



Welcome to Core Spring

A 4-day bootcamp that trains you how to use the Spring Framework to create well-designed, testable business applications

How You will Benefit

- Learn to use Spring for web and other applications
- Gain hands-on experience
 - 50/50 presentation and labs
- Access to SpringSource professionals



Logistics

- Participants list
- Working hours
- Lunch
- Toilets
- Fire alarms
- Emergency exits
- Internet access
- Other questions?



LOGISTICS

A green rounded rectangular button with the word "LOGISTICS" in white, bold, sans-serif capital letters. The button has a thin black border and a slight shadow underneath.

Covered in this section



- **Agenda**
- SpringSource, a division of VMware

Official SpringSource
Training Only
Do Not Distribute

Course Agenda: Day 1



- Introduction to Spring
- Using Spring to configure an application
- Simplifying XML-based configuration
- Annotation-based dependency injection
- Java-based dependency injection

1

Course Agenda: Day 2

- Understanding the bean life-cycle
- Testing a Spring-based application using multiple profiles
- Introducing data access with Spring
- Adding behavior to an application using aspects
- Simplifying JDBC-based data access

2

Course Agenda: Day 3



- Driving database transactions in a Spring environment
- Introducing object-to-relational mapping (ORM)
- Working with JPA in a Spring environment
- Effective web application architecture
- Getting started with Spring MVC

3

Course Agenda: Day 4



- Securing web applications with Spring Security
- Understanding Spring's remoting framework
- Simplifying message applications with Spring JMS
- Adding manageability to an application with Spring JMX

4

Covered in this section



- Agenda
- **SpringSource, a division of VMware**

Official SpringSource
Training Only
Do Not Distribute

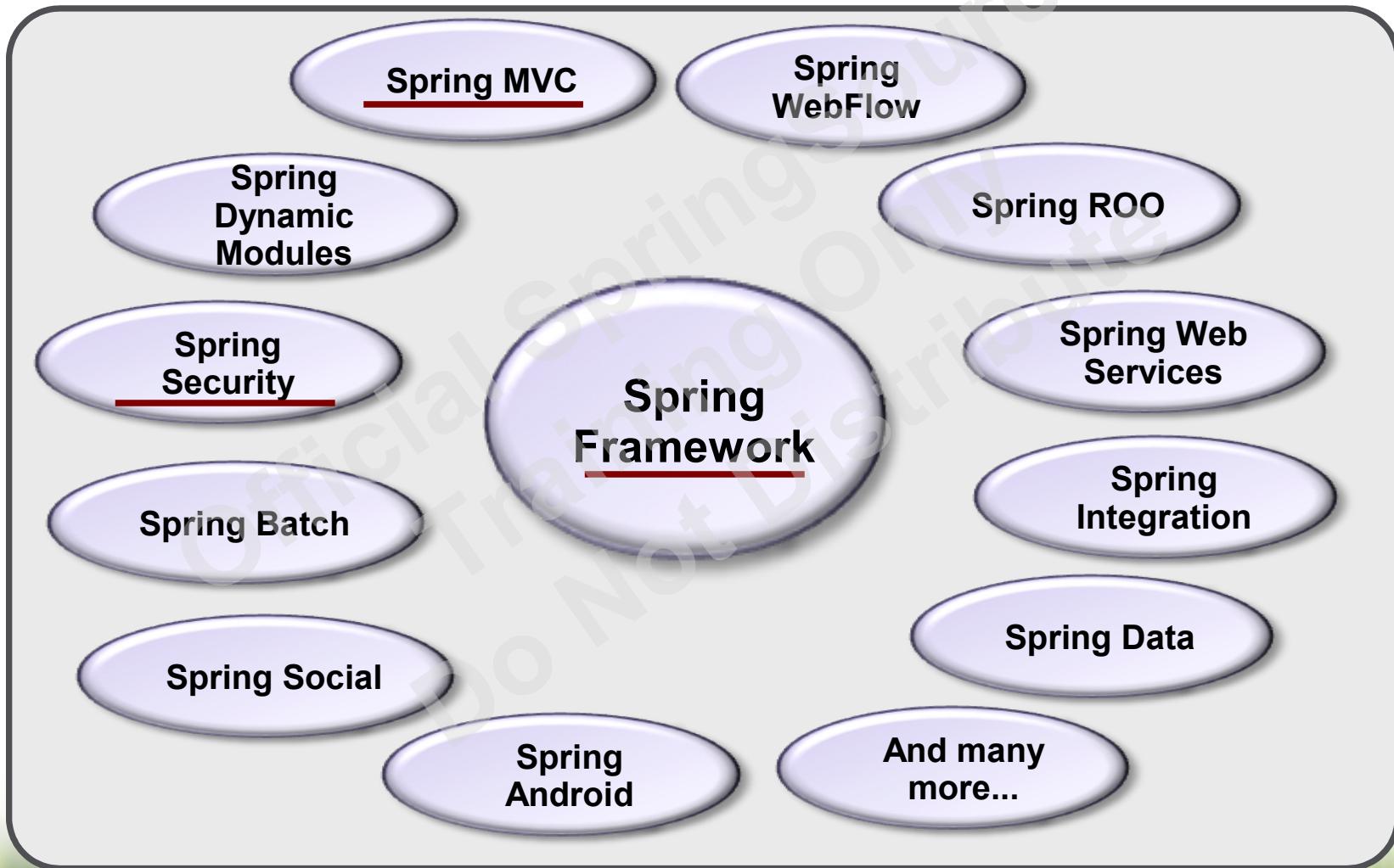
SpringSource, a division of VMware



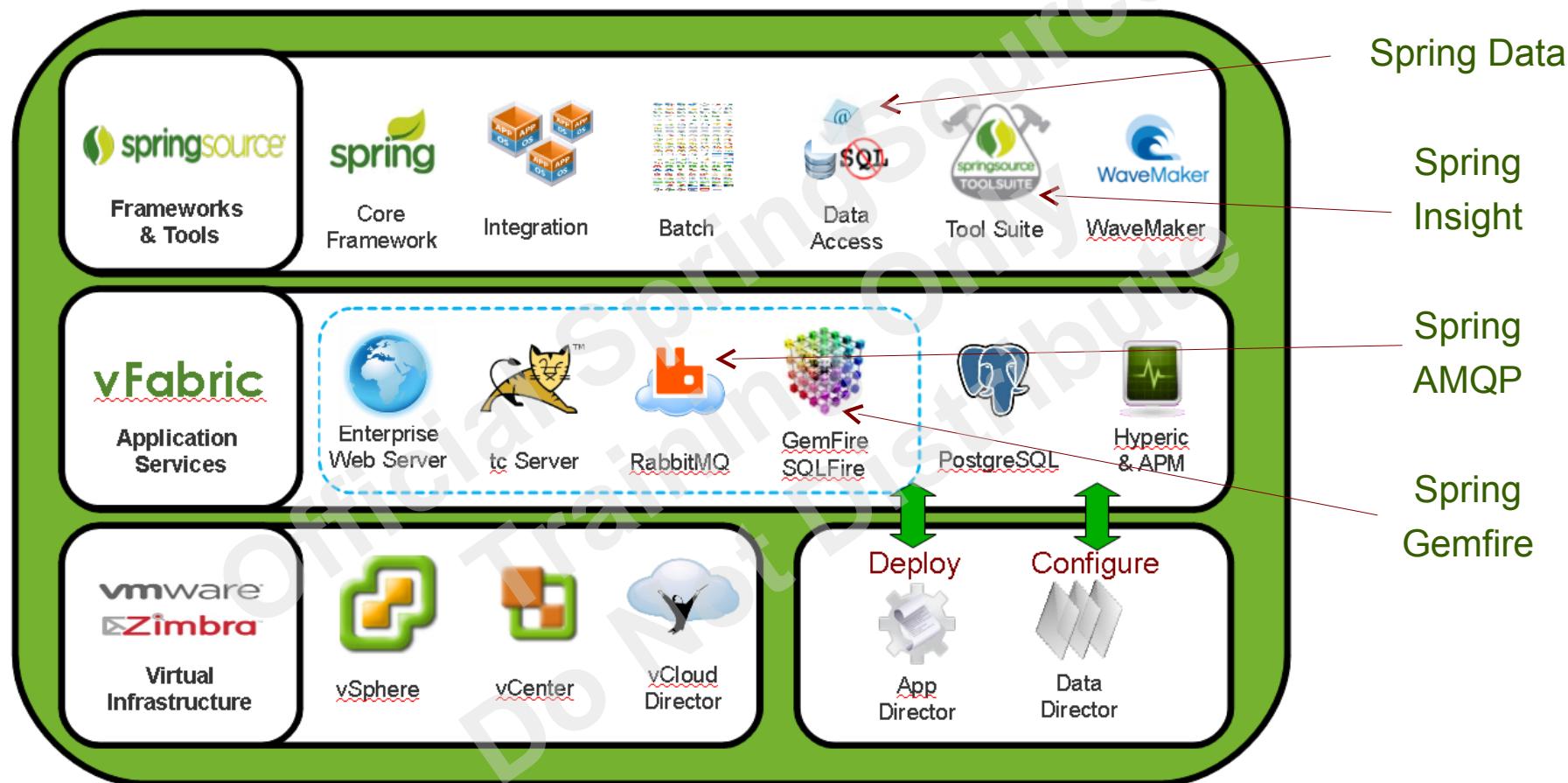
- SpringSource, the company behind Spring
 - acquired by VMware in Oct 2009
- Spring projects key to VMware's cloud strategy
 - Virtualize your Java Apps
 - vFabric: tc Server + Elastic Memory for Java
 - Save license costs
 - Deploy to private, public, hybrid clouds
 - Flexible deployment strategy
 - Handle peak loads cost-effectively
 - Charge per-use, ITSM



The Spring projects



VMWare and Spring



VMware's Goal: create the *best* place to run a Spring application

Open Source contributions



- Spring technologies
 - VMware develops 95% of the code of the Spring framework
 - Also leaders on the other Spring projects (Batch, Security, Web Flow...)
- Tomcat
 - 60% of code commits in the past 2 years
 - 80% of bug fixes
- Apache httpd
- Hyperic
- Groovy/Grails
- Rabbit MQ
- ...



Covered in this section



- Agenda
- SpringSource, a division of VMware

Let's get on with the course..!





Overview of the Spring Framework

Introducing Spring in the context of enterprise application architecture

Topics in this session



- Goal of the Spring Framework
- Spring's role in enterprise application architecture
 - Core support
 - Web application development support
 - Enterprise application development support
- Spring Framework history

Goal of the Spring Framework



- Provide comprehensive infrastructural support for developing enterprise Java™ applications
 - Spring deals with the plumbing
 - So you can focus on solving the domain problem

Spring's Support (1)



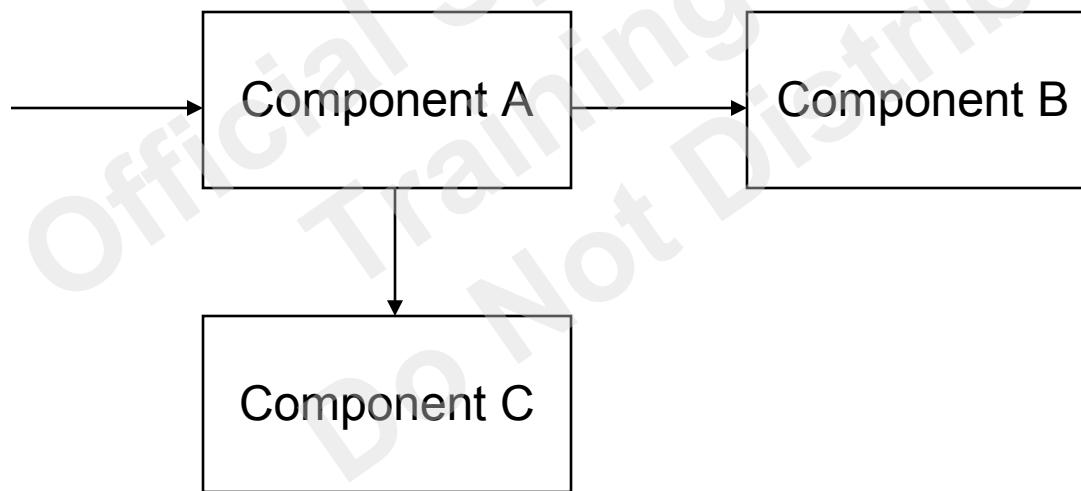
- Core support
 - Application Configuration
 - Enterprise Integration
 - Testing
 - Data Access

Official Springsource
Training Only
Do Not Distribute

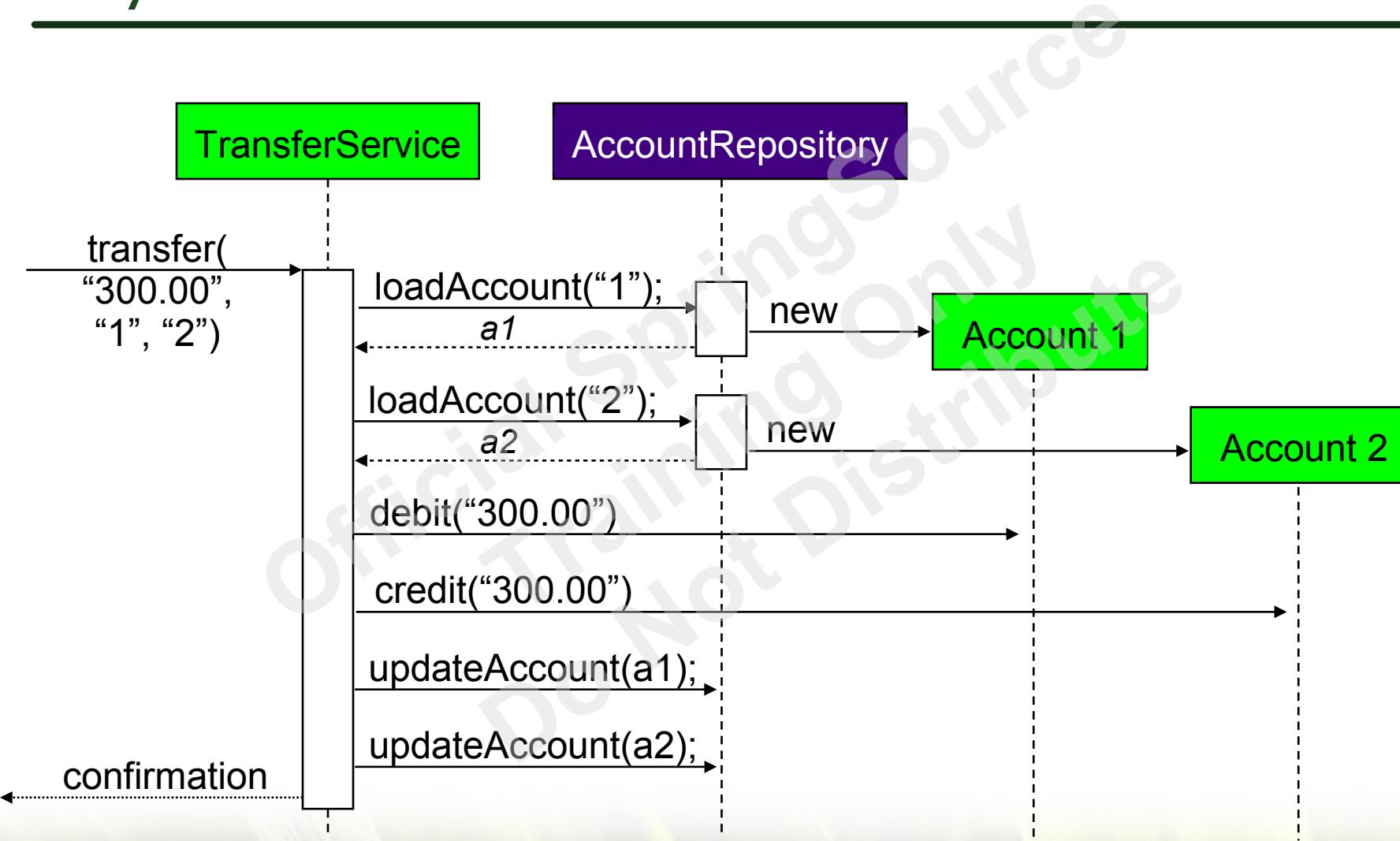
Application Configuration



- A typical application system consists of several parts working together to carry out a use case



Example: A Money Transfer System



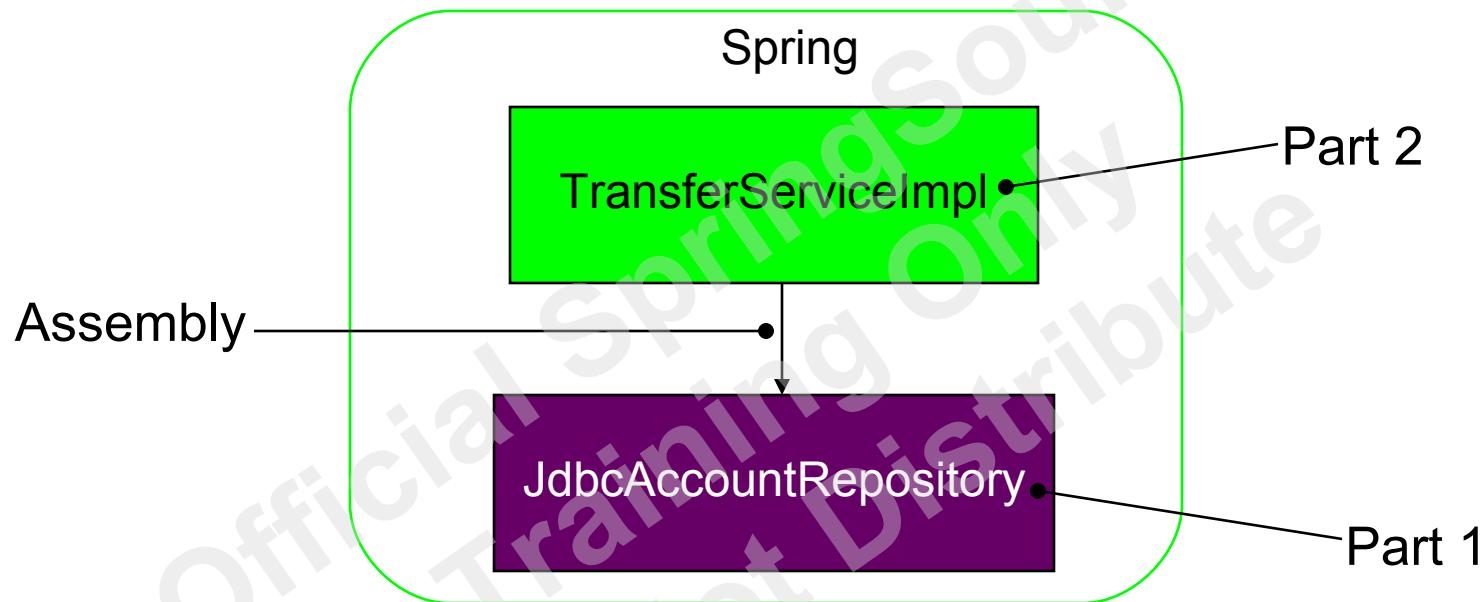
Spring's Configuration Support



- Spring provides support for assembling such an application system from its parts
 - Parts do not worry about finding each other
 - Any part can easily be swapped out

Official Spring Source
Training
Do Not Distribute

Money Transfer System Assembly



- ```
(1) new JdbcAccountRepository(...);
(2) new TransferServiceImpl();
(3) service.setAccountRepository(repository);
```

# Parts are Just Plain Old Java Objects



```
public class JdbcAccountRepository implements
 AccountRepository {
 ...
}
```

Implements a service interface

Part 1

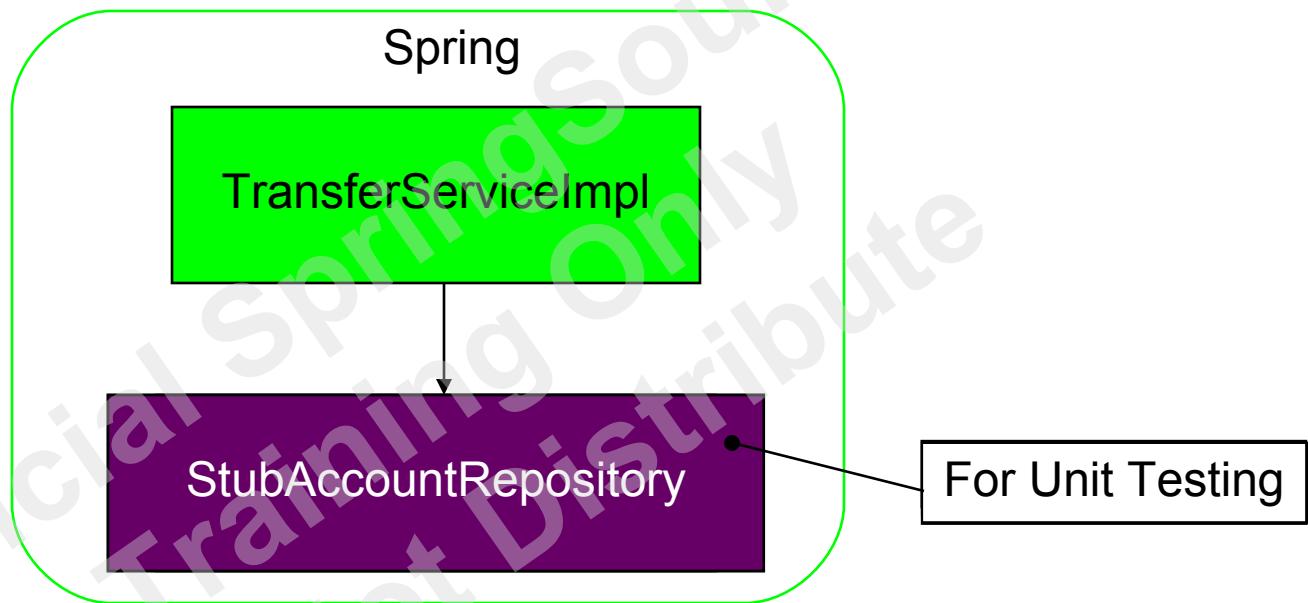
```
public class TransferServiceImpl implements TransferService {
 private AccountRepository accountRepository;

 public void setAccountRepository(AccountRepository ar) {
 accountRepository = ar;
 }
 ...
}
```

Part 2

Depends on service interface;  
conceals complexity of implementation;  
allows for swapping out implementation

# Swapping Out Part Implementations



- ```
(1) new StubAccountRepository();
(2) new TransferServiceImpl();
(3) service.setAccountRepository(repository);
```

Enterprise Integration



- Enterprise applications do not work in isolation
- They require enterprise services and resources
 - Database Transactions
 - Security
 - Messaging
 - Remote Access
 - Caching

Spring Enterprise Integration



- Spring helps you integrate powerful enterprise services into your application
 - While keeping your application code simple and testable
- Plugs into all Java EE™ application servers
 - While capable of standalone usage

Testing



- Automated testing is essential
- Spring enables unit testability
 - Decouples objects from their environment
 - Making it easier to test each piece of your application in isolation
- Spring provides system testing support
 - Helps you test all the pieces together

Enabling Unit Testing



```
import static org.junit.Assert.*;  
  
public class TransferServiceImplTests {  
    private TransferServiceImpl transferService;  
  
    @Before public void setUp() {  
        AccountRepository repository = new StubAccountRepository();  
        transferService = new TransferServiceImpl(repository);  
    }  
}
```

Minimizing dependencies increases testability

```
@Test public void transferMoney() {  
    TransferConfirmation confirmation =  
        transferService.transfer(new MonetaryAmount("300.00"), "1", "2");  
    assertEquals(new MonetaryAmount("200.00"),  
                confirmation.getNewBalance());  
}
```

Testing logic in isolation measures unit design
and implementation correctness

Accessing Data



- Most enterprise applications access data stored in a relational database
 - To carry out business functions
 - To enforce business rules

Official SpringSource
Training Only
Do Not Distribute

Spring Data Access



- Spring makes data access easier to do effectively
 - Manages resources for you
 - Provides API helpers
 - Supports all major data access technologies
 - JDBC
 - Hibernate
 - JPA (versions 1 and 2)
 - MyBatis (iBATIS)
 - ...



See also Spring Data <http://www.springsource.org/spring-data>

Spring's Support (2)

- Web application development support
 - Struts integration
 - JSF Integration
 - Spring MVC and Web Flow
 - Handle user actions
 - Validate input forms
 - Enforce site navigation rules
 - Manage conversational state
 - Render responses (HTML, XML, etc)
 - REST and AJAX support



Spring-Web course offers 4 days on Spring's Web support

Spring's Support (3)



- Enterprise application development support
 - Developing web services
 - Adding manageability
 - Integrating messaging infrastructures
 - Securing services and providing object access control
 - Tasks and Job Scheduling
 - Spring Batch (not this course)



These topics and more: *Enterprise Integration with Spring* course

Spring Framework History



- Spring 3.2 (released Dec 2012)
 - Env. + Profiles, @Cacheable, @EnableXXX ...
 - Supports Java 7, Hibernate 4, Servlet 3
- Spring 3.0 (released Dec 2009, currently 3.0.7)
 - Requires Java 1.5+ and JUnit 4.7+
 - REST support, JavaConfig, SpEL, more annotations
- Spring 2.5 (released Nov 2007, currently 2.5.6)
 - Requires Java 1.4+ and supports JUnit 4
 - Annotation DI, @MVC controllers, XML namespaces
 - From 2.0: XML simplification, async JMS, JPA, AspectJ support



Spring can be downloaded here: <http://www.springsource.org/download>



LAB

Developing an Application from Plain Java
Objects



Dependency Injection using XML

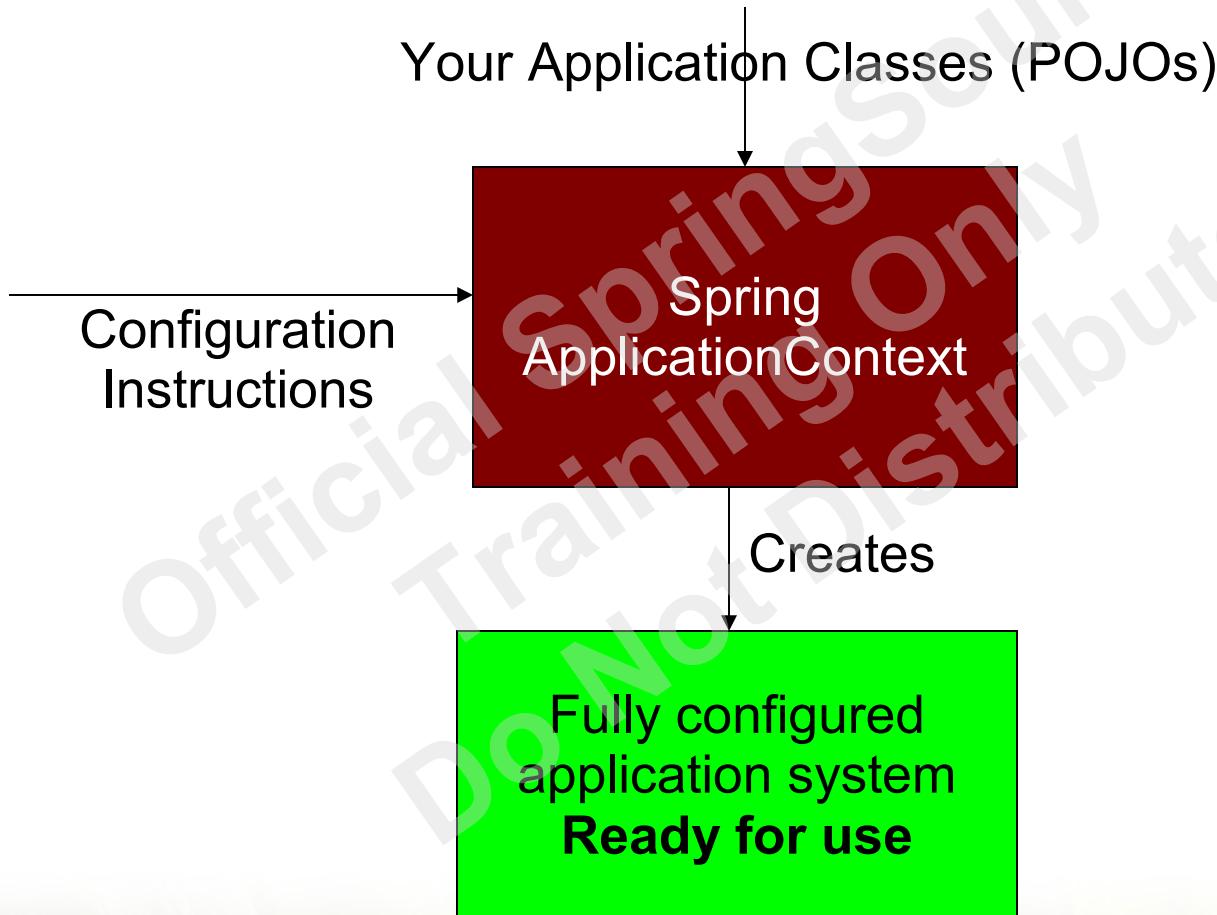
Introducing the Spring Application Context and
Spring's XML-based configuration language

<?xml?>

Topics in this session

- **Spring quick start**
- Writing bean definitions
- Creating an application context
- Bean scope and FactoryBean elements
- Lab
- Advanced Topics

How Spring Works



Your Application Classes



```
public class TransferServiceImpl implements TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```

Needed to perform money transfers between accounts

```
public class JdbcAccountRepository implements AccountRepository {  
    public JdbcAccountRepository(DataSource ds) {  
        this.dataSource = ds;  
    }  
    ...  
}
```

Needed to load accounts from the database

Configuration Instructions



```
<beans>

    <bean id="transferService" class="com.acme.TransferServiceImpl">
        <constructor-arg ref="accountRepository" />
    </bean>

    <bean id="accountRepository" class="com.acme.JdbcAccountRepository">
        <constructor-arg ref="dataSource" />
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="org.postgresql.Driver" />
        <property name="URL" value="jdbc:postgresql://localhost/transfer" />
        <property name="user" value="transfer-app" />
        <property name="password" value="secret45" />
    </bean>

</beans>
```

Creating and Using the Application



```
// Create the application from the configuration
ApplicationContext context =
    new ClassPathXmlApplicationContext("application-config.xml");

// Look up the application service interface
TransferService service =
    (TransferService) context.getBean("transferService");

// Use the application
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```

Accessing a bean

- Multiple ways

```
ApplicationContext context = new ClassPathXmlApplicationContext(...);
```

```
// Classic way: cast is needed
```

```
ClientService service1 = (ClientService) context.getBean("clientService");
```

```
// Since Spring 3.0: no more cast, type is a method param
```

```
ClientService service2 = context.getBean("clientService", ClientService.class);
```

```
// New in 3.0: No need for bean id if type is unique
```

```
ClientService service3 = context.getBean(ClientService.class );
```

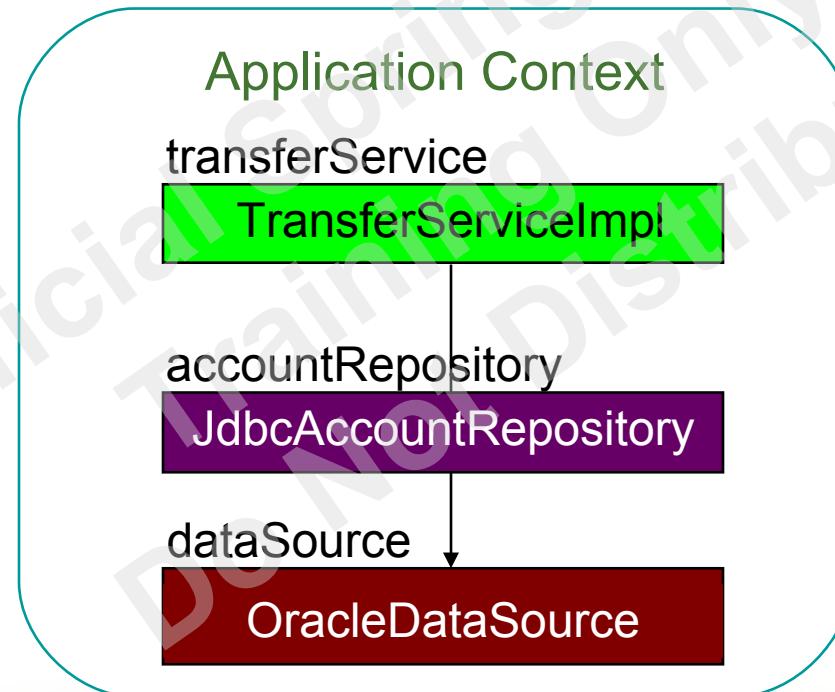
No cast in Spring 3.x

From Spring 3.0

Inside the Spring Application Context



```
// Create the application from the configuration  
ApplicationContext context =  
    new ClassPathXmlApplicationContext("application-config.xml");
```



Quick Start Summary



- Spring manages the lifecycle of the application
 - All beans are *fully* initialized before use
- Beans are always created in the right order
 - Based on their dependencies
- Each bean is bound to a unique id
 - The id reflects the service or *role* the bean provides to clients
 - Bean id *should not* contain implementation details

Topics in this session

- Spring quick start
- **Writing bean definitions**
- Creating an application context
- Bean scope and FactoryBean elements
- Lab
- Advanced Topics

Constructor Injection configuration



- One parameter

```
<bean id="transferService" class="com.acme.TransferServiceImpl">
    <constructor-arg ref="accountRepository"/>
</bean>

<bean id="accountRepository" class="com.acme.AccountRepositoryImpl"/>
```

- Multiple parameters

```
<bean id="transferService" class="com.acme.TransferServiceImpl">
    <constructor-arg ref="accountRepository"/>
    <constructor-arg ref="customerRepository"/>
</bean>

<bean id="accountRepository" class="com.acme.AccountRepositoryImpl"/>
<bean id="customerRepository" class="com.acme.CustomerRepositoryImpl"/>
```

Parameters injected according to their type

Constructor Injection under the hood



```
<bean id="service" class="com.acme.ServiceImpl">
    <constructor-arg ref="repository"/>
</bean>

<bean id="repository" class="com.acme.RepositoryImpl"/>
```

Equivalent to:

```
RepositoryImpl repository = new RepositoryImpl();
ServiceImpl service = new ServiceImpl(repository);
```

ApplicationContext

service -> instance of ServiceImpl
repository -> instance of RepositoryImpl

Setter Injection

```
<bean id="service" class="com.acme.ServiceImpl">  
    <property name="repository" ref="repository"/>  
</bean>
```

Convention: implicitly refers to method `setRepository(...)`

```
<bean id="repository" class="com.acme.RepositoryImpl"/>
```

Equivalent to:

```
RepositoryImpl repository = new RepositoryImpl();  
ServiceImpl service = new ServiceImpl();  
service.setRepository(repository);
```

ApplicationContext

`service` -> instance of `ServiceImpl`
`repository` -> instance of `RepositoryImpl`

When to Use Constructors vs. Setters?



- Spring supports both
- You can mix and match

Official Springsource
Training Only
Do Not Distribute

The Case for Constructors



- Enforce mandatory dependencies
- Promote immutability
 - Assign dependencies to final fields
- Concise for programmatic usage
 - Creation and injection in one line of code

The Case for Setters



- Allow optional dependencies and defaults
- Have descriptive names
- Inherited automatically
- Cyclic references

Official SpringSource
Training Only
Do Not Distribute

General Recommendations



- Follow standard Java design guidelines
 - Use constructors to set required properties
 - Use setters for optional or those with default values
- Some classes are designed for a particular injection strategy
 - In that case go with it, do not fight it
- Be consistent above all



Combining Constructor and Setter Injection



```
<bean id="service" class="com.acme.ServiceImpl">
    <constructor-arg ref="required" />
    <property name="optional" ref="optional" />
</bean>

<bean id="required" class="com.acme.RequiredImpl" />
<bean id="optional" class="com.acme.OptionallImpl" />
```

Equivalent to:

```
RequiredImpl required = new RequiredImpl();
OptionallImpl optional = new OptionallImpl();
ServiceImpl service = new ServiceImpl(required);
service.setOptional(optional);
```

Injecting Scalar Values



```
<bean id="service" class="com.acme.ServiceImpl">  
    <property name="stringProperty" value="foo" />  
</bean>
```

Equivalent

```
<property name="stringProperty">  
    <value>foo</value>  
</property>
```

```
public class ServiceImpl {  
    public void setStringProperty(String s) { ... }  
    // ...  
}
```

```
ServiceImpl service = new ServiceImpl();  
service.setStringProperty("foo");
```

ApplicationContext

service -> instance of ServiceImpl

Automatic Value Type Conversion



```
<bean id="service" class="com.acme.ServiceImpl">  
    <property name="intProperty" value="29" />  
</bean>
```

```
public class ServiceImpl {  
    public void setIntProperty(int i) { ... }  
    // ...  
}
```

Equivalent to:

```
ServiceImpl service = new ServiceImpl();  
int value = 29;  
service.setIntProperty(value);
```

- Spring can convert:
- Numeric types
- BigDecimal,
- boolean: “true”, “false”
- Date
- Locale
- Resource

ApplicationContext

service -> instance of ServiceImpl

Topics in this session

- Spring quick start
- Writing bean definitions
- **Creating an application context**
- Bean scope and FactoryBean elements
- Lab
- Advanced Topics

Creating a Spring Application Context



- Spring application contexts can be bootstrapped in any environment, including
 - JUnit system test
 - Web application
 - Standalone application
- Loadable with bean definitions from files
 - In the class path
 - On the local file system
 - At an environment-relative resource path

Example: Using an Application Context Inside a JUnit System Test



```
import static org.junit.Assert.*;  
  
public void TransferServiceTests {  
    private TransferService service;  
  
    @Before public void setUp() {  
        // Create the application from the configuration  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("application-config.xml");  
        // Look up the application service interface  
        service = (TransferService) context.getBean("transferService");  
    }  
  
    @Test public void moneyTransfer() {  
        Confirmation receipt =  
            service.transfer(new MonetaryAmount("300.00"), "1", "2"));  
        assertEquals(receipt.getNewBalance(), "500.00");  
    }  
}
```

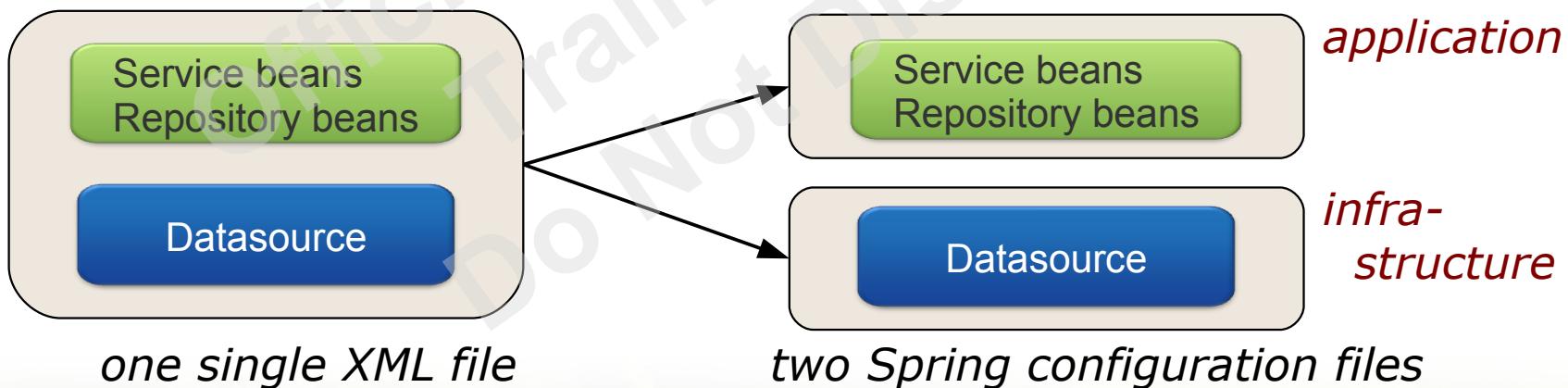
Bootstraps the system to test

Tests the system

Creating a Spring Application Context from Multiple Files



- A context can be configured from multiple files
 - Can partition bean definitions into logical groups
- *Best practice:* separate out “application” beans from “infrastructure” beans
 - Infrastructure often changes between environments



Mixed Configuration

```
<beans>
```

```
  <bean id="transferService" class="com.acme.TransferServiceImpl">
    <constructor-arg ref="accountRepository" />
  </bean>
```

application beans

```
  <bean id="accountRepository" class="com.acme.JdbcAccountRepository">
    <constructor-arg ref="dataSource" />
  </bean>
```

Coupled to a local Postgres environment

```
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="URL" value="jdbc:postgresql://localhost/transfer" />
    <property name="user" value="transfer-app" />
    <property name="password" value="secret45" />
  </bean>
```

infrastructure bean

```
</beans>
```

Partitioning Configuration



```
<beans>
  <bean id="transferService" class="com.acme.TransferServiceImpl">
    <constructor-arg ref="accountRepository" />
  </bean>

  <bean id="accountRepository" class="com.acme.JdbcAccountRepository">
    <constructor-arg ref="dataSource" />
  </bean>
</beans>
```

application-config.xml

Now substitutable for other environments

```
<beans>
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="URL" value="jdbc:postgresql://localhost/transfer" />
    <property name="user" value="transfer-app" />
    <property name="password" value="secret45" />
  </bean>
</beans>
```

test-infrastructure-config.xml

Bootstrapping in Each Environment



- In the test environment

```
// Create the application from the configuration  
ApplicationContext context =  
    new ClassPathXmlApplicationContext(  
        "application-config.xml", "test-infrastructure-config.xml" );
```

- In the production environment

```
// Create the application from the configuration  
ApplicationContext context =  
    new ClassPathXmlApplicationContext(  
        "application-config.xml", "prod-infrastructure-config.xml");
```



From Spring 3.1, Bean Profiles provide more options when working with various environments (to be discussed later in this course)

Spring's Flexible Resource Loading Mechanism



- ApplicationContext implementations have *default resource loading rules*

```
new ClassPathXmlApplicationContext("com/acme/application-config.xml");
```

\$CLASSPATH/com/acme/application-config.xml

```
new FileSystemXmlApplicationContext("C:\\\\etc\\\\application-config.xml");  
// absolute path: C:\\etc\\application-config.xml
```

```
new FileSystemXmlApplicationContext("./application-config.xml");  
// path relative to the JVM working directory
```



XmlWebApplicationContext is also available

- The path is relative to the Web application
- Usually created indirectly via a declaration in web.xml

Working with prefixes



- Default rules can be overridden

```
// Create the application from the configuration  
ApplicationContext context =  
    new ClassPathXmlApplicationContext(  
        "config/dao-config.xml", "config/service-config.xml",  
        "file:oracle-infrastructure-config.xml" );
```

prefix

- Various prefixes
 - classpath:
 - file:
 - http:



These prefixes can be used anywhere Spring needs to deal with resources (not just in constructor args to application context)

Topics in this session

- Spring quick start
- Writing bean definitions
- Creating an application context
- **Bean scope and FactoryBean elements**
- Lab
- Advanced Topics

Bean scope: default

service1 == service2

- Default scope is *singleton*

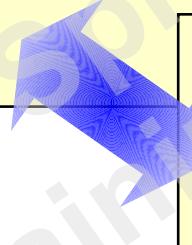
```
<bean id="accountService"
      class="com.acme.AccountServiceImpl">
</bean>
```

Spring config

```
<bean id="accountService"
      class="com.acme.AccountServiceImpl"
      scope="singleton">
</bean>
```

Spring config

One single instance



```
AccountService service1 = (AccountService) context.getBean("accountService");
AccountService service2 = (AccountService) context.getBean("accountService");
```

Bean scope: prototype

service1 != service2

- **scope="prototype"**
 - New instance created every time bean is referenced

```
<bean id="accountService" class="com.acme.AccountServiceImpl"  
      scope="prototype">  
</bean>
```

Spring config

```
AccountService service1 = (AccountService) context.getBean("accountService");  
AccountService service2 = (AccountService) context.getBean("accountService");
```

2 instances

Available Scopes

singleton

A single instance is used

prototype

A new instance is created each time
the bean is referenced

session

A new instance is created once per
user session - **web environment**

request

A new instance is created once per
request - **web environment**

**custom
scope
name**

You define your own rules and a new
scope name - ***advanced feature***

Advanced Bean Instantiation



- How can Spring instantiate the following?
 - Classes with private constructors (such as Singleton pattern)
 - Objects from Factories

```
public class AccountServiceSingleton implements AccountService {  
    private static AccountServiceSingleton inst = new AccountServiceSingleton();  
  
    private AccountServiceSingleton() { ... }  
  
    public static AccountService getInstance() {  
        // ...  
        return inst;  
    }  
}
```

Implementing a FactoryBean



- 2 ways
 - XML declaration
 - Use *factory-method* attribute
 - No need to update Java class
 - Implement *FactoryBean* interface
 - Spring can auto-detect that FactoryBean interface is implemented
 - Bean is declared as a regular bean in xml config
 - *FactoryBean* interface is widely used within the Spring framework

The factory-method attribute



- Non-intrusive
 - Useful for existing Singletons or Factories

```
public class AccountServiceSingleton implements AccountService {  
    ...  
    public static AccountService getInstance() { // ... }  
}
```

```
<bean id="accountService" class="com.acme.AccountServiceSingleton"  
      factory-method="getInstance" />
```

Spring configuration

```
AccountService service1 = (AccountService) context.getBean("accountService");  
AccountService service2 = (AccountService) context.getBean("accountService");
```

Test class

Spring uses `getInstance()` method – so
service1 and service2 point to the *same* object

The FactoryBean interface



- If implemented, no need to use factory-method attribute in xml

```
public class AccountServiceFactoryBean
    implements FactoryBean<AccountService>
{
    public AccountService getObject() throws Exception {
        //...
        return accountService;
    }
    //...
}
```

```
<bean id="accountService"
      class="com.acme.AccountServiceFactoryBean" />
```

The FactoryBean interface



- Beans implementing *FactoryBean* are *auto-detected*
- Dependency injection using the factory bean id causes *getObject()* to be invoked transparently

```
<bean id="accountService"
      class="com.acme.AccountServiceFactoryBean"/>

<bean id="customerService" class="com.acme.CustomerServiceImpl">
    <property name="service" ref="accountService" />
</bean>
```

getObject() called
by Spring
automatically

EmbeddedDatabaseFactoryBean



- Common example of a *FactoryBean*
 - Part of the Spring framework
 - Used to create an in-memory database

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.embedded.  
                           EmbeddedDatabaseFactoryBean">  
    <property name="databasePopulator" ref="populator"/>  
</bean>  
  
<bean id="populator" class="org.springframework.jdbc.datasource.init.  
                           ResourceDatabasePopulator">  
    <property name="scripts">  
        <list>  
            <value>classpath:testdb/setup.sql</value>  
        </list>  
    </property>  
</bean>
```

FactoryBean

Populate with test-data

FactoryBeans in Spring



- EmbeddedDatabaseFactoryBean
- JndiObjectFactoryBean
 - One option for looking up JNDI objects
- FactoryBeans for creating remoting proxies
- FactoryBeans for configuring data access technologies like JPA, Hibernate or MyBatis

Dependency Injection Summary



- Your object is handed what it needs to work
 - Frees it from the burden of resolving its dependencies
 - Simplifies your code, improves code reusability
- Promotes programming to interfaces
 - Conceals implementation details of dependencies
- Improves testability
 - Dependencies easily stubbed out for unit testing
- Allows for centralized control over object lifecycle
 - Opens the door for new possibilities



LAB

Using Spring to Configure an Application

Topics in this session

- Spring quick start
- Writing bean definitions
- Creating an application context
- Bean scope and FactoryBean elements
- Lab
- **Advanced Topics**
 - **Constructor Args**

More on Constructor Args



- Constructor args matched by type
 - `<constructor-arg>` elements can be in *any* order
 - When ambiguous: indicate order with *index*

```
class MailService {  
    public MailService(int maxEmails, String email) { ... }
```

Both look like Strings to XML

```
<bean name="example" class="com.app.MailService">  
    <constructor-arg index="0" value="10000"/>  
    <constructor-arg index="1" value="foo@foo.com"/>  
</bean>
```

Index from zero

Using Constructor Names



- Constructor args can have names
 - Since Spring 3.0 they can be used for arg matching
 - BUT: need to compile with debug-symbols enabled
 - OR: Use @java.beans.ConstructorProperties

```
class MailService {  
    @ConstructorProperties( { "maxEmails", "email" } )  
    public MailService(int maxEmails, String email) { ... }
```

Specify arg names *in order*

```
<bean name="example" class="com.app.MailService">  
    <constructor-arg name="maxEmails" value="10000"/>  
    <constructor-arg name="email" value="foo@foo.com"/>  
</bean>
```

XML Dependency Injection

Advanced features and Best Practices

Techniques for Creating Reusable and
Concise Bean Definitions

A blurred background image of green grass at the bottom of the slide.

<?xml?>

Topics in this session

- **Best Practices**
 - **Making the best use of XML namespaces**
 - STS Wizard, version numbers, 'c' and 'p' namespaces
 - Externalizing values into properties files
 - Working with a high number of conf. Files
 - <import/> tag and wildcards usage
 - Using Bean definition inheritance
- Lab
- Advanced Features
 - Inner beans, collections, SpEL

Default namespace

- You have used the beans namespace already
 - commonly used as the default namespace in Spring

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



dozens of other namespaces are available!

Other namespaces

- Defined for subsets of framework functionality*
 - aop (Aspect Oriented Programming)
 - tx (transactions)
 - security
 - jms
 - mvc
 - ...
- They allow hiding of actual bean definitions
 - Greatly reduce size of bean files (see next slide)

* see <http://www.springframework.org/schema/> for a complete list

Power of Namespaces



- Greatly simplifies Spring configuration
 - Many advanced features of Spring need to declare a large number of beans

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <context:property-placeholder location="db-config.properties" />
    <aop:aspectj-autoproxy />
    <tx:annotation-driven />
</beans>
```

hides 1 bean definition

AOP configuration: hides 5+ bean definitions

Transactions configuration: hides more than 15 bean definitions!



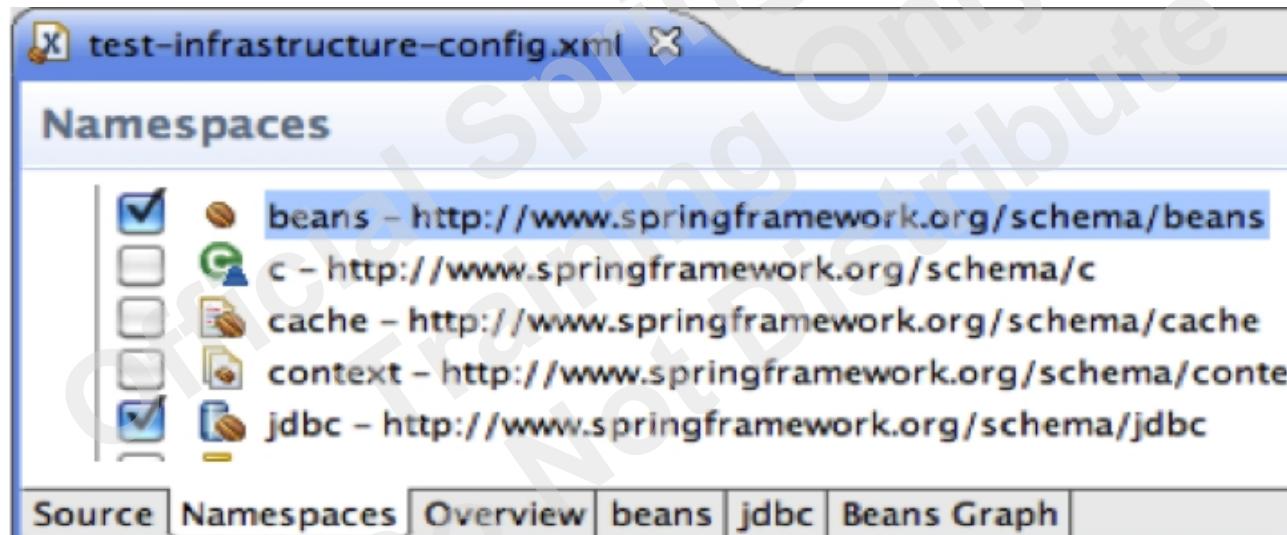
tx and aop namespaces will be discussed later

Adding namespace declaration



- XML syntax is error-prone
 - Use the dedicated STS wizard!

xsi:schemaLocation="..."



Click here and select appropriate namespaces

Schema version numbers



spring-beans-3.1.xsd OR spring-beans.xsd ?

- Common practice: do not use a version number
 - Triggers use of most recent schema version
 - Easier migration
 - Will make it easier to upgrade to the next version of Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">
    <!-- ... -->
</beans>
```

Not
needed!

The *c* and *p* namespaces

- Before

```
<bean id="transferService" class="com.acme.BankServiceImpl">
    <constructor-arg ref="bankRepository" />
    <property name="accountService" ref="accountService" />
    <property name="customerService" ref="customerService" />
</bean>
```

- After

```
<bean id="transferService" class="com.acme.BankServiceImpl"
    c:bankRepository-ref="bankRepository"
    p:accountService-ref="accountService"
    p:customer-service-ref="customerService" />
```

Use camel case
or hyphens



c namespace was introduced in Spring 3.1

The *c* and *p* namespaces

- *c* and *p* namespaces should be declared on top
 - Use '-ref' suffix for references

Namespace declaration

```
<beans xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       ...>
```

```
  <bean id="transferService" class="com.acme.ServiceImpl"
        p:url="jdbc://..." p:service-ref="service" />
</beans>
```

Inject value for property 'url'

Inject reference for bean 'service'

No schemaLocation needed



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

p and c
namespace
definitions

no extra schemaLocation
entry required (no xsd)

'c' and 'p' Pros and Cons



- Pros
 - More concise
 - Well supported in STS
 - CTRL+space works well
- Cons
 - Fairly new feature
 - The 'c' namespace was implemented in Spring 3.1
 - (*the 'p' namespace was introduced in Spring 2.0*)
 - Less widely known than the usual XML configuration syntax



Topics in this session

- **Best Practices**
 - Making the best use of XML namespaces
 - STS Wizard, version numbers, 'c' and 'p' namespaces
 - **Externalizing values into properties files**
 - Working with a high number of conf. Files
 - <import/> tag and wildcards usage
 - Using Bean definition inheritance
- Lab
- Advanced Features
 - Inner beans, collections, SpEL

Externalizing values to a properties file (1)



- Namespace declaration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd" ...>

    <context:property-placeholder ... />

</beans>
```

Context namespace

Externalizing values to a properties file (2)



```
<beans ...>
  <context:property-placeholder location="db-config.properties" />

  <bean id="dataSource" class="com.oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="${db.url}" />
    <property name="user" value="${db.user.name}" />
  </bean>
</beans>
```



db.url=jdbc:oracle:...
db.user.name=moneytransfer-app

db-config.properties

```
<bean id="dataSource"
  class="com.oracle.jdbc.pool.OracleDataSource">
  <property name="URL" value="jdbc:oracle:..." />
  <property name="user" value="moneytransfer-app" />
</bean>
```

Resolved at
startup time



Topics in this session

- Best Practices
 - Making the best use of XML namespaces
 - STS Wizard, version numbers, 'c' and 'p' namespaces
 - Externalizing values into properties files
 - **Working with a high number of conf. Files**
 - <import/> tag and wildcards usage
 - Using Bean definition inheritance
- Lab
- Advanced Features
 - Inner beans, collections, SpEL

Organizing Spring config files



- Listing all configuration files becomes tedious when working with a high number of files

```
ApplicationContext context =
```

```
    new ClassPathXmlApplicationContext(  
        "config/dao-config.xml", "config/service-config.xml" );
```

What if there are
50 files to be listed here?

- 2 main ways to improve configuration
 - Importing configuration files
 - Using wildcards

Importing Config Files (1)



- Use the import tag

```
<beans>
    <import resource="accounts/account-config.xml" />
    <import resource="restaurant/restaurant-config.xml" />
    <import resource="rewards/rewards-config.xml" />
</beans>
```

application-config.xml

```
<beans>
    <import resource="application-config.xml"/>
    <bean id="dataSource" class="example.TestDataSourceFactory" />
</beans>
```

system-test-config.xml

```
new ClassPathXmlApplicationContext("system-test-config.xml");
```

Client declares one file instead of four

Importing Config Files (2)



- Import tag uses relative path by default

```
<beans>
  <import resource="account-config.xml" />
  <import resource="rewards-config.xml" />
  ...
</beans>
```

In same folder/package
as the current file

- This can be overridden using prefixes

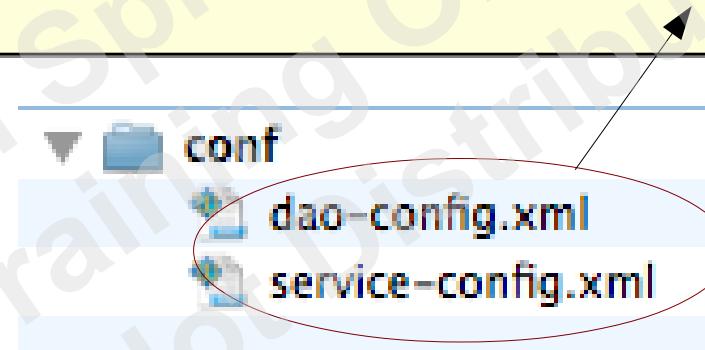
```
<beans>
  <import resource="classpath:com/springsource/account-config.xml" />
  ...
</beans>
```

Path starts from the
classpath root

Using wildcards

- Wildcards can be used to load a set of configuration files

```
new ClassPathXmlApplicationContext("classpath*:conf/*-config.xml");
```



"classpath*" indicates that all classpath sources should be included (jar files and source folders)
Without "classpath*", only the **first** matching folder/jar found is taken into account

Topics in this session

- Best Practices
 - Making the best use of XML namespaces
 - STS Wizard, version numbers, 'c' and 'p' namespaces
 - Externalizing values into properties files
 - Working with a high number of conf. Files
 - <import/> tag and wildcards usage
 - **Using Bean definition inheritance**
- Lab
- Advanced Features
 - Inner beans, collections, SpEL

Bean Definition Inheritance

(1)



- Sometimes several beans need to be configured in the same way
- Use bean definition inheritance to define the common configuration once
 - Inherit it where needed

Without Bean Definition Inheritance



```
<beans>
    <bean id="pool-A" class="com.oracle.jdbc.pool.DataSource">
        <property name="URL" value="jdbc:oracle:thin:@server-a:1521:BANK" />
        <property name="user" value="moneytransfer-app" />
    </bean>

    <bean id="pool-B" class="com.oracle.jdbc.pool.DataSource">
        <property name="URL" value="jdbc:oracle:thin:@server-b:1521:BANK" />
        <property name="user" value="moneytransfer-app" />
    </bean>

    <bean id="pool-C" class="com.oracle.jdbc.pool.DataSource">
        <property name="URL" value="jdbc:oracle:thin:@server-c:1521:BANK" />
        <property name="user" value="moneytransfer-app" />
    </bean>
</beans>
```

Can you find the duplication?

Abstract Parent bean

```
<beans>
  <bean id="abstractPool"
    class="com.oracle.jdbc.pool.DataSource" abstract="true">
    <property name="user" value="moneytransfer-app" />
  </bean>

  <bean id="pool-A" parent="abstractPool">
    <property name="URL" value="jdbc:oracle:thin:@server-a:1521:BANK" />
  </bean>
  <bean id="pool-B" parent="abstractPool">
    <property name="URL" value="jdbc:oracle:thin:@server-b:1521:BANK" />
  </bean>
  <bean id="pool-C" parent="abstractPool">
    <property name="URL" value="jdbc:oracle:thin:@server-c:1521:BANK" />
    <property name="user" value="bank-app" />
  </bean>

</beans>
```

Will not be instantiated

Can override

Each pool inherits its *parent* configuration

Default Parent Bean



```
<beans>
```

```
  <bean id="defaultPool" class="com.oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@server-a:1521:BANK" />
    <property name="user" value="moneytransfer-app" />
  </bean>
```

Overrides URL property

```
  <bean id="pool-B" parent="defaultPool">
    <property name="URL" value="jdbc:oracle:thin:@server-b:1521:BANK" />
  </bean>
```

```
  <bean id="pool-C" parent="defaultPool" class="example.SomeOtherPool">
    <property name="URL"
              value="jdbc:oracle:thin:@server-c:1521:BANK" />
  </bean>
```

Overrides class as well

```
</beans>
```

Inheritance for service and repository beans



- Bean inheritance commonly used for definition of Service and Repository (or DAO) beans





LAB

Using Bean definition inheritance
and Property Placeholders

Topics in this session

- Best Practices
 - Making the best use of XML namespaces
 - Externalizing values into properties files
 - Working with a high number of conf. Files
 - Using Bean definition inheritance
- Lab
- **Advanced Features**
 - **Inner beans**
 - Injecting a collection of dependencies
 - Spring Expression Language

Inner beans

- Inner bean only visible from surrounding bean

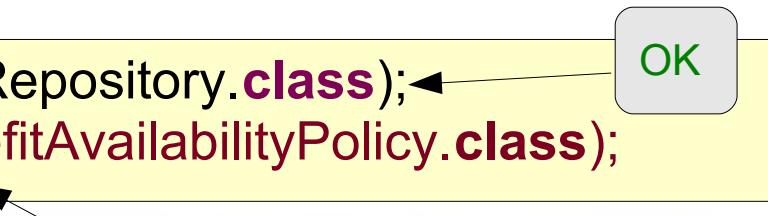
```
<bean id="restaurantRepository"  
      class="rewards.internal.restaurant.JdbcRestaurantRepository">  
    <property name="benefitAvailabilityPolicy">  
      <bean class="rewards...DefaultBenefitAvailabilityPolicy" />  
    </property>  
</bean>
```



No bean id

- Cannot be accessed from the applicationContext

```
applicationContext.getBean(RestaurantRepository.class);  
applicationContext.getBean(DefaultBenefitAvailabilityPolicy.class);
```



OK

NoSuchBeanDefinitionException!!

Without an Inner Bean

```
<beans>

    <bean id="restaurantRepository"
        class="rewards.internal.restaurant.JdbcRestaurantRepository">
        <property name="dataSource" ref="dataSource" />
        <property name="benefitAvailabilityPolicyFactory" ref="factory" />
    </bean>

    <bean id="factory"
        class="rewards.internal.restaurant.availability.
            DefaultBenefitAvailabilityPolicyFactory">
        <constructor-arg ref="rewardHistoryService" />
    </bean>

    ...
</beans>
```

Can be referenced by other beans
(even if it should not be)

With an Inner Bean



```
<beans>

    <bean id="restaurantRepository"
        class="rewards.internal.restaurant.JdbcRestaurantRepository">
        <property name="dataSource" ref="dataSource" />
        <property name="benefitAvailabilityPolicyFactory">
            <bean class="rewards.internal.restaurant.availability.
                DefaultBenefitAvailabilityPolicyFactory">
                <constructor-arg ref="rewardHistoryService" />
            </bean>
        </property>
    </bean>
    ...
</beans>
```

Inner bean has no id (it is anonymous)
Cannot be referenced outside this scope

Multiple Levels of Nesting



```
<beans>
  <bean id="restaurantRepository"
    class="rewards.internal.restaurant.JdbcRestaurantRepository">
    <property name="dataSource" ref="dataSource" />
    <property name="benefitAvailabilityPolicyFactory">
      <bean class="rewards.internal.restaurant.availability.
        DefaultBenefitAvailabilityPolicyFactory">
        <constructor-arg>
          <bean class="rewards.internal.rewards.JdbcRewardHistory">
            <property name="dataSource" ref="dataSource" />
          </bean>
        </constructor-arg>
      </bean>
    </property>
  </bean>
</beans>
```

Inner Beans: pros and cons



- Pros
 - You only expose what needs to be exposed
- Cons
 - Harder to read
- General recommendation
 - Use them sparingly
 - As for inner classes in Java
 - Common choice: Complex "infrastructure beans" configuration



Topics in this session

- Best Practices
 - Making the best use of XML namespaces
 - Externalizing values into properties files
 - Working with a high number of conf. Files
 - Using Bean definition inheritance
- Lab
- **Advanced Features**
 - Inner beans
 - **Injecting a collection of dependencies**
 - Spring Expression Language

beans and *util* collections



- *beans* collections
 - From the default *beans* namespace
 - Simple and easy
- *util* collections
 - From the *util* namespace
 - Requires additional namespace declaration
 - More features available



Both offer support for *set*, *map*, *list* and *properties* collections

Using the *beans* namespace



```
<bean id="service" class="com.acme.service.TransferServiceImpl">
    <property name="customerPolicies">
        <list>
            <ref bean="privateBankingCustomerPolicy"/>
            <ref bean="retailBankingCustomerPolicy"/>
            <bean class="com.acme.DefaultCustomerPolicy"/>
        </list>
    </property>
</bean>
```

```
public void setCustomerPolicies(java.util.List policies) { .. }
```

Equivalent to:

```
TransferServiceImpl service = new TransferServiceImpl();
service.setCustomerPolicies(list); // create list with bean references
```

ApplicationContext

service -> instance of TransferServiceImpl

beans collections limitation



- Can't specify the collection type
 - eg. for *java.util.List* the implementation is *ArrayList*
- Collection has no bean id
 - Can't be accessed from the ApplicationContext
 - Only valid as inner beans

```
<bean id="service" class="com.acme.service.TransferServiceImpl">
    <property name="customerPolicies">
        <list> ... </list>
    </property>
</bean>
```

```
applicationContext.getBean("service");
applicationContext.getBean("customerPolicies");
```

OK

NoSuchBeanDefinitionException!!

