Introduction to the Spring Framework

Eduardo Issao Ito
<ziba@summa-tech.com>

Overview

What it is?

- J2EE Framework
- Comprehensive and modular
  - All tiers

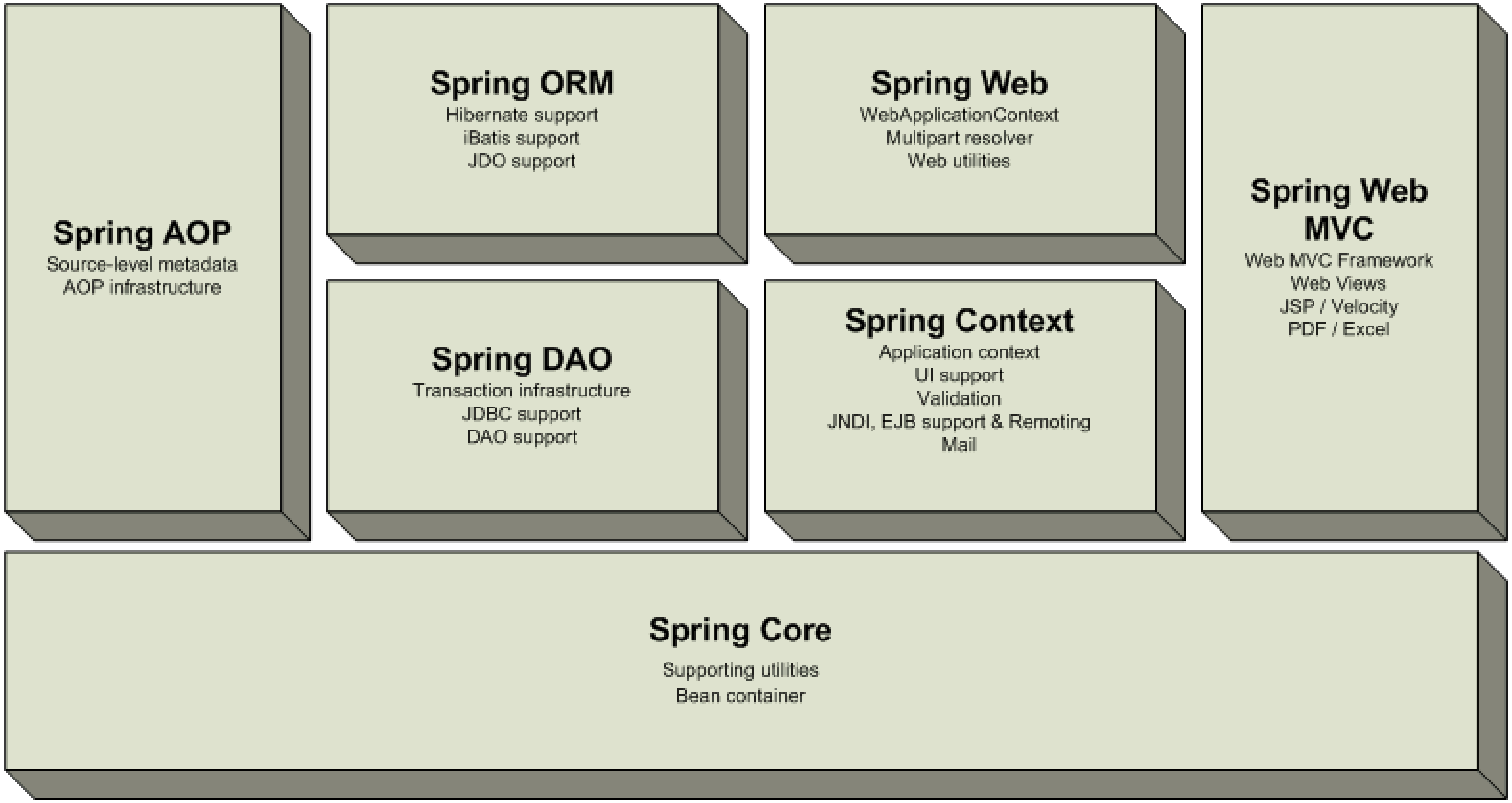
Spring objectives

- To make J2EE easier to use and promote good programming practice
- To make existing technologies easier to use
- To be portable between application servers
- To integrate with other projects (not reinvent the wheel)

Spring Benefits

- Organizes middle tier objects, takes care of plumbing
- Eliminates the proliferation of Singletons
- Applications depend on as few of its APIs as possible
- Applications are easy to unit test
- Can make the use of EJB an implementation choice
- Provides a consistent framework for data access
- You can choose to use just about any part of it in isolation, yet its architecture is internally consistent

Spring features



Spring AOP
Source-level metadata
AOP infrastructure

Spring ORM
Hibernate support
iBatis support
JDO support

Spring DAO
Transaction infrastructure
JDBC support
DAO support

Spring Web
WebApplicationContext
Multipart resolver
Web utilities

Spring Context
Application context
UI support
Validation
JNDI, EJB support & Remoting
Mail

Spring Web MVC
Web MVC Framework
Web Views
JSP / Velocity
PDF / Excel

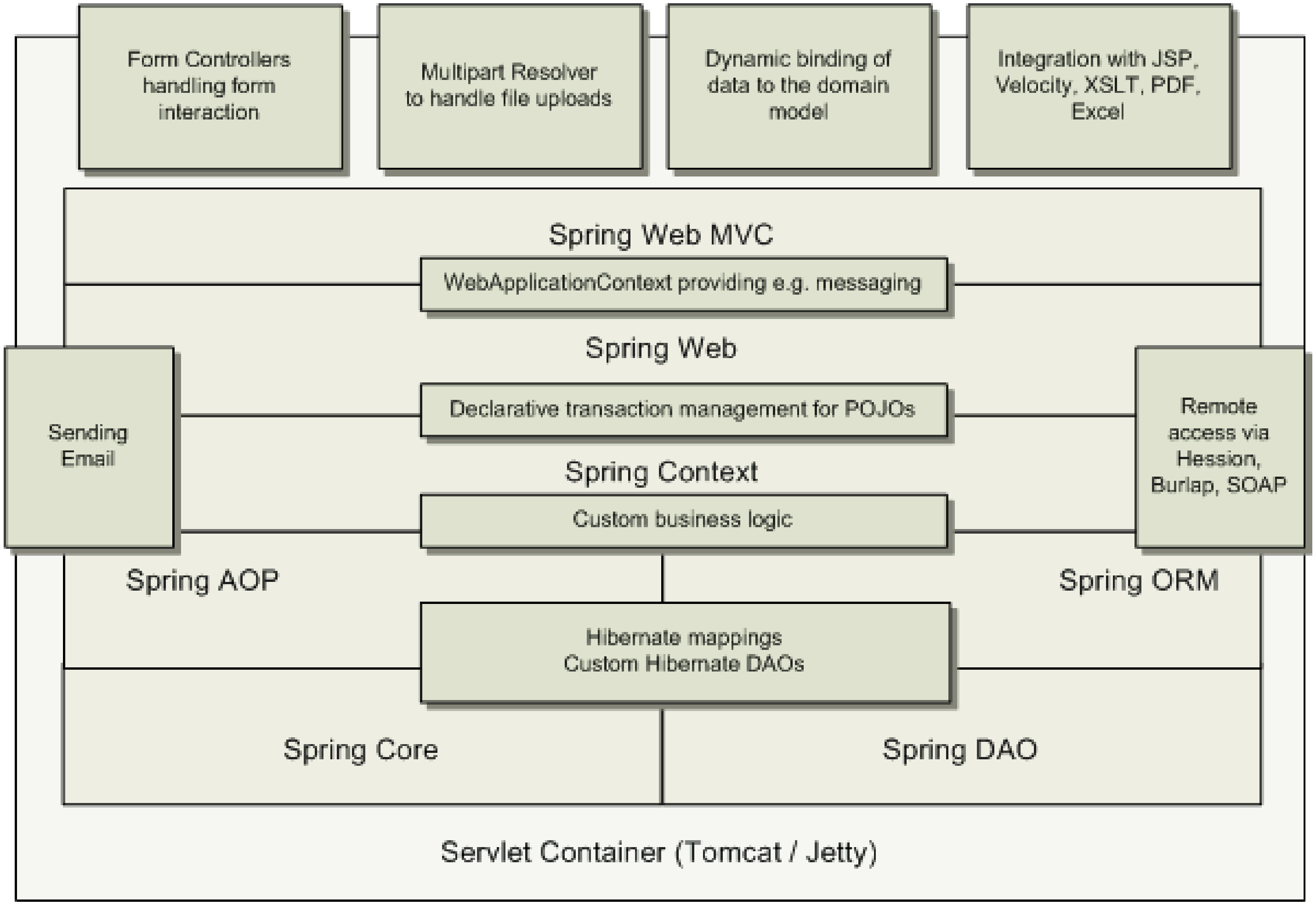
Spring Core
Supporting utilities
Bean container

