Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Spring Framework

March 24, 2008

-Narendra Thite

CONFIDENTIAL

Agenda

- Day 1
  - Introduction to Spring
  - Spring framework
  - Refectoring a Hello Word program
  - IoC (Dependency Injection)
- Day 2
  - AOP (Aspect Oriented Programming)
  - Spring JDBC
- Day 3
  - Transaction Management
  - Spring with Hibernate

Ver 1.0 © 2007 Capgemini - All rights reserved

Agenda

- Day 4
  - Spring MVC
  - Spring with Struts
- Day 5
  - A Project

Ver 1.0 © 2007 Capgemini - All rights reserved

Why Spring?

- Spring is an open source application framework.
- Problem with traditional approach,
  - Excessive amount of “plumbing” code.
  - EJB component model is unduly complex.
  - Hard to unit test.
- Lightweight framework uses POJO.
- Noninvasive framework.
- Provides a consistent programming model.
- Promotes pluggability.
- Test driven Development.

Ver 1.0 © 2007 Capgemini - All rights reserved

Why Spring?

- Wiring of component through dependency injection.
- Declarative programming through AOP.
- Conversion of checked exceptions to unchecked.

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

```
public class HelloWorld {
  public static void main(String[] args) {
     System.out.println("Hello World!");
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

1. HelloWorld with command line arguments
2. HelloWorld with decoupling
3. HelloWorld with decoupling through Factory
4. HelloWorld using Spring framework's Dependency Injection

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

HelloWorld: Problems

- The code is not extensible. You have to change code to handle situations below.
  - What if we want to change the message?

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

Support a simple and flexible mechanism for changing the message

```
public class HelloWorldCL {
   public static void main (String cArgs[]) {
       if (cArgs.length > 0) {
             System.out.println(cArgs[0]);
       }
       else {
             System.out.println("Hello World");
       }
   }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

HelloWorldCL: Problems

The code responsible for the rendering message (renderer – the code that does println) is also responsible for obtaining the message

– Changing how the message is obtained means changing the code in the renderer

- The renderer cannot be changed easily

– What if we want to output the message differently, maybe to stderr instead of stdout, or enclosed in HTML tags rather than as plain text?

– Doing so means changing the class that launches the application

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

- Rendering logic should be in a separate component
- Message retrieval logic should be in a separate component

```
public class HelloWorldMsgProvider {
   public String getMessage() {
      return "Hello World";
   }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

- Decouple message rendering logic implementation from the rest of the code
- Message rendering logic uses HelloWorldMsgProvider object given by someone – this is Dependency Injection behavior

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

```
public class StdOutMsgRenderer {
   private HelloWorldMsgProvider msgProvider = null;

   public void setMsgProvider(HelloWorldMsgProvider
                                          msgProvider) {
       this.msgProvider = msgProvider;
   }
   public void render() {
       if (msgProvider == null) {
            throw new RuntimeException("Message
                  Provider is required");
       }
       System.out.println(msgProvider.getMessage());
   }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

```
public class HelloWorldDecoupled {
  public static void main (String cArgs[]) {
     StdOutMsgRenderer renderer = new
          StdOutMsgRenderer();
     HelloWorldMsgProvider msgProvider = new
          HelloWorldMsgProvider();
     renderer.setMsgProvider(msgProvider);
     renderer.render();
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

HelloWorld with decoupling : Problems

Particular *MsgRenderer* implementation and *MsgProvider* implementation are hardcoded in the main code

Solution,

- Let these components implement interfaces and define interdependencies between components and the launcher use these interfaces

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

Decouple message rendering logic

```
public interface MsgProvider {
  public String getMessage();
}
public interface MsgRenderer {
  public void render();
  public MsgProvider getMsgProvider();
  public void setMsgProvider(MsgProvider
  msgProvider);
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

```
public class HelloWorldDecoupled {
  public static void main(String[] args) {
     MessageRenderer mr = new
          SomeMsgRenderer();
     MessageProvider mp = new
          SomeMsgProvider();
     mr.setMsgProvider(mp);
     mr.render();
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

HelloWorld with decoupling : Problems

Using different implementation of either the MsgRenderer or MsgProvider interfaces means a change to the business logic code (launcher in this example)

Solution,

- Create a simple factory class that reads the implementation class names from a properties file and instantiate them on behalf of the application

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

```
public class MsgSupportFactory {
   private static MsgSupportFactory self = new
   MsgSupportFactory();
   private Properties props = new Properties();
   private MsgRenderer renderer = null;
   private MsgProvider provider = null;
   private MsgSupportFactory() {
       props.load(new
             FileInputStream("msf.properties"));
       String rendererClass =
       props.getProperty("renderer.class");
       String providerClass =
            props.getProperty("provider.class");
       renderer = (MessageRenderer)
       Class.forName(rendererClass).newInstance();
       provider = (MessageProvider)
       Class.forName(providerClass).newInstance();
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Refactoring a Hello World Program

```
public class HelloWorldDecoupledWithFactory {
   public static void main(String[] args) {
       MsgRenderer mr =
   MsgSupportFactory.getInstance().
       getMessageRenderer();
       MsgProvider mp =
   MsgSupportFactory.getInstance().
       getMessageProvider();
       mr.setMsgProvider(mp);
       mr.render();
   }
}

# msf.properties
renderer.class=StdOutMsgRenderer
provider.class=HelloWorldMsgProvider
```

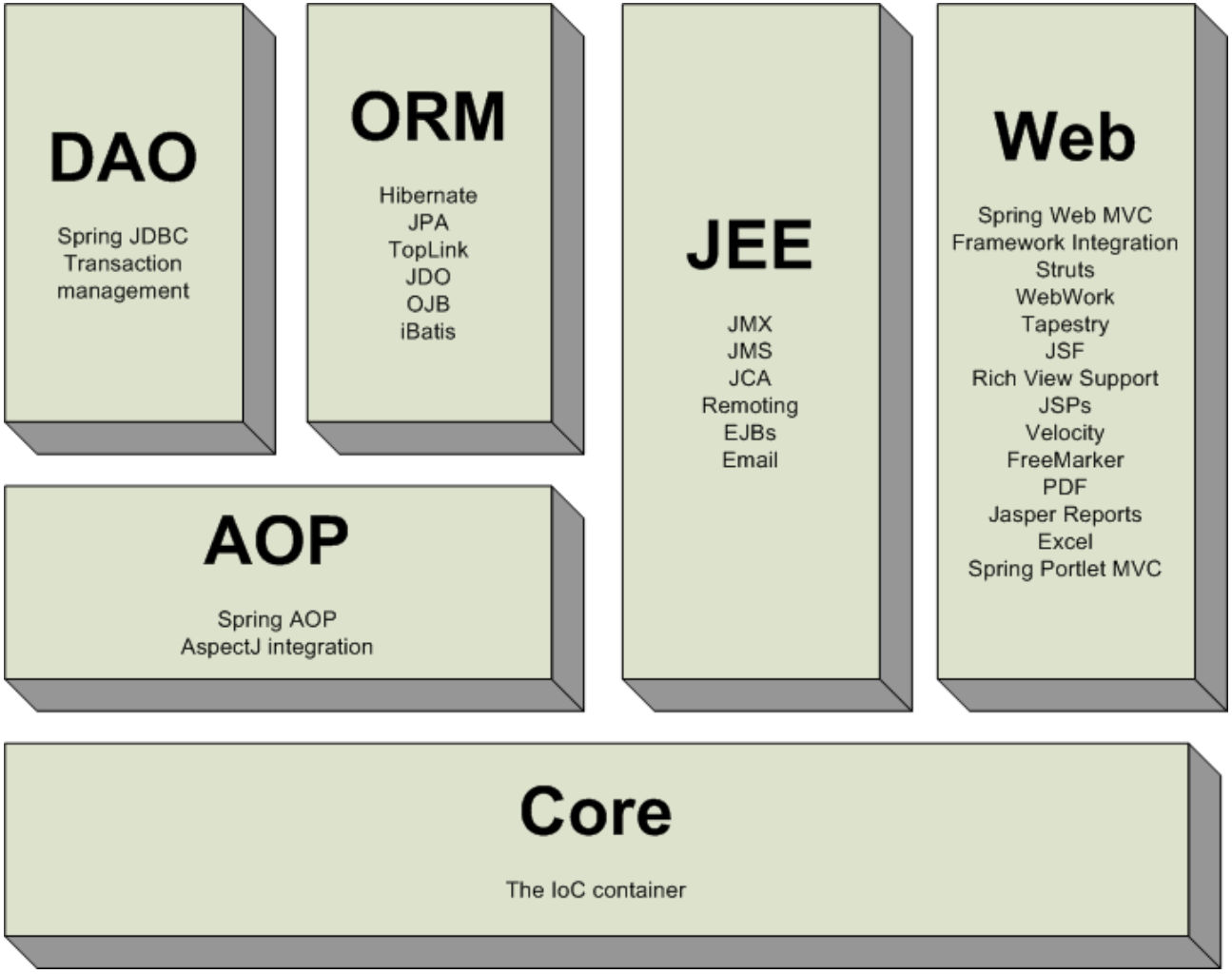
Ver 1.0 © 2007 Capgemini - All rights reserved

HelloWorld with factory : Problems

- We still have to write a lot of glue code ourselves to pieces the application together
  - We have to write *MsgSupportFactory* class
- We still have to provide the implementation of MsgRenderer with an instance of MsgProvider manually

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring architecture overview



Ver 1.0 © 2007 Capgemini - All rights reserved

Core Package

- Core package is the most fundamental part of the framework and provides the IoC and Dependency Injection features
- The basic concept here is the BeanFactory, which provides a sophisticated implementation of the factory pattern which removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic

Ver 1.0 © 2007 Capgemini - All rights reserved

DAO Package

- The DAO package provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes
- The JDBC package provides a way to do programmatic as well as declarative transaction management, not only for classes implementing special interfaces, but for all your POJOs (plain old Java objects)

Ver 1.0 © 2007 Capgemini - All rights reserved

ORM Package

- The ORM package provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- Using the ORM package you can use all those O/R-mappers in combination with all the other features Spring offers, such as the simple declarative transaction management feature mentioned previously

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Package

- Spring's AOP package provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated
- Using source-level metadata functionality you can also incorporate all kinds of behavioral information into your code

Ver 1.0 © 2007 Capgemini - All rights reserved

MVC Package

- Spring's MVC package provides a Model-View-Controller (MVC) implementation for webapplications
- Spring's MVC framework is not just any old implementation; it provides a clean separation between domain model code and web forms, and allows you to use all the other features of the Spring Framework.

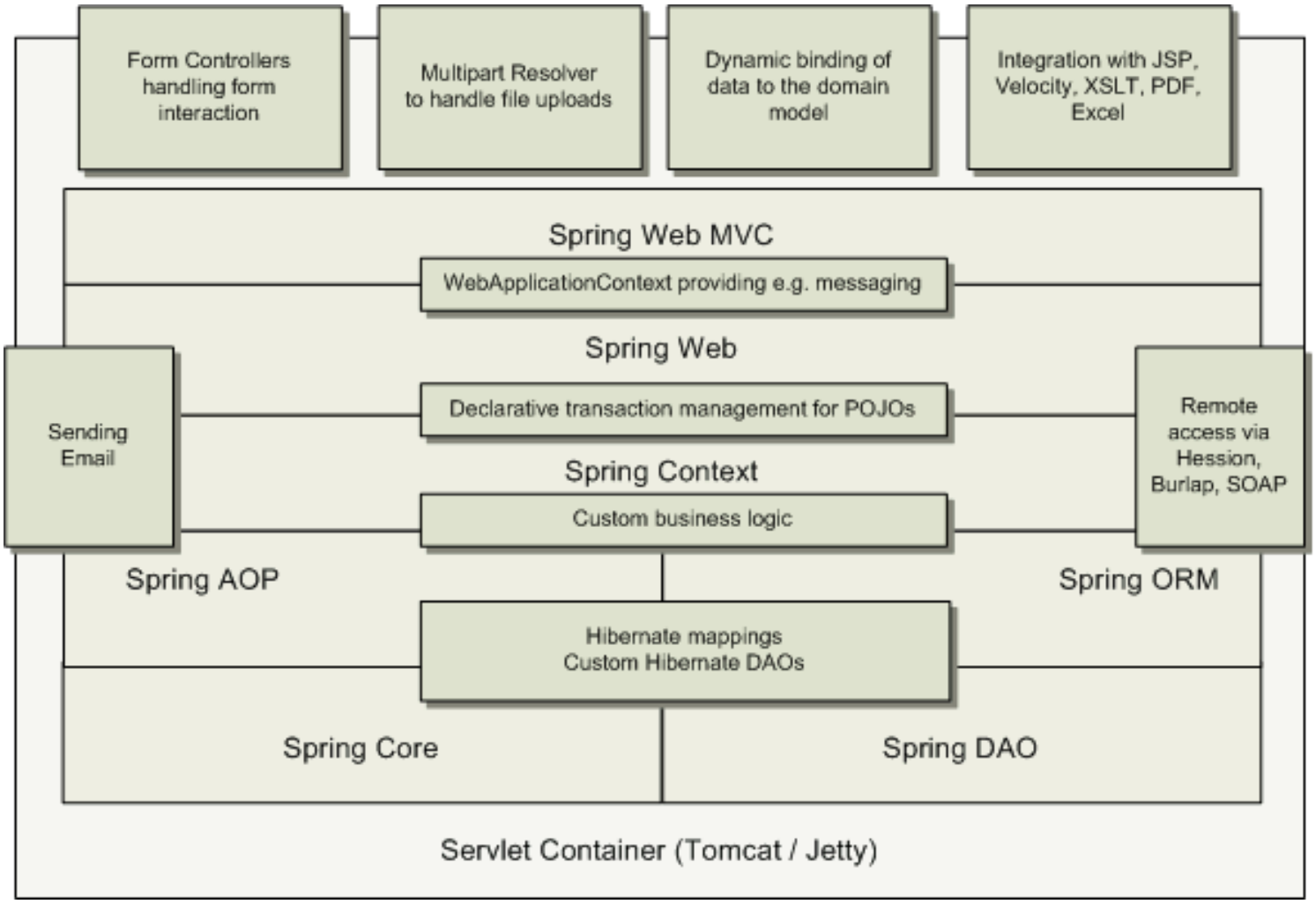
Ver 1.0 © 2007 Capgemini - All rights reserved

Usage Scenario

- You can use Spring in all sorts of scenarios, from applets up to fully-fledged enterprise applications using Spring's transaction management functionality and web framework integration

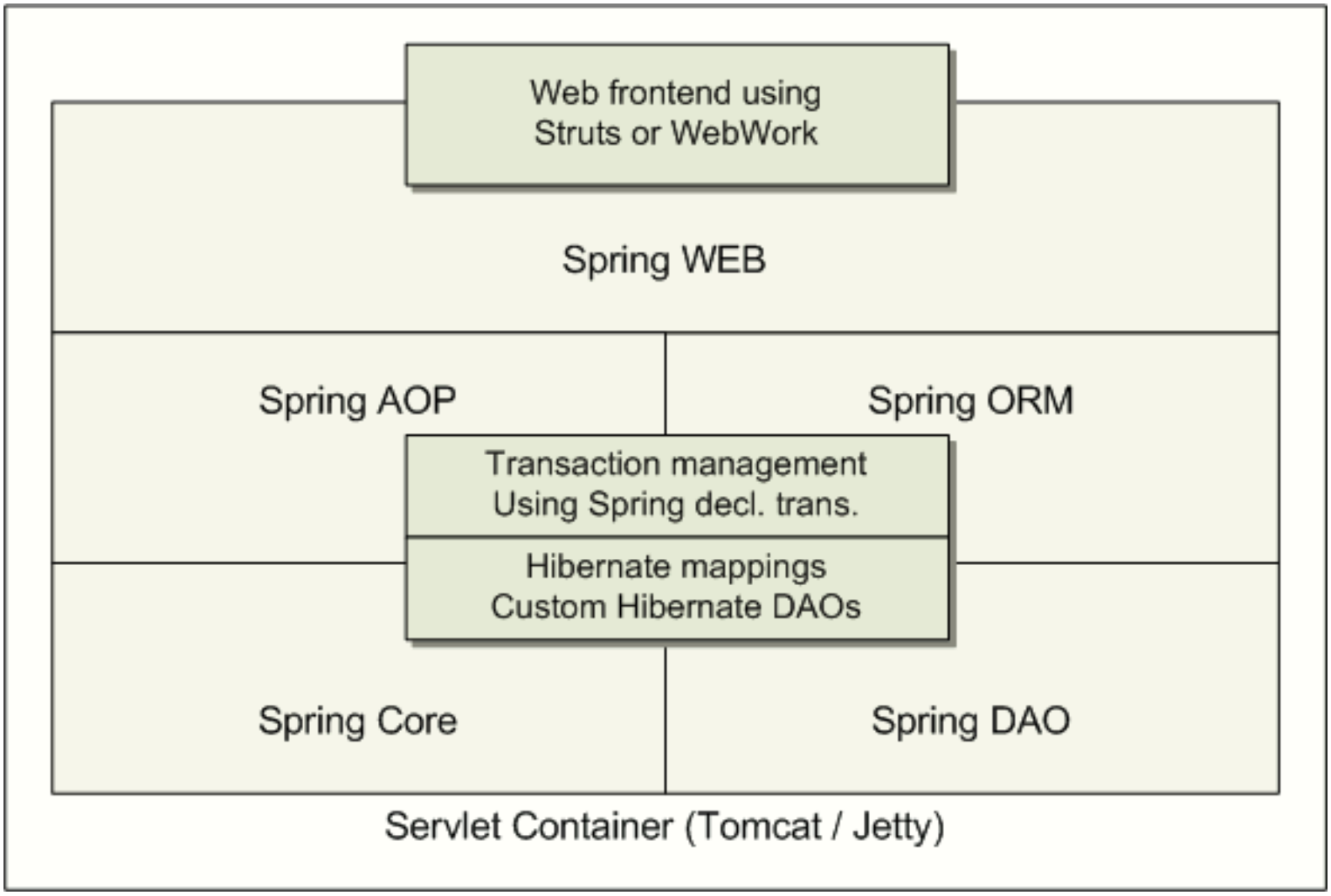
Ver 1.0 © 2007 Capgemini - All rights reserved