Injecting a Set or Map



- Similar support available for
 - Set

```
<property name="customerPolicies">
  <set>
    <ref bean="privateBankingCustomerPolicy"/>
    <ref bean="retailBankingCustomerPolicy"/>
  </set>
</property>
```

- Map (through map / entry / key elements)

```
<property name="customerPolicies">
  <map>
    <entry key="001-pbcp" value-ref="privateBankingCustomerPolicy"/>
    <entry key-ref="keyBean" value-ref="retailBankingCustomerPolicy"/>
  </map>
</property>
```

Key can use primitive type or ref to bean

Injecting a collection of type *Properties*



- Convenient alternative to creating a dedicated properties file
 - Use when property values are unlikely to change

```
<property name="properties">  
  <value>  
    server.host=mailer  
    server.port=1010  
  </value>  
</property>
```

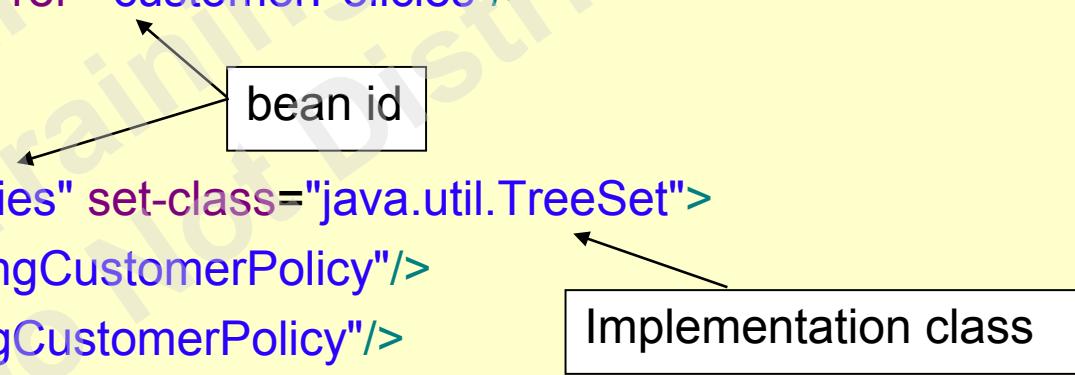
```
<property name="properties">  
  <props>  
    <prop key="server.host">mailer</prop>  
    <prop key="server.port">1010</prop>  
  </props>  
</property>
```

```
public void setProperties(java.util.Properties props) { .. }
```

util collections

- util: collections allow:
 - specifying collection implementation-type and scope
 - declaring a collection as a top-level bean

```
<bean id="service" class="com.acme.service.TransferServiceImpl"  
      p:customerPolicies-ref="customerPolicies"/>  
  
<util:set id="customerPolicies" set-class="java.util.TreeSet">  
  <ref bean="privateBankingCustomerPolicy"/>  
  <ref bean="retailBankingCustomerPolicy"/>  
</util:set>
```



Also: util:list, util:map, util:properties

beans or *util* collections?



- In most cases, the default *beans* collections will suffice
- Just remember the additional features of collections from the `<util/>` namespace in case you would need them
 - Declare a collection as a top-level bean
 - Specify collection implementation-type

Topics in this session

- Best Practices
 - Making the best use of XML namespaces
 - Externalizing values into properties files
 - Working with a high number of conf. Files
 - Using Bean definition inheritance
- Lab
- **Advanced Features**
 - Inner beans
 - Injecting a collection of dependencies
 - **Spring Expression Language**

Spring Expression Language



- SpEL for short
- Inspired by the Expression Language used in Spring WebFlow
- Pluggable/extendable by other Spring-based frameworks



SpEL was introduced in Spring 3.0

SpEL examples

```
<bean class="com.acme.RewardsTestDatabase">  
    <property name="keyGenerator"  
        value="#{strategyBean.databaseKeyGenerator}" />  
</bean>
```

Can refer a
nested property

```
<bean id="strategyBean" class=""com.acme.DefaultStrategies">  
    <property name="databaseKeyGenerator" ref="myKeyGenerator"/>  
</bean>
```

```
<bean id="taxCalculator" class=""com.acme.TaxCalculator">  
    <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>  
</bean>
```

Equivalent to
System.getProperty(...)

- EL Attributes can be:
 - Named Spring beans
 - Implicit references
 - *systemProperties* and *systemEnvironment* available by default
- SpEL allows to create custom functions and references
 - Widely used in Spring projects
 - Spring Security
 - Spring WebFlow
 - Spring Batch
 - Spring Integration
 - ...

Using SpEL functions

- In Spring Security

```
<security:intercept-url pattern="/accounts/**"  
access="isAuthenticated() and hasIpAddress('192.168.1.0/24')"/>
```

- In Spring Batch

```
<bean id="flatFileItemReader" scope="step"  
class="org.springframework.batch.item.file.FlatFileItemReader">  
    <property name="resource" value="#{jobParameters['input.file.name']}"/>  
</bean>
```



Spring Security will be discussed later in this course. *Spring Batch* is part of the "Enterprise Integration with Spring" course

Summary

- Spring offers many techniques to simplify configuration
 - We've seen just a few here
 - It's about expressiveness and elegance, just like code
- Best practices we've discussed can be used daily in most Spring XML projects
 - Imports, Bean inheritance, imports...
- Advanced features are more specialized



Annotations in Spring

Annotations for Dependency Injection and
interceptors

Topics in this Session

- **Fundamentals**
 - **Annotation-based Configuration**
 - Best practices for component-scanning
 - XML versus annotations: when to use what?
 - Mixing XML and annotations for Dependency Injection
 - @PostConstruct and @PreDestroy
 - Stereotypes and meta annotations
- Lab
- Advanced features
 - Standard annotations (JSR 330)

XML/Annotation DI



- XML-based dependency injection

```
<bean id="transferService" class="com.acme.TransferServiceImpl">
    <constructor-arg ref="accountRepository" />
</bean>
```

External configuration

- Annotation-based Configuration

```
@Component
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl(AccountRepository repo) {
        this.accountRepository = repo;
    }
}
```

Annotations embedded with POJOs

```
<context:component-scan base-package="com.springsource"/>
```

Minimal xml config

Usage of @Autowired: many ways



- constructor-injection

```
@Autowired  
public TransferServiceImpl(AccountRepository a) {  
    this.accountRepository = a;  
}
```

One constructor
must be
autowired

- method-injection

```
@Autowired  
public void setAccountRepository(AccountRepository a) {  
    this.accountRepository = a;  
}
```

Usually not
recommended
to create a
getter method

- field-injection

```
@Autowired  
private AccountRepository accountRepository;
```

Even when field is private!!
– but hard to unit test.

@Autowired dependencies: required or not?



- Default behavior: required

```
@Autowired  
public void setAccountRepository(AccountRepository a) {  
    this.accountRepository = a;  
}
```

Exception if no dependency found

- Use *required* attribute to override default behavior

```
@Autowired(required=false)  
public void setAccountRepository(AccountRepository a) {  
    this.accountRepository = a;  
}
```

Only inject if dependency exists

Autowiring and Disambiguation



- What happens here?

```
@Component  
public class TransferServiceImpl implements TransferService {  
    @Autowired  
    public TransferServiceImpl(  
        AccountRepository accountRepository)  
    { ... }
```

```
@Component  
public class JdbcAccountRepository implements AccountRepository {..}
```

```
@Component  
public class HibernateAccountRepository implements AccountRepository {..}
```

Which one should get injected?

At startup: NoSuchBeanDefinitionException, no unique bean of type [AccountRepository] is defined: expected single bean but found 2...

Autowiring and Disambiguation



- Use of the @Qualifier annotation

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl( @Qualifier("jdbcAccountRepository")
        AccountRepository accountRepository) { ... }

    @Component("jdbcAccountRepository") ◀
    public class JdbcAccountRepository implements AccountRepository {..}

    @Component("hibernateAccountRepository")
    public class HibernateAccountRepository implements AccountRepository {..}
```

qualifier



@Qualifier can also be used with method injection and field injection



Component names should not show implementation details unless there are 2 implementations of the same interface (as it is the case here)

Component names

- When not specified
 - Names are auto-generated
 - De-capitalized non-qualified classname by default
 - *But* will pick up implementation details from classname
 - *Recommendation:* never rely on generated names!
- When specified
 - Allow disambiguation when 2 bean classes implement the same interface

Common strategy: avoid using qualifiers when possible.



In many cases, it is not necessary to have 2 implementations of the same bean in the ApplicationContext – see filters below

XML vs Annotations syntax



- Same options are available

```
@Component("transferService")
@Scope("prototype")
@Lazy(false)
public class TransferServiceImpl
    implements TransferService {
    @Autowired
    public TransferService
        (AccountRepository accRep) ...
    @Value("#{limits.max}")
    public void setMaxTransfers(int max) ...
}
```

annotations

```
<bean id="transferService"
      scope="prototype"
      lazy-init="false"
      class="TransferServiceImpl">
    <property id="accountRepo"
              ref="accountRepository"/>
    <property id="maxTransfers"
              ref="#{limits.max}"/>
</bean>
```

xml



Autowiring is also possible using XML (using the *autowire* attribute)

Topics in this Session

- Fundamentals
 - Annotation-based Configuration
 - **Best practices for component-scanning**
 - XML versus annotations: when to use what?
 - Mixing XML and annotations for Dependency Injection
 - @PostConstruct and @PreDestroy
 - Stereotypes and meta annotations
- Lab
- Advanced features
 - Standard annotations (JSR 330)

Component scanning

- Components are scanned at startup
 - Jar dependencies also scanned!
 - Could result in slower startup time if too many files scanned
 - Especially for large applications
 - A few seconds slower in the worst case
- What are the best practices?

Best practices

- Really bad:

```
<context:component-scan base-package="org,com"/>
```

All “org” and “com” packages in the classpath will be scanned!!

- Still bad:

```
<context:component-scan base-package="com"/>
```

- OK:

```
<context:component-scan base-package="com.acme.app"/>
```

- Optimized:

```
<context:component-scan  
base-package="com.acme.app.repository,  
com.acme.app.service, com.acme.app.controller"/>
```

Including & Excluding Beans



- Using filters, possibility to include or exclude beans
 - Based on type
 - Based on class-level annotation

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

Based on name

Topics in this Session

- **Fundamentals**
 - Annotation-based Configuration
 - Best practices for component-scanning
 - **XML versus annotations: when to use what?**
 - Mixing XML and annotations for Dependency Injection
 - @PostConstruct and @PreDestroy
 - Stereotypes and meta annotations
- Lab
- Advanced features
 - Standard annotations (JSR 330)

When to use what?

<?xml?>

XML

- For infrastructure and more 'static' beans
- Pros:
 - Is centralized in one (or a few) places
 - Most familiar configuration format for most people
 - Can be used for all classes (not just your own)
- Cons:
 - Some people just don't like XML!
 - But the STS bean-file editor makes it a lot easier
 - More verbose than annotations

When to use what?



Annotations

- Nice for frequently changing beans
- Pros:
 - Single place to edit (just the class)
 - Allows for very rapid development
- Cons:
 - Configuration spread across your code base
 - Harder to debug/maintain
 - Only works for your own code
 - Merges configuration and code (bad sep. of concerns)



Topics in this Session

- **Fundamentals**
 - Annotation-based Configuration
 - Best practices for component-scanning
 - XML versus annotations: when to use what?
 - **Mixing XML and annotations for Dependency Injection**
 - @PostConstruct and @PreDestroy
 - Stereotypes and meta annotations
- Lab
- Advanced features
 - Standard annotations (JSR 330)

Mixing XML and annotations



- You can mix and match in many ways. A few options:
 - Use annotations for Spring MVC beans and use XML for Service and Repository beans
 - Use annotations when possible, but still use XML for legacy code that can't be changed
 - Use @Autowired annotations without component scanning
 - Will be discussed as an example in the following slides

Using @Autowired without component scanning (1)



- Use of <context:annotation-config/>
 - Processes all DI annotations
 - Does not perform component-scanning
 - Beans should be declared explicitly

```
<beans ...>
  <context:annotation-config/> ← Detects annotations such as @Autowired
                                without performing component scanning
  <bean id="transferService" class="com.acme.TransferServiceImpl"/>
  <bean id="accountRepository" class="com.acme.JdbcAccountRepository"/>
</beans>
```



<property /> or <constructor-arg/> are not needed any more because classes use @Autowired

Using @Autowired without component scanning (2)



- @Component no longer needed

```
public class TransferServiceImpl implements TransferService {  
    @Autowired  
    public TransferServiceImpl(AccountRepository accountRepository) { ... }  
}
```

```
public class JdbcAccountRepository implements AccountRepository {...}
```

- With Qualifier

```
public class JdbcAccountRepository implements AccountRepository {  
    public JdbcAccountRepository(@Qualifier("mysqlDatasource")  
        DataSource dataSource) { ... }  
}
```

Refers to
declared bean id

```
public class JdbcAccountRepository implements AccountRepository {...}
```

Topics in this Session

- **Fundamentals**
 - Annotation-based Configuration
 - Best practices for component-scanning
 - XML versus annotations: when to use what?
 - Mixing XML and annotations for Dependency Injection
 - **@PostConstruct and @PreDestroy**
 - Stereotypes and meta annotations
- Lab
- Advanced features
 - Standard annotations (JSR 330)

Introduction to @PostConstruct and @PreDestroy



- Can be used to add behavior at startup and shutdown

```
public class JdbcAccountRepository {  
    @PostConstruct  
    void populateCache() {}  
  
    @PreDestroy  
    void clearCache() {}  
}
```

Method called at startup *after* dependency injection

Method called at shutdown prior to destroying the bean instance



Annotated methods can have any visibility but *must* take *no* parameters and *only* return *void*

@PostConstruct and @PreDestroy configuration



- <context:annotation-config />
 - Not needed if <component-scan /> already enabled

```
public class JdbcAccountRepository {  
    @PostConstruct void populateCache() { ... }  
    @PreDestroy void clearCache() { ... }  
}
```

JSR-250 annotations
– EJB3 also uses them

```
<beans>  
    <bean id="accountRepository"  
          class="app.impl.JdbcAccountRepository" />  
    <context:annotation-config/>  
</beans>
```

Enables processing of
@PostConstruct and @PreDestroy

@PostConstruct

- Called after setter methods are called

```
public class JdbcAccountRepository {  
    private DataSource dataSource;  
    @Autowired  
    public void setDataSource(DataSource dataSource)  
    { this.dataSource = dataSource; }  
  
    @PostConstruct  
    public void populateCache()  
    { Connection conn = dataSource.getConnection(); //... }  
}
```



Constructor
called

Setter(s) called

@PostConstruct
method(s) called

@PreDestroy

- Called when ApplicationContext is closed
 - Useful for releasing resources and 'cleaning up'

```
ConfigurableApplicationContext context = ...
```

```
// Destroy the application
```

```
context.close();
```

Triggers call of all @PreDestroy annotated methods

```
public class JdbcAccountRepository {
```

```
//...
```

```
@PreDestroy
```

```
public void clearCache() {
```

```
//...
```

```
}
```

```
}
```

Tells Spring to call this method prior to destroying the bean instance

XML alternative to @PostConstruct and @PreDestroy



- **init-method** and **destroy-method** attributes
 - The way to go for third-party beans
 - message brokers, data sources ...
 - Use them for other beans if you prefer XML over annotations

```
<bean id="accountRepository"
      class="app.impl.JdbcAccountRespository"
      init-method="populateCache"
      destroy-method="clearCache">
    ...
</bean>
```

Topics in this Session

- Fundamentals
 - Annotation-based Configuration
 - Best practices for component-scanning
 - XML versus annotations: when to use what?
 - Mixing XML and annotations for Dependency Injection
 - @PostConstruct and @PreDestroy
 - **Stereotypes and meta annotations**
- Lab
- Advanced features
 - Standard annotations (JSR 330)

Stereotype annotations

- Component scanning also checks for annotations that are themselves annotated with @Component
 - So-called *stereotype annotations*
- Example:

```
<context:component-scan  
    base-package="app.impl"/>
```

scans

```
@Service("transferService")  
public class TransferServiceImpl  
    implements TransferService {...}
```

*Declaration of the
@Service annotation*

```
@Target({ElementType.TYPE})  
...  
@Component  
public @interface Service {...}
```

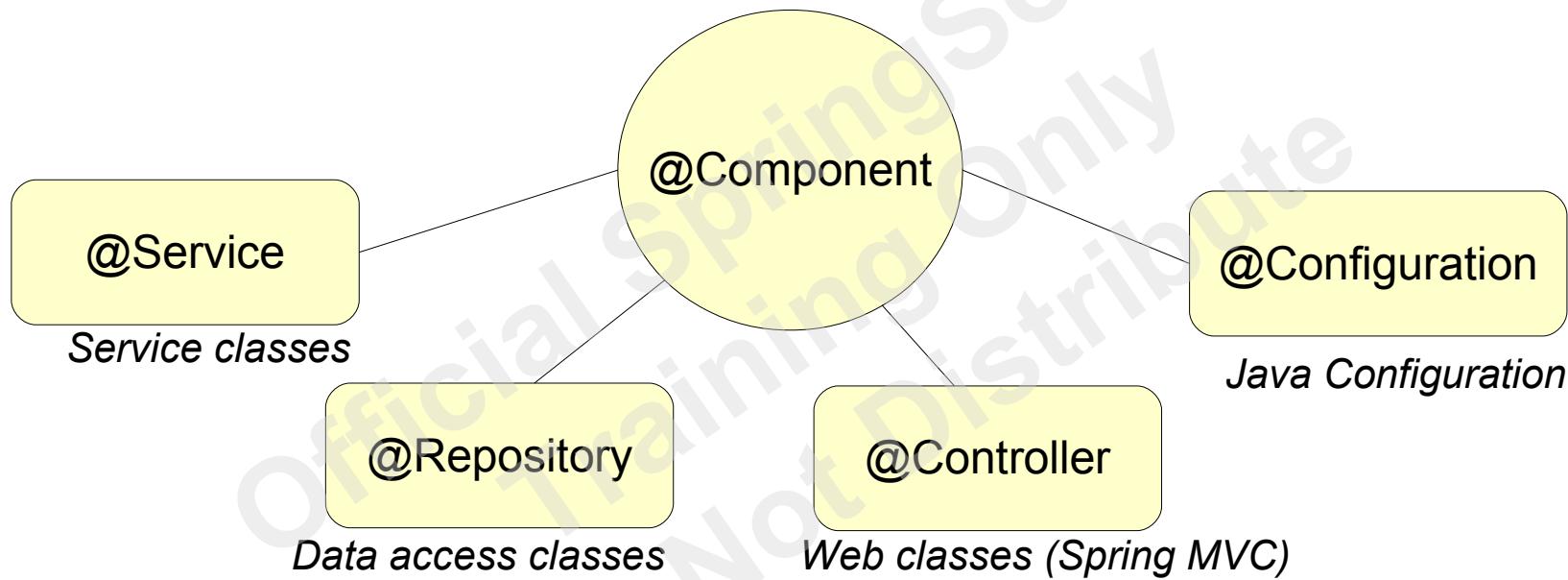


@Service annotation is part of the Spring framework

Stereotype annotations



- Spring framework stereotype annotations



Other Spring projects provide their own stereotype annotations
(Spring Web-Services, Spring Integration...)

Meta-annotations

- Annotation which can be used to annotate other annotations
 - e.g. all service beans should be configurable using component scanning and be transactional

```
<context:component-scan  
base-package="app.impl"/>
```

scans

recognizes

```
@MyTransactionalService  
public class TransferServiceImpl  
implements TransferService {...}
```

Custom annotation

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@Service  
@Transactional(timeout=60)  
public @interface  
MyTransactionalService {  
    String value() default "";  
}
```



LAB

Using Spring Annotations
to configure and test an application

Topics in this Session

- Fundamentals
 - Annotation-based Configuration
 - Best practices for component-scanning
 - XML versus annotations: when to use what?
 - Mixing XML and annotations for Dependency Injection
 - @PostConstruct and @PreDestroy
 - Stereotypes and meta annotations
- Lab
- Advanced features
 - Standard annotations (**JSR 330**)

- Java Specification Request 330
 - Also known as @Inject
 - Joint JCP effort by Google and SpringSource
 - Standardizes internal DI annotations
 - Published late 2009
 - Spring is a valid JSR-330 implementation
- Subset of functionality compared to Spring's @Autowired support
 - @Inject has 80% of what you need
 - Rely on @Autowired for the rest

JSR 330 annotations



Also scans JSR-330 annotations

```
<context:component-scan base-package="app.impl"/>
```

```
import javax.inject.Inject;  
import javax.inject.Named;
```

Should be specified for component scanning (even without a name)

```
@Named  
public class TransferServiceImpl implements TransferService {  
    @Inject  
    public TransferServiceImpl( @Named("accountRepository")  
        AccountRepository accountRepository) { ... }  
}
```

```
import javax.inject.Named;  
  
@Named("accountRepository")  
public class JdbcAccountRepository implements  
    AccountRepository {..}
```

From @Autowired to @Inject



Spring	JSR 330	Comments
@Autowired	@Inject	@Inject always mandatory, has no required option
@Component	@Named	Spring also scans for @Named
@Scope	@Scope	JSR 330 Scope for meta-annotation and injection points only
@Scope ("singleton")	@Singleton	JSR 330 default scope is like Spring's 'prototype'
@Qualifier	@Named	
@Value	No equivalent	SpEL specific
@Required	Redundant	@Inject always required
@Lazy	No equivalent	Probably a good thing!

Summary

- Spring's configuration directives can be written in XML, using Java or using annotations
- You can mix and match XML, Java and annotations as you please
- Autowiring with @Component allows for almost empty XML configuration files



Dependency Injection using Java Configuration

Official SpringSource
Training
Do Not Distribute

Topics in this session

- **Spring Java Configuration**
- Best practices: where to use Spring Java Configuration?

Official SpringSource
Training Only
Do Not Distribute

Introduction

- Third way of doing Dependency Injection
 - After XML and Annotations
- Was introduced in Spring 3.0
- External configuration as with XML
 - Uses Java syntax instead of XML

@Configuration Comparison



Java configuration class

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public TransferService transferService()  
    {  
        TransferService service  
            = new TransferServiceImpl();  
        service.setRepository(repository());  
        return service;  
    }  
  
    @Bean("accountRepository")  
    public AccountRepository repository()  
    { //... }  
}
```

XML configuration file

```
<beans>  
    bean id  
    <bean id="transferService"  
          class="com.acme.TransferServiceImpl">  
        <property name="repository"  
                 ref="accountRepository" />  
    </bean>  
  
    Dependency injection  
  
    bean id  
    <bean id="accountRepository"  
          class="com.acme.JdbcAccountRepository">  
        ...  
    </bean>  
  
</beans>
```

@Configuration Syntax

```
@Configuration  
public class ApplicationConfig {
```

Configuration class

```
@Bean
```

```
public TransferService transferService() {  
    TransferService service = new TransferServiceImpl();  
    service.setRepository(repository());  
    return service;  
}
```

Bean definition

```
@Bean @Scope("prototype")  
public AccountRepository repository() {  
    //...  
}
```

Can specify scope
(default is *singleton*)

Note: A default constructor is *mandatory*.

Option 1: Using a Java Configuration Explicitly



- Create a context based on @Configuration classes

```
ApplicationContext context = new AnnotationConfigApplicationContext(  
    InfraConfig.class, AppConfig.class);
```

```
@Configuration  
public class InfraConfig {  
  
    @Bean  
    public DataSource dataSource() {  
        ...  
    }  
}
```

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public TransferService transferService() {  
        ...  
    }  
}
```

Option 2: Use Component Scanning



- Appropriate when Java Config used together with XML or annotations

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("application-config.xml");
```

main method/ JUnit test

```
<context:component-scan base-package="com.acme.config" />
```

XML configuration

```
@Configuration //component-scanned  
public class ApplicationConfig {
```

```
    @Bean  
    public TransferService transferService() { ... }  
}
```

Java configuration

Quiz

Which is the best implementation?

```
@Bean  
public AccountRepository accountRepository()  
{ return new JdbcAccountRepository(); }
```

```
@Bean  
public TransferService transferService1() {  
    TransferServiceImpl service = new TransferServiceImpl();  
    service.setAccountRepository(accountRepository());  
    return service;  
}
```

```
@Bean  
public TransferService transferService2() {  
    return new TransferServiceImpl( new JdbcAccountRepository() );  
}
```

Method call

New instance

Rule: never instantiate without calling dedicated method. Let's discuss why ...

Working with singletons

- *singleton* is the default scope

```
@Bean
public AccountRepository accountRepository() {
    return new JdbcAccountRepository(); }
```



```
@Bean
public TransferService transferService() {
    TransferServiceImpl service = new TransferServiceImpl();
    service.setAccountRepository(accountRepository());
    return service;
}
@Bean
public AccountService accountService() {
    return new TransferServiceImpl( accountRepository() );
}
```

Singleton??

Method
called twice

HOW IS IT POSSIBLE?

Inheritance-based Proxies



- At startup time, a child class is created
 - For each bean, an instance is cached in the child class
 - Child class only calls *super* at first instantiation

```
@Configuration  
public class AppConfig {  
    @Bean  
    public AccountRespository accountRepository() { ... }  
    @Bean  
    public TransferService transferService() { ... }  
}
```



```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {  
  
    public AccountRespository accountRepository() { // ... }  
  
    public TransferService transferService() { // ... }
```

Inheritance-based Proxies



- Child class is the entry point

```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {  
  
    public AccountRespository accountRepository() {  
        // if bean is in the applicationContext  
        //   return bean  
        // else call super.accountRepository() and store bean in context  
    }  
  
    public TransferService transferService() {  
        // if bean is in the applicationContext, return bean  
        // else call super.transferService() and store bean in context  
    }  
}
```



Java Configuration uses *cglib* for inheritance-based proxies

Equivalent to namespaces

- In Spring 3.1, several namespace-equivalent annotations have been added
 - Component-scanning, AOP, transactions ...

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <context:component-scan
        base-package="com.acme" />

    <aop:aspectj-autoproxy />

    <tx:annotation-driven />

</beans>
```

```
@Configuration
@ComponentScan("com.acme")

@EnableAspectJAutoProxy

@EnableTransactionManagement
public class AppConfig {  
}
```

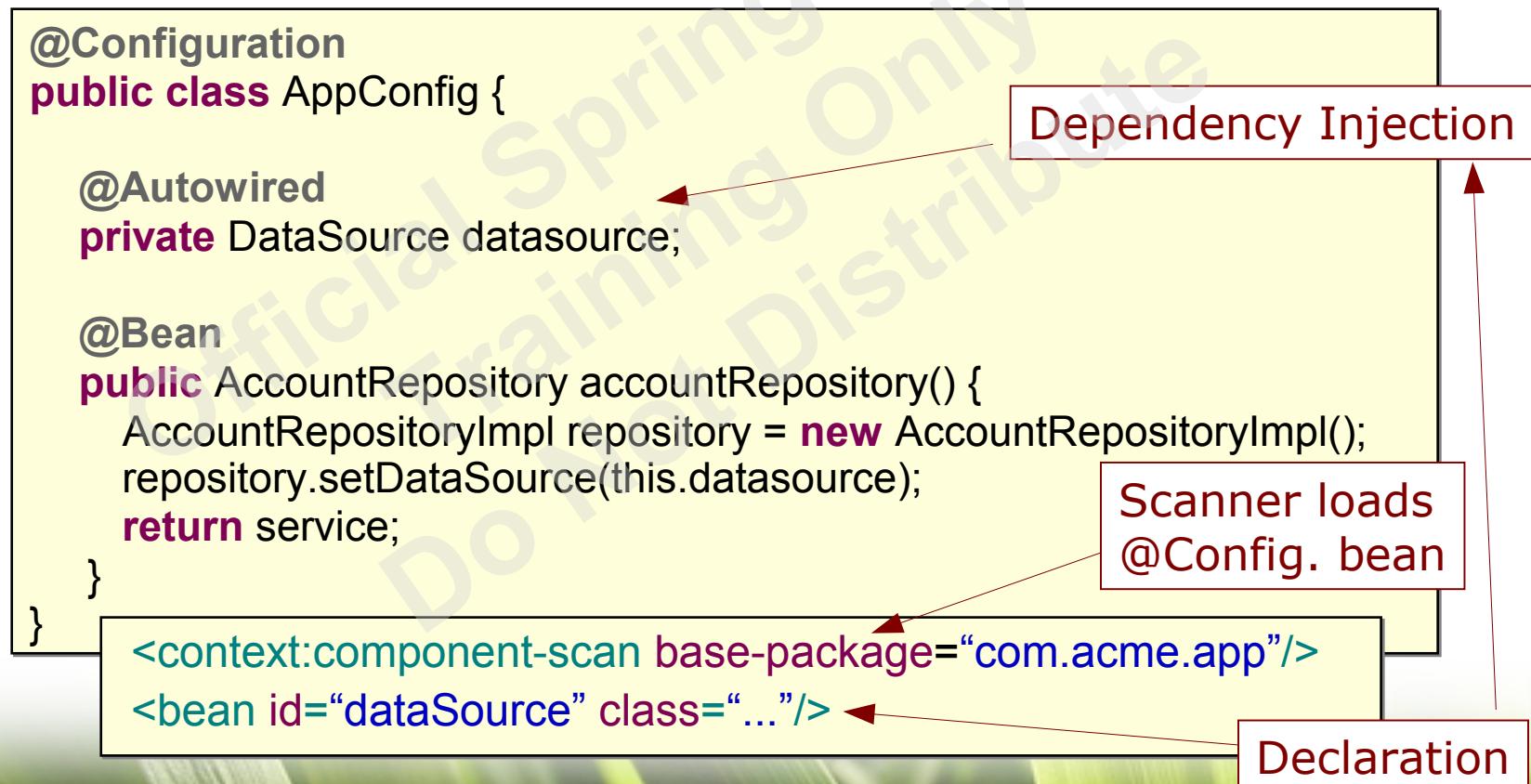


tx and aop namespaces will be discussed later in dedicated modules

Mixing configuration styles: XML & @Configuration (1)



- XML drives the process
 - Can use XML dependencies in @Configuration



Mixing configuration styles: @Configuration & XML (2)



- Or let @Configuration class drive everything

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

Declaration

```
@Configuration  
@ImportResource("classpath:system-config.xml")  
@ComponentScan("com.acme.app")  
public class AppConfig {  
    @Autowired  
    private DataSource datasource;
```

Load XML config

Run scanner – Spring 3.1+

Dependency Injection

```
    @Bean  
    public AccountRepository accountRepository() { //... as previous slide ... }  
}
```

```
ctx = new AnnotationConfigApplicationContext(ApplicationConfig.class);
```

Topics in this session

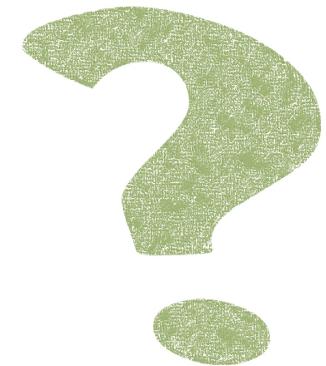
- Spring Java Configuration
- **Best practices: where to use Spring Java Configuration?**

Official Springsource
Training Only
Do Not Distribute

Where to use Java Configuration



- Not the most common way
 - Annotations and XML are more common
- In some advanced situations, Java Configuration is the way to go
 - Let's discuss is with some practical examples



Case 1

- Inject appropriate dependency according to a specific algorithm
 - eg. injecting a local or remote dependency

```
@Bean public TransferService transferService() {  
    boolean isRemote=...;  
    if (isRemote) {  
        clientService.setRepository(remoteRepository());  
    } else {  
        clientService.setRepository(localRepository());  
    }  
    return service;  
}
```



In that case you can mix and match:

- Use XML or annotations for most of the application's configuration
- Only use Java Configuration for the bean definitions that require such an algorithm

Case 2: no Spring tooling



- I do not want to mix code and configuration
 - Therefore using annotations is not an option
- I am considering using XML config. However I do not have access to Spring tooling
 - Eg: I use Eclipse but STS/Spring IDE hasn't been installed/approved
- In that case, Java Configuration is a good choice
 - @Configuration classes are compiled thus validated
 - Auto-complete always works in Java-classes
 - Refactoring friendly in all modern IDEs

Summary

- Java Configuration is less frequently used than XML and annotations
- Extremely helpful in some specific situations
- Equivalents to XML namespaces are available as of Spring 3.1



Understanding the Bean Lifecycle

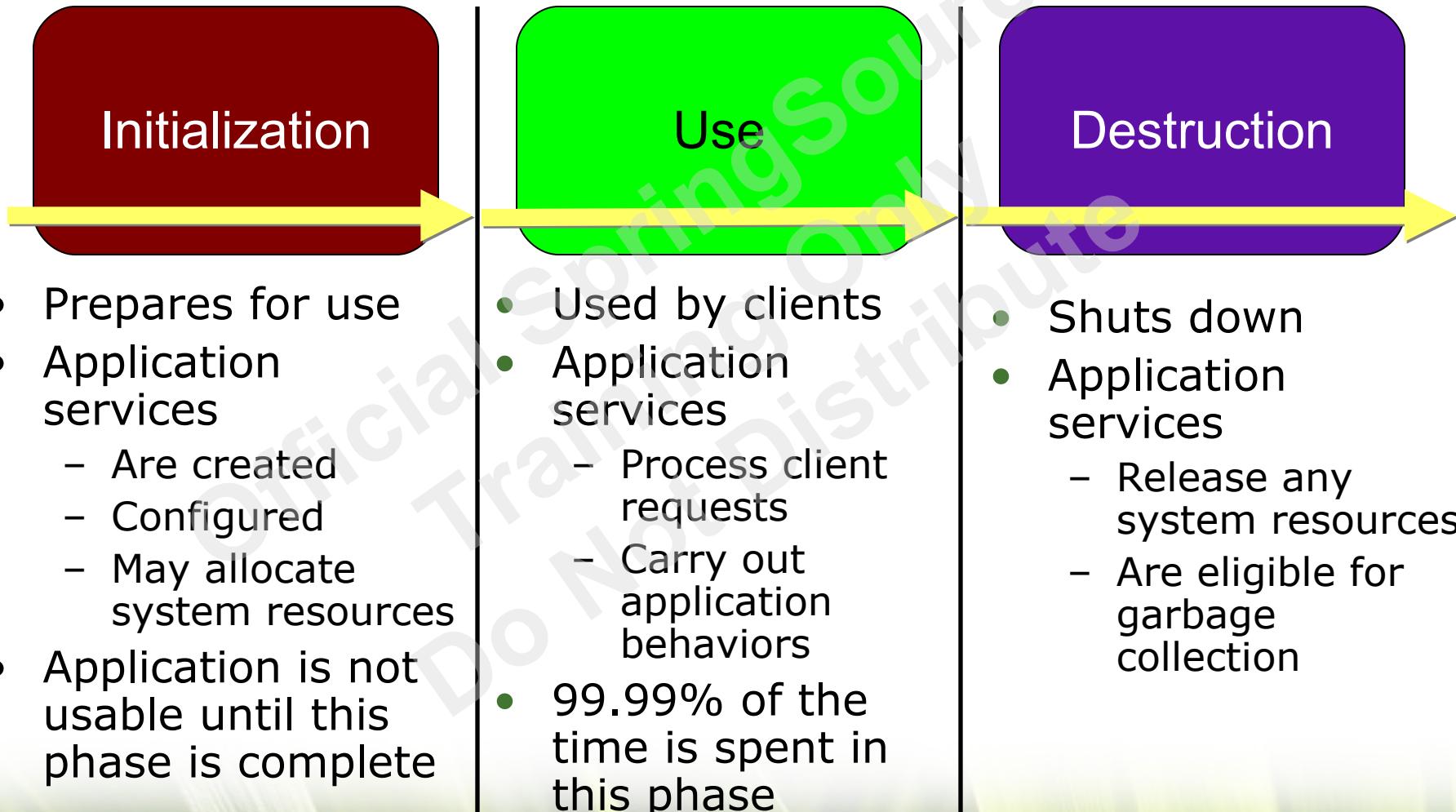
An In-depth Look at Spring's Lifecycle
Management Features

Topics in this session

- **Introduction**
- The initialization phase
- The use phase
- The destruction phase

Official SpringSource
Training Only
Do Not Distribute

Phases of the Application Lifecycle



Spring's Role as a Lifecycle Manager



- This lifecycle applies to *any* class of application
 - Standalone Java application
 - JUnit System Test
 - Java EE™ (web or full profile)
- Spring fits in to manage application lifecycle
 - Plays an important role in all phases
 - *Lifecycle is the same in all these configurations*
- Lifecycle is the same for all 3 dependency injection styles
 - XML, annotations and Java Configuration



Topics in this session

- Introduction
- **The initialization phase**
- The use phase
- The destruction phase



Lifecycle of a Spring Application Context

(1) - The Initialization Phase

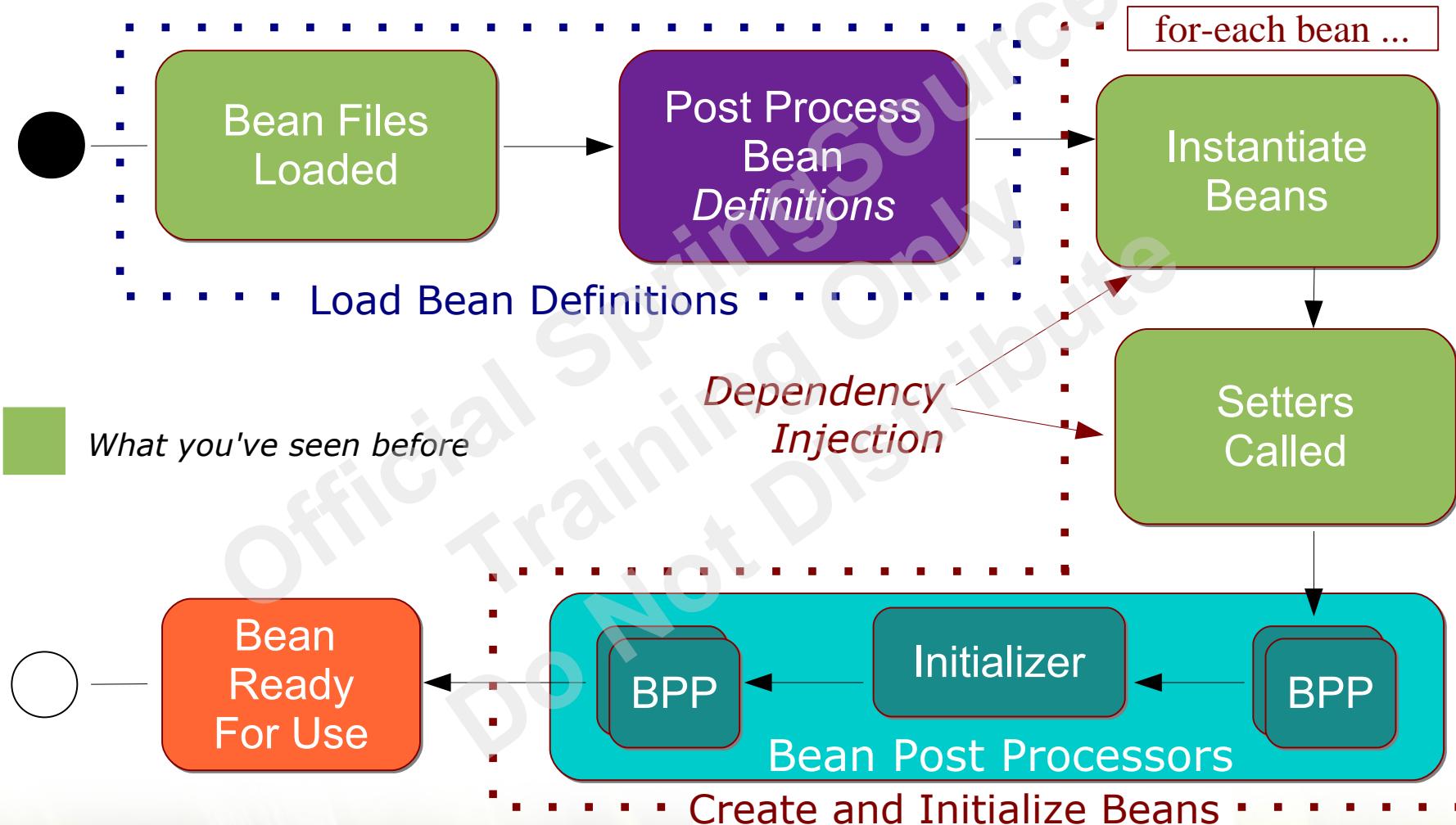


- When a context is created the initialization phase completes

```
// Create the application from the configuration  
ApplicationContext context = new ClassPathXmlApplicationContext(  
    "application-config.xml",  
    "test-infrastructure-config.xml"  
);
```

- But what exactly happens in this phase?

Bean Initialization Steps

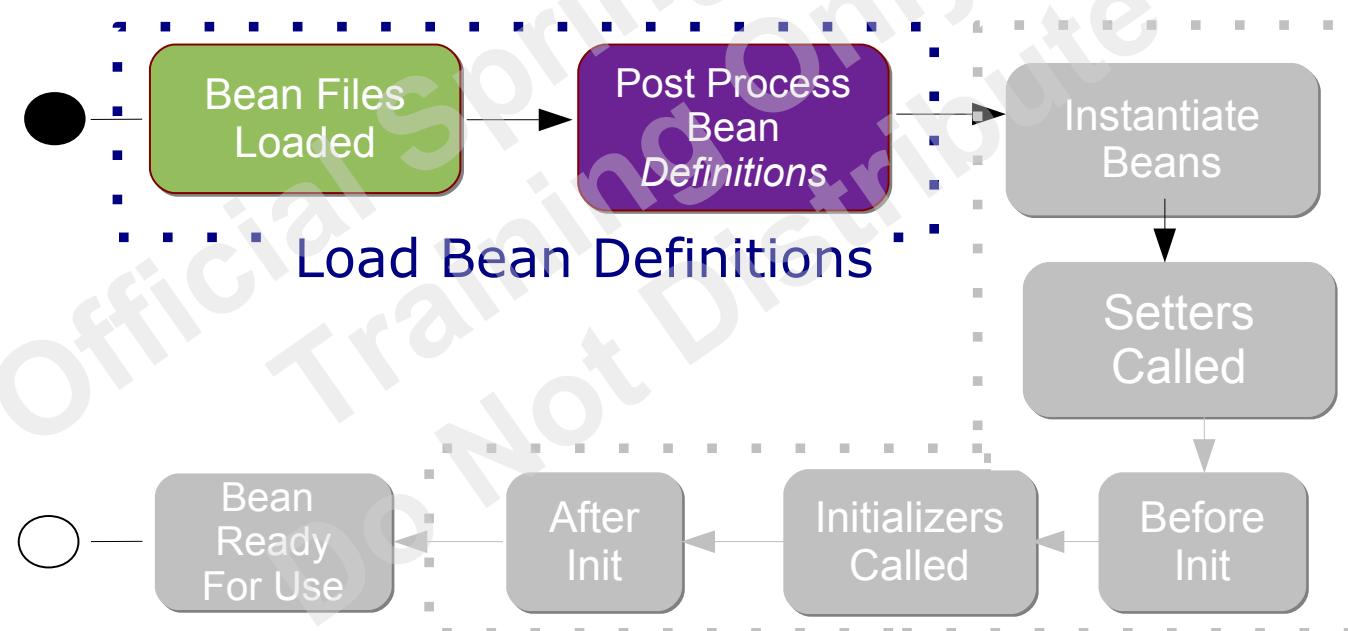


Inside The Application Context

- Initialization Lifecycle



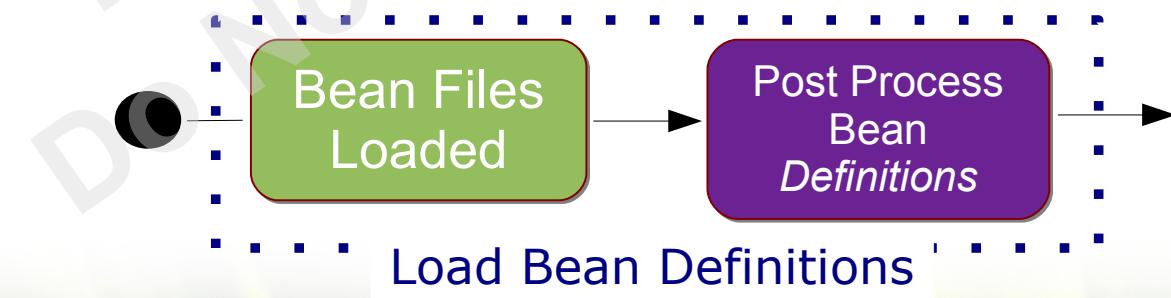
- **Load bean definitions**
- Initialize bean instances



Load Bean Definitions



- The XML files are parsed
- Bean definitions are loaded into the context's BeanFactory
 - Each indexed under its id
- Special BeanFactoryPostProcessor beans are invoked
 - Can modify the definition of any bean



Load Bean Definitions

application-config.xml

```
<beans>
    <bean id="transferService" ...
    <bean id="accountRepository"...
</beans>
```

test-infrastructure-config.xml

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

Can modify the definition of
any bean in the factory
before any objects are created

ApplicationContext

BeanFactory

transferService
accountRepository
dataSource

postProcess(BeanFactory)

BeanFactoryPostProcessors

The BeanFactoryPostProcessor Extension Point



- Useful for applying transformations to groups of bean *definitions*
 - Before any objects are actually created
- Several useful implementations are provided by the framework
 - You can also write your own
 - Implement the **BeanFactoryPostProcessor** interface

```
public interface BeanFactoryPostProcessor {  
    public void postProcessBeanFactory  
        (ConfigurableListableBeanFactory beanFactory);  
}
```

Most Common Example of BeanFactoryPostProcessor



- Remember `<context:property-placeholder>` ?

```
<beans ...>
    <context:property-placeholder location="db-config.properties" />

    <bean id="dataSource" class="com.oracle.jdbc.pool.DataSource">
        <property name="URL" value="${dbUrl}" />
        <property name="user" value="${dbUserName}" />
    </bean>
</beans>
```



dbUrl=jdbc:oracle:...
dbUserName=moneytransfer-app



```
<bean id="dataSource"
      class="com.oracle.jdbc.pool.DataSource">
    <property name="URL" value="jdbc:oracle:..." />
    <property name="user" value="moneytransfer-app" />
</bean>
```

Bean definition behind the namespace



- The namespace is just an elegant way to hide the corresponding bean declaration

```
<context:property-placeholder location="db-config.properties" />
```

```
<bean class="org.springframework...PropertySourcesPlaceholderConfigurer">
    <property name="location" value="db-config.properties"/>
</bean>
```

Implements
BeanFactoryPostProcessor

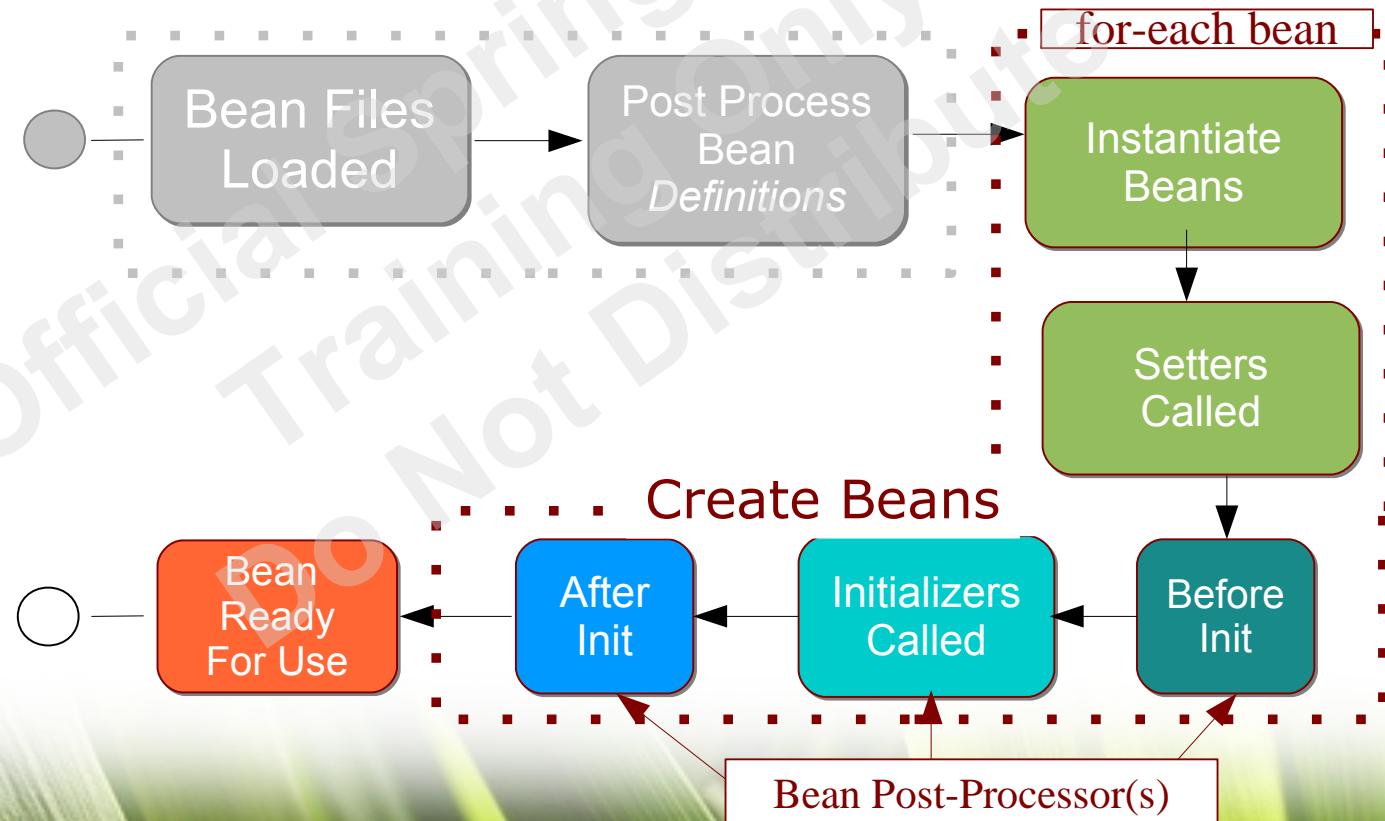


PropertySourcesPlaceHolderConfigurer was introduced in Spring 3.1. Prior to that, *PropertyPlaceHolderConfigurer* was used instead

Inside the Application Context Initialization Lifecycle



- Load bean definitions
- **Initialize bean instances**



Initializing Bean Instances



- Each bean is *eagerly* instantiated by default
 - Created in right order with its dependencies injected
- After dependency injection each bean goes through a post-processing phase
 - Further configuration and initialization may occur
- After post processing the bean is fully initialized and ready for use
 - Tracked by its id until the context is destroyed



Lazy beans are supported – only created when `getBean()` called.
Not recommended, often misused. `<bean lazy-init="true" ...>`

Bean Post Processing



- There are two types of bean post processors
 - Initializers
 - Initialize the bean if instructed
 - All the rest!
 - Allow for additional configuration
 - May run before or after the initialize step



Bean Post Processing Steps



- **Initialize the bean if instructed**
- Call special BeanPostProcessors to perform additional configuration



Inside the Application Context Init Lifecycle



- All *init* methods are called
 - CommonAnnotationBeanPostProcessor

```
public class JdbcAccountRepo {  
  
    @PostConstruct  
    public void populateCache() {  
        //...  
    }  
}
```

By Annotation

```
<bean id="accountRepository"  
      class="com.acme.JdbcAccountRepo"  
      init-method="populateCache">  
    ...  
</bean>
```

Using XML only

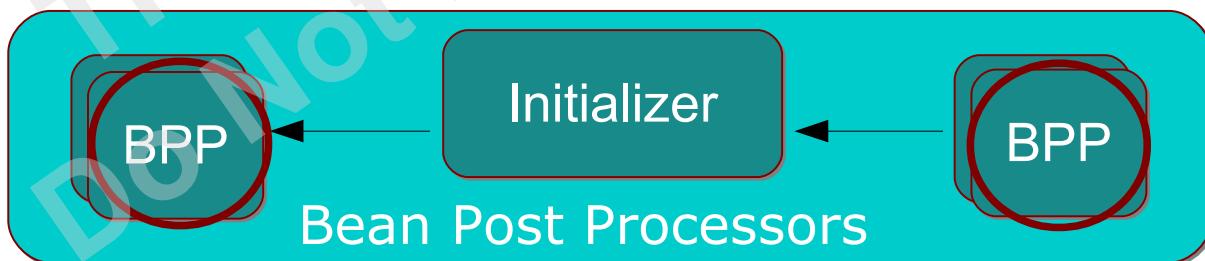
```
<context:annotation-config/>
```

Declares several BPPs *including*
CommonAnnotationBeanPostProcessor

Bean Post Processing Steps



- Initialize the bean if instructed
- **Call special BeanPostProcessors to perform additional configuration**



The BeanPostProcessor Extension Point



- An important extension point in Spring
 - Can modify bean *instances* in any way
 - *Powerful* enabling feature
- Spring provides several implementations
 - Not common to write your own
 - Must implement the **BeanPostProcessor** interface

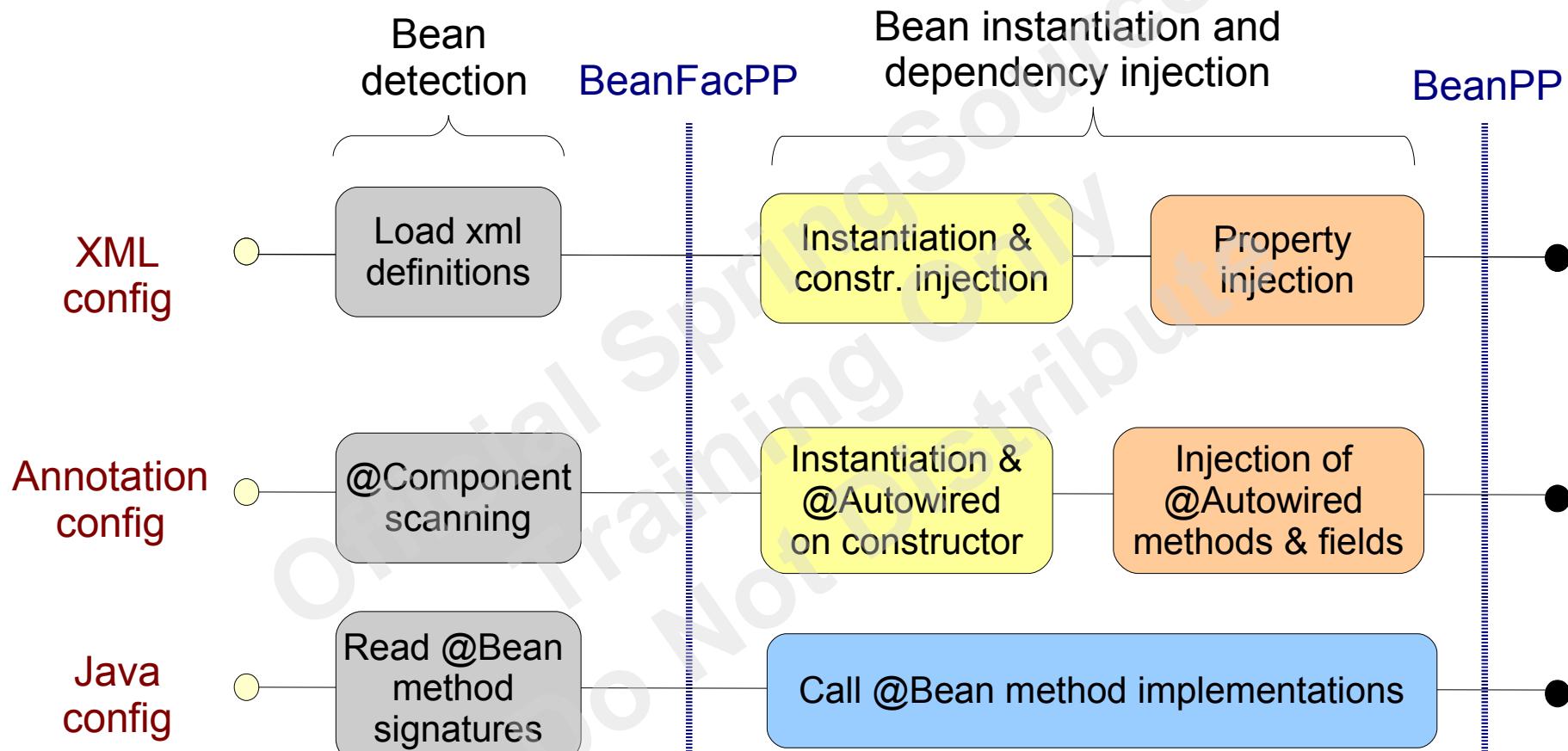
Course
will show
several
BPPs

```
public interface BeanPostProcessor {  
    public Object postProcessAfterInitialization(Object bean, String beanName);  
    public Object postProcessBeforeInitialization(Object bean, String beanName);  
}
```

Post-processed bean

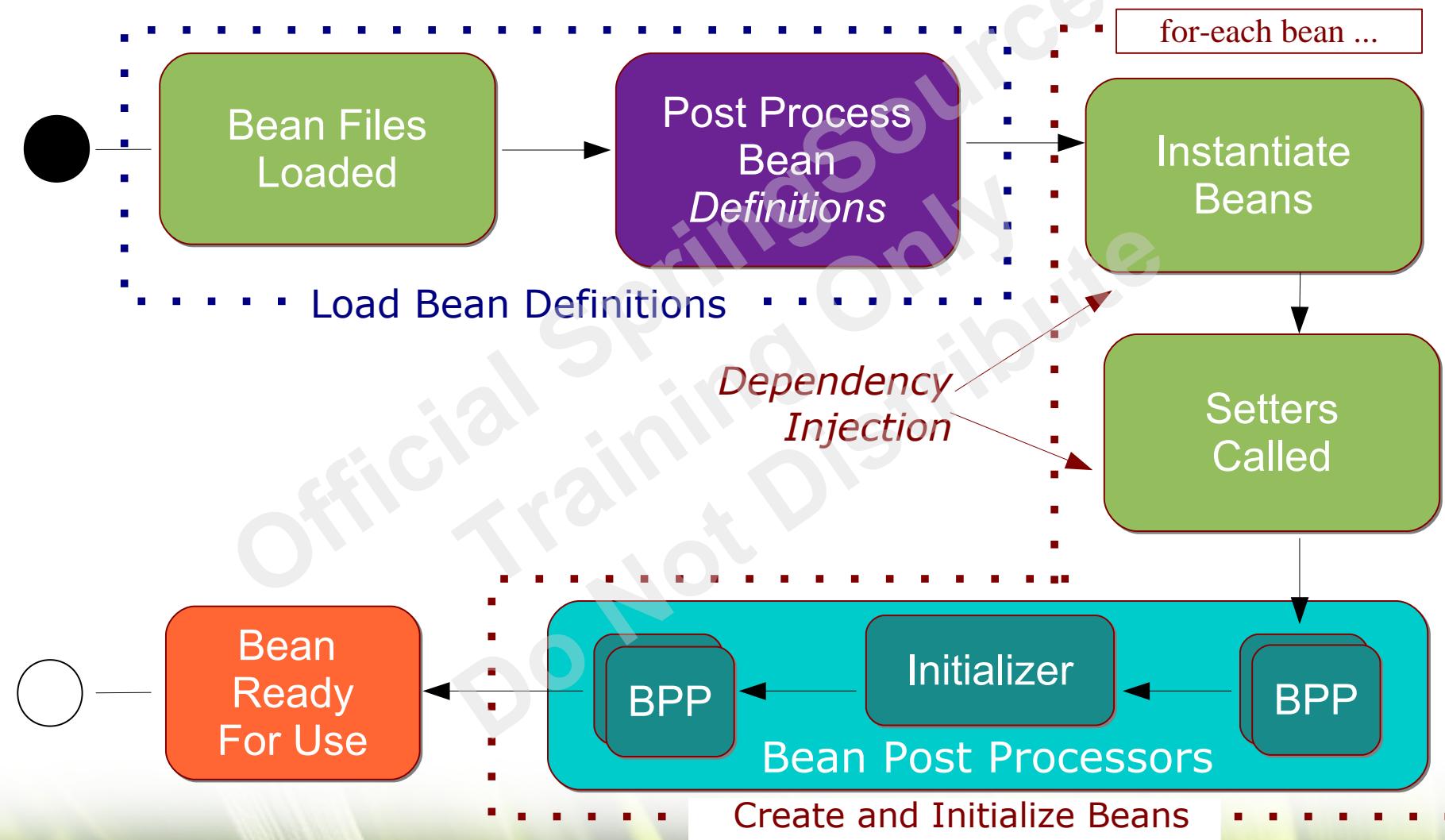
Original bean

Configuration Lifecycle



BeanFacPP → BeanFactoryPostProcessor
BeanPP → BeanPostProcessor

The Full Initialization Lifecycle



Topics in this session

- Introduction
- The initialization phase
- **The use phase**
- The destruction phase



Lifecycle of a Spring Application Context (2) - The Use Phase



- When you invoke a bean obtained from the context the application is used

```
ApplicationContext context = // get it from somewhere  
// Lookup the entry point into the application  
TransferService service =  
    (TransferService) context.getBean("transferService");  
// Use it!  
service.transfer(new MonetaryAmount("50.00"), "1", "2");
```

- But exactly what happens in this phase?

Inside The Bean Request (Use) Lifecycle



- The bean is just your raw object
 - it is simply invoked directly (nothing special)

transfer("50", "1", 2")



- Your bean has been wrapped in a *proxy*
 - things become more interesting

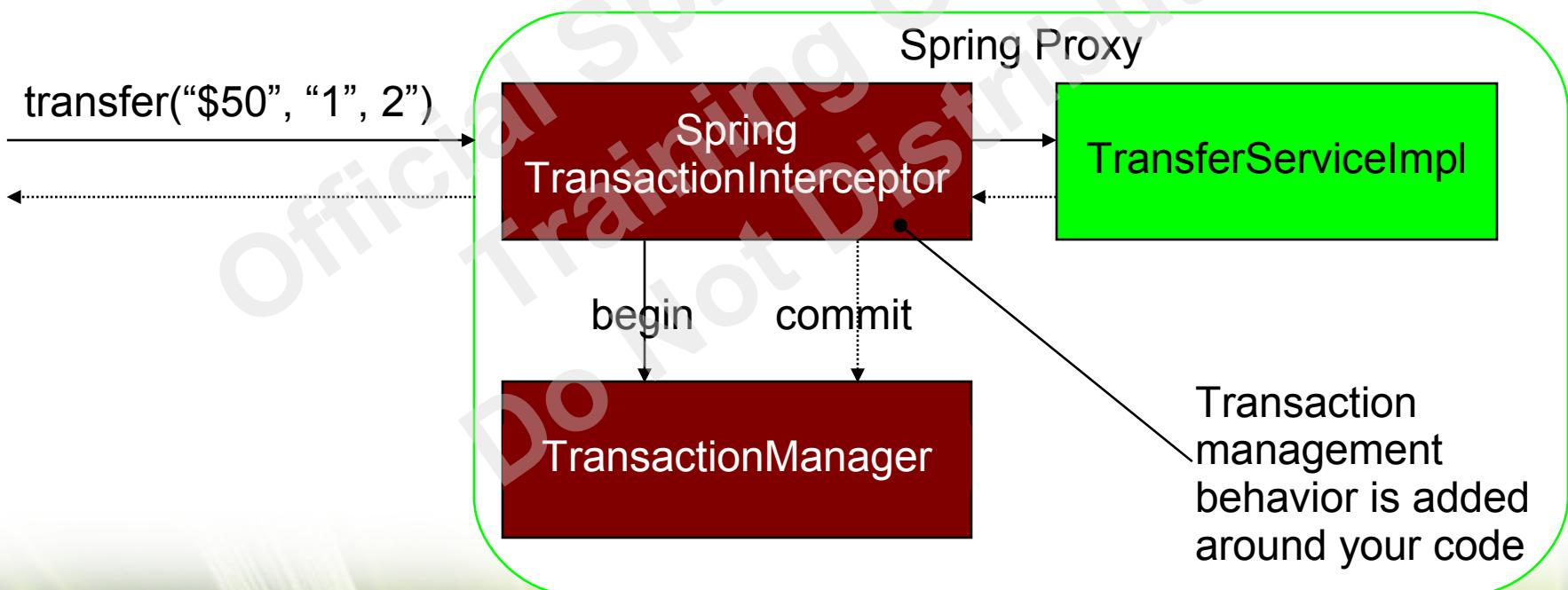
transfer("50", "1", 2")



Proxy classes are created in the init phase by dedicated *BeanPostProcessors*

Proxy Power

- A BeanPostProcessor may wrap your beans in a dynamic proxy and add behavior to your application logic *transparently*



Topics in this session

- Introduction
- The initialization phase
- The use phase
- **The destruction phase**



The Lifecycle of a Spring Application Context (3) - The Destruction Phase



- When you close a context the destruction phase completes

```
ConfigurableApplicationContext context = ...  
// Destroy the application  
context.close();
```

- But exactly what happens in this phase?