Enterprise Architecture

- Architecture with Spring is flexible
- All tiers can benefit from Spring

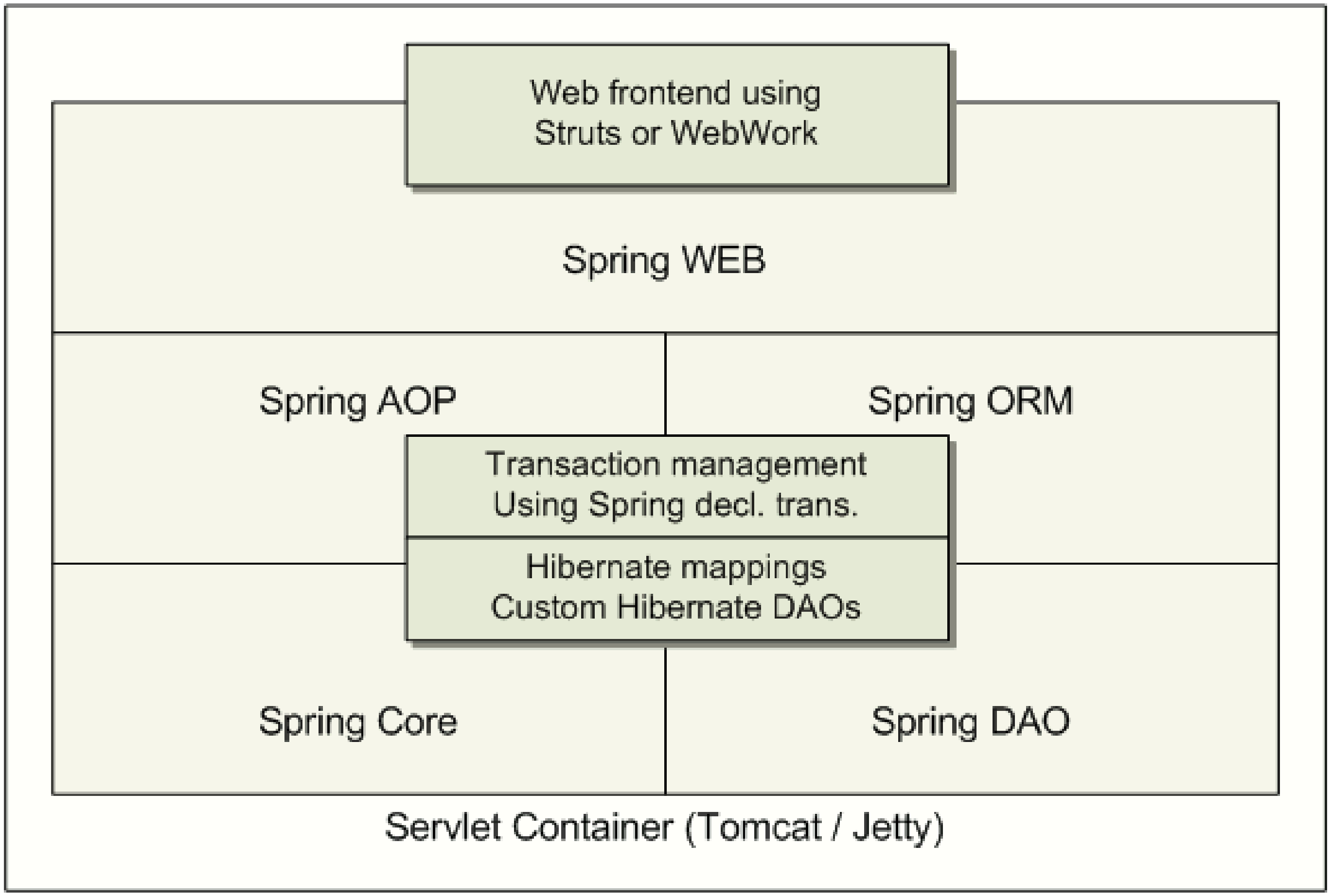
Scenario 1

- Full-fledged Spring web application


Form Controllers handling form interaction
Multipart Resolver to handle file uploads
Dynamic binding of data to the domain model
Integration with JSP, Velocity, XSLT, PDF, Excel
Spring Web MVC
WebApplicationContext providing e.g. messaging
Spring Web
Sending Email
Declarative transaction management for POJOs
Remote access via Hession, Burlap, SOAP
Spring Context
Custom business logic
Spring AOP
Spring ORM
Hibernate mappings
Custom Hibernate DAOs
Spring Core
Spring DAO
Servlet Container (Tomcat / Jetty)

