# SPRING

1. Overview of Spring – ¼ Hours
2. Over view of Spring Modules ¼ Hours
3. IOC – 1 Hour
4. Resources – ½ Hour
5. Validation, Data Binding, type Conversion – ½ Hour
6. Spring EPL – ½ Hour
7. AOP – 1 Hour
8. Data Access – ½ Hour
9. Web MVC – 1 Hour
10. Remoting – ½ Hour  Total – 6 Hours

# Overview of Spring Framework

- The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications.

- Spring handles the infrastructure so you can focus on your application.

- Spring enables you to build applications from "plain old Java objects" (POJOs)

# Overview of Spring Framework

- As an application developer, can use the Spring platform advantage:

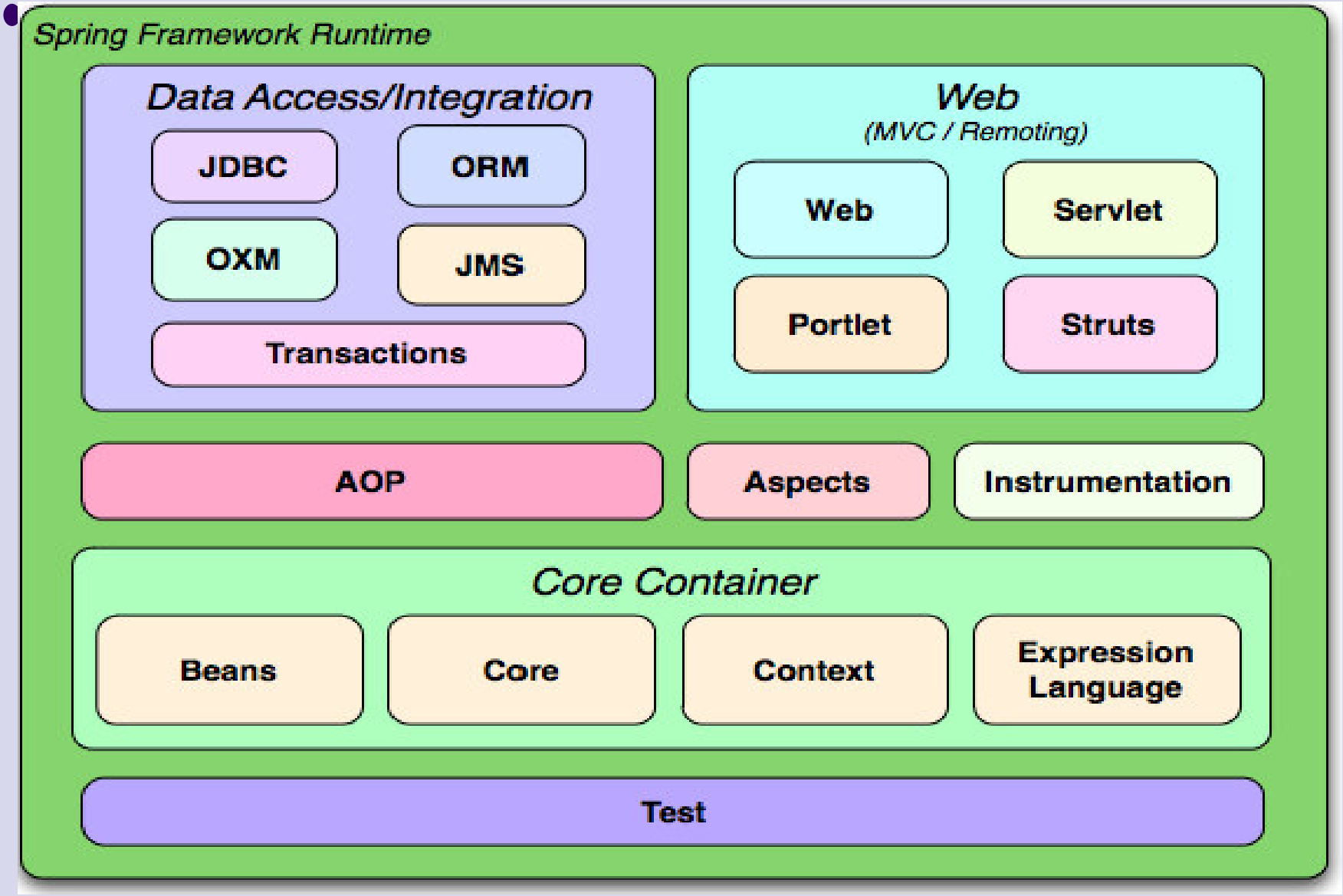
• Make a Java method execute in a database transaction without having to deal with transaction APIs.

• Make a local Java method a management operation without having to deal with JMX APIs.

• Make a local Java method a message handler without having to deal with JMS APIs.

Make a local Java method a remote procedure without having to deal with remote APIs.

# Overview of Spring Modules



# SPRING 3 New Features

**Overview of new features**

This is a list of new features for Spring 3.0. We will cover these features in more detail later in this

section.

- Spring Expression Language
- IoC enhancements/Java based bean metadata
- General-purpose type conversion system and field formatting system
- Object to XML mapping functionality (OXM) moved from Spring Web Services project
- Comprehensive REST support
- @MVC additions
- Declarative model validation
- Early support for Java EE 6
- Embedded database support

# Core Technologies

- *The IoC container*
- *Resources*
- *Validation, Data Binding, and Type Conversion*
- *Spring Expression Language (SpEL)*
- *Aspect Oriented Programming with Spring*
- *Spring AOP APIs*
- *Testing*

# IOC Container

- The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container.