Inside the Application Context Destruction Lifecycle



- Destroy bean instances if instructed
- Destroy itself
 - The context is not usable again
- All *destroy* methods are called

```
public class TransferServiceImpl {  
  
    @PreDestroy  
    public void clearCache() {  
        //...  
    }  
}
```

By Annotation

```
<bean id="accountRepository"  
      class="app.impl.AccountRespository"  
      destroy-method="clearCache">  
    ...  
</bean>
```

Using XML

Topics Covered

- Spring Lifecycle
 - The initialization phase
 - Bean Post Processors for *initialization* and *proxies*
 - The use phase
 - Proxies at Work – most of Spring's “magic” uses a proxy
 - The destruction phase
 - Allow application to terminate cleanly





Testing Spring Applications

Unit Testing without Spring,
and Integration Testing with Spring

Topics in this session



- **Test Driven Development**
- Unit Testing vs. Integration Testing
- Integration Testing with Spring
- Environment Abstraction and Profiles

Official SpringSource
Training Only
Do Not Distribute

What is TDD



- TDD = Test Driven Development
- Is it writing tests before the code? Is it writing tests at the same time as the code?
 - That is not what is most important
- TDD is about:
 - writing automated tests that verify code actually works
 - Driving development with well defined requirements in the form of tests

"But I Don't Have Time to Write Tests!"



- Every development process includes testing
 - Either automated or manual
- Automated tests result in a faster development cycle overall
 - Your IDE is better at this than you are
- Properly done TDD is *faster* than development without tests

TDD and Agility



- Comprehensive test coverage provides confidence
- Confidence enables refactoring
- Refactoring is essential to agile development

Official Spring Source
Training
Do Not Distribute

TDD and Design



- Testing makes you think about your design
- If your code is hard to test then the design should be reconsidered

Official SpringSource
Training Only
Do Not Distribute

TDD and Focus



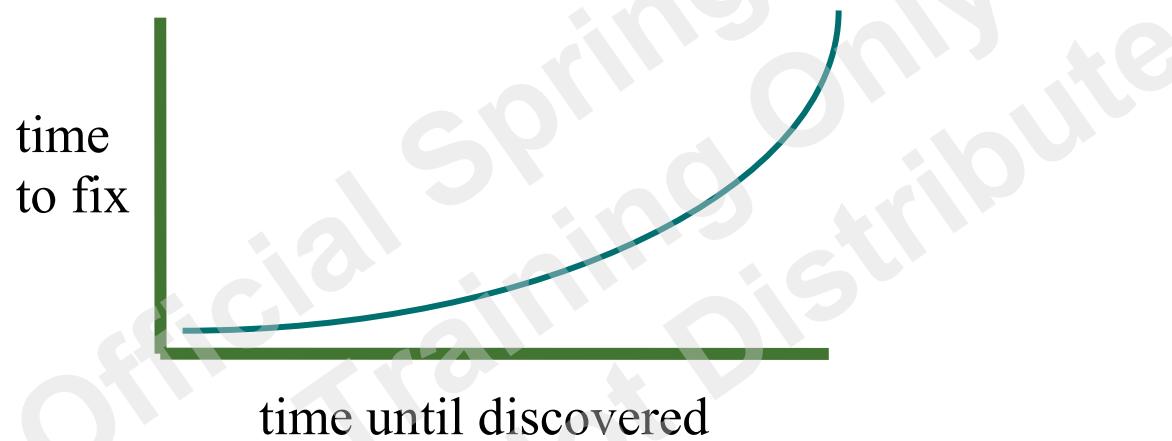
- A test case helps you focus on what matters
- It helps you not to write code that you don't need
- Find problems early

Official SpringSource
Training Only
Do Not Distribute

Benefits of Continuous Integration



- The cost to fix a bug grows exponentially in proportion to the time before it is discovered



- Continuous Integration (CI) focuses on reducing the time before the bug is discovered
 - Effective CI requires automated tests

Topics in this session



- Test Driven Development
- **Unit Testing vs. Integration Testing**
- Integration Testing with Spring
- Environment Abstraction and Profiles

Official Training Only
Do Not Distribute

Unit Testing vs. Integration Testing

- Unit Testing
 - Tests one unit of functionality
 - Keeps dependencies minimal
 - Isolated from the environment (including Spring)
- Integration Testing
 - Tests the interaction of multiple units working together
 - Integrates infrastructure

Unit Testing

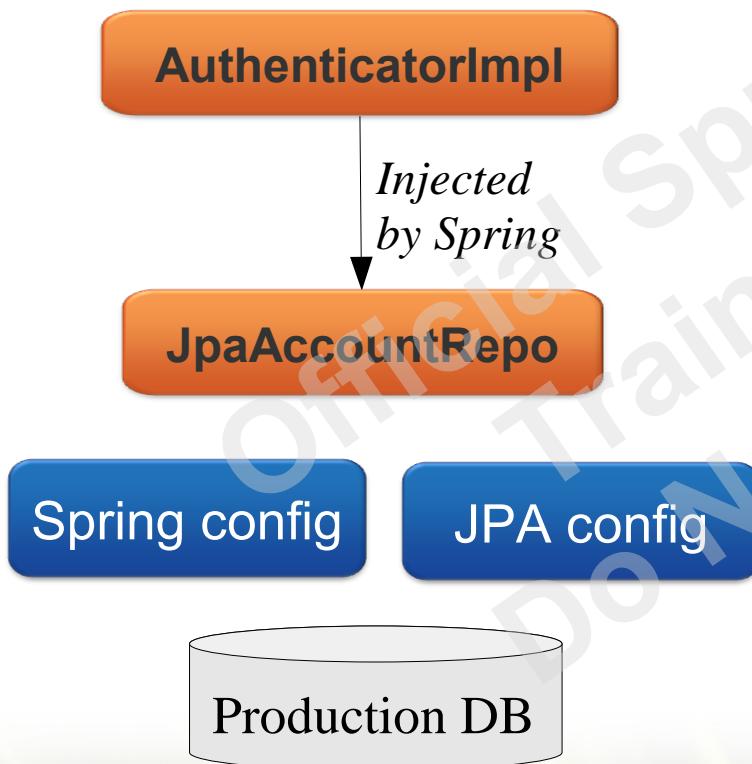


- Remove links with dependencies
 - The test shouldn't fail because of external dependencies
 - Spring is also considered as a dependency
- 2 ways to create a “testing-purpose” implementation of your dependencies:
 - **Stubs** Create a simple test implementation
 - **Mocks** Dependency class generated at startup-time using a “Mocking framework”

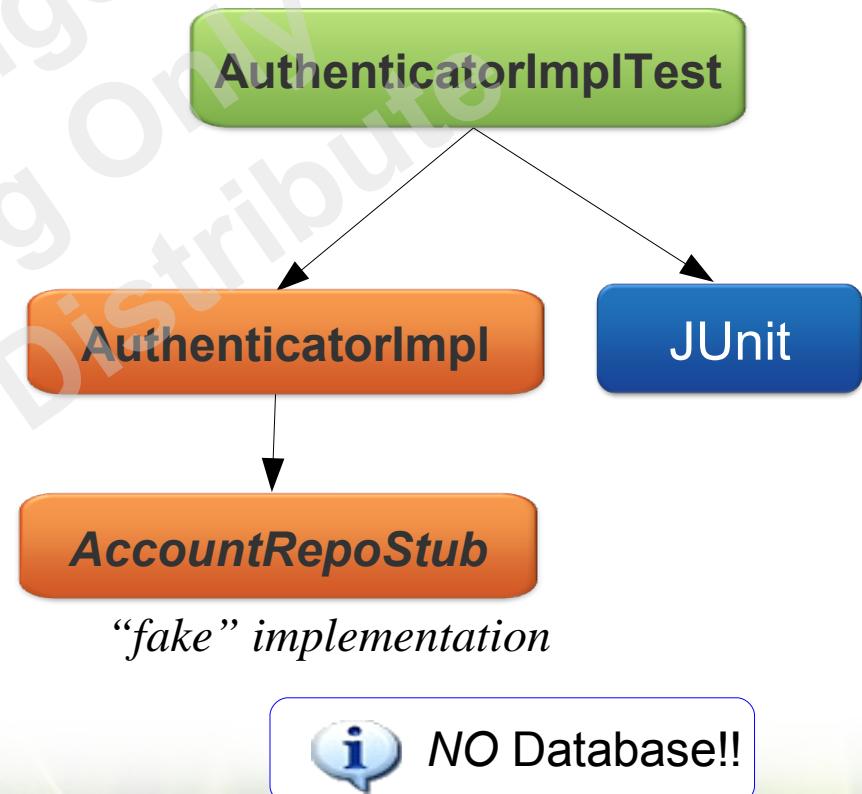
Unit Testing example



- Production mode



- Unit test with Stubs



Example Unit to be Tested



```
public class AuthenticatorImpl implements Authenticator {  
    private AccountRepository accountRepository;  
  
    public AuthenticatorImpl(AccountRepository accountRepository) {  
        this.accountRepository = accountRepository;  
    }  
  
    public boolean authenticate(String username, String password) {  
        Account account = accountRepository.getAccount(username);  
  
        return account.getPassword().equals(password);  
    }  
}
```

External dependency

Unit *business logic*
– 2 paths: success or fail

Note: Validation failure paths ignored for simplicity

Implementing a Stub



- Class created manually
 - Implements Business interface

```
class StubAccountRepository implements AccountRepository {  
    public Account getAccount(String user) {  
        return "lisa".equals(user) ? new Account("lisa", "secret") : null;  
    }  
}
```

A callout box labeled "Simple state" points to the line of code that creates a new Account object.



Unit Test using a Stub



```
import org.junit.Before; import org.junit.Test; ...
```

```
public class AuthenticatorImplTests {
```

```
    private AuthenticatorImpl authenticator;
```

```
@Before public void setUp() {
```

```
    authenticator = new AuthenticatorImpl(  
        new StubAccountRepository());
```

```
}
```

```
@Test public void successfulAuthentication() {
```

```
    assertTrue(authenticator.authenticate("lisa", "secret"));
```

```
}
```

```
@Test public void invalidPassword() {
```

```
    assertFalse(authenticator.authenticate("lisa", "invalid"));
```

```
}
```

Spring **not** in charge of
injecting dependencies

OK scenario

KO scenario

Unit Testing with Stubs



- Advantages
 - Easy to implement and understand
 - Reusable
- Disadvantages
 - Change to an interface requires change to stub
 - Your stub must implement *all* methods
 - even those not used by a specific scenario
 - If a stub is reused refactoring can break other tests

Steps to Testing with a Mock



1. Use a mocking library to generate a mock object
 - Implements the dependent interface on-the-fly
2. Record the mock with expectations of how it will be used for a scenario
 - What methods will be called
 - What values to return
3. Exercise the scenario
4. Verify mock expectations were met

Example: Using a Mock - I

- Setup
 - A Mock class is created at startup time

```
import static org.easymock.classextensions.EasyMock.*;  
  
public class AuthenticatorImplTests {  
    private AccountRepository accountRepository  
        = createMock(AccountRepository.class);  
  
    private AuthenticatorImpl authenticator  
        = new AuthenticatorImpl(accountRepository);  
  
    // continued on next slide ...
```

static import

Implementation of interface
AccountRepository is created

Example: Using a Mock - II

```
// ... continued from previous slide
```

```
@Test public void validUserWithCorrectPassword() {  
    expect(accountRepository.getAccount("lisa")).  
        andReturn(new Account("lisa", "secret"));  
  
    replay(accountRepository); ←  
  
    boolean res = authenticator.  
        authenticate("lisa", "secret"); ←  
    assertTrue(res);  
  
    verify(accountRepository); ←  
}  
}
```

Recording

What behavior to expect?

Recording → Playback

“playback” mode

Mock now fully available

Verification

*No planned method call
has been omitted*

Mock Considerations



- Several mocking libraries available
 - Mockito, JMock, EasyMock
- Advantages
 - No additional class to maintain
 - You only need to setup what is necessary for the scenario you are testing
 - Test behavior as well as state
 - Were all mocked methods used? If not, why not?
- Disadvantages
 - A little harder to understand at first

Mocks or Stubs?



- You will probably use both
- General recommendations
 - Favor mocks for non-trivial interfaces
 - Use stubs when you have simple interfaces with repeated functionality
 - Always consider the specific situation
- Read “Mocks Aren’t Stubs” by Martin Fowler
 - <http://www.martinfowler.com/articles/mocksArentStubs.html>

Topics in this session



- Test Driven Development
- Unit Testing vs. Integration Testing
- **Integration Testing with Spring**
- Environment Abstraction and Profiles

Integration Testing

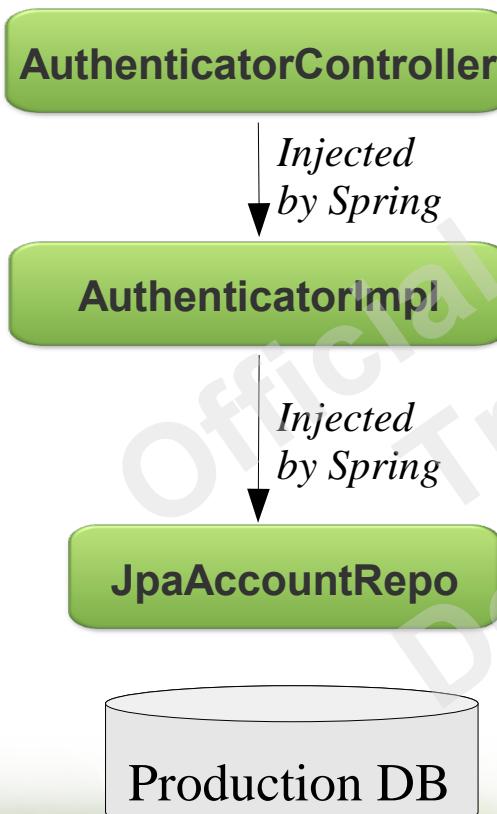


- Tests the interaction of multiple units
- Tests application classes in the context of their surrounding infrastructure
 - Infrastructure may be “scaled down”
 - e.g. use Apache DBCP connection pool instead of container-provider pool obtained through JNDI

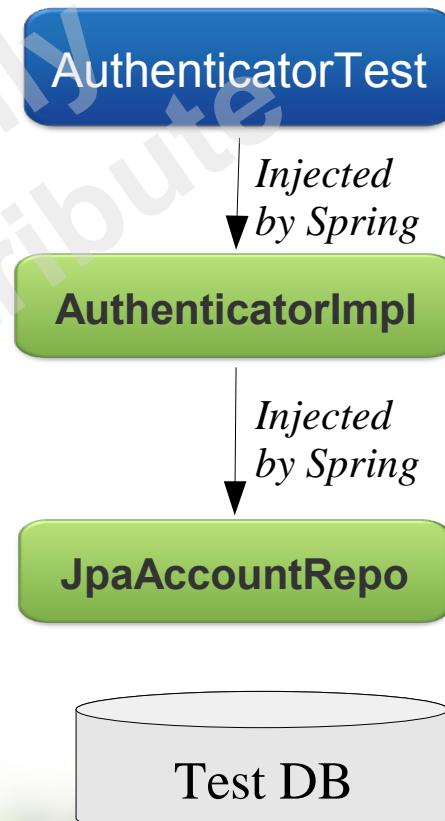
Integration test example



- Production mode



- Integration test



Spring's Integration Test Support



- Packaged as a separate module
 - spring-test.jar
- Consists of several JUnit test support classes
- Central support class is SpringJUnit4ClassRunner
 - Caches a *shared* ApplicationContext across test methods

Using Spring's Test Support



Run with Spring support

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:system-test-config.xml")
public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() {
        TransferConfirmation conf = transferService.transfer(...);
        ...
    }
}
```

Point to system test configuration file

Define bean to test

Test the system as normal

No need for @Before method

@ContextConfiguration config samples



```
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration  
public final class TransferServiceTests { ... }
```

Defaults to
\${classname}-context.xml
in same package

```
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration("classpath:com/acme/system-test-config.xml")  
public final class TransferServiceTests { ... }
```

```
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration({"classpath:config-1.xml", "file:db-config.xml"})  
public final class TransferServiceTests { ... }
```

Multiple files allowed

```
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(classes={AppConfig.class, SystemConfig.class})  
public final class TransferServiceTests { ... }
```

For Java Configuration classes
(annotated with `@Configuration`)

Multiple test methods

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:system-test-config.xml")
public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() {
        ...
    }

    @Test
    public void failedTransfer() {
        ...
    }
}
```

The ApplicationContext is instantiated only *once* for all tests that use the same set of config files (even across test classes)



Annotate test method with `@DirtiesContext` to force recreation of the cached ApplicationContext *if* method changes the contained beans

Benefits of Testing with Spring



- No need to deploy to an external container to test application functionality
 - Run everything quickly inside your IDE
- Allows reuse of your configuration between test and production environments
 - Application configuration logic is typically reused
 - Infrastructure configuration is environment-specific
 - DataSources
 - JMS Queues

Topics in this session



- Test Driven Development
- Unit Testing vs. Integration Testing
- Integration Testing with Spring
- **Environment Abstraction and Profiles**

Environment Abstraction



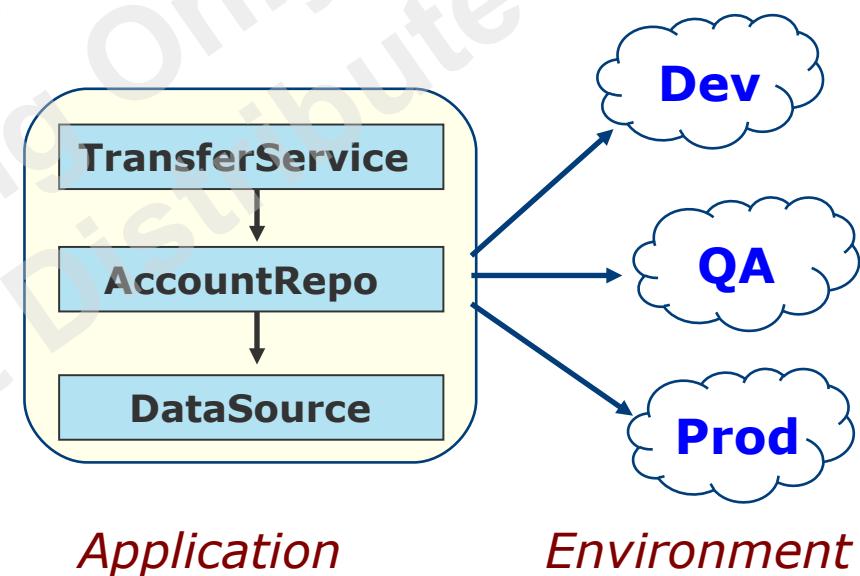
Spring 3.1

- Group bean definitions for use in specific environments
 - Development, Testing, UAT, QA, Production
 - Multiple deployment setups
- Custom resolution of placeholders
 - Dependent on the environment
 - Hierarchy of property sources
- Injectable environment abstraction API
 - org.springframework.core.env.Environment

Bean Definition Profiles



- XML
 - XML profile attribute on the <beans> element
 - <beans> can be nested
- Java-based
 - @Profile annotation on @Configuration classes
 - Or on individual @Component classes



Profile Configuration XML



- All bean definitions

```
<beans xmlns="http://www.springframework.org/schema/beans ...  
    profile="dev"> ... </beans>
```

Profile applies to all beans

- Subset of bean definitions

```
<beans xmlns="http://www.springframework.org/schema/beans ...>  
    <bean id="dataSource" ... /> <!-- Available to all profiles -->  
    ...  
    <beans profile="dev"> ... </beans>  
    <beans profile="prod"> ... </beans>  
</beans>
```

Different subset
of beans for each
profile, plus some
shared beans

Sample XML Configuration



```
<beans xmlns="http://www.springframework.org/schema/beans  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns:jdbc="http://www.springframework.org/schema/jdbc  
       xsi:schemaLocation="...>  
  
<beans profile="dev">  
  
    <jdbc:embedded-database id="dataSource">  
        <jdbc:script location="classpath:com/bank/sql/schema.sql"/>  
        <jdbc:script location="classpath:com/bank/sql/test-data.sql"/>  
    </jdbc:embedded-database>  
  
</beans>  
  
<beans profile="production">  
  
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource" />  
  
</beans>  
  
</beans>
```

Profiles in an Integration Test



- Using the `@ActiveProfiles` annotation
 - Make one or more profiles active

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:system-test-config.xml")
@ActiveProfiles("dev")
public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() {
        TransferConfirmation conf = transferService.transfer(...);
        ...
    }
}
```

Profiles and Annotation-based injection



- **@ActiveProfiles** inside the test class
- **@Profile** inside the Component class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("infra-test-conf.xml")
@ActiveProfiles("jdbc")
```

```
public class TransferServiceTests
{...}
```

```
@Repository
@Profile("jdbc")
```

```
public class
JdbcAccountRepository
{ ... }
```



Only the beans with the current profile are component-scanned

Profiles in Java Configuration



- Using the `@Profile` annotation
 - Can also use on a `@Component`

```
@Configuration  
@Profile("dev")  
public final class DevConfig {  
  
    @Bean  
    public DataSource dataSource() {  
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();  
        return builder.setName("testdb")  
            .setType(EmbeddedDatabaseType.HSQL)  
            .addScript("classpath:/testdb/schema.db")  
            .addScript("classpath:/testdb/test-data.db").build();  
    }  
}
```

```
@Component  
@Profile("dev")  
public class DevOnlyClass { ... }
```

Quiz: Which of the following is/are selected?



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("infra-test-conf.xml")
@ActiveProfiles("jpa")
```

```
public final class TransferServiceTests
{...}
```

?

?

?

```
@Service
public class
TransferServiceImpl { ...}
```

```
@Repository
@Profile("jpa")
public class
JpaAccountRepository
{ ...}
```

```
@Repository
@Profile("jdbc")
public class
JdbcAccountRepository
{ ...}
```

Ways to activate a Profile



- A profile must be activated at run-time
 - Using @ActiveProfiles inside an Integration Test (above)
 - Programmatically on ApplicationContext (next)
 - Commonly used when Spring is started from a Main method
 - System property in the command line

```
-Dspring.profiles.active=dev
```

- In web.xml (for Web-based application)

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>jpa</param-value>
</context-param>
```

web.xml

Programmatically

- Commonly used for Batch programming
- Select profile API on environment
 - Every application context now has one
 - Must set profile *before* loading XML files
 - Can no longer pass via constructor

```
ClassPathXmlApplicationContext applicationContext =  
    new ClassPathXmlApplicationContext();  
applicationContext.getEnvironment(). setActiveProfiles("dev");  
applicationContext.setConfigLocations("classpath:/application-config.xml");  
applicationContext.refresh(); // Forces context to load beans
```

Property Sources



- PropertySources
 - Different property values in different environments

```
@Configuration  
@Profile("dev")  
@PropertySource("classpath:/config/dev.properties")  
public final class DevConfig { ..}
```

- Property Resolvers
 - Same property may be defined multiple times
 - Works out which value to use
 - Used by <context:property-placeholder/> since 3.1

Profiles and Properties



```
<import resource="classpath:config/${current.env}-config.xml"/>

<context:property-placeholder properties-ref="configProps"/>

<beans profile="dev">
    <util:properties id="configProps" location="config/dev.properties">
</beans>

<beans profile="prod">
    <util:properties id="configProps" location="config/prod.properties">
</beans>
```

current.env=dev
database.url=jdbc:derby:/test
database.user=tester

current.env=prod
database.url=jdbc:oracle:thin:...
database.user=admin

Summary

- Testing is an *essential* part of any development
- Unit testing tests a class in isolation
 - External dependencies should be minimized
 - Consider creating stubs or mocks to unit test
 - *You don't need Spring to unit test*
- Integration testing tests the interaction of multiple units working together
 - Spring provides good integration testing support
 - Profiles for different test & deployment configurations



LAB

Testing Spring Applications

Official SpringSource
Training Only
Do Not Distribute



Developing Aspects with Spring AOP

Aspect-Oriented Programming for
Declarative Enterprise Services

Topics in this session



- **What Problem Does AOP Solve?**
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Advanced topics
 - Named Pointcuts
 - Context selecting pointcuts
 - Working with annotations

What Problem Does AOP Solve?



- Aspect-Oriented Programming (AOP) enables modularization of cross-cutting concerns

Official SpringSource
Training Only
Do Not Distribute

What are Cross-Cutting Concerns?



- Generic functionality that is needed in many places in your application
- Examples
 - Logging and Tracing
 - Transaction Management
 - Security
 - Caching
 - Error Handling
 - Performance Monitoring
 - Custom Business Rules

An Example Requirement



- Perform a role-based security check before **every** application method

A sign this requirement is a cross-cutting concern



Implementing Cross Cutting Concerns Without Modularization



- Failing to modularize cross-cutting concerns leads to two things
 1. Code tangling
 - A coupling of concerns
 2. Code scattering
 - The same concern spread across modules

Symptom #1: Tangling



```
public class RewardNetworkImpl implements RewardNetwork {  
    public RewardConfirmation rewardAccountFor(Dining dining) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        Account a = accountRepository.findByCreditCard(...);  
        Restaurant r = restaurantRepository.findByMerchantNumber(...);  
        MonetaryAmount amt = r.calculateBenefitFor(account, dining);  
        ...  
    }  
}
```

A callout box with the text "Mixing of concerns" has an arrow pointing from the "if" statement in the code to the "accountRepository" and "restaurantRepository" lines.

Symptom #2: Scattering

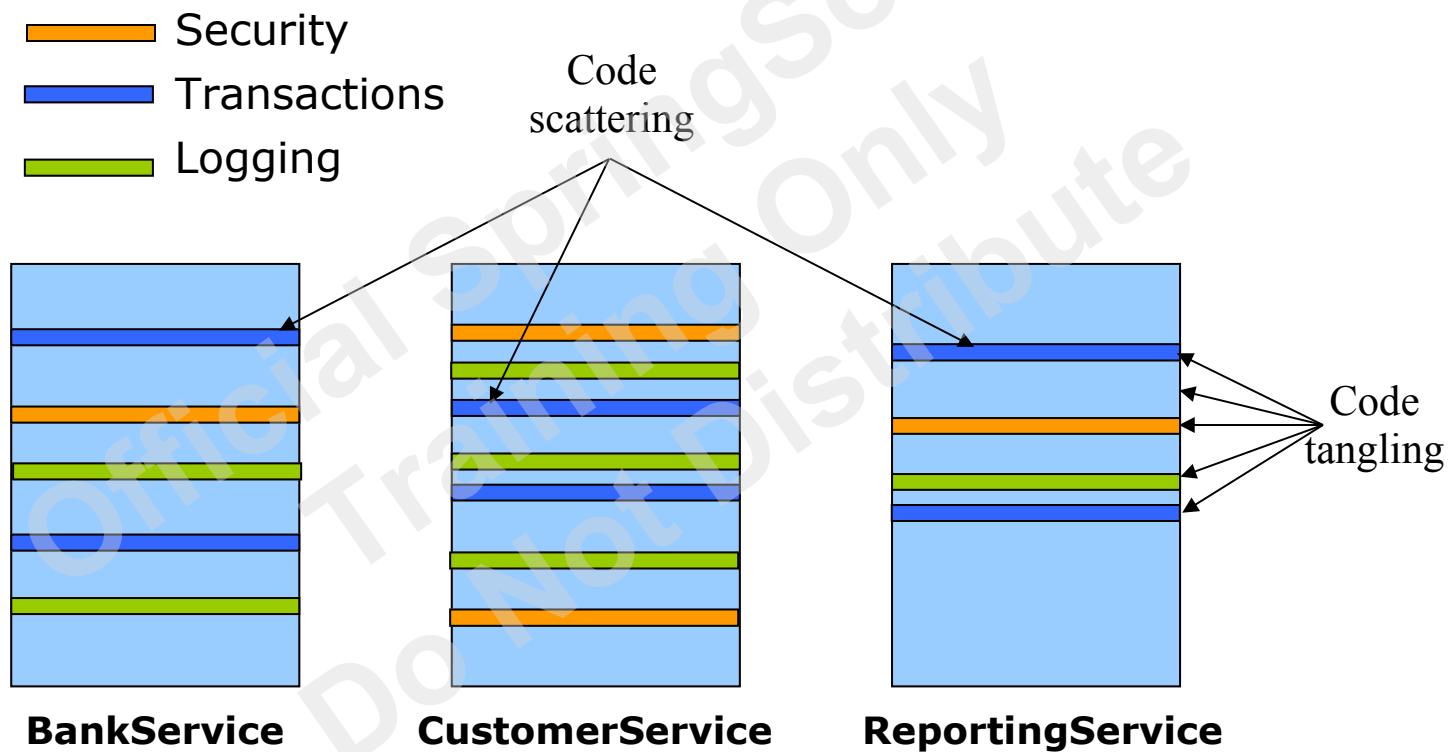


```
public class HibernateAccountManager implements AccountManager {  
    public Account getAccountForEditing(Long id) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        ...  
    }
```

Duplication

```
public class HibernateMerchantReportingService implements  
MerchantReportingService {  
    public List<DiningSummary> findDinings(String merchantNumber,  
                                            DateInterval interval) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        ...  
    }
```

System Evolution Without Modularization



Aspect Oriented Programming (AOP)



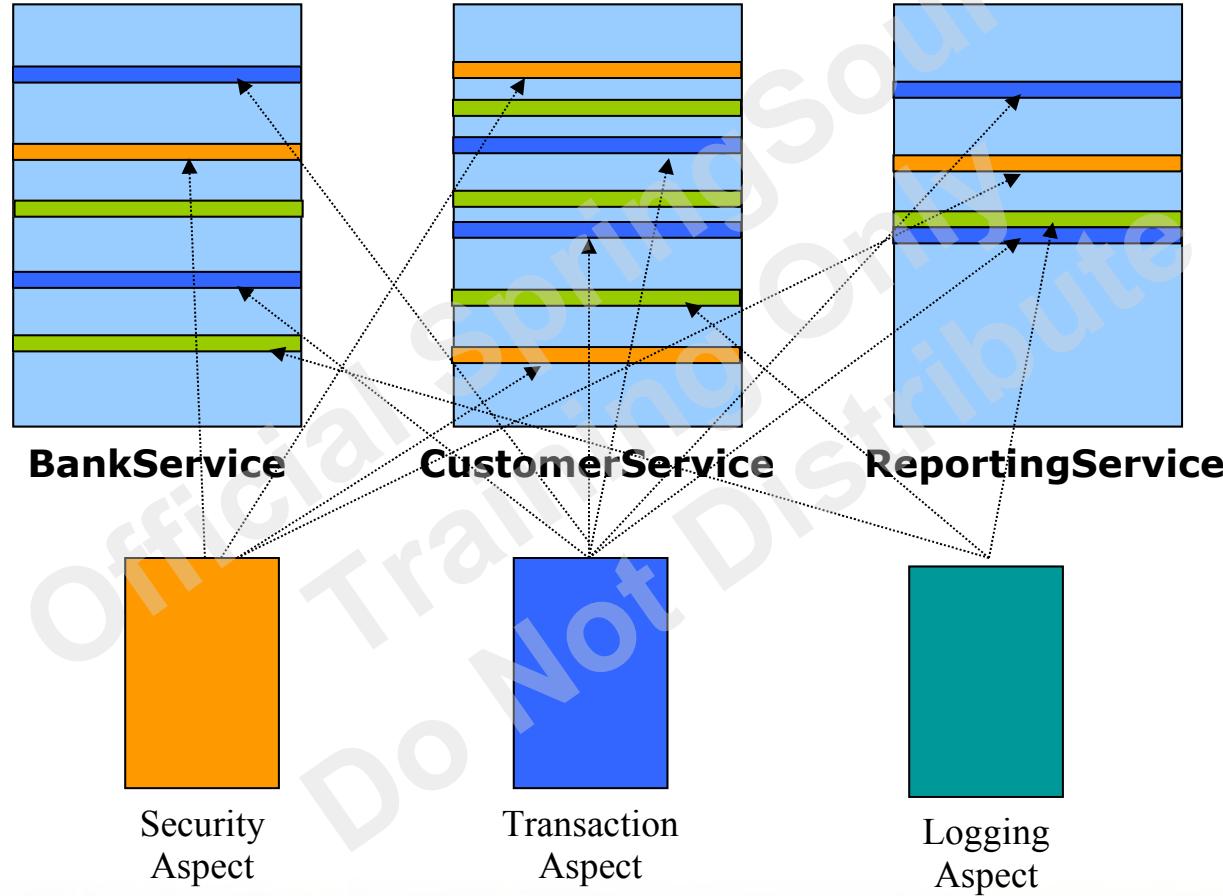
- Aspect-Oriented Programming (AOP) enables modularization of cross-cutting concerns
 - To avoid tangling
 - To eliminate scattering

How AOP Works



1. Implement your mainline application logic
 - Focusing on the core problem
2. Write aspects to implement your cross-cutting concerns
 - Spring provides many aspects out-of-the-box
3. Weave the aspects into your application
 - Adding the cross-cutting behaviours to the right places

System Evolution: AOP based



Leading AOP Technologies



- AspectJ
 - Original AOP technology (first version in 1995)
 - Offers a full-blown Aspect Oriented Programming language
 - Uses byte code modification for aspect weaving
- Spring AOP
 - Java-based AOP framework with AspectJ integration
 - Uses dynamic proxies for aspect weaving
 - Focuses on using AOP to solve enterprise problems
 - **The focus of this session**

Topics in this session



- What Problem Does AOP Solve?
- **Core AOP Concepts**
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Advanced topics
 - Named Pointcuts
 - Context selecting pointcuts
 - Working with annotations

Core AOP Concepts



- Join Point
 - A point in the execution of a program such as a method call or field assignment
- Pointcut
 - An expression that selects one or more Join Points
- Advice
 - Code to be executed at a Join Point that has been selected by a Pointcut
- Aspect
 - A module that encapsulates pointcuts and advice

Topics in this session



- What Problem Does AOP Solve?
- Core AOP Concepts
- **Quick Start**
- Defining Pointcuts
- Implementing Advice
- Advanced topics
 - Named Pointcuts
 - Context selecting pointcuts
 - Working with annotations

AOP Quick Start



- Consider this basic requirement

Log a message every time a property is about to change

- How can you use AOP to meet it?

An Application Object Whose Properties Could Change



```
public class SimpleCache implements Cache, BeanNameAware
{
    private int cacheSize;
    private DataSource dataSource;
    private String name;

    public void setCacheSize(int size) { cacheSize = size; }

    public void setDataSource(DataSource ds) { dataSource = ds; }

    // Allow bean to know its name – for logging
    public void setBeanName(String beanName) { name = beanName; }

    public String toString() { return name; }

    ...
}
```

```
public interface Cache {
    public void setCacheSize(int size);
}
```

BeanNameAware for logging only – *nothing* to do with AOP

Implement the Aspect

```
@Aspect  
public class PropertyChangeTracker {  
    private Logger logger = Logger.getLogger(getClass());  
  
    @Before("execution(void set*(*))")  
    public void trackChange() {  
        logger.info("Property about to change...");  
    }  
}
```

Configure the Aspect as a Bean



aspects-config.xml

```
<beans>
  <aop:aspectj-autoproxy>
    <aop:include name="propertyChangeTracker" />
  </aop:aspectj-autoproxy>
  <bean id="propertyChangeTracker" class="example.PropertyChangeTracker" />
</beans>
```

Configures Spring to apply the @Aspect to your beans

Include the Aspect Configuration



application-config.xml

```
<beans>

    <import resource=“aspects-config.xml”/>

    <bean name=“cache-A” class=“example.SimpleCache” ..>
    <bean name=“cache-B” class=“example.SimpleCache” ..>
    <bean name=“cache-C” class=“example.SimpleCache” ..>

</beans>
```

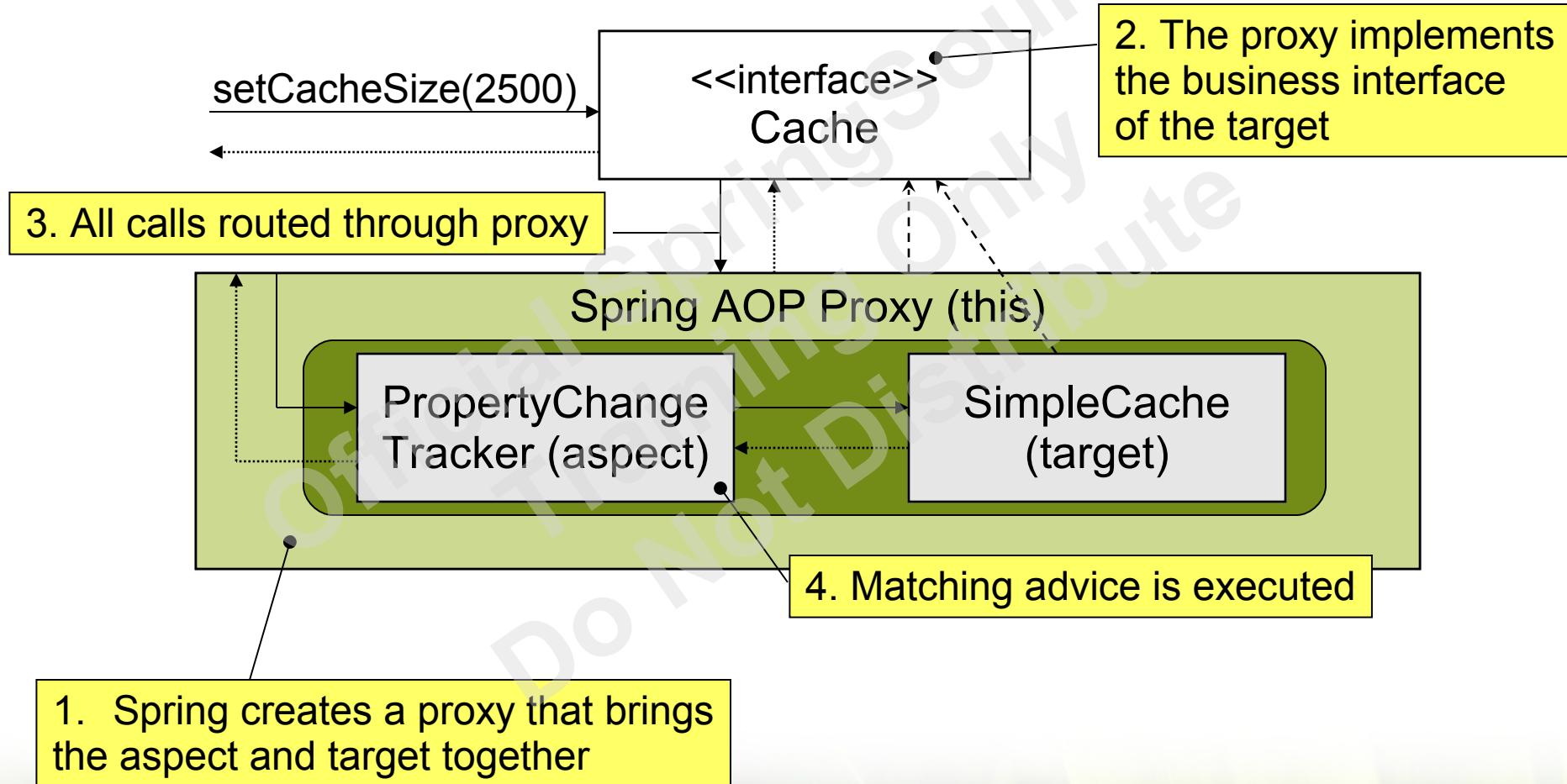
Test the Application



```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("application-config.xml");  
Cache cache = (Cache) context.getBean("cache-A");  
cache.setCacheSize(2500);
```

INFO: Property about to change...

How Aspects are Applied



Tracking Property Changes – With Context



- Context provided by the *JoinPoint* parameter

```
@Aspect  
public class PropertyChangeTracker {  
    private Logger logger = Logger.getLogger(getClass());  
  
    @Before("execution(void set*(*))")  
    public void trackChange(JoinPoint point) {  
        String name = point.getSignature().getName();  
        Object newValue = point.getArgs()[0];  
        logger.info(name + " about to change to " + newValue +  
                    " on " + point.getTarget());  
    }  
}
```

Context about the intercepted point

toString() returns bean-name

INFO: setCacheSize about to change to 2500 on cache-A

Topics in this session



- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- **Defining Pointcuts**
- Implementing Advice
- Advanced topics
 - Named Pointcuts
 - Context selecting pointcuts
 - Working with annotations

Defining Pointcuts



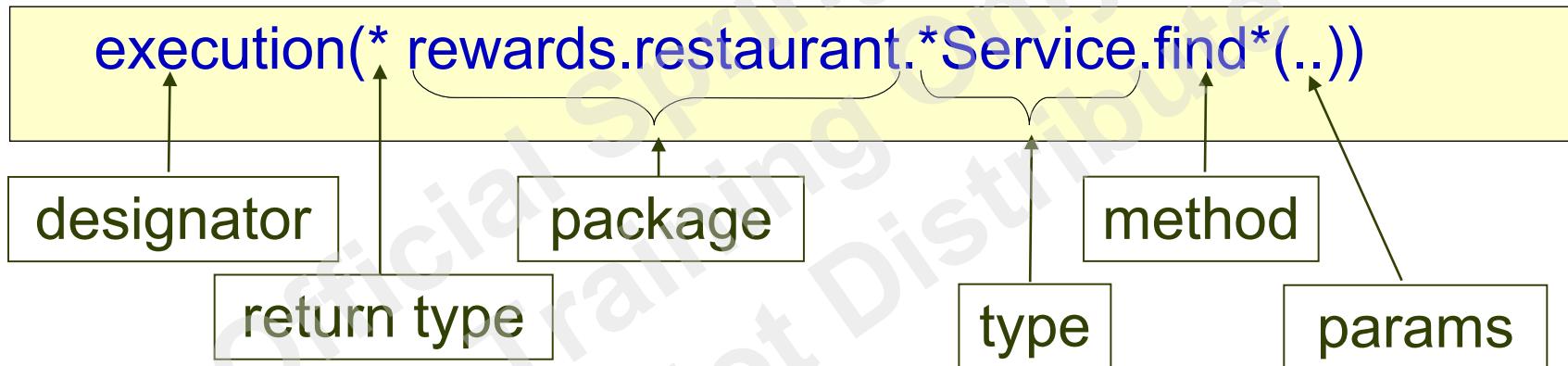
- With Spring AOP you write pointcuts using AspectJ's pointcut expression language
 - For selecting *where* to apply advice
- Complete expression language reference available at
 - <http://www.eclipse.org/aspectj>

Common Pointcut Designator



- execution(<method pattern>)
 - The method must match the pattern
- Can chain together to create composite pointcuts
 - && (and), || (or), ! (not)
- Method Pattern
 - [Modifiers] ReturnType [ClassType]MethodName ([Arguments]) [throws ExceptionType]

Writing expressions



Execution Expression Examples



`execution(void send*(String))`

- Any method starting with *send* that takes a single *String* parameter and has a *void* return type

`execution(* send(*))`

- Any method named *send* that takes a single parameter

`execution(* send(int, ..))`

- Any method named *send* whose first parameter is an *int* (the *“..”* signifies 0 or more parameters may follow)

Execution Expression Examples



`execution(void example.MessageServiceImpl.*(..))`

- Any visible *void* method in the `MessageServiceImpl` class

`execution(void example.MessageService.send(*))`

- Any *void* method named *send* in any object of type `MessageService` (that include possible child classes or implementors of `MessageService`)

`execution(@javax.annotation.security.RolesAllowed void send*(..))`

- Any *void* method starting with *send* that is annotated with the `@RolesAllowed` annotation

Execution Expression Examples working with packages



`execution(* rewards.*.restaurant.*.*(..))`

- There is one directory between *rewards* and *restaurant*

`execution(* rewards..restaurant.*.*(..))`

- There may be several directories between *rewards* and *restaurant*

`execution(* *..restaurant.*.*(..))`

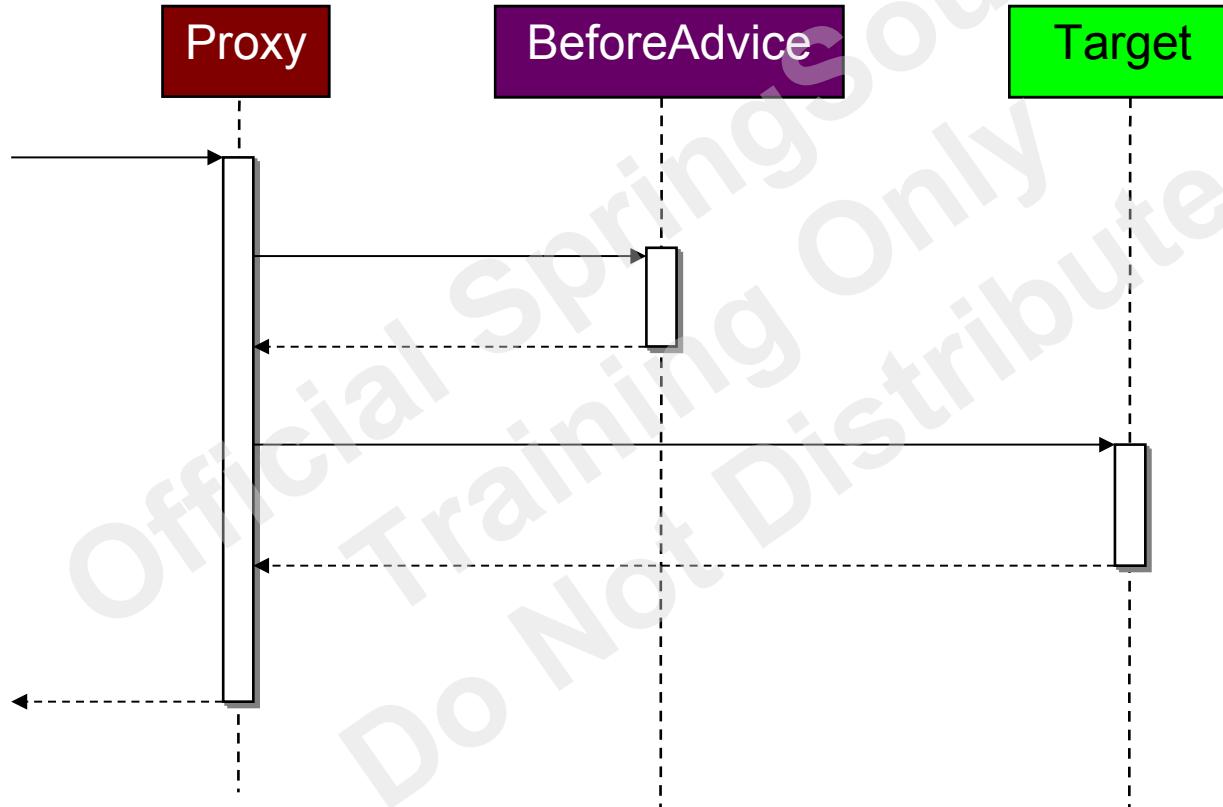
- Any sub-package called *restaurant*

Topics in this session



- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- **Implementing Advice**
- Advanced topics
 - Named Pointcuts
 - Context selecting pointcuts
 - Working with annotations

Advice Types: Before



Before Advice Example

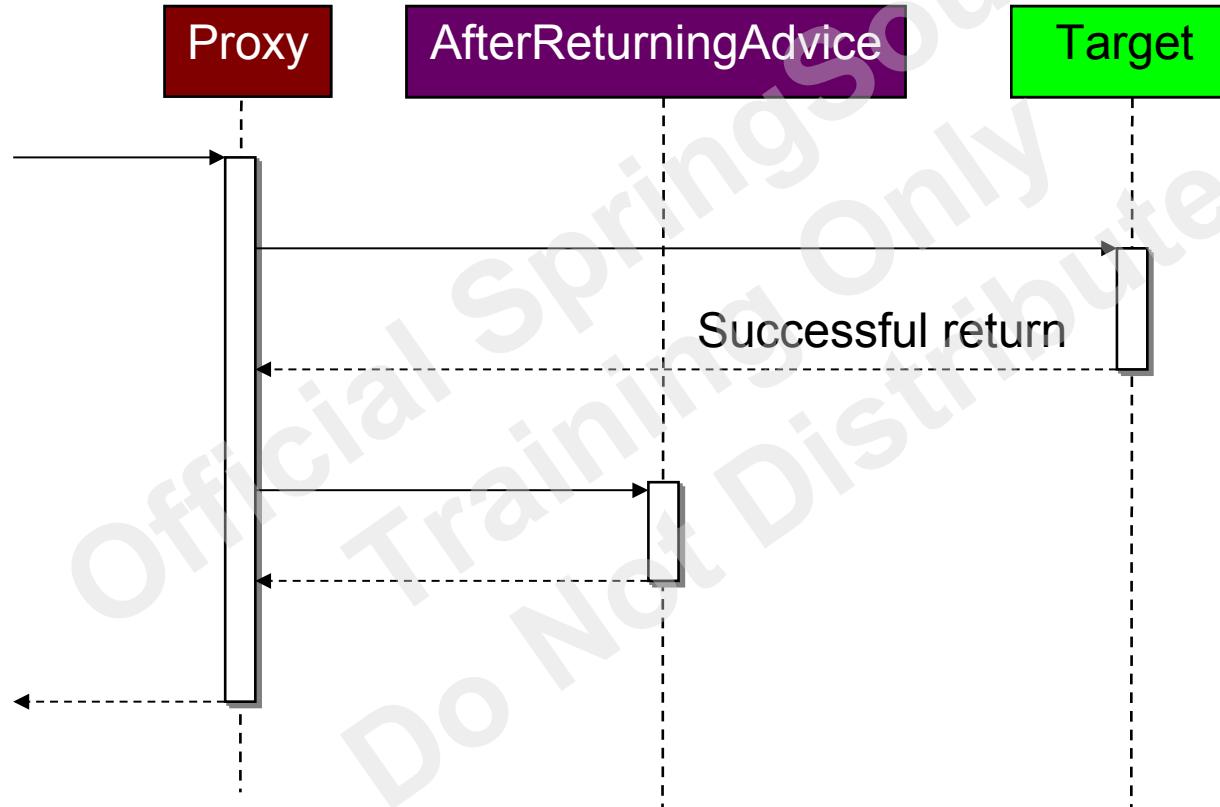


- Use `@Before` annotation
 - If the advice throws an exception, target will not be called

Track calls to all setter methods

```
@Aspect  
public class PropertyChangeTracker {  
    private Logger logger = Logger.getLogger(getClass());  
  
    @Before("execution(void set*(*))")  
    public void trackChange() {  
        logger.info("Property about to change...");  
    }  
}
```

Advice Types: After Returning



After Returning Advice - Example

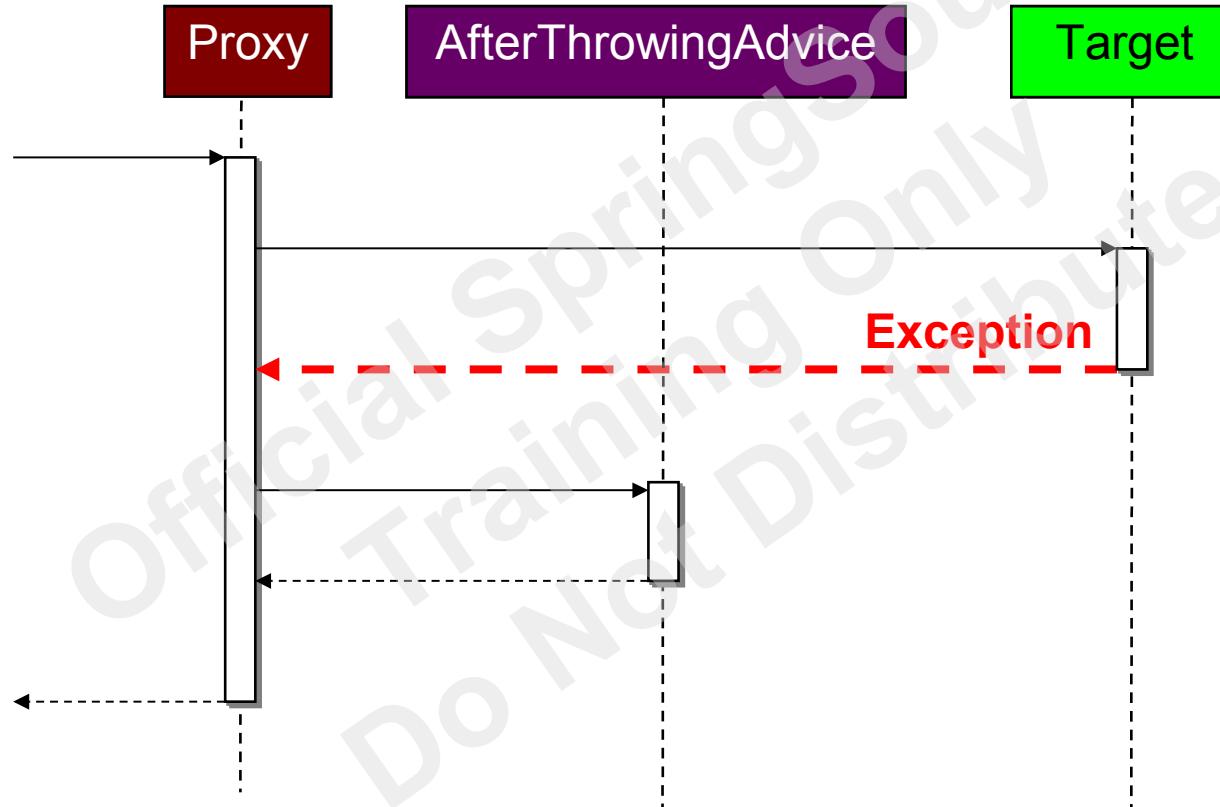


- Use `@AfterReturning` annotation with the *returning* attribute

Audit all operations in the service package that return a *Reward* object

```
@AfterReturning(value="execution(* service..*.*(..))",
               returning="reward")
public void audit(JoinPoint jp, Reward reward) {
    auditService.logEvent(jp.getSignature() +
        " returns the following reward object :" + reward.toString());
}
```

Advice Types: After Throwing



After Throwing Advice - Example



- Use `@AfterThrowing` annotation with the `throwing` attribute

Send an email every time a Repository class throws an exception of type `DataAccessException`

```
@AfterThrowing(value="execution(* *..Repository.*(..))", throwing="e")
public void report(JoinPoint jp, DataAccessException e) {
    mailService.emailFailure("Exception in repository", jp, e);
}
```

After Throwing Advice - Propagation



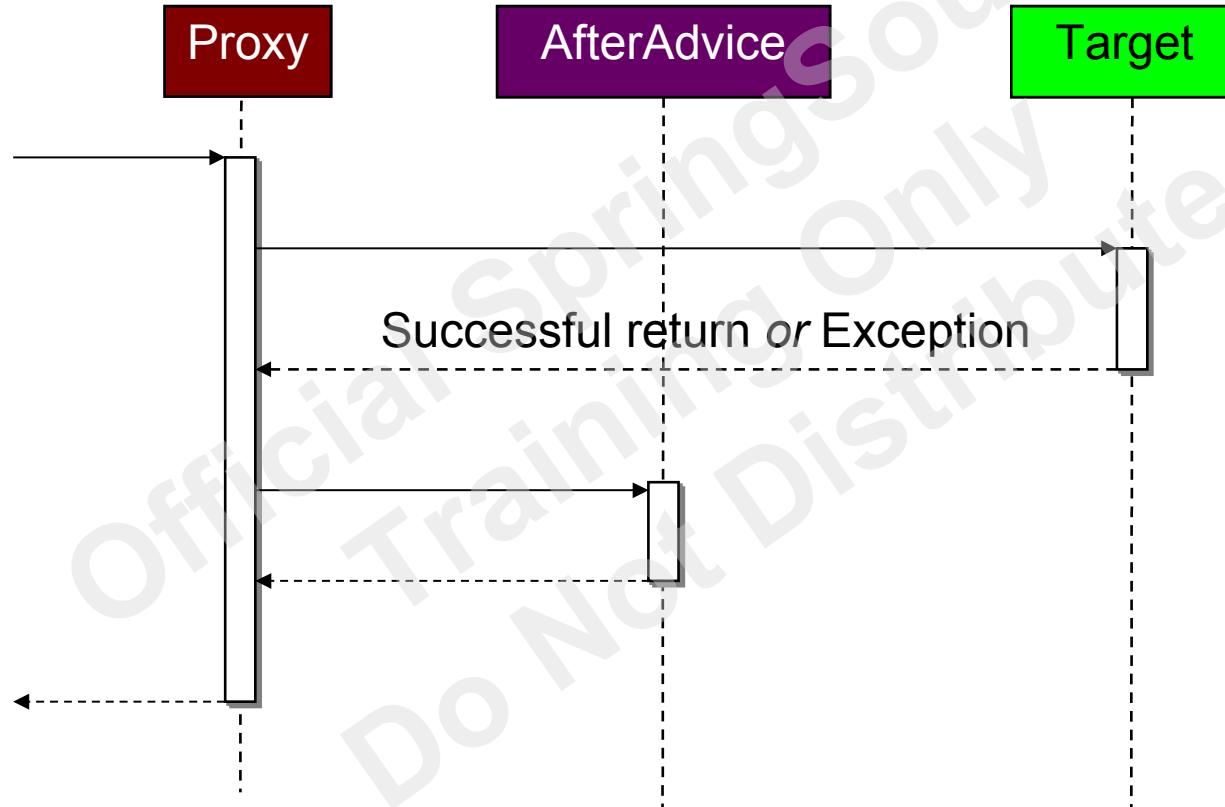
- The @AfterThrowing advice will not stop the exception from propagating
 - However it can throw a different type of exception

```
@AfterThrowing(value="execution(* *..Repository.*(..))", throwing="e")
public void report(JoinPoint jp, DataAccessException e) {
    mailService.emailFailure("Exception in repository", jp, e);
    throw new RewardsException(e);
}
```



If you wish to stop the exception from propagating any further, you can use an @Around advice (see later)

Advice Types: After



After Advice Example



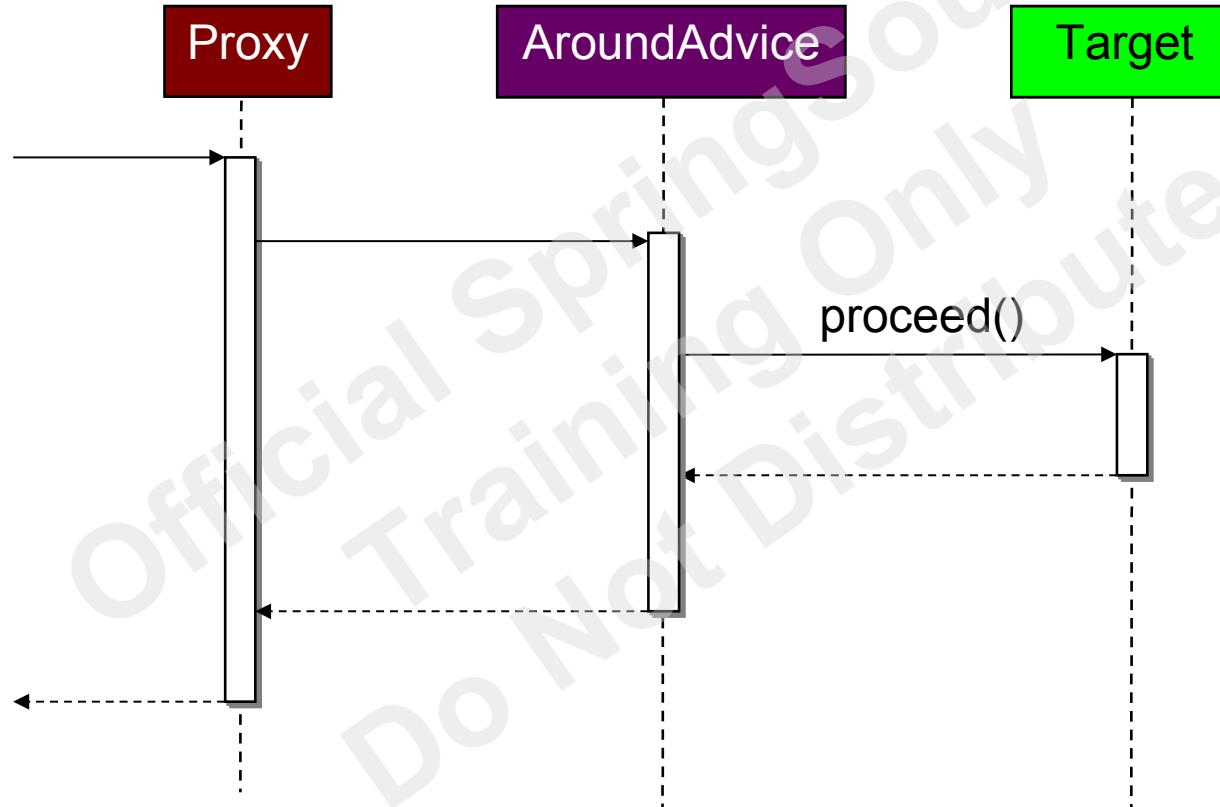
- Use `@After` annotation
 - Called regardless of whether an exception has been thrown by the target or not

Track calls to all update methods

```
@Aspect  
public class PropertyChangeTracker {  
    private Logger logger = Logger.getLogger(getClass());  
  
    @After("execution(void update*(*))")  
    public void trackUpdate() {  
        logger.info("An update has been attempted ...");  
    }  
}
```

We don't know how the method terminated

Advice Types: Around



Around Advice Example



- Use `@Around` annotation
 - `ProceedingJoinPoint` parameter
 - Inherits from `JoinPoint` and adds the `proceed()` method

Cache values returned by cacheable services

```
@Around("execution(@example.Cacheable * rewards.service..*.*(..))")
public Object cache(ProceedingJoinPoint point) throws Throwable {
    Object value = cacheStore.get(cacheKey(point));
    if (value == null) {
        value = point.proceed(); ← Proceed only if not already cached
        cacheStore.put(cacheKey(point), value);
    }
    return value;
}
```

Alternative Spring AOP Syntax - XML



- Annotation syntax is Java 5+ only
- XML syntax works on Java 1.4
- Approach
 - Aspect logic defined Java
 - Aspect configuration in XML
 - Uses the aop namespace

Tracking Property Changes - Java Code



```
public class PropertyChangeTracker {  
    public void trackChange(JoinPoint point) {  
        ...  
    }  
}
```

Aspect is a Plain Java Class with no Java 5 annotations

Tracking Property Changes - XML Configuration



- XML configuration uses the `aop` namespace

```
<aop:config>
  <aop:aspect ref="propertyChangeTracker">
    <aop:before pointcut="execution(void set*(*))" method="trackChange"/>
  </aop:aspect>
</aop:config>

<bean id="propertyChangeTracker" class="example.PropertyChangeTracker" />
```



LAB

Developing Aspects with Spring AOP

Official SpringSource
Training Only
Do Not Distribute

Topics in this session



- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- **Advanced topics**
 - **Named Pointcuts**
 - Context selecting pointcuts
 - Working with annotations
 - Limitations of Spring AOP

Named Pointcuts in XML

- A pointcut expression can have a name
 - Reuse it in multiple places

```
<aop:config>
  <aop:pointcut id="setterMethods" expression="execution(void set*(*))"/>

  <aop:aspect ref="propertyChangeTracker">
    <aop:after-returning pointcut-ref="setterMethods" method="trackChange"/>
    <aop:after-throwing pointcut-ref="setterMethods" method="logFailure"/>
  </aop:aspect>
</aop:config>

<bean id="propertyChangeTracker" class="example.PropertyChangeTracker" />
```

Named Pointcut Annotation



- Also allow you to reuse and combine pointcuts

```
@Aspect  
public class PropertyChangeTracker {  
    private Logger logger = Logger.getLogger(getClass());  
  
    @Before("serviceMethod() || repositoryMethod()")  
    public void monitor() {  
        logger.info("A business method has been accessed...");  
    }  
  
    @Pointcut("execution(* rewards.service..*Service.*(..))")  
    public void serviceMethod() {}  
  
    @Pointcut("execution(* rewards.repository..*Repository.*(..))")  
    public void repositoryMethod() {}  
}
```

Named Pointcuts



- Expressions can be externalized

```
public class SystemArchitecture {  
    @Pointcut("execution(* rewards.service..*Service.*(..))")  
    public void serviceMethods() {}  
}
```

```
@Aspect  
public class ServiceMethodInvocationMonitor {  
    private Logger logger = Logger.getLogger(getClass());  
  
    @Before("com.acme.SystemArchitecture.serviceMethods()")  
    public void monitor() {  
        logger.info("A service method has been accessed...");  
    }  
}
```

Fully-qualified pointcut name

Named pointcuts - Summary



- Can break one complicated expression into several sub-expressions
- Allow pointcut expression reusability
- Best practice: consider externalizing expressions into one dedicated class
 - When working with many pointcuts
 - When writing complicated expressions

Topics in this session



- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- **Advanced topics**
 - Named Pointcuts
 - **Context selecting pointcuts**
 - Working with annotations
 - Limitations of Spring AOP

Context Selecting Pointcuts



- Pointcuts may also select useful join point context
 - The currently executing object (proxy)
 - The target object
 - Method arguments
 - Annotations associated with the method, target, or arguments
- Allows for simple POJO advice methods
 - Alternative to working with a JoinPoint object directly

Context Selecting Example



- Consider this basic requirement

Log a message every time Server is about to start

```
public interface Server {  
    public void start(Map input);  
    public void stop();  
}
```

In the advice, how do we access Server? Map?