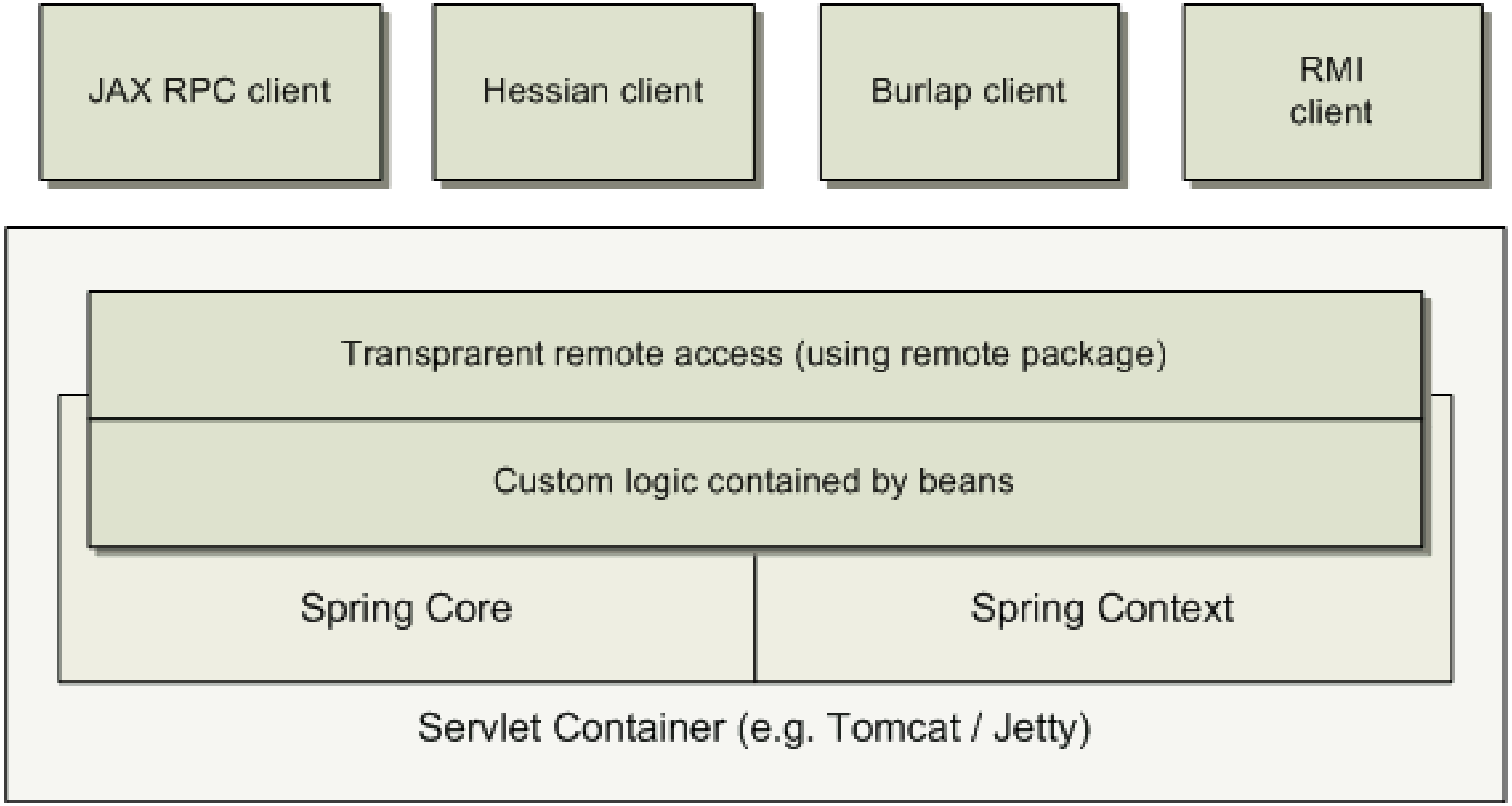
Scenario 2

- Spring middle-tier using a third-party web framework



Scenario 3

- Remoting usage scenario



JAX RPC client

Hessian client

Burlap client

RMI client

Transprarent remote access (using remote package)

Custom logic contained by beans

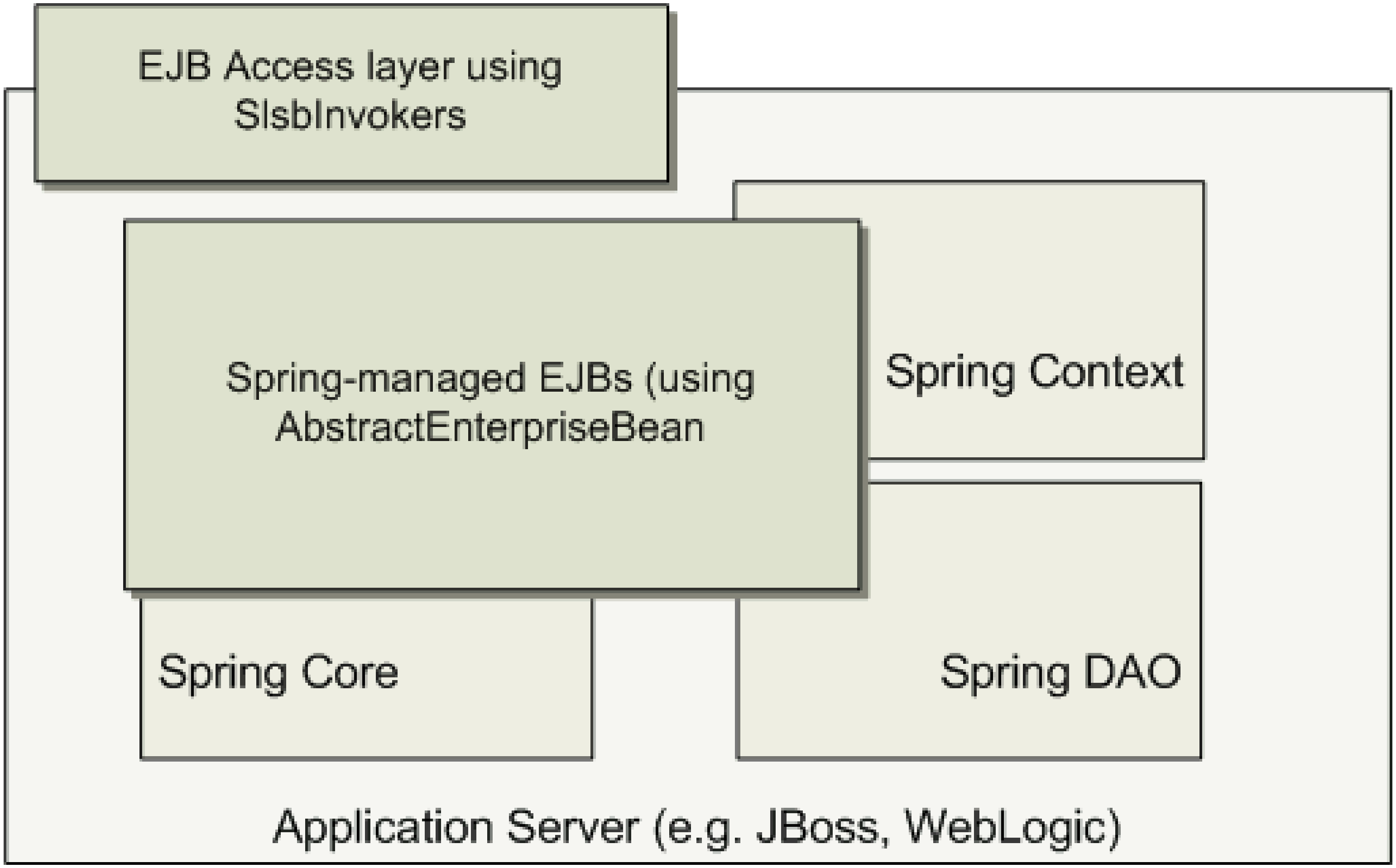
Spring Core

Spring Context

Servlet Container (e.g. Tomcat / Jetty)

Scenario 4

- EJBs wrapping existing POJOs



EJB Access layer using SlsbInvokers

Spring-managed EJBs (using AbstractEnterpriseBean

Spring Context

Spring Core

Spring DAO

Application Server (e.g. JBoss, WebLogic)

Spring background

- ▶ Background
  - – Based on "Expert one-on-one J2EE Design and Development" by Rod Johnson
  - – Open source project since February 2003
  - – Actual version is 1.0
- ▶ Main developers
  - – Jürgen Höller
  - – Rod Johnson

Part 1
Spring basics

Core

IoC/Dependency Injection

- Inversion of Control/Dependency Injection
  - Beans do not depend on framework
  - Container injects the dependencies
- Spring lightweight container
  - Configure and manage beans

BeanFactory

- Lightweight bean container
- Loads Bean definition
  - Bean definition contains
    - id/name
    - class
    - singleton or prototype
    - properties
    - constructor arguments
    - initialization method
    - destruction method

XmlBeanFactory

- BeanFactory implementation

- Beans definition example

```
<beans>
    <bean id="exampleBean" class="eg.ExampleBean"/>
    <bean id="anotherExample" class="eg.ExampleBeanTwo"/>
</beans>
```

XmlBeanFactory

▶ Usage example

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

ExampleBean eb =
        (ExampleBean)factory.getBean("exampleBean");

ExampleBeanTwo eb2 =
        (ExampleBeanTwo)factory.getBean("anotherExample");
```

Can throw **NoSuchBeanDefinitionException**

```
ExampleBean eb =
  (ExampleBean)factory.getBean("exampleBean", ExampleBean.class);
```

Can throw **BeanNotOfRequiredTypeException**

Bean collaborators

- Other beans your bean needs to do its work