Full-fledging Spring Web Application



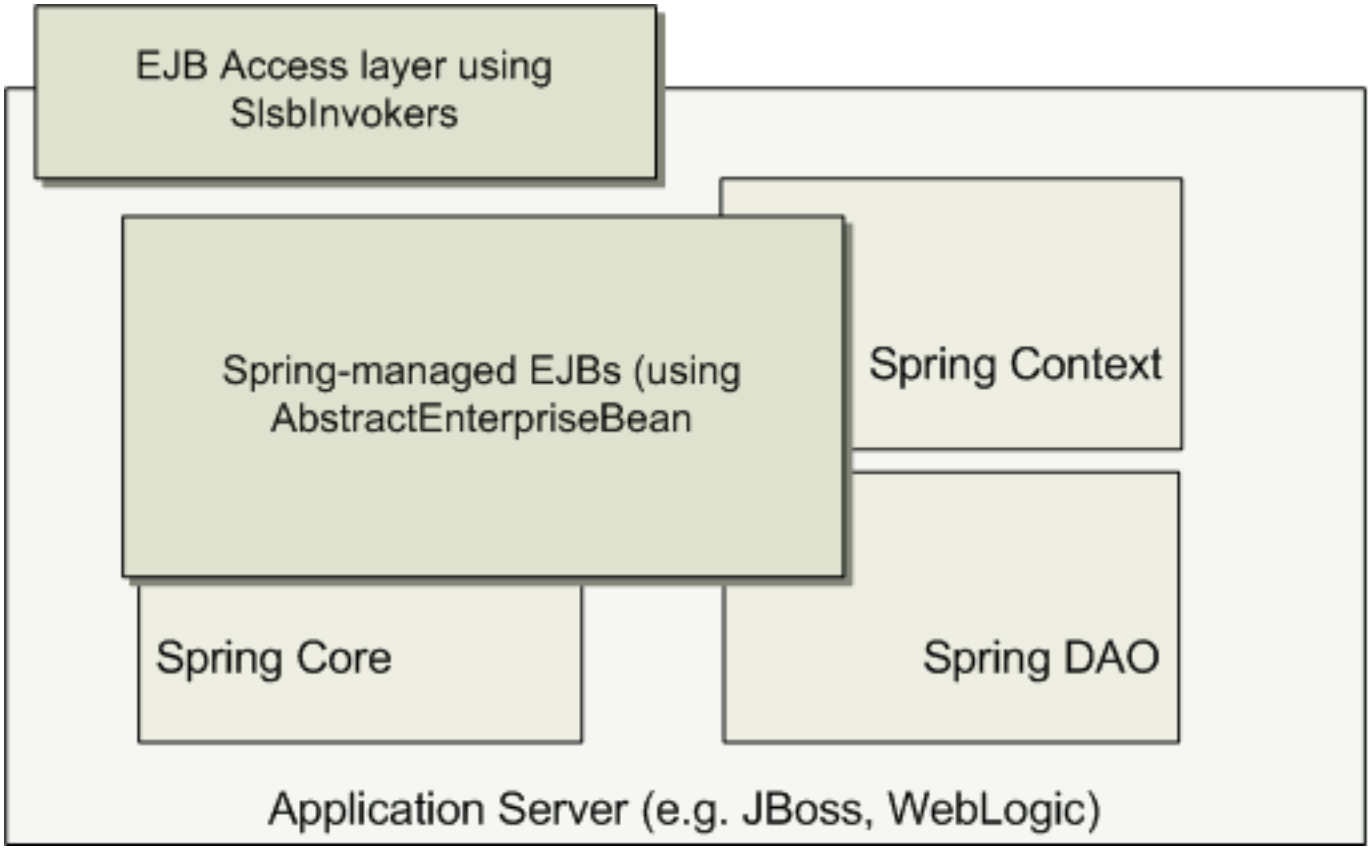
Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Middle-Tier Using 3rd party Web Framework



Ver 1.0 © 2007 Capgemini - All rights reserved

EJBs – Wrapping Existing POJOs



Ver 1.0 © 2007 Capgemini - All rights reserved

Inversion of Control/DI

Ver 1.0 © 2007 Capgemini - All rights reserved

Inversion of Control / Dependency Injection

- A kind of Inversion of Control (IoC)
- "Hollywood Principle“

  - Don't call me, I'll call you

- "Container" resolves (injects) dependencies of components by setting implementation object (push)
- As opposed to component instantiating or Service Locator pattern where component locates implementation (pull)

Ver 1.0 © 2007 Capgemini - All rights reserved

Benefits of Dependency Injection

- Flexible
  - – Avoid adding lookup code in business logic
- Testable
  - – No need to depend on external resources or containers for testing
- Maintainable
  - – Allows reuse in different application environments by changing configuration files instead of code
  - – Promotes a consistent approach across all applications and teams

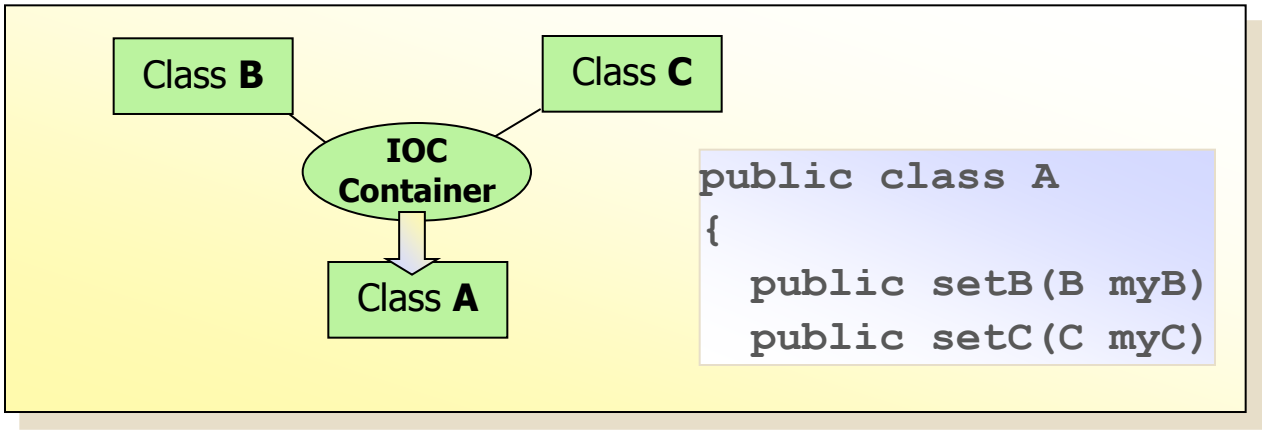
Ver 1.0 © 2007 Capgemini - All rights reserved

Inversion of Control / Dependency Injection

Normal Way

```
public class A
{
  B b = new B()
  C c = new C()
```

Using IoC



Dependency Injection Styles

Two Supported By Spring:

Setter/getter based

Constructor based

Ver 1.0 © 2007 Capgemini - All rights reserved

Two Dependency Injection Variants

- Constructor dependency Injection

  - Dependencies are provided through the constructors of the component

- Setter dependency injection

  - Dependencies are provided through the JavaBean style setter methods of the component

  - More popular than Constructor dependency injection

Ver 1.0 © 2007 Capgemini - All rights reserved

Constructor Dependency Injection

```
public class ConstructorInjection {
  private Dependency dep;
  public ConstructorInjection(Dependency dep)
  {
     this.dep = dep;
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Setter Dependency Injection

```
public class SetterInjection {
   private Dependency dep;
   public void setMyDependency(Dependency dep)
   {
       this.dep = dep;
   }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Container

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Container

- Bean Factory
- Application Context

Ver 1.0 © 2007 Capgemini - All rights reserved

BeanFactory

- Lightweight container that loads bean definitions and manages your beans.
- Knows how to serve and manage a singleton or prototype defined bean
- Responsible for managing bean lifecycle.
- Injects dependencies into defined beans when served
- Avoids the use of singletons and factories

```
BeanFactory factory =
new XmlBeanFactory(new
   FileSystemResource("c:/beans.xml"))
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Understanding Beans, Bean Factory and Application Context

Application Context

Application context provides all the features that BeanFactory provides additionally it

- Application contexts provide a means for resolving text messages, including support for internationalization (I18N) of those messages.
- Application contexts provide a generic way to load file resources, such as images.
- Application contexts can publish events to beans that are registered as listeners.

Several ways to configure a context:

- **ClassPathXMLApplicationContext -** Loads beans configuration from classpath.

  ApplicationContext context = new ClassPathXmlApplicationContext("foo.xml");

- **FileSystemXmlApplicationContext -** Load from the file system (relative/absolute)

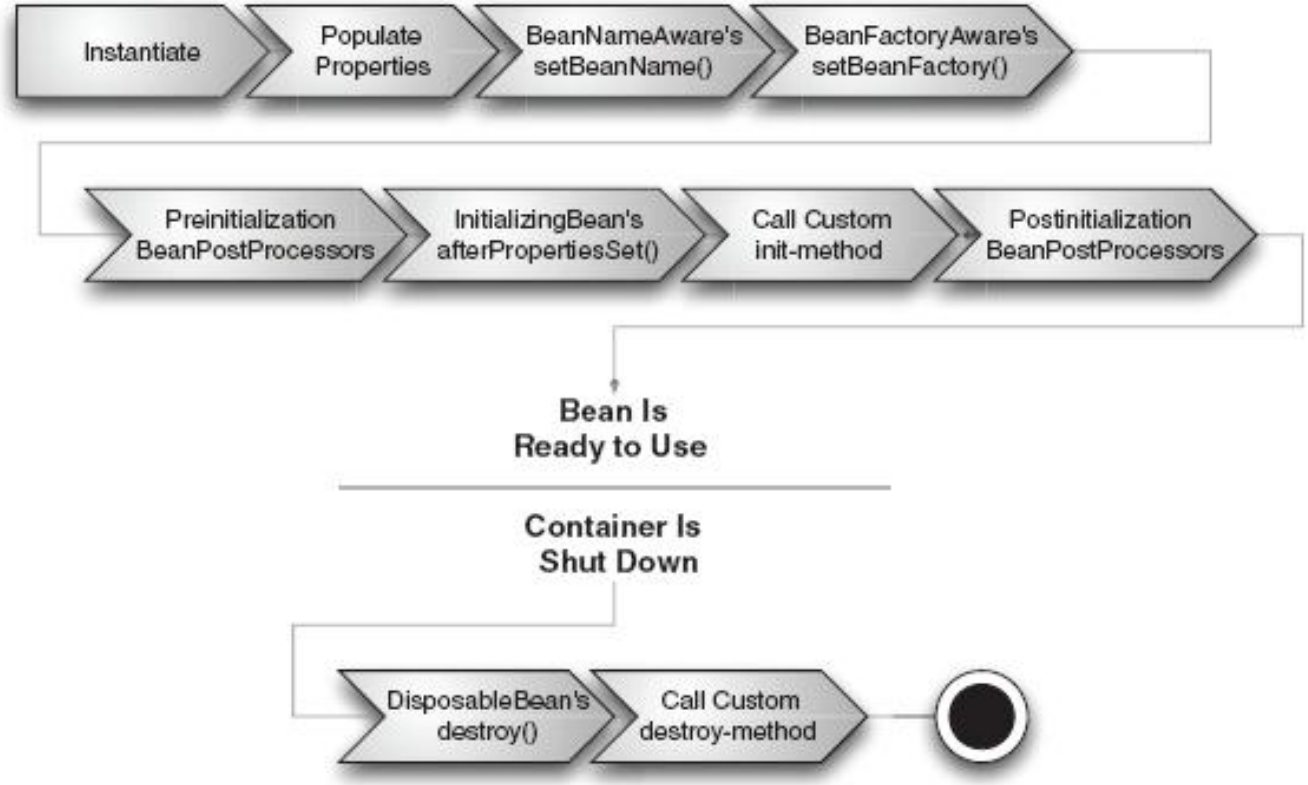
  ApplicationContext context = new FileSystemXmlApplicationContext("c:/foo.xml");

XMLWebApplicationContext– Configuration for a web application under ServletContext

Ver 1.0 © 2007 Capgemini - All rights reserved

A bean's Life

A Bean's Life



Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Beans

- Following Bean properties can be set via dependency injection.
  - References to other managed beans.
  - Strings
  - Primitive types
  - Collections (list, set, map, props)

Ver 1.0 © 2007 Capgemini - All rights reserved

Understanding Beans, BeanFactory and Application Context

Spring Beans

- Spring bean is plain old java object configured in a xml file as

```
<bean id="serviceOne" class="com.spring.beans.ConsumerServiceOne">
        <property name="name">
                <value>"Naren"</value>
        </property>
</bean>
```

- No standard definition of bean
- Bean examples – DAO, DataSource, Transaction Manager, Persistence Managers, Service objects, etc
- Bean behaviors include:
  - Singleton or prototype
  - Autowiring
- Initialization and destruction methods
  - init-method
  - destroy-method
- Beans can be configured to have property values set.
- Can read simple values, collections, maps, references to other beans, etc.

Ver 1.0 © 2007 Capgemini - All rights reserved

Reading XML Configuration File via XmlBeanFactory class

```
public class XmlConfigWithBeanFactory {
  public static void main(String[] args) {
        XmlBeanFactory factory = new XmlBeanFactory(new
        FileSystemResource("beans.xml"));
        SomeBeanInterface b = (SomeBeanInterface)
        factory.getBean("nameOftheBean");
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Configuration File

- Each bean is defined using *<bean>* tag under the root of the <beans> tag
- The *id* attribute is used to give the bean its default name
- The *class* attribute specifies the type of the bean

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Configuration File Example: Setter DI

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN“
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id=“renderer”  class="StandardOutMessageRenderer">
        <property name="messageProvider">
            <ref local="provider"/>
        </property>
    </bean>
    <bean id="provider" class="HelloWorldMessageProvider"/>
</beans>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Configuration File Example: Setter DI

Definition of beans:

```
<bean id="exampleBean" class="examples.ExampleBean">
        <property name="beanOne">
                <ref bean="anotherExampleBean"/>
        </property>
        <property name="beanTwo" ref="yetAnotherBean"/>
        <property name="integerProperty" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

Java objects with setters:

```
public class ExampleBean {
  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public void setBeanOne(AnotherBean beanOne) {
    this.beanOne = beanOne;
  }
  public void setBeanTwo(YetAnotherBean beanTwo) {
    this.beanTwo = beanTwo;
  }
  public void setIntegerProperty(int i) {
    this.i = i;
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Configuration File Example: Constructor DI

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN“
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="renderer" class="StandardOutMessageRenderer">
        <property name="messageProvider">
            <ref local="provider"/>
        </property>
    </bean>
    <bean id="provider" class="ConfigurableMessageProvider">
        <constructor-arg>
            <value>This is a configurable message</value>
        </constructor-arg>
    </bean>
</beans>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Configuration File Example: Constructor DI

Definition of beans:

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- constructor injection using the nested <ref/> element -->
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
  <!-- constructor injection using the neater 'ref' attribute -->
  <constructor-arg ref="yetAnotherBean"/>
  <constructor-arg type="int" value="1"/>
</bean>

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

Java object with a constructor:

```
public class ExampleBean {
  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public ExampleBean(
    AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
      this.beanOne = anotherBean;
      this.beanTwo = yetAnotherBean;
      this.i = i;
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Lab Exercise

Ver 1.0 © 2007 Capgemini - All rights reserved

IoC in Spring –Bean Reference

Reference to any bean in the same container:

```
<bean id="myService" class="examples.MyService">
  <property name="dao">
    <ref bean="myDAO"/>
  </property>
</bead>
```

Reference to a bean in the same XML file:

```
<bean id="myService" class="examples.MyService">
  <property name="dao">
    <ref local="myLocalDAO"/>
  </property>
</bead>
```

Inner bean:

```
<bean id="myService" class="examples.MyService">
  <property name="dao">
    <bean class="examples.MyDAO">
      <property name="name" value="apple"/>
      <property name="kind" value="big"/>
    </bean>
  </property>
</bead>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Wiring Collections

Collection element	Useful for...
`<list>`	Wiring a list of values, allowing duplicates.
`<set>`	Wiring a set of values, ensuring no duplicates
`<map>`	Wiring a collection of name-value pairs where name and value can be of any type
`<props>`	Wiring a collection of name-value pairs where the name and value are both `Strings`

Ver 1.0 © 2007 Capgemini - All rights reserved

Collection Example

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@somecompany.org</prop>
        <prop key="support">support@somecompany.org</prop>
        <prop key="development">development@somecompany.org</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
        <value>a list element followed by a reference</value>
        <ref bean="myDataSource" />
    </list>
  </property>
  <!-- results in a setSomeMap(java.util.Map) call -->
  <property name="someMap">
    <map>
        <entry>
            <key>
                <value>yup an entry</value>
            </key>
            <value>just some string</value>
        </entry>
        <entry>
            <key>
                <value>yup a ref</value>
            </key>
            <ref bean="myDataSource" />
        </entry>
    </map>
  </property>
  <!-- results in a setSomeSet(java.util.Set) call -->
  <property name="someSet">
    <set>
        <value>just some string</value>
        <ref bean="myDataSource" />
    </set>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

IoC in Spring – Bean definition inheritance

Definition of a bean that inherit from an abstract bean definition:

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass"
      class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBeanWithoutClass"
      init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit the value of 1 from the parent bean
       definition-->
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Creating beans from factory methods

```
Pacakage examples.ExampleBean;

Class ExampleBean{

        static ExampleBean exb = new ExampleBean();

        private ExampleBean(){}

}

Public static ExampleBean createInstance(){

        return exb;

}

<bean id="exampleBean‟ class="examples.ExampleBean"
        factory-method="createInstance"/>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Scopes

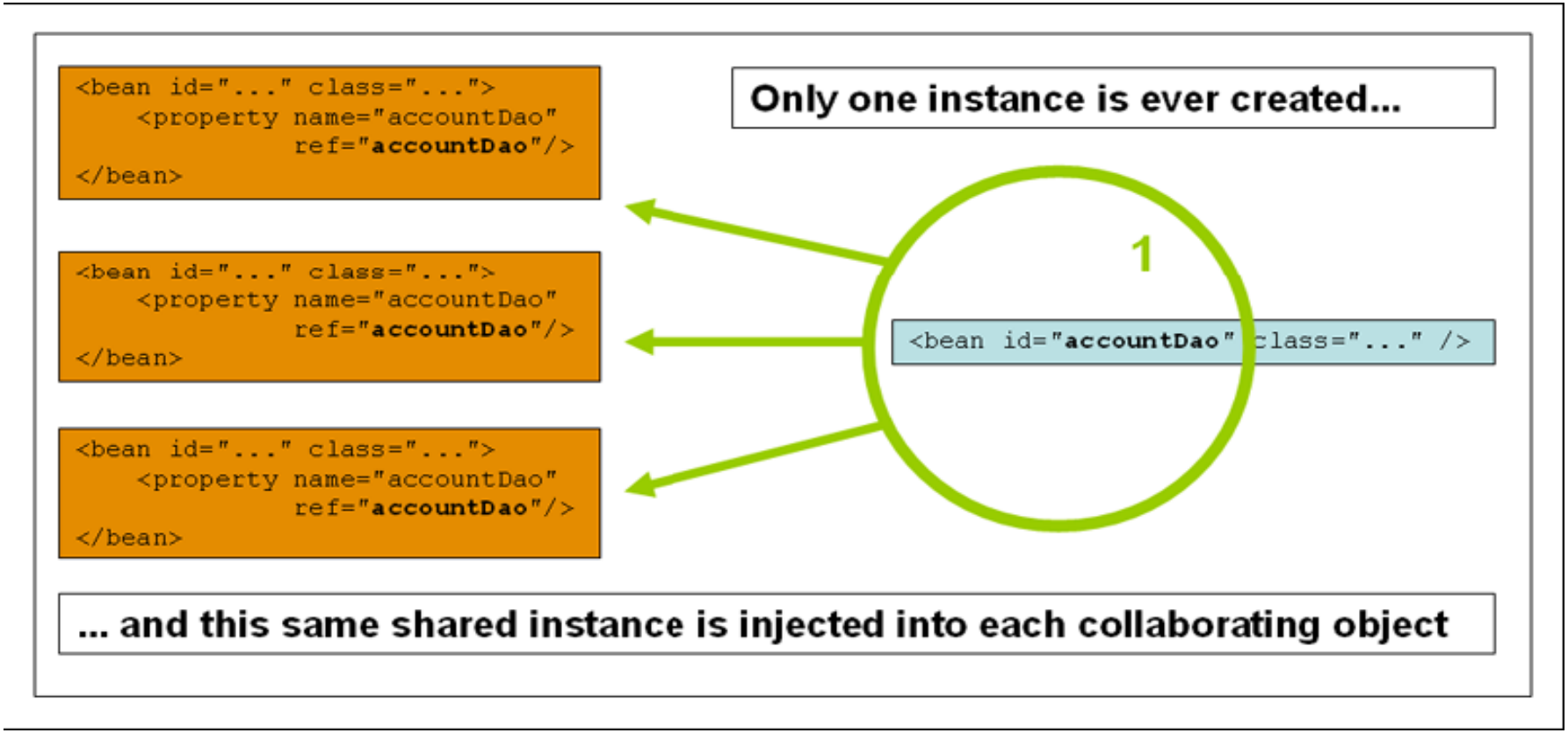
Bean Scope

Scope	What it does
`singleton`	Scopes the bean definition to a single instance per Spring container (default).
`prototype`	Allows a bean to be instantiated any number of times (once per use).
`request`	Scopes a bean definition to an HTTP request. Only valid when used with a web-capable Spring context (such as with Spring MVC).
`session`	Scopes a bean definition to an HTTP session. Only valid when used with a web-capable Spring context (such as with Spring MVC).
`global-session`	Scopes a bean definition to a global HTTP session. Only valid when used in a portlet context.

Ver 1.0 © 2007 Capgemini - All rights reserved

IOC in Spring – Bean Scopes

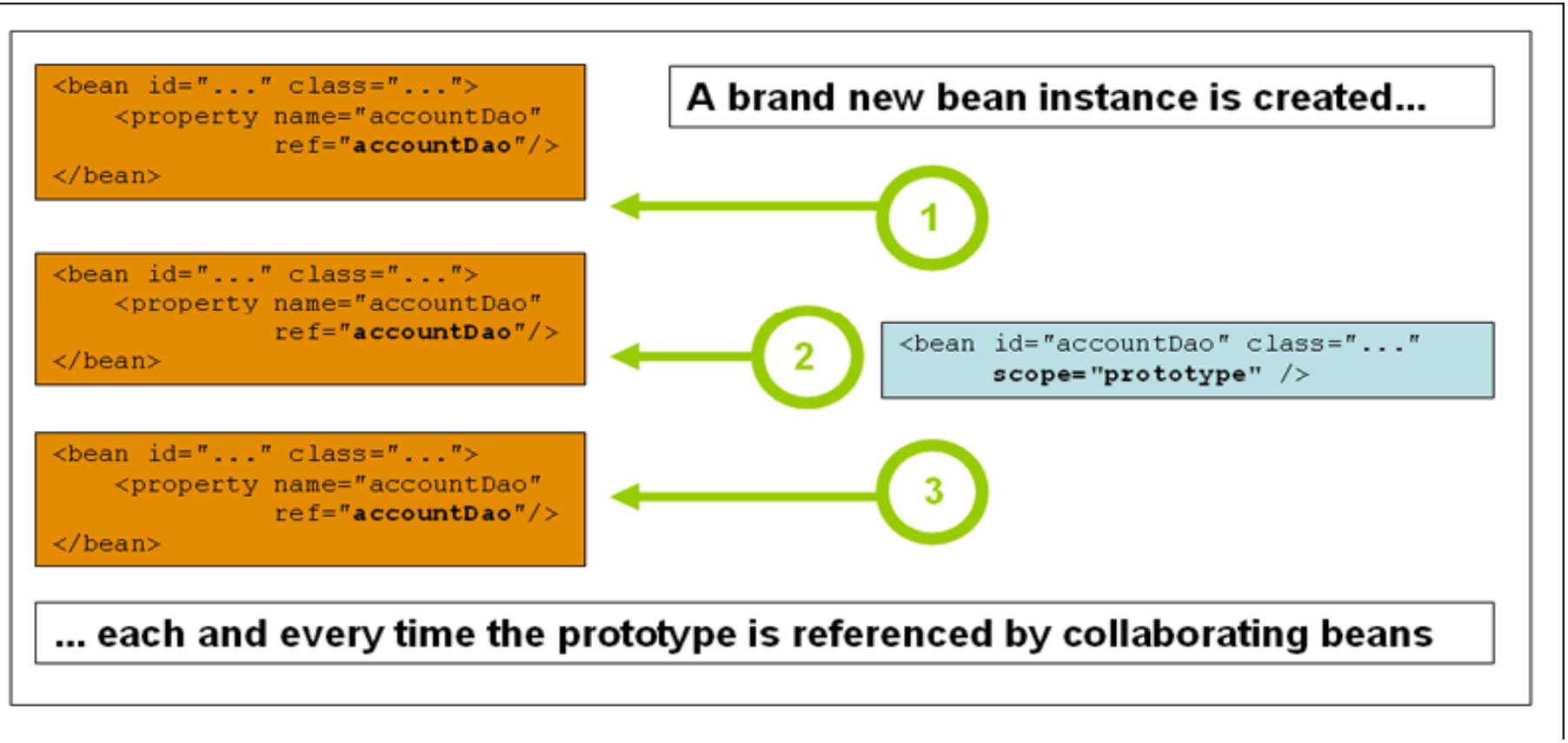
‘Singleton’ bean scope:



Ver 1.0 © 2007 Capgemini - All rights reserved

IoC in Spring – Bean Scopes

'prototype' bean scope:



Ver 1.0 © 2007 Capgemini - All rights reserved

Spring IoC - Autowiring

Four types of Autowiring:

Spring provides four flavors of autowiring:

- **byName**—Attempts to find a bean in the container whose name (or ID) is the same as the name of the property being wired. If a matching bean is not found, the property will remain unwired.
- **byType**—Attempts to find a single bean in the container whose type matches the type of the property being wired. If no matching bean is found, the property will not be wired. If more than one bean matches, an org.springframework.beans.factory.UnsatisfiedDependencyException will be thrown.
- **constructor**—Tries to match up one or more beans in the container with the parameters of one of the constructors of the bean being wired. In the event of ambiguous beans or ambiguous constructors, an org.springframework. beans.factory.UnsatisfiedDependencyException will be thrown.
- **autodetect**—Attempts to autowire by constructor first and then using byType. Ambiguity is handled the same way as with constructor and byType wiring.

Ver 1.0 © 2007 Capgemini - All rights reserved

IoC in Spring – Auto wire By Name

Definition of beans with auto wire by name:

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>

  ...

</bean>

<bead id="myFriend" class="examples.MyFriendImpl">
  <property name="name" value="apple"/>
</bean>


<bead id="myService" class="examples.MyService" autowire="byName">
</bean>
```

Java object with setters:

```
public class MyService {

  private DataSource dataSource;
  private MyFriend myFriend;

  public void setDataSource(dataSource) {
    this.dataSource = dataSource;
  }

  public void setMyFriend(myFriend) {
    this.myFriend = myFriend;
  }
  ...
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

IoC in Spring – Auto wire by Type

Definition of beans with auto wire by type:

```
<bean id="oneDataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>

  ...

</bean>

<bead id="oneFriend" class="examples.MyFriendImpl">
  <property name="name" value="apple"/>
</bean>


<bead id="myService" class="examples.MyService" autowire="byType">
</bean>
```

Java object with setters:

```
public class MyService {

  private DataSource dataSource;
  private MyFriend myFriend;

  public void setDataSource(dataSource) {
    this.dataSource = dataSource;
  }

  public void setMyFriend(myFriend) {
    this.myFriend = myFriend;
  }
  ...
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Default initialization & destroy methods

Default initialization & destroy methods

```
<bead id="blogServive" class="examples.DefaultBlogService " init-
method="init" destroy-method="destroy">
</bean>
```

```
public class DefaultBlogService implements BlogService {
private BlogDao blogDao;
public void setBlogDao(BlogDao blogDao) {
this.blogDao = blogDao;}
// this is (unsurprisingly) the initialization callback method
public void init() {
System.out.println("inside init method");
} public void destroy() {
System.out.println("inside init method");
}}
```

```
<beans default-init-method="init">
<bean id="blogService" class="com.foo.DefaultBlogService">
<property name="blogDao" ref="blogDao" /></bean></beans>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Lab Exercise

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP
Aspect Oriented Programming

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Overview

```
void transfer(Account fromAccount, Account
  toAccount, int amount) {

if (fromAccount.getBalance() < amount) {
    throw new InsufficientFundsException();
}
  fromAccount.withdraw(amount);
  toAccount.deposit(amount);

}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Overview

Money transfer method with cross cutting concerns:

```
void transfer(Account fromAccount, Account toAccount, int amount) {
    if (!getCurrentUser().canPerform(OP_TRANSFER)) {
          throw new SecurityException();
    }
    if (amount < 0) {
          throw new NegativeTransferException();
    }
    if (fromAccount.getBalance() < amount) {
          throw new InsufficientFundsException();
    }
    Transaction tx = database.newTransaction();
    try {
          fromAccount.withdraw(amount);
          toAccount.deposit(amount);
          tx.commit();
          systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
    } catch(Exception e) {
          tx.rollback();
    }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Overview

- Cross-cutting concerns
  - Functionality whose implementation spans multiple modules
  - Examples of cross-cutting concerns
    - Logging
    - Transaction Management
    - Security
    - Auditing
    - Locking
    - Event Handling
- AOP a programming methodology to help with cross-cutting concerns.
- Aspects can be added or removed as needed without changing your code

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP overview

Many AOP implementations:

- Java:
  - The Spring Framework
  - AspectJ
  - JBoss AOP
  - Jakarta Hivemind
  - Many more…
- .NET:
  - The Spring.NET Framework
  - Aspect.NET
  - ACA.NET
- Many more for all other languages

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP in Spring

Join point

- An identifiable point in the execution of a program.

Advice

- A join point *always* represents a method execution.

Pointcut

- Spring uses the AspectJ pointcut language by default

Aspect

- Aspects are implemented using regular classes

Weaving

- Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime

Target object

- Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object

Introduction

- Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any proxied object

AOP Proxy

- Object created by the AOP framework, including advice. In Spring, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Concepts: Join point

- Well-defined point during the execution of your application
- You can insert additional logic at Joinpoint's
- Examples of Jointpoint's
  - Method invocation
  - Class initialization
  - Object initialization

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Concepts: Advice

- The code that is executed at a particular joinpoint
- Types of Advice
  - before advice, which executes before joinpoint
  - after advice, which executes after joinpoint
  - around advice, which executes around joinpoint

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Concepts: Pointcuts

- A collection of joinpoints that you use to define when advice should be executed
- By creating pointcuts, you gain fine-grained control over how you apply advice to the components
- Example
  - A typical joinpoint is a method invocation.
  - A typical pointcut is a collection of all method invocations in a particular class
- Pointcuts can be composed in complex relationships to further constrain when advice is executed

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Concepts: Aspect

- An aspect is the combination of advice and pointcuts

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Concepts: Weaving

- Process of actually inserting aspects into the application code at the appropriate point
- Types of Weaving
  - Compile time weaving
  - Runtime weaving

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Concepts: Target

- An object whose execution flow is modified by some AOP process
- They are sometimes called advised object

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP Concepts: Introduction

- Process by which you can modify the structure of an object by introducing additional methods or fields to it
- You use the Introduction to make any object implement a specific interface without needing the object's class to implement that interface explicitly

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP in Spring

Spring AOP has built in aspects such as providing transaction management and performance monitoring

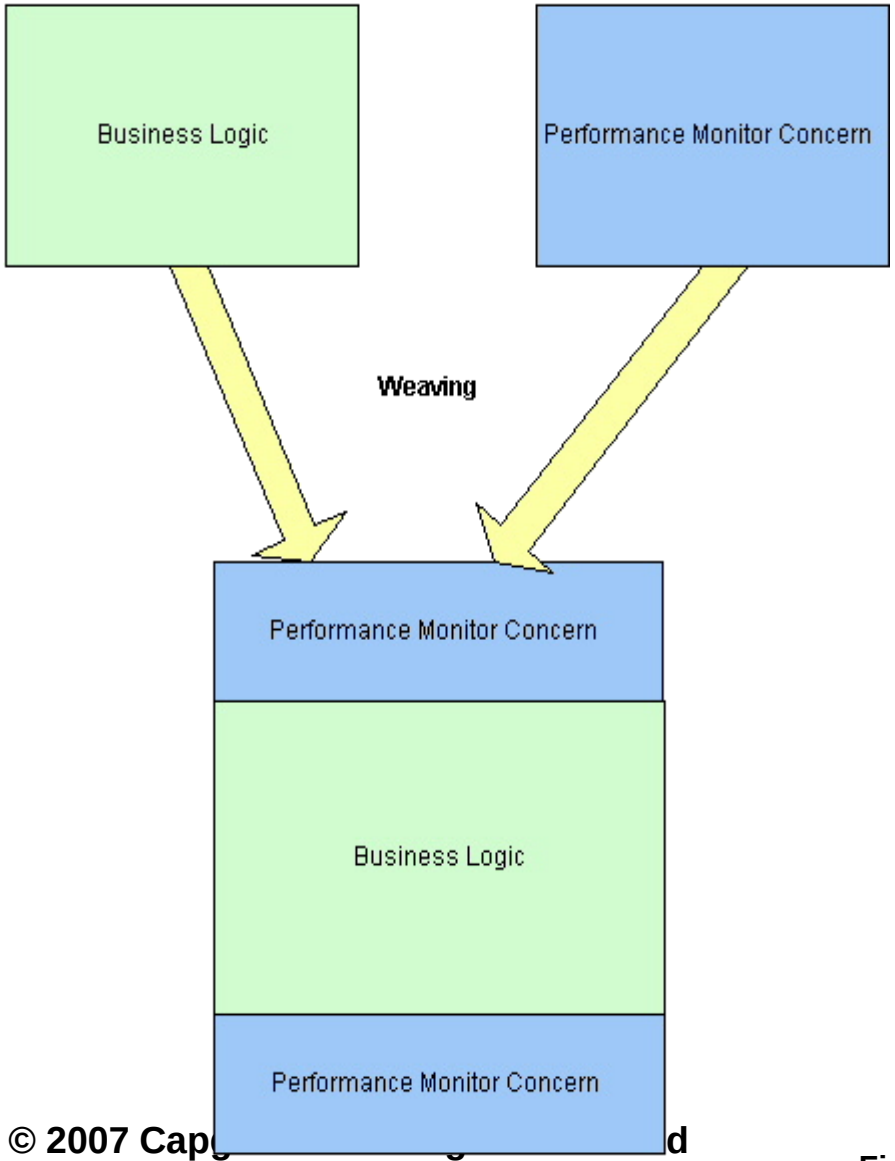
Two options:

- **Spring AOP**
- **AspectJ**

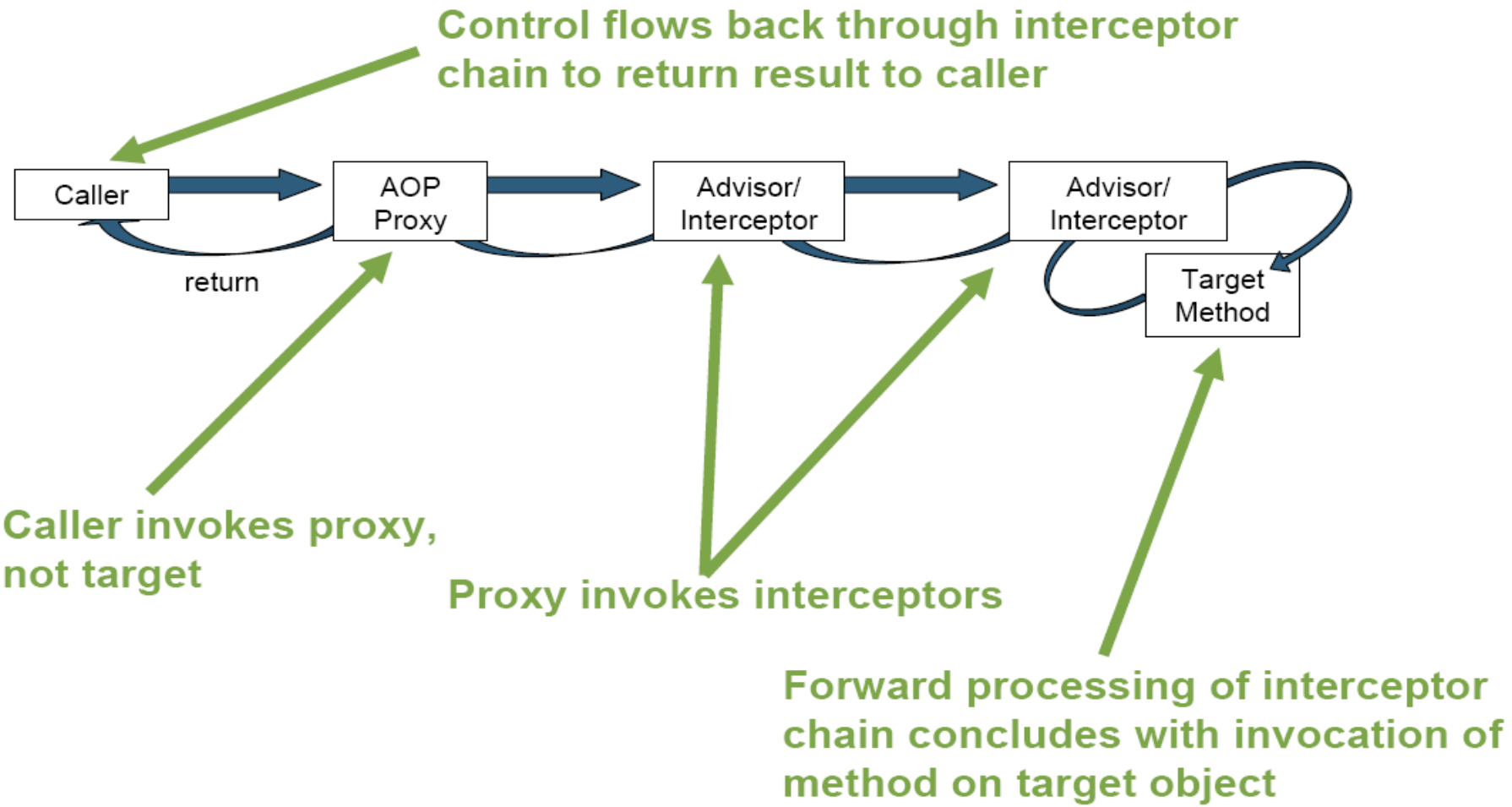
IoC + AOP is a great combination that is non-invasive

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP in Spring



AOP Control Flow



Ver 1.0 © 2007 Capgemini - All rights reserved

Type of Advice

Types of advice:

Before advice: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

```
MethodBeforeAdvice
```

After returning advice: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

```
AfterReturningAdvice
```

After throwing advice: Advice to be executed if a method exits by throwing an exception.