- **BeanFactory** provides the configuration framework and basic functionality, and the **ApplicationContext** adds more enterprise-specific functionality.

# IOC Container

**1. BeanFactory**

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

**2. ApplicationContext**

```
ApplicationContext context=
        new classPathXmlApplicationContext("myspring.xml");
                or new ClassPathXmlApplicationContext(new String[]
{"services.xml", "daos.xml"});
        new FileSystemXmlApplicationContext("myspring.xml");
                or
        new XmlWebApplicationContext("myspring.xml");
```

| | |
|---|---|
| Bean Factory the heart of spring based applications | This module extends the support of BeanFactory and makes spring a framework. |
| Implementation of Factory Pattern that applies to IOC. | Add Support for I18N, application life cycle events and validation |
| | Supplies many enterprise services such as JNDI access, EJB integration, remoting and scheduling. |

# Sample Configuration XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
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-3.0.xsd">
<!-- services -->
<bean id="petStore"
class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
<property name="accountDao" ref="accountDao"/>
<property name="itemDao" ref="itemDao"/>
<!-- additional collaborators and configuration for this bean go here -->
</bean>
<!-- more bean definitions for services go here -->
</beans>
```

# Managing Multiple XML Files

**Composing XML-based configuration metadata**

It can be useful to have bean definitions span multiple XML files.

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

# Wiring Beans

1. **Prototyping Vs Singleton – default is singleton**

```
<bean id="bean2" class="shoepack.LakhaniShoeFactory"
        singleton="false"/>
```

**2. Initialization and destruction**

```
<bean id="bean2" class="shoepack.LakhaniShoeFactory"
        init-method="setup" destroy-method="teardown"/>
```

**3. Injecting dependencies via setter methods**

```
<beans>
<bean id="bean1" class="shoepack.ShoeShop">
<property name="name"><value>Haaris</value></property></bean>
</beans>
```

**4.Referencing Other Beans**

```
<beans>
<bean id="bean1" class="shoepack.ShoeShop">
        <property name="factory"><ref local="bean2"/></property>
</bean>
<bean id="bean2" class="shoepack.LakhaniShoeFactory"></bean>
</beans>
```

# Wiring Beans

## Wiring Collections

Collections supported by Spring's wiring

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

**Wiring lists and arrays**

```
<property name="nameList">
        <list>
                <value>bar1</value>
                <ref bean="bar2"/>
        </list>
</property>
```

**Wiring sets**

```
<property name="nameList">
        <set>
                <value>bar1</value>
                <ref bean="bar2"/>
        </set>
</property>
```

# Wiring Beans

**Wiring maps**

```
<property name="nameList">
    <map>
        <entry key="key1">
            <value>bar1</value>
        </entry>
        <entry key="key2">
            <ref bean="bar2"/>
        </entry>
    </map>
</property>
```

**Wiring properties**

```
<property name="nameList">
    <props>
<prop key="key1">bar1</prop>
<prop key="key2"><null/></prop> -- setting null values
    </props>
</property>
```

Haaris Infotech

Driven by Technology

# Wiring Beans

**Injecting dependancies via Constructor**

```
<beans>
        <bean id="bean1" class="shoepack.ShoeShop">
            <constructor-arg>
<value>44</value> or <ref bean="bar"/>
            </constructor-arg>
        </bean>
</beans>
```

**Handling ambiguous constructor arguments**

```
<beans>
  <bean id="bean1"
class="shoepack.ShoeShop">
    <constructor-arg index="1">
      <value>44</value>
    </constructor-arg>
    <constructor-arg index="0">
      <value>44</value>
    </constructor-arg>
   </bean>
</beans>
```

OR

```
<beans>
 <bean id="bean1" class="shoepack.ShoeShop">
        <constructor-arg type="java.lang.String>
            <value>44</value>
        </constructor-arg>
        <constructor-arg type="java.lang.Integer">
            <value>44</value>
        </constructor-arg>
  </bean>
</beans>
```

# Wiring Beans

**Autowiring**

```
<beans>
        <bean id="bean1" class="shoepack.ShoeShop"
autowire="byName"/>
</beans>
or
<beans>
        <bean id="bean1" class="shoepack.ShoeShop"
autowire="constructor"/>
</beans>
or
<beans>
        <bean id="bean1" class="shoepack.ShoeShop"
autowire="autodetect"/>
</beans>
```

By setting autowire to autodetect, you instruct the Spring container to attempt to autowire by constructor first. If it can't find a suitable match between constructor arguments and beans, it will then try to autowire using byType.

# Wiring Beans

**Mixing Auto and explicit wiring**

```
<beans>
        <bean id="bean1" class="shoepack.ShoeShop"
autowire="byName">
            <property name="courseDAO">
                <ref bean="some bean"/>
            </property>
        </bean>
</beans>
```

**Autowiring by default**

```
<beans default-autowire="byName" >
```

# Wiring Beans-PostProcessingBean

**Working with Spring's Special Beans**

The postProcessingBeforeInitialization() method is called immediately prior to bean initialization (the call to afterPropertiesSet() and the bean's custom init-method). Likewise, the postProcessAfterInitialization() method is called immediately after initialization.