```
package eg;
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;

    public void setBeanOne(AnotherBean b) { beanOne = b; }
    public void setBeanTwo(YetAnotherBean b) { beanTwo = b; }
}


<bean id="exampleBean" class="eg.ExampleBean">
    <property name="beanOne"><ref bean="anotherExampleBean"/></pro
    <property name="beanTwo"><ref bean="yetAnotherBean"/></propert
</bean>

<bean id="anotherExampleBean" class="eg.AnotherBean"/>
<bean id="yetAnotherBean" class="eg.YetAnotherBean"/>
```

Bean properties

- Setting bean properties

```
package eg;
public class ExampleBean {

    private String s;
    private int i;

    public void setStringProperty(String s) { this.s = s; }
    public void setIntegerProperty(int i) { this.i = i; }
}
```

```
<bean id="exampleBean" class="eg.ExampleBean">
    <property name="stringProperty"><value>Hi!</value></property>
    <property name="integerProperty"><value>1</value></property>
</bean>
```

Property editor

- Convert String to objects
- Implement java.beans.PropertyEditor
  - getValue()/setValue(), getAsText()/setAsText()
- Standard Java
  - Bool, Byte, Color, Double, Float, Font, Int, Long, Short, String
- Standard Spring
  - Class, File, Locale, Properties, StringArray, URL
- Custom Spring
  - CustomBoolean, CustomDate, CustomNumber, StringTrimmer

Standard property editors

- Examples

```
<property name="intProperty"><value>7</value></property>

<property name="doubleProperty"><value>0.25</value></property>

<property name="booleanProperty"><value>true</value></property>

<property name="colorProperty"><value>0,255,0</value></property>
```

java.awt.Color is initialized with RGB values

Spring property editors

- Examples

```
<property name="classProperty">
    <value>java.lang.Object</value>
</property>

<property name="fileProperty">
    <value>/home/ziba/file.txt</value>
</property>

<property name="localeProperty">
    <value>pt_BR</value>
</property>

<property name="urlProperty">
    <value>http://java.net</value>
</property>

<property name="stringArrayProperty">
    <value>foo,bar,baz</value>
</property>
```

Custom property editors

Date example

```
DateFormat fmt = new SimpleDateFormat("d/M/yyyy");
CustomDateEditor dateEditor = new CustomDateEditor(fmt, false);
beanFactory.registerCustomEditor(java.util.Date.class, dateEditor)

<property name="date"><value>19/2/2004</value></property>
```

StringTrimmer example

- Trim string and transform an empty string into null value

```
StringTrimmerEditor trimmer = new StringTrimmerEditor(true);
beanFactory.registerCustomEditor(java.lang.String.class, trimmer);

<property name="string1"><value>  hello  </value></property>
<property name="string2"><value></value></property>

<property name="string2"><null/></property>
```

java.util.Properties

- Properties example

```
<property name="propertiesProperty">
    <value>
        foo=1
        bar=2
        baz=3
    </value>
</property>


<property name="propertiesProperty">
    <props>
        <prop key="foo">1</prop>
        <prop key="bar">2</prop>
        <prop key="baz">3</prop>
    </props>
</property>
```

java.util.List

- List example

```
<property name="listProperty">
    <list>
        <value>a list element</value>
        <ref bean="otherBean"/>
        <ref bean="anotherBean"/>
    </list>
</property>
```

java.util.Set

- Set example

```
<property name="setProperty">
    <set>
        <value>a set element</value>
        <ref bean="otherBean"/>
        <ref bean="anotherBean"/>
    </set>
</property>
```

java.util.Map

- Map example

```
<property name="mapProperty">
    <map>
        <entry key="yup an entry">
            <value>just some string</value>
        </entry>
        <entry key="yup a ref">
            <ref bean="otherBean"/>
        </entry>
    </map>
</property>
```

Constructors

▶ Constructor example

```
public class ExampleBean {
    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;
    public ExampleBean(AnotherBean b1, YetAnotherBean b2, int i) {
        this.beanOne = b1;
        this.beanTwo = b2;
        this.i = i;
    }
}

<bean id="exampleBean" class="eg.ExampleBean">
    <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
    <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>
    <constructor-arg><value>1</value></constructor-arg>
</bean>

<bean id="anotherExampleBean" class="eg.AnotherBean"/>
<bean id="yetAnotherBean" class="eg.YetAnotherBean"/>
```

Bean lifecycle

- Beans can be initialized by the factory before its first use

```
public class ExampleBean {
    public void init() {
        // do some initialization work
    }
}

<bean id="exampleBean" class="eg.ExampleBean"
       init-method="init"/>
```

Bean lifecycle

- Beans can be cleaned up when not used anymore

```
public class ExampleBean {
    public void cleanup() {
        // do some destruction work
    }
}

<bean id="exampleBean" class="eg.ExampleBean"
       destroy-method="cleanup"/>
```

PropertyPlaceholderConfigurer

- Merge properties from an external Properties file

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

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

jdbc.properties

PropertyPlaceholderConfigurer

- Installing the configurer

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

Properties props = new Properties();
props.load(new FileInputStream("jdbc.properties"));

PropertyPlaceholderConfigurer cfg =
        new PropertyPlaceholderConfigurer();
cfg.setProperties(props);
cfg.postProcessBeanFactory(factory);

DataSource ds = (DataSource)factory.getBean("dataSource");
```

MethodInvokingFactoryBean

- Expose a bean that uses the singleton pattern

```
package eg;
public class MySingleton {
    private static MySingleton instance = new MySingleton();
    private MySingleton() {}
    public static MySingleton getInstance() {
        return instance;
    }
}
```

A **FactoryBean** delegates the bean creation to another class

```
<bean name="mySingleton"
    class="...beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod">
        <value>eg.MySingleton.getInstance</value>
    </property>
</bean>
```

FactoryBean reference

- Getting a reference to the factory itself

```
MySingleton singleton =
    (MySingleton)ctx.getBean("mySingleton");
```

Return the **bean created by the factory**

```
FactoryBean factory =
    (FactoryBean)ctx.getBean("&mySingleton");
```

Return the **factory**

Advanced features

- Singletons/Prototypes
- Autowiring
  - By **type** requires a single instance of each required type
  - By **name** requires a bean name that matches each property name (for non-simple properties)
- Dependency checking
- BeanWrapper
- InitializingBean/DisposableBean interfaces
- BeanFactoryAware/BeanNameAware interfaces

Application Context

What is an ApplicationContext?

- Aggregates info about the application that can be used by all components

- Location of bean definitions
- Loading of multiple contexts
- Hierarchical contexts
- i18n, message sources
- Access to resources
- Event propagation

ApplicationContext

- ▶ Extends BeanFactory
- ▶ Can have a parent context
- ▶ Implementations
  - – FileSystemXmlApplicationContext
  - – ClassPathXmlApplicationContext
  - – XmlWebApplicationContext
- ▶ Example

```
ApplicationContext ctx =
       new FileSystemXmlApplicationContext("c:/beans.xml");

ExampleBean eb = (ExampleBean)ctx.getBean("exampleBean");
```

ApplicationContext

- ApplicationContext can be read from many files

```
String[] ctxs = new String[]{"ctx1.xml", "ctx2.xml"};

ApplicationContext ctx = new FileSystemXmlApplicationContext(ctxs);
```

Hierarchical contexts

- If a bean is not found in a context it is searched in the parent context

- Creating a context hierarchy

```
ApplicationContext parent =
        new ClassPathXmlApplicationContext("ctx1.xml");

ApplicationContext ctx =
        new FileSystemXmlApplicationContext("ctx2.xml", parent);
```