Without context selection



- All needed info must be obtained from the *JoinPoint* object
 - No type-safety guarantees
 - Write advice *defensively*

```
@Before("execution(void example.Server.start(java.util.Map))")  
public void logServerStartup(JoinPoint jp) {  
    Server server = (Server) jp.getTarget();  
    Object[] args= jp.getArgs();  
    Map map = (Map) args[0];  
    ...  
}
```

With context selection



- Best practice: use context selection
 - Method attributes are bound automatically
 - Types must match or advice skipped

```
@Before("execution(void example.Server.start(java.util.Map))  
    && target(server) && args(input)")  
public void logServerStartup(Server server, Map input) {  
    ...  
}
```

- target(server) selects the target of the execution (your object)
- this(server) would have selected the proxy

Context Selection - Named Pointcut



```
@Before("serverStartMethod(server, input)")
public void logServerStartup(Server server, Map input) {
    ...
}

@Pointcut("execution(void example.Server.start(java.util.Map))
          && target(server) && args(input)")
public void serverStartMethod (Server server, Map input) {}
```

The code snippet illustrates the use of named pointcuts. Two annotations are shown: `@Before` and `@Pointcut`. The `@Before` annotation has a pointcut expression "serverStartMethod(server, input)". The `@Pointcut` annotation has a pointcut expression "execution(void example.Server.start(java.util.Map)) && target(server) && args(input)". Below the code, two callout boxes explain the meaning of the arguments: "'target' binds the server starting up" and "'args' binds the argument value". Arrows from these boxes point to the corresponding parts in the pointcut expressions.

Topics in this session



- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- **Advanced topics**
 - Named Pointcuts
 - Context selecting pointcuts
 - **Working with annotations**
 - Limitations of Spring AOP

Pointcut expression examples using annotations



- Can match annotations everywhere
 - annotated methods, methods with annotated arguments, returning annotated objects, on annotated classes
- `execution(@org.*.transaction.annotation.Transactional * *(..))`
 - Any method marked with the `@Transactional` annotation
- `execution((@example.Sensitive *) *(..))`
 - Any method that returns a type marked as `@Sensitive`

```
@Sensitive  
public class MedicalRecord { ... }  
  
public class MedicalService {  
    public MedicalRecord lookup(...) { ... }  
}
```

- Use of the *annotation()* designator

```
@Around("execution(* *(..)) && @annotation(txn)")  
public Object execute(ProceedingJoinPoint jp, Transactional txn) {  
    TransactionStatus tx;  
  
    try {  
        TransactionDefinition defintion = new DefaultTransactionDefinition();  
        definition.setTimeout(txn.timeout());  
        definition.setReadOnly(txn.readOnly());  
  
        ...  
        tx = txnMgr.getTransaction(defintion);  
        return jp.proceed();  
    }  
    ... // commit or rollback  
}
```

No need for `@Transactional` in *execution* expression – the `@annotation` matches it instead

AOP and annotations

– Named pointcuts



- Same example using a named-pointcut

```
@Around("transactionalMethod(txn)")  
public Object execute(ProceedingJoinPoint jp, Transactional txn) {  
    ...  
}  
  
@Pointcut("execution(* *(..)) && @annotation(txn)")  
public void transactionalMethod(Transactional txn) {}
```

Topics in this session



- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- **Advanced topics**
 - Named Pointcuts
 - Context selecting pointcuts
 - Working with annotations
 - **Limitations of Spring AOP**

Limitations of Spring AOP



- Can only advise *public* Join Points
- Can only apply aspects to *Spring Beans*
- Limitations of weaving with proxies
 - Spring adds behavior using dynamic proxies *if* a Join Point is declared on an interface
 - If a Join Point is in a class *without* an interface, Spring reverts to using *CGLIB* for weaving
 - CGLib proxies cannot be applied to final classes or methods
 - When using dynamic proxies, suppose method a()
calls method b() on the same class/interface
 - advice will never be executed for method b()

Summary



- Aspect Oriented Programming (AOP) modularizes cross-cutting concerns
- An aspect is a module containing cross-cutting behavior
 - Behavior is implemented as “advice”
 - Pointcuts select where advice applies
 - Five advice types: Before, AfterThrowing, AfterReturning, After and Around
- Aspects can be defined using Java with annotations and/or in XML configuration



Introduction to Data Management with Spring

Spring's role in supporting data access
within an enterprise application

Topics in this Session



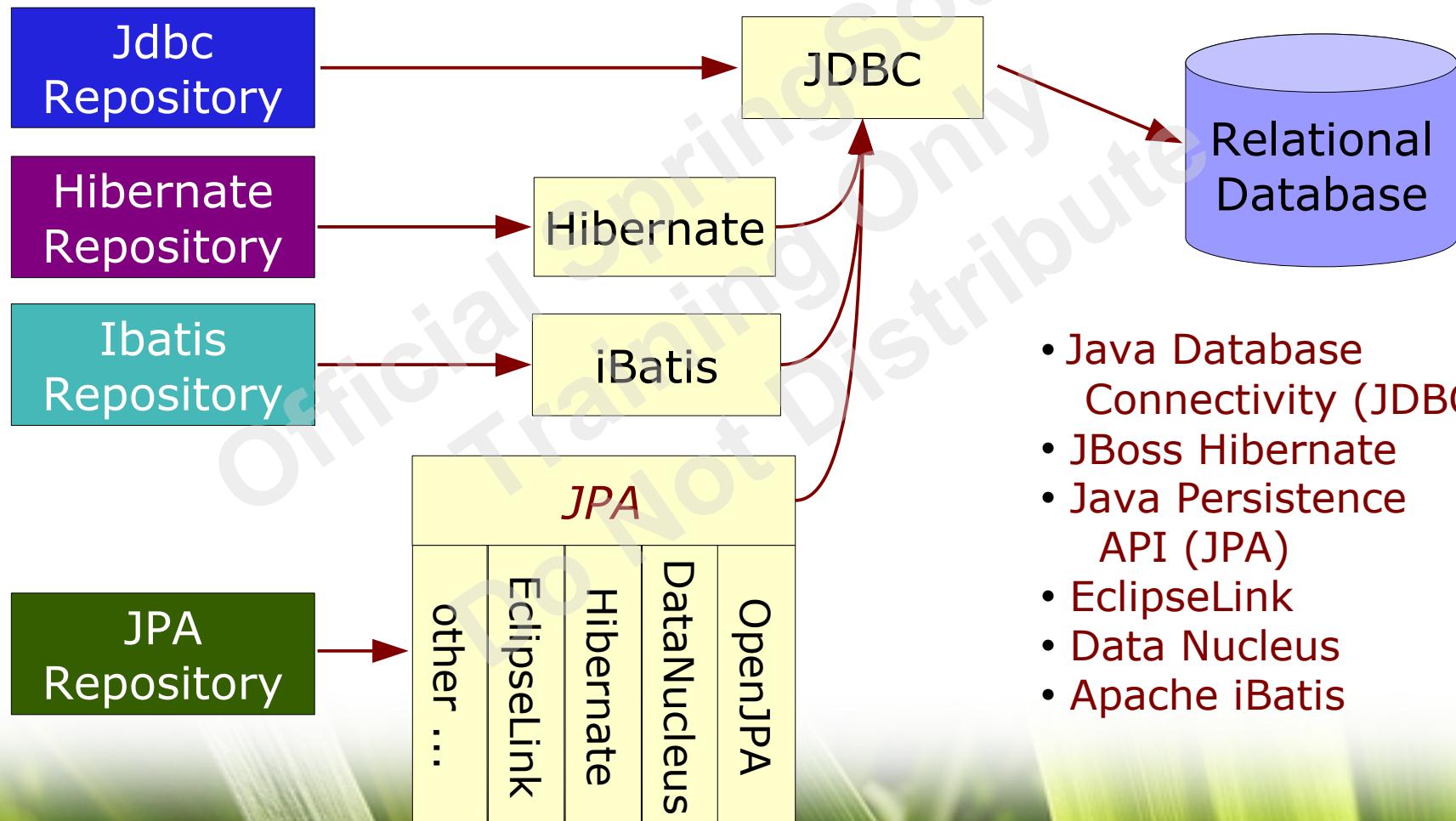
- **The Role of Spring in Enterprise Data Access**
- The `DataAccessException` Hierarchy
- The `jdbc` Namespace
- Implementing Caching
- NoSQL databases

Official Spring Training Only
Do Not Distribute

Spring Resource Management Works Everywhere



Works consistently with leading data access technologies



Key Resource Management Features

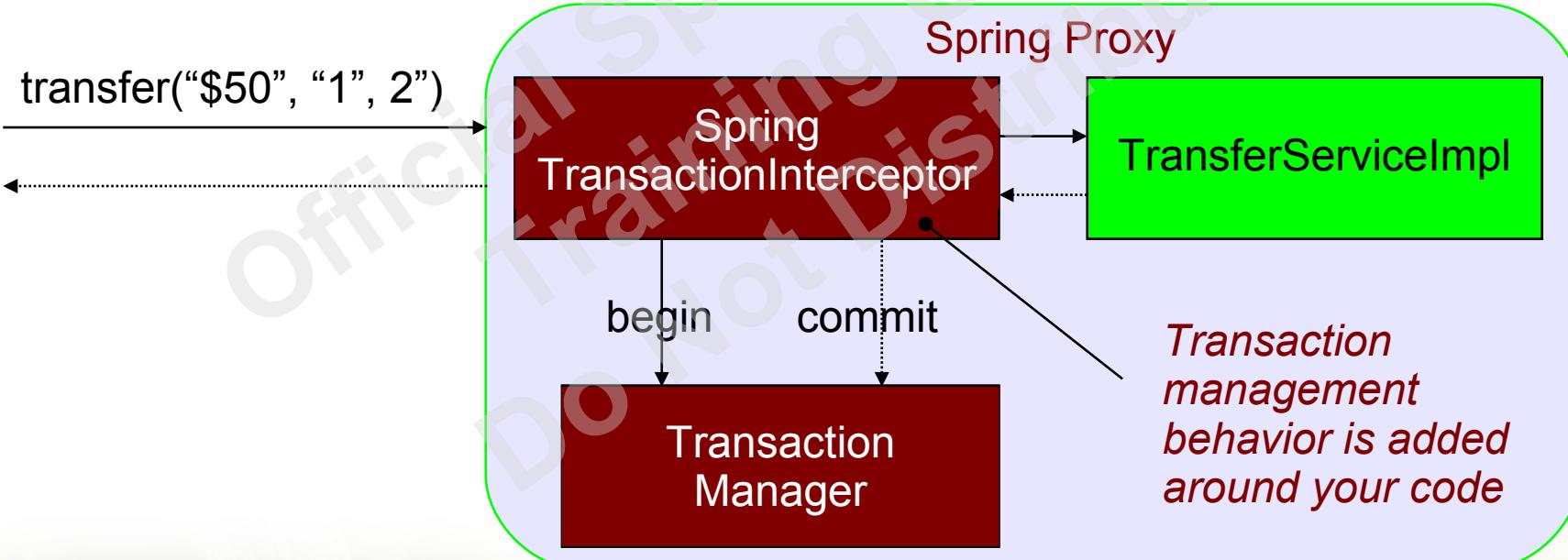


- Limited resources need to be managed
 - Doing it manually risks resource leakage
- Spring does it all for you!
 - Declarative transaction management
 - Transactional boundaries declared via configuration
 - Enforced by a Spring transaction manager
 - Automatic connection management
 - Connections acquired/released automatically
 - No possibility of resource leak
 - Intelligent exception handling
 - Root cause failures always reported
 - Resources always released properly

Declarative Transaction Management



```
public class TransferServiceImpl implements TransferService {  
    @Transactional // marks method as needing a txn  
    public void transfer(...) { // your application logic }  
}
```



The Resource Management Problem



- To access a data source an application must
 - Establish a connection
- To start its work the application must
 - Begin a transaction
- When done with its work the application must
 - Commit or rollback the transaction
 - Close the connection

Template Design Pattern



- Widely used and useful pattern
 - http://en.wikipedia.org/wiki/Template_method_pattern
- Define the outline or skeleton of an algorithm
 - Leave the details to specific implementations later
 - Hides away large amounts of *boilerplate* code
- Spring provides many template classes
 - JdbcTemplate
 - JmsTemplate, RestTemplate, WebServiceTemplate ...
 - Most hide low-level resource management

Where are my Transactions?



- Every thread needs its own transaction
 - Typically: a web-driven request
- Spring transaction management
 - Transaction manager handles transaction
 - Puts it into thread-local storage
 - Data-access code, like JdbcTemplate, finds it automatically
 - Or you can get it yourself:

```
DataSourceUtils.getConnection(dataSource)
```
 - Hibernate sessions, JTA (Java EE) work similarly

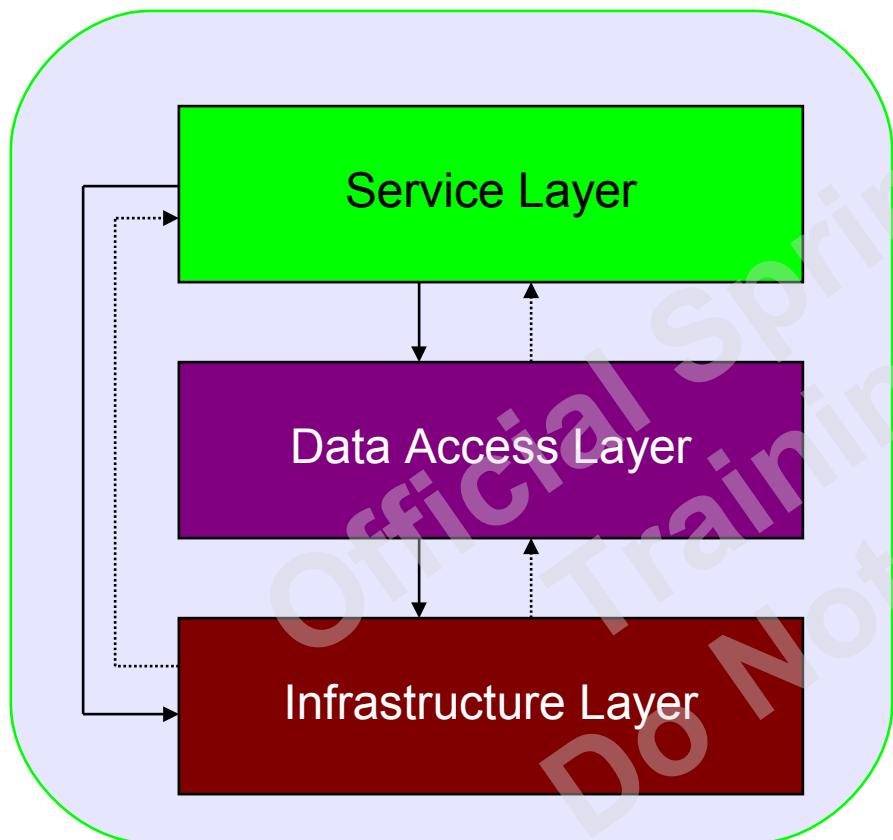


Data Access in a Layered Architecture



- Spring enables layered application architecture
- Most enterprise applications consist of three logical layers
 - Service layer (or application layer)
 - Exposes high-level application functions
 - Use-cases and business logic defined here
 - Data access layer
 - Defines the interface to the application's data repository (such as a relational database)
 - Infrastructure layer
 - Exposes low-level services needed by the other layers

Layered Application Architecture



- Classic example of a *Separation of Concerns*
 - Each layer is abstracted from the others
 - Spring configures it how you want

Topics in this Session



- The Role of Spring in Enterprise Data Access
- **The `DataAccessException` Hierarchy**
- The jdbc Namespace
- Implementing Caching
- NoSQL databases

Exception Handling



- Checked Exceptions
 - Force developers to handle errors
 - But if you can't handle it, must declare it
 - **Bad:** intermediate methods must declare exception(s) from *all* methods below
 - A form of tight-coupling
- Unchecked Exceptions
 - Can be thrown up the call hierarchy to the best place to handle it
 - **Good:** Methods in between don't know about it
 - Better in an Enterprise Application
 - Spring throws Runtime (unchecked) Exceptions

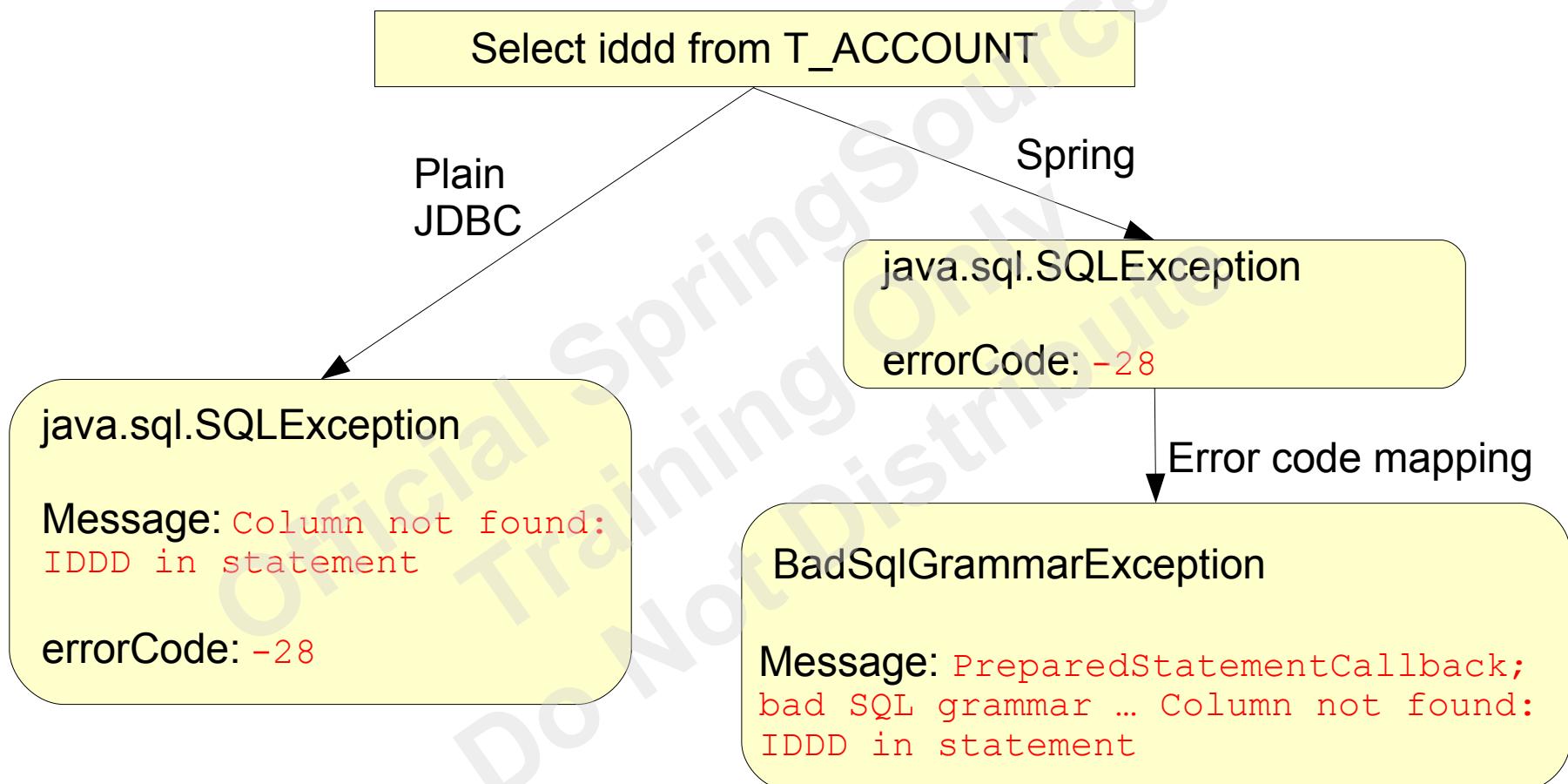


Data Access Exceptions



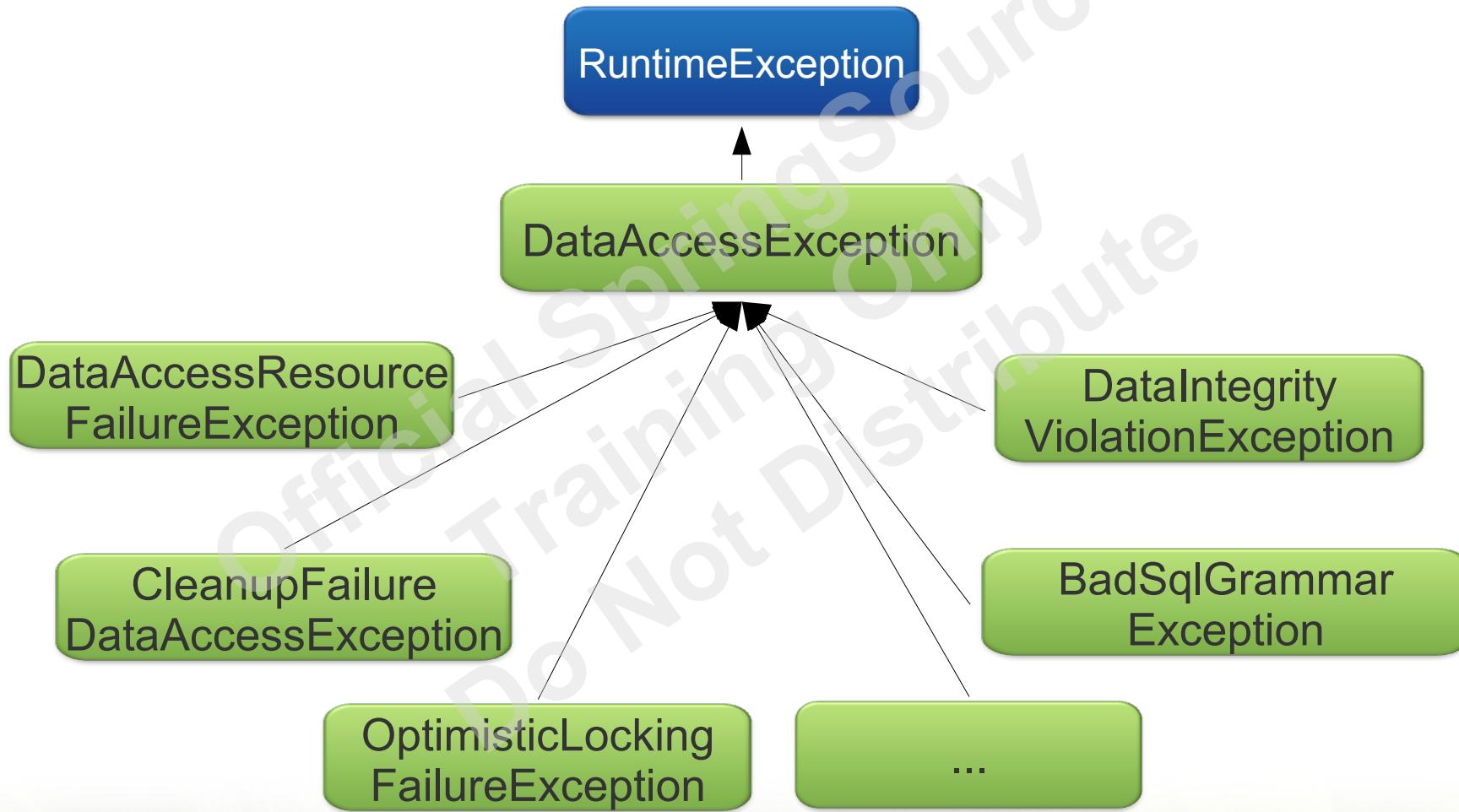
- SQLException
 - Too general – one exception for every database error
 - Calling class 'knows' you are using JDBC
 - Tight coupling
- Would like a hierarchy of exceptions
 - Consistent among all Data Access technologies
 - Needs to be unchecked
 - Not just one exception for everything
- Spring provides **DataAccessException** hierarchy
 - Hides whether you are using JPA, Hibernate, JDBC ...

Example: bad SQL grammar



For more details on error codes: see [spring-jdbc.jar/
org/springframework/jdbc/support/sql-error-codes.xml](http://spring-jdbc.jar/org/springframework/jdbc/support/sql-error-codes.xml)

Spring Data Access Exceptions



Topics in this Session



- The Role of Spring in Enterprise Data Access
- The `DataAccessException` Hierarchy
- **The `jdbc` Namespace**
- Implementing Caching
- NoSQL databases

Official Spring Training Only
Do Not Distribute

JDBC Namespace

- Introduced with Spring 3.0
- Especially useful for testing
- Supports H2, HSQL and Derby

```
<bean class="example.order.JdbcOrderRepository" >
  <property name="dataSource" ref="dataSource" />
</bean>

<jdbc:embedded-database id="dataSource" type="H2">
  <jdbc:script location="classpath:schema.sql" />
  <jdbc:script location="classpath:test-data.sql" />
</jdbc:embedded-database>
```

In memory database
(created at startup)

- Allows populating other DataSources, too

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

<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:schema.sql" />
    <jdbc:script location="classpath:test-data.sql" />
</jdbc:initialize-database>
```

Initializes an external database

Topics in this Session



- The Role of Spring in Enterprise Data Access
- The `DataAccessException` Hierarchy
- The `jdbc` Namespace
- **Implementing Caching**
- NoSQL databases

Official Spring Training Only
Do Not Distribute

About Caching

- What is a cache?
 - In this context: a key-value store = Map
- What are we caching?
 - Any method that always returns the same result for the same argument(s)
 - This method could do anything
 - Calculate data on the fly
 - Execute a database query
 - Request data via RMI, JMS, a web-service ...
 - A unique key must be generated from the arguments
 - That's the cache key

Caching Support

Spring 3.1

- Available since Spring 3.1
 - Transparently applies caching to Spring beans (AOP)
 - Define one or more caches in Spring configuration
 - Mark methods cacheable
 - Indicate caching key(s)
 - Name of cache to use (multiple caches supported)
 - Use annotations or XML

Spring Application

```
@Cacheable  
public Country[] loadAllCountries() { ... }
```

In-memory
cache

Caching with @Cacheable



- `@Cacheable` marks a method for caching
 - its result is stored in a cache
 - subsequent invocations (with the *same arguments*)
 - fetch data from cache using key, method not executed
- `@Cacheable` attributes
 - value: name of cache to use
 - key: the key for each cached data-item
 - Uses SpEL and argument(s) of method

```
@Cacheable(value="books", key="#refId.toUpperCase()")  
public Book findBook(String refId) {...}
```

Caching via Annotations



Use 'books' cache

```
@Cacheable(value="books", key="#title" condition="#title.length < 32")  
public Book findBook(String title, boolean checkWarehouse);
```

```
@Cacheable(value="books", key="#author.name")  
public Book findBook2(Author author, boolean checkWarehouse);
```

use object
property

```
@Cacheable(value="books", key="T(example.KeyGen).hash(#author)")  
public Book findBook3(Author author, boolean checkWarehouse);
```

custom key
generator

```
@CacheEvict(value="books")  
public Book loadBooks();
```

clear cache *before* method invoked

```
<cache:annotation-driven />
```

```
<bean id="bookService" class="example.BookService">
```

Pure XML Cache Setup



- Or use XML instead
 - For example with third-party class

```
<bean id="bookService" class="example.BookService">

<aop:config>
    <aop:advisor advice-ref="bookCache"
                  pointcut="execution(* *..BookService.*(..))"/>
</aop:config>

<cache:advice id="bookCache" cache-manager="cacheManager">
    <cache:caching cache="books">
        <cache:cacheable method="findBook" key="#refId"/>
        <cache:cache-evict method="loadBooks" all-entries="true" key="#refId"/>
    </cache:caching>
</cache:advice>
```

XML Cache Setup – no @Cachable

Setup 'JDK' Cache Manager



- Must specify a cache-manager
 - Some provided, or write your own
 - See `org.springframework.cache` package.

```
<bean id="cacheManager" class="o.s.cache.support.SimpleCacheManager">
  <property name="caches">
    <set>
      <bean class="o.s.cache.concurrent.ConcurrentMapCacheFactoryBean"
            p:name="authors" />
      <bean class="o.s.cache.concurrent.ConcurrentMapCacheFactoryBean"
            p:name="books" />
    </set>
  </property>
</bean>
```

Concurrent Map Cache

JDK

Third-Party Cache Managers



- Two are supported already

```
<bean id="cacheManager" class="...EhCacheCacheManager"  
      p:cache-manager-ref="ehcache" />
```

EHcache

```
<bean id="ehcache"  
      class="o.s.cache.ehcache.EhCacheCacheManagerFactoryBean"  
      p:config-location="ehcache.xml" />
```

 EHCache

```
<gfe:cache-manager p:cache-ref="gemfire-cache"/>
```

VMware Gemfire
Cache

```
<gfe:cache id="gemfire-cache"/>
```

```
<gfe:replicated-region id="authors" p:cache-ref="gemfire-cache"/>  
<gfe:partitioned-region id="books" p:cache-ref="gemfire-cache"/>
```

 GEMFIRE®

Spring Gemfire Project



- GemFire configuration in Spring config files
 - Also enables configuration injection for environments
- Features
 - Exception translation
 - GemfireTemplate
 - Transaction management
(GemfireTransactionManager)
 - Injection of transient dependencies during deserialization
 - *Gemfire Cache Manager class*

Topics in this Session



- The Role of Spring in Enterprise Data Access
- The `DataAccessException` Hierarchy
- The `jdbc` Namespace
- Implementing Caching
- **NoSQL databases**

Not Only SQL!

- NoSQL
 - Relational databases only store some data
 - LDAP, data-warehouses, files
 - Most documents and spreadsheets aren't in *any* database
- Other database products exist
 - Have strengths where RDB are weak
 - Non-tabular data
 - Hierarchical data: parts inventory, org chart
 - Network structures: telephone cables, roads, molecules
 - Documents: XML, spreadsheets, contracts, ...
 - Geographical data: maps, GPS navigation
 - Many more ...

So Many Databases ...

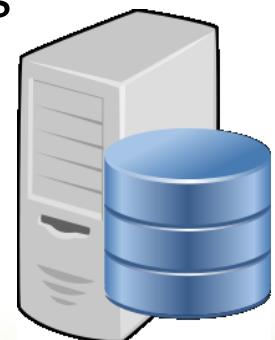


- Many options – each has a particular strength
 - Document databases
 - MongoDB, *CouchDB coming*
 - Distributed key-value Stores (smart caches)
 - Redis, Riak
 - Network (graph) database
 - Neo4j
 - Big Data
 - Apache Hadoop (VMware Serengeti)
 - Data Grid
 - Gemfire
 - Column Stores *coming*: *HBase, Cassandra*



Summary

- Data Access with Spring
 - Enables layered architecture principles
 - Higher layers should not know about data management below
 - Isolate via Data Access Exceptions
 - Hierarchy makes them easier to handle
 - Provides consistent transaction management
 - Supports most leading data-access technologies
 - Relational and non-relational (NoSQL)
 - A key component of the core Spring libraries
 - Automatic caching facility





Introduction to Spring JDBC

Simplifying JDBC-based data access with Spring JDBC

Topics in this Session

- **Problems with traditional JDBC**
 - Results in redundant, error prone code
 - Leads to poor exception handling
- Spring's JdbcTemplate
 - Configuration
 - Query execution
 - Working with result sets
 - Exception handling

Redundant, Error Prone Code



```
public List findByLastName(String lastName) {  
    List personList = new ArrayList();  
    Connection conn = null;  
    String sql = "select first_name, age from PERSON where last_name=?";  
    try {  
        DataSource dataSource = DataSourceUtils.getDataSource();  
        conn = dataSource.getConnection();  
        PreparedStatement ps = conn.prepareStatement(sql);  
        ps.setString(1, lastName);  
        ResultSet rs = ps.executeQuery();  
        while (rs.next()) {  
            String firstName = rs.getString("first_name");  
            int age = rs.getInt("age");  
            personList.add(new Person(firstName, lastName, age));  
        }  
    } catch (SQLException e) { /* ??? */ }  
    finally {  
        try {  
            conn.close();  
        } catch (SQLException e) { /* ??? */ }  
    }  
    return personList;  
}
```

Redundant, Error Prone Code



```
public List findByLastName(String lastName) {  
    List personList = new ArrayList();  
    Connection conn = null;  
    String sql = "select first_name, age from PERSON where last_name=?";  
    try {  
        DataSource dataSource = DataSourceUtils.getDataSource();  
        conn = dataSource.getConnection();  
        PreparedStatement ps = conn.prepareStatement(sql);  
        ps.setString(1, lastName);  
        ResultSet rs = ps.executeQuery();  
        while (rs.next()) {  
            String firstName = rs.getString("first_name");  
            int age = rs.getInt("age");  
            personList.add(new Person(firstName, lastName, age));  
        }  
    } catch (SQLException e) { /* ??? */}  
    finally {  
        try {  
            conn.close();  
        } catch (SQLException e) { /* ??? */}  
    }  
    return personList;  
}
```

The bold matters - the
rest is boilerplate

Poor Exception Handling



```
public List findByLastName(String lastName) {  
    List personList = new ArrayList();  
    Connection conn = null;  
    String sql = "select first_name, age from PERSON where last_name=?";  
    try {  
        DataSource dataSource = DataSourceUtils.getDataSource();  
        conn = dataSource.getConnection();  
        PreparedStatement ps = conn.prepareStatement(sql);  
        ps.setString(1, lastName);  
        ResultSet rs = ps.executeQuery();  
        while (rs.next()) {  
            String firstName = rs.getString("first_name");  
            int age = rs.getInt("age");  
            personList.add(new Person(firstName, lastName, age));  
        }  
    } catch (SQLException e) { /* ??? */ }  
    finally {  
        try {  
            conn.close();  
        } catch (SQLException e) { /* ??? */ }  
    }  
    return personList;  
}
```

What can
you do?

Topics in this session

- Problems with traditional JDBC
 - Results in redundant, error prone code
 - Leads to poor exception handling
- **Spring's JdbcTemplate**
 - Configuration
 - Query execution
 - Working with result sets
 - Exception handling

Spring's JdbcTemplate



- Greatly simplifies use of the JDBC API
 - Eliminates repetitive boilerplate code
 - Alleviates common causes of bugs
 - Handles SQLExceptions properly
- Without sacrificing power
 - Provides full access to the standard JDBC constructs

JdbcTemplate in a Nutshell



```
int count = jdbcTemplate.queryForInt(  
    "SELECT COUNT(*) FROM CUSTOMER");
```

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling any exceptions
- Release of the connection

All handled
by Spring

JdbcTemplate Approach Overview



```
List results = jdbcTemplate.query(sql,  
    new RowMapper<Customer>() {  
        public Customer mapRow(ResultSet rs, int row) throws SQLException {  
            // map the current row to a Customer object  
        }  
   });
```

```
class JdbcTemplate {  
    public List<Customer> query(String sql, RowMapper rowMapper) {  
        try {  
            // acquire connection  
            // prepare statement  
            // execute statement  
            // for each row in the result set  
            results.add(rowMapper.mapRow(rs, rowNum));  
        } catch (SQLException e) {  
            // convert to root cause exception  
        } finally {  
            // release connection  
        }  
    }  
}
```

Creating a JdbcTemplate



- Requires a DataSource

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```

- Create a template once and re-use it
 - Do not create one for each use
 - Thread safe after construction

When to use JdbcTemplate



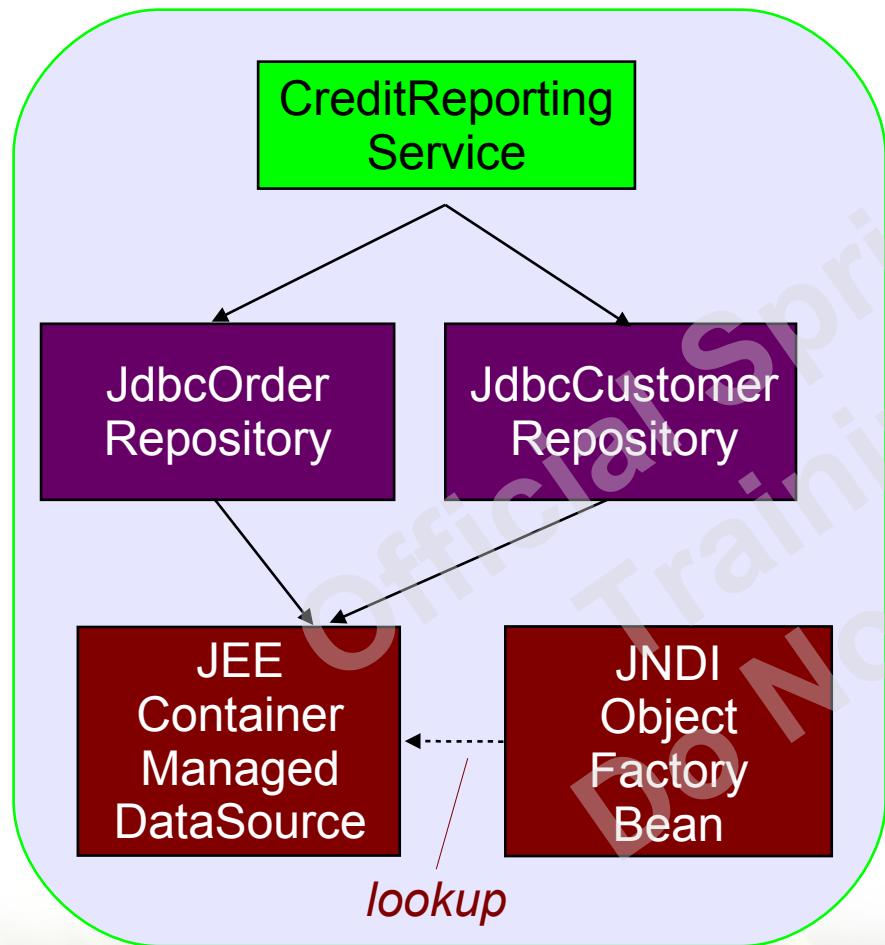
- Useful standalone
 - Anytime JDBC is needed
 - In utility or test code
 - To clean up messy legacy code
- Useful for implementing a **repository** in a layered application
 - Also known as a data access object (DAO)

Implementing a JDBC-based Repository



```
public class JdbcCustomerRepository implements CustomerRepository {  
  
    private JdbcTemplate jdbcTemplate;  
  
    public JdbcCustomerRepository(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    public int getCustomerCount() {  
        String sql = "select count(*) from customer";  
        return jdbcTemplate.queryForInt(sql);  
    }  
}
```

Accessing JNDI Resources



- JDBC Options
 - Use local datasource
 - Use container-managed datasource (via JNDI)
- JEE Namespace
 - Provides access to JNDI resources
 - Looks like Spring bean
 - Hides factory-bean

```
<jee:jndi-lookup id="dataSource"  
                jndi-name="jdbc/credit" />
```

Integrating a Repository into an Application



```
<bean id="creditReportingService" class="example.CreditReportingService">
    <property name="orderRepository" ref="orderRepository" />
    <property name="customerRepository" ref="customerRepository" />
</bean>
```

Inject the repository into application services

```
<bean id="orderRepository" class="example.order.JdbcOrderRepository">
    <constructor-arg ref="dataSource"/>
</bean>
```

```
<bean id="customerRepository"
      class="example.customer.JdbcCustomerRepository">
    <constructor-arg ref="dataSource" />
</bean>
```

Configure the repository's DataSource

```
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/credit" />
```

Access container-managed data-source

Querying with JdbcTemplate



- JdbcTemplate can query for
 - Simple types (int, long, String)
 - Generic Maps
 - Domain Objects

Querying for Simple Java Types (1)



- Query with no bind variables
 - `queryForInt`, `queryForLong`

```
public long getPersonCount() {  
    String sql = "select count(*) from PERSON";  
    return jdbcTemplate.queryForLong(sql);  
}
```

- `queryForObject`

```
public Date getOldestPerson() {  
    String sql = "select max(dob) from PERSON";  
    return (Date) jdbcTemplate.queryForObject(sql, Date.class);  
}
```

Querying With JdbcTemplate



- Query with a bind variable
 - Note the use of a variable argument list

```
private JdbcTemplate jdbcTemplate;

public int getCountOfPersonsOlderThan(int age) {
    return jdbcTemplate.queryForInt(
        "select count(*) from PERSON where age > ?", age);
}
```

Generic Queries



- JdbcTemplate can return each row of a ResultSet as a Map
- When expecting a single row
 - Use queryForMap(..)
- When expecting multiple rows
 - Use queryForList(..)
- Useful for reporting, testing, and ‘window-on-data’ use cases
 - The data fetched does not need mapping to a Java object
 - Be careful with very large data-sets

Querying for Generic Maps

(1)



- Query for a single row

```
public Map getPersonInfo(int id) {  
    String sql = "select * from PERSON where id=?";  
    return jdbcTemplate.queryForMap(sql, id);  
}
```

- returns:
Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }

A Map of [Column Name | Field Value] pairs

Querying for Generic Maps (2)



- Query for multiple rows

```
public List getAllPersonInfo() {  
    String sql = "select * from PERSON";  
    return jdbcTemplate.queryForList(sql);  
}
```

- returns:

```
List {  
    0 - Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }  
    1 - Map { ID=2, FIRST_NAME="Jane", LAST_NAME="Doe" }  
    2 - Map { ID=3, FIRST_NAME="Junior", LAST_NAME="Doe" }  
}
```

A List of Maps of [Column Name | Field Value] pairs

Domain Object Queries



- Often it is useful to map relational data into domain objects
 - e.g. a ResultSet to an Account
- Spring's JdbcTemplate supports this using a *callback* approach
- You may prefer to use ORM for this
 - Need to decide between JdbcTemplate queries and JPA (or similar) mappings
 - Some tables may be *too hard* to map with JPA

RowMapper

- Spring provides a RowMapper interface for mapping a single row of a ResultSet to an object
 - Can be used for both single and multiple row queries
 - Parameterized as of Spring 3.0

```
public interface RowMapper<T> {  
    T mapRow(ResultSet rs, int rowNum)  
        throws SQLException;  
}
```

Querying for Domain Objects (1)



- Query for single row with JdbcTemplate

```
public Person getPerson(int id) {  
    return jdbcTemplate.queryForObject(  
        "select first_name, last_name from PERSON where id=?",  
        new PersonMapper(), id);  
}
```

No need to cast

Maps rows to Person objects

Parameterizes return type

```
class PersonMapper implements RowMapper<Person> {  
    public Person mapRow(ResultSet rs, int i) throws SQLException {  
        return new Person(rs.getString("first_name"),  
                         rs.getString("last_name"));  
    }  
}
```

Querying for Domain Objects (2)



- Query for multiple rows

No need to cast

```
public List<Person> getAllPersons() {  
    return jdbcTemplate.query(  
        "select first_name, last_name from PERSON",  
        new PersonMapper());
```

Same row mapper can be used

```
class PersonMapper implements RowMapper<Person> {  
    public Person mapRow(ResultSet rs, int i) throws SQLException {  
        return new Person(rs.getString("first_name"),  
                         rs.getString("last_name"));  
    }  
}
```

RowCallbackHandler

- Spring provides a simpler RowCallbackHandler interface when there is no return object
 - Streaming rows to a file
 - Converting rows to XML
 - Filtering rows before adding to a Collection
 - but filtering in SQL is *much* more efficient
 - Faster than JPA equivalent for big queries
 - avoids result-set to object mapping

```
public interface RowCallbackHandler {  
    void processRow(ResultSet rs) throws SQLException;  
}
```

Using a RowCallbackHandler



```
public class JdbcOrderRepository {  
    public void generateReport(Writer out) {  
        // select all orders of year 2009 for a full report  
        jdbcTemplate.query("select * from order where year=?",  
                           new OrderReportWriter(out), 2009);  
    }  
}  
returns "void"
```

```
class OrderReportWriter implements RowCallbackHandler {  
    public void processRow(ResultSet rs) throws SQLException {  
        // parse current row from ResultSet and stream to output  
    }  
    /* stateful object: may add convenience methods like getResults(), getCount() etc. */  
}
```

ResultSetExtractor



- Spring provides a ResultSetExtractor interface for processing an entire ResultSet at once
 - You are responsible for iterating the ResultSet
 - e.g. for mapping entire ResultSet to a single object

```
public interface ResultSetExtractor<T> {  
    T extractData(ResultSet rs) throws SQLException,  
        DataAccessException;  
}
```

Using a ResultSetExtractor



```
public class JdbcOrderRepository {  
    public Order findByConfirmationNumber(String number) {  
        // execute an outer join between order and item tables  
        return jdbcTemplate.query(  
            "select...from order o, item i...conf_id = ?",
            new OrderExtractor(), number);  
    } } 
```

```
class OrderExtractor implements ResultSetExtractor<Order> {  
    public Order extractData(ResultSet rs) throws SQLException {  
        Order order = null;  
        while (rs.next()) {  
            if (order == null) {  
                account = new Account(rs.getString("ID"), rs.getLong("BALANCE"));  
            }  
            order.addItem(mapItem(rs));  
        }  
        return order;  
    } } 
```

Summary of Callback Interfaces



- RowMapper
 - Best choice when *each* row of a ResultSet maps to a domain object
- RowCallbackHandler
 - Best choice when no value should be returned from the callback method for *each* row
- ResultSetExtractor
 - Best choice when *multiple* rows of a ResultSet map to a *single* object

Inserts and Updates (1)



- Inserting a new row

```
public int insertPerson(Person person) {  
    return jdbcTemplate.update(  
        "insert into PERSON (first_name, last_name, age)" +  
        "values (?, ?, ?)",  
        person.getFirstName(),  
        person.getLastName(),  
        person.getAge());  
}
```

Inserts and Updates (2)



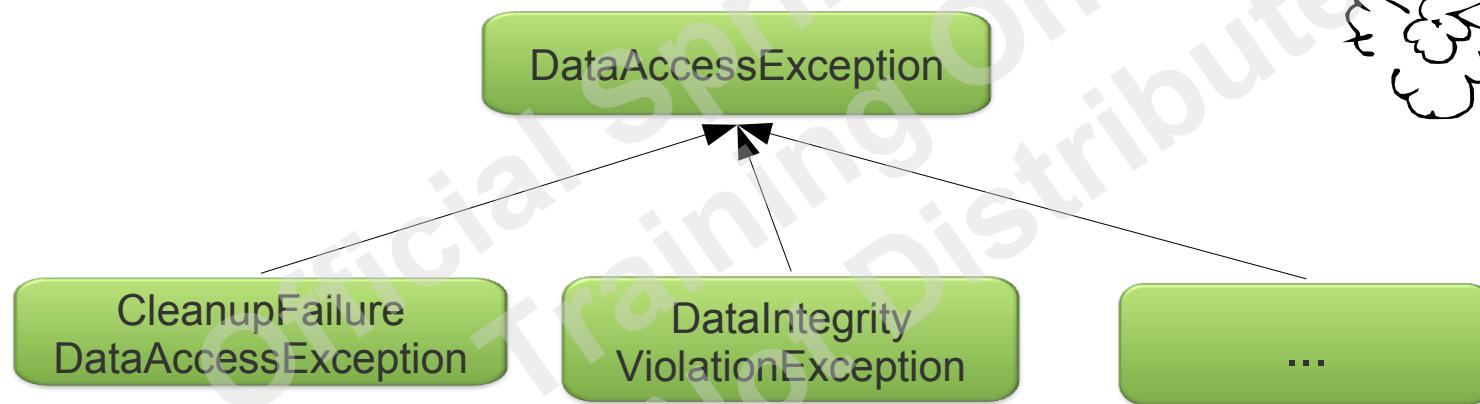
- Updating an existing row

```
public int updateAge(Person person) {  
    return jdbcTemplate.update(  
        "update PERSON set age=? where id=?",  
        person.getAge(),  
        person.getId());  
}
```

Exception Handling



- The JdbcTemplate transforms SQLExceptions into DataAccessExceptions



The *DataAccessException* hierarchy has already been discussed in the module called “Introduction to Data Access”. You can refer to it for more information on this topic.



LAB

Introduction to Spring JDBC



Transaction Management with Spring

An Overview of Spring's Consistent Approach to
Managing Transactions

Topics in this session



- **Why use Transactions?**
- Local Transaction Management
- Spring Transaction Management
- Annotations or XML
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

Why use Transactions?

To Enforce the ACID Principles:



- **Atomic**
 - Each unit of work is an all-or-nothing operation
- **Consistent**
 - Database integrity constraints are never violated
- **Isolated**
 - Isolating transactions from each other
- **Durable**
 - Committed changes are permanent

Transactions in the RewardNetwork



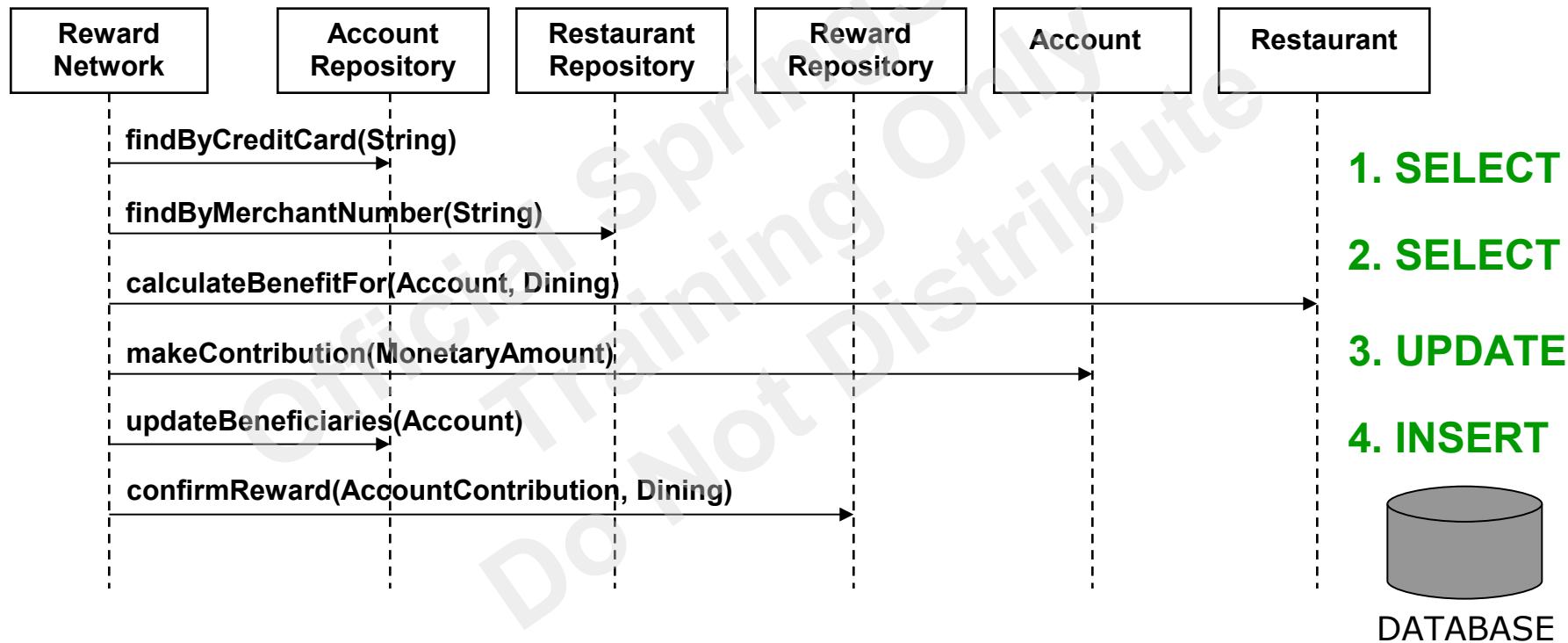
- The rewardAccountFor(Dining) method represents a unit-of-work that should be atomic

Official SpringSource
Training Only
Do Not Distribute

RewardNetwork Atomicity



- The `rewardAccountFor(Dining)` unit-of-work:

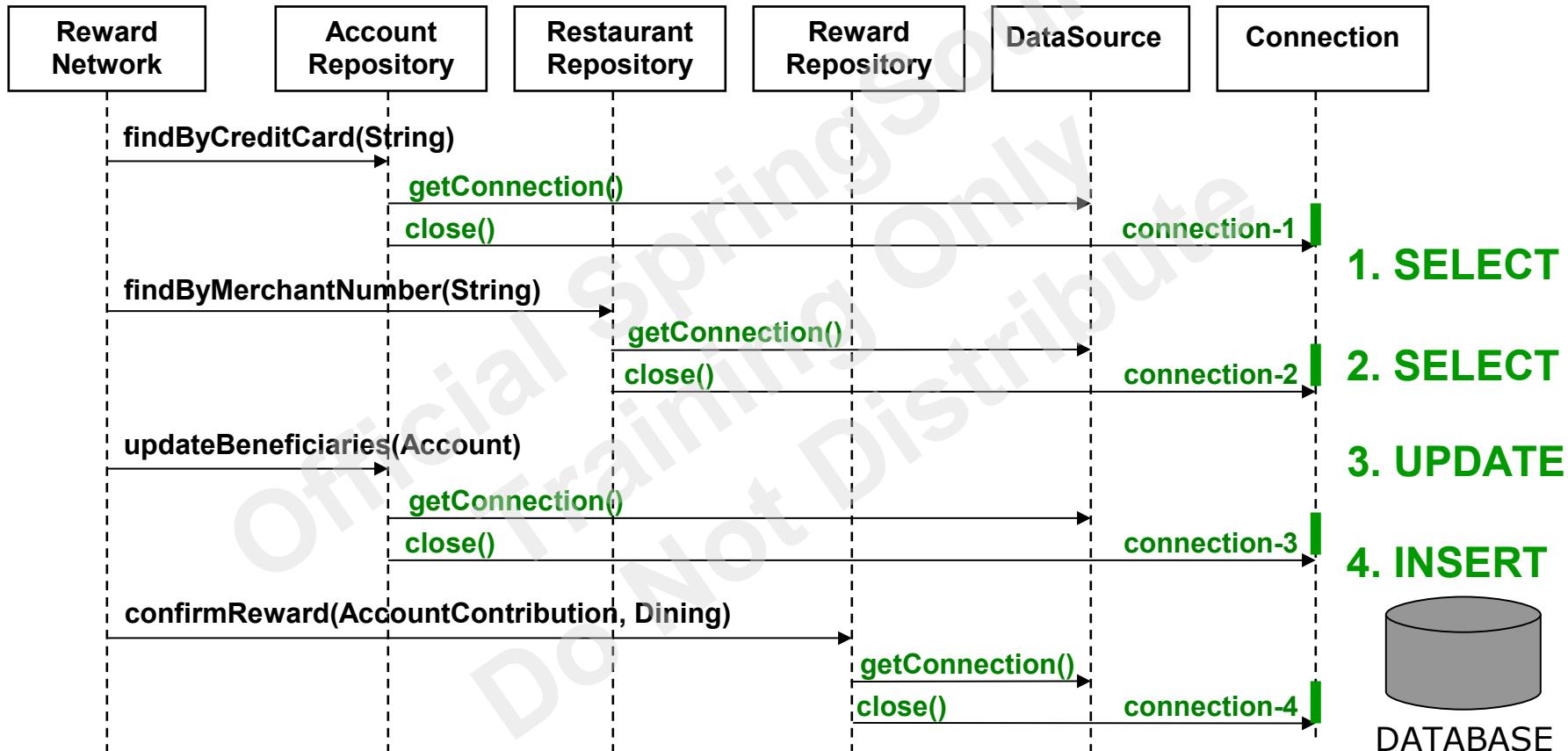


Naïve Approach: Connection per Data Access Operation



- This unit-of-work contains 4 data access operations
- Each acquires, uses, and releases a distinct Connection
- The unit-of-work is ***non-transactional***

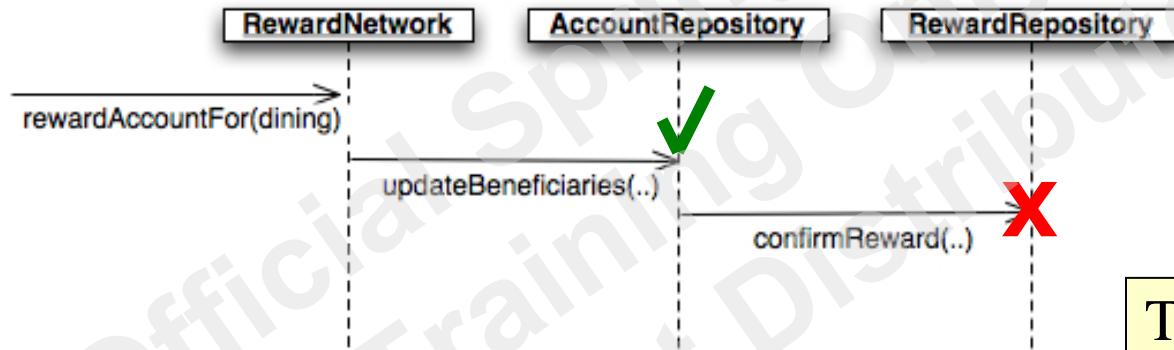
Running non-Transactionally



Partial Failures



- Suppose an Account is being rewarded



- If the beneficiaries are updated...
- But the reward confirmation fails...
- There will be no record of the reward!

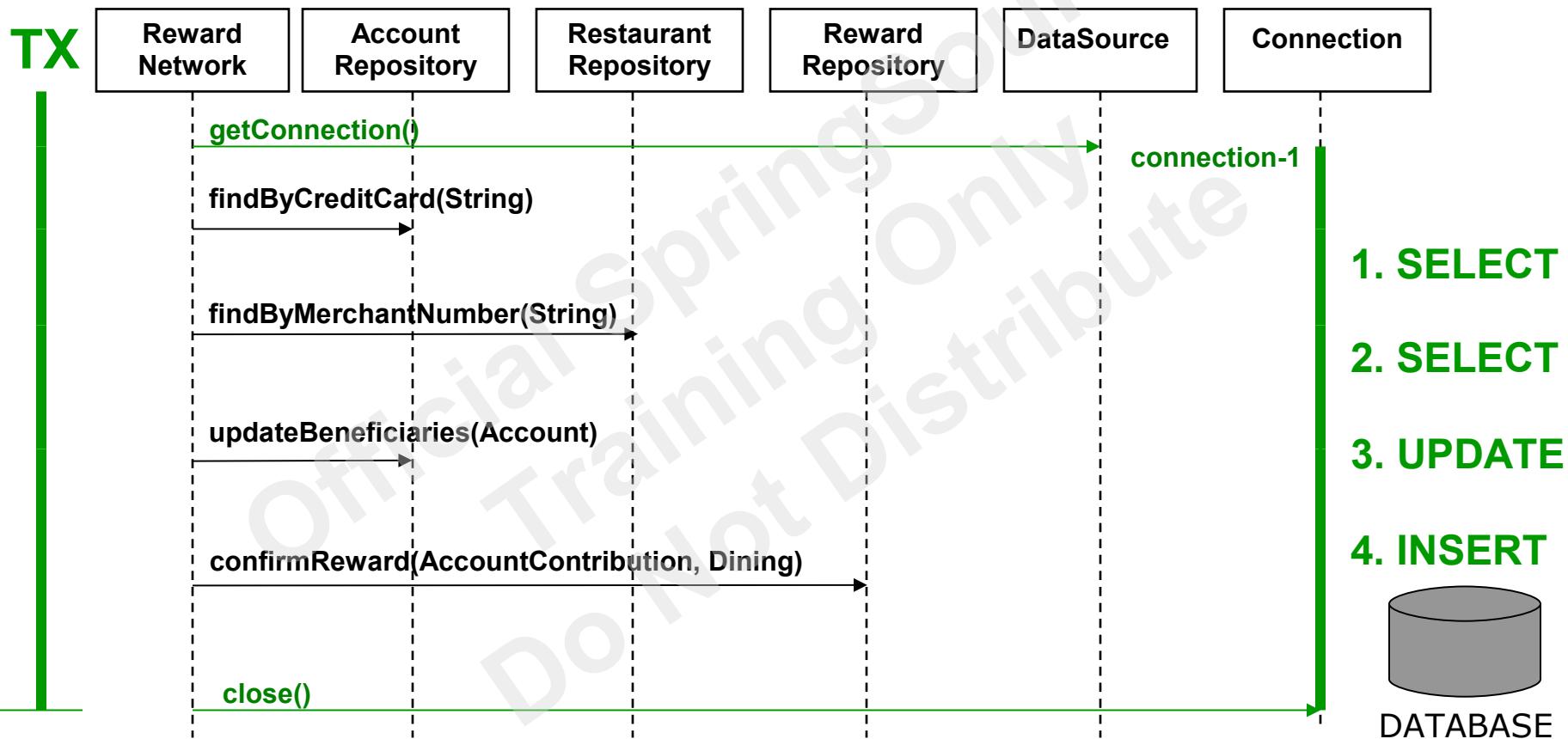
The unit-of-work
is not *atomic*

Correct Approach: Connection per Unit-of-Work



- More efficient
 - Same Connection reused for each operation
- Operations complete as an atomic unit
 - Either all succeed or all fail
- The unit-of-work can run in a ***transaction***

Running in a Transaction



Topics in this session



- Why use Transactions?
- **Local Transaction Management**
- Spring Transaction Management
- Annotations or XML
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

Local Transaction Management



- Transactions can be managed at the level of a local resource
 - Such as the database
- Requires programmatic management of transactional behavior on the Connection

Local Transaction Management Example



```
public void updateBeneficiaries(Account account) {  
    ...  
    try {  
        conn = dataSource.getConnection();  
        conn.setAutoCommit(false);  
        ps = conn.prepareStatement(sql);  
        for (Beneficiary b : account.getBeneficiaries()) {  
            ps.setBigDecimal(1, b.getSavings().asBigDecimal());  
            ps.setLong(2, account.getEntityId());  
            ps.setString(3, b.getName());  
            ps.executeUpdate();  
        }  
        conn.commit();  
    } catch (Exception e) {  
        conn.rollback();  
        throw new RuntimeException("Error updating!", e);  
    }  
}
```

Problems with Local Transactions



- Connection management code is error-prone
- Transaction demarcation belongs at the service layer
 - Multiple data access methods may be called within a transaction
 - Connection must be managed at a higher level

Topics in this session



- Why use Transactions?
- Local Transaction Management
- **Spring Transaction Management**
- Annotations or XML
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

Spring Transaction Management



- Declarative transactions are the recommended approach
 - Spring provides a flexible and powerful abstraction layer for transaction management
- There are only 2 steps
 - Define a **TransactionManager**
 - Declare the transactional methods
 - Using Annotations, XML, Programmatic
 - Can mix and match
- Optimization for local transactions
 - Database connection automatically bound to the current thread

PlatformTransactionManager



- Spring's **PlatformTransactionManager** is the base interface for the abstraction
- Several implementations are available
 - DataSourceTransactionManager
 - HibernateTransactionManager
 - JpaTransactionManager
 - JtaTransactionManager
 - WebLogicJtaTransactionManager
 - WebSphereUowTransactionManager
 - *and more*



Spring allows you to configure whether you use JTA or not. It does not have any impact on your Java classes

Deploying the Transaction Manager



- Pick the specific implementation

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

- or for JTA, also possible to use custom tag:

```
<tx:jta-transaction-manager/>
```

- Resolves to appropriate impl for environment
 - OC4JJtaTransactionManager
 - WebLogicJtaTransactionManager
 - WebSphereUowTransactionManager
 - JtaTransactionManager

@Transactional configuration using XML



- In the code:

```
public class RewardNetworkImpl implements RewardNetwork {  
    @Transactional  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // atomic unit-of-work  
    }  
}
```

- In the configuration:

```
<tx:annotation-driven/>  
<bean id="transactionManager"  
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
</bean>  
<jdbc:embedded-database id="dataSource"> ... </jdbc:embedded-database>
```

Defines a Bean Post-Processor
– proxies @Transactional beans

@Transactional configuration using Java configuration



Since Spring 3.1

- In the code:

```
public class RewardNetworkImpl implements RewardNetwork {  
    @Transactional  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // atomic unit-of-work  
    }  
}
```

- In the configuration:

```
@EnableTransactionManagement // Equivalent to <tx:annotation-driven/>  
public class AppConfig {  
    @Bean  
    public PlatformTransactionManager transactionManager(DataSource ds);  
    return new DataSourceTransactionManager(ds) {  
    }  
}
```

A dataSource must be defined elsewhere

@Transactional: what happens exactly?

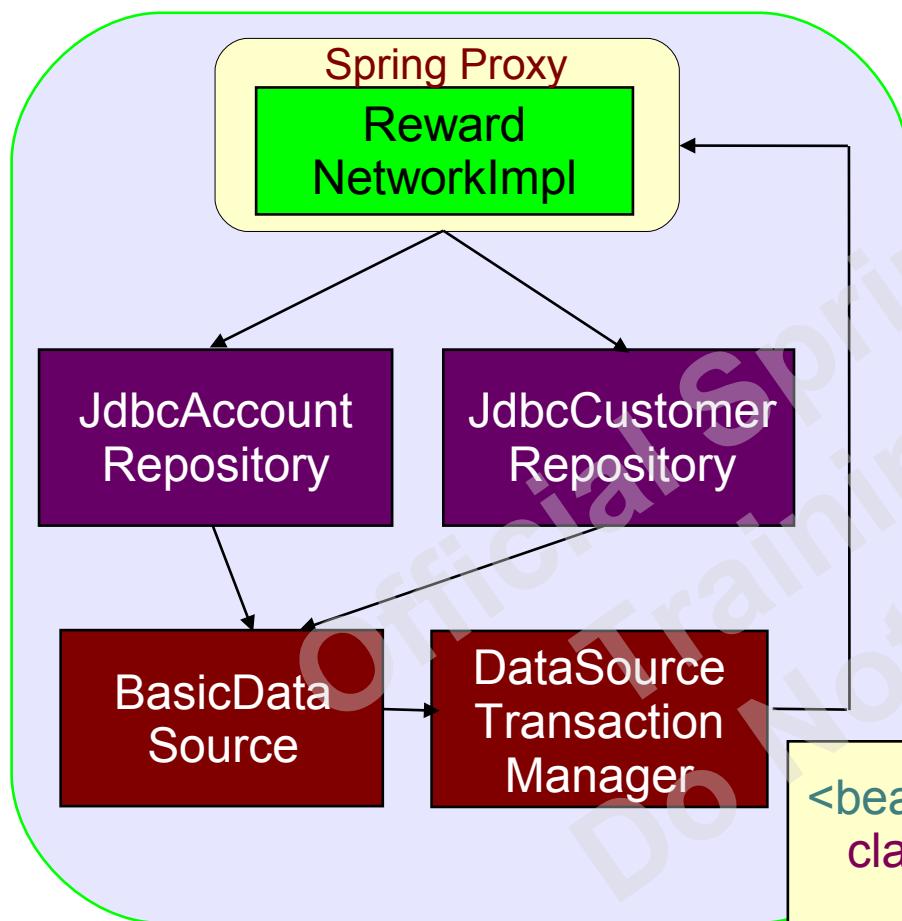


- Transactional object wrapped in a proxy
 - Uses an Around advice
- Proxy implements the following behavior
 - Transaction started before entering the method
 - Commit at the end of the method
 - Rollback if method throws a RuntimeException
 - Default behavior
 - Can be overridden (see later)
- All controlled by *configuration*

Spring Proxy

Reward
NetworkImpl

Local JDBC Configuration

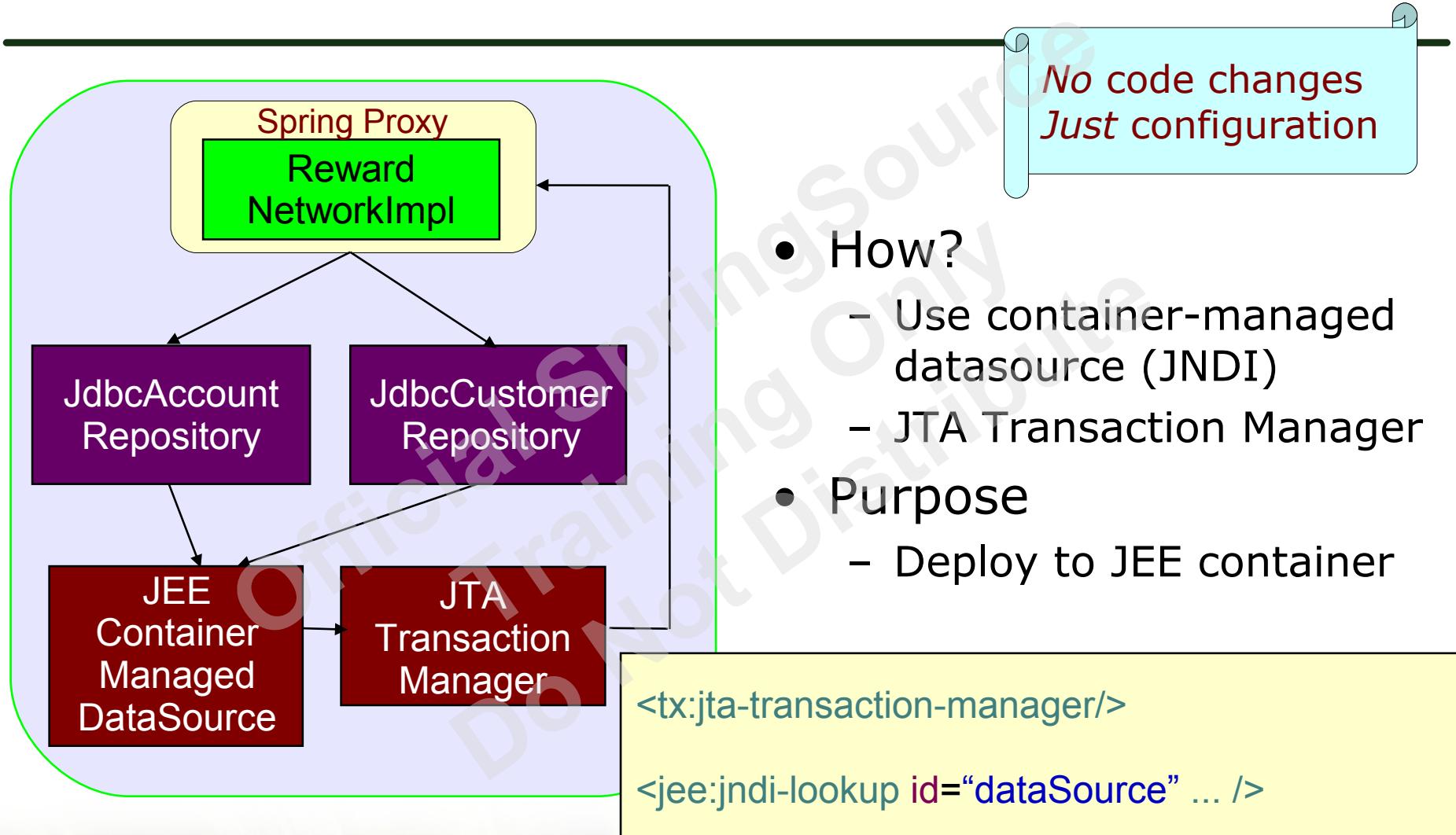


- How?
 - Define local data source
 - DataSource Transaction Manager
- Purpose
 - Integration testing
 - Deploy to Tomcat or other servlet container

```
<bean id="transactionManager"  
      class="...DataSourceTransactionManager"> ...
```

```
<jdbc:embedded-database id="dataSource"> ...
```

JDBC Java EE Configuration



- How?

- Use container-managed datasource (JNDI)
- JTA Transaction Manager

- Purpose

- Deploy to JEE container

Topics in this session



- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- **Annotations or XML**
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

- Applies to all methods declared by the interface(s)

@Transactional

```
public class RewardNetworkImpl implements RewardNetwork {  
  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // atomic unit-of-work  
    }  
  
    public RewardConfirmation updateConfirmation(RewardConfirmantion rc) {  
        // atomic unit-of-work  
    }  
}
```



@Transactional can also be declared at the interface/parent class level

@Transactional

– Class and method levels



- Combining class and method levels

```
@Transactional(timeout=60) ← default settings
public class RewardNetworkImpl implements RewardNetwork {

    public RewardConfirmation rewardAccountFor(Dining d) {
        // atomic unit-of-work
    }

    @Transactional(timeout=45) ← overriding attributes at
    public RewardConfirmation updateConfirmation(RewardConfirmation rc) {
        // atomic unit-of-work
    }
}
```

XML-based Spring Transactions



- Cannot always use @Transactional
 - Annotation-driven transactions require JDK 5
 - Someone else may have written the service (without annotations)
- Spring also provides an option for XML
 - An AOP pointcut declares what to advise
 - Spring's `tx` namespace enables a concise definition of transactional advice
 - Can add transactional behavior to any class used as a Spring Bean

Declarative Transactions: XML



```
<aop:config>
  <aop:pointcut id="rewardNetworkMethods"
    expression="execution(* rewards.service.*Service.*(..))"/>
  <aop:advisor pointcut-ref="rewardNetworkMethods" advice-ref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true" timeout="10"/>
    <tx:method name="find*" read-only="true" timeout="10"/>
    <tx:method name="*" timeout="30"/>
  </tx:attributes>
</tx:advice>

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

AspectJ named pointcut expression

Method-level configuration for transactional advice

Includes rewardAccountFor(..) and updateConfirmation(..)

Topics in this session



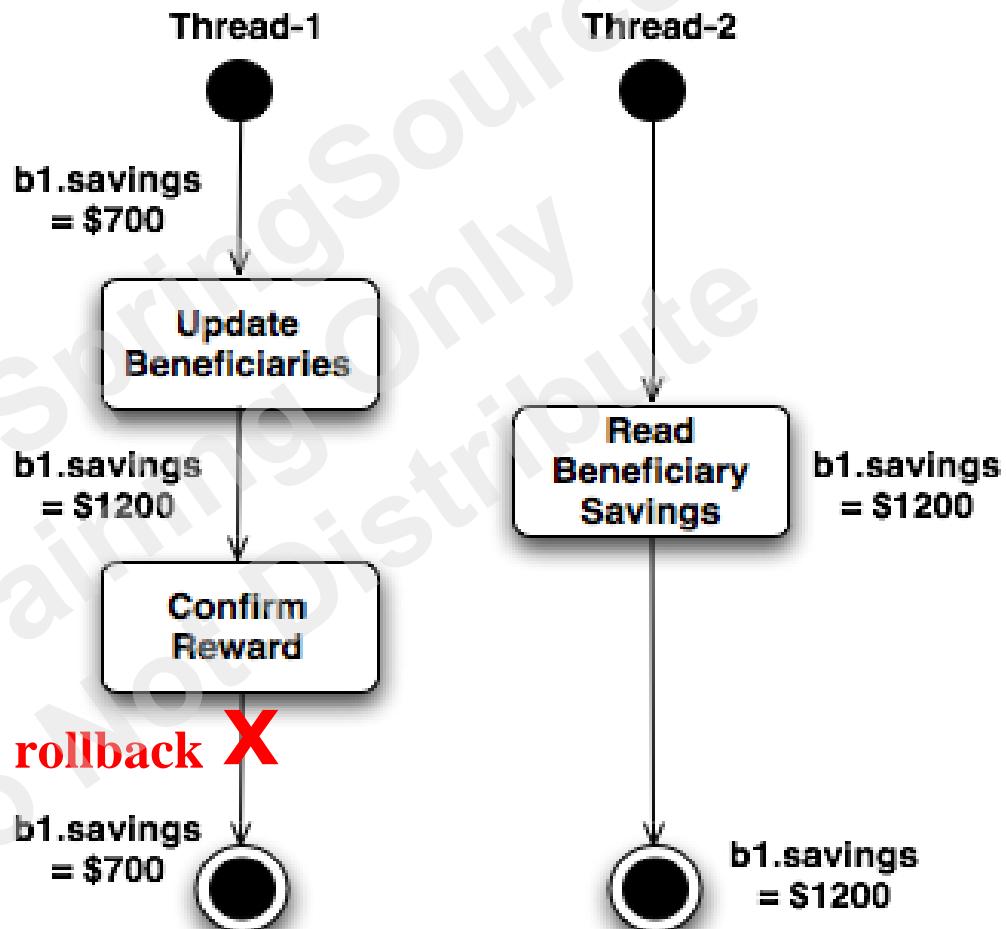
- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- Annotations or XML
- **Isolation levels**
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

Isolation levels

- 4 isolation levels can be used:
 - READ_UNCOMMITTED
 - READ_COMMITTED
 - REPEATABLE_READ
 - SERIALIZABLE
- Some DBMSs do not support all isolation levels
- Isolation is a complicated subject
 - DBMS all have differences in the way their isolation policies have been implemented
 - We just provide general guidelines

Dirty Reads

Transactions should be isolated – unable to see the results of another uncommitted unit-of-work



READ_UNCOMMITTED



- Lowest isolation level
- Allows *dirty reads*
- Current transaction can see the results of another uncommitted unit-of-work

```
public class RewardNetworkImpl implements RewardNetwork {  
    @Transactional (isolation=Isolation.READ_UNCOMMITTED)  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // atomic unit-of-work  
    }  
}
```

READ_COMMITTED

- Does not allow dirty reads
 - Only committed information can be accessed
- Default strategy for most databases

```
public class RewardNetworkImpl implements RewardNetwork {  
    @Transactional (isolation=Isolation.READ_COMMITTED)  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // atomic unit-of-work  
    }  
}
```

Highest isolation levels

- REPEATABLE_READ
 - Does not allow dirty reads
 - Non-repeatable reads are prevented
 - If a row is read twice in the same transaction, result will always be the same
 - Might result in locking depending on the DBMS
- SERIALIZABLE
 - Prevents non-repeatable reads and dirty-reads
 - Also prevents phantom reads

Topics in this session



- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- Annotations or XML
- Isolation levels
- **Transaction Propagation**
- Rollback rules
- Testing
- Advanced topics

Understanding Transaction Propagation: Example

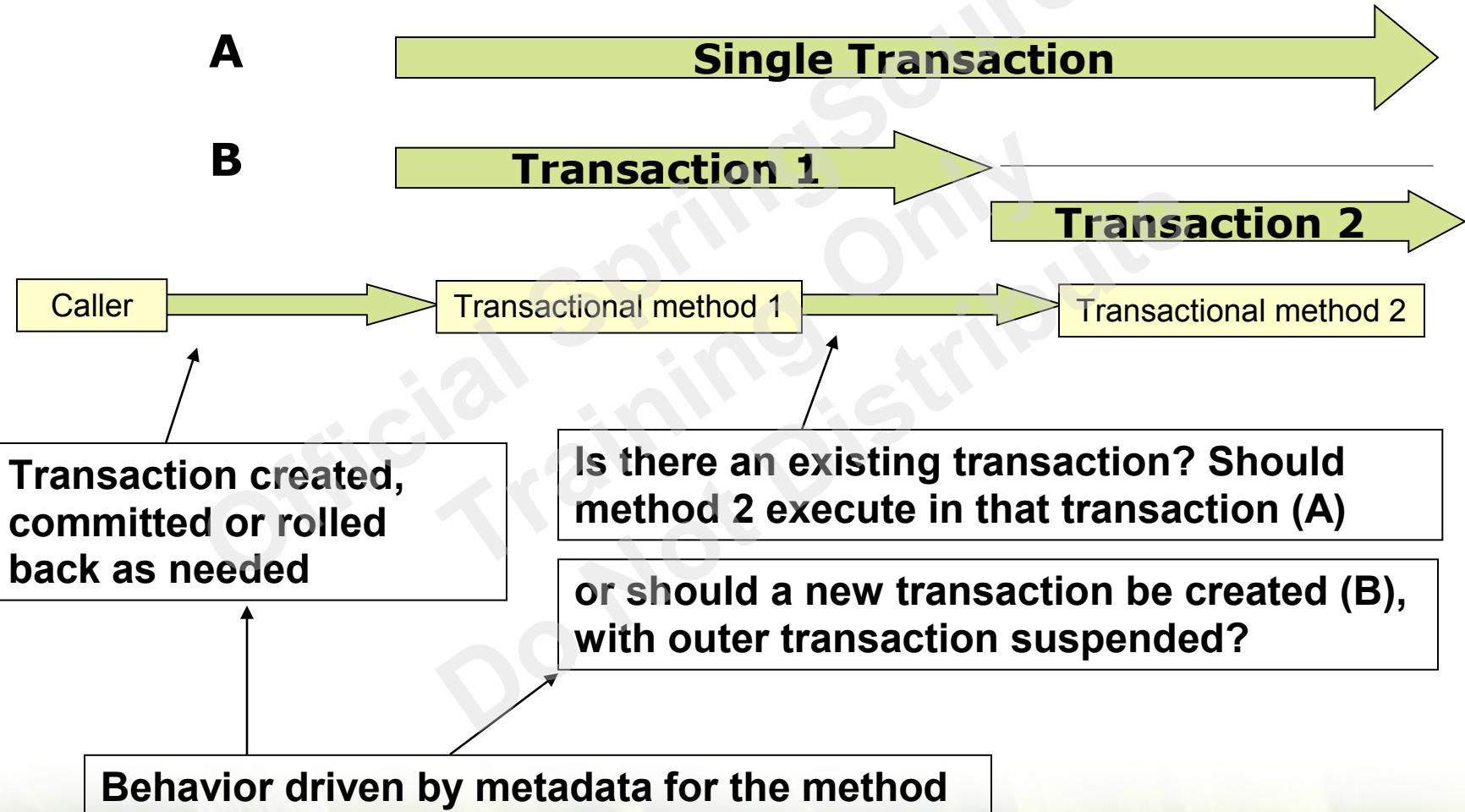


- Consider the sample below. What should happen if ClientServiceImpl calls AccountServiceImpl?
 - Should everything run into a single transaction?
 - Should each service have its own transaction?

```
public class ClientServiceImpl  
    implements ClientService {  
  
    @Autowired  
    private AccountService accountService;  
  
    @Transactional  
    public void updateClient(Client c)  
    { // ...  
        this.accountService.update(c.getAccounts());  
    }  
}
```

```
public class AccountServiceImpl  
    implements AccountService  
{  
    @Transactional  
    public void update(List <Account> l)  
    { // ... }  
}
```

Understanding Transaction Propagation



Transaction propagation with Spring

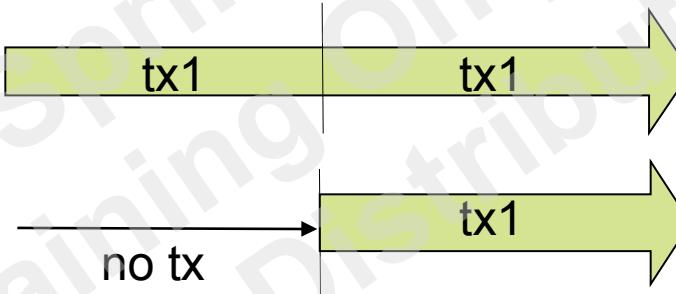


- 7 levels of propagation
- The following examples show *REQUIRED* and *REQUIRES_NEW*
 - *Check the documentation for other levels*
- Can be used as follows:

```
@Transactional( propagation=Propagation.REQUIRES_NEW )
```

REQUIRED

- REQUIRED
 - Default value
 - Execute within a current transaction, create a new one if none exists

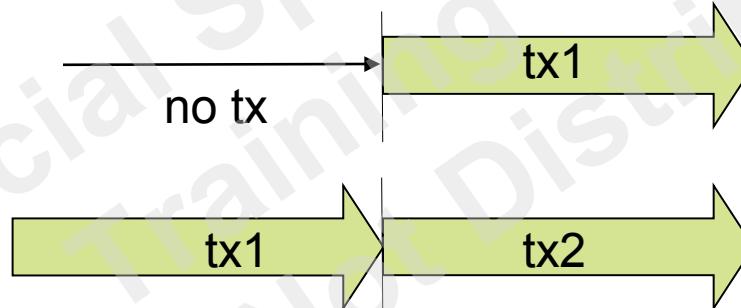


```
@Transactional(propagation=Propagation.REQUIRED)
```

REQUIRES_NEW



- REQUIRES_NEW
 - Create a new transaction, suspending the current transaction if one exists



```
@Transactional(propagation=Propagation.REQUIRES_NEW)
```

Topics in this session



- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- Annotations or XML
- Isolation Levels
- Transaction Propagation
- **Rollback rules**
- Testing
- Advanced topics

Default behavior

- By default, a transaction is rolled back if a `RuntimeException` has been thrown
 - Could be any kind of `RuntimeException`: `DataAccessException`, `HibernateException` etc.

```
public class RewardNetworkImpl implements RewardNetwork {  
    @Transactional  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // ...  
        throw new RuntimeException();  
    }  
}
```

A callout box with a black border and white background points from the word "RuntimeException" in the code to the text "Triggers a rollback" in the adjacent box.

Triggers a rollback

rollbackFor and noRollbackFor



- Default settings can be overridden with *rollbackFor/noRollbackFor* attributes

```
public class RewardNetworkImpl implements RewardNetwork {  
  
    @Transactional(rollbackFor=MyCheckedException.class)  
    public void updateConfirmation(Confirmation c) throws MyCheckedException {  
        // ...  
    }  
  
    @Transactional(noRollbackFor={JmxException.class, MailException.class})  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // ...  
    }  
}
```

Topics in this session



- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- Annotations or XML
- Isolation Levels
- Transaction Propagation
- Rollback rules
- **Testing**
- Advanced topics

@Transactional in an Integration Test



- Annotate test method (or type) with @Transactional to run test methods in a transaction that will be rolled back afterwards
 - No need to clean up your database after testing!

```
@ContextConfiguration(locations={"/rewards-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class RewardNetworkTest {

    @Test @Transactional
    public void testRewardAccountFor() {
        ...
    }
}
```

Advanced use of @Transactional in a Unit Test



```
@ContextConfiguration(locations={"/rewards-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
@TransactionConfiguration(defaultRollback=false, transactionManager="txMgr")
@Transactional
public class RewardNetworkTest {

    @Test
    @Rollback(true)
    public void testRewardAccountFor() {
        ...
    }
}
```

Transactions does a *commit* at the end by default

Overrides default *rollback* settings

 Inside *@TransactionConfiguration*, no need to specify the *transactionManager* attribute if the bean id is “transactionManager”

@Before vs @BeforeTransaction



```
@ContextConfiguration(locations={"/rewards-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class RewardNetworkTest {
    @BeforeTransaction
    public void verifyInitialDatabaseState() {...}
    @Before
    public void setUpTestDataInTransaction() {...}
    @Test @Transactional
    public void testRewardAccountFor() {
        ...
    }
}
```

Run before transaction is started

Within the transaction



@After and @AfterTransaction work in the same way as @Before and @BeforeTransaction



LAB

Managing Transactions Declaratively with Spring
Annotations

Topics in this session



- **Advanced topics**

- Programmatic transactions
- Read-only transactions
- Multiple transaction managers
- Distributed transactions

Official SpringSource
Training Only
Do Not Distribute

Programmatic Transactions with Spring



- Declarative transaction management is highly recommended
 - Clean code
 - Flexible configuration
- Spring does enable programmatic transaction
 - Works with local or JTA transaction manager



Can be useful inside a technical framework that would not rely on external configuration

Programmatic Transactions: example



```
public RewardConfirmation rewardAccountFor(Dining dining) {  
    ...  
    txTemplate = new TransactionTemplate(txManager);  
    return txTemplate.execute(new TransactionCallback() {  
        public Object doInTransaction(TransactionStatus status) {  
            try {  
                ...  
                accountRepository.updateBeneficiaries(account);  
                confirmation = rewardRepository.confirmReward(contribution, dining);  
            }  
            catch (RewardException e) {  
                status.setRollbackOnly();  
                confirmation = new RewardFailure();  
            }  
            return confirmation;  
        }  
    });  
}
```

No longer
@Transactional

Method no longer throws exception
– automatic rollback not possible
Must force rollback *manually*

Read-only transactions (1)



- Why use transactions if you're only planning to read data?
 - Performance: allows Spring to optimize the transactional resource for read-only data access

```
public void rewardAccount1() {  
    jdbcTemplate.queryForList(...);  
    jdbcTemplate.queryForInt(...);  
}  
  
@Transactional(readOnly=true)  
public void rewardAccount2() {  
    jdbcTemplate.queryForList(...);  
    jdbcTemplate.queryForInt(...);  
}
```

2 connections

1 single connection

Read-only transactions (2)



- Why use transactions if you're only planning to read data?
 - Isolation: with a high isolation level, a readOnly transaction prevents data from being modified until the transaction commits

```
@Transactional(readOnly=true, isolation=Isolation.READ_COMMITTED)
public void rewardAccount2() {
    jdbcTemplate.queryForList(...);
    jdbcTemplate.queryForInt(...);
}
```

Multiple Transaction Managers



- `@Transactional` can declare the id of the transaction manager that should be used

```
@Transactional("myOtherTransactionManager")
public void rewardAccount1() {
    jdbcTemplate.queryForList(...);
    jdbcTemplate.queryForInt(...);
}

@Transactional
public void rewardAccount2() {
    jdbcTemplate.queryForList(...);
    jdbcTemplate.queryForInt(...);
}
```

Uses the bean with id
"myOtherTransactionManager"

Uses "transactionManager"
bean by default

Distributed transactions



- A transaction might involve several data sources
 - 2 different databases
 - 1 database and 1 JMS queue
 - ...
- Distributed transactions often require specific drivers (XA drivers)
- In Java, Distributed transactions often rely on JTA
 - Java Transaction API

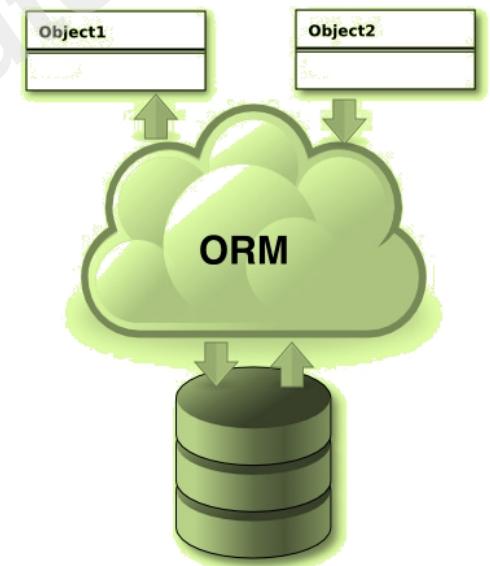
Distributed transactions – Spring integration



- Many possible strategies
 - Spring allows you to switch easily from a non-JTA to a JTA transaction policy
 - Just change the type of the transaction manager
- Reference:
 - Distributed transactions with Spring, with and without XA (Dr. Dave Syer)
 - <http://www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html>

ORM with Spring

Object Relational Mapping with
Spring and Java Persistence API



Topics in this session

- **Introduction to JPA**
 - General Concepts
 - Mapping
 - Querying
- Configuring an EntityManager in Spring
- Implementing JPA DAOs
- Exception Translation
- Lab
- Spring Data - JPA

Introduction to JPA



- The Java Persistence API is designed for operating on domain objects
 - Defined as POJO entities
 - No special interface required
- Replaces previous persistence mechanisms
 - EJB Entity Beans
 - Java Data Objects (JDO)
- A common API for object-relational mapping
 - Derived from the experience of existing products such as JBoss Hibernate and Oracle TopLink

About JPA

- Java Persistence API
 - Released May 2006
 - Version 2 since Dec 2009
 - Removes many of the limitations of JPA 1
 - Less need for ORM specific annotations and extensions
- Key Concepts
 - Entity Manager
 - Entity Manager Factory
 - Persistence Context

- **EntityManager**

- Manages a unit of work and persistent objects therein: the *PersistenceContext*
- Lifecycle often bound to a Transaction (i.e. container-managed)

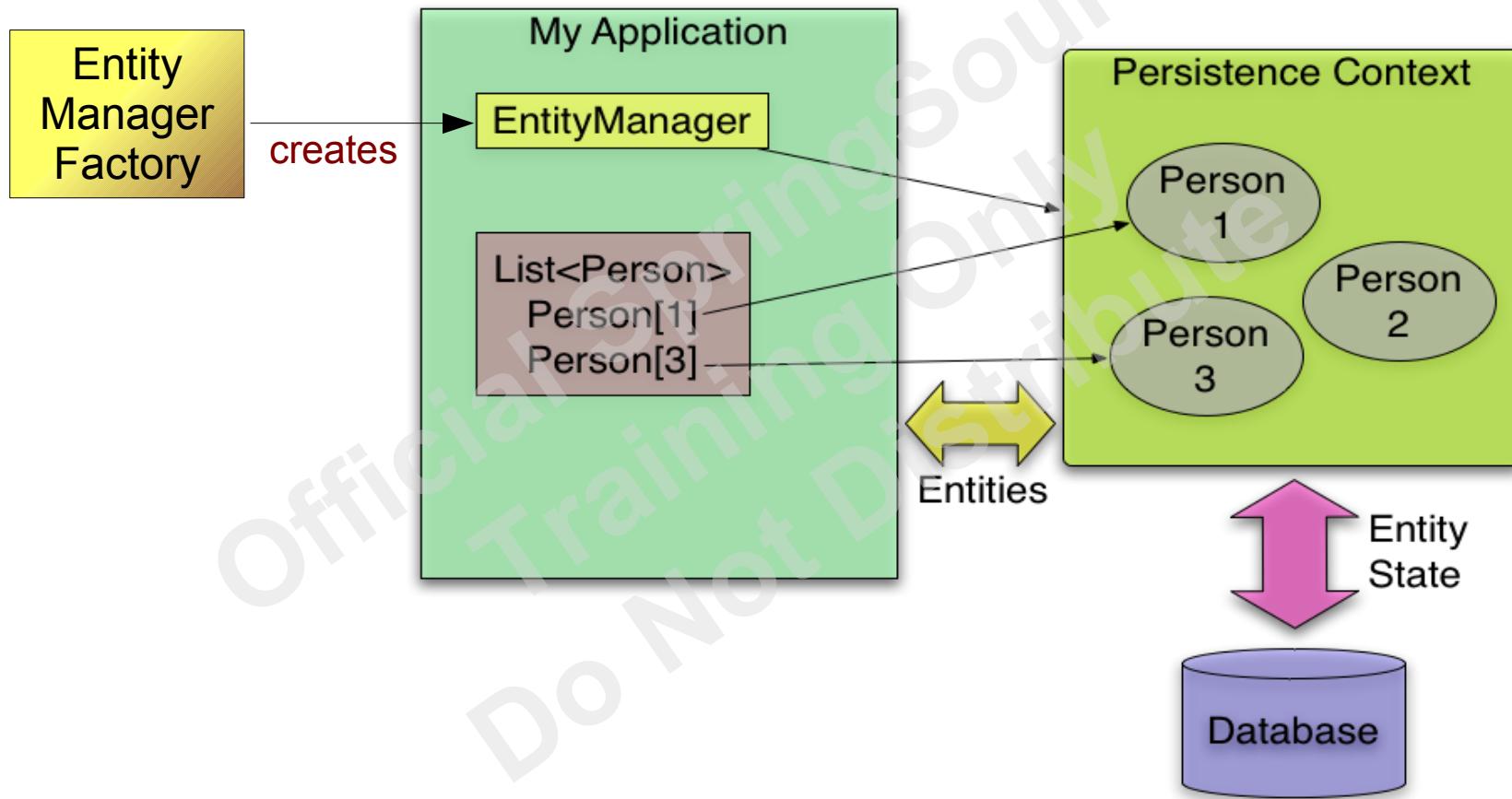
- **EntityManagerFactory**

- thread-safe, shareable object that represents a single data source / persistence unit
- Provides access to new application-managed EntityManagers

- **Persistence Unit**
 - Describes a group of persistent classes (entities)
 - Defines provider(s)
 - Defines transactional types (local vs JTA)
 - Multiple Units per application are allowed
- In a Spring JPA application
 - The configuration can be in the Persistence Unit
 - Or in the Spring bean-file
 - Or a combination of the two



Persistence Context and EntityManager



The EntityManager API



<code>persist(Object o)</code>	Adds the entity to the Persistence Context: <i>SQL: insert into table ...</i>
<code>remove(Object o)</code>	Removes the entity from the Persistence Context: <i>SQL: delete from table ...</i>
<code>find(Class entity, Object primaryKey)</code>	Find by primary key: <i>SQL: select * from table where id = ?</i>
<code>Query createQuery(String jpqlString)</code>	Create a JPQL query
<code>flush()</code>	Force changed entity state to be written to database immediately

Plus other methods ...

JPA Providers

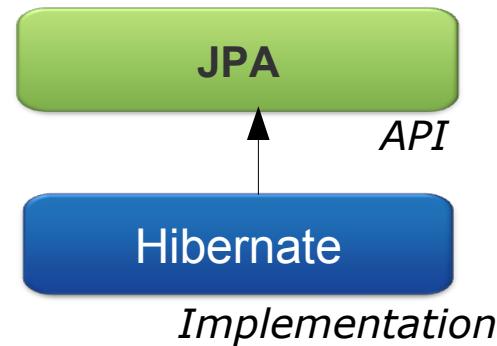


- Several major implementations of JPA spec
 - Hibernate EntityManager
 - Used inside JBoss
 - EclipseLink (RI)
 - Used inside Glassfish
 - Apache OpenJPA
 - Used IBM Websphere
 - Data Nucleus
 - Used by Google App Engine
- Can all be used without application server as well

Hibernate JPA

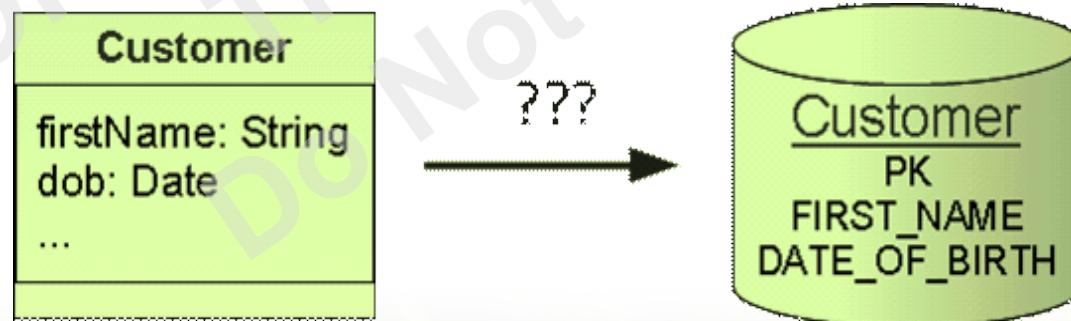


- Hibernate adds JPA support through an additional library
 - The *Hibernate EntityManager*
 - Hibernate sessions used behind JPA *interfaces*
 - Custom annotations for Hibernate specific extensions not covered by JPA
 - less important since JPA version 2



JPA Mapping

- JPA requires metadata for mapping classes/fields to database tables/columns
 - Usually provided as annotations
 - XML mappings also supported (`orm.xml`)
 - Intended for overrides only – not shown here
- JPA metadata relies on defaults
 - No need to provide metadata for the obvious



What can you Annotate?



- Classes
 - Applies to the entire class (such as table properties)
- Fields
 - Typically mapped to a column
 - By default, *all* treated as persistent
 - Mappings will be defaulted
 - Unless annotated with `@Transient` (non-persistent)
 - Accessed directly via Reflection
- Properties (getters)
 - Also mapped to a column
 - Annotate getters instead of fields



Mapping using fields (Data-members)



```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    @Id  
    @Column (name="cust_id")  
    private Long id;  
  
    @Column (name="first_name")  
    private String firstName;  
  
    @Transient  
    private User currentUser;  
  
    ...
```

Only `@Entity` and `@Id` are mandatory

Mark as an *entity*
Optionally override
table name

Mark *id-field*
(primary key)

Optionally override
column names

Not stored in database

Data members set *directly*
- using reflection
- “field” access
- no setters needed

Mapping using accessors (Properties)



Must place @Id on the *getter* method

Other annotations now also placed on *getter* methods



Beware of Side-Effects
getter/setter methods
may do additional work
– such as invoking listeners

```
@Entity @Table(name= "T_CUSTOMER")
public class Customer {
    private Long id;
    private String firstName;

    @Id
    @Column (name="cust_id")
    public String getId()
    { return this.id; }

    @Column (name="first_name")
    public String getFirstName()
    { return this.firstName; }

    public void setFirstName(String fn)
    { this.firstName = fn; }
}
```

Relationships

- Common relationship mappings supported
 - Single entities and entity collections both supported
 - Associations can be uni- or bi-directional

```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    @Id  
    @Column (name="cust_id")  
    private Long id;  
  
    @OneToMany(cascade=CascadeType.ALL)  
    @JoinColumn (name="cust_id")  
    private Set<Address> addresses;  
    ...  
}
```

Propagate all operations
to the child objects

Foreign key in
Address table

JoinTable also supported

Embeddables



```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    @Id  
    @Column (name="cust_id")  
    private Long id;  
  
    @Embedded  
    @AttributeOverride  
        (name="postcode", column=@Column(name="ZIP"))  
    private Address office;  
    ...}
```

```
@Embeddable  
public class Address {  
    private String street;  
    private String suburb;  
    private String city;  
    private String postcode;  
    private String country;  
}
```

maps to ZIP
column in
T_CUSTOMER

- Map a table row to multiple classes
 - Address fields also columns in T_CUSTOMER
 - @AttributeOverride overrides mapped column name

- JPA provides several options for accessing data
 - Retrieve an object by primary key
 - Query for objects using JPA Query Language (JPQL)
 - Similar to SQL and HQL
 - Query for objects using Criteria Queries
 - API for creating ad hoc queries
 - Only in JPA 2
 - Execute SQL directly to underlying database
 - “Native” queries, allow DBMS-specific SQL to be used
 - Consider JdbcTemplate instead when not using managed objects – more options/control, more efficient

JPA Querying: By Primary Key



- To retrieve an object by its database identifier simply call **find** on the EntityManager

```
Long customerId = 123L;  
Customer customer = entityManager.find(Customer.class, customerId);
```

returns **null** if no object exists for the identifier

No cast required – JPA uses generics

JPA Querying: JPQL



- To query for objects based on properties or associations use JPQL

- SELECT clause required
- can't use *

```
// Query with named parameters
```

```
Query query = entityManager.createQuery(  
    "select c from Customer c where c.address.city = :city");  
query.setParameter("city", "Chicago");  
List customers = query.getResultSet();
```

```
// ... or using a single statement
```

```
List customers2 = entityManager.createQuery("from Customer c ...").  
    setParameter("city", "Chicago").getResultSet();
```

```
// ... or if expecting a single result
```

```
Customer customer = (Customer) query.getSingleResult();
```

Can also use bind ? variables – indexed from 1 like JDBC

- Criteria Query API (JPA 2)
 - Build type safe queries: fewer run-time errors
 - Much more verbose

```
public List<Customer> findByLastName(String lastName) {  
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();  
    CriteriaQuery<Customer> cq = builder.createQuery(Customer.class);  
    Predicate condition =  
        builder.eq( cq.from(Customer.class).get(Customer_.name), lastName);  
    cq.where(condition);  
  
    return entityManager.createQuery(cq).getResultSet();  
}
```

Meta-data class
created by JPA
(note underscore)

JPA Querying: SQL



- Use a *native* query to execute raw SQL

```
// Query for multiple rows
```

```
Query query = entityManager.createNativeQuery(  
    "SELECT cust_num FROM T_CUSTOMER c WHERE cust_name LIKE ?");  
query.setParameter(1, "%ACME%");  
List<String> customerNumbers = query.getResultSet();
```

No named parameter support

Indexed from 1
– like JDBC

```
// ... or if expecting a single result
```

```
String customerNumber = (String) query.getSingleResult();
```

```
// Query for multiple columns
```

```
Query query = entityManager.createNativeQuery(  
    "SELECT ... FROM T_CUSTOMER c WHERE ...");  
List<Object[]> customers = query.getResultSet();
```

Object[] per row,
element per field
fetched

Topics in this session

- Introduction to JPA
 - General Concepts
 - Mapping
 - Querying
- **Configuring an EntityManager in Spring**
- Implementing JPA DAOs
- Exception Translation
- Lab
- Spring Data - JPA

Setting up an EntityManagerFactory



- Three ways to set up an EntityManagerFactory:
 - LocalEntityManagerFactoryBean
 - LocalContainerEntityManagerFactoryBean
 - Use a JNDI lookup
- **persistence.xml** required for configuration
 - Spring 3.1 allows no *persistence.xml* with LocalContainerEntityManagerFactoryBean

- Always stored in META-INF
- Specifies:
 - persistence unit
 - optional vendor-dependent information

```
<persistence version="1.0"  
    xmlns="http://java.sun.com/xml/ns/persistence"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">  
  
<persistence-unit name="rewardNetwork"/>  
  
</persistence>
```

<?xml?>

LocalEntityManagerFactoryBean



- Useful for standalone apps, integration tests
- Cannot specify a DataSource
 - Useful when only data access is via JPA
 - Uses standard JPA service location (SPI) mechanism
`/META-INF/services/javax.persistence.spi.PersistenceProvider`

```
<bean id="entityManagerFactory"
      class="o.s.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName"
              value="rewardNetwork"/>
</bean>
```

LocalContainer EntityManagerFactoryBean



- Provides full JPA capabilities
- Integrates with existing DataSources
- Useful when fine-grained customization needed
 - Can specify vendor-specific configuration



Configuration using Spring



```
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="persistenceUnitName" value="rewardNetwork"/> ←

  <property name="jpaVendorAdapter">
    <bean class="org.sfwk.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="showSql" value="true"/>
      <property name="generateDdl" value="true"/>
      <property name="database" value="HSQL"/>
    </bean>
  </property>

  <property name="jpaProperties">
    <props>
      <prop key="hibernate.format_sql">true</prop>
    </props>
  </property>
</bean>
```

```
  <persistence version="1.0" ...>
    <persistence-unit name="rewardNetwork"/>
  </persistence>
```

NOTE: no persistence.xml needed when using new packagesToScan property in Spring 3.1

Configuration using Spring and Persistence Unit

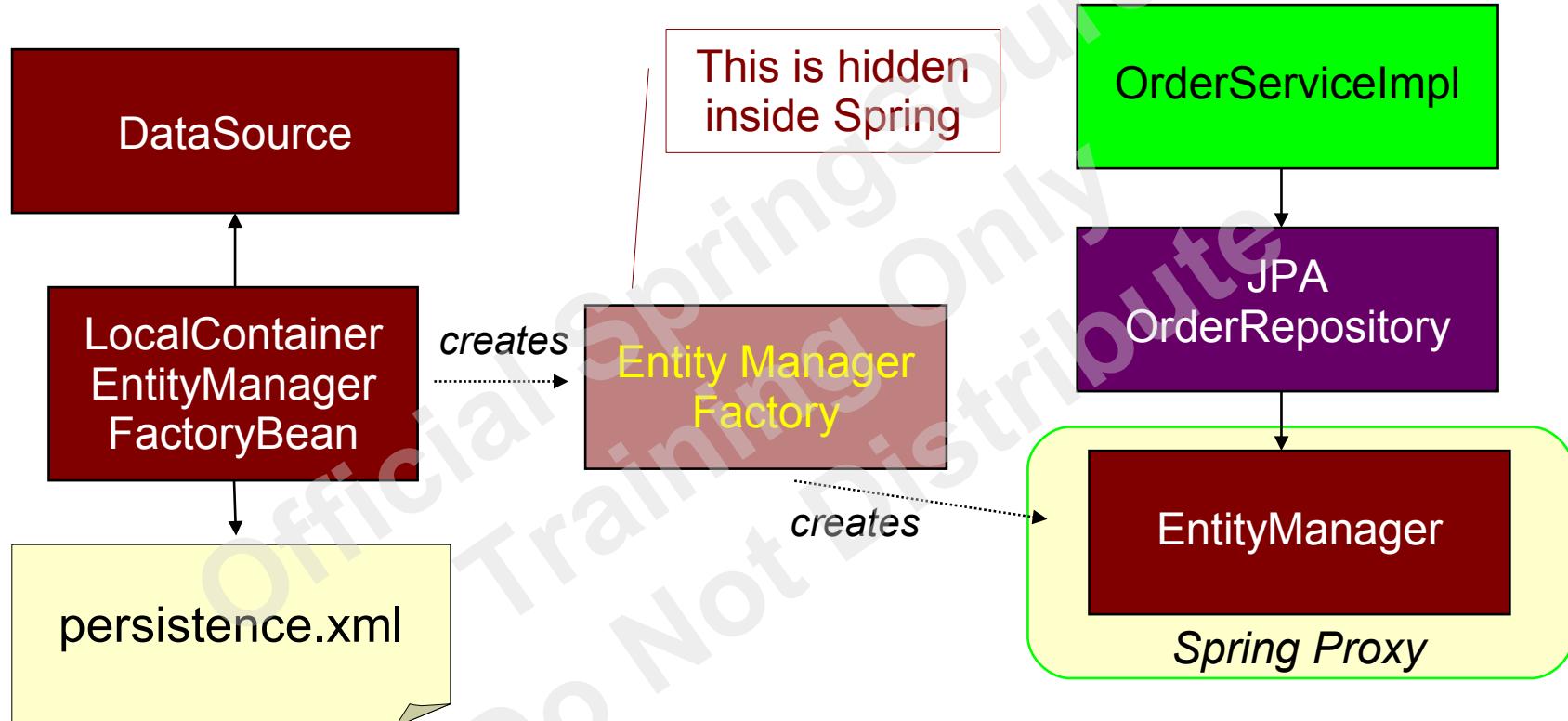


```
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="persistenceUnitName" value="rewardNetwork"/>
</property>
</bean>
```

```
<persistence-unit name="rewardNetwork">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <properties>
    <property name="hibernate.dialect"
      value="org.hibernate.dialect.HSQLDialect"/>
    <property name="hibernate.hbm2ddl.auto" value="create"/>
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.format_sql" value="true" />
  </properties>
</persistence-unit>
```

If using JTA – declare <jta-data-source> in the persistence-unit

EntityManagerFactoryBeans



Proxy automatically finds entity-manager for current transaction

JNDI Lookups



- A *jee:jndi-lookup* can be used to retrieve EntityManagerFactory from application server
- Useful when deploying to JEE Application Servers (WebSphere, WebLogic, etc.)

```
<jee:jndi-lookup id="entityManagerFactory"  
                  jndi-name="persistence/rewardNetwork"/>
```

Topics in this session

- Introduction to JPA
 - General Concepts
 - Mapping
 - Querying
- Configuring an EntityManager in Spring
- **Implementing JPA DAOs**
- Exception Translation

Implementing JPA DAOs



- JPA provides configuration options so Spring can manage transactions and the EntityManager
- Use AOP for transparent exception translation to Spring's DataAccessException hierarchy
- There are *no* Spring dependencies in your DAO implementations

Spring-managed Transactions and EntityManager (1)



- To transparently participate in Spring-driven transactions
 - Simply use one of Spring's FactoryBeans for building the EntityManagerFactory
 - Inject an EntityManager reference with @PersistenceContext
- Define a transaction manager
 - JpaTransactionManager
 - JtaTransactionManager

Spring-managed Transactions and EntityManager (2)



- The code – no Spring dependencies

```
public class JpaOrderRepository implements OrderRepository {  
    private EntityManager entityManager;  
  
    @PersistenceContext  
    public void setEntityManager (EntityManager entityManager) {  
        this.entityManager = entityManager;  
    }  
  
    public Order findOrderId(long orderId) {  
        return entityManager.find(Order.class, orderId);  
    }  
}
```

Automatic injection of EM Proxy

Proxy resolves to EM when used

Spring-managed Transactions and EntityManager (3)



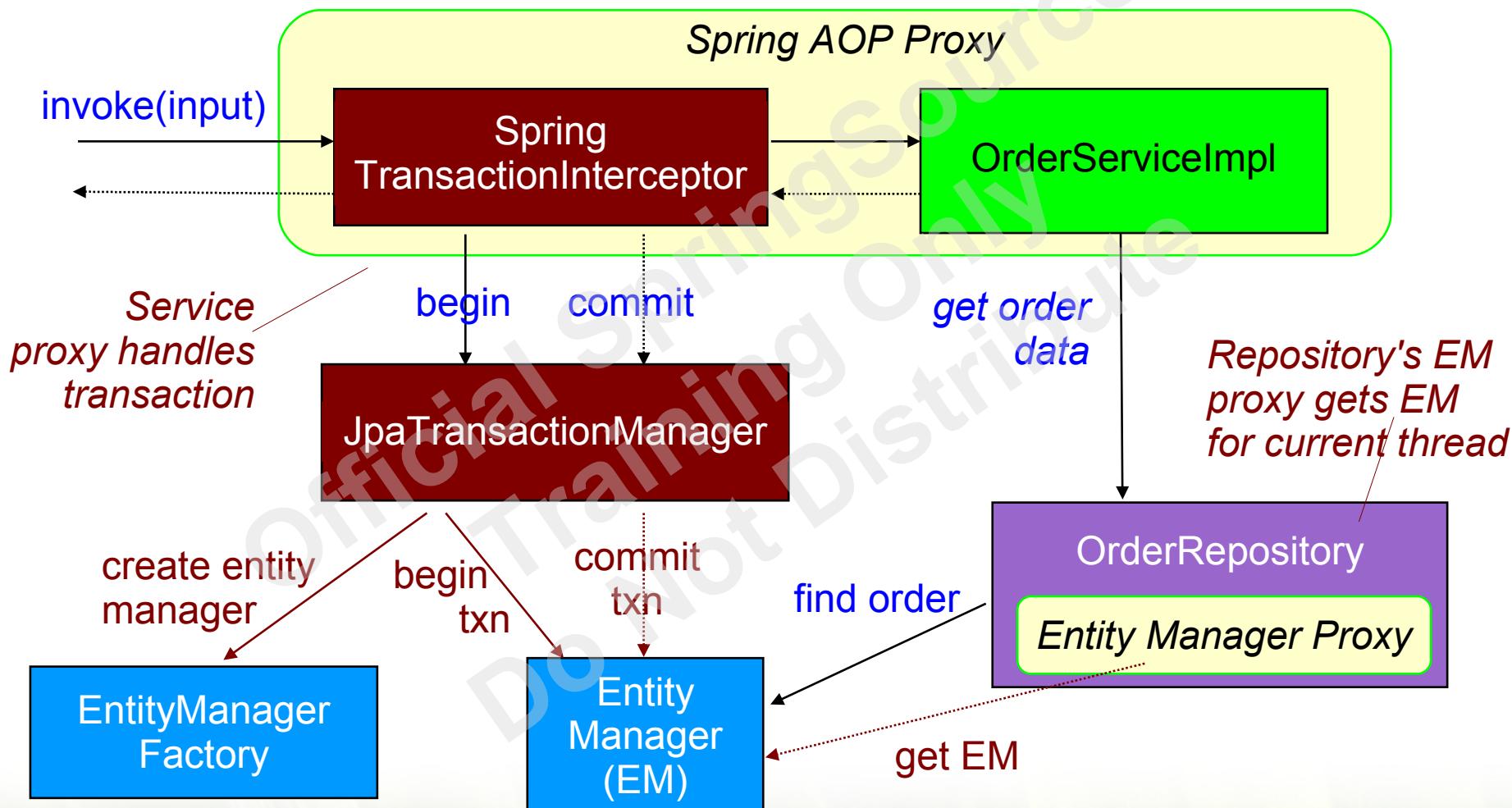
- The configuration

```
<beans>
  <bean id="entityManagerFactory"
    class="org...orm.jpa.LocalContainerEntityManagerFactoryBean">
    ...
  </bean>
  <bean id="orderRepository" class="JpaOrderRepository"/>
  <bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
  </bean>
  <context:annotation-config/>
</beans>
```

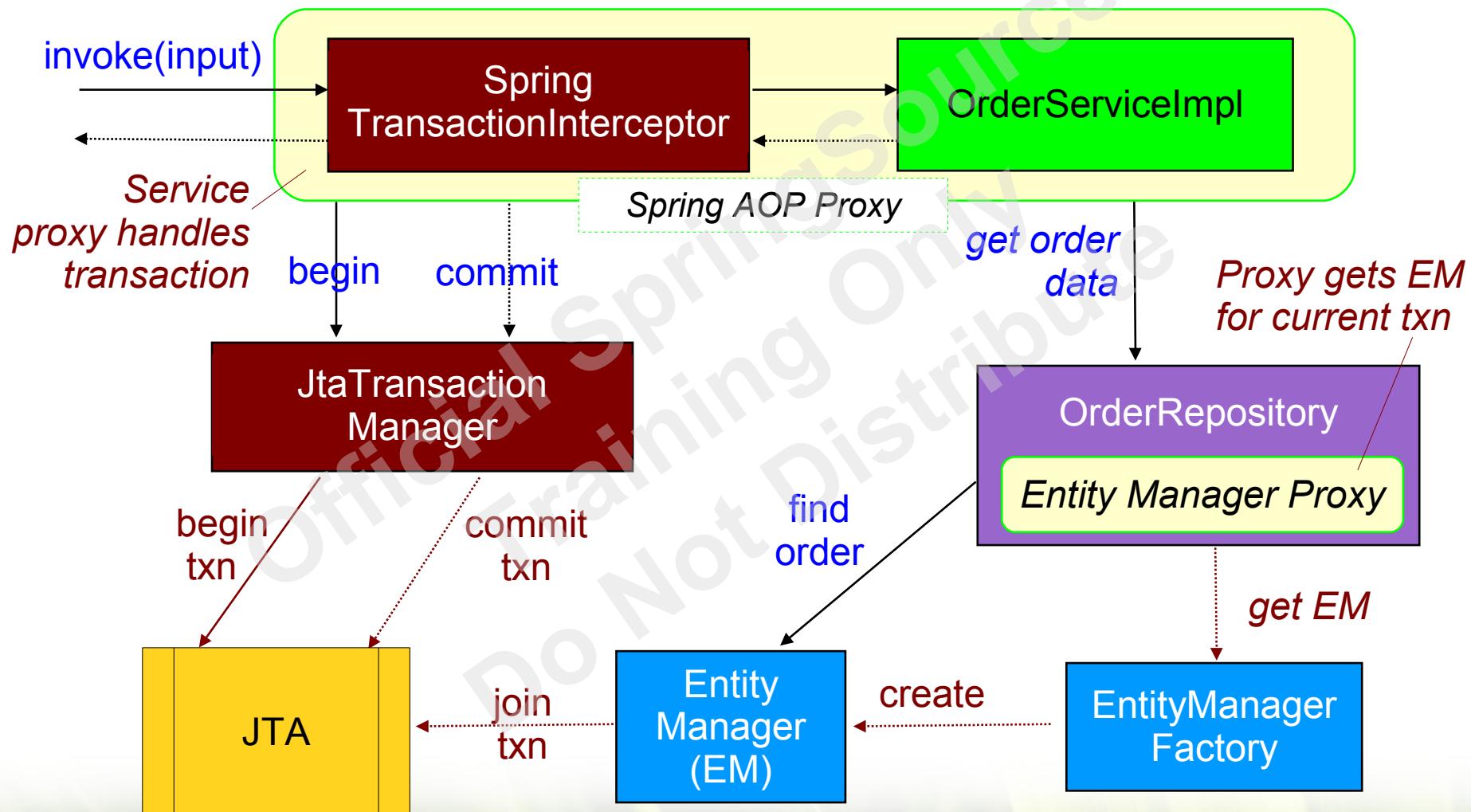
Automatic injection of entity-manager proxy

Provides injection via @PersistenceContext

How it Works (JPA)



How it Works (JTA)



Topics in this session

- Introduction to JPA
 - General Concepts
 - Mapping
 - Querying
- Configuring an EntityManager in Spring
- Implementing JPA DAOs
- **Exception Translation**
- Lab
- Spring Data - JPA

Transparent Exception Translation (1)



- Used as-is, the previous DAO implementation will throw unchecked **PersistenceExceptions**
 - Not desirable to let these propagate up to the service layer or other users of the DAOs
 - Introduces dependency on the specific persistence solution that should not exist
- AOP allows translation to Spring's rich, vendor-neutral **DataAccessException** hierarchy
 - Hides access technology used

Transparent Exception Translation (2)



- Spring provides this capability out of the box
 - Annotate with **@Repository**
 - Define a Spring-provided BeanPostProcessor

```
@Repository  
public class JpaOrderRepository implements OrderRepository {  
    ...  
}
```

```
<bean class="org.springframework.dao.annotation.  
PersistenceExceptionTranslationPostProcessor"/>
```

Transparent Exception Translation (3)



- Can't always use annotations
 - For example with a third-party repository
 - Use XML to configure instead

```
public class JpaOrderRepository implements OrderRepository {  
    ...  
}
```

No annotations

```
<bean id="persistenceExceptionInterceptor"  
      class="org.springframework.dao.support.  
          PersistenceExceptionTranslationInterceptor"/>  
  
<aop:config>  
    <aop:advisor pointcut="execution(* *..OrderRepository+.*(..))"  
                 advice-ref="persistenceExceptionInterceptor" />  
</aop:config>
```

Features in JPA 2



JPA 2

- Improved mappings
 - Compound keys: @IdClass
 - Nested embeddables
 - Collections of non-entities and primitives
 - Support for maps and lists in relationships
- Fewer limitations
 - Criteria and type-safe query API
 - ORM specific getters/setters: @Access
 - Full support for @JoinTable and @JoinColumn
 - Locking modes: Optimistic and Pessimistic
 - Supports JSR 303 validation: @NotNull, @Max ...



LAB

Using JPA with Spring

Topics in this session

- Introduction to JPA
 - General Concepts
 - Mapping
 - Querying
- Configuring an EntityManager in Spring
- Implementing JPA DAOs
- Exception Translation
- Lab
- **Spring Data - JPA**

What is Spring Data?

- Reduces boiler plate code for data access
 - Works in many environments



Instant Repositories



- How?
 - Annotate domain classes suitably to define keys and enable persistence
 - Define your repository as an *interface*
- Spring will implement it at run-time
 - Scans for interfaces implementing Repository <T, K>
 - CRUD methods auto-generated
 - Paging, custom queries and sorting supported
 - Variations exist for most Spring Data sub-projects

```
<jpa:repositories base-package="com.acme.**.repository" />  
  
<mongo:repositories base-package="com.acme.**.repository" />  
  
<gfe:repositories base-package="com.acme.**.repository" />
```

Annotate Domain Class

- You do this for JPA anyway
 - Spring Data defines similar annotations for other storage products

```
@Entity @Table(...)  
public class Customer {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
  
    private Date firstOrderDate;  
    private String email;  
  
    // Other data-members  
}
```

JPA – map to a Table

```
@Document  
public class Customer {  
    ...
```

MongoDB – map to a JSON document

```
@Region  
public class Customer {  
    ...
```

Gemfire – map to a region

Extend Predefined Repository Interfaces



```
public interface Repository<Customer, Long> { }
```

Marker interface – add any methods from CrudRepository or finders

```
public interface CrudRepository<T, ID  
extends Serializable> extends Repository<T, ID> {
```

```
public <S extends T> save(S entity);  
public <S extends T> Iterable<S> save(Iterable<S> entities);
```

```
public T findOne(ID id);  
public Iterable<T> findAll();
```

You get all these methods automatically

```
public void delete(ID id);  
public void delete(T entity);  
public void deleteAll();
```

PagingAndSortingRepository<T, K>
- adds Iterable<T> findAll(Sort)
- adds Page<T> findAll(Pageable)

```
}
```

Defining a JPA Repository



- Auto-generated finders obey naming convention
 - findBy<*DataMember*><*Op*>
 - <*Op*> can be Gt, Lt, Ne, Between, Like ... etc

```
public interface CustomerRepository  
    extends CrudRepository<Customer, Long> {  
  
    public Customer findByEmail(String someEmail);      // No <Op> for Equals  
    public Customer findByFirstOrderDateGt(Date someDate);  
    public Customer findByFirstOrderDateBetween(Date d1, Date d2);  
  
    @Query("SELECT c FROM Customer c WHERE c.email NOT LIKE '%@%'")  
    public List<Customer> findInvalidEmails();  
}
```

ID

Custom query uses query-language of underlying product (here JPQL)

Convention over Configuration



- Note: Repository is an interface (*not a class!*)

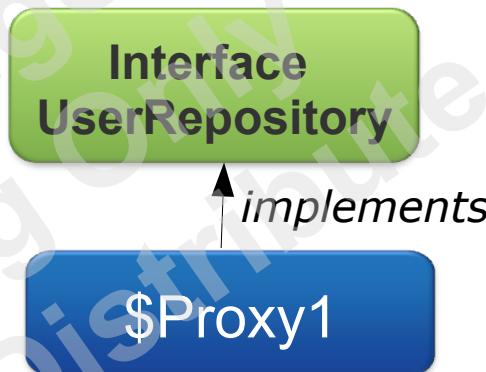
```
import org.springframework.data.repository.Repository;  
import org.springframework.data.jpa.repository.Query;  
  
public interface UserRepository extends Repository<User, Long> {  
  
    <S extends User> save(S entity); // Definition as per CRUDRepository  
  
    User findById(long i); // Query determined from method name  
  
    User findByNameIgnoreCase(String name); // Case insensitive search  
  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmail(String email); // ?1 replaced by method param  
}
```

Internal behavior

- Before startup



- After startup



```
<jpa:repositories base-package="com.acme.repository"/>
```



You can conveniently use Spring to inject a dependency of type UserRepository. Implementation will be generated at startup time.