```
public class MyPostProcessor implements BeanPostProcessor
{
public Object postProcessAfterInitialization(Object bean,String name)throws
BeansException
{
Field[] fields=bean.getClass().getDeclaredFields();
        try{
for(int i=0;i<fields.length;i++)        {
        if(fields[].getType().equals(java.lang.String.class))  {
        fields[i].setAccessible(true);
        String original=(String)fields[i].get(bean);
        fields[i].set(bean,fuddify(original));  }
        }}catch(Exception e){}   return bean;                    }

public String fuddify(String orig) {
if(orig == null) return orig; return orig.replaceAll("(r|l)","w").replaceAll(" (R|L)","w"); }
public Object postProcessBeforeInitialization(Object bean,String name)throws Exception    {
                return bean;                    }
}
```

# Wiring Beans-PostProcessingBean

**Registering bean post processors**

If your application is running within a bean factory, you'll need to programmatically register each BeanPostProcessor using the factory's addBeanPostProcessor() method.

```
BeanPostProcessor fuddifier=new Fuddifier();
factory.addBeanPostProcessor(fuddifier);
```

If you're using an application context, you'll only need to register the post processor as a bean within the context.

```
<bean id="fuddifier" class="com.Fuddifier"/>
```

The container will recognize the fuddifier bean as a BeanPostProcessor and call its postprocessing methods before and after each bean is initialized.

As a result of the fuddifier bean, all string properties of all beans will be fuddified. For example, you had the following bean defined in XML.

```
<bean id="bugs" class="aaaa">
        <property name="des">
                <value>That is really a rabbit</value>
        </property>
</bean>
```

When the fuddifier processor is finished, the description property will hold "The weawwy wabbit.

# Wiring Beans-PropertyPlaceHolder

The properties file

database.url=jdbc:hsqlib:Training
database.driver=org.hsqldb.jdbcDriver

To enable reading the above properties file, configure the following bean in your bean wiring file.

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
">
        <property name="locations">
                <list>
                        <value>jdbc.properties</value>
                        <value>security.properties</value>
                </list>
        </property>
        or
        <property name="location">
                        <value>jdbc.properties</value>
                </property>
        </bean>
```

# Wiring – Factory 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;
}
}
```

# Method injection

A solution is to forego some inversion of control.

(a typically new) bean B instance every time bean A needs it.

```
protected Command createCommand() {
// notice the Spring API dependency!
return this.applicationContext.getBean("command",
Command.class);
}
```

Even if the command is declared as singleton, still a new instance will be created every time you call this.

# Spring Annotations

@Scope

- Indicates the scope to use for annotated class instances
  - Default == "singleton"
    - Options:
      - Singleton
      - Prototype
  - Web Options:
    - Request
    - Session
    - Global session

@Autowired

– Marks a constructor, field, setter or config method for injection.

– Fields are injected

• After construction

• Before config methods

– @Autowired(required=false)

– Config:

• AutowiredAnnotationBeanPostProcessor

# Annotation

- Constructor

```
public class AccountService {

    private AccountDAO dao;

     @Autowired
    public AccountService(AccountDAO dao) {
        this.dao = dao;
    }
```

- Setter

```
public class AccountService {

    private AccountDAO dao;

     @Autowired
    public void setDao(AccountDAO dao) {
        this.dao = dao;
    }
```

- Field

```
public class AccountService {

    @Autowired
    private AccountDAO dao;
```

# SPRING-Annotation

```
@Autowired
@Qualifier("myDSName")
Private DataSource dataSource
```

- Or

```
@Autowired
public void init(@Qualifier("srcName") DataSource
                 dataSource)
{...}
```

# SPRING-Annotation

- JSR-250 javax.annotations
  - Requires
    - CommonAnnotationBeanPostProcessor bean or
    - <context:annotation-config />
  - @Resource
    - Injects a named resource
      - Spring name not JNDI
        - » Unless configured :) to use JNDI
  - @PostConstruct
    - Method invocated after construction
    - No XML
    - Multiple methods possible
  - @PreDestroy
    - Method invocated when application context hosting object is closed

# Composing Java-based configurations

Using the @Import annotation

Much as the <import/> element is used within Spring XML files to aid in modularizing configurations,

the @Import annotation allows for loading @Bean definitions from another configuration class:

```
@Configuration
public class ConfigA {
public @Bean A a() { return new A(); }
}
@Configuration
@Import(ConfigA.class)
public class ConfigB {
public @Bean B b() { return new B(); }
}
```

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating

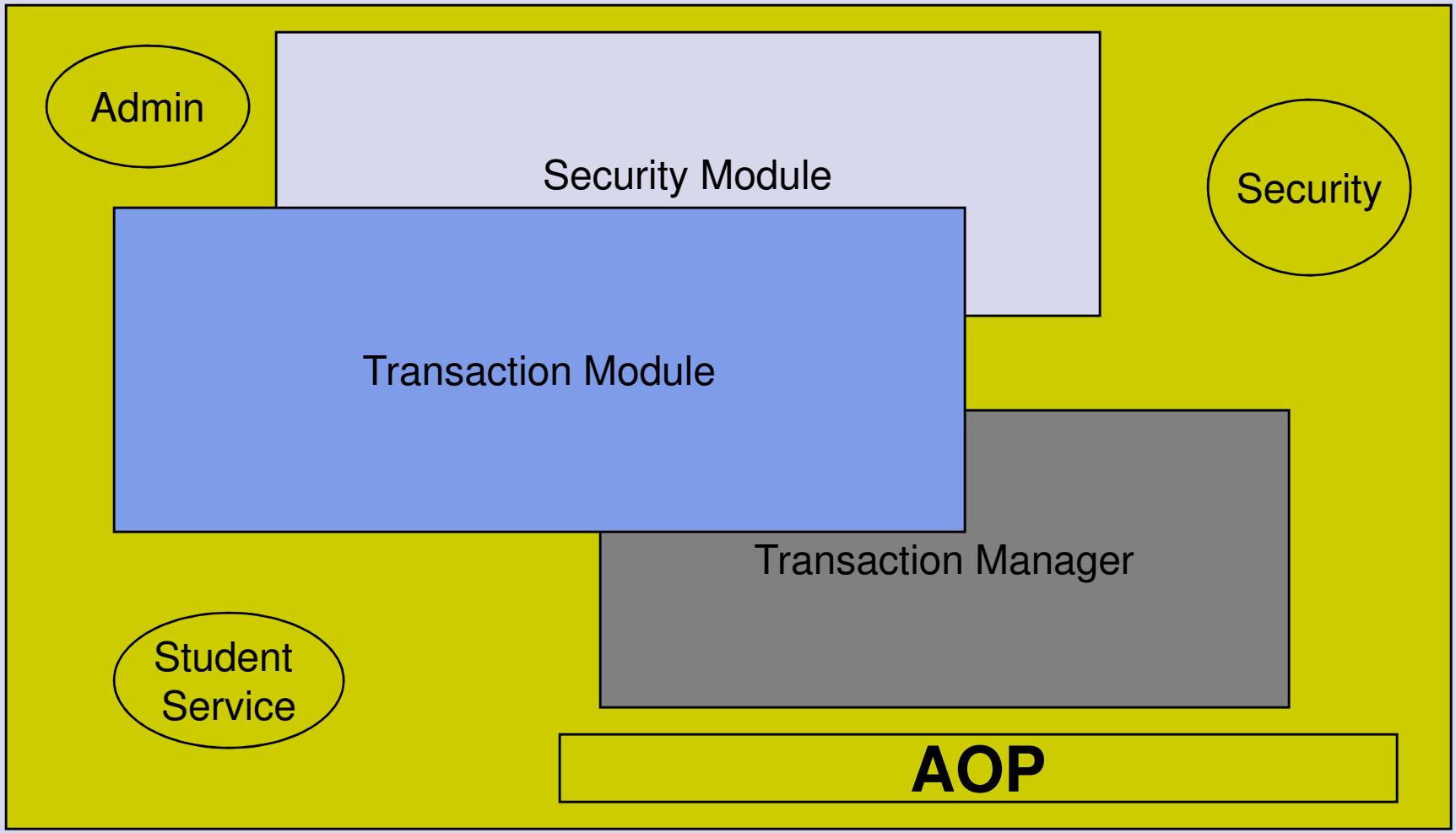
the context, only ConfigB needs to be supplied explicitly:

```
public static void main(String[] args) {
ApplicationContext ctx = new
AnnotationConfigApplicationContext(ConfigB.class);
// now both beans A and B will be available...
A a = ctx.getBean(A.class);
B b = ctx.getBean(B.class);
}
```

This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring the developer to remember a potentially large number of @Configuration classes during construction.

# Aspect-Oriented Programming

While Inversion of control makes it possible to tie software components together loosely, Aspect Oriented programming enables you to capture functionality that is used throughout your application in reusable components.



# Aspect-Oriented Programming

Creating Aspects

1. Creating Advice

a. Around - org.aopalliance.intercept.MethodInterceptor - Intercepts calls to the target method

b. Before - org.springframework.aop.MethodBeforeAdvice - Called before the target method is invoked

c. After - org.springframework.aop.AfterReturningAdvice - Called after the target method returns

d. Throws - org.springframework.aop.ThrowsAdvice - Called when a method throws an exception

Refer Lab2 for implementations and examples on the above.

# Aspect-Oriented Programming

```
public class WelcomeAdvice implements
MethodBeforeAdvice {
                public void before(Method method,Object[]
args, Object target)
                {
                        Customer customer=(Customer)args[0];
- Calls first argument to Customer
                        System.out.println("Welcome Mr.
:"+customer.getName());
                }
        }
```

# Aspect-Oriented Programming

```
<beans>
        <bean id="ShoeShopTarget" class="com.RamuShoeShop"/>
        <bean id="welcomeAdvice" class="com.WelcomeAdvice"/>
        <bean id="ShoeShop"
class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <value>com.ShoeShop</value>
                </property>
                <property name="interceptorNames">
                        <value>welcomeAdvice</value>
                </property>
                <property>
                        <list>
                                <value>welcomeAdvice</value>
                        </list>
                </property>
                <property name="target">
                        <ref bean="ShoeShopTarget"/>
                </property>
        </bean>
</beans>
```

# Aspect-Oriented Programming

```
public class ThankYouAdvice implements AfterReturningAdvice {
        public void afterReturning(Object returnValue, Method
method, Object[] args, Object target)
        {                       }
}

public class MyInterceptor implements MethodInterceptor
{
        private Set customers=new HashSet();
public Object invoke(MethodInvocation invocation)throws Throwable
        {
Customer customer=(Customer)invocation.getArguments()[0];
if(customer.contains(customer)
        {
        throw new ShoeException("one per customer");
        }
                Object shoe =invocation.proceed();
                customers.add(customer);
                return shoe;
        }
}
```

# Aspect-Oriented Programming

```
public class ShoeExceptionAdvice implements ThrowsAdvice
{
        public void afterThrowing(ShoeException se){    ------logic ------}
        public void afterThrowing(Method m,Object[] args,Object
target,Throwable thro){}
}
```

**Defining Pointcuts**

Pointcuts determine if a particular method on a particular class matches a particular criterion. If the method is indeed a match then advice will be applied to this method. Spring's pointcuts allow us to define where our advice is woven into our classes in a very flexible manner.

# Aspect-Oriented Programming

**Defining a pointcut**

```
        <bean id="ShoeShopTarget" class="com.RamuShoeShop"/>
        <bean id="welcomeAdvice" class="com.WelcomeAdvice"/>
<bean id=”pointcutadvisor”
class=”org.springframework.aop.support.NameMatchMethodPointcutAdvisor”
>
        <property name=”mappedName”><value>order*</value></property>
        <property name=”advice”><ref bean=”welcomeAdvice”/></property>
</bean>
<bean id="ShoeShop“
class="org.springframework.aop.framework.ProxyFactoryBean">
<property name="proxyInterfaces"><value>com.ShoeShop</value></property>
<property
name="interceptorNames"><value>pointcutadvisor</value></property>
<property name="target"><ref bean="ShoeShopTarget"/></property>
</bean>
</beans>
```

Instead of supplying wildcard characters, we just as easily explicity name each of the methods.