```
ThrowsAdvice
```

Around advice: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

```
MethodInterceptor
```

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP in Spring

```
public interface IBusinessLogic {
   public void foo();
   public void foo(int i);
}
public class BusinessLogic implements IBusinessLogic {
   public void foo() {
        System.out.println("Inside BusinessLogic.foo()");
   }

   public void foo(int i) {
        System.out.println("Inside BusinessLogic.foo() " +
   i);
   }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP in Spring

```
public class AroundAdvice implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
          System.out.println("Hello world! (by " + this.getClass().getName() +
    ")");
          invocation.getArguments()[0] = new Integer(20);
          invocation.proceed();
          System.out.println("Goodbye! (by " + this.getClass().getName() +
    ")");
          return null;
    }
}
public class MainApplication {
    public static void main(String [] args) {
          ApplicationContext ctx = new
    ClassPathXmlApplicationContext("com\\oj\\aop\\springconfig.xml");
          IBusinessLogic testObject = (IBusinessLogic)
    ctx.getBean("businesslogicbean");
          testObject.foo(3);
    }
}
```

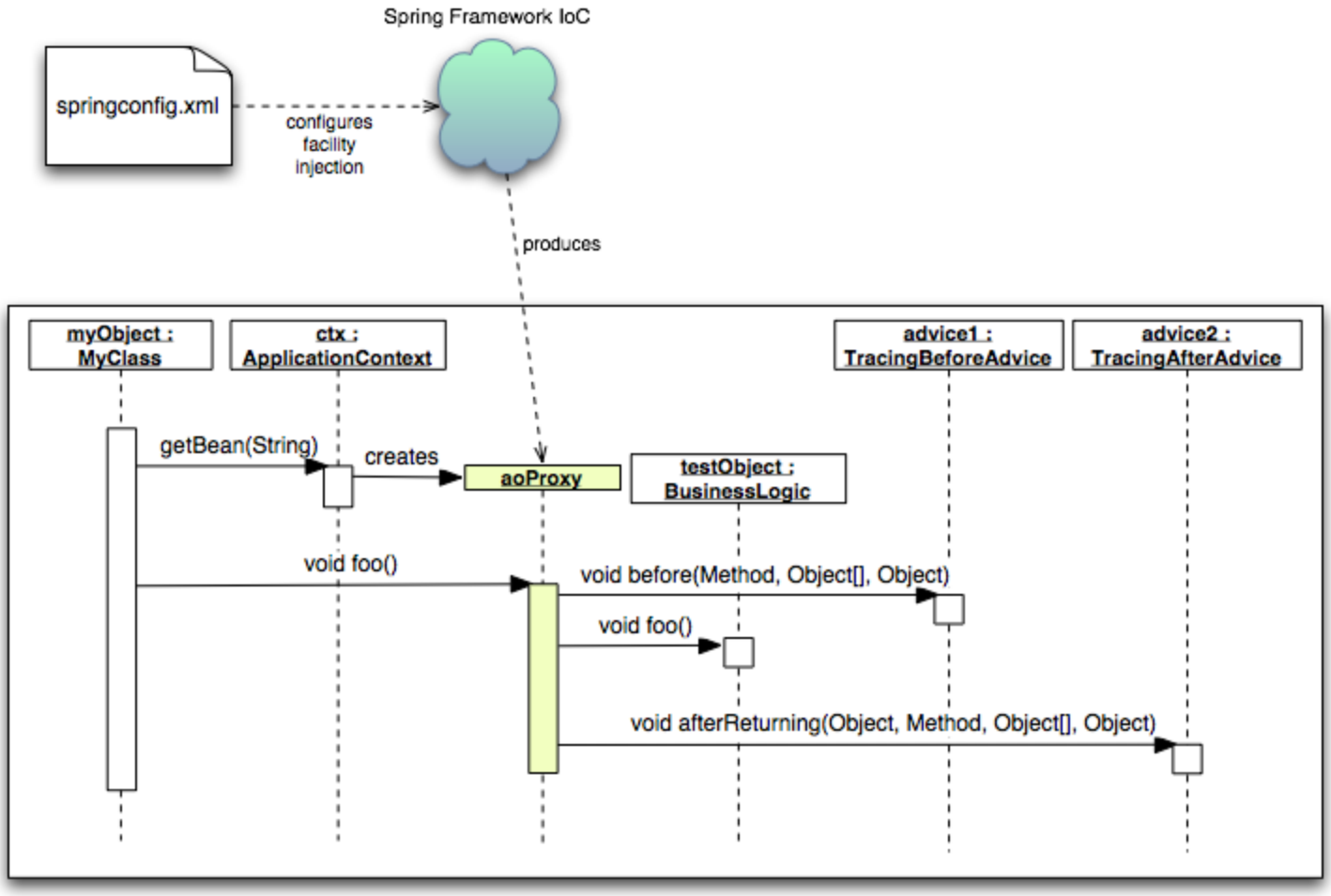
Ver 1.0 © 2007 Capgemini - All rights reserved

AOP in Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<bean id="businesslogicbean" class="org.springframework.aop.framework.ProxyFactoryBean">
		<property name="proxyInterfaces">
			<value>com.oj.aop.IBusinessLogic</value>
		</property>
		<property name="target">
			<ref local="beanTarget"/>
		</property>
		<property name="interceptorNames">
			<list>
				<value>theAroundAdvisor</value>
			</list>
		</property>
	</bean>
	<!-- Bean Classes -->
	<bean id="beanTarget" class="com.oj.aop.BusinessLogic"/>
	 <!-- Advisor pointcut definition for around advice -->
	<bean id="theAroundAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
		<property name="advice">
			<ref local="theAroundAdvice"/>
		</property>
		<property name="pattern">
			<value>.*</value>
		</property>
	</bean>
	<!-- Advice classes -->
	<bean id="theAroundAdvice" class="com.oj.aop.AroundAdvice"/>
</beans>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP in Spring



Ver 1.0 © 2007 Capgemini - All rights reserved

AOP

- the execution of any public method:

```
execution(public * *(..))
```

- the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

- the execution of any method defined by the `AccountService` interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

- the execution of any method defined in the service package:

```
execution(* com.xyz.service.*.*(..))
```

- the execution of any method defined in the service package or a sub-package:

Ver 1.0 © 2007 Capgemini - All rights reserved

AOP

```
execution(* com.xyz.service..*.*(..))
```

- any join point (method execution only in Spring AOP) within the service package:

```
within(com.xyz.service.*)
```

- any join point (method execution only in Spring AOP) within the service package or a sub-package:

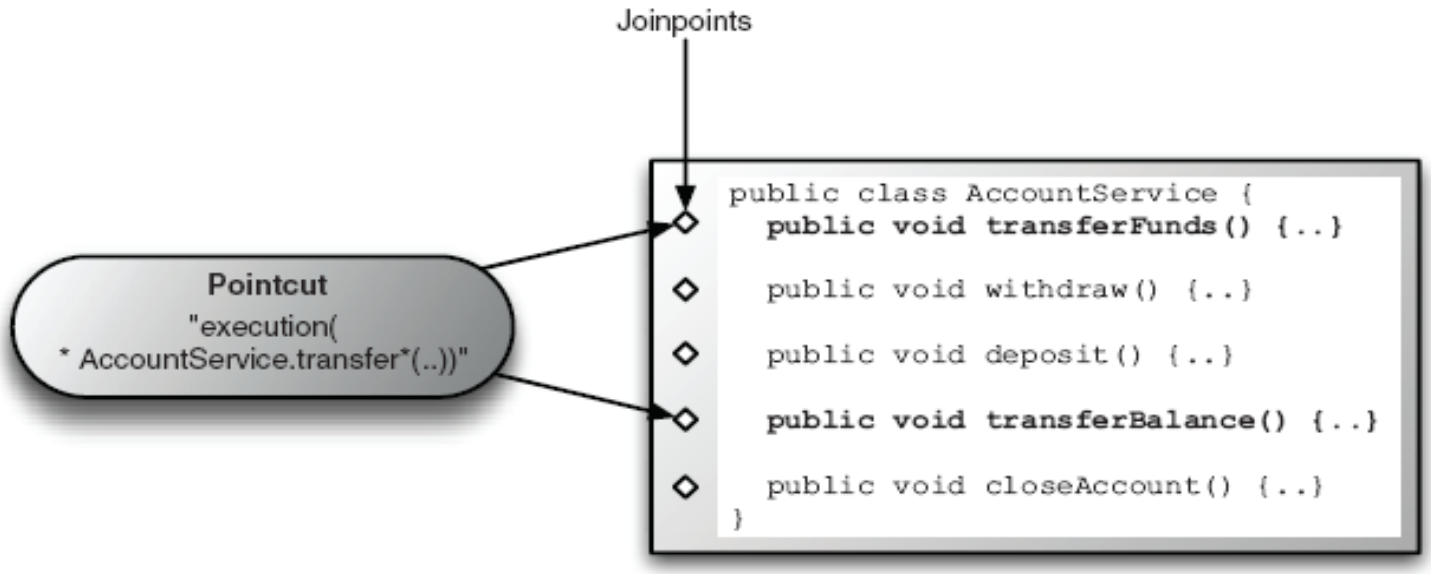
```
within(com.xyz.service..*)
```

- any join point (method execution only in Spring AOP) where the proxy implements the `AccountService` interface:

```
this(com.xyz.service.AccountService)
```

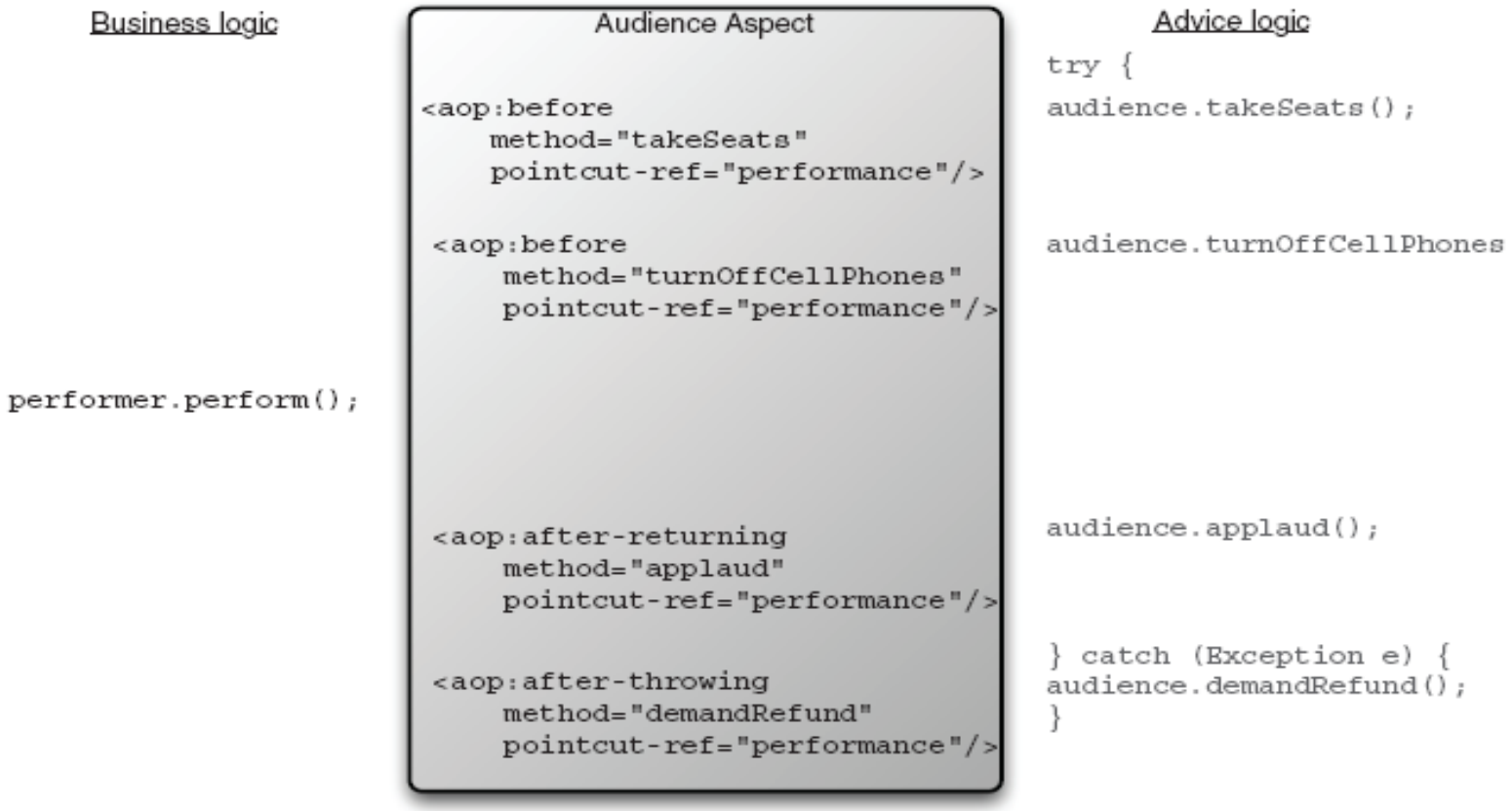
Ver 1.0 © 2007 Capgemini - All rights reserved

AOP



Ver 1.0 © 2007 Capgemini - All rights reserved

AOP



Ver 1.0 © 2007 Capgemini - All rights reserved

Advisors

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

  <aop:advisor
      pointcut-ref="businessService"
      advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Configuration through XML: Defining Advisor

```
<bean id="example" class="com.sample.aop.ExampleImpl"/>
<bean id="someInterceptor"
   class="com.sample.aop.SomeInterceptor"/>
<bean id="someAdvisor"
   class="o.s.apo.support.DefaultPointcutAdvisor">
    <property name="advice">
        <bean class="com.sample.aop.SomeInterceptor"/>
    </property>
    <property name="pointcut">
       <bean
       class="o.s.aop.support.JdkRegexpMethodPointcut"/>
              <property name="pattern" value=".*test.*"/>
        </bean>
    </property>
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Configuration through XML: Defining Advisor

```
<bean id="exampleProxy"
   class="o.s.aop.framework.ProxyFactoryBean">
    <property name="target" ref="example"/>
    <property name="interceptorNames">
       <list>
          <value>someAdvisor</value>
       </list>
    </property>
    <property name="proxyInterfaces">
       <list>
          <value>com.sample.aop.Example</value>
       </list>
    </property>
</bean>
```

Target

Advice/Advisor

Proxy interface (optional)

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Client

```
package com.sample.aop;
...

public class Main {
  public static void main(String[] args) {
    ApplicationContext context
      = new ClassPathXmlApplicationContext("beans.xml");
    Example example = (Example)
   context.getBean(“exampleProxy");
    example.test();
    example.testAbc();
  }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Lab Exercise

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring JDBC Support

Ver 1.0 © 2007 Capgemini - All rights reserved

Vanilla Jdbc coding

1. Define connection parameters
2. Open the connection
3. *Specify the statement*
4. Prepare and execute the statement
5. Set up the loop to iterate through the results (if any)
6. *Do the work for each iteration*
7. Process any exception
8. Handle transactions
9. Close the connection

Ver 1.0 © 2007 Capgemini - All rights reserved

JDBC - Coding

```
public class JdbcDao {
public int getBeerCount() {
            Connection conn = null;
            Statement stmt = null;
            ResultSet rs = null;
            int count = 0;
            Properties properties = new Properties();
            try {
            properties.load(new FileInputStream("jdbc.properties"));
            } catch (IOException e) {
                        throw new MyDataAccessException("I/O Error", e);
            }
            try {
                        Class.forName(properties.getProperty("driverClassName"));
                        conn = DriverManager.getConnection(properties.getProperty("url"), properties);
                        stmt = conn.createStateent();
                        rs = stmt.executeQuery("select count(*) from beers");
                        if (rs.next()) {
                                    count = rs.getInt(1);
                        }
            } catch (ClassNotFoundException e) {
                        throw new MyDataAccessException("JDBC Error", e);
            } catch (SQLException se) {
                        throw new MyDataAccessException("JDBC Error", se);
            }
            finally {
                        if (conn != null) {
                                    try {
                                                conn.close();
                                    } catch (SQLException ignore) {}
                        }
            }
            return count;
}
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Using Spring

```
public class SpringDao {
public int getBeerCount() {
        DriverManagerDataSource dataSource = new
DriverManagerDataSource();
        int count = 0;
        Properties properties = new Properties();
        try {
                properties.load(new
FileInputStream("jdbc.properties"));
        } catch (IOException e) {
                logger.severe(e.toString());
        }
        dataSource.setConnectionProperties(properties);
        dataSource.setDriverClassName(properties.getProperty("driverCl
assName"));
        dataSource.setUrl(properties.getProperty("url"));
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        count = jdbcTemplate.queryForInt("select count(*) from
beers");
        return count;
}
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Using Spring with JdbcDaoSupport

```
public class SpringDao {
      private DataSource dataSource;
      private JdbcTemplate jdbcTemplate;
      public void setDataSource(DataSource dataSource){
            this.dataSource = dataSource;
      this.jdbcTemplate = new JdbcTemplate(dataSource);
      }
      public int getBeerCount() {
      int count = jdbcTemplate.queryForInt("select
            count(*) from beers");
      return count;
      }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Using Spring with Dependency Injection

```
public class SpringDao extends JdbcDaoSupport
{
     public int getBeerCount() {
          int count = getJdbcTemplate().
     queryForInt("select count(*) from
          beers");
          return count;
     }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring -JDBC

- Utilities for JDBC SQL access (JdbcTemplate)
- Based on Spring IoC concepts
  - Define data source, connection and transaction settings in spring XML definition files
- Hides various technical aspects:
  - Database connection management
  - Database transaction management
  - No specific database or data source binding
  - No need to catch exceptions (all exceptions are runtime).
  - Better hierarchy of exceptions (instead of a single SQLException).

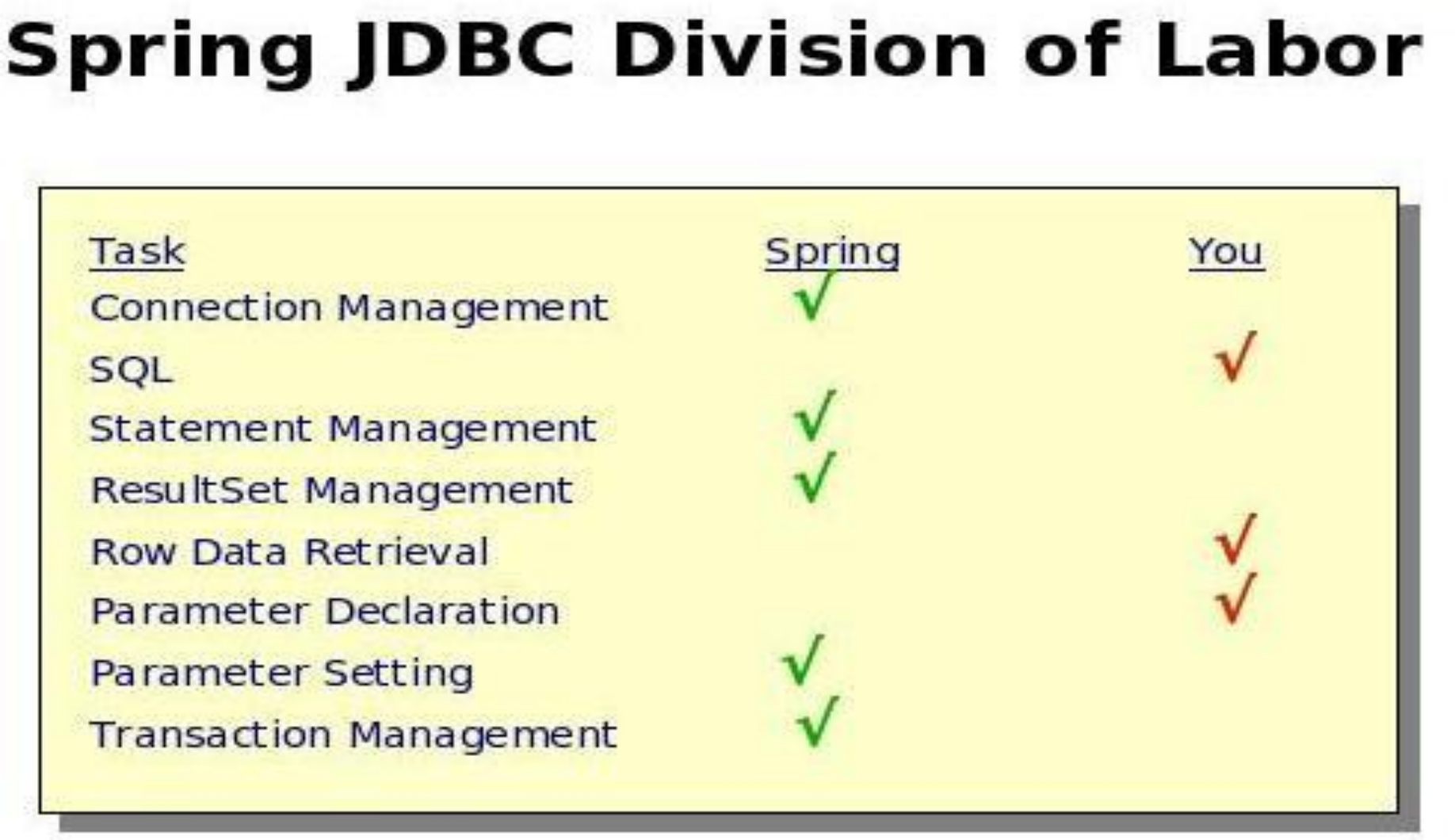
Ver 1.0 © 2007 Capgemini - All rights reserved

JDBC / Spring Comparison

JDBC / Spring comparison

	JDBC	Spring
Connections	Need to explicitly open and close connections. Need a separate strategy for making code reusable in a variety of environments.	Uses a DataSource with the framework managing connections. Code following the framework strategy is automatically reusable.
Exceptions	Must catch SQLExceptions and interpret database specific SQL error code or SQL state code.	Framework translates exceptions to a common hierarchy based on configurable translation mappings.
Testing	Hard to test standalone if code uses JNDI lookup for connection pools.	Can be tested standalone since a DataSource is easily configurable for a variety of environments.
Transactions	Programmatic transaction management is possible but makes code less reusable in systems with varying transaction requirements. CMT is available for EJBs.	Programmatic or declarative transaction management is possible. Declarative transaction management works with single data source or JTA without any code changes.

