space |
| PeakUsage | javax.management.openmbean.C... |
| Type | HEAP |
| Usage | javax.management.openmbean.C... |
| UsageThreshold | Unavailable |
| UsageThresholdCount | Unavailable |
| UsageThresholdExceeded | Unavailable |
| UsageThresholdSupported | false |
| Valid | true |

At the bottom of the console window, there is a "Refresh" button.

- Web requests



Runtime Overview



Spring Framework 2.5

New Platform support

- Java 6 Support
- Java EE 5
- OSGi

JDK 1.6

- New JDK 1.6 API's supported:
 - JDBC 4.0
 - JMX MBeans
 - JDK ServiceLoader API
- JDK 1.4 and 1.5 still fully supported
- JDK 1.3 no longer supported

JDBC support

- JDBC 4.0 feature support
 - Native connections (`java.sql.Wrapper`)
 - LOB Handling (`setBlob/setClob`)
 - New **SQLException** subclasses
- Other JDBC improvements:
 - **SimpleJdbcTemplate**
 - **SimpleJdbcCall & SimpleJdbcInsert**

JMX MXBeans

...

JDK ServiceLoader API

- Registration of service providers for services
 - META-INF/services/foo.bar.service defines implementing classes for foo.bar.**service**
- Used by Service(List)FactoryBean

```
<bean id="sqlDriver"  
class="org.sfw...ServiceFactoryBean">  
<property name="serviceType"  
value="java.sql.Driver"/>  
</bean>
```

Support for Built-in HTTP Server

- HTTP-based Remoting using
 - **SimpleHttpInvokerServiceExporter**
 - **SimpleHessian/BurlapServiceExporter**
- Set-up JRE 1.6 HttpServer using
 - **SimpleHttpServerFactoryBean**

Java EE 5 support

- Support for Java EE 5
- New Java EE 5 API's supported:
 - Servlet 2.5, JSP 2.1 & JSF 1.2
 - JTA 1.1, JAX-WS 2.0 & JavaMail 1.4
- J2EE 1.4 and 1.3 still fully supported

Spring Framework 2.5

- J2EE API:
 - JSF 1.2: **SpringBeanFacesELResolver**
 - Consistent use of JSR-250 annotations
 - JTA 1.1: support new
TransactionSynchronization-Registry

Spring Framework 2.5

- Deploy Spring app as RAR file
 - Add a META-INF/ra.xml file that contains special adapter that references a Spring applicationContext.xml file
- 2.5 officially supported on IBM WAS 6.x
 - Websphere V6.0.2.19 or V6.1.0.9
 - (WebSphereUowTransactionManager)**

OSGI

- 2.5 jars now are compliant bundles
- Integration with OSGi provided by the **Spring Dynamic Modules for OSGi(tm) Service Platforms**
 - Integration with OSGi service registry
 - ApplicationContext per bundle

New Configuration features

- Annotation-driven configuration
- JMS & JCA support
- Enhanced AspectJ support
- Annotation-driven MVC Controllers

Annotations in Spring

- Annotations in Spring
 - JSR-250 Common Annotations
 - Spring **Autowired** Annotation
 - Component scanning for autodetection of components
- Additional configuration option
 - XML is in no way deprecated!
 - Different from Spring JavaConfig

JSR-250 Common Annotations

```
public class GreetingServiceImpl implements GreetingService {
    @Resource(name="jdbcMessageRepository")
    private MessageRepository messageRepository;
    private MessageRepository messageRepository2;

    @Resource(name="stubMessageRepository")
    public void asdasdsad( MessageRepository messageRepository2 ) {
        this.messageRepository2 = messageRepository2;
    }
    @PostConstruct
    public void initialize() { ... }
    @PreDestroy
    public void shutdown() { ... }
}
```

```
<context:annotation-config />
```

```
<bean id="jdbcMessageRepository" class="blog.JdbcMessageRepository"/>
```

```
<bean id="stubMessageRepository" class="blog.StubMessageRepository"/>
```

```
<bean id="greetingServiceImpl" class="blog.GreetingServiceImpl"/>
```