Summary

- Use 100% JPA to define entities and access data
 - Repositories have no Spring dependency
- Use Spring to configure JPA entity-manager factory
 - Smart proxy works with Spring-driven transactions
 - Optional translation to DataAccessExceptions

Spring-Hibernate – 3 day in-depth JPA and Hibernate course
– emphasis on Hibernate, but many JPA features used also



Overview of Spring Web

Developing modern web applications
with Spring

Topics in this Session



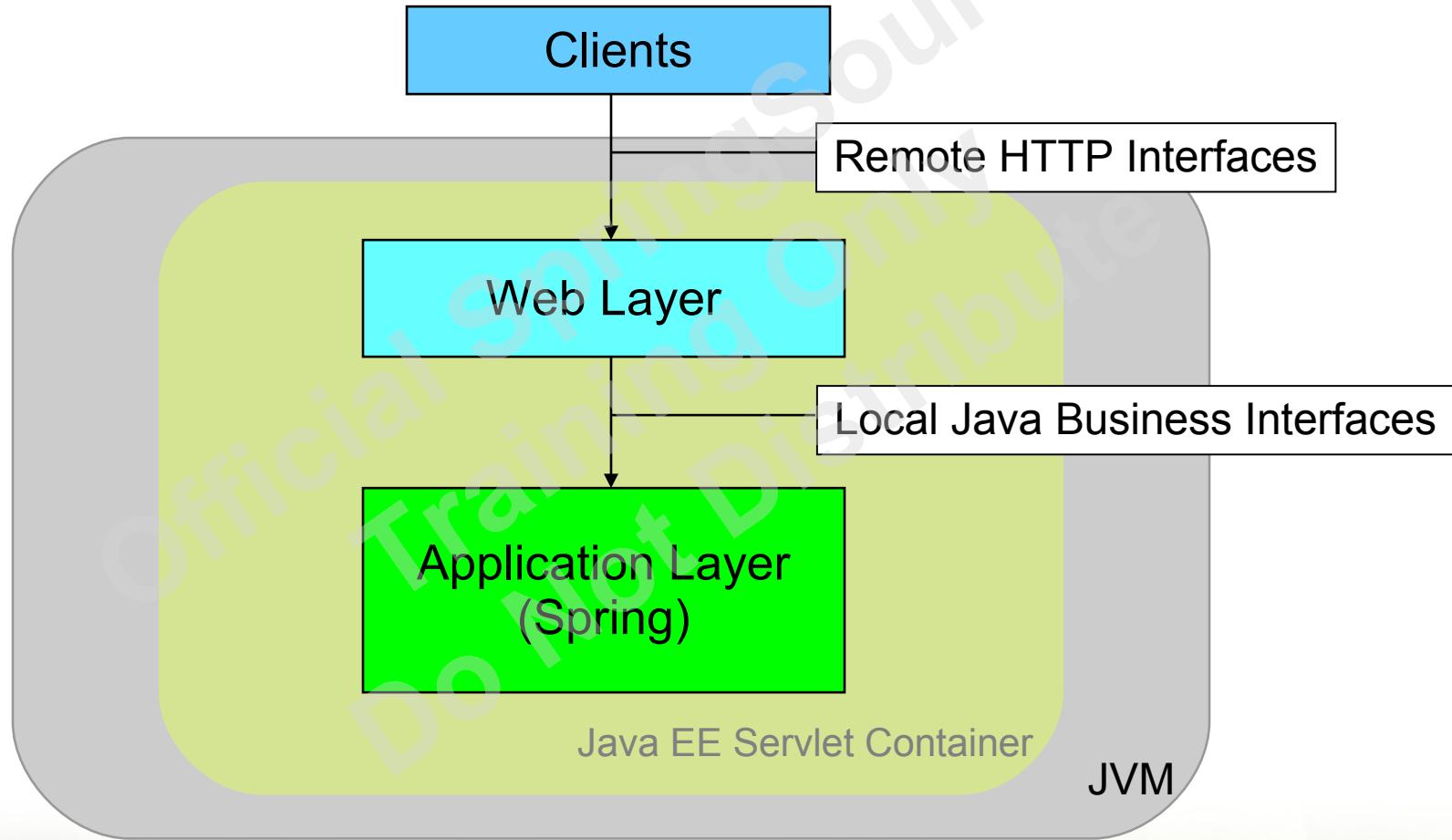
- **Introduction**
- Using Spring in Web Applications
- Overview of Spring Web
- Spring and other Web frameworks

Official Training Only
Do Not Distribute

Web layer integration

- SpringSource provides support in the Web layer
 - Spring MVC, Spring WebFlow...
- However, you are free to use Spring with any Java web framework
 - Integration might be provided by Spring or by the other framework itself

Effective Web Application Architecture



Topics in this Session

- Introduction
- **Using Spring in Web Applications**
- Overview of Spring Web
- Spring and other Web frameworks

Official SpringSource
Training Only
Do Not Distribute

Spring Application Context Lifecycle in Webapps



- Spring can be initialized within a webapp
 - start up business services, repositories, etc.
- Uses a standard servlet listener
 - initialization occurs before any servlets execute
 - application ready for user requests
 - `ApplicationContext.close()` is called when the application is stopped

Configuration in web.xml



- Just add a Spring-provided servlet listener

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/merchant-reporting-webapp-config.xml
    </param-value>
</context-param>
```

web.xml

The application context's configuration file(s)

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

Loads the ApplicationContext into the ServletContext
before any Servlets are initialized

Configuration Options

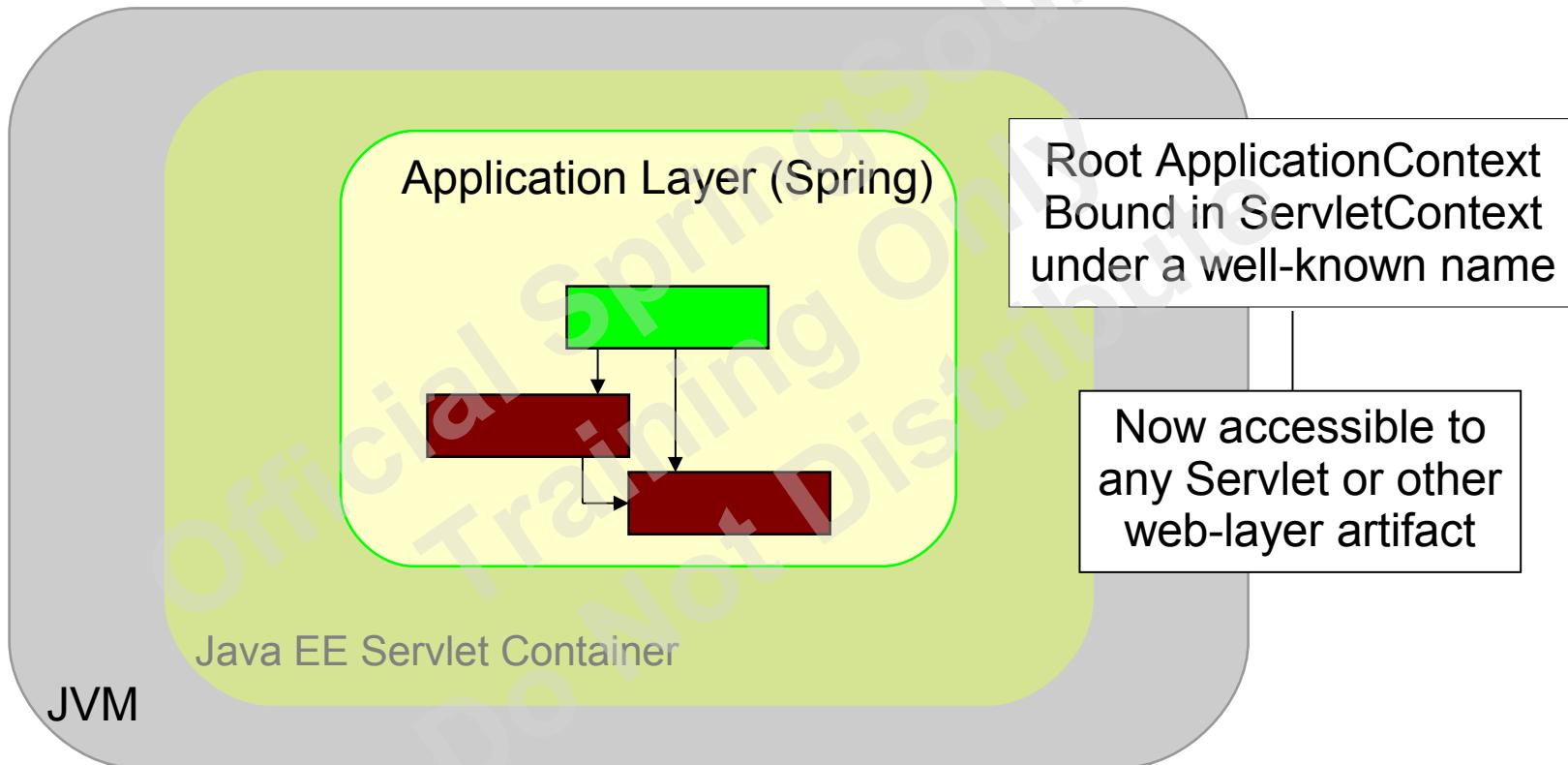


- Default resource location is document-root
 - Can use *classpath:* designator
 - Defaults to WEB-INF/applicationContext.xml
- Can optionally select profile to use

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:/rewards/internal/application-config.xml
        /WEB-INF/merchant-reporting-webapp-config.xml
    </param-value>
</context-param>
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>web</param-value>
</context-param>
```

web.xml

Servlet Container After Starting Up



- Static helper class
 - Provides access to the Spring ApplicationContext
 - Spring retrieves the ApplicationContext from the ServletContext

```
ServletContext servletContext = ...;
```

```
ApplicationContext context = WebApplicationContextUtils.  
    getRequiredWebApplicationContext(servletContext);
```



Only necessary when dependencies are not directly injected (eg. in a Servlet)

WebApplicationContextUtils (2)



- Example using a plain Java servlet

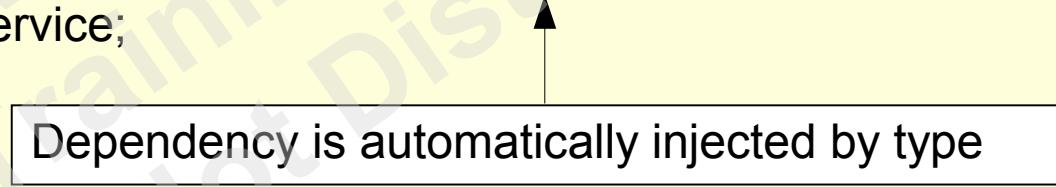
```
public class TopSpendersReportGenerator extends HttpServlet {  
    private ClientService clientService;  
  
    public void init() {  
        ApplicationContext context = WebApplicationContextUtils.  
            getRequiredWebApplicationContext(getServletContext());  
        clientService = (ClientService) context.getBean("clientService");  
    }  
    ...  
}
```

Saves the reference to the application entry-point for invocation
during request handling

Dependency injection

- Example using Spring MVC

```
@Controller  
public class TopSpendersReportController {  
    private ClientService clientService;  
  
    @Autowired  
    public TopSpendersReportController(ClientService service) {  
        this.clientService = service;  
    }  
    ...  
}
```



Dependency is automatically injected by type



No need for *WebApplicationContextUtils* anymore

Dependency injection

- Example using Spring MVC

```
@Controller  
public class TopSpendersReportController {  
    private ClientService clientService;  
  
    @Autowired  
    public TopSpendersReportController(ClientService service) {  
        this.clientService = service;  
    }  
    ...  
}
```

Dependency is automatically injected by type



No need for *WebApplicationContextUtils* anymore

Topics in this Session

- Introduction
- Using Spring in Web Applications
- **Overview of Spring Web**
- Spring and other Web frameworks

Official SpringSource
Training Only
Do Not Distribute

- Spring MVC
 - Web framework bundled with Spring
- Spring WebFlow
 - Plugs into Spring MVC
 - Implements navigation flows
- Spring BlazeDS Integration
 - Integration between Adobe Flex clients and Spring applications

- Spring's web framework
 - Uses Spring for its own configuration
 - Controllers are Spring beans
 - testable artifacts
- Annotation-based model since Spring 2.5
- Builds on the Java Servlet API
 - A parallel Portlet version is also provided
- The core platform for developing web applications with Spring
 - All higher-level modules such as WebFlow build on it

Spring Web Flow



- Plugs into Spring Web MVC as a Controller technology for implementing stateful "flows"
 - Checks that users follow the right navigation path
 - Manages back button and multiple windows issues
 - Provides scopes beyond request and session
 - such as the *flow* and *flash* scope
 - Addresses the double-submit problem elegantly

Example Flow Definition

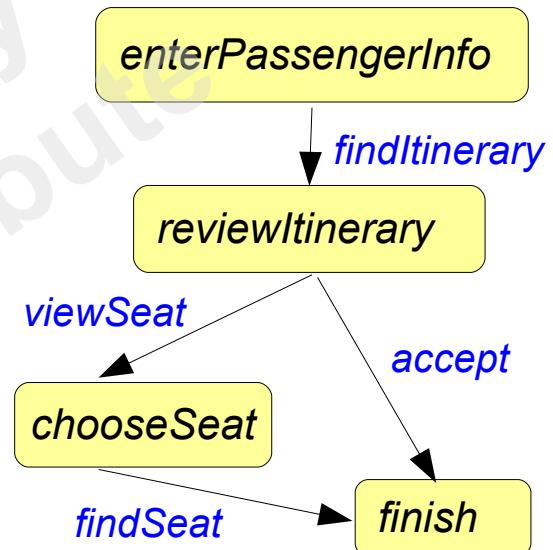
- Flows are declared in Xml

```
<flow ...>
  <view-state id="enterPassengerInfo">
    <transition on="findItinerary" to="reviewItinerary" />
  </view-state>

  <view-state id="reviewItinerary">
    <transition on="viewSeat" to="chooseSeat" />
    <transition on="accept" to="finish" />
  </view-state>

  <view-state id="chooseSeat">
    <transition on="findSeat" to="finish" />
  </view-state>

  <end-state id="finish"/>
</flow>
```



More about WebFlow

- Online sample application is available here:
<http://richweb.springframework.org/swf-booking-faces/spring/intro>
- Sample applications can be downloaded here:
<http://www.springsource.org/webflow-samples>



Topics in this Session

- Introduction
- Using Spring in Web Applications
- Overview of Spring Web
- **Spring and other Web frameworks**

Official Training Only
Do Not Distribute

Spring – Struts 1 integration



- Integration provided by the Spring framework
 - Inherit from ActionSupport instead of Action

```
public class UserAction extends ActionSupport {  
    public ActionForward execute(ActionMapping mapping,  
                                ActionForm form,...) throws Exception {  
        WebApplicationContext ctx = getWebApplicationContext();  
        UserManager mgr = (UserManager) ctx.getBean("userManager");  
        return mapping.findForward("success");  
    }  
}
```

Provided by the
Spring framework



This is one of the 2 ways to integrate Spring and Struts 1 together. More info in the [reference documentation](#)

Spring – JSF integration



- Two options
 - Spring-centric integration
 - Provided by Spring Faces
 - JSF-centric integration
 - Spring plugs in as a JSF managed bean provider

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>com.springsource.web.ClientController</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

JSF-centric integration

Integration provided by other frameworks



- Struts 2
 - provides a Spring plugin
<http://struts.apache.org/2.0.8/docs/spring-plugin.html>
- Wicket
 - Comes with an integration to Spring
<http://cwiki.apache.org/WICKET/spring.html>
- Tapestry 5
 - Comes with an integration to Spring
<http://tapestry.apache.org/tapestry5/tapestry-spring/>

Summary

- Spring can be used with any web framework
 - Spring provides the ContextLoaderListener that can be declared in web.xml
- Spring MVC is a lightweight web framework where controllers are Spring beans
 - More about Spring MVC in the next module
- WebFlow plugs into Spring MVC as a Controller technology for implementing stateful "flows"



Spring Web MVC Essentials

Getting started with Spring Web MVC

Topics in this Session



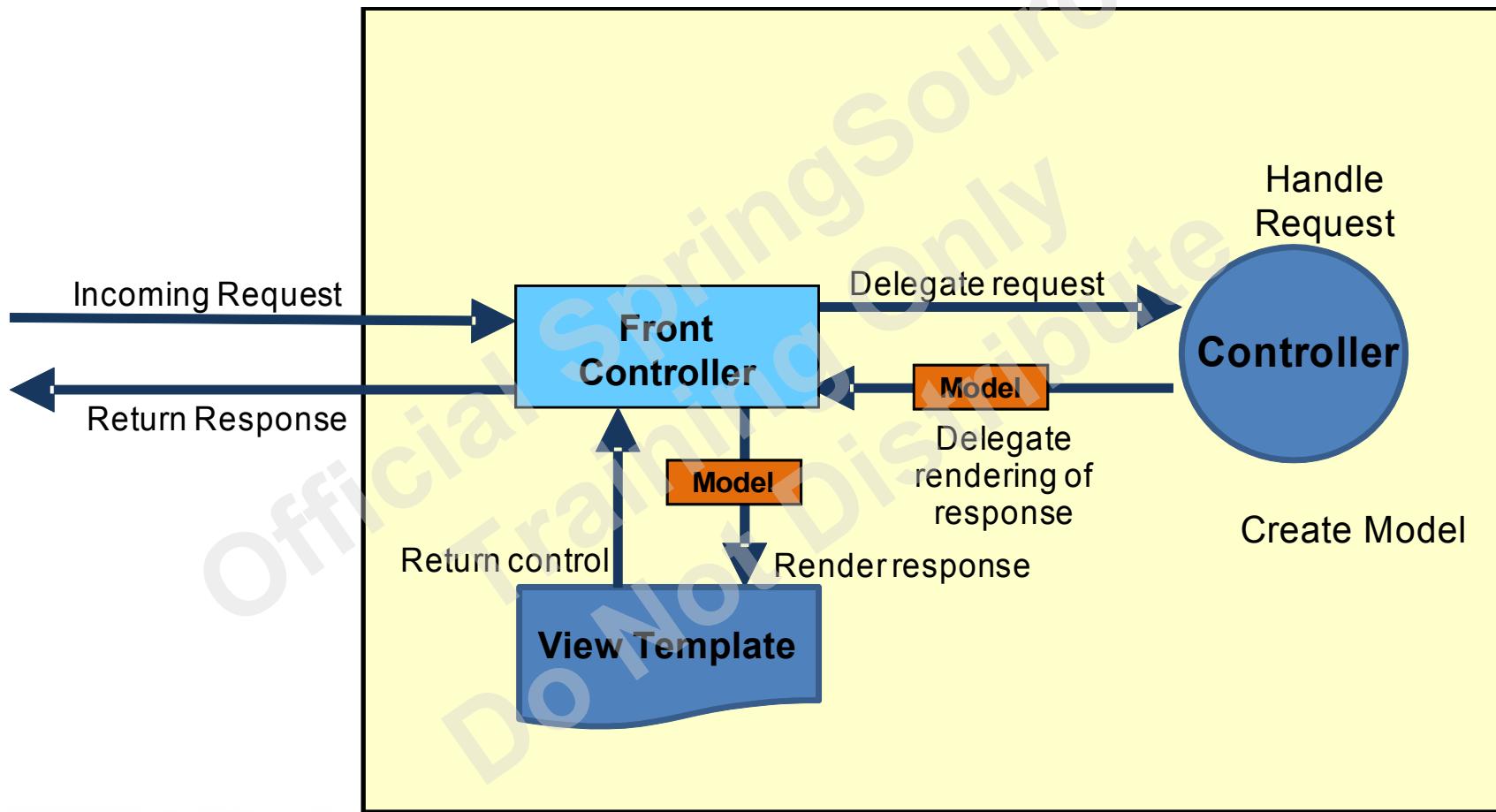
- **Request Processing Lifecycle**
- Key Artifacts
 - DispatcherServlet
 - Handlers
 - Views
- Quick Start

Web Request Handling Overview

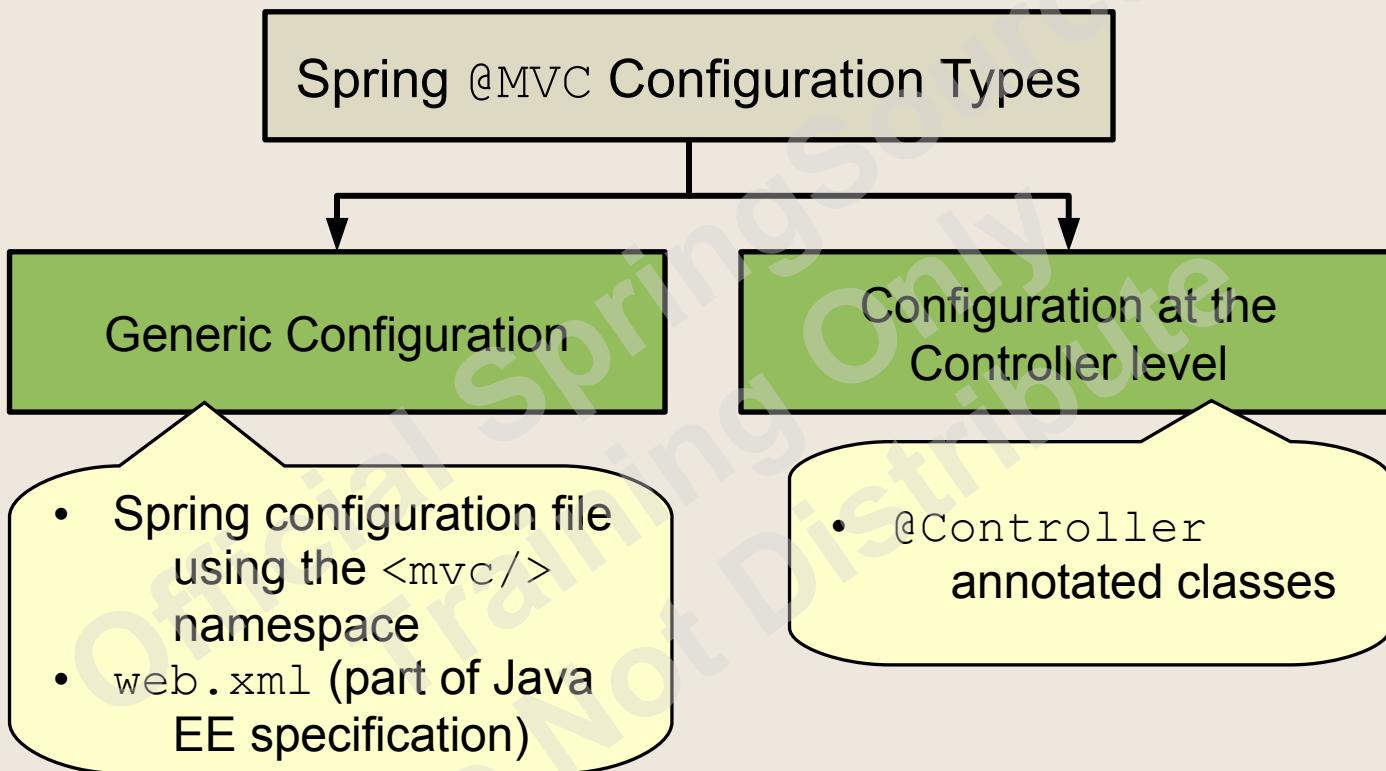


- Web request handling is rather simple
 - Based on an incoming URL...
 - ...we need to call a method...
 - ...after which the return value (if any)...
 - ...needs to be rendered using a view

Request Processing Lifecycle



Spring @MVC Configuration

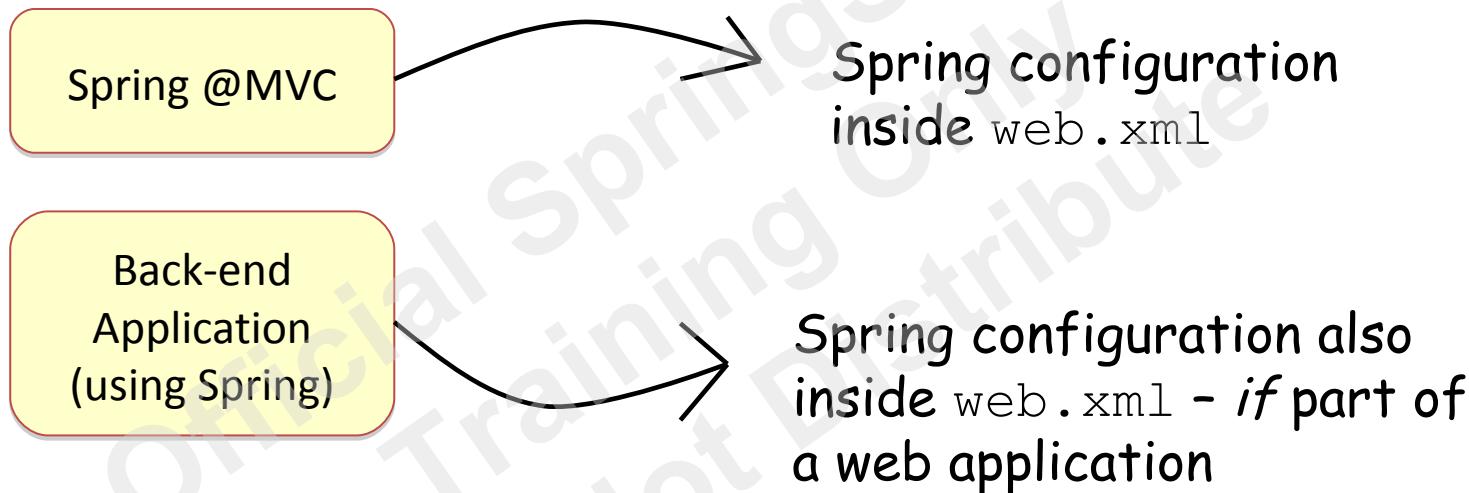


This diagram *only* shows Spring @MVC configuration. Integration with other Spring configuration files will be shown in the *next* slide

Spring in a Web Application



- How does Spring @MVC interact with Spring



- Two options:
 - Separate configurations or All-In-One

1. Separate Configurations

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/app-config.xml</param-value>
</context-param>

<listener> <listener-class>
    org.springframework.web.context.ContextLoaderListener
</listener-class> </listener>

<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/web-config.xml</param-value>
    </init-param>
</servlet>

<servlet-mapping>...</servlet-mapping>
```

Back-end

Spring
@MVC

web.xml

2. All-In-One Alternative

- All Spring configuration is loaded by the MVC DispatcherServlet
 - Works well if the MVC DispatcherServlet is the only entry point to Spring
 - No need for the ContextLoaderListener

DispatcherServlet configuration

- Controllers
- Transactions
- Data access
- ...

2. All-In-One – Example

```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/web-config.xml
            /WEB-INF/spring/application-config.xml
        </param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/main/*</url-pattern>
</servlet-mapping>
```

Spring @MVC

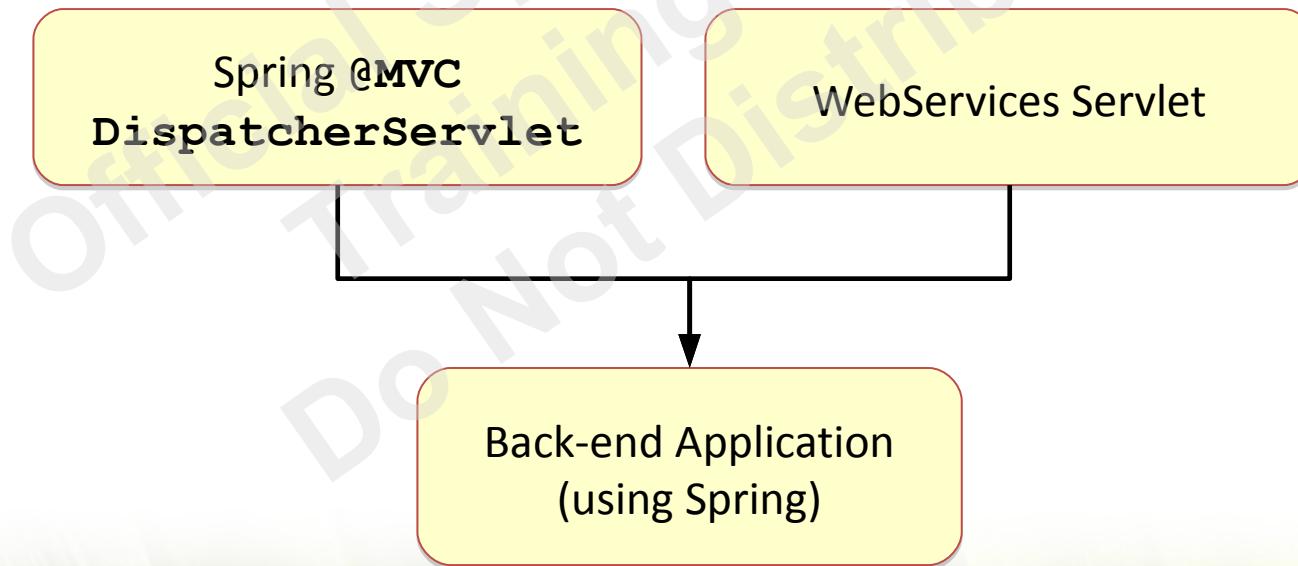
Back-end

web.xml

2. All-In-One – Limitation



- What if Spring MVC is not the only entry-point?
 - For example, you want Web Services as well
 - Use the separate configuration approach (option 1)
 - Root application context *shared* by both servlets



1 or 2: Configuration Options



```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/*-beans.xml
            classpath:com/springsource/application-config.xml
        </param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/main/*</url-pattern>
</servlet-mapping>
```

Wildcards are accepted

Can use the *classpath* prefix

Topics in this Session

- Request Processing Lifecycle
- **Key Artifacts**
 - DispatcherServlet
 - Handlers
 - Views
- Quick Start

DispatcherServlet: The Heart of Spring Web MVC



- A “front controller”
 - coordinates all request handling activities
 - analogous to Struts ActionServlet / JSF FacesServlet
- Delegates to Web infrastructure beans
- Invokes user Web components
- Fully customizable
 - interfaces for all infrastructure beans
 - many extension points

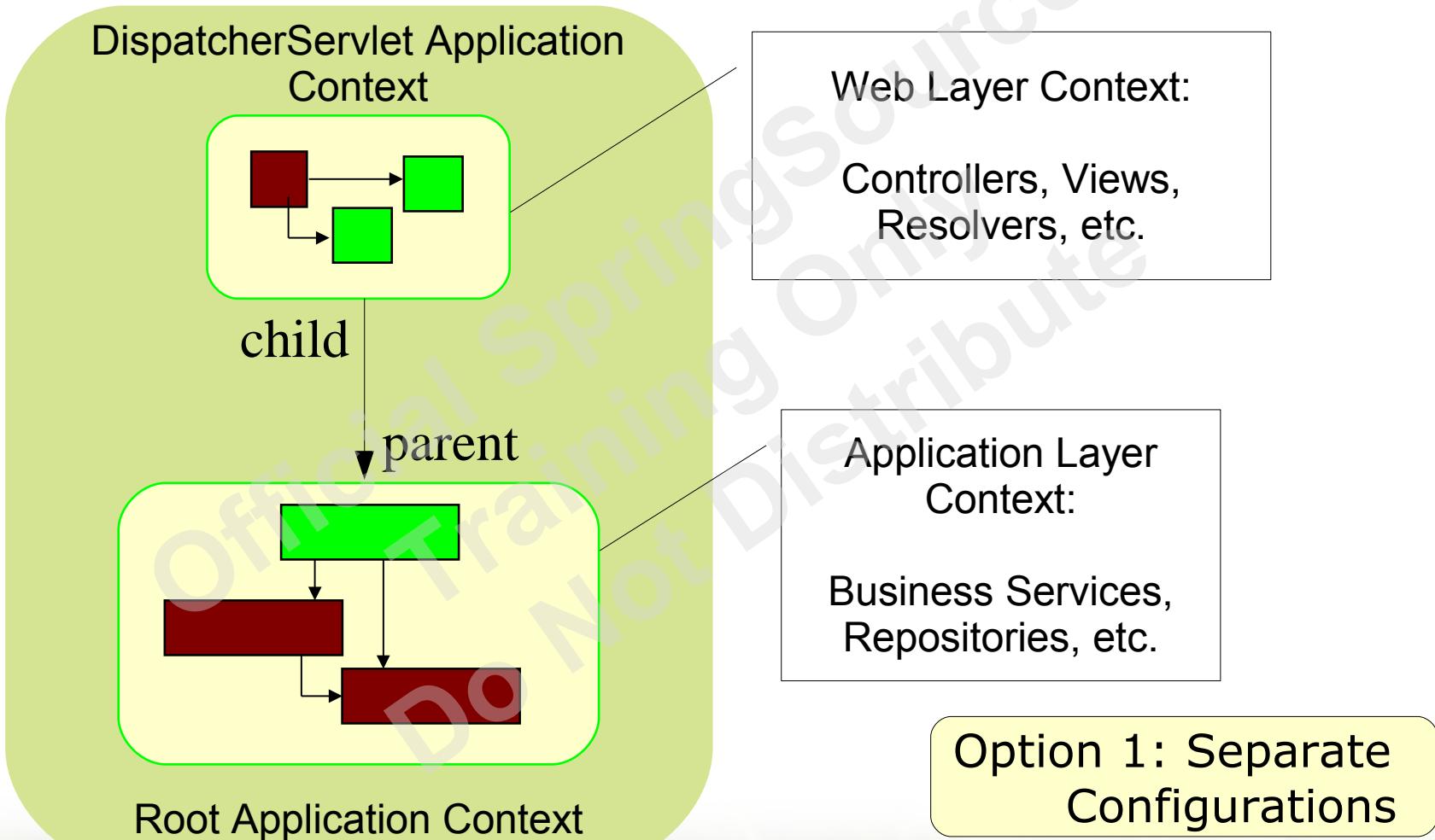
DispatcherServlet Configuration



- Defined in web.xml
- Uses Spring for its configuration
 - programming to interfaces + dependency injection
 - easy to swap parts in and out
- Creates separate “servlet” application context
 - configuration is private to DispatcherServlet
- Full access to the parent “root” context
 - instantiated via ContextLoaderListener
 - shared across servlets

Option 1

Servlet Container After Starting Up

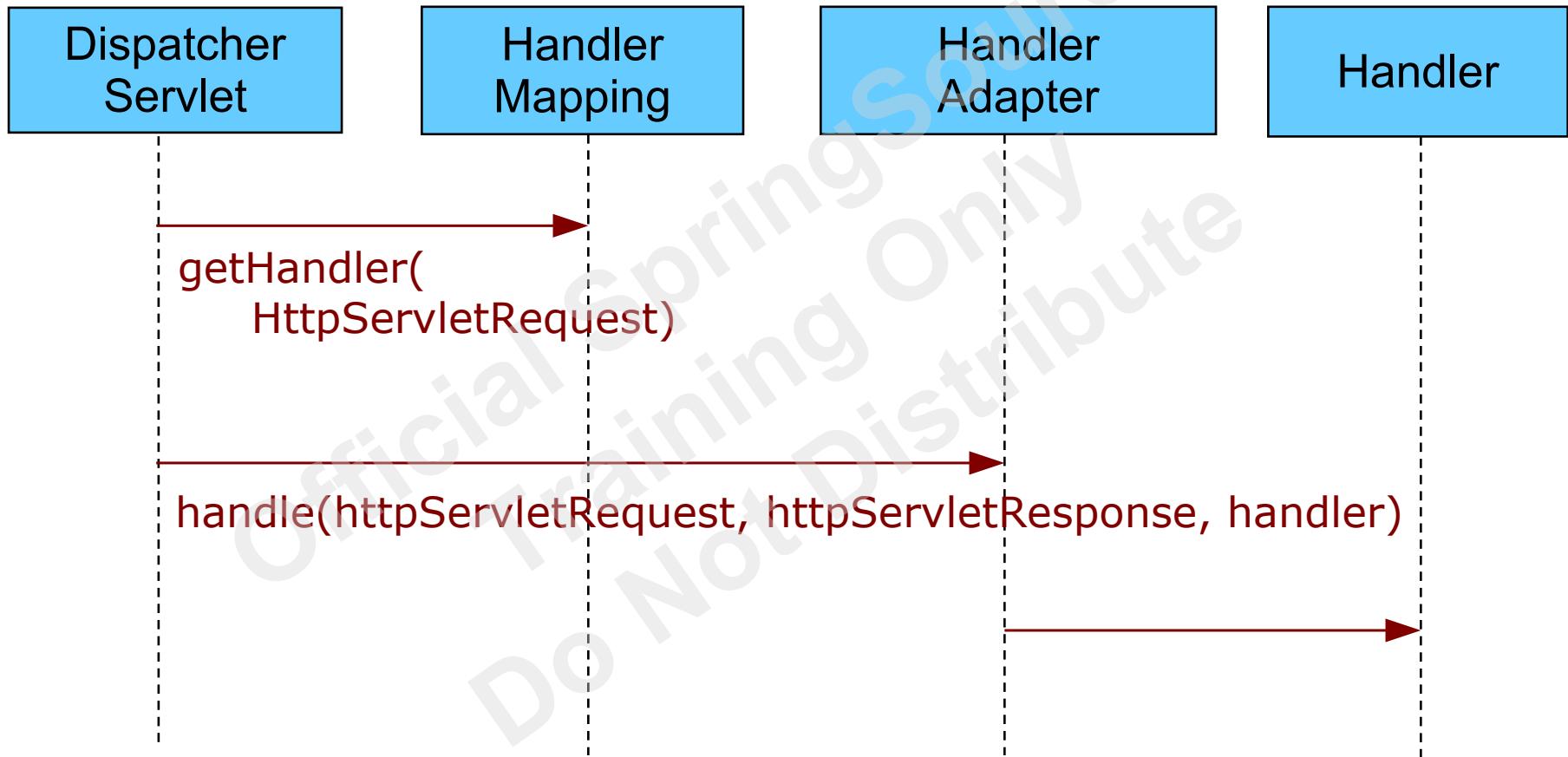


Topics in this Session



- Request Processing Lifecycle
- Key Artifacts
 - DispatcherServlet
 - **Handlers**
 - Views
- Quick Start

Request Processing



- Since Spring 2.5, annotated MVC is preferred
 - Informally referred to as “Spring @MVC”
 - Use `<mvc:annotation-config/>` to setup default handlers and adapters
- Spring 2.0 used controller base-classes & XML
 - Controller class hierarchy deprecated in Spring 3.0
 - Spring 3 *is* backwards compatible by default
 - do *not* specify `<mvc:annotation-config/>`
- This training focuses on Spring @MVC

Controllers as Request Handlers



- Handlers are typically called controllers and are usually annotated by `@Controller`
- `@RequestMapping` tells Spring what method to execute when processing a particular request

```
@Controller  
public class AccountController {
```

```
    @RequestMapping("/listAccounts.htm")  
    public String list(Model model) {...}
```

```
}
```

Example of calling URL:

[http://localhost:8080 / mvc-1 / rewardsadmin / listAccounts.htm](http://localhost:8080/mvc-1/rewardsadmin/listAccounts.htm)

application server

webapp

servlet mapping

request mapping

URL-Based Mapping Rules



- Mapping rules typically URL-based, optionally using wild cards:
 - /login
 - /editAccount
 - /reward/**
- Mapping rules in Spring 2.5+: defined using annotations or XML:
 - @RequestMapping("/login")
- Mapping rules in Spring < 2.5: defined in XML
 - Using the normal <beans/> config language

Handler Method Parameters



- Need context from current request
- Spring MVC will 'fill in' declared parameters
 - HttpServletRequest, HttpSession, Principle ...
 - Model and other Spring MVC classes
 - allows for very flexible method signatures

```
@Controller  
public class AccountController {  
  
    @RequestMapping("/listAccounts")  
    public String list(Model model) {  
        ...  
    }  
}
```

View name

Model holds data for view

Extracting Request Parameters



- Use `@RequestParam` annotation
 - Extracts parameter from the request
 - Performs type conversion

```
@Controller  
public class AccountController {  
  
    @RequestMapping("/showAccount")  
    public String show(@RequestParam("entityId") long id,  
                      Model model) {  
  
        ...  
    }  
}
```

Example of calling URL:

<http://localhost:8080/mvc-1/rewardsadmin/showAccount.htm?entityId=123>

URI Templates

- Spring 3.0 can also extract values from request URLs using so-called *URI Templates*
 - not Spring-specific concept, used in many frameworks
 - Use {...} placeholders and @PathVariable
- Allows clean URLs without request parameters

```
@Controller  
public class AccountController {  
  
    @RequestMapping("/accounts/{accountId}")  
    public String show(@PathVariable("accountId") long id,  
                      Model model) {  
  
        ...  
    }  
}
```

Example of calling URL:

<http://localhost:8080/mvc-1/rewardsadmin/accounts/123>

Topics in this Session



- Request Processing Lifecycle
- Key Artifacts
 - DispatcherServlet
 - Handlers
 - **Views**
- Quick Start

Selecting a View

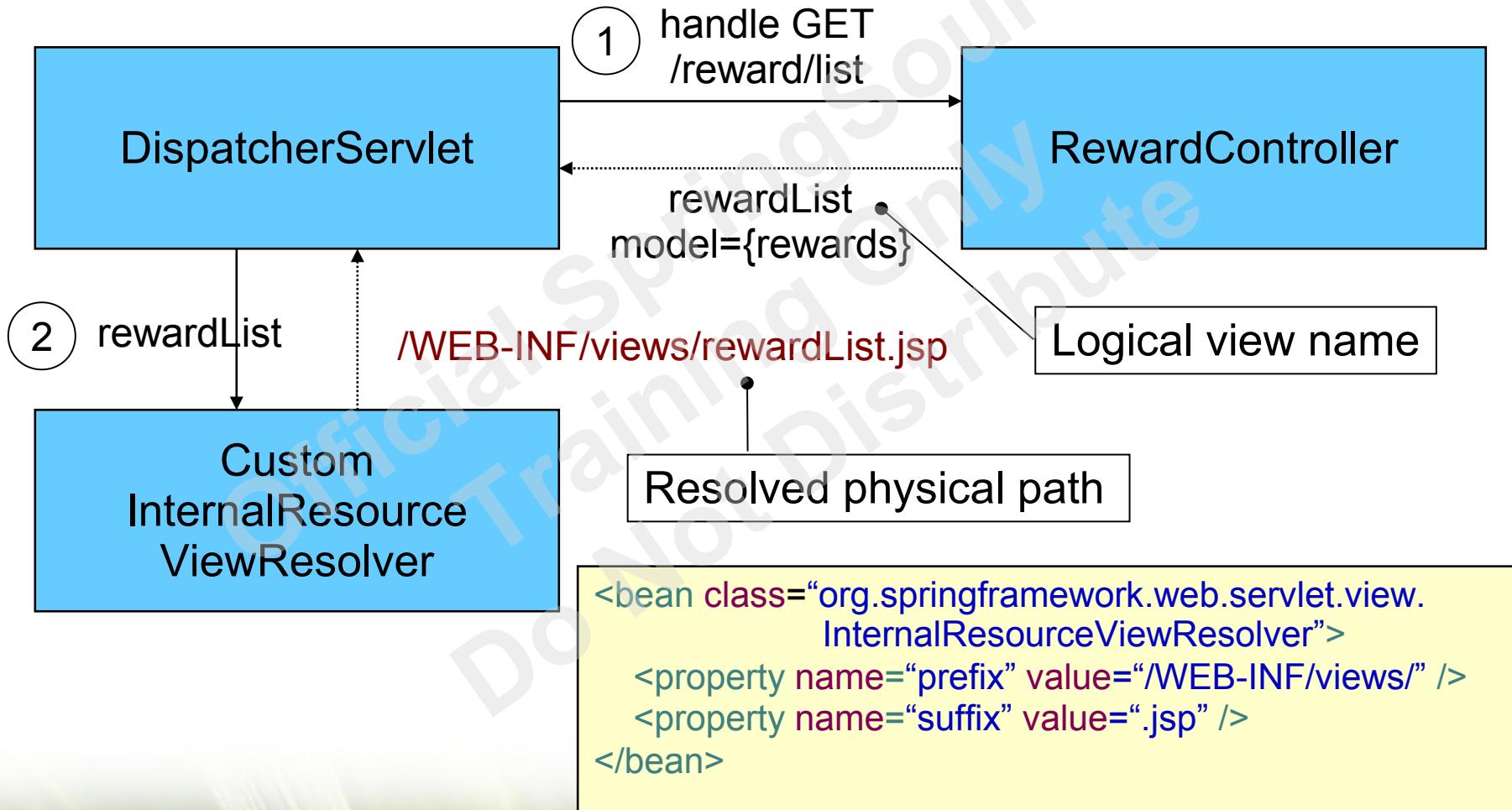
- Controllers typically return a view name
 - By default interpreted as path to JSP page
 - *Example: /WEB-INF/reward/list.jsp*
- Controllers may return **null** (or **void**)
 - Default view then selected based on the request URL
- Controllers may also return a concrete View
 - `new JstlView("/WEB-INF/reward/list.jsp")`
 - `new RewardListingPdf()`

Not common – Spring does this anyway!

View Resolvers

- The DispatcherServlet delegates to a ViewResolver to map returned view names to View implementations
- The default ViewResolver treats the view name as a Web Application-relative file path
- Override this default by registering a ViewResolver bean with the DispatcherServlet
 - Internal resource (default)
 - Bean name

Custom Internal Resource View Resolver Example



Topics in this Session



- Request Processing Lifecycle
- Key Artifacts
 - DispatcherServlet
 - Handlers
 - Views
- **Quick Start**

Quick Start

Steps to developing a Spring MVC application

1. Deploy a Dispatcher Servlet (one-time only)
2. Implement a request handler (controller)
3. Implement the View(s)
4. Register the Controller with the DispatcherServlet
5. Deploy and test

Repeat steps 2-5 to develop new functionality

1.a Deploy DispatcherServlet



```
<servlet>
  <servlet-name>rewardsadmin</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>

  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/mvc-config.xml
    </param-value>
  </init-param>

  <init-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>dev</param-value>
  </init-param>

  <load-on-startup>1</load-on-startup>
</servlet>
```

web.xml

Contains Spring MVC configuration

Optional profile – Spring 3.1

Default is on *first* request

1.b Deploy Dispatcher Servlet



- Map the Servlet to a URL pattern

```
<servlet-mapping>
    <servlet-name>rewardsadmin</servlet-name>
    <url-pattern>/rewardsadmin/*</url-pattern>
</servlet-mapping>
```

- Will now be able to invoke the Servlet like

```
http://localhost:8080/mvc-1/rewardsadmin/reward/list
http://localhost:8080/mvc-1/rewardsadmin/reward/new
http://localhost:8080/mvc-1/rewardsadmin/reward/show?id=1
```

Initial DispatcherServlet Configuration



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

/WEB-INF/mvc-config.xml

2. Implement the Controller



```
@Controller  
public class RewardController {  
    private RewardLookupService lookupService;  
  
    public RewardController(RewardLookupService svc) {  
        this.lookupService = svc;  
    }  
  
    @RequestMapping("/reward/show")  
    public String show(@RequestParam("id") long id, Model model) {  
        Reward reward = lookupService.lookupReward(id);  
        model.addAttribute("reward", reward);  
        return "rewardView";  
    }  
}
```

Depends on application service

Selects the "rewardView" to render the reward

Automatically filled in by Spring

3. Implement the View

```
<html>
  <head><title>Your Reward</title></head>
  <body>
    Amount=${reward.amount} <br/>
    Date=${reward.date} <br/>
    Account Number=${reward.account} <br/>
    Merchant Number=${reward.merchant}
  </body>
</html>
```

↑
References result model object by name

/WEB-INF/views/rewardView.jsp

4. Register the Controller

```
<beans>

    <bean class="org.springframework.web...InternalResourceViewResolver">
        ...
    </bean>

    <bean id="rewardController" class="rewardsadmin.RewardController">
        <constructor-arg ref="rewardLookupService" />
    </bean>

</beans>
```

/WEB-INF/mvc-config.xml

5. Deploy and Test

`http://localhost:8080/rewardsadmin/reward/show?id=1`

Your Reward

Amount = \$100.00

Date = 2006/12/29

Account Number = 123456789

Merchant Number = 1234567890

Optionally: Enable Component Scanning



Spring MVC Controllers can be auto-detected

```
<!-- only scan for controllers, not services, repositories etc. that belong in root ctx -->
<context:component-scan base-package="rewardsadmin" use-default-filters="false">
    <context:include-filter type="annotation"
        expression="org.springframework.stereotype.Controller" />
</context:component-scan>
```

```
@Controller
public class RewardController {
    private RewardLookupService lookupService;

    @Autowired
    public RewardController(RewardLookupService svc) {
        this.lookupService = svc;
    }

    @RequestMapping("/reward/show")
    public String show(@RequestParam("id") long id, Model model) { ... }
}
```

MVC Additions in Spring 3.0



- @MVC and legacy Controllers enabled by default
 - Appropriate Handler Mapping and Adapters registered out-of-the-box
- Spring 3.0 introduces new features *not* enabled by default
 - Stateless converter framework for binding & formatting
 - Support for JSR-303 declarative validation
 - HttpMessageConverters (for RESTful web services)
- *How do you use these features?*

MVC Namespace

- Use MVC namespace to enable these:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="...>

    <!-- Provides default conversion service, validator and message converters -->
    <mvc:annotation-driven/>
```

- Registers Handler Mapping/Adapter for @MVC only
 - You lose legacy default mappings and adapters!
 - Enables custom conversion service and validators
 - mvc namespace provides many useful features
 - Beyond scope of *this* course

Spring-Web – 4 day course on Spring Web Modules

Spring 3.1

- Same functionality for Java configuration

```
@Configuration  
@EnableWebMvc  
public class RewardConfig extends WebMvcConfigurerAdapter {  
  
    @Bean  
    public rewardController(RewardLookupService svc) {  
        return new RewardController(svc);  
    }  
  
    @Override  
    public void addFormatters(FormatterRegistry registry) {  
        // Register your own type converters and formatters...  
        // For using non-simple types as @RequestParams and and in forms  
        // example: convert “8.25%” to/from instances of a Percentage class  
    }  
}
```

Spring 3.1 also supports Servlet 3 – no web.xml



LAB

Spring Web MVC Essentials



Web Application Security with Spring

Addressing Common Web Application
Security Requirements

Topics in this Session



- **High-Level Security Overview**
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Security Concepts



- Principal
 - User, device or system that performs an action
- Authentication
 - Establishing that a principal's credentials are valid
- Authorization
 - Deciding if a principal is allowed to perform an action
- Secured item
 - Resource that is being secured

Authentication



- There are many authentication mechanisms
 - e.g. basic, digest, form, X.509
- There are many storage options for credential and authority information
 - e.g. Database, LDAP, in-memory (development)

Authorization



- Authorization depends on authentication
 - Before deciding if a user can perform an action, user identity must be established
- The decision process is often based on roles
 - ADMIN can cancel orders
 - MEMBER can place orders
 - GUEST can browse the catalog

Topics in this Session



- High-Level Security Overview
- **Motivations of Spring Security**
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Motivations: Portability



- Servlet-Spec security is not portable
 - Requires container specific adapters and role mappings
- Spring Security is portable across containers
 - Secured archive (e.g. WAR) can be deployed as-is
 - Also runs in standalone environments

Motivations: Flexibility



- Supports all common authentication mechanisms
 - Basic, Form, X.509, Cookies, Single-Sign-On, etc.
- Provides configurable storage options for user details (credentials and authorities)
 - RDBMS, LDAP, Properties file, custom DAOs, etc.
- Uses Spring for configuration

Motivations: Extensibility



- Security requirements often require customization
- With Spring Security, all of the following are extensible
 - How a principal is defined
 - Where authentication information is stored
 - How authorization decisions are made
 - Where security constraints are stored

Motivations: Separation of Concerns



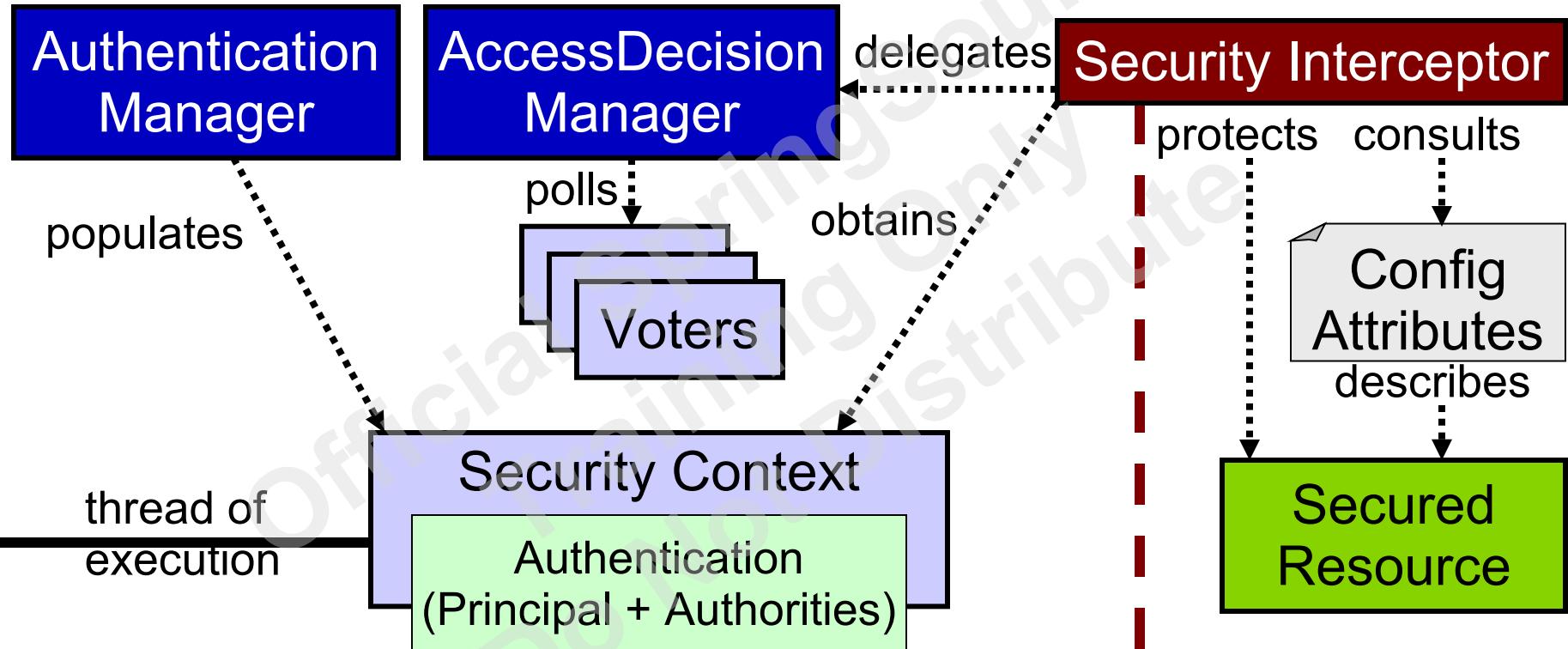
- Business logic is decoupled from security concerns
 - Leverages Servlet Filters and Spring AOP for an interceptor-based approach
- Authentication and Authorization are decoupled
 - Changes to the authentication process have no impact on authorization

Motivations: Consistency



- The goal of authentication is always the same regardless of the mechanism
 - Establish a security context with the authenticated principal's information
- The process of authorization is always the same regardless of resource type
 - Consult the attributes of the secured resource
 - Obtain principal information from security context
 - Grant or deny access

Spring Security: the Big Picture



Topics in this Session



- High-Level Security Overview
- Motivations of Spring Security
- **Spring Security in a Web Environment**
 - **Authorization by URL**
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Configuration in the Application Context



- Spring configuration
- Using Spring Security's "Security" namespace

```
<beans>
  <security:http>

    <security:intercept-url pattern="/accounts/**"
        access="IS_AUTHENTICATED_FULLY" />

    <security:form-login login-page="/login.htm"/>

    <security:logout logout-success-url="/index.html"/>

  </security:http>
</beans>
```

Match all URLs starting with /accounts/ (ANT-style path)

Spring configuration file

Configuration in web.xml



- Define the single proxy filter
 - springSecurityFilterChain is a mandatory name
 - Refers to an existing Spring bean with same name

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

web.xml

intercept-url

- intercept-urls are evaluated in the order listed
 - the first match will be used
 - specific matches should be put on top

```
<beans>
  <security:http>

    <security:intercept-url pattern="/accounts/edit"
      access="ROLE_ADMIN" />
    <security:intercept-url pattern="/accounts/account*"
      access="ROLE_ADMIN,ROLE_USER" />
    <security:intercept-url pattern="/accounts/**"
      access="IS_AUTHENTICATED_FULLY" />
    <security:intercept-url pattern="/customers/**"
      access="IS_AUTHENTICATED_ANONYMOUSLY" />

  </security:http>
</beans>
```



Syntax available from Spring Security 2.0

Security EL expressions



- `hasRole('role')`
 - Checks whether the principal has the given role
- `hasAnyRole('role1', 'role2', ...)`
 - Checks whether the principal has any of the given roles
- `isAnonymous()`
 - Allows access for unauthenticated principals
- `isAuthenticated()`
 - Allows access for authenticated or remembered principals



Available from Spring Security 3.0
Previous syntax still works in Spring Security 3.0

Intercept-url and Expression Language



- Expression Language provides more flexibility
 - Many built-in expressions available

Expression Language needs
to be enabled explicitly

```
<beans>
  <security:http use-expressions="true">

    <security:intercept-url pattern="/accounts/edit*"
      access="hasRole('ROLE_ADMIN')"/>
    <security:intercept-url pattern="/accounts/account*"
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_USER')"/>
    <security:intercept-url pattern="/accounts/**"
      access="isAuthenticated() and hasIpAddress('192.168.1.0/24')"/>

  </security:http>
</beans>
```

Spring configuration file

Working with roles

- Checking if the user has one single role

```
<security:intercept-url pattern="/accounts/update*" access="hasRole('ROLE_ADMIN')"/>
```

- "or" clause

```
<security:intercept-url pattern="/accounts/update**"  
access="hasAnyRole('ROLE_ADMIN', 'ROLE_MANAGER')"/>
```

- "and" clause

```
<security:intercept-url pattern="/accounts/update**"  
access="hasRole('ROLE_ADMIN') and hasRole('ROLE_MANAGER')"/>
```

- Previous and new syntax can't be mixed

```
<security:intercept-url pattern="/accounts/update**"  
access="hasRole('ROLE_MANAGER')"/>  
<security:intercept-url pattern="/accounts/update**" access="ROLE_ADMIN"/>
```

Not correct!!

Specifying login and logout



```
<beans ...>
  <security:http pattern="/accounts/login" security="none"/>

  <security:http use-expressions="true">
    <security:form-login login-page="/accounts/login"
      default-target-url="/accounts.home"/>

    <security:intercept-url pattern="/accounts/update*"
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_MANAGER')"/>

    <security:intercept-url pattern="/accounts/**"
      access="hasRole('ROLE_ADMIN')"/>

    <security:logout logout-success-url="/home.html"/>
  </security:http>
  ...
</beans>
```

Exempt login page
(Spring Security 3.1)

Specify login options

Must be declared explicitly
or no logout possible

Spring configuration file

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- **Configuring Web Authentication**
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Configure Authentication



- DAO Authentication provider is default
 - Expects a *UserDetailsService* implementation to provide credentials and authorities
 - Built-in: In-memory (properties), JDBC (database), LDAP
 - Custom
- Or define your own Authentication provider
 - *Example:* to get pre-authenticated user details when using single sign-on
 - CAS, TAM, SiteMinder ...
 - See online examples

Setting up User Login

- Default auth. provider assumes form-based login
 - This is *web* security after all
 - *Must* specify form-login element
 - A basic form is provided
 - Configure to use your own login-page

```
<security:http>
  <security:form-login/>
...
</security:http>

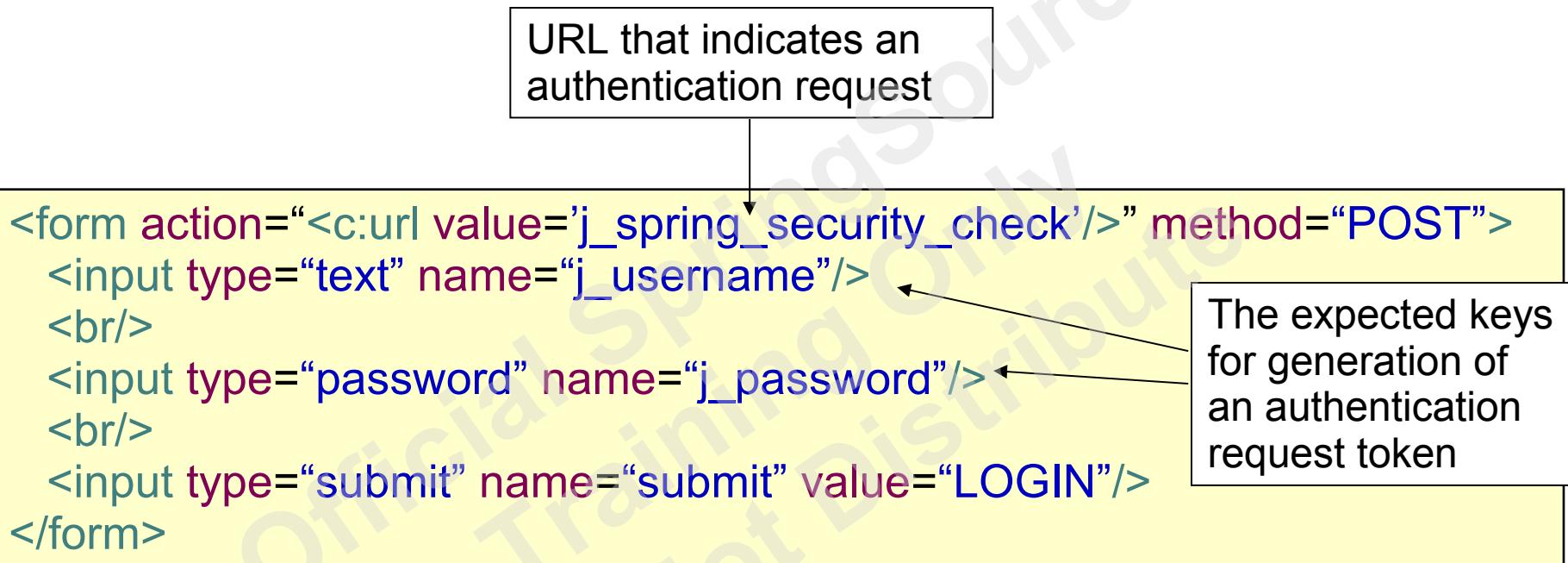
<security:authentication-manager>
  <security:authentication-provider>
    ...
  </security:authentication-provider>
<security:authentication-manager>
```

Login with Username and Password

User:

Password:

An Example Login Page



Above example shows default values (*j_spring_security_check*, *j_username*, *j_password*). All of them can be redefined using `<security:form-login/>`

The In-Memory User Service



- Useful for development and testing
 - Note: must restart system to reload properties

```
<security:authentication-manager>                                Spring configuration file
    <security:authentication-provider>
        <security:user-service properties="/WEB-INF/users.properties" />
    </security:authentication-provider>
<security:authentication-manager>
```

```
admin=secret,ROLE_ADMIN,ROLE_MEMBER,ROLE_GUEST
testuser1=pass,ROLE_MEMBER,ROLE_GUEST
testuser2=pass,ROLE_MEMBER
guest=guest,ROLE_GUEST
```

List of roles separated by commas

login

password

The JDBC user service (1/2)



Queries RDBMS for users and their authorities

- Provides default queries
 - `SELECT username, password, enabled FROM users WHERE username = ?`
 - `SELECT username, authority FROM authorities WHERE username = ?`
- Groups also supported
 - groups, group_members, group_authorities tables
 - See online documentation for details
- Advantage
 - Can modify user info whilst system is running

The JDBC user service (2/2)



- Configuration:

```
<beans>
  <security:http> ... <security:http>

  <security:authentication-manager>
    <security:authentication-provider>
      <security:jdbc-user-service data-source-ref="myDatasource" />
    </security:authentication-provider>
  <security:authentication-manager>
</beans>
```

Spring configuration file

Can customize queries using attributes:
users-by-username-query
authorities-by-username-query
groupAuthorities-by-username-query

Password Encoding

- Can encode passwords using a hash
 - sha, md5, ...

```
<security:authentication-provider>
    <security:password-encoder hash="md5" /> ← simple md5 encoding
    <security:user-service properties="/WEB-INF/users.properties" />
</security:authentication-provider>
```

- Secure passwords using a well-known string
 - Known as a 'salt'
 - Makes brute force attacks against passwords harder

```
<security:authentication-provider>
    <security:password-encoder hash="md5"> ← md5 encoding with salt
        <security:salt-source system-wide="MySalt" />
    </security:password-encoder>
    <security:user-service properties="/WEB-INF/users.properties" />
</security:authentication-provider>
```

Other Authentication Options



- Implement a custom `UserDetailsService`
 - Delegate to an existing User repository or DAO
- LDAP
- X.509 Certificates
- JAAS Login Module
- Single-Sign-On
 - SiteMinder
 - Kerberos
 - JA-SIG Central Authentication Service

Authorization is not affected by changes to Authentication!

Topics in this Session



- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- **Using Spring Security's Tag Libraries**
- Method security
- Advanced security: working with filters

Tag library declaration



- The Spring Security tag library is declared as follows

available since Spring Security 2.0

```
<%@ taglib prefix="security"  
    uri="http://www.springframework.org/security/tags" %>
```

jsp

- Facelet tags for JSF are also available
 - You need to define and install them manually
 - See "*Using the Spring Security Facelets Tag Library*" in the Spring Webflow documentation
 - Principle is available in SpEL: #{{principle.username}}

Spring Security's Tag Library



- Display properties of the Authentication object

You are logged in as:

```
<security:authentication property="principal.username"/>
```

jsp

- Hide sections of output based on role

```
<security:authorize access="hasRole('ROLE_MANAGER')">  
TOP-SECRET INFORMATION  
Click <a href="/admin/deleteAll">HERE</a> to delete all records.  
</security:authorize>
```

jsp

Authorization in JSP based on intercept-url



- Role declaration can be centralized in Spring config files

```
<security:authorize url="/admin/deleteAll">  
TOP-SECRET INFORMATION  
Click <a href="/admin/deleteAll">HERE</a>  
</security:authorize>
```

jsp

URL to protect

```
<security:intercept-url pattern="/admin/*"  
access="hasAnyRole('ROLE_MANAGER', 'ROLE_ADMIN') />
```

Spring configuration file

Pattern that matches the URL to be protected

Matching roles

Topics in this Session



- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- **Method security**
- Advanced security: working with filters

Method Security



- Spring Security uses AOP for security at the method level
 - xml configuration with the Spring Security namespace
 - annotations based on Spring annotations or JSR-250 annotations
- Typically secure your services
 - Do not access repositories directly, bypasses security (and transactions)

Method Security using XML



- Can apply security to multiple beans with only a simple declaration

```
<security:global-method-security>
    <security:protect-pointcut
        expression="execution(* com.springframework..*Service.*(..))"
        access="ROLE_USER,ROLE_MEMBER" />
</security:global-method-security>
```

Spring configuration file



Spring Security 2 syntax only. SpEL not supported here.