Resources

- ApplicationContext deals with resource location
- ApplicationContext method
  - Resource getResource(String location)
    - fully qualified URLs, e.g. "**`file:C:/test.dat`**"
    - relative file paths, e.g. "**`WEB-INF/test.dat`**"
    - classpath pseudo-URLs, e.g. "**`classpath:test.dat`**"

```
interface Resource {
    boolean exists();
    boolean isOpen();
    String getDescription();
    File getFile() throws IOException;
    InputStream getInputStream() throws IOException;
}
```

ResourceEditor

- Built-in PropertyEditor
- Can be used to configure Resource properties in bean definitions
- Example

```
<property name="resourceProperty">
    <value>example/image.gif</value>
</property>
```

i18n

- Internationalization of application messages
- ApplicationContext method
  - String getMessage (String code, Object[] args, String default, Locale loc)

Delegated to a “**messageSource**” bean

i18n

- ApplicationContext searches for the “messageSource” bean
  - Must implement MessageSource interface
- Example
  - definition of two resource bundles in classpath: messages and errors

```
<bean id="messageSource" class="...ResourceBundleMessageSource">
    <property name="basenames">
        <value>messages,errors</value>
    </property>
</bean>
```

Search in classpath:

```
messages_pt_BR.properties    errors_pt_BR.properties
messages_pt.properties       errors_pt.properties
messages.properties          errors.properties
```

Events

- Event propagation
  - ApplicationContext handles events and call listeners
  - Beans must implement ApplicationListener to receive events
  - Applications can extend ApplicationEvent
  - Built-in events
    - ContextRefreshedEvent
    - ContextClosedEvent
    - RequestHandledEvent

Events

▶ Listening events

```
public class MyListenerBean implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent e) {
        // process event
    }
}
```

▶ Sending an event

```
public class ExampleBean implements ApplicationContextAware {
    ApplicationContext ctx;
    public void setApplicationContext(ApplicationContext ctx)
        throws BeansException {
        this.ctx = ctx;
    }

    public void sendEvent() {
        ctx.publishEvent(new MyApplicationEvent(this));
    }
}
```

BeanFactoryPostProcessor

- Can be used to configure the BeanFactory or beans in it
  - Application contexts can auto-detect BeanFactoryPostProcessor beans in their bean definitions and apply them before any other beans get created
- The post processor bean must implement BeanFactoryPostProcessor interface

BeanFactoryPostProcessor

- Example: adding custom editors to a context

```
public class MyPostProcessor implements BeanFactoryPostProcessor {

    void postProcessBeanFactory(
        ConfigurableListableBeanFactory bf) {

        DateFormat fmt = new SimpleDateFormat("d/M/yyyy");
        CustomDateEditor dateEditor =
            new CustomDateEditor(fmt, false);

        bf.registerCustomEditor(java.util.Date.class, dateEditor);
    }

}


<bean id="myPostProcessor" class="eg.MyPostProcessor"/>
```

CustomEditorConfigurer

- BeanFactoryPostProcessor implementation that allows for convenient registration of custom property editors

```
<bean id="customEditorConfigurer" class="...CustomEditorConfigurer">
  <property name="customEditors">
    <map>
      <entry key="java.util.Date">
        <bean class="...CustomDateEditor">
          <constructor-arg index="0">
            <bean class="java.text.SimpleDateFormat">
              <constructor-arg><value>d/M/yyyy</value></constructor-arg>
            </bean>
          </constructor-arg>
          <constructor-arg index="1"><value>false</value></constructor-arg>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

Typical application contexts

- Application contexts are usually associated with a scope defined by the J2EE server
  - Web application (javax.servlet.ServletContext): Spring provides the ability to instantiate such a context through a listener or servlet
  - Servlet: each servlet can have its own application context, derived from the web application context
  - EJB: loaded from an XML document in the EJB Jar file
- There is no need to use a Singleton to bootstrap a bean factory

AOP

Aspect-Oriented Programming

- Complements OOP
- Decomposition of *aspects* (or concerns)
- Modularization of concerns that would otherwise cut across multiple objects
- Usages
  - Persistence
  - Transaction management
  - Security
  - Logging
  - Debugging

AOP concepts

- ▶ Aspect
  - – Modularization of a concern
- ▶ Joinpoint
  - – Point during the execution of a program
- ▶ Advice
  - – Action taken at a particular joinpoint
- ▶ Pointcut
  - – Set of joinpoints specifying when an advice should fire
- ▶ Introduction
  - – Adding methods or fields to an advised class

Pointcut

- Set of joinpoints specifying when an advice should fire

```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
}

public interface ClassFilter {
    boolean matches(Class clazz);
}

public interface MethodMatcher {
    boolean matches(Method m, Class targetClass);
    boolean matches(Method m, Class targetClass, Object[] args);
    boolean isRuntime();
}
```

Restricts the pointcut to a given set of target classes

Static pointcuts don't use the method arguments

Pointcut implementations

- Regexp

```
<bean id="gettersAndSettersPointcut"
    class="org.springframework.aop.support.RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*\.get.*</value>
            <value>.*\.set.*</value>
        </list>
    </property>
</bean>
```

Match a Perl5 regexp to a fully qualified method name

Advice

- Action taken at a particular joinpoint

```
public interface MethodInterceptor extends Interceptor {
    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

Spring implements an advice with an ***interceptor chain*** around the jointpoint

- Example

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation)
        throws Throwable {
        System.out.println(">> " + invocation); // before
        Object rval = invocation.proceed();
        System.out.println("<< Invocation returned"); // after
        return rval;
    }
}
```

Advice types

- ▶ Around advice
  - – The previous example
- ▶ Before advice
- ▶ Throws advice
- ▶ After returning advice
- ▶ Introduction advice

Spring Advisors

- PointcutAdvisor = Pointcut + Advice
- Each built-in advice has an advisor

- Example

```
<bean id="gettersAndSettersAdvisor"
    class="...aop.support.RegexpMethodPointcutAroundAdvisor">
    <property name="interceptor">
        <ref local="interceptorBean"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*\.get.*</value>
            <value>.*\.set.*</value>
        </list>
    </property>
</bean>
```

ProxyFactory

- With a ProxyFactory you get advised objects
  - You can define pointcuts and advices that will be applied
  - It returns an interceptor as a proxy object
  - It uses Java Dynamic Proxy or CGLIB 2
    - It can proxy interfaces or classes

- Creating AOP proxies programmatically

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);

MyBusinessInterface b = (MyBusinessInterface)factory.getProxy();
```

ProxyFactoryBean

- ▶ Used to get proxies for beans
- ▶ The bean to be proxied

```
<bean id="personTarget" class="eg.PersonImpl">
    <property name="name"><value>Tony</value></property>
    <property name="age"><value>51</value></property>
</bean>
```

PersonImpl implements **Person** interface

ProxyFactoryBean

▶ The interceptors/advisors

```
<bean id="myAdvisor" class="eg.MyAdvisor">
    <property name="someProperty"><value>Something</value></property>
</bean>

<bean id="debugInterceptor" class="...aop.interceptor.NopInterceptor">
</bean>
```

▶ The proxy

```
<bean id="person" class="...aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"><value>eg.Person</value></property>

    <property name="target"><ref local="personTarget"/></property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

ProxyFactoryBean

- Using the bean
  - Clients should get the **person** bean instead of **personTarget**
  - Can be accessed in the application context or programmaticaly

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref local="person" /></property>
</bean>

Person person = (Person) factory.getBean("person");
```

ProxyFactoryBean

- If you need to proxy a class instead of an interface
  - Set the property proxyTargetClass to true, instead of proxyInterfaces
  - Proxy will extend the target class
    - constructed by CGLIB

```
<bean id="person" class="...aop.framework.ProxyFactoryBean">
    <property name="proxyTargetClass"><value>true</value></property>

    <property name="target"><ref local="personTarget"/></property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

AutoProxy

- ▸ Automatic proxy creation
  - – Just declare the targets
  - – Selected beans will be automatically proxied

- ▸ No need to use a ProxyFactoryBean for each target bean

BeanNameAutoProxyCreator

- Select targets by bean name