Further Java EE 5 Annotations

- `@WebServiceRef` / `@EJB`
 - injecting a JAX-WS / EJB 3 service proxy
- `@TransactionAttribute`
 - EJB 3 transaction demarcation
- EJB 3 transaction demarcation
 - `@PersistenceContext` / `@PersistenceUnit`
 - JPA resource injection (since 2.0)

@Autowired

- Autowiring by type (hints - @Qualifier)
- Of fields, methods and constructors

```
public class GreetingServiceImpl implements GreetingService {
    private MessageRepository messageRepository;
    private MessageRepository messageRepository2;
    @Autowired
    @Qualifier("httpMessageRepository")
    private MessageRepository messageRepository3;
    @Autowired
    public void inject( @Qualifier("stubMessageRepository")
    MessageRepository messageRepository2 ) {
        this.messageRepository2 = messageRepository2;
    }
    @Autowired
    public GreetingServiceImpl( @Qualifier("jdbcMessageRepository")MessageRepository
    messageRepository ) {
        this.messageRepository = messageRepository;
    }
}
```

@Qualifier

```
public class GreetingServiceImpl implements GreetingService {  
    private MessageRepository messageRepository2;  
    @Autowired  
    @Qualifier("httpRepository")  
    private MessageRepository messageRepository3;  
    @Autowired  
    public void inject( @Qualifier(" httpRepos ")  
        MessageRepository messageRepository2 ) {  
        this.messageRepository2 = messageRepository2;  
    }  
}
```

```
<bean id="httpMessageRepository" class="foo.bar.HttpMessageRepository">  
    <qualifier value="httpRepos"/>  
</bean>  
<bean id="stubMessageRepository" class="foo.bar.StubMessageRepository"/>
```

Custom qualifier annotations

```
@Target({ElementType.FIELD, ElementType.PARAMETER})  
@Retention(RetentionPolicy.RUNTIME)  
@Qualifier  
public @interface Genre {  
    String value();  
}
```

```
public class MovieRecommender {  
    @Autowired @Genre("Action")  
    private MovieCatalog actionCatalog;  
    private MovieCatalog comedyCatalog;  
    @Autowired  
    public void setComedyCatalog(  
        @Genre("Comedy") MovieCatalog comedyCatalog){  
        this.comedyCatalog = comedyCatalog;  
    }  
}
```

```
<bean class="example.SimpleMovieCatalog">  
    <qualifier type="example.Genre" value="Action"/>  
</bean>  
<bean class="example.SimpleMovieCatalog">  
    <qualifier type="example.Genre" value="Comedy"/>  
</bean>
```

@Autowired variations

```
public class GreetingServiceImpl3 implements GreetingService {
    @Autowired
    private List<MessageRepository> repositories;
    @Autowired(required=false)
    private Strategy strategy = new DefaultStrategy();
    private String prop;
    public void setProp( String prop ) {
        this.prop = prop;
    }
    @Autowired
    public void injectContext( ApplicationContext applicationContext, BeanFactory beanFactory ) {}
}
```

```
<bean id="jdbcMessageRepository" class="foo.bar.JdbcMessageRepository" />
<bean id="stubMessageRepository" class="foo.bar.StubMessageRepository" primary="true"/>
<bean id="httpMessageRepository" class="foo.bar.HttpMessageRepository"/>
<bean id="springStrategy" class="foo.bar.SpringStrategy"/>
<bean id="greetingServiceImpl3" class="foo.bar.GreetingServiceImpl3">
    <property name="prop" value="foo"></property>
</bean>
```