Ver 1.0 © 2007 Capgemini - All rights reserved



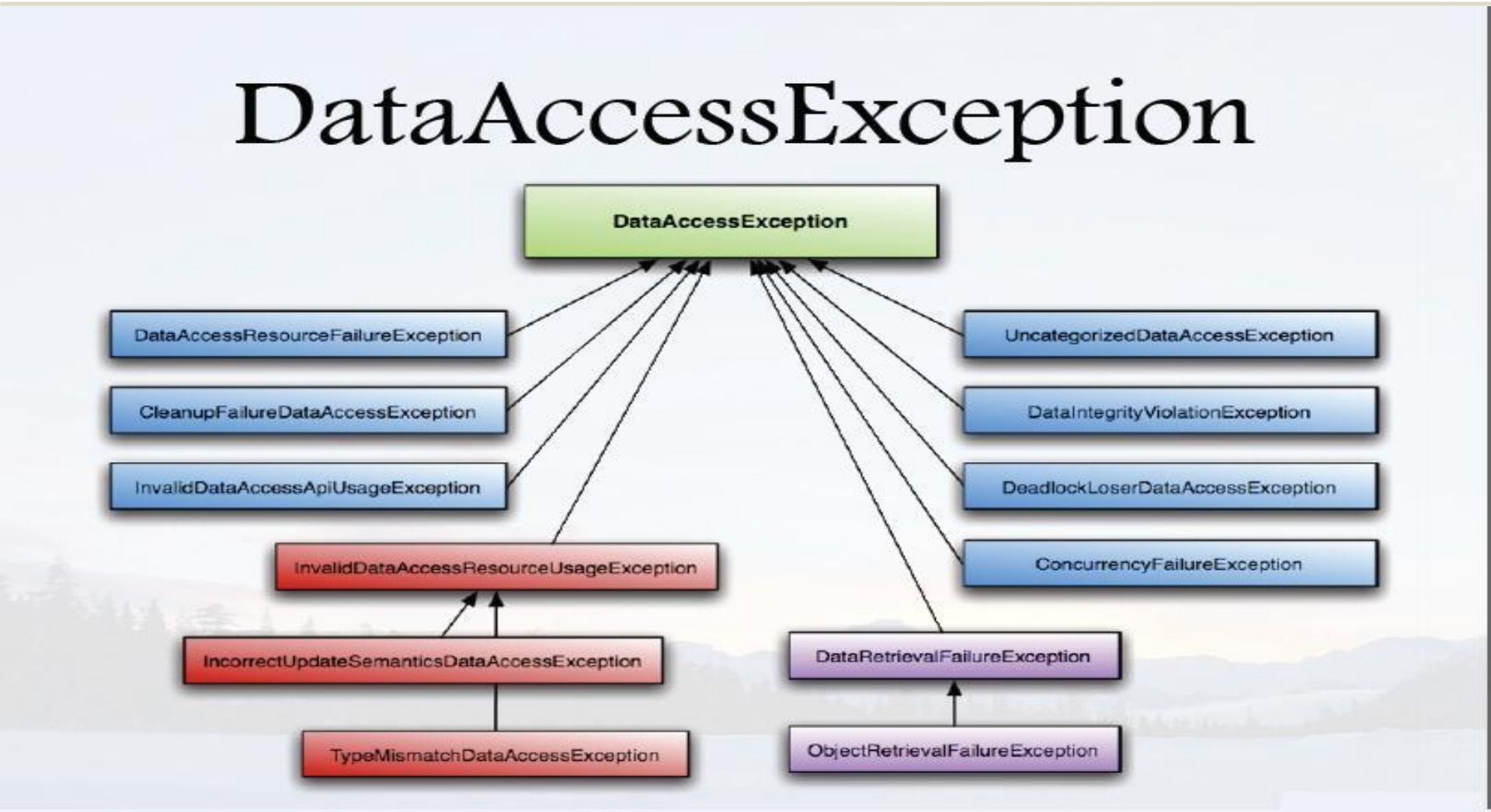
Ver 1.0 © 2007 Capgemini - All rights reserved

JDBC Features

- **JDBC abstraction layer** that
  - provides exception translation that offers a meaningful exception hierarchy and simplifies error handling.
  - Includes a JdbcTemplate with many convenience methods for easier data access.
  - Includes an object layer on top of the JdbcTempate. This layer gives you SqlQuery, SqlUpdate and StoredProcedure classes for more “object oriented” use.
  - manages the connections - you'll never need to write another finally block to use JDBC again.
  - greatly reduces the amount of code you'll need to write.

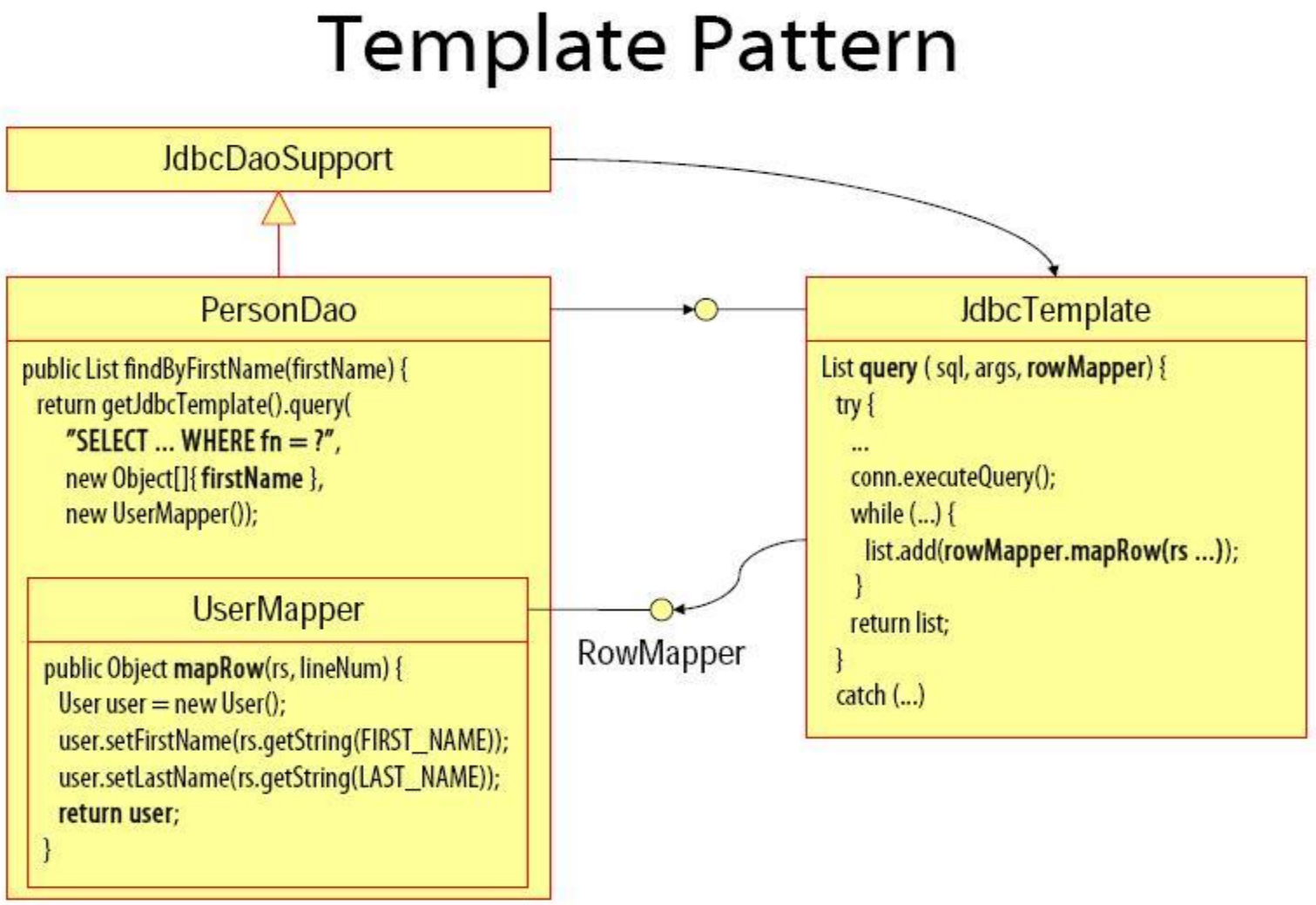
Ver 1.0 © 2007 Capgemini - All rights reserved

Exception Hierarchy



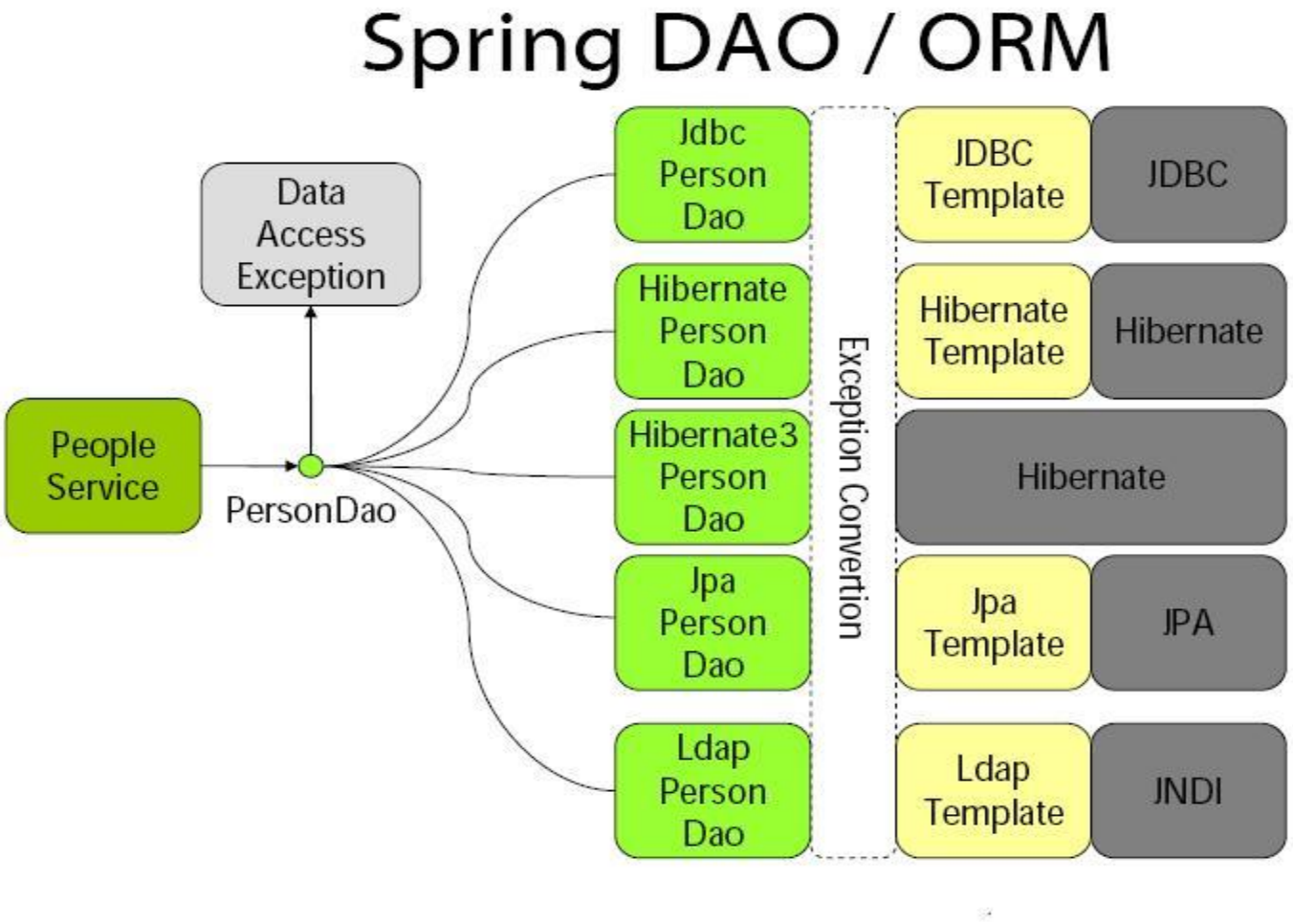
Ver 1.0 © 2007 Capgemini - All rights reserved

Template Pattern



Ver 1.0 © 2007 Capgemini - All rights reserved

Spring DAO / ORM



Ver 1.0 © 2007 Capgemini - All rights reserved

Transaction Consistency

Transaction Consistency

- Common Interface
  - PlatformTransactionManager
- Used
  - Proxy / AOP
  - Annotations
  - Programmatic
- Implementations
  - JTA
  - JDBC Connection
  - Hibernate2
  - Hibernate3
  - JPA
  - JDO
  - JMS (1.0 and 1.1)
  - Toplink
  - ...

Ver. 2.0 © 2007 Capgemini - All rights reserved

Working with JDBC Template

JdbcTemplate—The most basic of Spring's JDBC templates, this class provides simple access to a database through JDBC and simple indexed-parameter queries.

NamedParameterJdbcTemplate—This JDBC template class enables you to perform queries where values are bound to named parameters in SQL, rather than indexed parameters.

SimpleJdbcTemplate—This version of the JDBC template takes advantage of Java 5 features such as autoboxing, generics, and variable parameter lists to simplify how a JDBC template is used.

Ver 1.0 © 2007 Capgemini - All rights reserved

JDBC Template

A simple query for getting the number of rows in a relation:

```
int rowCount = this.jdbcTemplate.queryForInt("select count(0) from t_accrual");
```

A simple query using a bind variable:

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForInt(
        "select count(0) from t_actors where first_name = ?",
        new Object[]{"Joe"});
```

Querying for a String:

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

Ver 1.0 © 2007 Capgemini - All rights reserved

NamedParameterJdbcTemplate

```
// some JDBC-backed DAO class...
Private NamedParameterJdbcTemplate
   namedParameterJdbcTemplate;
public void setDataSource(DataSource dataSource) {
   this.namedParameterJdbcTemplate = new
   NamedParameterJdbcTemplate(dataSource);
}
public int countOfActorsByFirstName(String firstName) {
   String sql = "select count(0) from T_ACTOR where
   first_name = :first_name"; SqlParameterSource
   namedParameters = new
MapSqlParameterSource ("first_name", firstName);
return namedParameterJdbcTemplate.queryForInt(sql,
   namedParameters);
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

JDBC Template

Querying and populating a number of domain objects.:

```
Collection actors = this.jdbcTemplate.query(
        "select first_name, surname from t_actor",

         new RowMapper() {
           public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
             Actor actor = new Actor();
             actor.setFirstName(rs.getString("first_name"));
             actor.setSurname(rs.getString("surname"));
             return actor;
           }
         }
        );
```

Updating (INSERT/UPDATE/DELETE):

```
this.jdbcTemplate.update(
     "insert into t_actor (first_name, surname) values (?, ?)",
     new Object[] {"Leonor", "Watling"});

this.jdbcTemplate.update("update t_actor set weapon = ? where id = ?",
                         new Object[] {"Banjo", new Long(5276)});

this.jdbcTemplate.update("delete from orders");
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Example Retrieving a Specific object

How to retrieve a specific instance of a Beer:

```
package com.springcheers.model;
import java.math.BigDecimal;
public class Beer {
   private Long id;
   private String brand;
   private BigDecimal price;
   ...
   // getters and setters
   ...
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

DAO

```
public class BeerDaoImpl extends JdbcDaoSupport implements BeerDao {
    public Beer getBeer(Long id) {
        Beer beer = (Beer)getJdbcTemplate().queryForObject("select id, brand,
        price from beers where id = ?", new Object[] {id},
        new BeerRowMapper());

        return beer;
    }
    ...
    private static class BeerRowMapper implements RowMapper {
        public Object mapRow(ResultSet resultSet,int i) throws SQLException {
                Beer b = new DomesticBeer();
                b.setId(new Long(resultSet.getLong("id")));
                b.setBrand(resultSet.getString("brand"));
                b.setPrice(resultSet.getBigDecimal("price"));
                return b;
        }
    }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Bean Definition XML

```
<beans>
    <bean id="propertyConfigurer“ class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations“>
            <list>
                <value> com/jdbc/jdbc.properties</value >
                <value> com/jdbc/jdbc1.properties</value >
            </list>
        </property>
</bean>

    <bean id="beerDao" class="com.springcheers.dao.jdbc.BeerDaoImpl">
        <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>
</beans>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

MappingSqlQuery

```
public class BeerDaoImpl extends JdbcDaoSupport implements BeerDao {
    private BeerQuery beerQuery;
    public void initDao() {
          beerQuery = new BeerQuery(getDataSource());
    }
    public Beer getBeer(Long id) {
          return beerQuery.find(id);
    }

    public class BeerQuery extends MappingSqlQuery {
    private static final String sql = "select id, brand, price from beers where id = ?";
          public BeerQuery(DataSource dataSource) {
                    super(dataSource, sql);
                    declareParameter(new SqlParameter("id", Types.INTEGER));
                    compile();
          }
          public Beer find(Long id) {
                    return (Beer) findObject(new Object[] {id});
          }
          protected Object mapRow(ResultSet resultSet,int i) throws SQLException {
                    Beer b = new DomesticBeer();
                    b.setId(new Long(resultSet.getLong("id")));
                    b.setBrand(resultSet.getString("brand"));
                    b.setPrice(resultSet.getBigDecimal("price"));
                    return b;
          }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

JdbcTemplate Update

```
public class BeerDaoImpl extends JdbcDaoSupport
   implements BeerDao {

...

   public void updateBeer(Beer beer) {

       getJdbcTemplate().update("update beers set brand
   = ?, price = ?  where id = ?",

       new Object[3] {beer.getBrand(),
       beer.getPrice(),

       beer.getId()});

   }

...