```
<bean id="employee1" class="eg.Employee">...</bean>
<bean id="employee2" class="eg.Employee">...</bean>

<bean id="myInterceptor" class="eg.DebugInterceptor"/>

<bean id="beanNameProxyCreator"
    class="...aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames"><value>employee*</value></property>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

AdvisorAutoProxyCreator

- Automatically applies advisors in context to beans
  - Each advisor has a pointcut and an advice
  - If a pointcut applies to a bean it will be intercepted by the advice
- Useful to apply the same advice consistently to many business objects
- Impossible to get an un-advised object

AdvisorAutoProxyCreator

▶ Example

```
<bean id="debugInterceptor" class="app.DebugInterceptor"/>

<bean id="getterDebugAdvisor"
    class="...aop.support.RegexpMethodPointcutAdvisor">
    <constructor-arg>
        <ref bean="debugInterceptor"/>
    </constructor-arg>
    <property name="pattern"><value>.*\.get.*</value></property>
</bean>
```

This advisor applies **debugInterceptor** to all **get** methods of any class

```
<bean id="autoProxyCreator"
    class="...aop.framework.autoproxy.AdvisorAutoProxyCreator">
    <property name="proxyTargetClass"><value>true</value></propert
</bean>
```

Advanced AOP Features

- Metadata-driven autoproxying
- TargetSources
  - Hot swappable target sources
    - Allow the target of a proxy to be switched while allowing callers to keep their references to it
  - Pooling target sources
    - A pool of identical instances is maintained, with method invocations going to free objects in the pool

Metadata attributes

Source-level metadata

- The addition of *attributes* or *annotations* to program elements: usually, classes and/or methods

Annotated class

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.DefaultTransactionAttribute()
 */
public class PetStoreImpl implements PetStoreFacade, OrderService {
    ...
}
```

Annotated method

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.RuleBasedTransactionAttribute()
 * @@org.springframework.transaction.interceptor.RollbackRuleAttribute(Exception.class)
 * @@org.springframework.transaction.interceptor.NoRollbackRuleAttribute("ServletException
 */
public void echoException(Exception ex) throws Exception {
    ....
}
```

Source-level metadata

- Spring provides a facade to metadata implementations
  - Uses Jakarta Commons Attributes
    - Build process needs an *attribute compilation* step
  - JSR-175 (JDK 1.5) planned

- Uses
  - With AOP
    - Attributes are used to specify aspects
  - Minimize web tier configuration
    - url to controller mapping
  - Validation

Part 2
Spring Integration

Integration

- BeanFactory, ApplicationContext and AOP are the base of Spring
- From now on we will see Spring integration with other tools or APIs

Mail

SimpleMailMessage

- Creating a message

```
SimpleMailMessage msg = new SimpleMailMessage();

msg.setFrom("me@mail.org");
msg.setTo("you@mail.org");
msg.setCc(new String[] {"he@mail.org", "she@mail.org"});
msg.setBcc(new String[] {"us@mail.org", "them@mail.org"});
msg.setSubject("my subject");
msg.setText("my text");
```

MessageSender

- Defining a message sender

```
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host"><value>smtp.mail.org</value></property>
    <property name="username"><value>joe</value></property>
    <property name="password"><value>abc123</value></property>
</bean>
```

- Sending the message

```
MailSender sender = (MailSender) ctx.getBean("mailSender");

sender.send(msg);
```

Scheduling

Schedulers

- Built-in support for
  - Java 2 Timer
    - Timer
    - TimerTask
  - Quartz (http://www.quartzscheduler.org/)
    - Schedulers
    - JobDetails
    - Triggers

SUMMA

ScheduledTimerTask

- The task that we want to run

```
public class MyTask extends TimerTask {
    public void run() {
        // do something
    }
}
```

Java bean that wraps a scheduled **java.util.TimerTask**

```
<bean id="myTask"
    class="...scheduling.timer.ScheduledTimerTask">
    <property name="timerTask">
        <bean class="eg.MyTask"/>
    </property>
    <property name="delay"><value>60000</value></property>
    <property name="period"><value>1000</value></property>
</bean>
```

TimerFactoryBean

- Creating the scheduler

Creates a **java.util.Timer** object

```
<bean id="scheduler"
    class="...scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list><ref bean="myTask"/></list>
    </property>
</bean>
```

- The Timer starts at bean creation time

JNDI

JndiTemplate

▸ Using JndiTemplate

```
Properties p = new Properties();
p.setProperty("java.naming.factory.initial",
              "org.jnp.interfaces.NamingContextFactory");
p.setProperty("java.naming.provider.url",
              "jnp://localhost:1099");

JndiTemplate jndi = new JndiTemplate(p);

Properties env = jndi.getEnvironment();

try {
    jndi.bind("Something", something);
    Object o = jndi.lookup("Something");
    jndi.unbind("Something");
}
catch(NamingException e) {
    ...
}
```

JndiObjectFactoryBean

- Using a bean instead of a lookup

```
<bean id="jndiTemplate"
    class="org.springframework.jndi.JndiTemplate">
    <constructor-arg>
        <props>
            <prop key="java.naming.factory.initial">org.jnp.interfaces.Namin
            <prop key="java.naming.provider.url">jnp://localhost:1099</prop>
        </props>
    </constructor-arg>
</bean>
```

A **FactoryBean** delegates the bean creation to another class

```
<bean id="something"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiTemplate"><ref bean="jndiTemplate"/></property>
    <property name="jndiName"><value>Something</value></property>
</bean>
```

```
Object o = ctx.getBean("something");
```

JDBC

JDBC abstraction

- Make JDBC easier to use and less error prone
- Framework handles the creation and release of resources
- Framework takes care of all exception handling

JdbcTemplate

- Executes SQL queries, update statements or stored procedure calls
- Iteration over ResultSets and extraction of returned parameter values

- Example

```
DataSource ds = DataSourceUtils.getDataSourceFromJndi("MyDS");
JdbcTemplate jdbc = new JdbcTemplate(ds);

jdbc.execute("drop table TEMP");

jdbc.update("update EMPLOYEE set FIRSTNME=? where LASTNAME=?",
            new String[] {"JOE", "LEE"});
```

JdbcTemplate

- ▶ Queries, using convenience methods

```
int maxAge = jdbc.queryForInt("select max(AGE) from EMPLOYEE");


String name = (String)jdbc.queryForObject(
        "select FIRSTNME from EMPLOYEE where LASTNAME='LEE'",
        String.class);


List employees = jdbc.queryForList(
        "select EMPNO, FIRSTNME, LASTNAME from EMPLOYEE");
```

Returns an **ArrayList** (one entry for each row) of **HashMaps** (one entry for each column using the column name as the key)

SUMMA

JdbcTemplate

▶ Queries, using callback method

```
final List employees = new LinkedList();

jdbc.query("select EMPNO, FIRSTNME, LASTNAME from EMPLOYEE",
    new RowCallbackHandler() {

    public void processRow(ResultSet rs) throws SQLException {

        Employee e = new Employee();
        e.setEmpNo(rs.getString(1));
        e.setFirstName(rs.getString(2));
        e.setLastName(rs.getString(3));

        employees.add(e);
    }

});
```

employees list will be populated with **Employee** objects

JdbcTemplate

- Stored procedures

```
jdbc.call(new CallableStatementCreator() {
  public CallableStatement createCallableStatement(Connection conn)
    throws SQLException {
    return conn.prepareCall("my query");
  }
}, params);
```