Annotation-based Autowiring

- Pros:
 - Self-contained: XML config reduction
 - Works in much more cases than generic autowiring (any method or field)
 - JSR-250 or custom annotations keep your code from depending on Spring
- Cons:
 - Requires classes to be annotated (@POJO)
 - Configuration only per class, not per instance
 - Changes require recompilation

Component scanning

- Annotated Components
 - with `@Component`
 - Or annotation which has `@Component`
 - Spring has `@Repository`, `@Service` and `@Controller` stereotypes
- Easy to create your own components

Annotated Components Example

@Component

```
public class GreetingServiceImpl implements GreetingService {  
    @Autowired  
    public GreetingServiceImpl( @Qualifier("jdbcMessageRepository")MessageRepository  
        messageRepository ) {}  
}
```

@Repository

```
public class JdbcMessageRepository implements MessageRepository {  
    ...  
}
```

```
<!-- Not even a plain XML <bean> tag is necessary! -->  
<context:component-scan base-package="foo.bar" annotation-config="true"/>
```

@Component

- Provide bean name using value element:
 - @Component("myService")
 - Defaults to uncapitalized short class name:
mypackage.MyService => myService
- Create your own stereotype annotations

```
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Component  
public @interface Manager {  
    String value() default "";  
}
```

Component scanning configuration

- Scan filters
 - RegExpr
 - Annotations
 - Aspectj
 - Assignable
 - Define your own strategy

```
<context:component-scan base-package="foo.bar">
  <context:include-filter type="regex"
    expression=".*Stub.*Repository"/>
  <context:exclude-filter type="annotation"
    expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

Component Scanning Pros

- XML reduction - unless you need the greater sophistication it allows
- Changes are picked up automatically
- Works great with Annotation Driven Injection
 - picking up further dependencies with `@Autowired`
- Highly configurable

Component Scanning Cons

- Not a 100% solution
- Bean per class
- Impossible to inject straight values
- Don't scan a huge number of classes!
 - use Spring's filtering mechanism

JMS Support

- New JMS namespace

```
<jms:listener-container  
  connection-factory="myConnectionFactory"  
  transaction-manager="myTransactionManager">  
    <jms:listener destination="myQueue" ref="myListener"/>  
    <jms:listener destination="myQueue2" ref="myListener2"/>  
</jms:listener-container>
```

- JMS Namespace support for JCA

```
<jms:jca-listener-container  
  resource-adapter="myResourceAdapter"  
  transaction-manager="myTransactionManager">  
    <jms:listener destination="myQueue" ref="myListener"/>  
    <jms:listener destination="myQueue2" ref="myListener2"/>  
</jms:jca-listener-container>
```

Enhanced AspectJ support

- New bean(name) pointcut element

```
<aop:config>
    <aop:pointcut id="valManDaoMethods" expression="bean(*Validator) or
    bean(*Manager) or bean(*DAO)"/>
    <aop:advisor advice-ref="timeMeasureinterceptor" order="1" pointcut-
    ref="valManDaoMethods" />
</aop:config>
```

- Support AspectJ load-time weaving through Spring's **LoadTimeWeaver**
- For any supported platform
 - Generic Spring VM agent
 - Various app servers: Tomcat, Glassfish, OC4J

@Configurable revisited

- Now supported with loadtime weaving
 - No AspectJ compiler needed
 - Good combination with annotation-driven config (i.e. @Autowired)

```
@Configurable
public class Person {
    @Autowired
    private MyService myService;
}
```