}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

SqlUpdate

```
public class BeerDaoImpl extends JdbcDaoSupport implements BeerDao {
    private BeerUpdate beerUpdate;
    public void initDao() {
          beerUpdate = new BeerUpdate(getDataSource());
    }
    public void updateBeer(Beer beer) {
          beerUpdate.update(beer);
    }

    public class BeerUpdate extends SqlUpdate {
          private static final String sql = "update beers set brand = ?, price = ? where
    id = ?";
          public BeerUpdate(DataSource dataSource) {
                super(dataSource, sql);
                declareParameter(new SqlParameter("brand", Types.VARCHAR));
                declareParameter(new SqlParameter("price", Types.DECIMAL));
                declareParameter(new SqlParameter("id", Types.INTEGER));
                compile();
          }
          public void update(Beer beer) {
                Object[] params = new Object[3];
                params[0] = beer.getBrand();
                params[1] = beer.getPrice();
                params[2] = beer.getId();
                update(params);
          }
    }
}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

StoredProcedure

```
public class OrderDaoImpl extends JdbcDaoSupport implements OrderDao {
    private DeleteOrderProc delete;
    public void initDao() {
          delete = new DeleteOrderProc(getDataSource());
    }
    public void deleteOrder(Long id) {
          delete.execute(id);
    }

    public class DeleteOrderProc extends StoredProcedure {
          private static final String sql = "delete_order";
          public DeleteOrderProc(DataSource dataSource) {
                  super(dataSource, sql);
                  declareParameter(new SqlParameter("id", Types.INTEGER));
                  compile();
          }
          public void execute(Long id) {
                  Map params = new HashMap(1);
                  params.put("id", id);
                  execute(params);
          }
    }
```

Ver 1.0 © 2007 Capgemini - All rights reserved

JDBC Template

Java DAO class that uses JdbcTemplate:

```
public class JdbcCorporateEventDao implements CorporateEventDao {
  private JdbcTemplate jdbcTemplate;

  public int countEmployees(String name) {

    return this.jdbcTemplate.queryForInt("select count(*) from employee where name = ?"
                                         new Object[]{name});

  }
}
```

Definitions of data source, jdbcTemplate and DAO beans:

```
<beans ... >

  <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
  </bean>

  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>

  </bean>

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

    <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>
</beans>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Configuring a Data Source

Configuring a data source

Data sources that are defined by a JDBC driver

Data sources that are looked up by JNDI

Data sources that pool connections

Ver 1.0 © 2007 Capgemini - All rights reserved

Configuring a Data Source

JNDI data sources

```
<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean“ scope="singleton">
<property name="jndiName" value="/jdbc/Vishal" />
<property name="resourceRef" value="true" />
</bean>
```

JNDI data sources in Spring

```
<bean id="serverDataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:comp/env/jdbc/blah"/>
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Configuring a Data Source

Using a pooled data source

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName" value="org.hsqldb.jdbcDriver" />
<property name="url“ value="jdbc:hsqldb:hsql://localhost/vishal/vishal" />
<property name="username" value="sa" />
<property name="password" value="" />
<property name="initialSize" value="5" />
<property name="maxActive" value="10" />
</bean>
```

JDBC driver-based data source

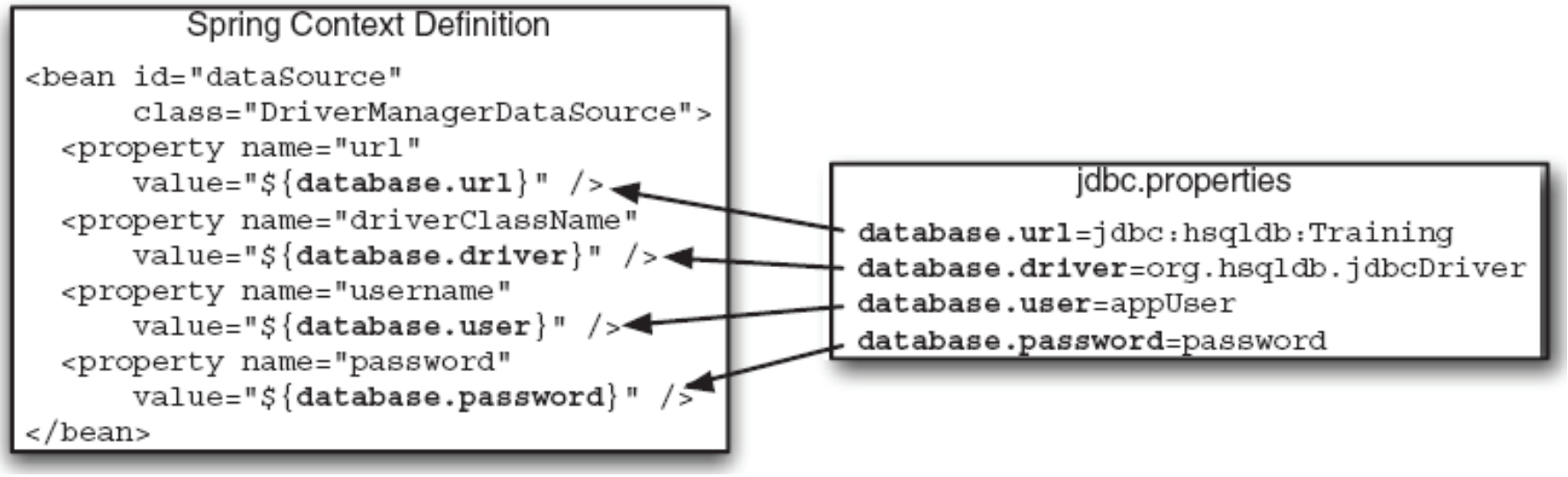
```
<bean id="dataSource“ class="org.springframework.jdbc.datasource. DriverManagerDataSource">
<property name="driverClassName“ value="com.mysql.jdbc.Driver" />
<property name="url“ value="jdbc:mysql://localhost:3306/training" />
<property name="username" value=“root" />
<property name="password" value="" />
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Externalizing Configuration Properties

Externalizing configuration properties

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



Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

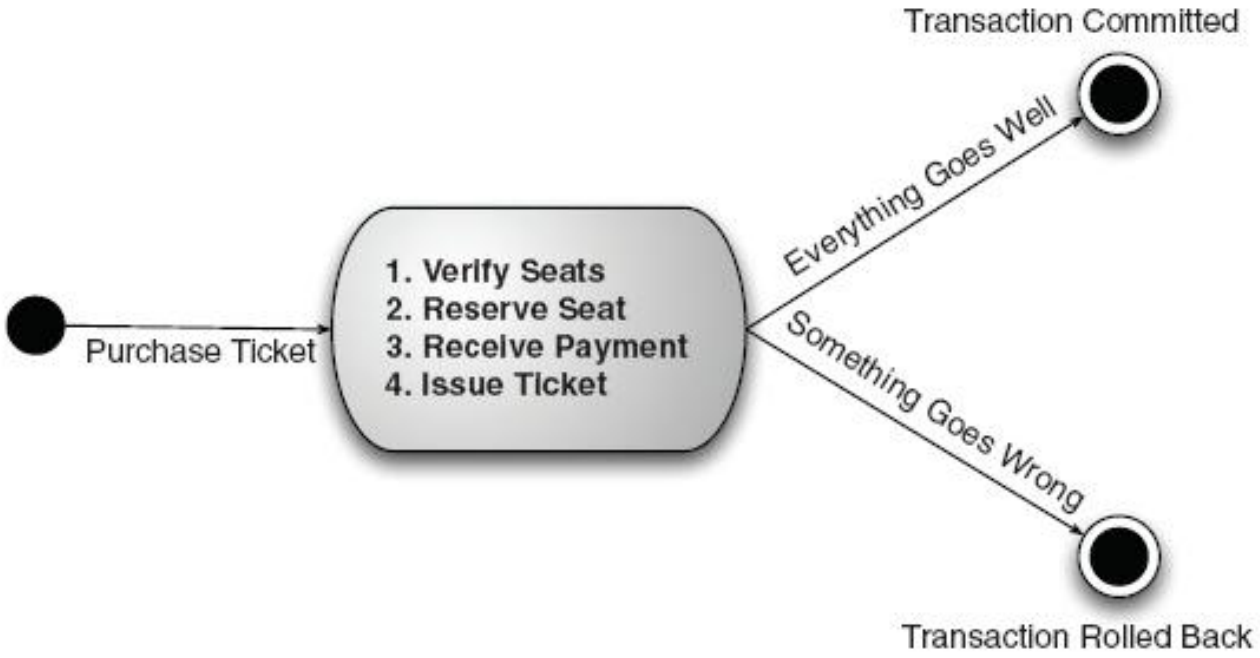
Understanding transactions

To illustrate transactions, consider the purchase of a movie ticket. Purchasing a ticket typically involves the following actions:

- The number of available seats will be examined to verify that there are enough seats available for your purchase.
- The number of available seats is decremented by one for each ticket purchased.
- You provide payment for the ticket.
- The ticket is issued to you.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support



Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Explaining transactions in only four words –ACID

Atomic—Transactions are made up of one or more activities bundled together as a single unit of work. Atomicity ensures that all the operations in the transaction happen or that none of them happen. If all the activities succeed, the transaction is a success. If any of the activities fail, the entire transaction fails and is rolled back.

Consistent—Once a transaction ends (whether successful or not), the system is left in a state that is consistent with the business that it models. The data should not be corrupted with respect to reality.

Isolated—Transactions should allow multiple users to work with the same data, without each user's work getting tangled up with the others. Therefore, transactions should be isolated from each other, preventing concurrent reads and writes to the same data from occurring. (Note that isolation typically involves locking rows and/or tables in a database.)

Durable—Once the transaction has completed, the results of the transaction should be made permanent so that they will survive any sort of system crash. This typically involves storing the results in a database or some other form of persistent storage.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Choosing a transaction manager

Transaction manager (`org.springframework.*`)	Use it when...
`jca.cci.connection.CciLocalTransactionManager`	Using Spring's support for J2EE Connector Architecture (JCA) and the Common Client Interface (CCI).
`jdbc.datasource.DataSourceTransactionManager`	Working with Spring's JDBC abstraction support. Also useful when using iBATIS for persistence.
`jms.connection.JmsTransactionManager`	Using JMS 1.1+.
`jms.connection.JmsTransactionManager102`	Using JMS 1.0.2.
`orm.hibernate.HibernateTransactionManager`	Using Hibernate 2 for persistence.
`orm.hibernate3.HibernateTransactionManager`	Using Hibernate 3 for persistence.
`orm.jdo.JdoTransactionManager`	Using JDO for persistence.

Ver 1.0 © 2007 Capgemini - All rights reserved

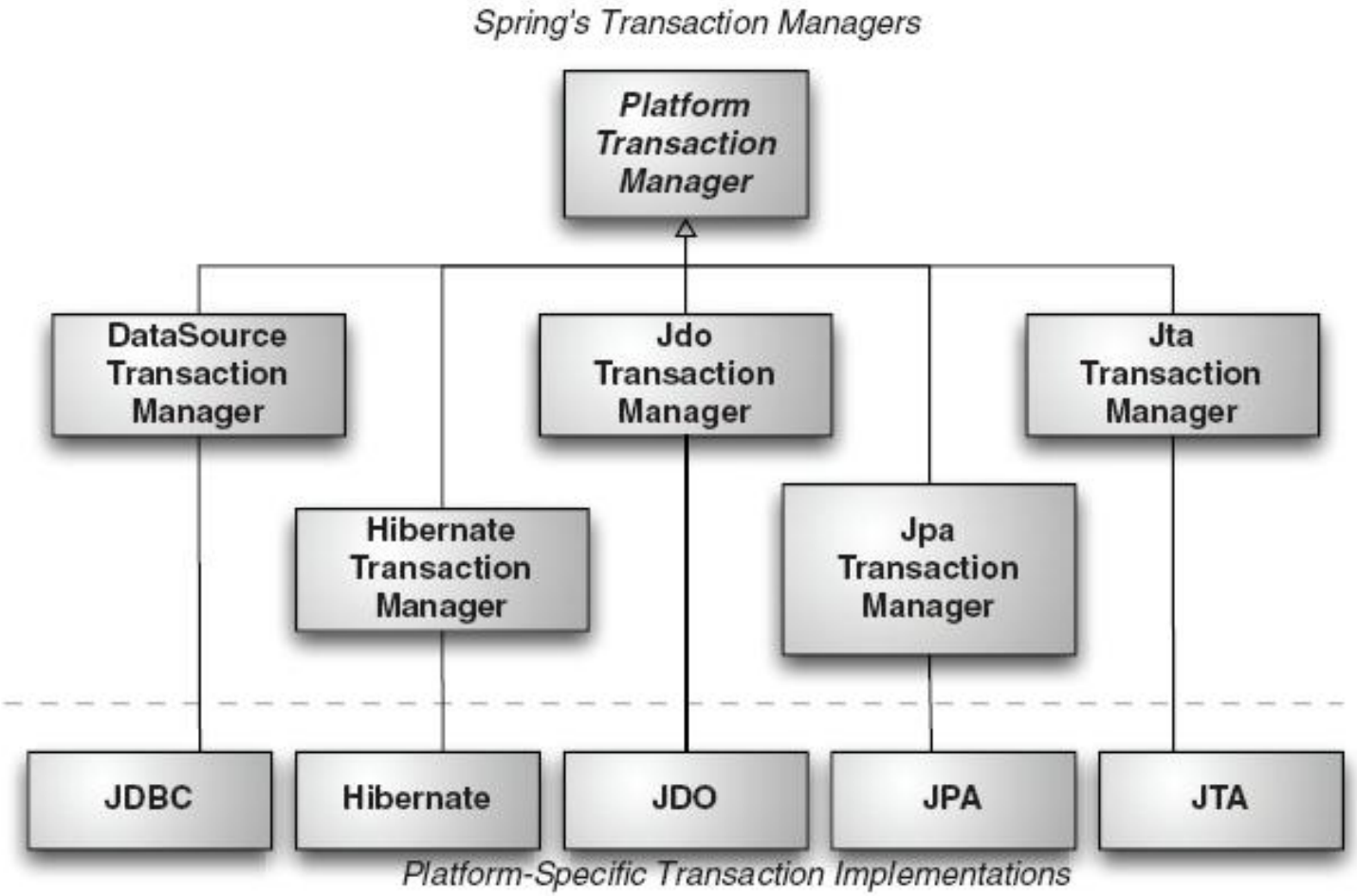
Spring Transaction Management Support

Choosing a transaction manager

Transaction manager (`org.springframework.*`)	Use it when...
`orm.jpa.JpaTransactionManager`	Using the Java Persistence API (JPA) for persistence.
`orm.toplink.TopLinkTransactionManager`	Using Oracle's TopLink for persistence.
`transaction.jta.JtaTransactionManager`	You need distributed transactions or when no other transaction manager fits the need.
`transaction.jta.OC4JJtaTransactionManager`	Using Oracle's OC4J JEE container.
`transaction.jta.WebLogicJtaTransactionManager`	You need distributed transactions and your application is running within WebLogic.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support



Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

JDBC transactions

If you're using straight JDBC for your application's persistence, DataSourceTransactionManager will handle transactional boundaries for you.

To use DataSource-TransactionManager, wire it into your application's context definition using the following XML:

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

Hibernate transactions

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
<property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Java Transaction API transactions

If none of the aforementioned transaction managers meet your needs or if your transactions span multiple transaction sources (e.g., two or more different databases), you'll need to use JtaTransactionManager:

```
<bean id="transactionManager" class="org.springframework.
transaction.jta.JtaTransactionManager">
<property name="transactionManagerName"
value="java:/TransactionManager" />
</bean>
```

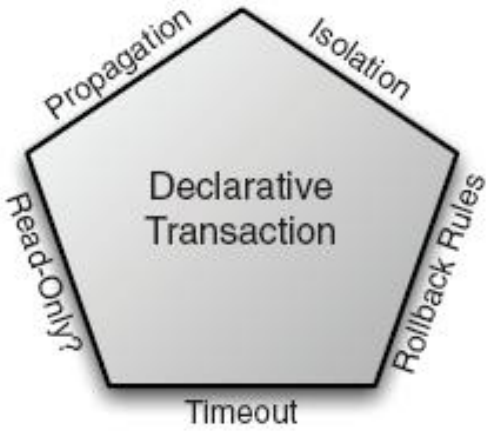
Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Declaring transactions

Spring's support for declarative transaction management is implemented through Spring's AOP framework. This is a natural fit because transactions are a system-level service above an application's primary functionality. You can think of a Spring transaction as an aspect that "wraps" a method with transactional boundaries.

In Spring, declarative transactions are defined with transaction attributes. A transaction attribute is a description of how transaction policies should be applied to a method. There are five facets of a transaction attribute, as illustrated in figure



Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Propagation behavior

Table 1 Propagation rules define when a transaction is created or when an existing transaction can be used. Spring provides several propagation rules to choose from.

Propagation behavior	What it means
PROPAGATION_MANDATORY	Indicates that the method must run within a transaction. If no existing transaction is in progress, an exception will be thrown.
PROPAGATION_NESTED	Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rolled back individually from the enclosing transaction. If no enclosing transaction exists, behaves like PROPAGATION_REQUIRED. Vendor support for this propagation behavior is spotty at best. Consult the documentation for your resource manager to determine if nested transactions are supported.
PROPAGATION_NEVER	Indicates that the current method should not run within a transactional context. If there is an existing transaction in progress, an exception will be thrown.
PROPAGATION_NOT_SUPPORTED	Indicates that the method should not run within a transaction. If an existing transaction is in progress, it will be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.

Spring Transaction Management Support

Propagation behavior

Propagation behavior	What it means
`PROPAGATION_REQUIRED`	Indicates that the current method must run within a transaction. If an existing transaction is in progress, the method will run within that transaction. Otherwise, a new transaction will be started.
`PROPAGATION_REQUIRES_NEW`	Indicates that the current method must run within its own transaction. A new transaction is started and if an existing transaction is in progress, it will be suspended for the duration of the method. If using `JTATransactionManager`, access to `TransactionManager` is required.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Isolation levels

The second dimension of a declared transaction is the isolation level. An isolation level defines how much a transaction may be impacted by the activities of other concurrent transactions

Dirty read—Dirty reads occur when one transaction reads data that has been written but not yet committed by another transaction. If the changes are later rolled back, the data obtained by the first transaction will be invalid.

Nonrepeatable read—Nonrepeatable reads happen when a transaction performs the same query two or more times and each time the data is different. This is usually due to another concurrent transaction updating the data between the queries.

Phantom reads—Phantom reads are similar to nonrepeatable reads. These occur when a transaction (T1) reads several rows, and then a concurrent transaction (T2) inserts rows. Upon subsequent queries, the first transaction (T1) finds additional rows that were not there before.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Isolation levels

Isolation level	What it means
ISOLATION_DEFAULT	Use the default isolation level of the underlying data store.
ISOLATION_READ_UNCOMMITTED	Allows you to read changes that have not yet been committed. May result in dirty reads, phantom reads, and nonrepeatable reads.
ISOLATION_READ_COMMITTED	Allows reads from concurrent transactions that have been committed. Dirty reads are prevented, but phantom and nonrepeatable reads may still occur.
ISOLATION_REPEATABLE_READ	Multiple reads of the same field will yield the same results, unless changed by the transaction itself. Dirty reads and nonrepeatable reads are prevented, but phantom reads may still occur.
ISOLATION_SERIALIZABLE	This fully ACID-compliant isolation level ensures that dirty reads, nonrepeatable reads, and phantom reads are all prevented. This is the slowest of all isolation levels because it is typically accomplished by doing full table locks on the tables involved in the transaction.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Proxying transactions

Listing 6.3 Proxying the rant service for transactions

```
<bean id="rantService'
    class="org.springframework.transaction.interceptor.
          ➥ TransactionProxyFactoryBean">

  <property name='target'
      ref='rantServiceTarget' />

  <property name='proxyInterfaces'
      value='com.roadrantz.service.RantService' />