```
        <property name=”mappedNames”>
                <list>  <value>order</value><value>order2</value></list>
        </property>
```

# Building Web Applications with Spring 3.0

Haaris Infotech

Driven by Technology

# Agenda

- Spring MVC Overview
- Spring MVC Infrastructure
- Developing Controllers
- RESTful Web Services
- Demo

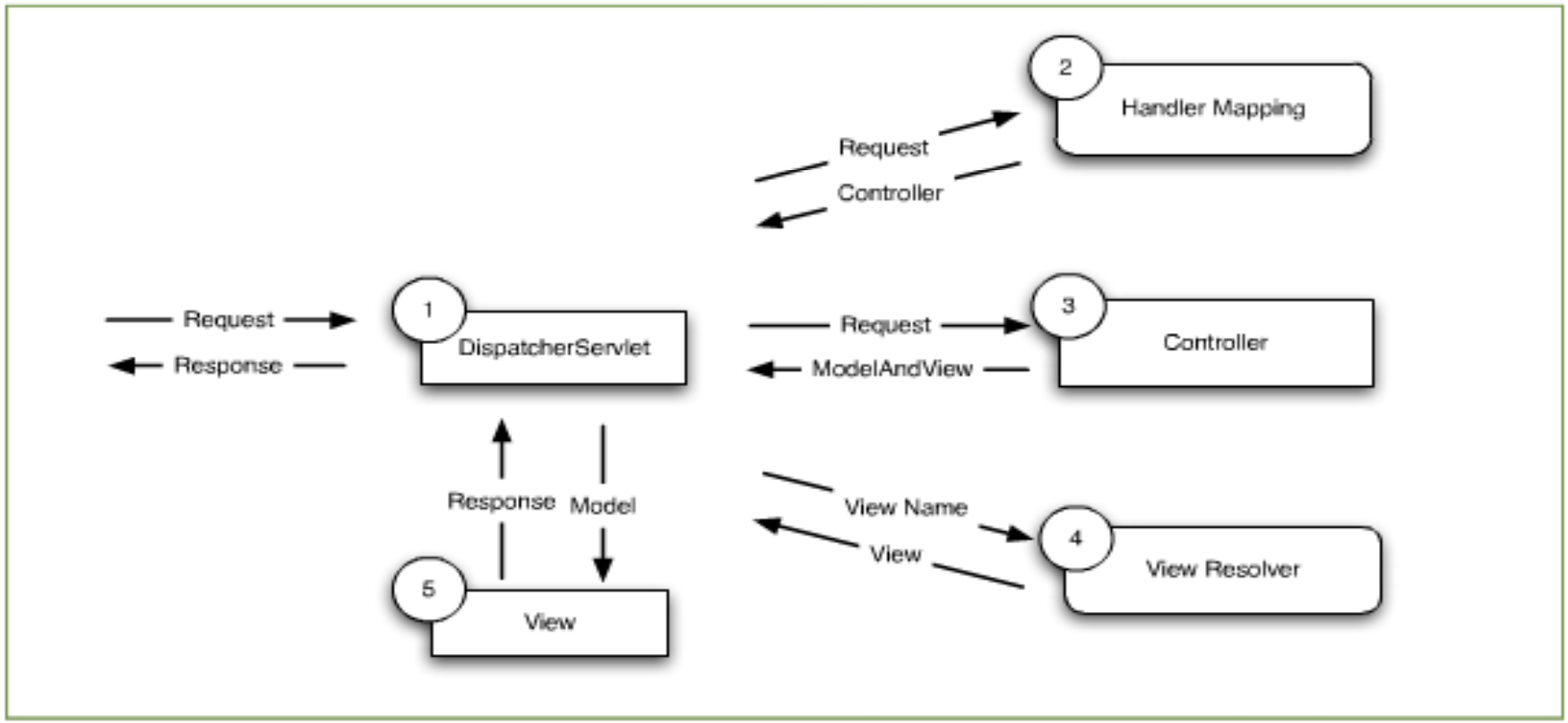
# What is MVC?

- Model
  - Domain-specific data and processing logic
  - Entities, services, repositories, etc.
- View
  - User-facing presentation
  - JSPs, Velocity/Freemarker Templates, XSLT, etc.
- Controller
  - Mediator between model and view
  - Responsible for routing requests and responses

# What is Spring MVC?

- Core web component of the Spring Framework
  - Foundation of Spring's web strategy
- Similar to Struts but with improvements:
  - Better separation of concerns
  - Robust data binding and validation
  - Customizable type coercion with property editors
- Supports wide variety of view technologies:
  - JSP/JSTL, XML/XSLT, Freemarker, Velocity, PDF, etc.
- Significantly enhanced in 2.5 release and further refined in 3.0

# Request/Response Handling



# DispatcherServlet

- *Front Controller* implementation in Spring MVC
  - Handles incoming request and dispatches to appropriate handler
- Coordinates communication between infrastructural components
- Wired as standard servlet in web.xml
- Upon initialization looks for *servletname*-`servlet.xml`
- Default values defined in `DispatcherServlet.properties`
  - Overriding defaults replaces default values!

# Configuring web.xml

## Servlet Configuration

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

## Servlet Mapping