- JSR-250 annotations should be enabled

```
<security:global-method-security jsr250-annotations="enabled" />
```

```
import javax.annotation.security.RolesAllowed;  
  
public class ItemManager {  
    @RolesAllowed("ROLE_MEMBER")  
    public Item findItem(long itemNumber) {  
        ...  
    }  
    @RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})  
}
```



Only supports *role-based* security – hence the name

Method Security - @Secured



- Secured annotation should be enabled

```
<security:global-method-security secured-nnotations="enabled" />
```

```
import org.springframework.security.annotation.Secured;
```

```
public class ItemManager {  
    @Secured("IS_AUTHENTICATED_FULLY")  
    public Item findItem(long itemNumber) {  
        ...  
    }  
}
```

```
    @Secured("ROLE_MEMBER")  
    @Secured({"ROLE_MEMBER", "ROLE_USER"})
```



Spring 2.0 syntax, so **not** limited to roles. SpEL **not** supported.

Method Security with SpEL



- Use Pre/Post annotations for SpEL

```
<security:global-method-security pre-post-annotations="enabled" />
```

```
import org.springframework.security.annotation.PreAuthorize;  
  
public class ItemManager {  
    @PreAuthorize("hasRole('ROLE_MEMBER')")  
    public Item findItem(long itemNumber) {  
        ...  
    }  
}
```

Topics in this Session



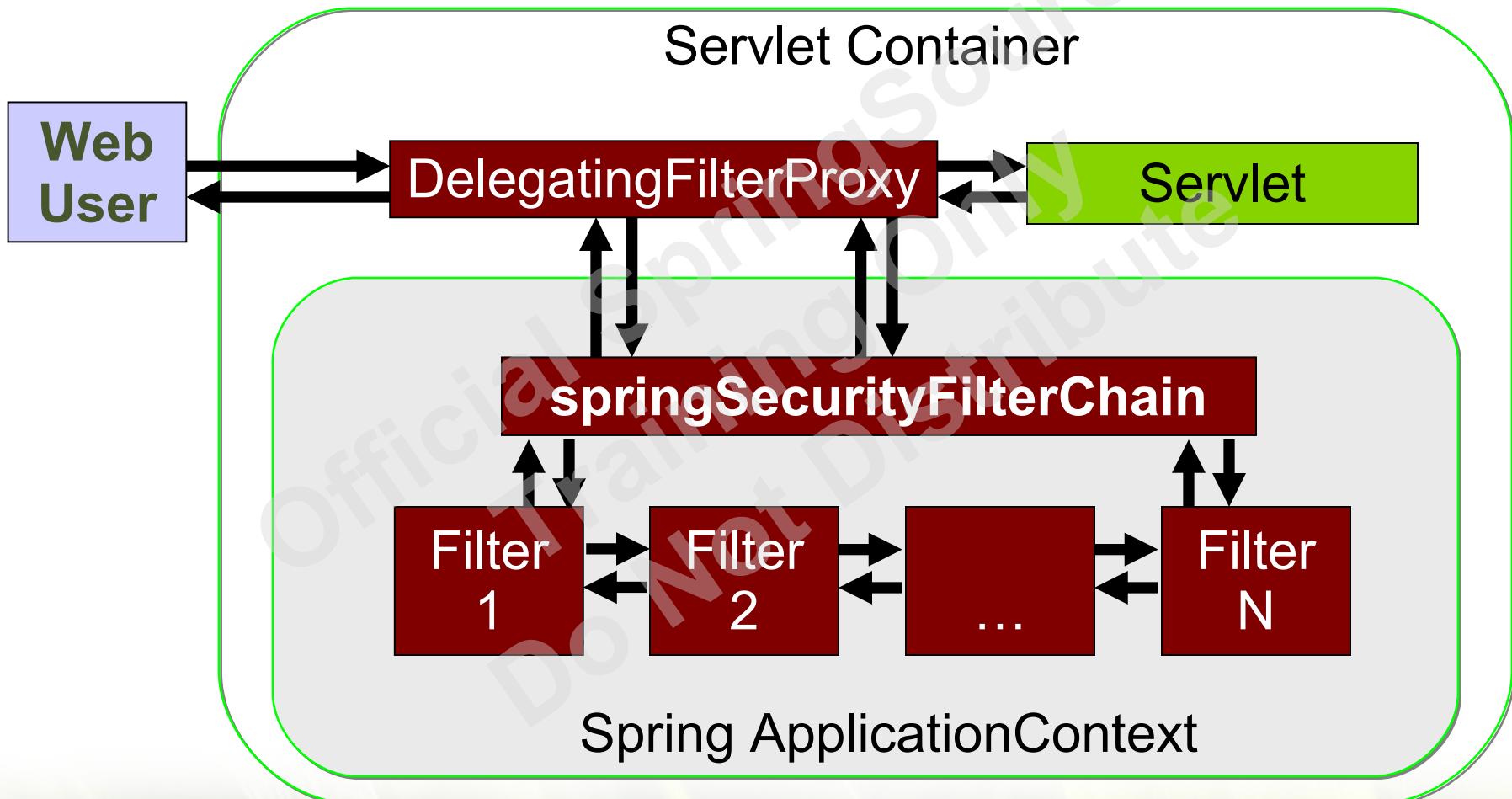
- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- **Advanced security: working with filters**

Spring Security in a Web Environment



- `springSecurityFilterChain` is declared in `web.xml`
- This single proxy filter delegates to a chain of Spring-managed filters
 - Drive authentication
 - Enforce authorization
 - Manage logout
 - Maintain `SecurityContext` in `HttpSession`
 - *and more*

Web Security Filter Configuration

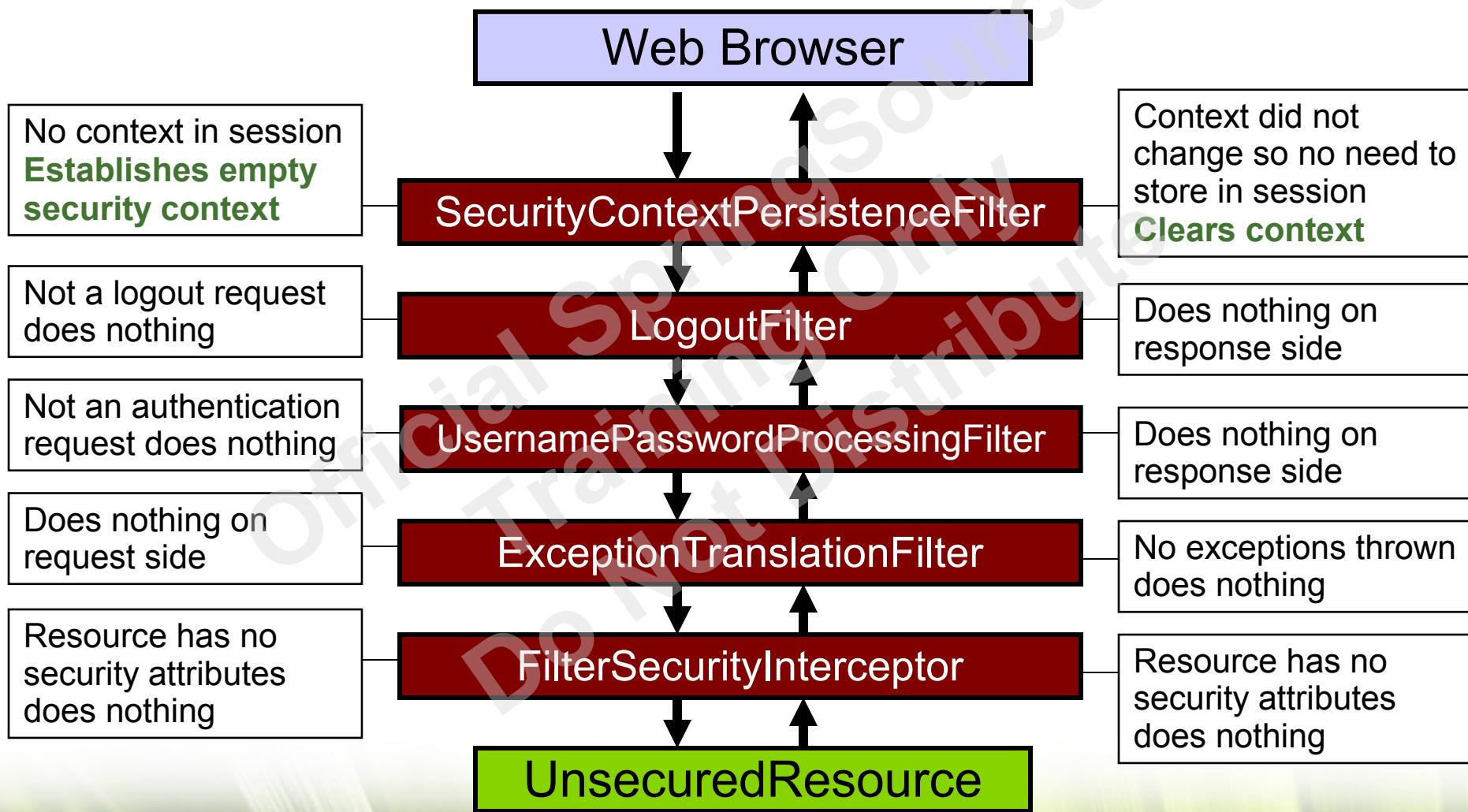


The Filter chain

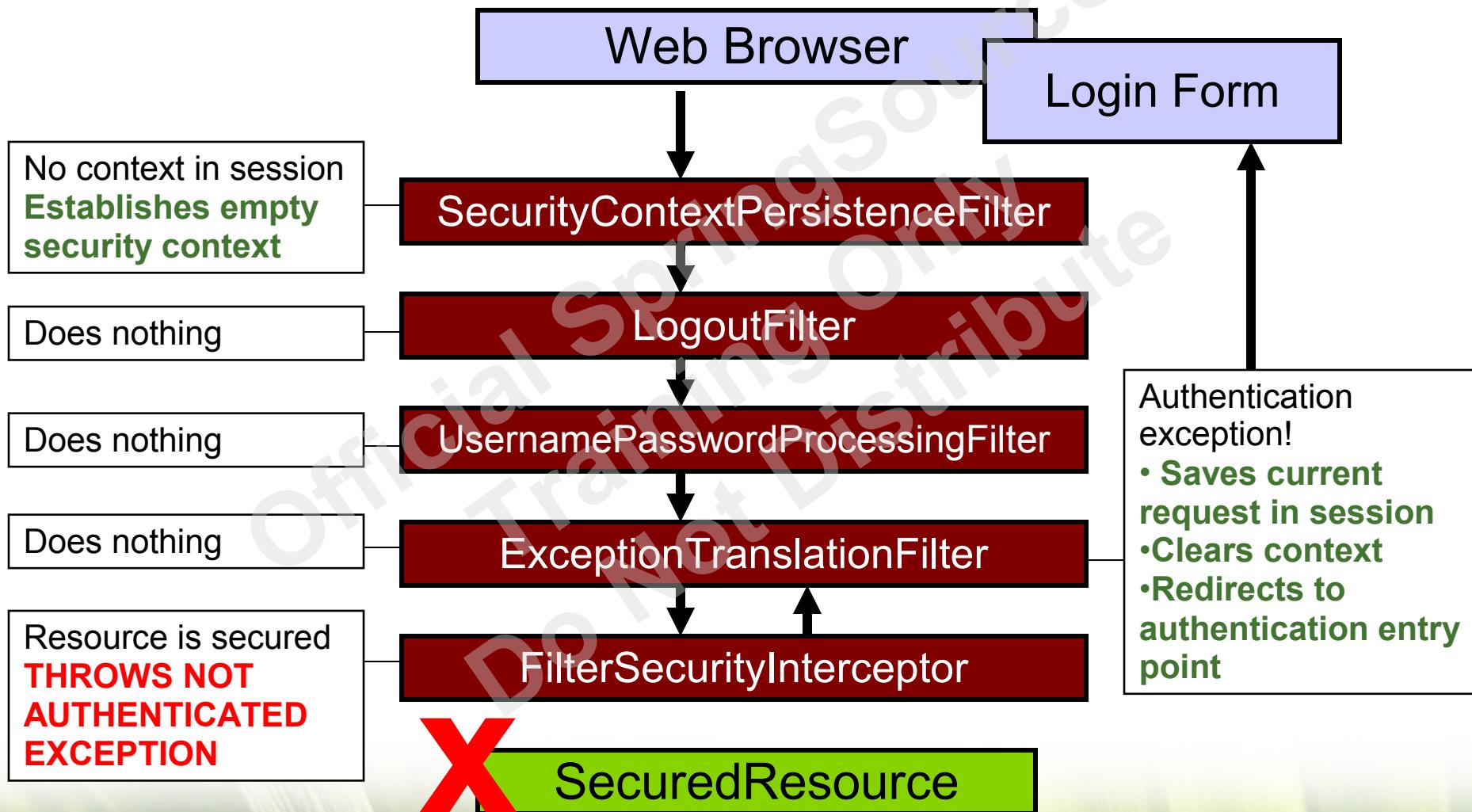


- With ACEGI Security 1.x
 - Filters were manually configured as individual <bean> elements
 - Led to verbose and error-prone XML
- Spring Security 2.x and 3.x
 - Filters are initialized with correct values by default
 - Manual configuration is not required **unless you want to customize Spring Security's behavior**
 - It is still important to understand how they work underneath

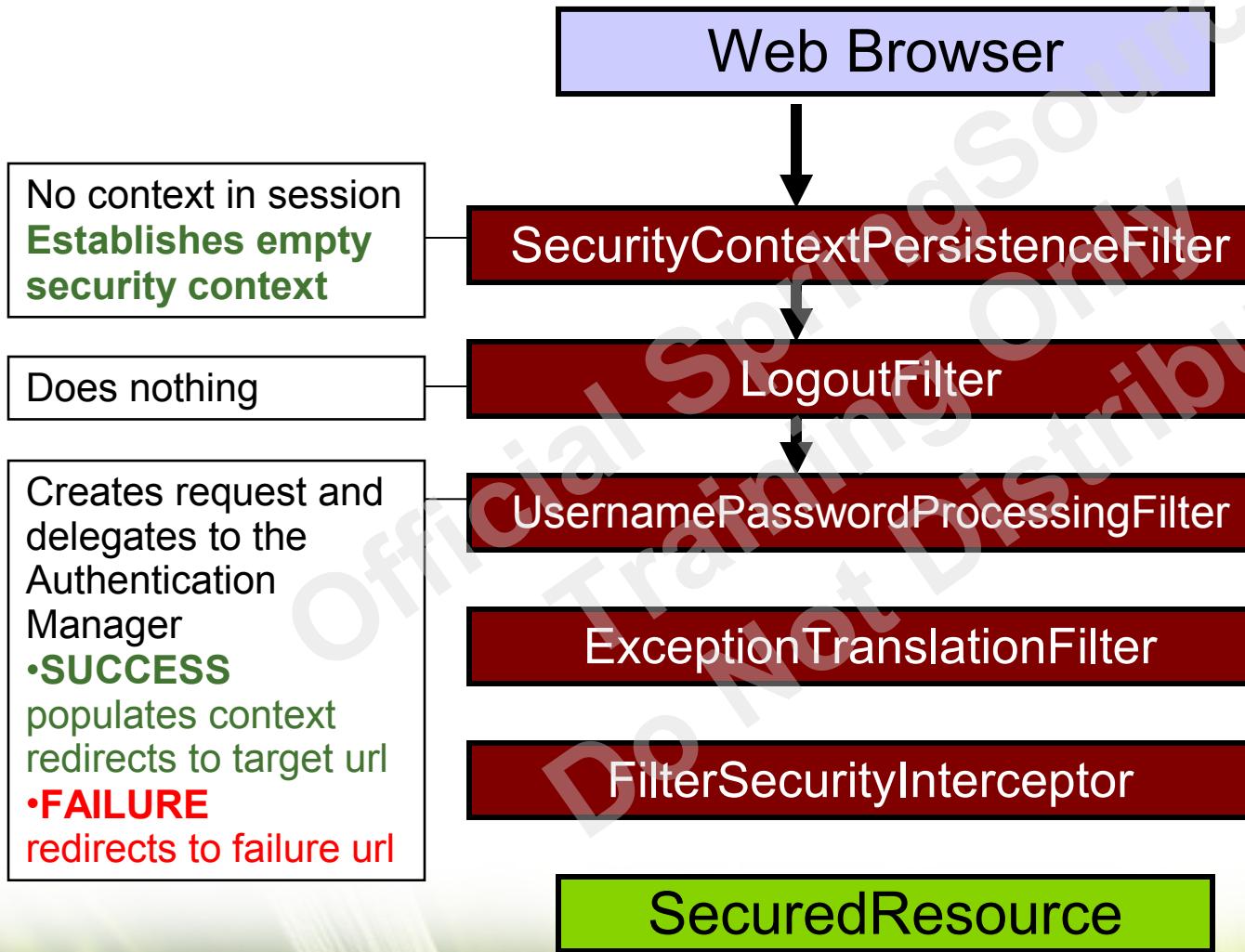
Access Unsecured Resource Prior to Login



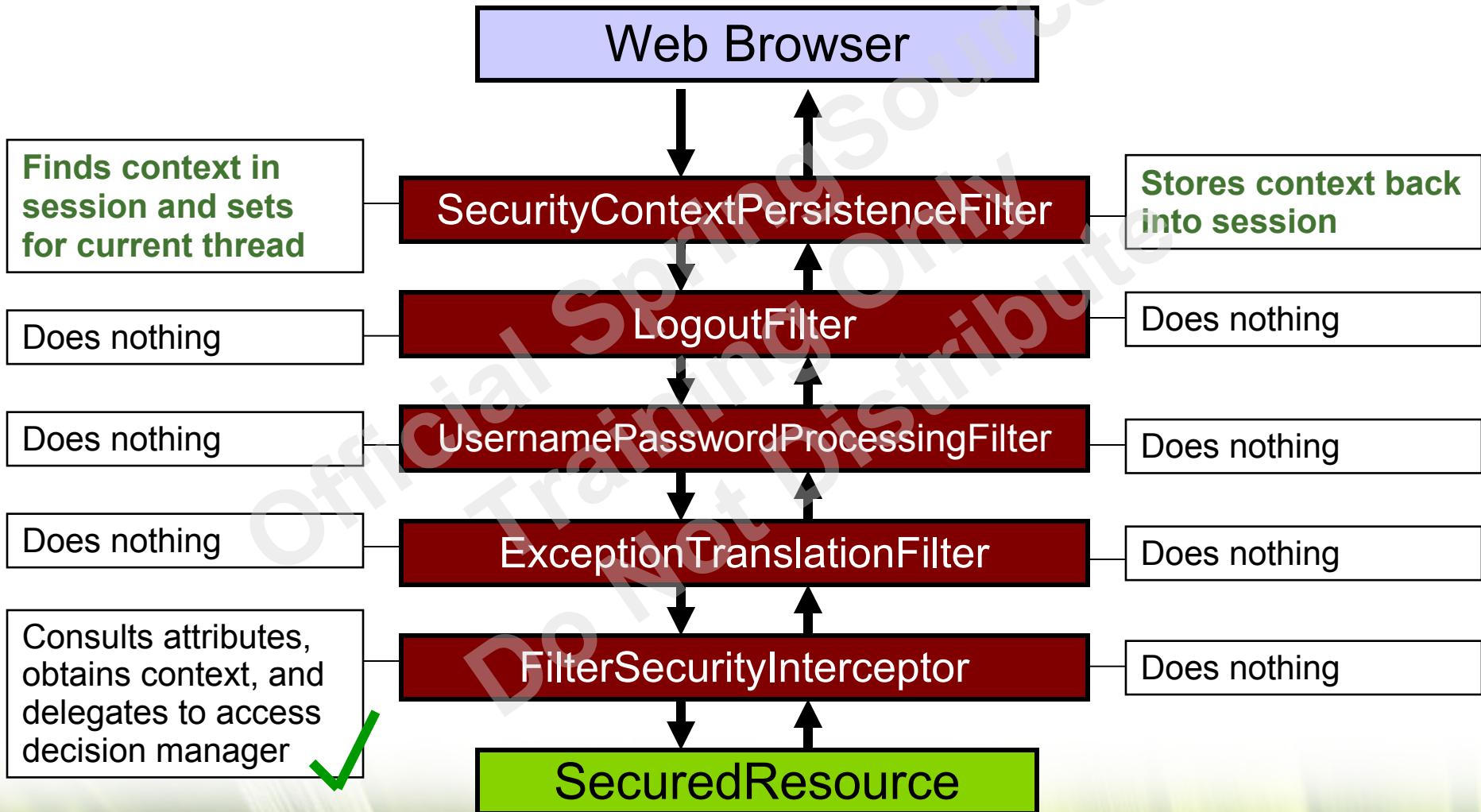
Access Secured Resource Prior to Login



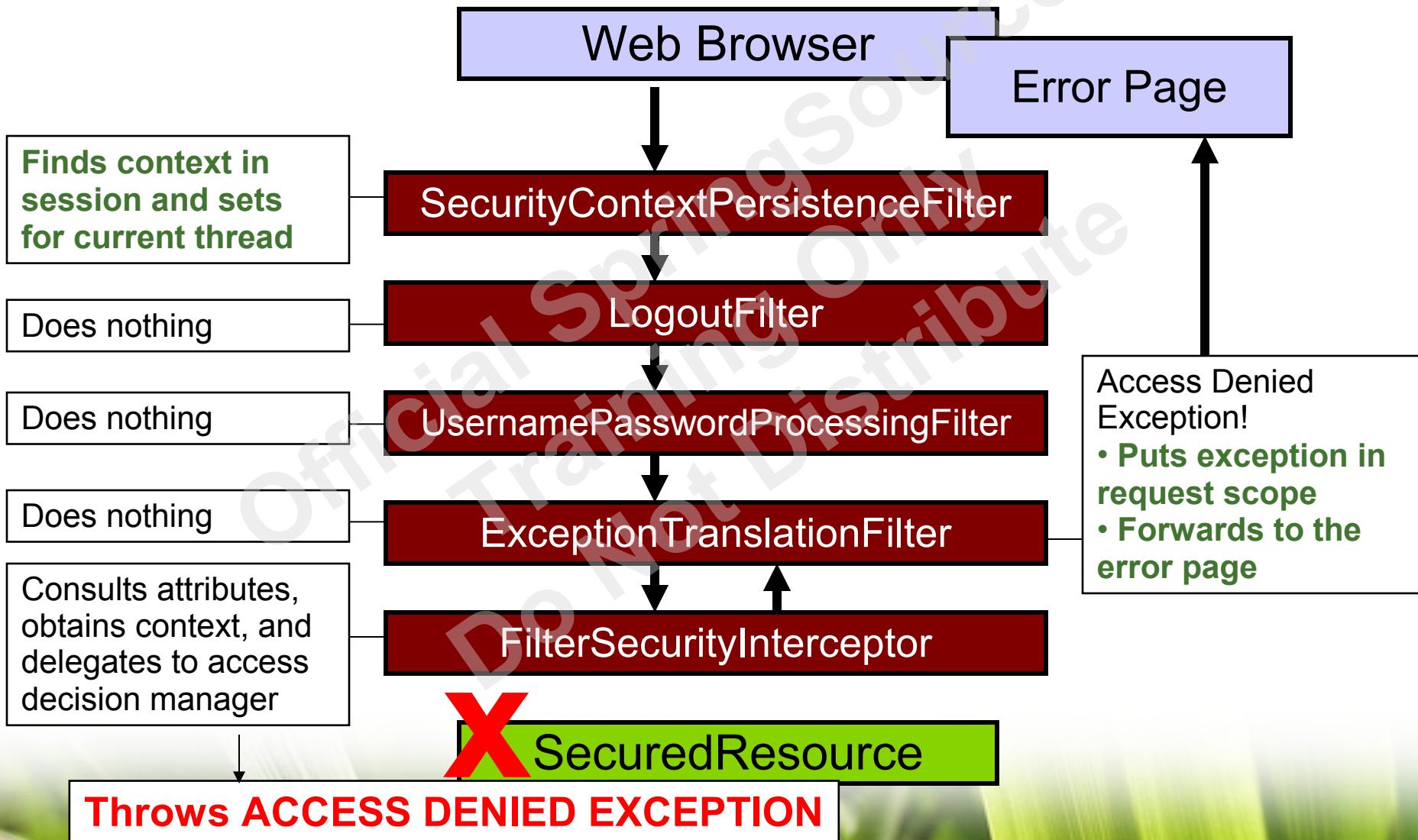
Submit Login Request



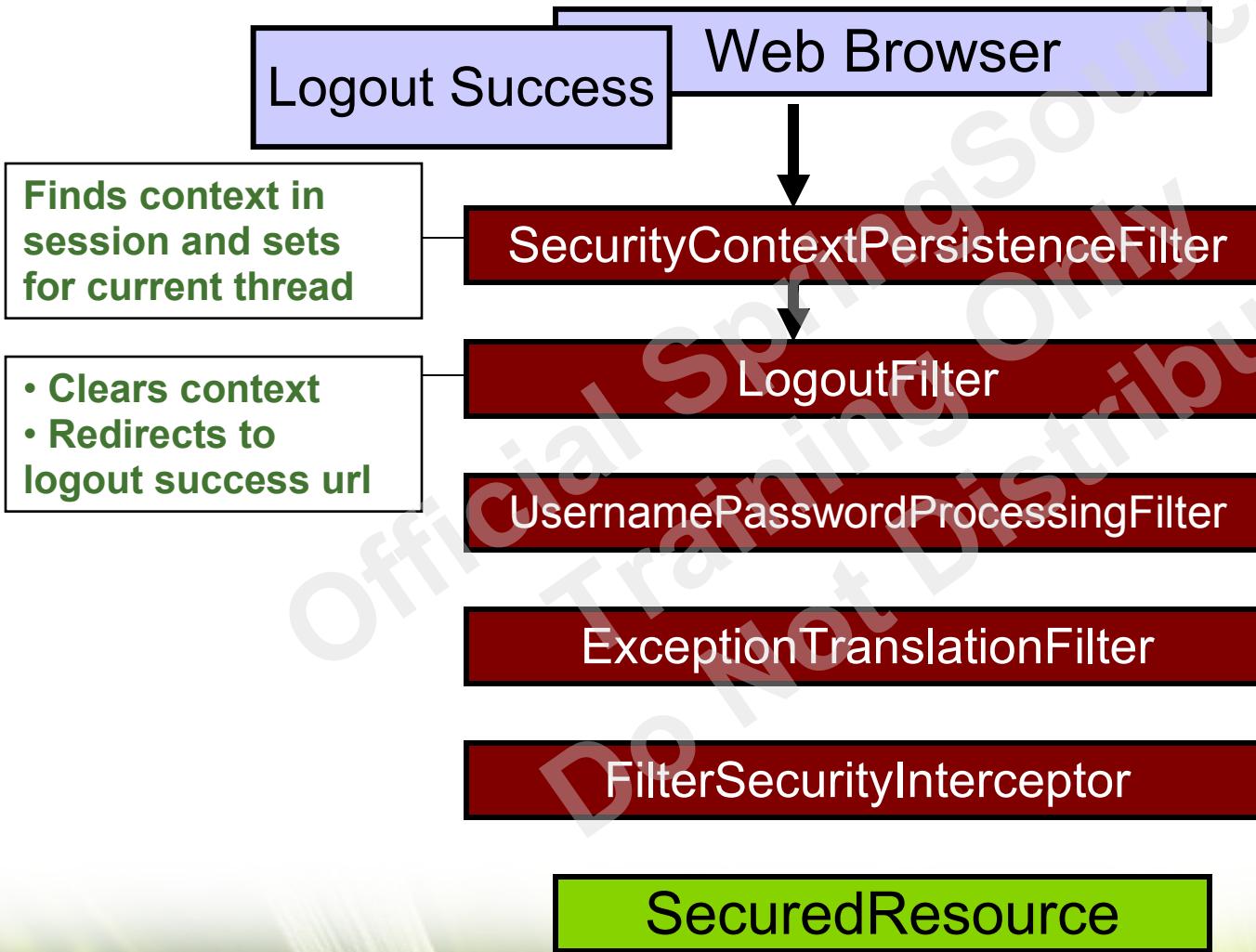
Access Resource With Required Role



Access Resource Without Required Role



Submit Logout Request



The Filter Chain: Summary



#	Filter Name	Main Purpose
1	SecurityContext IntegrationFilter	Establishes SecurityContext and maintains between HTTP requests <i>formerly: HttpSessionContextIntegrationFilter</i>
2	LogoutFilter	Clears SecurityContextHolder when logout requested
3	UsernamePassword Processing Filter	Puts Authentication into the SecurityContext on login request <i>formerly: AuthenticationProcessingFilter</i>
4	Exception TranslationFilter	Converts SpringSecurity exceptions into HTTP response or redirect
5	FilterSecurity Interceptor	Authorizes web requests based on config attributes and authorities

Custom Filter Chain

- One filter on the stack may be **replaced** by a custom filter

```
<security:http>
  <security:custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>

<bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter"/>
```

- One filter can be **added** to the chain

```
<security:http>
  <security:custom-filter after="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>

<bean id="myFilter" class="com.mycompany.MySpecialFilter"/>
```



LAB

Applying Security to a Web Application



Introduction to Spring Remoting

Simplifying Distributed Applications

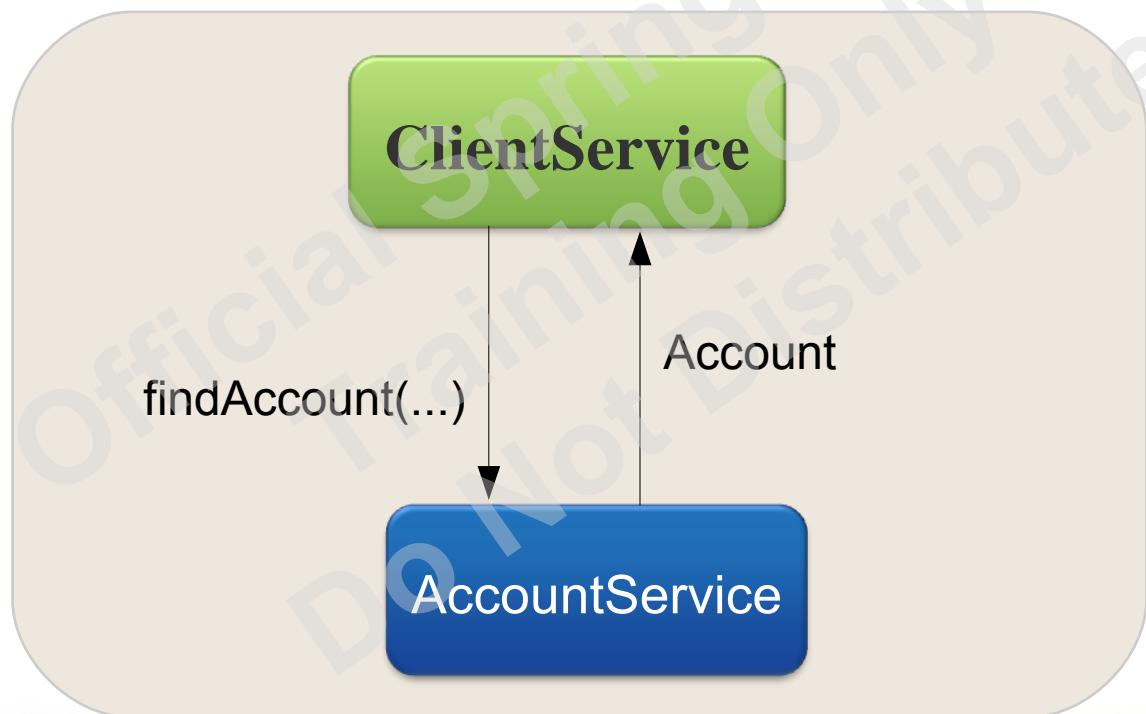
Topics in this Session

- **Introduction to Remoting**
- Spring Remoting Overview
- Spring Remoting and RMI
- HttpInvoker

Official SpringSource
Training Only
Do Not Distribute

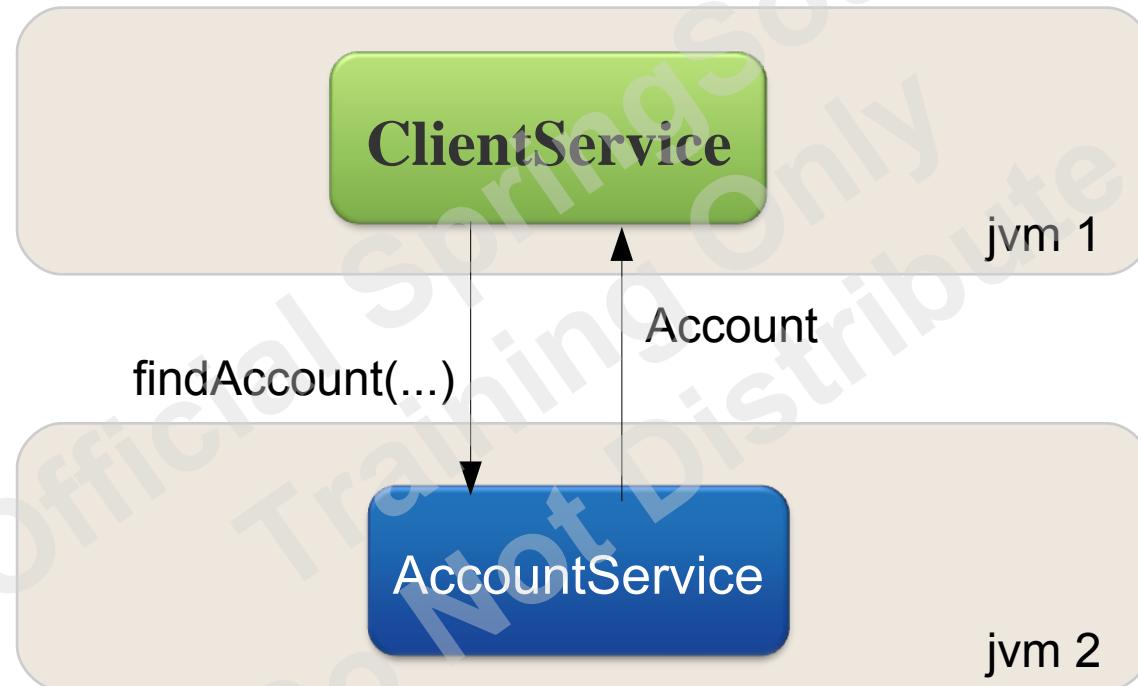
Local access

- So far, you have seen how to access objects locally



Remote access

- What if those 2 objects run in some separate JVMs?



In this module, only Java-to-Java communication is addressed
(as opposed to remote access using Web Services or JMS)

The RMI Protocol



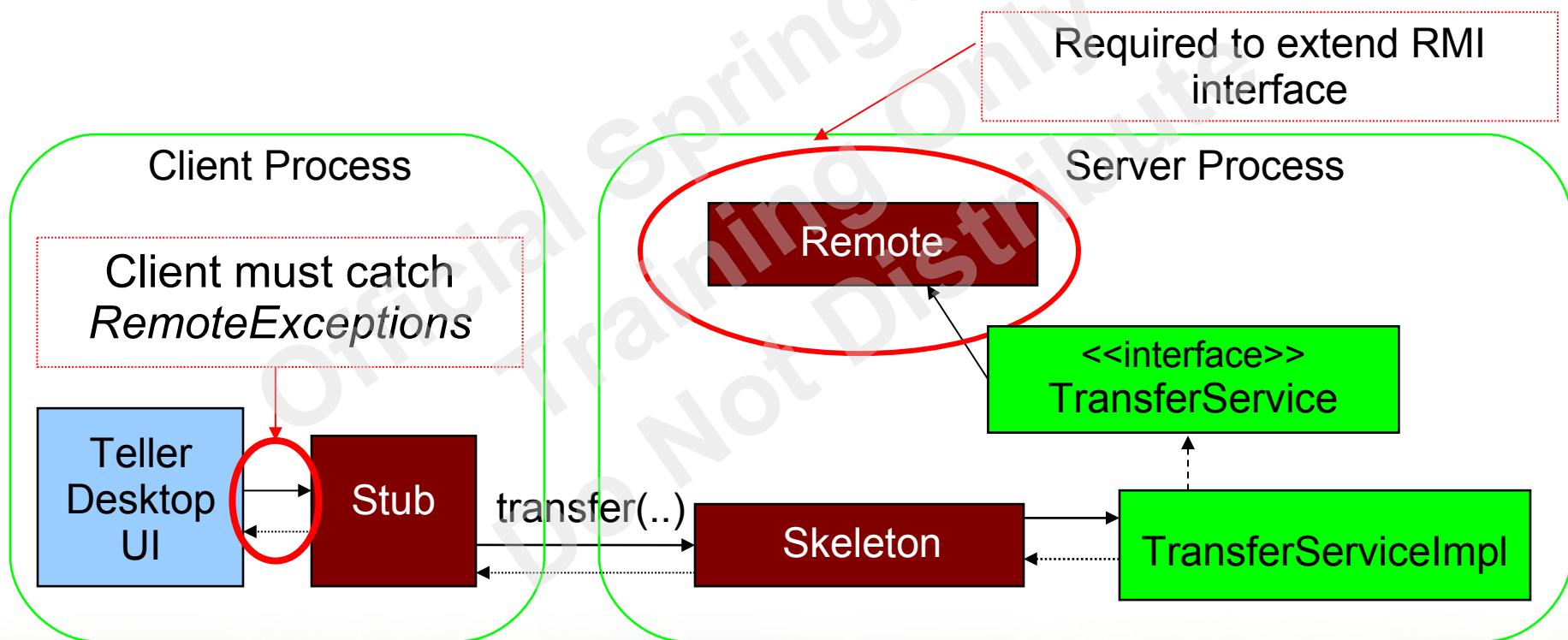
- Standard Java remoting protocol
 - Remote Method Invocation
 - Java's version of RPC
- Server-side exposes a *skeleton*
- Client-side invokes methods on a *stub* (proxy)
- Java serialization is used for marshalling

Working with plain RMI

- RMI violates separation of concerns
 - Couples business logic to remoting infrastructure
- For example, with RMI:
 - Service interface extends **Remote**
 - Client must catch **RemoteExceptions**
- Technical Java code needed for binding and retrieving objects on the RMI server

Traditional RMI

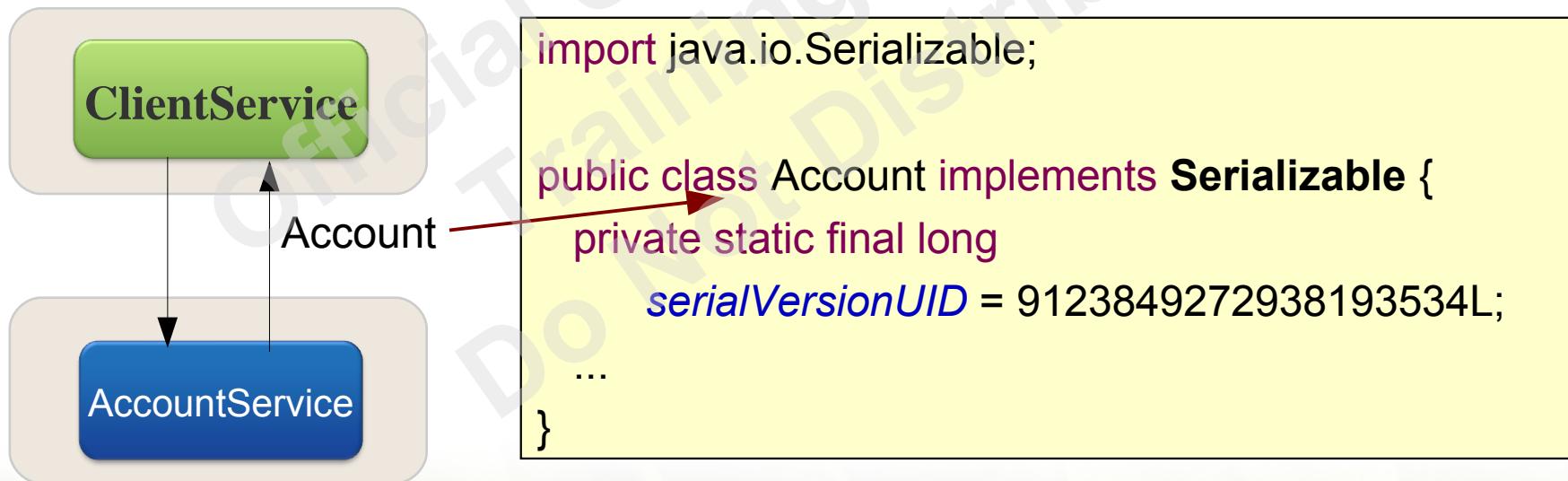
- The RMI model is invasive - server *and* client code is coupled to the framework



RMI and Serialization



- RMI relies on Object Serialization
 - Objects transferred using RMI should implement interface *Serializable*
 - Marker interface, no method to be implemented



Topics in this Session

- Introduction to Remoting
- **Spring Remoting Overview**
- Spring Remoting and RMI
- HttpInvoker

Official SpringSource
Training Only
Do Not Distribute

Goals of Spring Remoting



- Hide “plumbing” code
- Configure and expose services declaratively
- Support multiple protocols in a consistent way

Official Spring Online
Training
Do Not Distribute

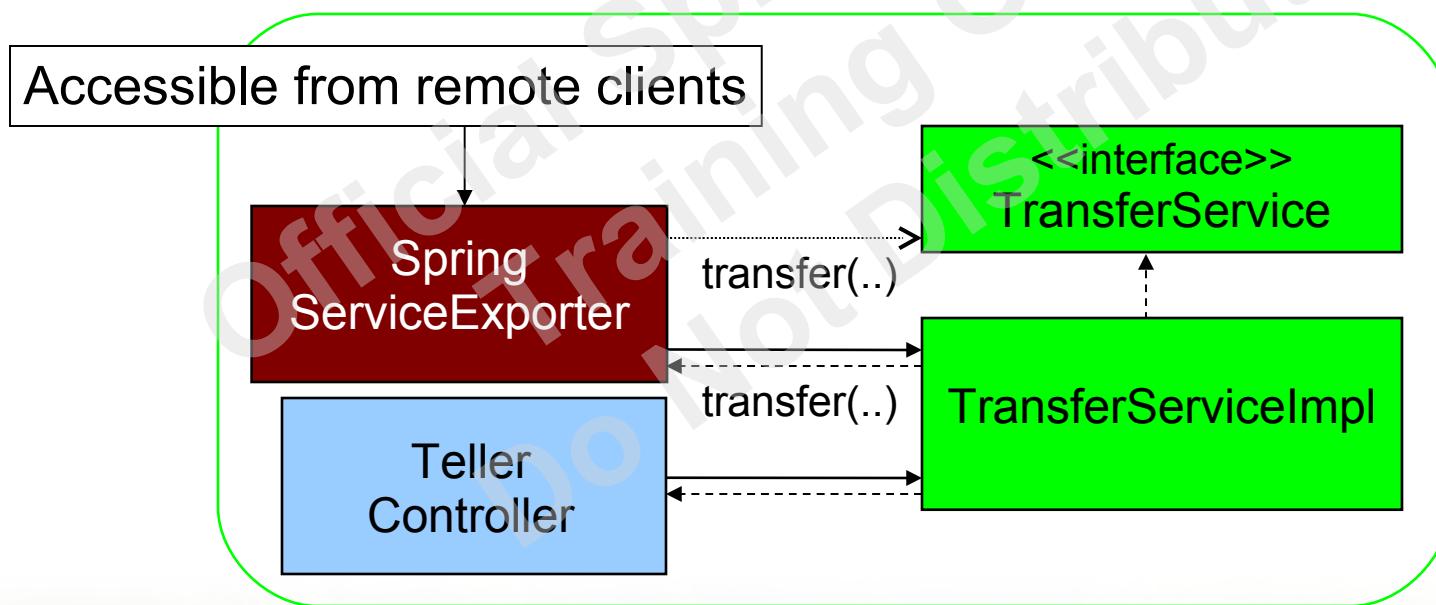
Hide the Plumbing

- Spring provides **exporters** to handle server-side requirements
 - Binding to registry or exposing an endpoint
 - Conforming to a programming model if necessary
- Spring provides FactoryBeans that generate **proxies** to handle client-side requirements
 - Communicate with the server-side endpoint
 - Convert remote exceptions to a runtime hierarchy

Service Exporters



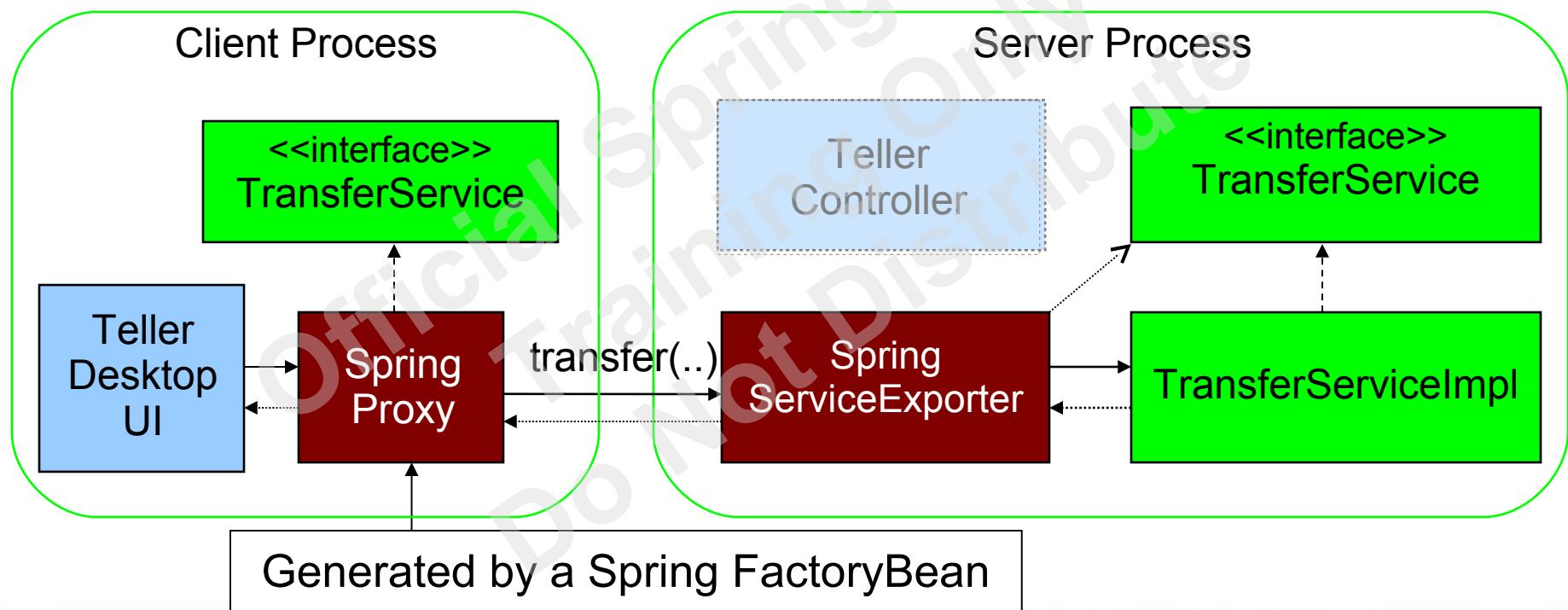
- Spring provides service exporters to enable declarative exposing of existing services



Client Proxies



- Dynamic proxies generated by Spring communicate with the service exporter



A Declarative Approach



- Uses a configuration-based approach
 - No code to write
- On the server side
 - Expose existing services with NO code changes
- On the client side
 - Invoke remote methods from existing code
 - Take advantage of polymorphism by using dependency injection
 - Migrate between remote vs. local deployments

Consistency across Protocols



- Spring's exporters and proxy FactoryBeans bring the same approach to *multiple* protocols
 - Provides flexibility
 - Promotes ease of adoption
- On the server side
 - Expose a single service over multiple protocols
- On the client side
 - Switch easily between protocols

Topics in this Session

- Introduction to Remoting
- Spring Remoting Overview
- **Spring Remoting and RMI**
- HttpInvoker

Official Spring Source
Training Only
Do Not Distribute

Spring's RMI Service Exporter



- Transparently expose an existing POJO service to the RMI registry
 - No need to write the binding code
- Avoid traditional RMI requirements
 - Service interface does not extend **Remote**
 - Service class is a POJO



Transferred objects still need to implement the interface
`java.io.Serializable`

Configuring the RMI Service Exporter



- Start with an existing POJO service

```
<bean id="transferService" class="app.impl.TransferServiceImpl">
    <property name="accountRepository" ref="accountRepository"/>
</bean>
```

- Define a bean to export it

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName" value="transferService"/> ←
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="service" ref="transferService"/>
    <property name="registryPort" value="1096"/>
</bean>
```

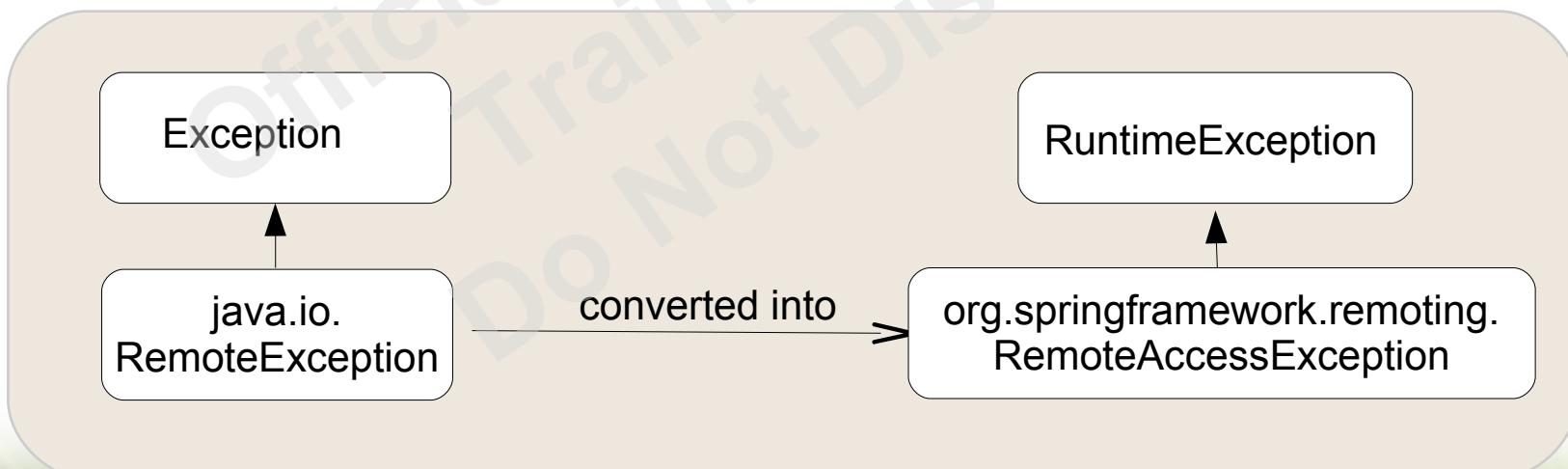
"registryPort" defaults to "1099"

Binds to rmiRegistry
as "transferService"

Spring's RMI Proxy Generator



- Spring provides FactoryBean implementation that generates RMI client-side proxy
- Simpler to use than traditional RMI stub
 - Converts checked RemoteExceptions into Spring's *runtime* hierarchy of RemoteAccessExceptions
 - Dynamically implements the business interface



Configuring the RMI Proxy



Client

- Define a factory bean to generate the proxy

```
<bean id="transferService"
      class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="serviceUrl" value="rmi://foo:1099/transferService"/>
</bean>
```

- Inject it into the client

```
<bean id="tellerDesktopUI" class="app.TellerDesktopUI">
  <property name="transferService" ref="transferService"/>
</bean>
```

TellerDesktopUI only depends on the TransferService interface

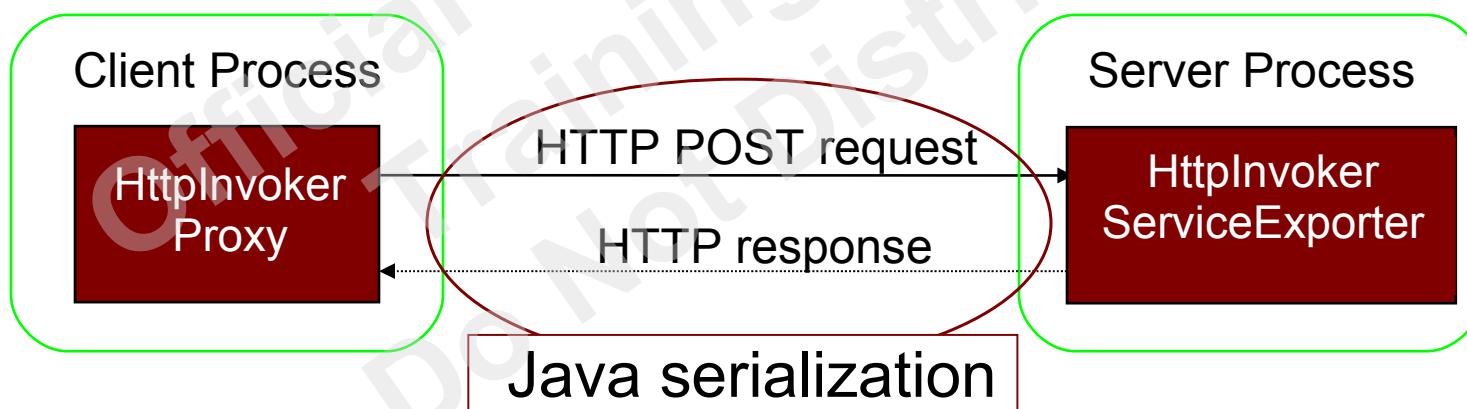
Topics in this Session

- Introduction to Remoting
- Spring Remoting Overview
- Spring Remoting and RMI
- **HttpInvoker**

Official SpringSource
Training Only
Do Not Distribute

Spring's HttpInvoker

- Lightweight HTTP-based remoting protocol
 - Method invocation converted to HTTP POST
 - Method result returned as an HTTP response
 - Method parameters and return values marshalled with standard Java serialization



HttpInvoker is using serialization → transferred objects need to implement the interface `java.io.Serializable`

Configuring the HttpInvoker Service Exporter (1)



Server

- Start with an existing POJO service

```
<bean id="transferService" class="app.impl.TransferServiceImpl">
    <property name="accountRepository" ref="accountRepository"/>
</bean>
```

- Define a bean to export it

```
<bean id="/transfer"          ← endpoint for HTTP request handling
      class="org.springframework.remoting.httpinvoker.
              HttpInvokerServiceExporter">
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="service" ref="transferService"/>
</bean>
```

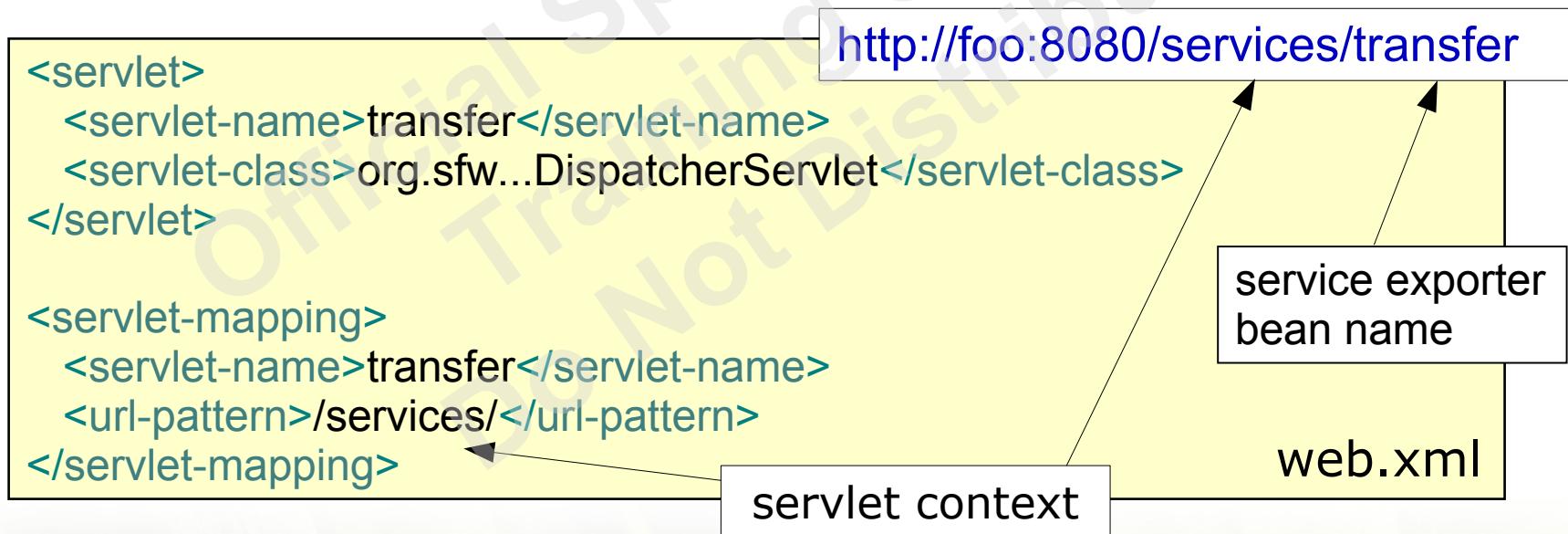


Pre Spring 3.1: Spring did not allow characters such as '/' in an *id* attribute. If special characters are needed, the *name* attribute could be used instead.

Configuring the HttpInvoker Service Exporter (2a)



- Expose the HTTP service via DispatcherServlet
 - Uses BeanNameUrlHandlerMapping to map requests to service
 - Can expose *multiple* exporters



Configuring the HttpInvoker Service Exporter (2b)



- Alternatively define HttpServletRequestHandlerServlet
 - Doesn't require Spring-MVC
 - Service exporter bean is defined in root context
 - No handler mapping – servlet can only be used by *one* HttpInvoker exporter
 - Servlet name *must* match bean name

```
<servlet>
  <servlet-name>transfer</servlet-name>
  <servlet-class>org.sfw...HttpServletRequestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>transfer</servlet-name>
  <url-pattern>/services/transfer</url-pattern>
</servlet-mapping>
```

http://foo:8080/services/transfer

service exporter bean name

web.xml

A diagram illustrating the configuration of a service exporter bean in a web.xml file. It shows two sections of XML code. The top section defines a servlet named 'transfer' with a specific class. The bottom section maps this servlet to a URL pattern. Two callout boxes point from the text 'http://foo:8080/services/transfer' and 'service exporter bean name' to their respective counterparts in the XML. An arrow points from the 'service exporter bean name' box to the 'servlet-name' attribute in the servlet definition.

Configuring the HttpInvoker Proxy



Client

- Define a factory bean to generate the proxy

```
<bean id="transferService"
      class="org.springframework.remoting.httpinvoker.
      HttpInvokerProxyFactoryBean">
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="serviceUrl" value="http://foo:8080/services/transfer"/>
</bean>
```

HTTP POST requests will be sent to this URL

- Inject it into the client

```
<bean id="tellerDesktopUI" class="app.TellerDesktopUI">
  <property name="transferService" ref="transferService"/>
</bean>
```

Remoting: Pros and Cons



- Advantages
 - (Too) Simple to setup and use
 - Abstracts away all messaging concerns
- Disadvantages
 - Client-server tightly coupled by shared interface
 - hard to maintain, especially with lots of clients
 - Can't control underlying messaging (hidden)
 - Difficult to scale
 - Make your requests *high-level*
 - Not interoperable, Java only

Enterprise Integration with Spring –
4 day course on application integration



LAB

Simplifying Distributed Applications with Spring
Remoting



Spring JMS

Simplifying Messaging Applications

Topics in this Session



- **Introduction to JMS**
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- Advanced Features

- The JMS API provides an abstraction for accessing Message Oriented Middleware
 - Avoid vendor lock-in
 - Increase portability
- JMS does *not* enable different MOM vendors to communicate
 - Need a bridge (expensive)
 - Or use AMQP (standard msg protocol, like SMTP)
 - See RabbitMQ from VMware

JMS Core Components



- Message
- Destination
- Connection
- Session
- MessageProducer
- MessageConsumer

Official Springsource
Training Only
Do Not Distribute

JMS Message Types



- Implementations of the Message interface
 - TextMessage
 - ObjectMessage
 - MapMessage
 - BytesMessage
 - StreamMessage

JMS Destination Types

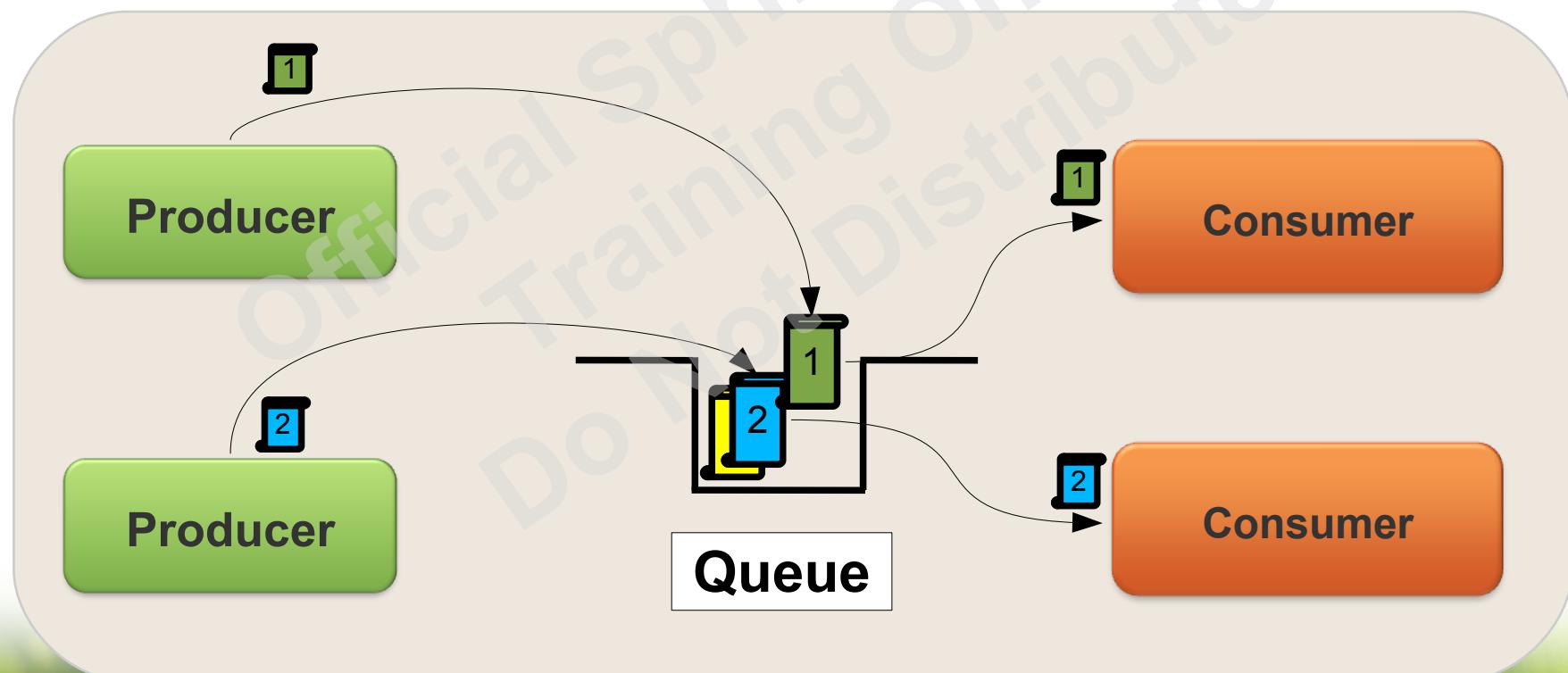


- Implementations of the Destination interface
 - Queue
 - Point-to-point messaging
 - Topic
 - Publish/subscribe messaging
- Both support *multiple* producers and consumers
 - Messages are different
 - Let's take a closer look ...

JMS Queues: Point-to-point



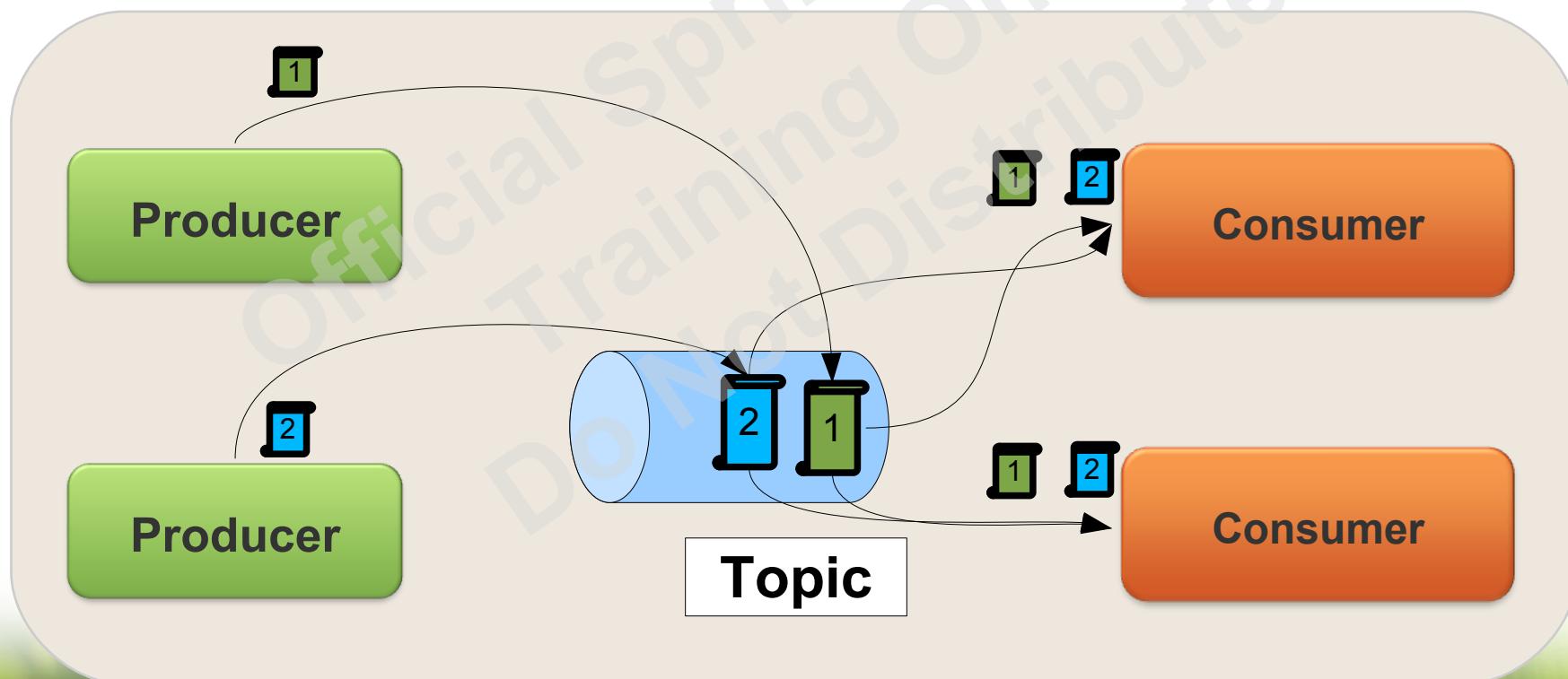
1. Message sent to queue
2. Message queued
3. Message consumed by *single* consumer



JMS Topics: Publish-subscribe



1. Message sent to topic
2. Message optionally stored
3. Message distributed to *all* subscribers



The JMS Connection



- A JMS Connection is obtained from a factory