  <property name='transactionManager'
      ref='transactionManager' />

  <property name='transactionAttributes'>
    <props>
      <prop key='add*'>PROPAGATION_REQUIRED</prop>
      <prop key='*'>PROPAGATION_SUPPORTS,readOnly</prop>
    </props>
  </property>
</bean>
```

Wires transaction target

Specifies proxy interface

Wires in transaction manager

Configures transaction rules, boundaries

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Proxying transactions

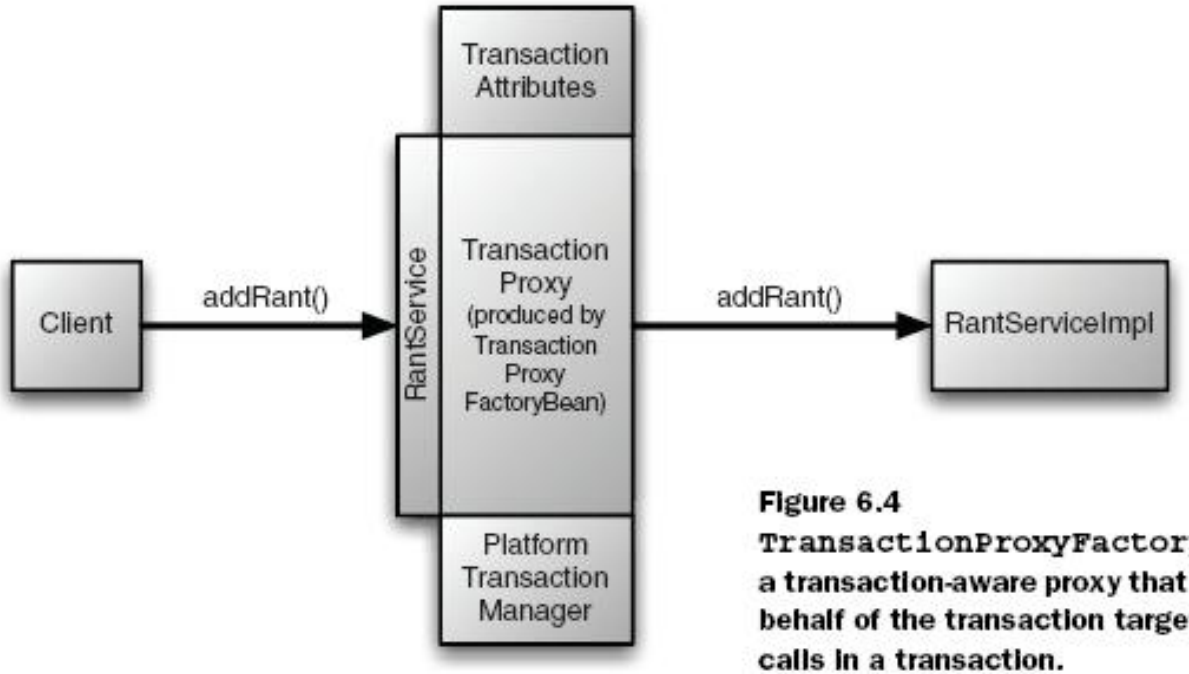


Figure 6.4
`TransactionProxyFactoryBean` produces a transaction-aware proxy that receives calls on behalf of the transaction target, wrapping the calls in a transaction.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring Transaction Management Support

Proxying transactions

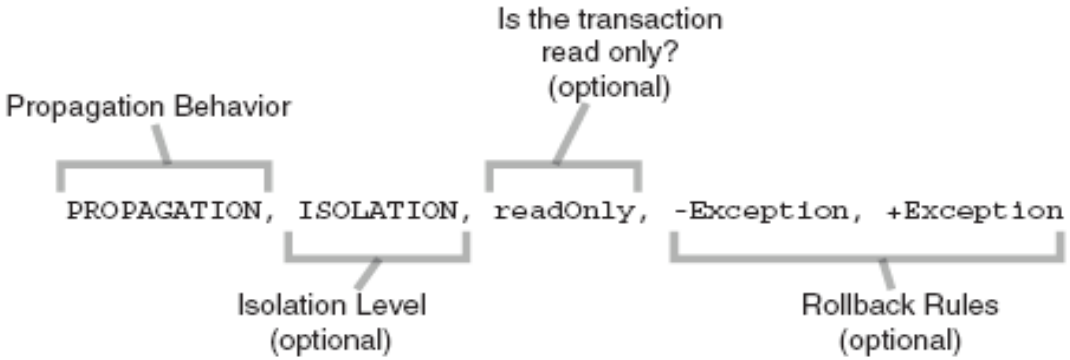


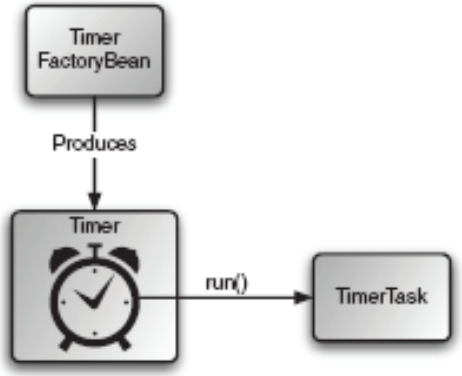
Figure 6.5 A transaction attribute definition is made up of a propagation behavior, an isolation level, a read-only flag, and rollback rules. The propagation behavior is the only required element.

Ver 1.0 © 2007 Capgemini - All rights reserved

Scheduling tasks

Ver 1.0 © 2007 Capgemini - All rights reserved

Scheduling with Java's Timer



Starting with Java 1.3, the Java SDK has included rudimentary scheduling functionality through its `java.util.Timer` class. This class lets you schedule a task (defined by a subclass `java.util.TimerTask`) to occur every so often.

Spring provides application context support for Java's `Timer` through `TimerFactoryBean`. `TimerFactoryBean` is a Spring factory bean that produces a Java `Timer` in the application context that kicks off a `TimerTask`. Figure 12.4 illustrates how `TimerFactoryBean` works.

Ver 1.0 © 2007 Capgemini - All rights reserved

Scheduling tasks

```
<bean id="myTask" class="com.scheduler.ScheduleTask"/>

<bean id="schTimerTask"
    class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject">
          <ref bean="myTask"/>
    </property>
    <property name="targetMethod">
          <value>doSomething</value>
    </property>
</bean>

<bean id="scheduledTask" class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <property name="timerTask" ref="schTimerTask"/>
    <property name="period" value="1000"/>
</bean>

<bean id="timerFactory"  class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
          <list>
                <ref bean="scheduledTask"/>
          </list>
    </property>
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Sending E-Mail

Configuring a mail send

Spring comes with an email abstraction API that makes simple work of sending emails. At the heart of Spring's email abstraction is the MailSender interface.

Figure 12.3 Spring's `MailSender` interface is primary component of Spring's email abstraction API. It simply sends an email to a mail server for delivery.

Ver 1.0 © 2007 Capgemini - All rights reserved

Sending E-Mail

```
<bean id="mailSender"
   class="org.springframework.mail.javamail.JavaMailSenderImpl">
   <property name="host"><value>myhost.com</value></property>
</bean>

<bean id="mailMessage"
   class="org.springframework.mail.SimpleMailMessage">
   <property name="from"><value>abc@xyz.com</value></property>
   <property name="subject"><value>Test Mail</value></property>
</bean>

<bean id="testMailImpl" class="com.mail.TestMailImpl">
   <property name="mailSender"><ref bean="mailSender"/></property>
   <property name="message"><ref bean="mailMessage"/></property>
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC

Ver 1.0 © 2007 Capgemini - All rights reserved

MVC

MVC = Model-View-Controller

Clearly separates business, navigation and presentation logic Proven mechanism for building a thin, clean web-tier

Ver 1.0 © 2007 Capgemini - All rights reserved

MVC Components

Three core collaborating components

Controller

- Handles navigation logic and interacts with the service tier for business logic

Model

- The contract between the Controller and the View
- Contains the data needed to render the View
- Populated by the Controller

View

- Renders the response to the request
- Pulls data from the model

Ver 1.0 © 2007 Capgemini - All rights reserved

MVC In Spring

A single **Front Controller** servlet that dispatches requests to individual Controllers

- Proven pattern shown in Struts and Core J2EE Patterns

Request routing is completely controlled by the Front Controller

- Individual Controllers can be used to handle many different URLs

Controllers are POJOs

Controllers are managed exactly like any other bean in the Spring ApplicationContext

Ver 1.0 © 2007 Capgemini - All rights reserved

Core Components of Spring MVC

DispatcherServlet

- Spring's Front Controller implementation

Controller

- User created component for handling requests
- Encapsulates navigation logic
- Delegates to the service objects for business logic

View

- Responsible for rendering output

Ver 1.0 © 2007 Capgemini - All rights reserved

ModelAndView

Created by the Controller

Stores the Model data

Associates a View to the request

• Can be a physical View implementation or a logical View name

ViewResolver

Used to map logical View names to actual View implementations

HandlerMapping

Strategy interface used by `DispatcherServlet` for mapping incoming requests to individual Controllers

Ver 1.0 © 2007 Capgemini - All rights reserved

MVC and Dependency Injection

All MVC components are configured in the Spring ApplicationContext

As such, all MVC components can be configured using Dependency Injection

Example:

```
<bean id="springCheersController"
  class="com....web.SpringCheersController">

  <property name="methodNameResolver”
     ref=“springCheersMethodResolver"/>

  <property name="service" ref="service"/>

</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Creating a Basic Controller

Goals

- Create a thin-wrapper around the business functionality
- Keep all business processing out of the web tier
- Handle only navigation logic

Process

- Create the Controller class
  - Implement the Controller interface
  - Or extend one of the pre-built Controller implementations
- Create a setter to inject the service object
- Implement the `handleRequest()` method

Ver 1.0 © 2007 Capgemini - All rights reserved

Creating a Basic Controller

```
public class BeerListController implements Controller {
   private SpringCheersService service;
   public void setService(SpringCheersService service) {
       this.service = service;
   }
   public ModelAndView handleRequest(
   HttpServletRequest request, HttpServletResponse
   response)
   throws Exception {
       List beers = this.service.findAllBeers();
return new ModelAndView("beerList", "beers", beers);
   }
}
```

View name

Model parameter name

Model parameter

Ver 1.0 © 2007 Capgemini - All rights reserved

Views in Spring MVC

- Extensive support for many different view technologies
  - JSP, JSTL, Velocity, FreeMarker, JasperReports, PDF, Excel
- Views are represented using logical view names which are returned by the Controller
- Can return an actual `View` class from the Controller if needed

Ver 1.0 © 2007 Capgemini - All rights reserved

View Resolution in Spring MVC

View names are mapped to actual view implementations using `ViewResolvers`

`ViewResolvers` are configured in the web-tier `ApplicationContext`

Automatically detected by `DispatcherServlet`

Can configure multiple `ViewResolvers`

Ver 1.0 © 2007 Capgemini - All rights reserved

ViewResolver Implementations

- InternalResourceViewResolver
  - Uses RequestDispatcher to route requests to internal resources such as JSPs
  - Model data is placed in request scope for access in the view
- FreeMarkerViewResolver
  - Uses FreeMarkerView to render the response using the FreeMarker template engine
- VelocityViewResolver
  - Uses VelocityView to render the response using the Velocity template engine
- BeanNameViewResolver
  - Maps the view name to the name of a bean in the ApplicationContext.
  - Allows for view instances to be explicitly configured

Ver 1.0 © 2007 Capgemini - All rights reserved

Creating a View with JSP and JSTL

```
<%@ page contentType="text/html;charset=UTF-8"
   language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
   prefix="c" %>
<html>
   <head><title>Beer List</title></head>
   <body>
   <table border="0">
   <c:forEach items="${beers}" var="beer">
        <tr>
        <td><c:out value="${beer.id}"/></td>
        <td><c:out value="${beer.brand}"/></td>
        </tr>
   </c:forEach>
   </table>
   </body>
</html>
```

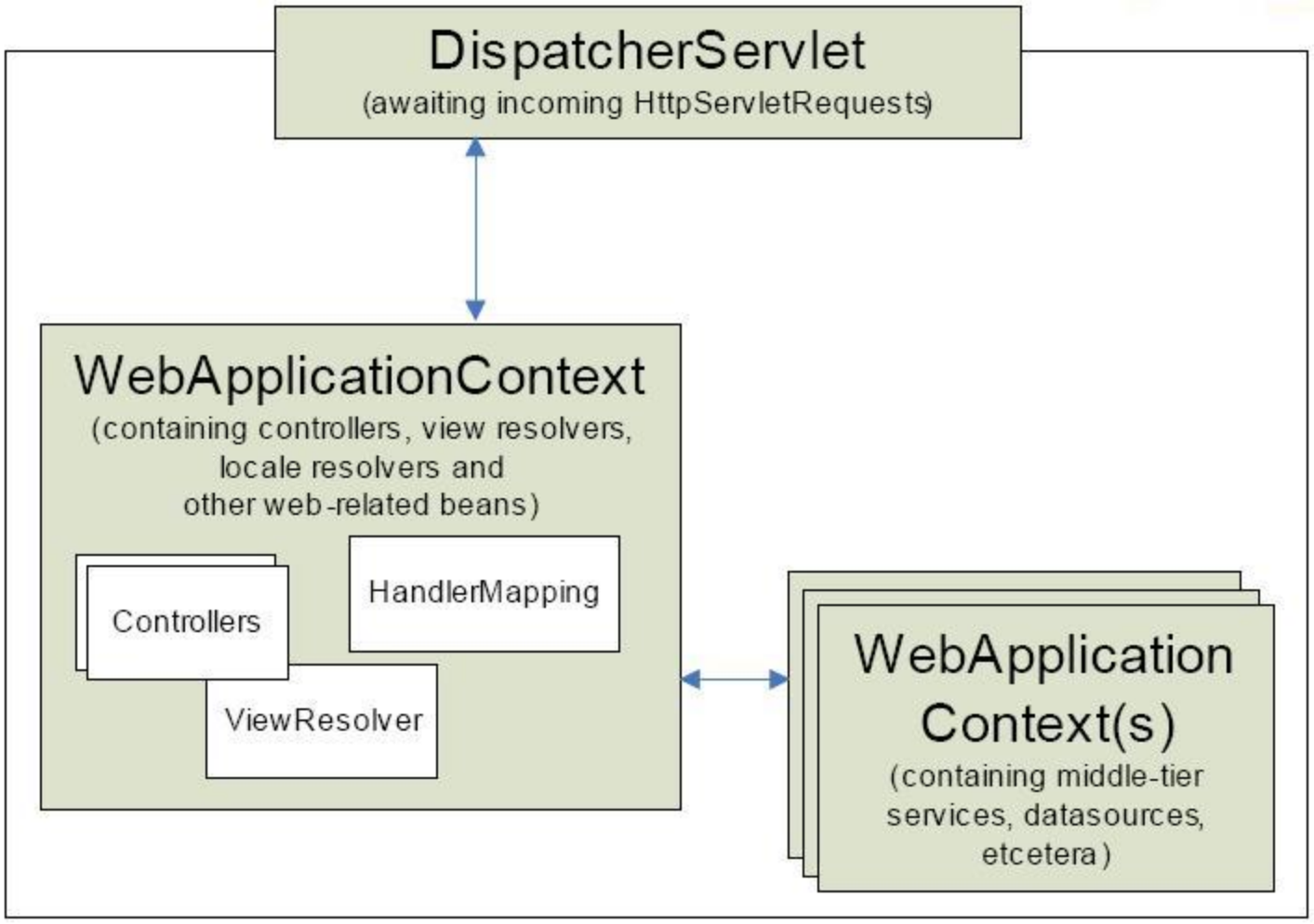
Ver 1.0 © 2007 Capgemini - All rights reserved

Configure a Spring MVC Application

- Configure the **`DispatcherServlet`** in web.xml
- Create the web-tier **`ApplicationContext`** configuration file
- Configure Controllers
- Map URLs to Controllers
- Map logical view names to view implementations

Ver 1.0 © 2007 Capgemini - All rights reserved

Configure a Spring MVC Application



Ver 1.0 © 2007 Capgemini - All rights reserved

Configuring DispatcherServlet

```
<servlet>
   <servlet-name>springcheers</servlet-name>
   <servlet-class>
       o.s.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>springcheers</servlet-name>
   <url-pattern>/myapp/</url-pattern>
</servlet-mapping>
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/applicationContext.xml</param-
   value>
</context-param>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Configuring ContextLoaderListener

```
<servlet>
   <servlet-name>springcheers</servlet-name>
   <servlet-class>o.s. b.s.DispatcherServlet</servlet-class>
   <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>springcheers</servlet-name>
   <url-pattern>/myapp/*</url-pattern>
</servlet-mapping>
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Creating a Spring MVC Application

Creating the web-tier `ApplicationContext` configuration:

- Naming is important – follows the pattern `/WEB-INF/<servlet_name>-servlet.xml`
- `DispatcherServlet` will automatically load this file when setting up its `ApplicationContext`
- In our example this would be `/WEB-INF/springcheers.xml`

Ver 1.0 © 2007 Capgemini - All rights reserved

Mapping URLs to Controllers

Mapping request (URLs) to Controller Controlled by implementations of the HandlerMapping interface

Useful out-of-the-box implementations

BeanNameUrlHandlerMapping

- Uses the Controller bean name as the URL mapping

SimpleUrlHandlerMapping

- Define a set of URL pattern to bean mappings

Ver 1.0 © 2007 Capgemini - All rights reserved

Configure a HandlerMapping

```
<bean id="urlMapping"
      class="o.s.web.servlet.handler.SimpleUrlHandlerMapping">
   <property name="mappings">
      <props>
         <prop key="/list.htm">springCheersController</prop>
         <prop key="/view.htm">springCheersController</prop>
         <prop key="/edit.htm">customerForm</prop>
         <prop key="/create.htm">customerForm</prop>
         <prop key="/beer/list.htm">beerListController</prop>
      </props>
   </property>
</bean>
```

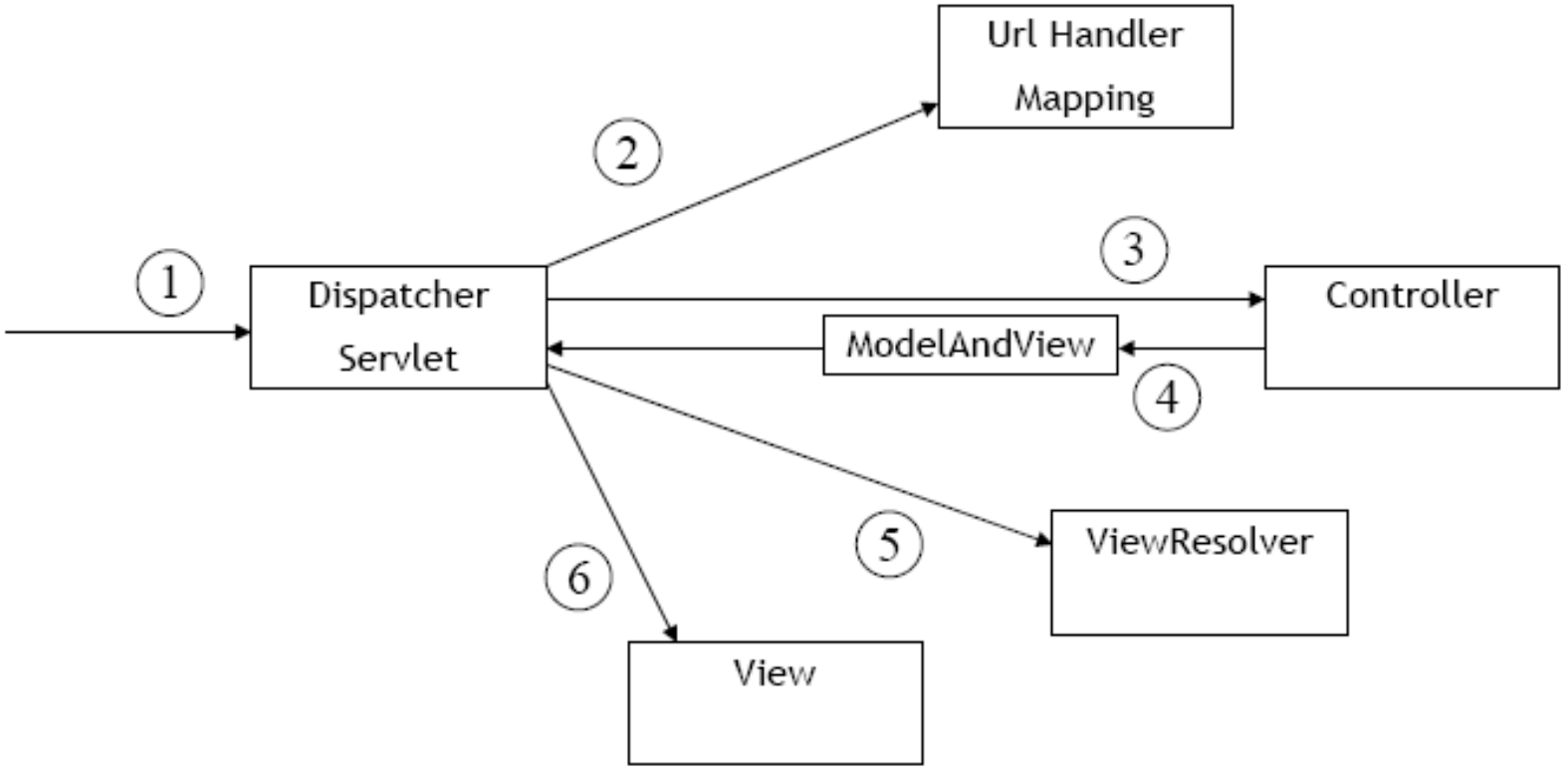
Ver 1.0 © 2007 Capgemini - All rights reserved