```
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/v3/*</url-pattern>
</servlet-mapping>
```

DispatcherServlet finds context configuration based on name

# HandlerMapping

- Defines interface to select appropriate handler (Controller)
- DispatcherServlet consults HandlerMapping collection to locate appropriate controller
- Interceptors can be applied to customize pre/post processing
- Common implementations:
  - `BeanNameUrlHandlerMapping`: Maps requests based on bean names
  - `SimpleUrlHandlerMapping`: Maps request URLs to Controller beans
  - `ControllerClassNameHandlerMapping`: Auto generates mappings by class name
  - `DefaultAnnotationHandlerMapping`: Maps request based on annotations

# Controller

- Controller defines the interface to handle and process requests.
  - Provides access to the `HttpServletRequest` and `HttpServletResponse`
- Returns `ModelAndView`
  - Composite data holder for the data model and a view
    - Model is typically a Map variant
    - View can be logical view name or actual `View` instance

```
public ModelAndView handleRequest(HttpServletRequest request,
                                  HttpServletResponse response) throws Exception {

    // Forward request to "artists list" view
    return new ModelAndView("artists");
}
```

# ViewResolver

- ViewResolver defines the interface to select an appropriate View for the response
- ViewResolver instances can be chained together into an ordered collection
- Common implementations include:
  - InternalResourceViewResolver
  - ResourceBundleViewResolver
  - XmlViewResolver

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

# View

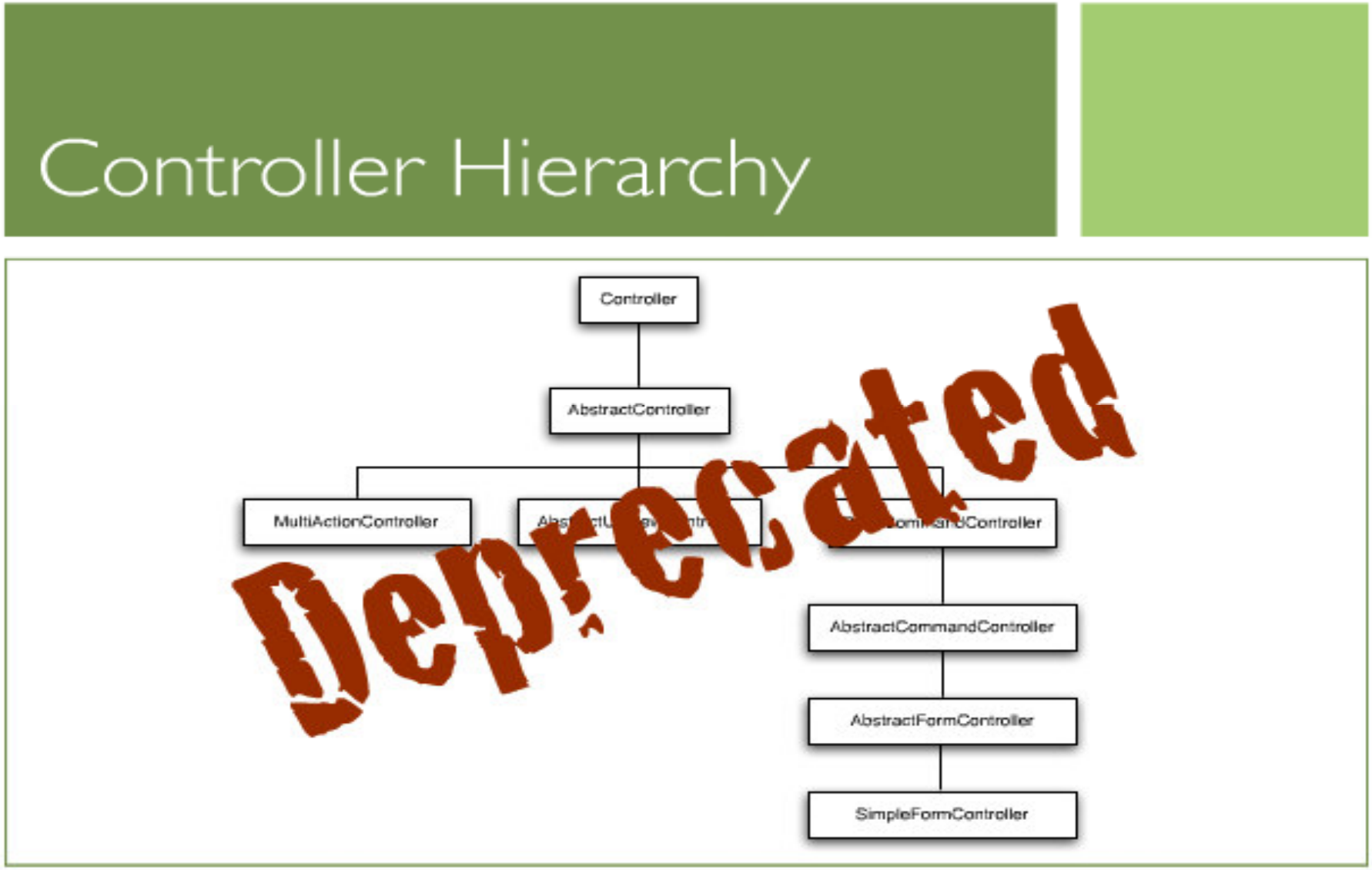
- Defines the interface for objects rendering a client response
- Iterates through the values in the model to generate response
- Framework provides a wide variety of View choices:
  - Common Web View:
    - JstlView, FreemarkerView, VelocityView, XsltView
  - Special Format View:
    - AbsractExcelView, AbstractPdfView, JasperReportsView
  - New Views in Spring 3.0: New in 3.0
    - MarshallingView, AbstractAtomFeedView, AbstractRssFeedView

# spring-servlet.xml