```
// The following domain object will be configured by Spring
Person obj = new Person ();
```

Annotation-driven MVC Controllers

- Java5 variant of MultiActionController
- POJO-based
- Several annotations:
 - @Controller
 - @RequestMapping
 - @RequestMethod
 - @RequestParam
 - @ModelAttribute
 - @SessionAttributes
 - @InitBinder

Quick MVC example

- `@Controller`
- `@RequestMapping("/order/*")`
- `public class OrderController {`
- `@Autowired`
- `private OrderService orderService;`
- `@RequestMapping("/print.*")`
- `public void printOrder(HttpServletRequest request,`
- `OutputStream responseOutputStream) {`
- `...`
- `// write directly to the OutputStream:`
- `orderService.generatePdf(responseOutputStream);`
- `}`
- `@RequestMapping("/display.*")`
- `public String displayOrder(`
- `@RequestParam("id") int orderId, Model model) {`
- `model.addAttribute(...);`
- `return "displayOrder";`
- `}`
- `}`

TestContextFramework

- Revised, annotation-based test framework
- Supports JUnit 4.4, TestNG as well as JUnit 3.8
- Consistent support for Spring's core annotations
- Convention over configuration

Spring-specific integration testing functions

- Context management & caching
- Dependency injection of tests
- Transactional test management

Test Context Key Abstractions

- TestContext
- TestContextManager
- TestExecutionListener
 - DependencyInjectionTestExecutionListener
 - DirtiesContextTestExecutionListener
 - TransactionalTestExecutionListener

Test example

```
@RunWith(SpringJUnit4ClassRunner.class)
@Transactional(defaultRollback = true)
@ContextConfiguration(locations = { "/util-dao-context.xml", "/tomcat-util-dao-
context.xml" })
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public class OrdersTypeDAOTest {
    @Autowired
    private SessionFactory sessionFactory;
    @Autowired
    private IOrdersTypeDAO ordersTypeDAO;

    @Test
    public void testFindOrdersTypesByOrderKind() {
        List<OrdersType> orderTypes =
        ordersTypeDAO.findOrdersTypesWithGroupAndKindByOrderKind(DolOrder
sKindTypeEnum.FINANCIAL);
    • ...
    • }
```

Test Context Configuration

- TestExecutionListeners
 - @TestExecutionListeners
- Application Contexts
 - @ContextConfiguration and @Context
- Dependency Injection
 - @Autowired, @Qualifier, @Resource, @Required, etc.
- Transactions
 - @Transactional, @NotTransactional,
 - @TransactionConfiguration, @Rollback,
 - @BeforeTransaction, and @AfterTransaction
- Testing Profiles (JUnit only)
 - @IfProfileValue and @ProfileValueSourceConfiguration
- JUnit extensions
 - @ExpectedException, @Timed, @Repeat

Spring Portfolio

- Seperate projects
- Motivation
 - Complexities
 - No shared common components between different styles of applications
 - Commonalities

Spring Portfolio projects

- Spring Integration
- Spring Web Flow
- Spring Security (formerly Acegi)
- Spring Web Services
- Spring Dynamic Modules for the OSGi Service Platform
- Spring Batch
- Pitchfork
- AspectJ
- Spring IDE
- Spring .NET
- Spring LDAP
- Spring Rich Client

Spring Batch

- Version 1.0
- SpringSource and Accenture cooperation
- Batch applications are very different than other applications styles.
- Provides:
 - Common I/O, DAO patterns
 - Transaction chunking
 - Retrying
 - Exception transaction
 - Partitioning
 - Pararelization

Spring DM

- OSGI platform
 - Modularity
 - Versioning
 - Life cycle of bundles
- Version 1.0

```
<bean id="myPojo" class="someClass"/>  
<osgi:service ref="myPojo"  
interface="com.springsource.sdm.demo.Painter"/>
```

```
<osgi:reference id="osgiService"  
interface="com.springsource.sdm.demo.Painter"/>  
<bean id="consumer" p:service-ref="osgiService" class=.../  
>
```

Spring Web Services

- Version 1.5.3
- Contract-first, document-driven
- Some features:
 - Supports WS-Security
 - XML API support
 - Flexible XML Marshalling
 - Additional transports: JMS and Email

Spring Security

- Formerly the Acegi Security System for Spring
- Version 2.0
- Addresses authentication and authorization

Summary

- Not another slide

Job trends