JdbcTemplate

- Batch updates

```
BatchPreparedStatementSetter setter =
    new BatchPreparedStatementSetter() {

    public void setValues(PreparedStatement ps, int i)
        throws SQLException {
        ...
    }
    public int getBatchSize() {
        return ...;
    }

};

jdbc.batchUpdate("update ...", setter);
```

SqlQuery/SqlUpdate objects

- Encapsulate queries and updates into Java classes

```
class EmployeeQuery extends MappingSqlQuery {

    public EmployeeQuery(DataSource ds) {
        super(ds, "select EMPNO, FIRSTNME, LASTNAME from EMPLOYEE where EMPNO = ?");
        declareParameter(new SqlParameter(Types.CHAR));
        compile();
    }

    protected Object mapRow(ResultSet rs, int rownum) throws SQLException {
        Employee e = new Employee();
        e.setEmpNo(rs.getString("EMPNO"));
        e.setFirstName(rs.getString("FIRSTNME"));
        e.setLastName(rs.getString("LASTNAME"));
        return e;
    }

    public Employee findEmployee(String id) {
        return (Employee) findObject(id);
    }
}
```

Map a result set row to a Java object

Convenience method to do strong typing

SqlFunction

- Encapsulate queries that return a single row

```
SqlFunction sf = new SqlFunction(dataSource,
        "select count(*) from mytable");
sf.compile();

int rows = sf.run();
```

Exception handling

- Translates SQLException to DataAccessException hierarchy
  - Generic, more informative, DB/JDBC independent (sql error codes are mapped to exceptions)
- Uses RuntimeExceptions (unchecked)
- we can still recover from an unchecked data access exception

```
try {
    // do work
}
catch (OptimisticLockingFailureException ex) {
    // I'm interested in this
}
```

Database connections

- DataSourceUtils
  - getConnection(), getDataSourceFromJndi()
  - closeConnectionIfNecessary()
- DriverManagerDataSource
  - Returns a new connection every time
  - To be used outside a container or in tests
- SingleConnectionDataSource
  - Returns always the same connection
  - To be used outside a container or in tests

Transaction Management

Transactions

- Global transactions
  - managed by the application server, using JTA
  - ability to work with multiple transactional resources
- Local transactions
  - resource-specific: for example, a transaction associated with a JDBC connection
  - cannot work across multiple transactional resources
  - cannot run within a global JTA transaction
- Different programming models

Spring solution

- Uses the same programming model for global or local transactions
  - Different transaction management strategies in different environments
- Transaction management can be
  - Programmatic
  - Declarative (like EJB CMT)

Transaction abstraction

- Transactions are abstracted by the interface PlatformTransactionManager
  - getTransaction(TransactionDefinition)
  - commit(TransactionStatus)
  - rollback(TransactionStatus)
- TransactionDefinition
  - Isolation, propagation, timeout, read-only status
- TransactionStatus
  - isNewTransaction()
  - setRollbackOnly()
  - isRollbackOnly()

Transaction managers

- Built-in platform transaction managers
  - JtaTransactionManager
  - DataSourceTransactionManager
  - HibernateTransactionManager
  - JdoTransactionManager

Examples

▸ Defining a JtaTransactionManager

```
<bean id="dataSource" class="...jndi.JndiObjectFactoryBean">
    <property name="jndiName"><value>MyDS</value></property>
</bean>
```

Data sources must be configured in the app server as transactional resources

```
<bean id="transactionManager"
    class="...transaction.jta.JtaTransactionManager"/>
```

Examples

▸ Defining a DataSourceTransactionManager

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


<bean id="transactionManager"
    class="...jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
</bean>
```

Examples

▶ Defining a HibernateTransactionManager

```
<bean id="sessionFactory"
    class="...orm.hibernate.LocalSessionFactoryBean">
    ...
</bean>


<bean id="transactionManager"
    class="...orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

To make Hibernate use JTA you don't need HibernateTransactionManager, just configure a **JtaTransactionManager** and give to sessionFactory data sources obtained from JNDI

TransactionTemplate

- Programmatic transaction management
- Create a TransactionTemplate

```
PlatformTransactionManager transactionManager =
  (PlatformTransactionManager) ctx.getBean("myTransactionManager");

TransactionTemplate transaction =
  new TransactionTemplate(transactionManager);
```

- Execute in a transaction

```
transaction.execute(new TransactionCallbackWithoutResult() {
    public void doInTransactionWithoutResult(TransactionStatus s) {
        updateOperation1();
        updateOperation2();
    }
});
```

TransactionTemplate

- Additional methods of TransactionTemplate
  - setPropagationBehavior(int)
  - setIsolationLevel(int)
  - setReadOnly(boolean)
  - setTimeout(int)

Advantages

- Transition from one transaction manager
  - Is just a matter of configuration
  - No need to change the code
- The same component can run in
  - Application server with JTA transactions
  - Stand-alone application or web container
    - with JDBC transactions
    - with an open source JTA as JOTM

Declarative transactions

- No need of TransactionTemplate
- Implemented using Spring AOP
- Similar to EJB CMT
  - You specify transaction behaviour (or lack of it) down to individual methods

Declarative transactions

- Different from EJB CMT
  - Can be applied to any POJO
  - Not tied to JTA (works with JDBC, JDO, Hibernate)
  - Has declarative rollback rules
  - Customisable transactional behaviour
  - Does not support propagation of transaction contexts across remote calls

TransactionAttributeSource

- Defines how transaction properties are applied
- TransactionAttributeEditor reads definition of form
  - **`PROPAGATION_NAME,ISOLATION_NAME,readOnly,+Exception1,-Exception2`**
  - A "+" before an exception name substring indicates that transactions should commit even if this exception is thrown; a "-" that they should roll back
- Example
  - **`PROPAGATION_MANDATORY,ISOLATION_DEFAULT, -CreateException,-DuplicateKeyException`**

Declarative transactions

- Defining a transaction interceptor

```
<bean id="txAttributes"
    class="...MatchAlwaysTransactionAttributeSource">
    <property name="transactionAttribute">
        <value>PROPAGATION_REQUIRED</value>
    </property>
</bean>
```

MatchAlwaysTransactionAttributeSource applies the same attributes to all methods

```
<bean id="txInterceptor"
    class="...transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager">
        <ref bean="myTransactionManager"/>
    </property>
    <property name="transactionAttributeSource">
        <ref bean="txAttributes"/>
    </property>
</bean>
```

Declarative transactions

- An alternative TransactionAttributeSource

```
<bean id="txAttributes"
    class="...interceptor.NameMatchTransactionAttributeSource">
    <property name="properties">
        <value>
            get*=PROPAGATION_REQUIRED,readOnly
            find*=PROPAGATION_REQUIRED,readOnly
            load*=PROPAGATION_REQUIRED,readOnly
            store*=PROPAGATION_REQUIRED
        </value>
    </property>
</bean>
```

NameMatchTransactionAttributeSource applies specific attributes to methods that match to a pattern

Declarative transactions

- ▶ Autoproxy for transactional beans

```
<bean id="autoProxyCreator"
    class="...framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="interceptorNames">
        <value>txInterceptor</value>
    </property>
    <property name="beanNames">
        <value>*Dao</value>
    </property>
</bean>
```

Declarative transactions

- Using metadata attributes

```
<bean id="autoproxy"
    class="...aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
</bean>

<bean id="txAdvisor"
    class="...transaction.interceptor.TransactionAttributeSourceAdvisor"
    autowire="constructor">
</bean>

<bean id="txInterceptor"
    class="...transaction.interceptor.TransactionInterceptor"
    autowire="byType">
</bean>

<bean id="txAttributeSource"
    class="...transaction.interceptor.AttributesTransactionAttributeSource"
    autowire="constructor">
</bean>

<bean id="attributes"
    class="...metadata.commons.CommonsAttributes">
</bean>
```