```
<beans>
  <bean id="handlerMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/v3/genres.html">genreController</prop>
        <prop key="/v3/artists.html">artistController</prop>
      </props>
    </property>
  </bean>

  <bean id="artistController" class="com.mccuneos.spring.web.controller.ArtistController"/>

  <bean id="genreController" class="com.mccuneos.spring.web.controller.GenreController"/>

  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
  </bean>
<beans>
```

# Controller

- Controllers historically built on Controller interface
- Specialized implementations included:
  - Simple Form Controllers
    - Form processing lifecycle
    - Databinding and validation
  - Wizard Form Controllers
    - Multi-page navigation
  - Multi-action Controllers
    - Grouped related, stateless operations


Controller Hierarchy
Controller
AbstractController
MultiActionController
AbstractCommandController
AbstractFormController
SimpleFormController
Deprecated

# Controller to @Controller

- Designates a class as a Spring MVC `Controller`
- Removes the need to implement or extend Spring-specific classes
- @Controller stereotype automatically be picked up if using component scanning

```
<context:component-scan base-package="com.company.spring.web" />
```

- Flexible method signatures
  - No need to implement/override *lifecycle* methods
- Can eliminate dependencies on Servlet API
  - No need to "mock out" dependencies in unit tests

# Flexible Method Arguments

- Method signatures are very flexible
- Arguments can be:
  - Servlet types: HttpServletRequest, HttpServletResponse, HttpSession
  - WebRequest: wrapper over Servlet specifics
  - I/O types: InputStream, OutputStream, Reader, Writer
  - Model types: Model, ModelMap, Map
  - Model attributes, Request parameters
  - Locale

# Flexible Return Types

- ModelAndView: Wraps model and View or logical view name
- Model or Map: Return model with view implicitly defined
  - Will use RequestToViewNameTranslator
- View: View instance to be used to render response
- String: Interpreted as logical view name
- void: Can return void if Controller handles response
- Any object: Will be used as single model value

# @RequestMapping

- Used to map URL patterns to controllers
- Can be applied at either a class and/or method level
  - Multi-action controllers commonly defined at method level
  - Form controllers applied at class level with method level annotations used to narrow request to particular type
- Greater mapping flexibility than previous releases

# @RequestParam

- Used to bind request parameters to method parameters
- Limits or removes needs to rely on Servlet API
- @RequestParam designates a required parameter
  - Can be made optional with required=false

```
@Controller
@RequestMapping("/edit.do")
public class EditController {

    @RequestMapping(method = RequestMethod.GET)
    public String configure(@RequestParam("id") int id, ModelMap model) {
        model.put(service.getContact(id));
        return "edit";
    }
}
```

# @ModelAttribute/ @SessionAttribute

- Used to map model data into controller
- Used at a method level to maps model attribute to method argument
  - No need to work directly with `HttpServletRequest`
- Can also be used at method level to provide reference data
- `@ModelAttribute` is processed before `@RequestMappings` so data is pre-populated before controller processing
- Can use `@SessionAttribute` to store and retrieve session data


REST Support

# What is REST?

- *RE*presentational *S*tate *T*ransfer
- Lightweight, resource-oriented architectural style
  - Unlike SOAP or XML-RPC, is not a standard
- Stateless, cacheable, scalable communication protocol
- Uses standard HTTP methods to read and write data
- Provides uniform interface for interacting with resources
  - Nouns used to represent resources: artists, albums, etc.
  - Verbs defined through standard HTTP methods

# HTTP Methods

- Resource interaction through standard HTTP methods
  - GET: Gets a representation of resource. Safe operation.
  - POST: Creates or updates resource.
  - PUT: Creates or updates resource. Idempotent.
  - DELETE: Deletes a resource. Idempotent.
  - HEAD: GET request without body. Returns headers only.
  - OPTIONS: Discovery method to determine *allows*.

# Spring 3.0 REST Support

- Builds on Spring 2.5's @Controller Model
- Focuses on making it simple to expose and consume RESTful Web Services
  - Client-side access greatly simplified via RestTemplate
  - Server-side development enhanced with expanded request mappings and path variables
- Provides competing approach to JAX-RS implementations:
  - Jersey
  - RESTEasy
  - Restlet

# URI Templates

- RESTful services define resource locations through URIs
  - http://www.host.com/orders/{orderId}
- Variable expansion would convert URI to:
  - http://www.host.com/orders/8675309
- Spring MVC implements URI Templates through its standard `@RequestMapping` annotation
- `@PathVariable` annotation can extract template values from template variables

# Mapping Requests

- Uses standard Spring MVC @RequestMapping annotation
  - Supports URI templates in path expressions

```
@Controller
@RequestMapping("/artists/{artistId}")
public class ArtistController {

    @RequestMapping(method = RequestMethod.POST)
    public String create(Artist artist) { return "artist"; }