Configuring the ViewResolver

```
<bean id="viewResolver"
  class="o.s.w.servlet.view.InternalResourceV
  iewResolver">
  <property name="prefix" value="/WEB-
  INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Request Lifecycle



Ver 1.0 © 2007 Capgemini - All rights reserved

Dispatcher Servlet

Spring MVC's front controller

Coordinates the entire request lifecycle

Configured in web.xml

```
<web-app>
<servlet>
<servlet-name>user</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>  <servlet-name>user</servlet-name> <url-pattern>*.htm</url-pattern>
</servlet-mapping>
</web-app>
```

- Loads Spring application context from XML file (default is<servlet-name>-servlet.xml) that usually contains <bean> definitions for the Spring MVC components

Ver 1.0 © 2007 Capgemini - All rights reserved

Project Re structuring

Web Project

1. Create web project with Name User Maintenance
2. Copy all jars from your previous project to WEB-INF/lib
3. Update web-xml file with dispatcher servlet entry
4. Copy all your source code from previous project to src folder
5. Set up application server , can somebody help to setup the application
6. server for all associates
7. Just check all compilation errors has fixed

Ver 1.0 © 2007 Capgemini - All rights reserved

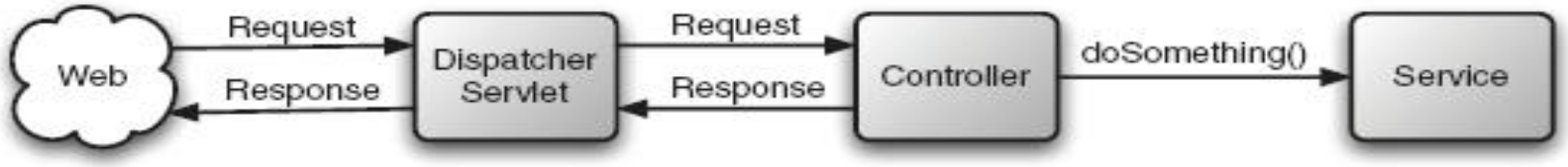
Spring MVC in nut shell

1. Write the controller class that performs the logic behind the userSummarypage. The logic involves using a UserService to retrieve the list of recent Users.
2. Configure the controller in the DispatcherServlet's context configuration file (user-servlet.xml).
3. Configure a view resolver to tie the controller to the JSP.
4. Write the JSP that will render the user Summary to the user.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC

Building the controller



A controller handles web requests on behalf of the DispatcherServlet. A well-designed controller doesn't do all of the work itself—it delegates to a service layer object for business logic.

Configuring a context loader

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/service-bean.xml /WEB-INF/dao-bean.xml</param-value>
</context-param>
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC

Listing **UserSummaryController**, which retrieves a list of recent users for display on the User Summary page

```
public class UserSummaryController extends AbstractController {
private IUserService userService;
protected ModelAndView handleRequestInternal(HttpServletRequest req,
HttpServletResponse res) throws Exception {
System.out.println("inside model view");
List<User> users = userService.findAllUser();
return new ModelAndView("userSummary","users",users);
}
public void setUserService(IUserService userService) {
this.userService = userService;
}}
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Introducing ModelAndView

A **ModelAndView** object, as its name implies, fully encapsulates the view and model data that is to be displayed by the view. In the case of **UserSummaryController,** the ModelAndView object is constructed as follows:

new ModelAndView(“userSummary", “users", users);

The first parameter of this **ModelAndView** constructor is the logical name of a view component that will be used to display the output from this controller. Here the logical name of the view is **userSummary**.

The next two parameters represent the **model** object that will be passed to the view. These two parameters act as a **name-value pair**. The second parameter is the name of the model object given as the third parameter. In this case, the list of **users** in the **users** variable will be passed to the view with a name of **users**.

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC

Configuring the controller bean

```
<bean name="/user.do"
class="com.spring.web.UserSummaryController">
<property name="users" ref="userService"></property>
</bean>
```

Declaring a view resolver

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

Ver 1.0 © 2007 Capgemini - All rights reserved

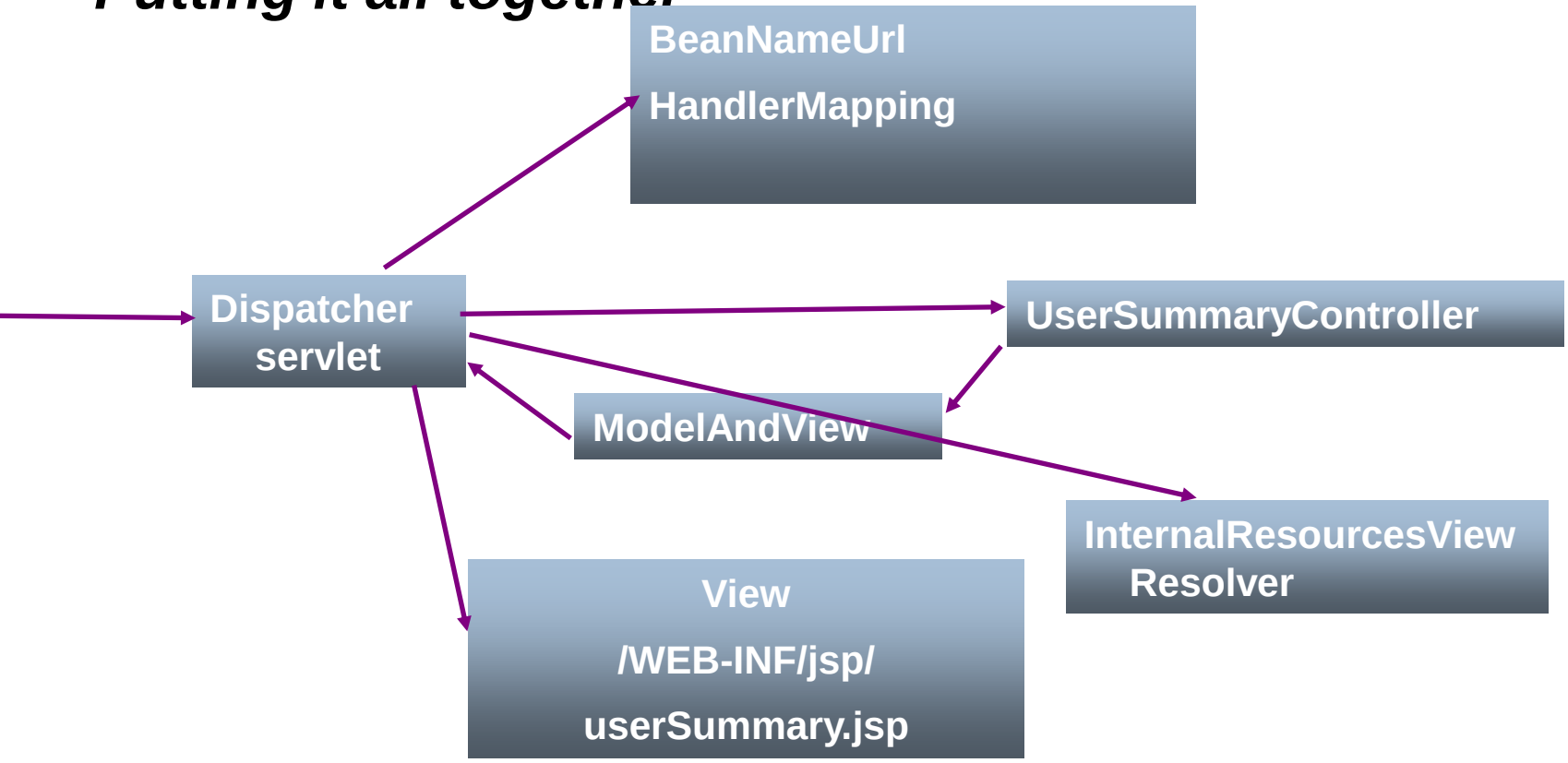
Creating the JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
        <head>
                <title>My JSP 'userSummary.jsp' starting page</title>
        </head>
                <body>
                        <h2>Welcome to UserSummary</h2>
<h3>Recent Users:</h3>
<ul><c:forEach items="${users}" var="user">
<li><c:out value="${user.userId}" />
<c:out value="${user.email}" /></li>
</c:forEach>
</ul>
</body></html>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC

Putting it all together



Ver 1.0 © 2007 Capgemini - All rights reserved

To recap this process:

DispatcherServlet receives a request whose URL pattern is /user.do.

DispatcherServlet consults BeanNameUrlHandlerMapping to find a controller whose bean name is /user.do; it finds the UserSummaryController bean.

DispatcherServlet dispatches the request to UserSummaryController for rocessing.

UserSummaryController returns a ModelAndView object with a logical view name of userSummary and a list of users in a property called users.

DispatcherServlet consults its view resolver (configured as nternalResourceViewResolver) to find a view whose logical name is userSummary. Internal-ResourceViewResolver returns the path to /WEB-INF/jsp/userSummary.jsp.

DispatcherServlet forwards the request to the JSP at /WEB-INF/jsp/

userSummary.jsp to render the User Summary page to the user.

Ver 1.0 © 2007 Capgemini - All rights reserved

Mapping requests to controllers

Handler mapping	How it maps requests to controllers
`BeanNameUrlHandlerMapping`	Maps controllers to URLs that are based on the controllers' bean name.
`SimpleUrlHandlerMapping`	Maps controllers to URLs using a property collection defined in the Spring application context.
`ControllerClassNameHandlerMapping`	Maps controllers to URLs by using the controller's class name as the basis for the URL.
`CommonsPathMapHandlerMapping`	Maps controllers to URLs using source-level metadata placed in the controller code. The metadata is defined using Jakarta Commons Attributes (http://jakarta.apache.org/commons/attributes).

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC

Using SimpleUrlHandlerMapping

```
<bean id="simpleUrlMapping" class="org.springframework.web.servlet.handler.
SimpleUrlHandlerMapping">
<property name="mappings">
<props>
<prop key="/user.do">userSummary</prop>
<prop key="/createUser.do">createUser</prop>
</props>
</property>
</bean>
<bean id=" userSummary" class="com.spring.web.UserSummaryController"/>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Using ControllerClassNameHandlerMapping

```
<bean id="urlMapping"
class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" /><bean id=" userSummary"
class="com.spring.web.UserSummaryController"/>
```

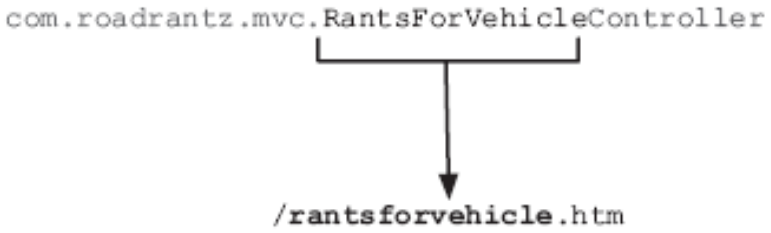


Figure 13.5 `ControllerClassNameHandler-Mapping` maps a request to a controller by stripping `Controller` from the end of the class name and normalizing it to lowercase.

Ver 1.0 © 2007 Capgemini - All rights reserved

Controller

Receive requests from **DispatcherServlet** and coordinate business functionality

Implement the Controller interface

public ModelAndView handleRequest(HttpServletRequest request,

HttpServletResponse response) throws Exception;

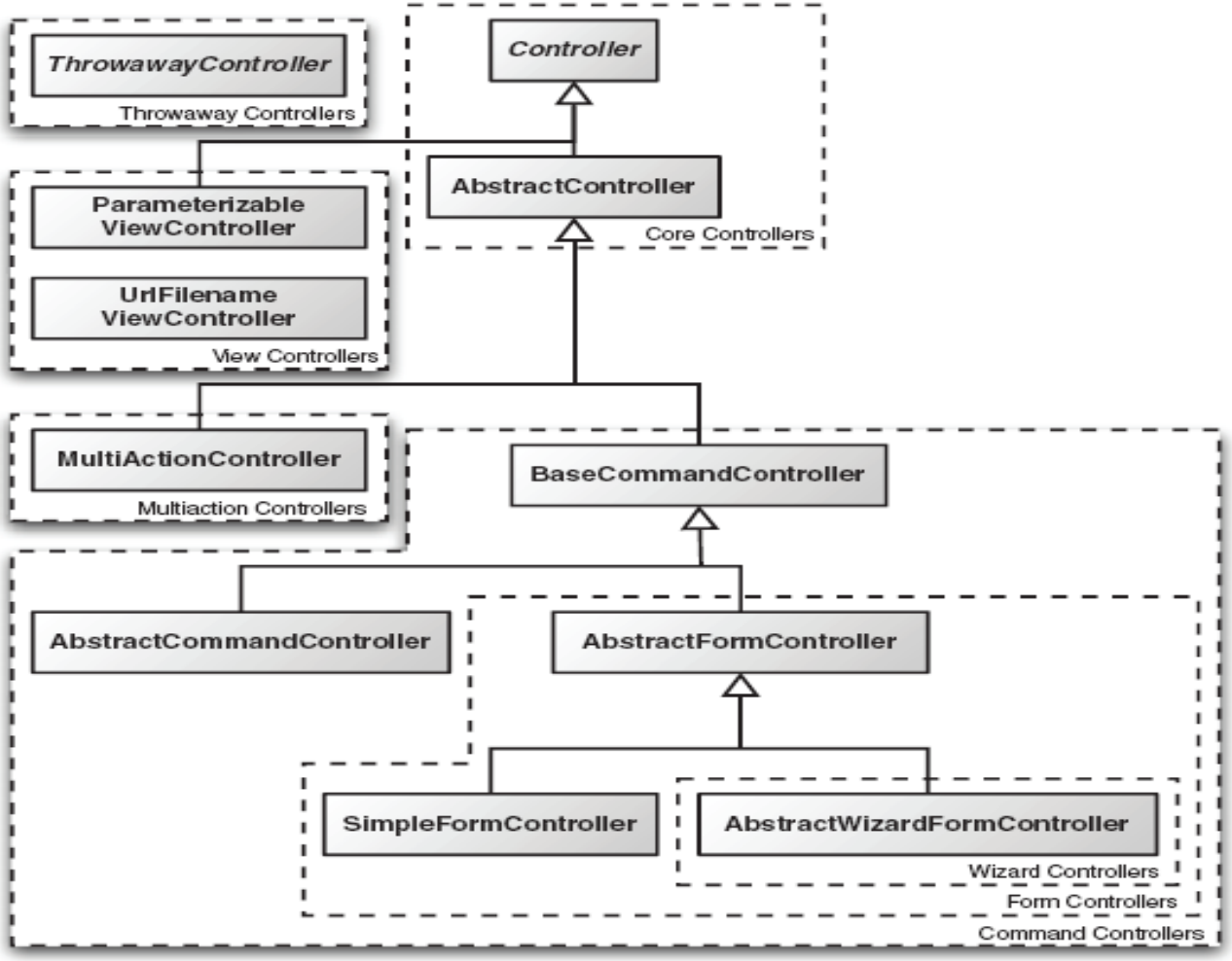
Return instance of **ModelAndView** to **DispatcherServlet**

ModelAndView contains the model (a **Map**) and either a logical view name, or an implementation of the **View** interface

Ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC

Handling requests with controllers



ver 1.0 © 2007 Capgemini - All rights reserved

Spring MVC's selection of controller classes

Controller type	Classes	Useful when...
View	`ParameterizableViewController` `UrlFilenameViewController`	Your controller only needs to display a static view—no processing or data retrieval is needed.
Simple	`Controller (interface)` `AbstractController`	Your controller is extremely simple, requiring little more functionality than is afforded by basic Java servlets.
Throwaway	`ThrowawayController`	You want a simple way to handle requests as commands (in a manner similar to WebWork `Actions`).
Multiaction	`MultiActionController`	Your application has several actions that perform similar or related logic.

Ver 1.0 © 2007 Capgemini - All rights reserved

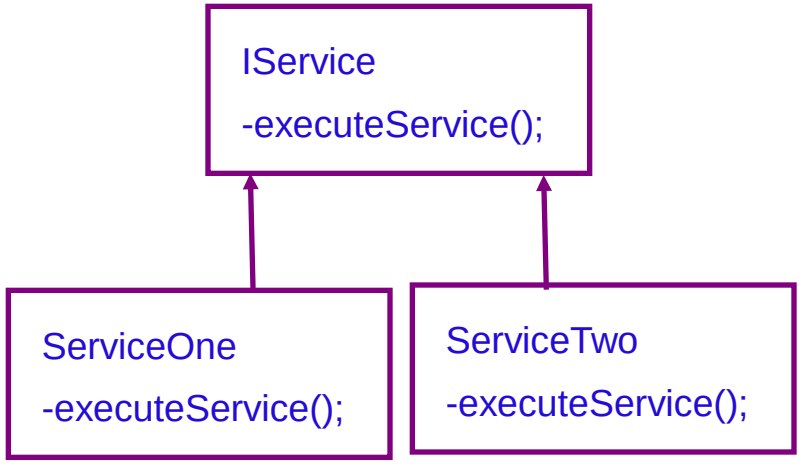
Spring MVC's selection of controller classes (continued ..)

Controller type	Classes	Useful when...
Command	`BaseCommandController` `AbstractCommandController`	Your controller will accept one or more parameters from the request and bind them to an object. Also capable of performing parameter validation.
Form	`AbstractFormController` `SimpleFormController`	You need to display an entry form to the user and also process the data entered into the form.
Wizard	`AbstractWizardFormController`	You want to walk your user through a complex, multipage entry form that ultimately gets processed as a single form.

Ver 1.0 © 2007 Capgemini - All rights reserved

Inversion of Control / Dependency injection

```
public class ConsumerBD {
    IService consumerService = null;

    public void setConsumerService(IService consumerService) {
        this.consumerService = consumerService;
    }

    public void execute(){
        consumerService.executeService();
    }
}
```

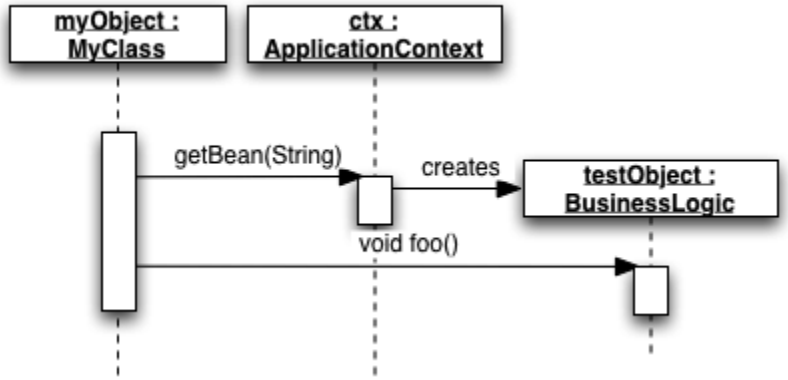



```
<bean id="consumerBD" class="com.spring.beans.ConsumerBD">
    <property name="consumerService" ref="serviceTwo" />
</bean>

<bean id="serviceOne" class="com.spring.beans.ConsumerServiceOne">
    <property name="dataAccessObject"><ref bean="dataAccessObject"/></property>
    <property name="exampleParam"><value>10</value></property>
</bean>

<bean id="serviceTwo" class="com.spring.beans.ConsumerServiceTwo">
    <property name="dataAccessObject"><ref bean="dataAccessObject"/></property>
    <property name="exampleParam"><value>10</value></property>
</bean>
```

Ver 1.0 © 2007 Capgemini - All rights reserved

Inversion of Control / Dependency injection

```
public class SpringRunner {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext appContext =
            new ClassPathXmlApplicationContext("/applicationContext.xml");
        ConsumerBD consumerBD = (ConsumerBD)appContext.getBean("consumerBD");
        consumerBD.execute();
    }
}
```



Ver 1.0 © 2007 Capgemini - All rights reserved