PlatformTransactionManager

ORM

ORM

- ORM
  - Object-Relational Mapping
- Built-in support to
  - JDO
  - iBatis
  - Hibernate

Hibernate configuration

- Define a DataSource and an Hibernate SessionFactory

```
<bean id="dataSource" ...> ... </bean>

<bean id="sessionFactory" class="...LocalSessionFactoryBean">
    <property name="mappingResources">
        <list>
            <value>employee.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">....DB2Dialect</prop>
        </props>
    </property>
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
```

HibernateTemplate

- Create HibernateTemplate

```
SessionFactory sessionFactory =
        (SessionFactory) ctx.getBean("sessionFactory");

HibernateTemplate hibernate =
        new HibernateTemplate(sessionFactory);
```

- Load & update

```
Employee e = (Employee) hibernate.load(Employee.class, "000330");
e.setFirstName("BOB");
hibernate.update(e);
```

HibernateTemplate

- Queries, using convenience methods

```
List employees = hibernate.find("from app.Employee");


List list = hibernate.find(
        "from app.Employee e where e.lastName=?",
        "LEE",
        Hibernate.STRING);

List list = hibernate.find(
        "from app.Employee e where e.lastName=? and e.firstName=?",
        new String[] { "BOB", "LEE" },
        new Type[] {Hibernate.STRING , Hibernate.STRING });
```

HibernateTemplate

- Queries, using callback method

```
List list = (List) hibernate.execute(new HibernateCallback() {
    public Object doInHibernate(Session session)
        throws HibernateException {

        List result = session.find("from app.Employee");
        // do some further stuff with the result list

        return result;
    }
});
```

Exception handling

- Translates Hibernate exceptions to DataAccessException hierarchy
- Uses the same strategy as with JDBC

EJB

Spring and EJBs

- Spring is a lightweight container and can be used instead of EJBs in many cases; however...
- Spring makes it easier to access and implement EJBs

EJB access

- With EJBs it is usual to have
  - A ServiceLocator
    - Takes care of JNDI, initial context, EJB home lookup
  - A BusinessDelegate
    - Reduces coupling, hides the implementation

- With Spring these patterns are not necessary

Accessing a Local SLSB

- To use a Local, Stateless, Session Bean

```
<bean id="myComponent"
    class="...ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName">
        <value>myComponent</value>
    </property>
    <property name="businessInterface">
        <value>com.mycom.MyComponent</value>
    </property>
</bean>
```

Creates a proxy (the *business delegate*) that uses a *service locator* to access the EJB

- You can swap the bean implementation without changing the client code
  - (the client uses the business interface not an EJB specific interface)

Accessing a Remote SLSB

- To use a Remote, Stateless, Session Bean

```
<bean id="myComponent"
    class="...SimpleRemoteStatelessSessionProxyFactoryBean">
    <property name="jndiEnvironment">
        <ref bean="myEnvironment"/>
    </property>
    <property name="jndiName">
        <value>myComponent</value>
    </property>
    <property name="businessInterface">
        <value>com.mycom.MyComponent</value>
    </property>
</bean>
```

EJB implementation

- AbstractEnterpriseBean
  - Loads a BeanFactory
    - EJB enviroment variable **`ejb/BeanFactoryPath`** specifies the location *on the classpath* of an XML bean factory definition
      - E.g. /com/mycom/mypackage/mybeans.xml
    - Default bean factory is XmlApplicationContext
- Applications should use the EJB only as a facade
  - Business logic deferred to beans in BeanFactory

Implementing a SLSB

- Stateless Session Beans
- Extend AbstractStatelessSessionBean
  - Saves the session context
  - Empty implementation of ejbRemove()
  - ejbCreate() method
  - Throws exception in ejbActivate() and ejbPassivate()
- Subclasses must implement onEjbCreate()

Implementing a SLSB

- Example

```
class MySlsb extends AbstractStatelessSessionBean {
    protected void onEjbCreate() throws CreateException {
        ...
    }

    public void businessMethod() {
        BeanFactory bf = getBeanFactory();
        MyBusinessBean mbb = bf.getBean("myBusinessBean");
        ...
    }
}
```

Implementing a SFSB

- Stateful Session Beans
- Extend AbstractStatefulSessionBean
  - Saves the session context
  - Empty implementation of ejbRemove()
  - ejbCreate() method
- Subclasses must implement ejbCreate(), ejbActivate() and ejbPassivate()

Implementing a SFSB

- Example

```
class MySfsb extends AbstractStatefulSessionBean {
    public void ejbCreate() throws CreateException {
        loadBeanFactory();
        ...
    }
    public void ejbActivate() {
        ...
    }
    public void ejbPassivate() {
        ...
    }

    public void businessMethod() {
        BeanFactory bf = getBeanFactory();
        MyBusinessBean mbb = bf.getBean("myBusinessBean");
        ...
    }
}
```

Implementing a MDB

- Example

```
class MyMdb extends AbstractJmsMessageDrivenBean {
    protected void onEjbCreate() throws CreateException {
        ...
    }

    public void onMessage(Message message) {
        BeanFactory bf = getBeanFactory();
        MyBusinessBean mbb = bf.getBean("myBusinessBean");
        ...
    }
}
```

Web

WebApplicationContext

- Application context located in the war file
  - Single root context per application
  - Default: /WEB-INF/applicationContext.xml
- Context is loaded by
  - ContextLoaderListener (Servlet 2.4)
  - ContextLoaderServlet (Servlet 2.3)
- Can be used with any web framework
  - Use Spring simply as a library

WebApplicationContext

- Example
  - web.xml

Load root application context from
/WEB-INF/applicationContext.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>...web.context.ContextLoaderListener</listener-class>
</listener>
```

  - Inside a Servlet

```
WebApplicationContextUtils.getWebApplicationContext(ServletContext);
```

Web MVC

To do

- To do

- Reference
  - Developing a Spring Framework MVC application step-by-step
    - http://www.springframework.org/docs/MVC-step-by-step/Spring-MVC-step-by-step.html

Remoting

RemoteExporter

- Any bean in the context can be exported
- A RemoteExporter exports a bean as a remote service
- Built-in support for
  - RMI
  - JAX-RPC
  - Burlap
  - Hessian

SUMMA

RmiServiceExporter

- The service to be exported

```
class MyServiceImpl implements MyService {
   ...
}

<bean id="myService" class="app.MyServiceImpl"/>
```

- The service exporter

```
<bean id="myService-rmi"
    class="...remoting.rmi.RmiServiceExporter">
    <property name="service"><ref local="myService"/></property>
    <property name="serviceInterface">
        <value>app.MyService</value>
    </property>
    <property name="serviceName">
        <value>myService</value>
    </property>
</bean>
```

More...

Road map

- Spring 1.1
  - JMS support
  - JMX support
  - declarative rules-based validator
  - AOP pointcut expression language, JSR-175 preview
- Spring 1.2
  - OGNL support
  - JCA support
  - enhanced RMI support
- Spring 1.3?
  - JSF
  - Portlets

Related projects

- Rich Client Platform (sandbox)
  - Spring RCP
- Validation (sandbox)
  - Commons-validator
  - Attribute based
- Security
  - Acegi Security System for Spring
    - http://acegisecurity.sourceforge.net/

References

- The Spring web site
  - http://www.springframework.org/
- Automated build (Javadocs, source Xref, changes, unit tests, etc.)
  - http://monkeymachine.co.uk/spring/maven-reports.html
- Mailing list archives (springframework-user)
  - http://news.gmane.org/gmane.comp.java.springframework.user
- Expert One-on-One J2EE Development without EJB (to be published)
  - By Rod Johnson, Jürgen Höller