    @RequestMapping(method = RequestMethod.DELETE)
    public String delete(Long id) { return "artist"; }
}
```

# @PathVariable

- @PathVariable maps URI template placeholders to argument values
  - Can use multiple @PathVariable annotations
  - Values can be explicitly defined or inferred through debug info

```
@Controller
public class AlbumController {

    @RequestMapping(value = "/artists/{artistId}/albums/{albumId}"
                    method = RequestMethod.GET)
    public Album getAlbum(@PathVariable("artistId") Long artistId,
                          @PathVariable("albumId") Long albumId) {

        return service.findAlbum(artistId, albumId);
    }
}
```

# Remoting

Spring features integration classes for remoting support using various technologies.

The remoting support eases the development of remote-enabled services, implemented by your usual (Spring) POJOs.

# Accessing Webservices

```
<bean id="shop" class="shoepack.RamuShoeShop">
        <property name="factory“><ref bean="lsf"/></property>
        <property name="hello“><ref bean="HelloService"/></property>
</bean>

<bean id="HelloService"
class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
        <property name="serviceInterface" value="service1.HelloServInter"/>
        <property name="wsdlDocumentUrl"
        value="http://localhost:8080/axis/services/hello1?wsdl"/>
        <property name="namespaceUri"
value="http://localhost:8080/axis/services/hello1"/>
        <property name="serviceName" value="HelloServiceService"/>
        <property name="portName" value="hello1"/>
</bean>
```

# Accessing EJB's

```
<bean id="myComponent"
class="org.springframework.ejb.access.LocalStatelessSession
ProxyFactoryBean">
<property name="jndiName" value="ejb/myBean"/>
<property name="businessInterface"
value="com.mycom.MyComponent"/>
</bean>
<bean id="myController"
class="com.mycom.myController">
<property name="myComponent" ref="myComponent"/>
</bean>
```

```
<jee:local-slsb id="myComponent" jndi-
name="ejb/myBean"
business-interface="com.mycom.MyComponent"/>
<bean id="myController"
class="com.mycom.myController">
<property name="myComponent" ref="myComponent"/>
</bean>
```

# DAO

Configuration

```
<beans…….
<bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
<property name="dataSource" ref="dataSource"/>
</bean>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
<property name="driverClassName" value="${jdbc.driverClassName}"/>
<property name="url" value="${jdbc.url}"/>
<property name="username" value="${jdbc.username}"/>
<property name="password" value="${jdbc.password}"/>
</bean>
<context:property-placeholder location="jdbc.properties"/>
</beans>
```

## Choosing an approach for JDBC database access

### JdbcTemplate

```
public class JdbcCorporateEventDao {
private JdbcTemplate jdbcTemplate;
public void setDataSource(DataSource dataSource) {
this.jdbcTemplate = new JdbcTemplate(dataSource);
}
// JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

**Examples of JdbcTemplate class usage**

```
int rowCount = this.jdbcTemplate.queryForInt("select count(*) from
    t_actor");

int countOfActorsNamedJoe = this.jdbcTemplate.queryForInt(
"select count(*) from t_actor where first_name = ?", "Joe");

String lastName = this.jdbcTemplate.queryForObject(
"select last_name from t_actor where id = ?",
new Object[]{1212L}, String.class);
```

**Querying and populating a *single domain object:***

```
Actor actor = this.jdbcTemplate.queryForObject(
"select first_name, last_name from t_actor where id = ?",
new Object[]{1212L},
new RowMapper<Actor>() {
public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
Actor actor = new Actor();
actor.setFirstName(rs.getString("first_name"));
actor.setLastName(rs.getString("last_name"));
return actor;
}
});
```

Querying and populating a number of domain objects:

```
List<Actor> actors = this.jdbcTemplate.query(
"select first_name, last_name from t_actor",
new RowMapper<Actor>() {
public Actor mapRow(ResultSet rs, int rowNum) throws
SQLException {
Actor actor = new Actor();
actor.setFirstName(rs.getString("first_name"));
actor.setLastName(rs.getString("last_name"));
return actor;
}
});
```

## An Alternate

```
public List<Actor> findAllActors() {
return this.jdbcTemplate.query( "select first_name, last_name from t_actor",
new ActorMapper());
}
private static final class ActorMapper implements RowMapper<Actor> {
public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
Actor actor = new Actor();
actor.setFirstName(rs.getString("first_name"));
actor.setLastName(rs.getString("last_name"));
return actor;
}
}
```

**Updating (INSERT/UPDATE/DELETE) with jdbcTemplate**

You use the update(..) method to perform insert, update and delete operations. Parameter values are

usually provided as var args or alternatively as an object array.

```
this.jdbcTemplate.update(
"insert into t_actor (first_name, last_name) values (?, ?)",
"Leonor", "Watling");
this.jdbcTemplate.update(
"update t_actor set = ? where id = ?",
"Banjo", 5276L);
this.jdbcTemplate.update(
"delete from actor where id = ?",
Long.valueOf(actorId));
```

**Other jdbcTemplate operations**

You can use the execute(..) method to execute any arbitrary SQL, and as such the method is often used for DDL statements. It is heavily overloaded with variants taking callback interfaces, binding variable arrays, and so on.

**this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");**

The following example invokes a simple stored procedure.

```
this.jdbcTemplate.update(
"call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
Long.valueOf(unionId));
```

# NamedParameterJdbcTemplate

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
public void setDataSource(DataSource dataSource) {
this.namedParameterJdbcTemplate = new
NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {
String sql = "select count(*) from T_ACTOR where first_name = :first_name";
SqlParameterSource namedParameters = new
MapSqlParameterSource("first_name", firstName);
return namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```