```
Connection conn = connectionFactory.createConnection();
```

- Typical enterprise application:
 - ConnectionFactory is a managed resource bound to JNDI

```
Properties env = new Properties();
// provide JNDI environment properties
Context ctx = new InitialContext(env);
ConnectionFactory connectionFactory =
    (ConnectionFactory) ctx.lookup("connFactory");
```

The JMS Session



- A Session is created from the Connection
 - Represents a unit-of-work
 - Provides transactional capability

```
Session session = conn.createSession(  
    boolean transacted, int acknowledgeMode);
```

```
// use session  
if (everythingOkay) {  
    session.commit();  
} else {  
    session.rollback();  
}
```

Creating Messages



- The Session is responsible for the creation of various JMS Message types

```
session.createTextMessage("Some Message Content");
```

```
session.createObjectMessage(someSerializableObject);
```

```
MapMessage message = session.createMapMessage();  
message.setInt("someKey", 123);
```

```
BytesMessage message = session.createBytesMessage();  
message.writeBytes(someByteArray);
```

Producers and Consumers



- The Session is also responsible for creating instances of MessageProducer and MessageConsumer

```
producer = session.createProducer(someDestination);  
  
consumer = session.createConsumer(someDestination);
```

Topics in this Session



- Introduction to JMS
- **Apache ActiveMQ**
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- Advanced Features

JMS Providers



- Most providers of Message Oriented Middleware (MoM) support JMS
 - WebSphere MQ, Tibco EMS, Oracle EMS, JBoss AP, SwiftMQ, etc.
 - Some are Open Source, some commercial
 - Some are implemented in Java themselves
- The lab for this module uses Apache ActiveMQ

Apache ActiveMQ

- Open source message broker written in Java
- Supports JMS and many other APIs
 - Including non-Java clients!
- Can be used stand-alone in production environment
 - 'activemq' script in download starts with default config
 - SpringSource provides support
- Can also be used embedded in an application
 - Configured through ActiveMQ or Spring xml files
 - What we use in the labs

Apache ActiveMQ Features



Support for:

- Many cross language clients & transport protocols
 - Incl. excellent Spring integration
- Flexible & powerful deployment configuration
 - Clustering incl. load-balancing & failover, ...
- Advanced messaging features
 - Message groups, virtual & composite destinations, wildcards, etc.
- Enterprise Integration Patterns when combined with Spring Integration or Apache Camel
 - from the book by Gregor Hohpe & Bobby Woolf

Topics in this Session



- Introduction to JMS
- Apache ActiveMQ
- **Configuring JMS Resources with Spring**
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- Advanced Features

Configuring JMS Resources with Spring



- Spring enables decoupling of your application code from the underlying infrastructure
 - Container provides the resources
 - Application is simply coded against the API
- Provides deployment flexibility
 - use a standalone JMS provider
 - use an application server to manage JMS resources

Configuring a ConnectionFactory



- ConnectionFactory may be standalone

```
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

- Or retrieved from JNDI

```
<jee:jndi-lookup id="connectionFactory"
    jndi-name="jms/ConnectionFactory"/>
```

Configuring Destinations



- Destinations may be standalone

```
<bean id="orderQueue"
      class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="queue.order"/>
</bean>
```

- Or retrieved from JNDI

```
<jee:jndi-lookup id="orderQueue"
    jndi-name="jms/OrderQueue"/>
```

Topics in this Session



- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- **Spring's JmsTemplate**
- Sending Messages
- Receiving Messages
- Advanced Features

Spring's JmsTemplate

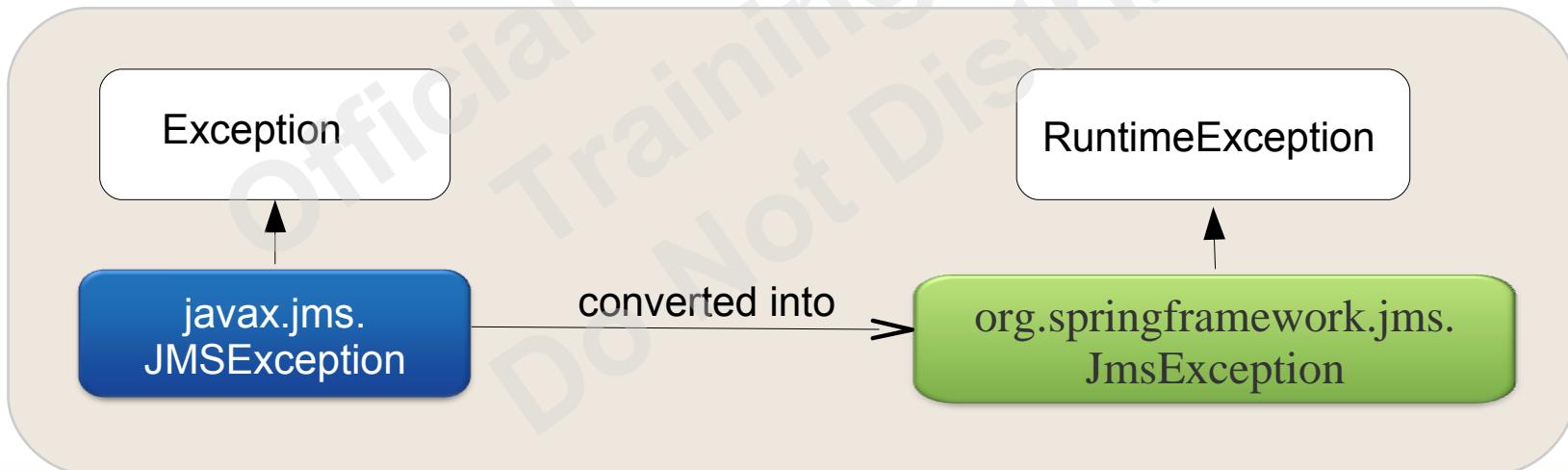


- The template simplifies usage of the API
 - Reduces boilerplate code
 - Manages resources transparently
 - Converts checked exceptions to runtime equivalents
 - Provides convenience methods and callbacks

NOTE: The *AmqpTemplate* (used with RabbitMQ) has an almost identical API to the *JmsTemplate* – they offer similar abstractions over very different products

Exception Handling

- Exceptions in JMS are checked by default
- JmsTemplate converts checked exceptions to runtime equivalents



JmsTemplate Strategies

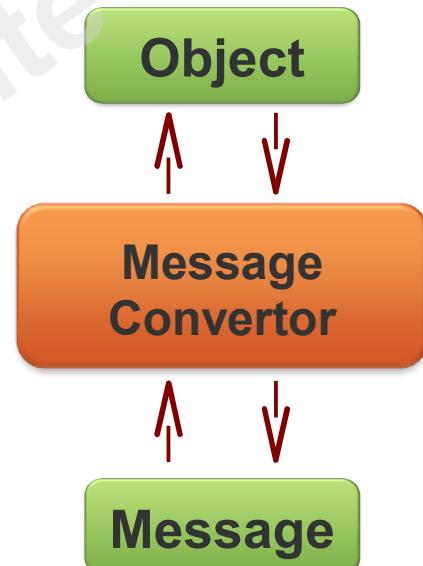


- The JmsTemplate delegates to two collaborators to handle some of the work
 - MessageConverter
 - DestinationResolver

MessageConverter



- The JmsTemplate uses a **MessageConverter** to convert between objects and messages
 - You only send and receive objects
- The default **SimpleMessageConverter** handles basic types
 - String to TextMessage
 - Map to MapMessage
 - byte[] to BytesMessage
 - Serializable to ObjectMessage



NOTE: It is possible to implement custom converters by implementing the *MessageConverter* interface

DestinationResolver

- Convenient to use destination names at runtime
- DynamicDestinationResolver used by default
 - Resolves topic and queue names
 - *Not* their Spring bean names
- JndiDestinationResolver also available



```
Destination resolveDestinationName(Session session,  
        String destinationName,  
        boolean pubSubDomain) ←  
throws JMSException;
```

publish-subscribe?
true → Topic
false → Queue

JmsTemplate configuration



- *Must* provide reference to ConnectionFactory
 - via either constructor or setter injection
- Optionally provide other facilities
 - setMessageConverter
 - setDestinationResolver
 - setDefaultDestination or setDefaultDestinationName

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <constructor-arg ref="connectionFactory"/>
    <property name="defaultDestination" ref="orderQueue"/>
</bean>
```

Topics in this Session



- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- **Sending Messages**
- Receiving Messages
- Advanced Features

Sending Messages



- The template provides options
 - Simple methods to send a JMS message
 - One line methods that leverage the template's MessageConverter
 - Callback-accepting methods that reveal more of the JMS API
- Use the simplest option for the task at hand

Sending POJO

- A message can be sent in one single line

```
public class JmsOrderManager implements OrderManager {  
    @Autowired  
    private JmsTemplate jmsTemplate;  
    @Autowired  
    private Destination orderQueue;  
  
    public void placeOrder(Order order) {  
        String stringMessage = "New order " + order.getNumber();  
        jmsTemplate.convertAndSend("messageQueue", stringMessage );  
            // use destination resolver and message converter  
  
        jmsTemplate.convertAndSend(orderQueue, order); // use message converter  
  
        jmsTemplate.convertAndSend(order); // use converter and default destination  
    }  
}
```

Sending JMS Messages

- Useful when you need to access JMS API
 - eg. set expiration, redelivery mode...

```
public void sendMessage(final String msg) {  
    MessageCreator messageCreator = new MessageCreator() {  
  
        public Message createMessage(Session session) throws JMSException {  
            TextMessage message = session.createTextMessage(msg);  
            message.setJMSExpiration(2000); // 2 seconds  
            return message;  
        }  
    };  
  
    this.jmsTemplate.send(messageCreator);  
}
```

Topics in this Session



- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- **Receiving Messages**
- Advanced Features

Synchronous Message Reception



- Template can also receive messages
 - but methods are blocking (synchronous)
 - optional timeout: `setReceiveTimeout()`

```
public void receiveMessages() {  
  
    // use message converter and destination resolver  
    String s = (String) this.jmsTemplate.receiveAndConvert("messageQueue");  
  
    // use message converter  
    Order order1 = (Order) this.jmsTemplate.receiveAndConvert(orderQueue);  
  
    // handle JMS native message from default destination  
    ObjectMessage orderMessage =  
        (ObjectMessage) this.jmsTemplate.receive();  
    Order order2 = (Order) orderMessage.getObject();  
}
```

The JMS MessageListener



- The JMS API defines this interface for asynchronous reception of messages

```
import javax.jms.Message;
import javax.jms.MessageListener;

public class OrderListener implements MessageListener {
    public void onMessage(Message message) { // ... }
}
```

- Traditionally, using a MessageListener required an EJB container
 - Message Driven Beans

Spring's MessageListener Containers



- Spring provides lightweight alternatives
 - *SimpleMessageListenerContainer*
 - Uses plain JMS client API
 - Creates a fixed number of Sessions
 - *DefaultMessageListenerContainer*
 - Adds transactional capability
- Advanced scheduling and endpoint management options available for each container option

Defining a plain JMS Message Listener



- Define listeners using jms:listener elements

```
<jms:listener-container connection-factory="myConnectionFactory">
    <jms:listener destination="queue.order" ref="myOrderListener"/>
    <jms:listener destination="queue.conf" ref="myConfListener"/>
</jms:listener-container>
```

- Listener needs to implement MessageListener
 - or SessionAwareMessageListener
 - jms:listener-container configurable
 - task execution strategy, concurrency, container type, transaction manager and more

Spring's Message-Driven POJO



- Spring also allows you to specify a plain Java object that can serve as a listener
 - MessageConverter provides parameter
 - Any return value sent to response-destination after conversion

```
public class OrderService { ①  
    public OrderConfirmation order(Order o) {}  
} ③ ②
```

```
<jms:listener  
    ref="orderService" ①  
    method="order" ②  
    destination="queue.orders"  
    response-destination="queue.confirmation"/>
```

Messaging: Pros and Cons



- Advantages
 - Resilience, guaranteed delivery
 - Asynchronous support
 - Application freed from messaging concerns
 - Interoperable – languages, environments
- Disadvantages
 - Requires additional third-party software
 - Can be expensive to install and maintain
 - More complex to use – *but not with JmsTemplate!*

*Enterprise Integration with Spring –
4 day course on application integration*



LAB

Sending and Receiving Messages in a Spring Environment

Topics in this Session



- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages
- **Advanced Features**

SessionCallback Example: Synchronous Request-Reply



```
RewardConfirmation confirmation =  
    jmsTemplate.execute(new SessionCallback() {  
  
public Object doInJms(Session session) throws JMSException {  
    TemporaryQueue replyQueue = session.createTemporaryQueue();  
    Message request = session.createObjectMessage(dining);  
    request.setJMSReplyTo(replyQueue);  
    MessageProducer producer = session.createProducer(diningQueue);  
    producer.send(request);  
  
    MessageConsumer consumer = session.createConsumer(replyQueue);  
    ObjectMessage reply = (ObjectMessage) consumer.receive(60000);  
    replyQueue.delete(); // delete here, since Connection might be cached  
    return reply != null ? (RewardConfirmation) reply.getObject : null;  
}  
});
```

Advanced Option: CachingConnectionFactory



- JmsTemplate aggressively closes and reopens resources like Sessions and Connections
 - Normally these are cached by connection factory
 - Without caching can cause lots of overhead
 - Resulting in poor performance
- Use our CachingConnectionFactory to add caching within the application if needed

```
<bean id="connectionFactory"
      class="org.springframework.jms.connection.CachingConnectionFactory">
  <property name="targetConnectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://embedded?broker.persistent=false"/>
    </bean>
  </property>
</bean>
```



Performance and Operations with Spring

Management and Monitoring of Java
Applications

Topics in this Session

- **Introduction**
- JMX
- Introducing Spring JMX
- Explicitly exporting beans with Spring
- Automatically exporting existing MBeans
- Spring Insight

Overall Goals

- Gather information about application during runtime
- Dynamically reconfigure app to align to external occasions
- Trigger operations inside the application
- Even adapt to business changes in smaller scope

Topics in this Session

- Introduction
- **JMX**
- Introducing Spring JMX
- Explicitly exporting beans with Spring
- Automatically exporting existing MBeans
- Spring Insight

What is JMX?

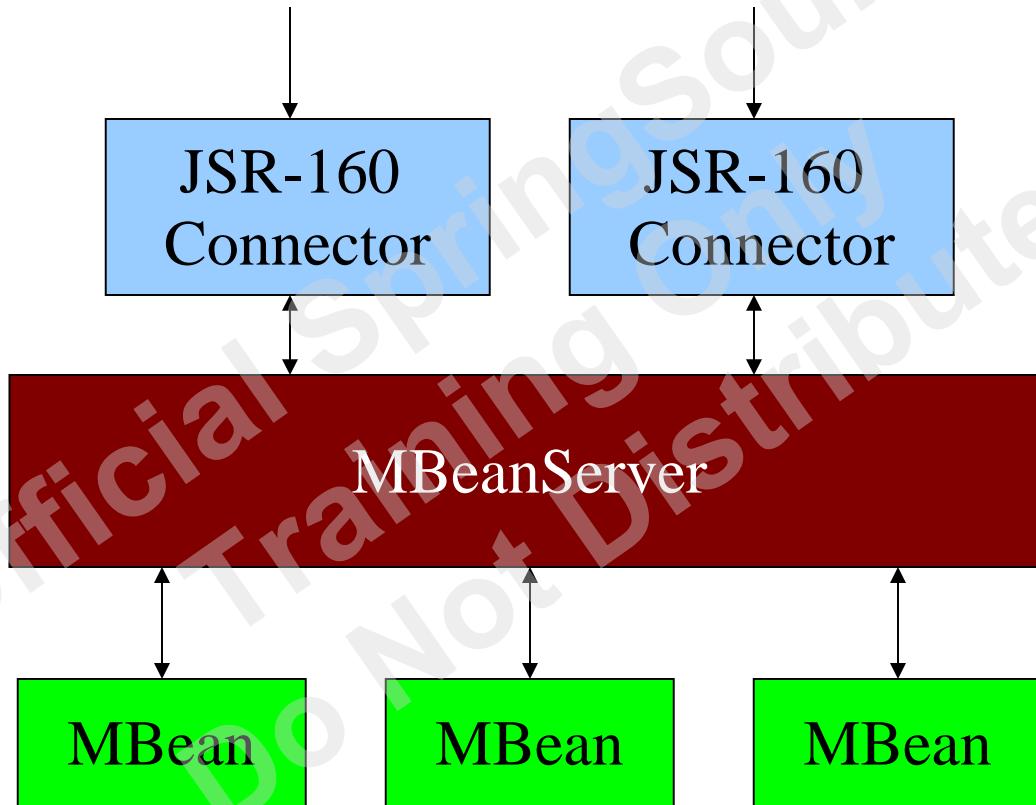
- The Java Management Extensions specification aims to create a standard API for adding management and monitoring to Java applications
- Management
 - Changing configuration properties at runtime
- Monitoring
 - Reporting cache hit/miss ratios at runtime

How JMX Works



- To add this management and monitoring capability, JMX instruments application components
- JMX introduces the concept of the MBean
 - An object with management metadata

JMX Architecture



JMX Architecture



- MBeanServer acts as broker for communication between
 - Multiple local MBeans
 - Remote clients and MBeans
- MBeanServer maintains a keyed reference to all MBeans registered with it
 - *object name*
- Many generic clients available
 - JDK: jconsole, jvisualvm
 - VMware: Hyperic, <http://www.hyperic.com/>



- An MBean is an object with additional management metadata
 - Attributes (→ properties)
 - Operations (→ methods)
- The management metadata can be defined statically with a Java interface or defined dynamically at runtime
 - Simple MBean or Dynamic MBean respectively

Plain JMX – Part I



```
public interface JmxCounterMBean {  
  
    int getCount();      // becomes Attribute named 'Count'  
  
    void increment();   // becomes Operation named 'increment'  
}
```

```
public class JmxCounter implements JmxCounterMBean {  
  
    public int getCount() {...}  
  
    public void increment() {...}  
}
```

Plain JMX – Part II



```
MBeanServer server = ManagementFactory.getPlatformMBeanServer();

JmxCounter bean = new JmxCounter(...);

try {
    ObjectName name = new ObjectName("ourapp:name=counter");
    server.registerMBean(bean, name);
} catch (Exception e) {
    e.printStackTrace();
}
```

Topics in this Session

- Introduction
- JMX
- **Introducing Spring JMX**
- Explicitly exporting beans with Spring
- Automatically exporting existing MBeans
- Spring Insight

Goals of Spring JMX

- Using the raw JMX API is difficult and complex
- The goal of Spring's JMX support is to simplify the use of JMX while hiding the complexity of the API

Goals of Spring JMX



- Configuring JMX infrastructure
 - Declaratively using context namespace or FactoryBeans
- Exposing Spring beans as MBeans
 - Annotation based metadata
 - Declaratively using Spring bean definitions
- Consuming JMX managed beans
 - Transparently using a proxy-based mechanism

Creating an MBeanServer



- To locate or create an MBeanServer declaratively, use the context namespace

```
<context:mbean-server />
```

... or declare it explicitly

```
<bean id="mbeanServer"
      class="org.springframework.jmx.support.
      MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true"/>
</bean>
```

Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- **Explicitly exporting beans with Spring**
- Automatically exporting existing MBeans
- Spring Insight

Export bean as JMX MBean using Annotations



- How it works
 - Annotate your class and methods to be exposed
 - Activate exporter in Bean XML
 - ObjectName derived from @ManagedResource attribute or fully qualified classname

```
@ManagedResource(objectName="statistics:name=counter",
                  description="A simple JMX counter")
public class JmxCounterImpl implements JmxCounter {

    @ManagedAttribute(description="The counter value")
    public int getCount() {...}

    @ManagedOperation(description="Increments the counter value")
    public void increment() {...}
}
```

<context:mbean-export>

Spring's MBeanExporter



- Transparently expose an existing POJO bean to the MBeanServer
 - No need to write the registration code
- By default avoids the need to create an explicit management interface or create an ObjectName instance
 - Uses reflection to manage all properties and methods
 - Uses map key as the ObjectName

Exporting a bean as an MBean



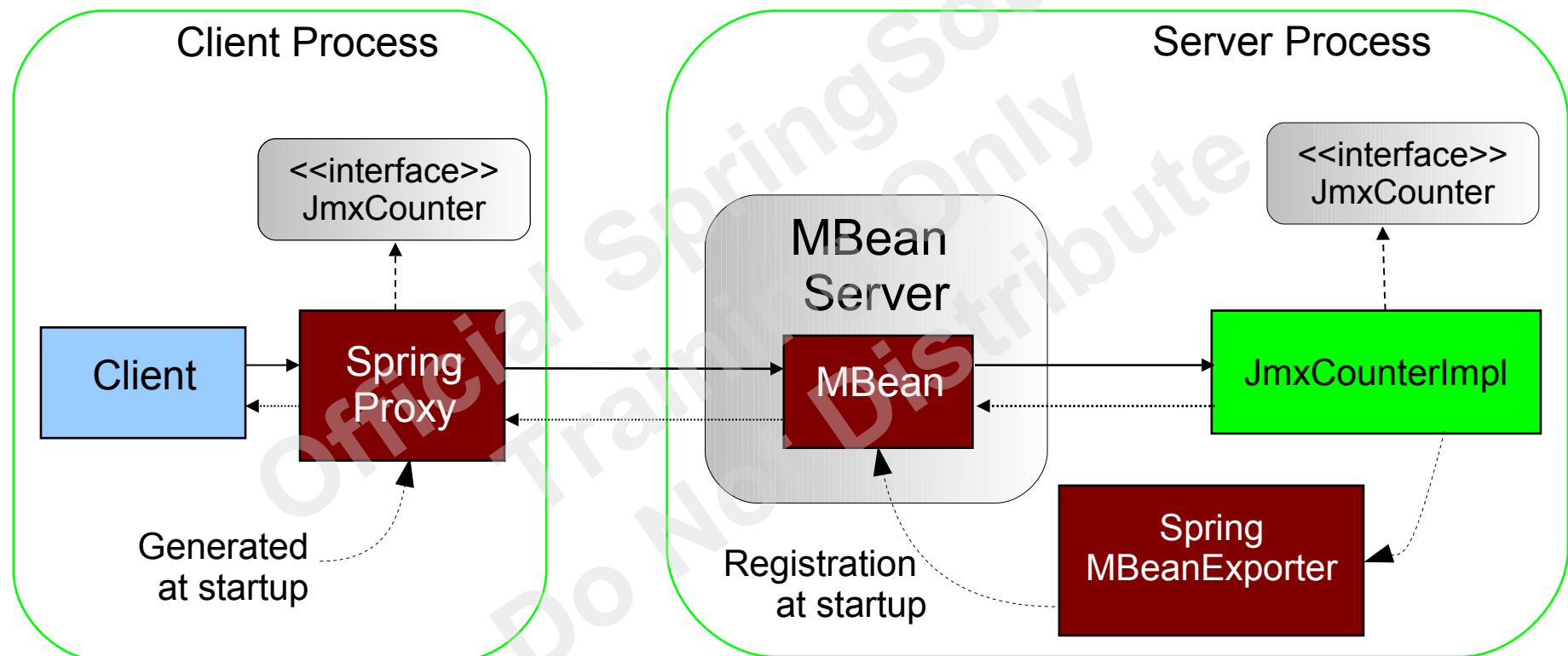
- Start with an existing POJO bean

```
<bean id="messageService" class="example.MessageService"/>
```

- Use the MBeanExporter to export it

```
<bean class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="service:name=messageService"
            value-ref="messageService"/>
    </map>
  </property>
</bean>
```

Spring in the JMX architecture



Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- Explicitly exporting beans with Spring
- **Automatically exporting existing MBeans**
- Spring Insight

Automatically exporting pre-existing MBeans



- Some beans are MBeans themselves
 - Example: Log4j's LoggerDynamicMBean
 - Spring will auto-detect and export them for you

```
<context:mbean-export>

<bean class="org.apache.log4j.jmx.LoggerDynamicMBean">
    <constructor-arg>
        <bean class="org.apache.log4j.Logger"
              factory-method="getLogger"/>
        <constructor-arg value="org.springframework.jmx" />
    </bean>
    </constructor-arg>
</bean>
```

Automatically exporting pre-existing MBeans



- Hibernate StatisticsService

```
<context:mbean-export/>
```

```
<bean id="statisticsService" class="org.hibernate.jmx.StatisticsService">
    <property name="statisticsEnabled" value="true" />
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Pure Hibernate

```
<property name="sessionFactory">
    <!-- Get sessionFactory property from Hibernate E.M.F. -->
    <util:property-path path="entityManagerFactory.sessionFactory" />
</property>
```

JPA + Hibernate

Need *util:property-path* to access custom property of HibernateEntityManagerFactory

Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- Explicitly exporting beans with Spring
- Automatically exporting existing MBeans
- **Spring Insight**

Spring Insight Overview



- Part of tc Server Developer Edition
 - Monitors web applications deployed to tc Server
 - <http://localhost:8080/insight>
- Focuses on what's relevant
 - esp. performance related parts of the application
- Detects performance issues during development
 - Commercial version for production: *vFabric APM*

Spring Insight Overview



springinsight vFabric Technology

Home Forum Blog Help

Browse Resources Recent Activity Administration

15m @ now ▾ ◀ ▶

APPLICATIONS

- All Applications
 - /jmx-solution
 - Servlet: rewards**
 - Servlet: default
 - / (Welcome to tc Runtime)
 - /manager (Tomcat Manager Appl)

END POINT

Servlet: rewards
POST /jmx-solution/rewards

Throughput Trend Error Rate Response Time Trend

9.0 tpm 6.0 tpm 3.0 tpm 0.0 tpm

15 minutes ago Live

VITALS

Throughput 0.1 tpm
Invocations 2
Errors 0 (0.0%)

RESPONSE TIME

95th Percentile 2.1 s
Mean 970 ms
Standard Deviation 1.3 s

RESPONSE TIME HISTOGRAM

Health Legend: Edit

- 1 Frustrated >800 ms, error
- 0 Tolerated 200 ms - 800 ms
- 1 Satisfied <200 ms

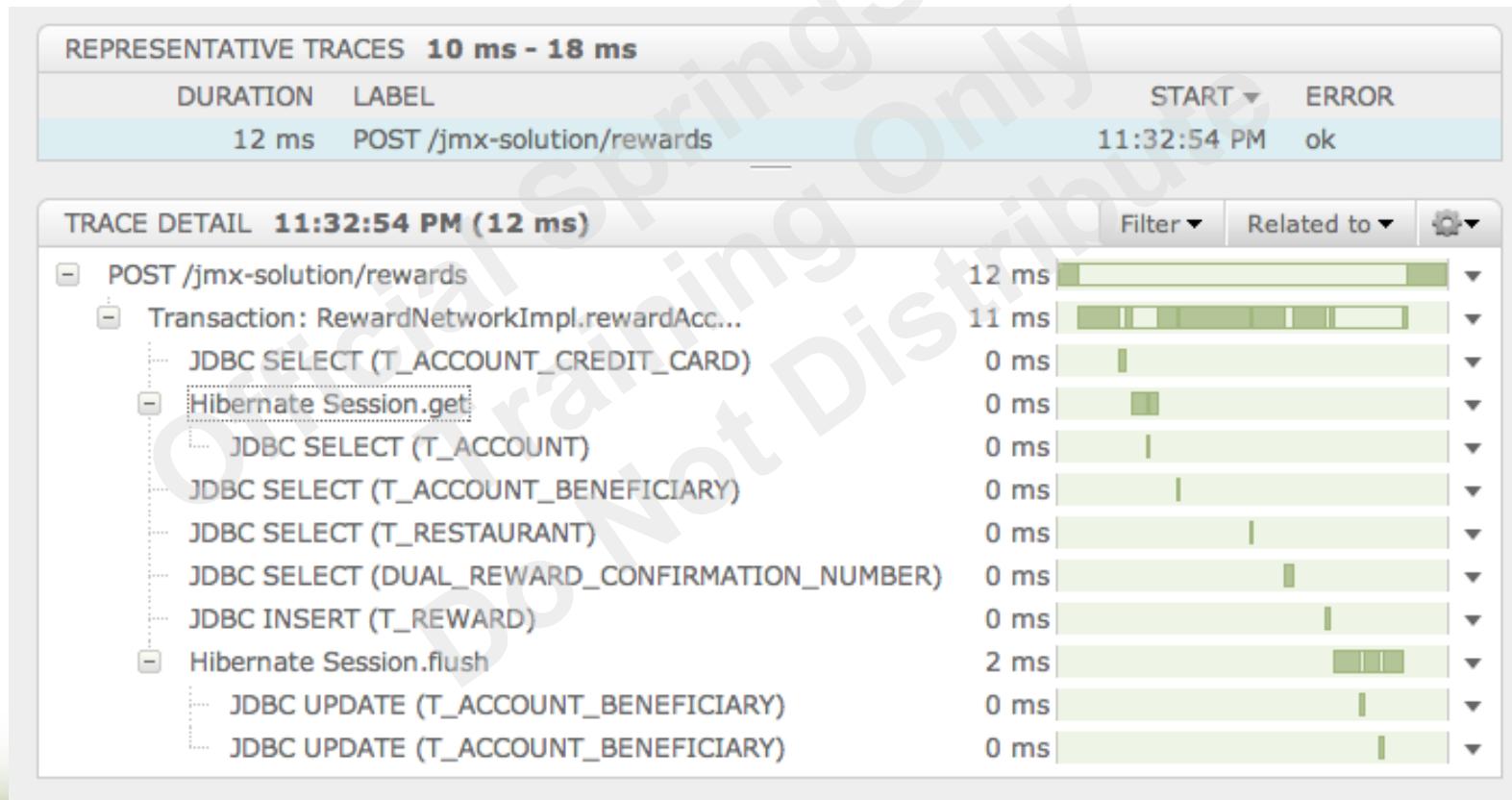
Timeline

Servlet Selector

Spring Insight Overview



- A request trace from HTTP POST to SQL



Summary

- Spring JMX
 - Export Spring-managed beans to a JMX MBeanServer
 - Simple value-add now that your beans are managed
- Steps
 - Use `<context:mbean-server>` to create MBean server
 - Use Spring annotations to declare JMX metadata
 - Use `<context:mbean-export>` to automatically export annotated and pre-existing MBeans
- Spring Insight
 - Deep view into your web-application in STS
 - Production version available in *vFabric APM*



What's next?

- Certification
- Other courses
- Resources
- Evaluation

Certification



- Computer-based exam
 - 50 multiple-choice questions
 - 90 minutes
 - Passing score: 76% (38 questions answered successfully)
- Preparation
 - See Core Spring 3.x certification guide

<http://www.springsource.com/training/certification/springprofessional>

- Review all the slides
- Redo the labs

Certification: Questions



Typical question

- Statements
 - a. An application context holds Spring beans
 - b. An application context can be reloaded at any time
 - c. Spring provides many types of application context
- Pick the correct response:
 - 1. Only a. is correct
 - 2. Both a. and c. are correct
 - 3. All are correct
 - 4. None are correct

Certification: Logistics



- Where?
 - At any Pearson VUE Test Center
 - Most large or medium-sized cities
 - See <http://www.pearsonvue.com/vtclocator/>
- How?
 - At the end of the class, you will receive a certification voucher by email
 - Make an appointment
 - Give them the voucher when you take the test
- For any further inquiry, you can write to
 - eduoperations@vmware.com

Other courses

- Many courses available
 - Web Applications with Spring
 - Enterprise Integration with Spring
 - Hibernate with Spring
 - Groovy and Grails
 - tc Server, Tomcat, Hyperic
 - ...
- More details here:
<http://www.springsource.com/training/curriculum>

Web Applications with Spring



- 4-day workshop
- Making the most of Spring in the web layer
 - Spring MVC
 - Spring Web Flow
 - REST using MVC and AJAX
 - Productivity with Spring Roo
 - Security of Web applications
 - Performance testing
- Spring Web Application Developer certification



Enterprise Integration with Spring (EIwS)



- Building loosely coupled, event-driven architectures
 - Separate processing, communications & integration
- 4 day course covering
 - Concurrency
 - Advanced transaction management
 - SOAP Web Services with Spring WS
 - REST Web Services with Spring MVC
 - Spring Batch
 - Spring Integration

Hibernate with Spring



- 3 day course covering
 - Configure Hibernate applications with Spring and Spring Transactions
 - Implement inheritance and relationships with JPA and Hibernate
 - Discover how Hibernate manages objects
 - Go more in depth on locking with Hibernate
 - Advanced features such as interceptors, caching and batch updates

Consulting Offerings



- Quick Scan
 - Expert reviews project or architecture and shows how to improve
- Hyperic Jump Start
 - Helps you starting with Hyperic to monitor your environment
- Java EE or Tomcat to tc Server Migration
 - Migrate your application and production environment to tc Server
- Custom consulting engagements
 - to fit your specific needs

Resources

- The Spring reference documentation
 - <http://www.springsource.org/documentation>
 - Already 800+ pages!
- The official technical blog
 - <http://blog.springsource.com/>
- The Spring community forums
 - <http://forum.springframework.org/>

Resources (2)

- You can register issues on our Jira repository
 - <http://jira.springframework.org/>
- The source code is available here
 - <https://src.springsource.org/svn/spring-framework>
 - More information on how to build Spring:
<http://blog.springsource.com/2009/03/03/building-spring-3/>
- Follow Spring development
 - <https://fisheye.springsource.org/browse/>

Thank You!

We hope you enjoyed the course

Please fill out the evaluation form on MyLearn

<http://www.vmware.com/education>





Object/Relational Mapping

Fundamental Concepts and Concerns When
Using O/R Mapping in Enterprise Applications

Topics in this session

- **The Object/Relational mismatch**
- ORM in context
- Benefits of O/R Mapping

The Object/Relational Mismatch (1)



- A domain object model is designed to serve the needs of the application
 - Organize data into abstract concepts that prove useful to solving the domain problem
 - Encapsulate behavior specific to the application
 - Under the control of the application developer

The Object/Relational Mismatch (2)



- Relational models relate business data and are typically driven by other factors:
 - Performance
 - Space
- Furthermore, a relational database schema often:
 - Predates the application
 - Is shared with other applications
 - Is managed by a separate DBA group

Object/Relational Mapping



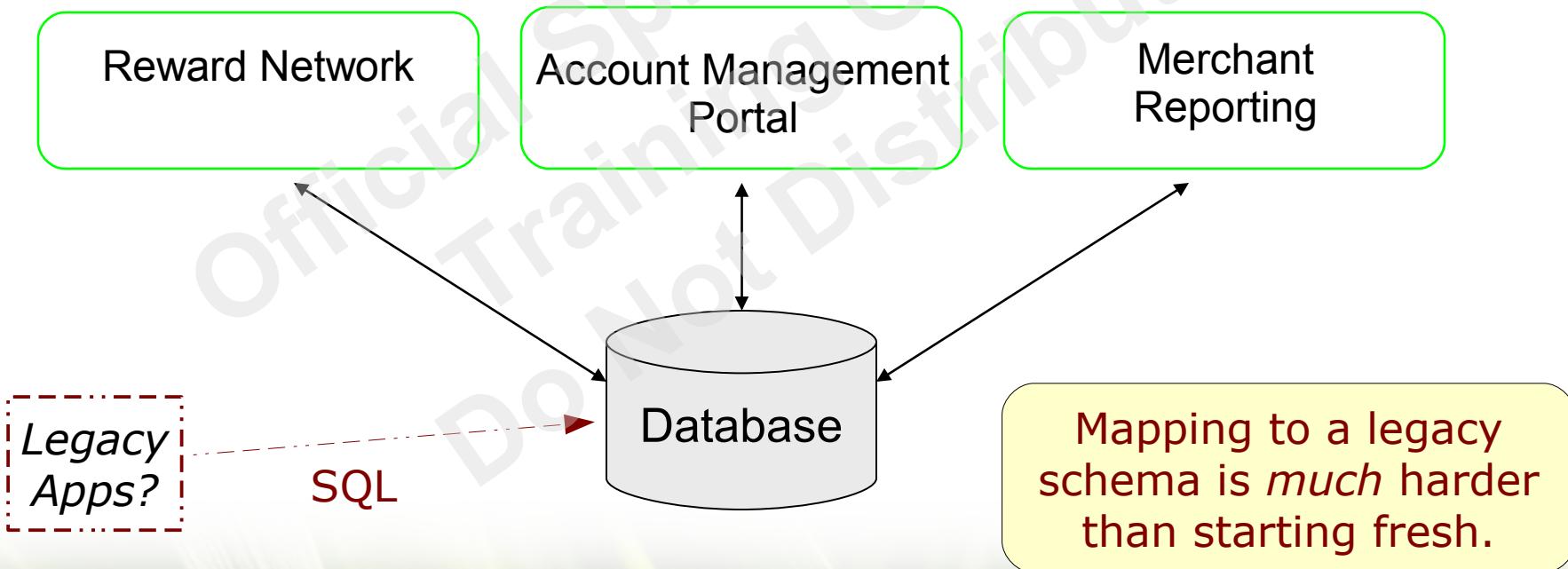
- Object/Relational Mapping (ORM) engines exist to mitigate the mismatch
- Spring supports all of the major ones:
 - Hibernate
 - EclipseLink
 - Other JPA (Java Persistence API) implementations, such as OpenJPA
- This session will focus on Hibernate

Topics in this session

- The Object/Relational Mismatch
- **ORM in context**
- Benefits of modern-day ORM engines

ORM in context

- For the **Reward Dining** domain
 - The database schema already exists
 - Several applications share the data



O/R Mismatch: Granularity (1)



- In an object-oriented language, cohesive fine-grained classes provide encapsulation and express the domain naturally
- In a database schema, granularity is typically driven by normalization and performance considerations

O/R Mismatch: Granularity (2)



just one example...

Domain Model in Java

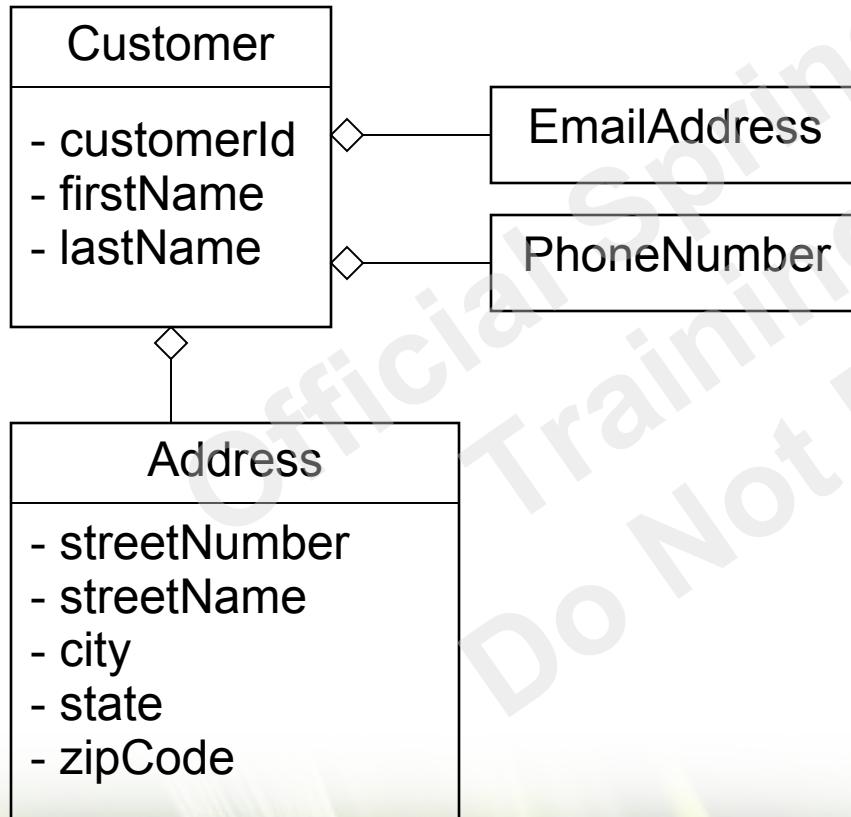


Table in Database

CUSTOMER
CUST_ID <<PK>>
FIRST_NAME
LAST_NAME
EMAIL
PHONE
STREET_NUMBER
STREET_NAME
CITY
STATE
ZIP_CODE

O/R Mismatch: Identity (1)



- In Java, there is a difference between Object identity and Object equivalence:
 - `x == y` *identity* (same memory address)
 - `x.equals(y)` *equivalence*
- In a database, identity is based solely on primary keys:
 - `x.getEntityId().equals(y.getEntityId())`

- When working with persistent Objects, the identity problem leads to difficult challenges
 - Two different Java objects may correspond to the same relational row
 - But Java says they are *not* equal
- Some of the challenges:
 - Implement equals() to accommodate this scenario
 - Determine when to update and when to insert
 - Avoid duplication when adding to a Collection

O/R Mismatch: Inheritance and Associations (1)



- In an object-oriented language:
 - *IS-A* relations are modeled with inheritance
 - *HAS-A* relations are modeled with composition
- In a database schema, relations are limited to what can be expressed by *foreign keys*

O/R Mismatch: Inheritance and Associations (2)



- Bi-directional associations are common in a domain model (e.g. Parent-Child)
 - This can be modeled naturally in each Object
- In a database:
 - One side (parent) provides a primary-key
 - Other side (child) provides a foreign-key reference
- For many-to-many associations, the database schema requires a *join table*

Topics in this session

- The Object/Relational Mismatch
- ORM in Context
- **Benefits of O/R Mapping**

Benefits of ORM

- Object Query Language
- Automatic Change Detection
- Persistence by Reachability
- Caching
 - Per-Transaction (1st Level)
 - Per-DataSource (2nd Level)

Object Query Language



- When working with domain objects, it is more natural to query based on objects.
 - Query with SQL:

```
SELECT c.first_name, c.last_name, a.city, ...
  FROM customer c, customer_address ca, address a
 WHERE ca.customer_id = c.id
   AND ca.address_id = a.id
   AND a.zip_code = 12345
```

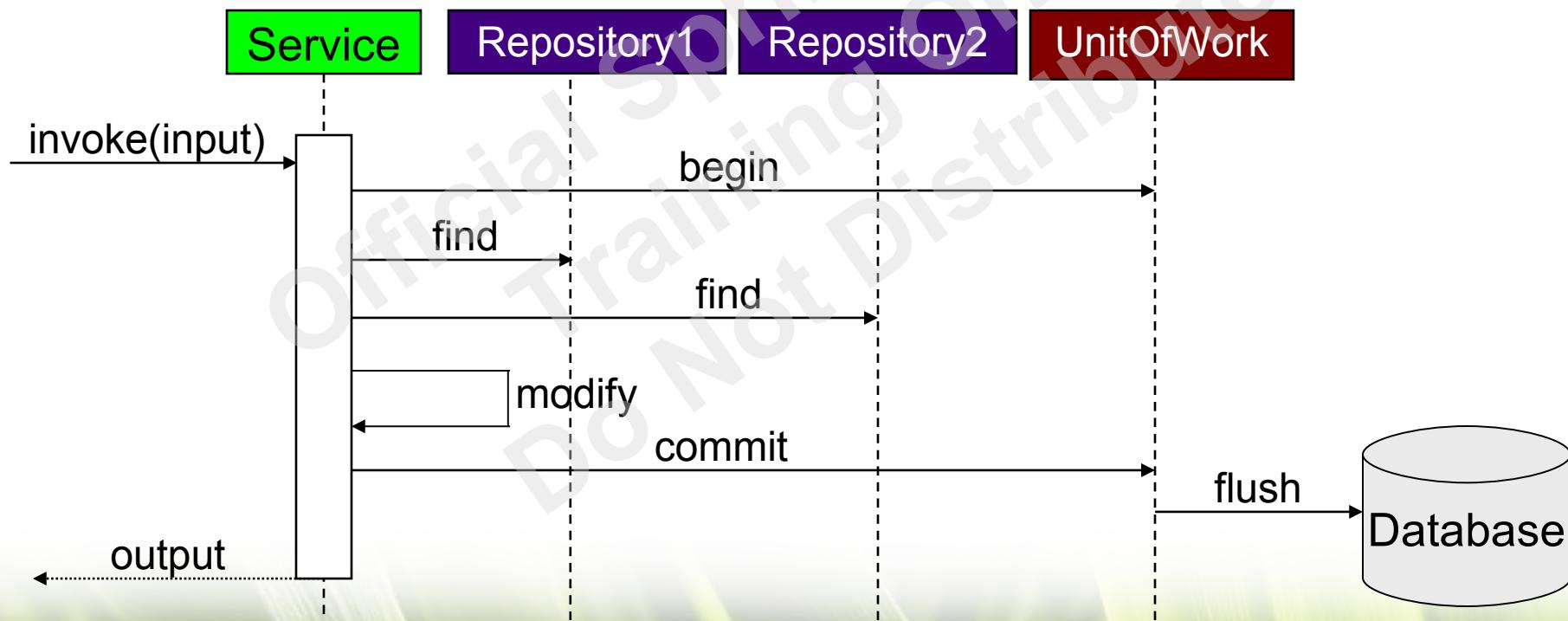
- Query with object properties and associations:

```
SELECT c FROM Customer c WHERE c.address.zipCode = 12345
```

Automatic Change Detection



- When a unit-of-work completes, all modified state will be synchronized with the database.



Persistence by Reachability



- When a persistent object is being managed, other associated objects may become managed transparently:

```
Order order = orderRepository.findByConfirmationId(cid);  
// order is now a managed object – retrieved via ORM
```

```
LineItem item = new LineItem(..);  
order.addLineItem(item);  
// item is now a managed object – reachable from order
```

(Un)Persistence by Reachability

= Make Transient



- The same concept applies for deletion:

```
Order order = orderRepository.findById(cid);  
// order is now a managed object – retrieved via ORM
```

```
List<LineItem> items = order.getLineItems();  
for (LineItem item : items) {  
    if (item.isCancelled()) { order.removeItem(item); }  
    // the database row for this item will be deleted  
}
```

Item becomes transient

```
if (order.isCancelled()) {  
    orderRepository.remove(order);  
    // all item rows for the order will be deleted  
}
```

Order and all its
items now transient

Caching

- The first-level cache (1LC) is scoped at the level of a unit-of-work
 - When an object is first loaded from the database within a unit-of-work it is stored in this cache
 - Subsequent requests to load that same entity from the database will hit this cache first
- The second-level cache (2LC) is scoped at the level of the SessionFactory
 - Reduce trips to database for read-heavy data
 - Especially useful when a single application has exclusive access to the database

Summary

- Managing persistent objects is hard
 - Especially if caching is involved
 - Especially on a shared, legacy schema with existing applications
- The ORM overcomes *some* of these problems
 - Automatic change detection, queries, caching
 - Ideal if your application *owns* its database
 - It is *not* a magic-bullet
 - JDBC may still be better for some tables/queries
 - True distributed cache coherency is *very* hard
 - *Design* for it and *test* performance



Object/Relational Mapping with Spring and Hibernate

ORM Simplification using Spring

Topics in this session



- **Introduction to Hibernate**
 - **Mapping**
 - **Querying**
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- Exception Mapping

Introduction to Hibernate

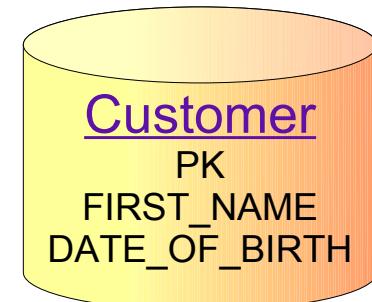
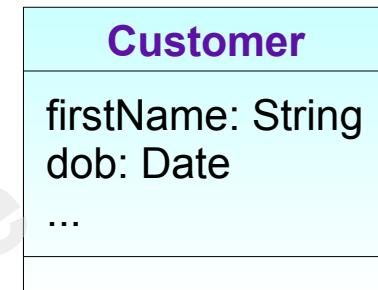


- A **SessionFactory** is a thread-safe, shareable object that represents a *single* data source
 - Provides access to a transactional Session
 - Ideal candidate for a singleton Spring bean
- Hibernate's **Session** is a stateful object representing a unit-of-work
 - Often corresponds at a higher-level to a JDBC Connection
 - Manages persistent objects within the unit-of-work
 - Acts as a transaction-scoped cache (1LC)



Hibernate Mapping

- Hibernate requires metadata
 - for mapping classes to database tables
 - and their properties to columns
- Metadata can be annotations or XML
 - This session will present JPA annotations
 - XML shown in the appendix



Annotations support in Hibernate



- Hibernate supports all JPA 2 annotations
 - Since Hibernate 3.5
 - javax.persistence.*
 - Suitable for most needs
- Hibernate Specific Extensions
 - In *addition* to JPA annotations
 - For behavior not supported by JPA
 - Performance enhancements specific to Hibernate
 - Less important since JPA 2



What can you Annotate?



- Classes
 - Applies to the entire class (such as table properties)
- Data-members
 - Typically mapped to a column
 - By default, *all* data-members treated as persistent
 - Mappings will be defaulted
 - Unless annotated with `@Transient` (non-persistent)
 - All data-members accessed directly
 - No accessors (getters/setters) needed
 - Any accessors will not be used
 - This is called “field” access

Mapping simple *data-members*: Field Access



```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    @Id  
    @Column (name="cust_id")  
    private Long id;  
  
    @Column (name="first_name")  
    private String firstName;  
  
    @Transient  
    private User currentUser;  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
...}
```

Only *@Entity* and *@Id* are mandatory

Mark as an *entity*
Optionally override
table name

Mark *id-field* (primary
key)

Optionally override
column names

Not stored in database

Setters and *not* used by
Hibernate

Mapping simple Properties

- Traditional Approach



Must place @Id on the *getter* method

Other annotations now also placed on *getter* methods

```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    private Long id;  
    private String firstName;  
  
    @Id  
    @Column (name="cust_id")  
    public String getId()  
    { return this.id; }  
  
    @Column (name="first_name")  
    public String getFirstName()  
    { return this.firstName; }  
  
    public void setFirstName(String name)  
    { this.firstName = name; }  
}
```



Beware of Side-Effects

getter/setter methods may do additional work – such as invoking listeners

Mapping collections with annotations



```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    @Id  
    @Column (name="cust_id")  
    private Long id;  
  
    @OneToMany(cascade=CascadeType.ALL)  
    @JoinColumn (name="cust_id")  
    private Set<Address> addresses;  
    ...  
}
```

Propagate all operations to the underlying objects

Foreign key in Address table

JoinTable also supported, more commonly used in
@ManyToMany associations

Accessing Persistent Data



- Hibernate's key class is the **Session**
 - Provides methods to manipulate entities
 - persist, delete, get ...
 - Create queries using Hibernate Query Language
 - Manages transactions
 - JPA defines a similar class: *EntityManager*
- Sessions more popular than EntityManagers
 - Already widely used before JPA
 - Common to write hybrid applications
 - JPA annotations, Hibernate API and HQL
 - But remember: it is not a JPA application



Hibernate Querying



- Hibernate provides several options for accessing data
 - Retrieve an object by primary key
 - Query for objects with the Hibernate Query Language (HQL)
 - Query for objects using Criteria Queries
 - Execute standard SQL

Hibernate Querying: by primary key



- To retrieve an object by its database identifier simply call get(..) on the Session

```
Long custId = new Long(123);
Customer customer = (Customer) session.get(Customer.class, custId);
```

↑
returns **null** if no object exists for the identifier

Hibernate Querying: HQL



- To query for objects based on properties or associations use HQL
 - Pre Java 5 API, not type aware

```
Query query = session.createQuery(  
    "from Customer c where c.address.city = :city");  
query.setString("city", "Chicago");  
List customers = query.list();
```

No generics

```
// Or if expecting a single result  
Customer customer = (Customer) query.uniqueResult();
```

Must cast

Topics in this session



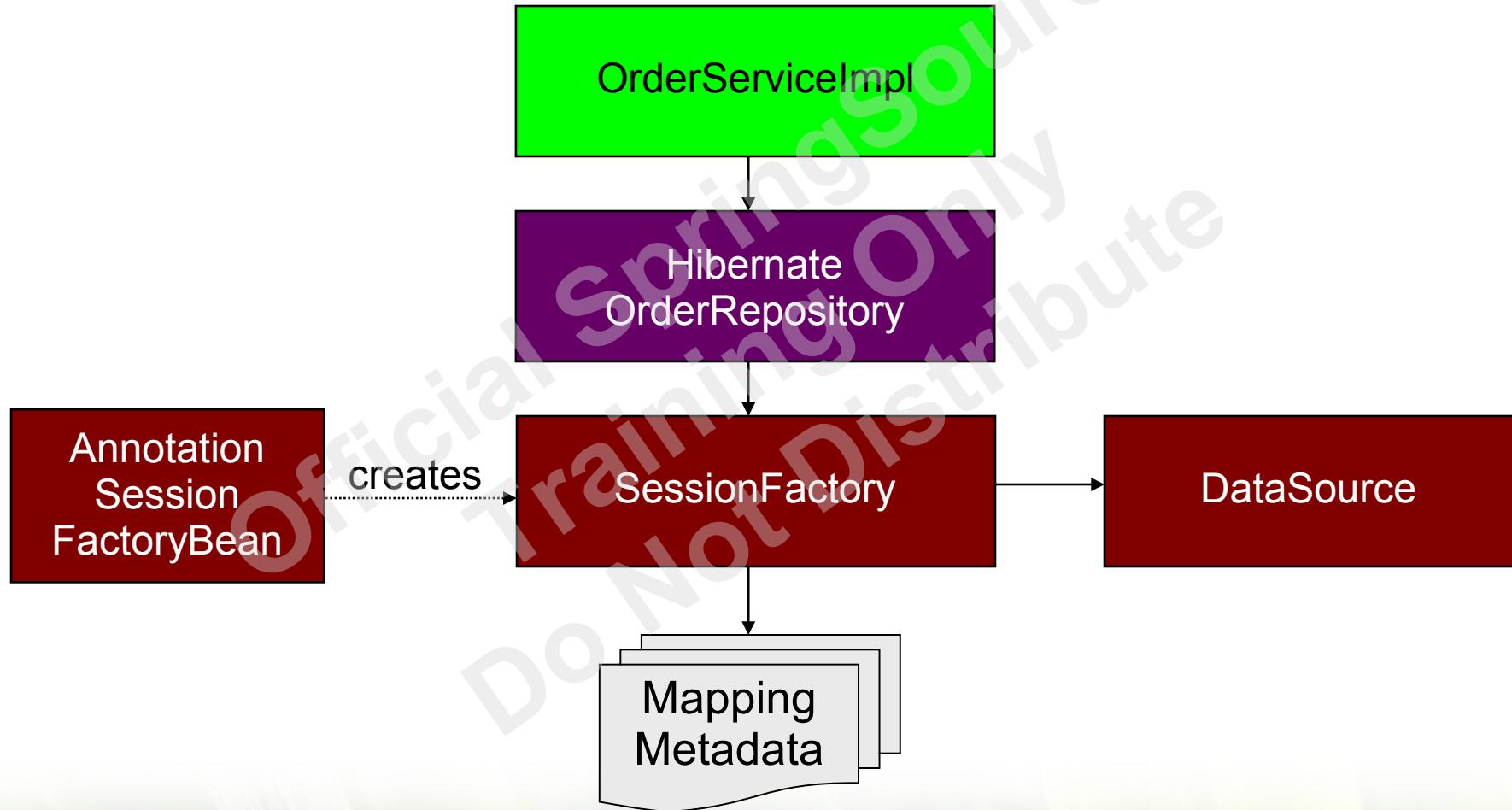
- Introduction to Hibernate
- **Configuring a Hibernate SessionFactory**
- Implementing Native Hibernate DAOs
- Exception Mapping

Configuring a SessionFactory (1)



- Hibernate implementations of data access code require access to the SessionFactory
- The SessionFactory requires
 - DataSource (local or container-managed)
 - Mapping metadata
- Spring provides a FactoryBean for configuring a shareable SessionFactory
 - Supports Hibernate V4 from Spring 3
 - There are different versions in hibernate2, hibernate3, hibernate4 packages

Configuring a SessionFactory (2)



SessionFactory and annotated classes



Spring 3.1

```
<bean id="orderRepository" class="example.HibernateOrderRepository">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="sessionFactory" class="org.springframework.orm.
    hibernate4.annotation.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>

    <property name="annotatedClasses">
        <list>
            <value>example.Customer</value>
            <value>example.Address</value>
        </list>
    </property>
</bean>

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/orders"/>
```

Entities listed one by one
- wildcards *not* supported

Prior to Spring 3.1 use the AnnotationSessionFactoryBean

SessionFactory and scanned packages



Spring 3.1

- Or use **packagesToScan** attribute
 - Wildcards are supported
 - Also scans all sub-packages

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate4.  
annotation.LocalSessionFactoryBean">  
    <property name="dataSource" ref="dataSource"/>  
    <property name="packagesToScan">  
        <list>  
            <value>com/springsource/*/entity</value>  
        </list>  
    </property>  
</bean>  
  
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/orders"/>
```

Topics in this session



- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- **Implementing Native Hibernate DAOs**
- Exception Mapping

Implementing Native Hibernate DAOs (1)



- Since Hibernate 3.1+
 - Hibernate provides hooks so Spring can manage transactions and Sessions in a transparent fashion
 - Use AOP for transparent exception translation to Spring's **DataAccessException** hierarchy
- No dependency on Spring in your DAO implementations
 - Used to require **HibernateTemplate** (obsolete)

Spring-managed Transactions and Sessions (1)



- Transparently participate in Spring-driven transactions
 - use one of Spring's Factory Beans to build the **SessionFactory**
- Provide it to the data access code
 - dependency injection
- Define a transaction manager
 - **HibernateTransactionManager**
 - **JtaTransactionManager**

Spring-managed Transactions and Sessions (2)



- The code – with no Spring dependencies

```
public class HibernateOrderRepository implements OrderRepository {  
    private SessionFactory sessionFactory;  
  
    public HibernateOrderRepository(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
    ...  
    public Order findOrderId(long orderId) {  
        // use the Spring-managed Session  
        Session session = this.sessionFactory.getCurrentSession();  
        return (Order) session.get(Order.class, orderId);  
    }  
}
```

dependency injection

use existing session

Spring-managed Transactions (3)



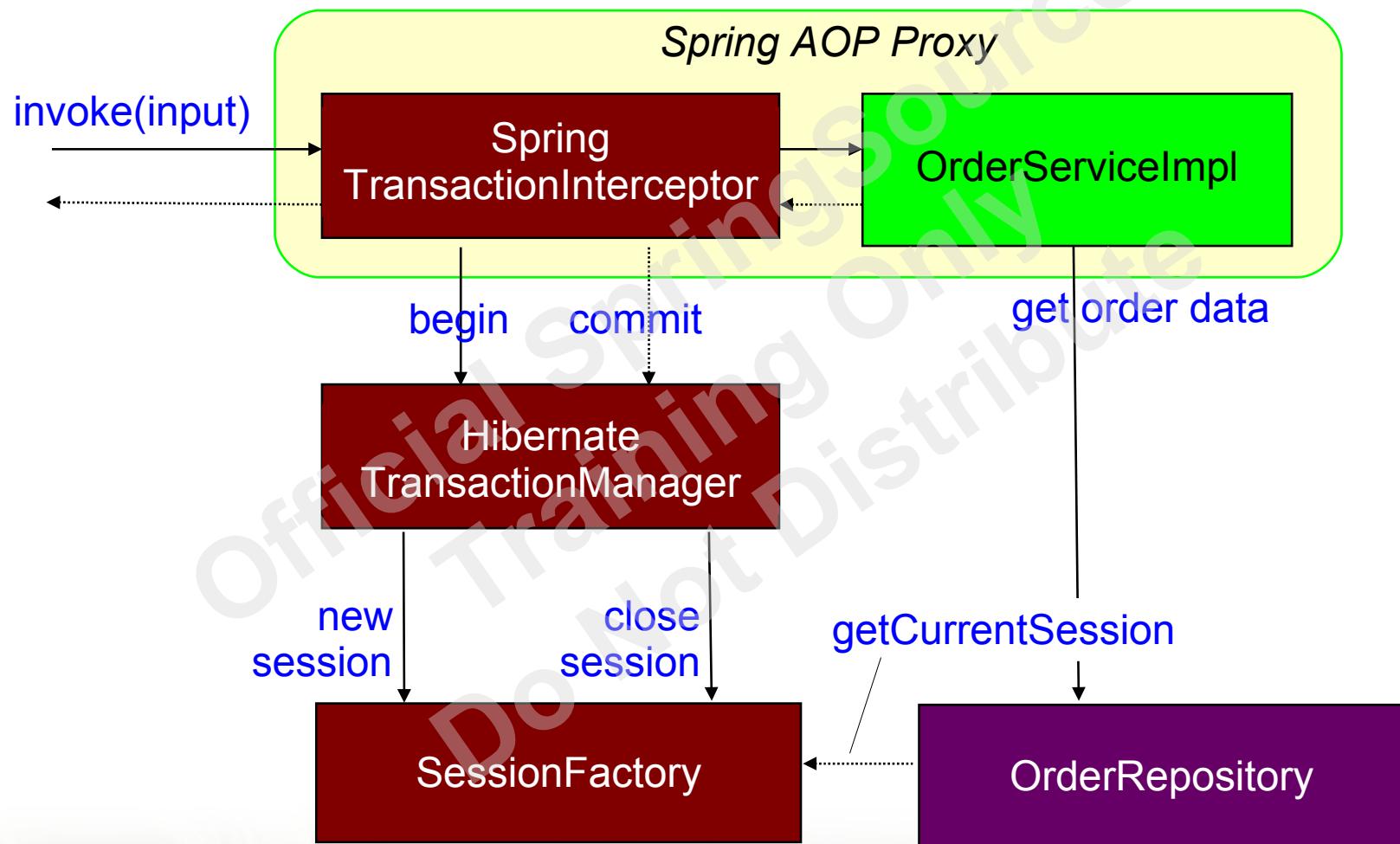
- The configuration

```
<beans>
  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.
      annotation.AnnotationSessionFactoryBean">
    ...
  </bean>
  <bean id="orderRepository" class="HibernateOrderRepository">
    <constructor-arg ref="sessionFactory"/>
  </bean>
  <bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>
</beans>
```

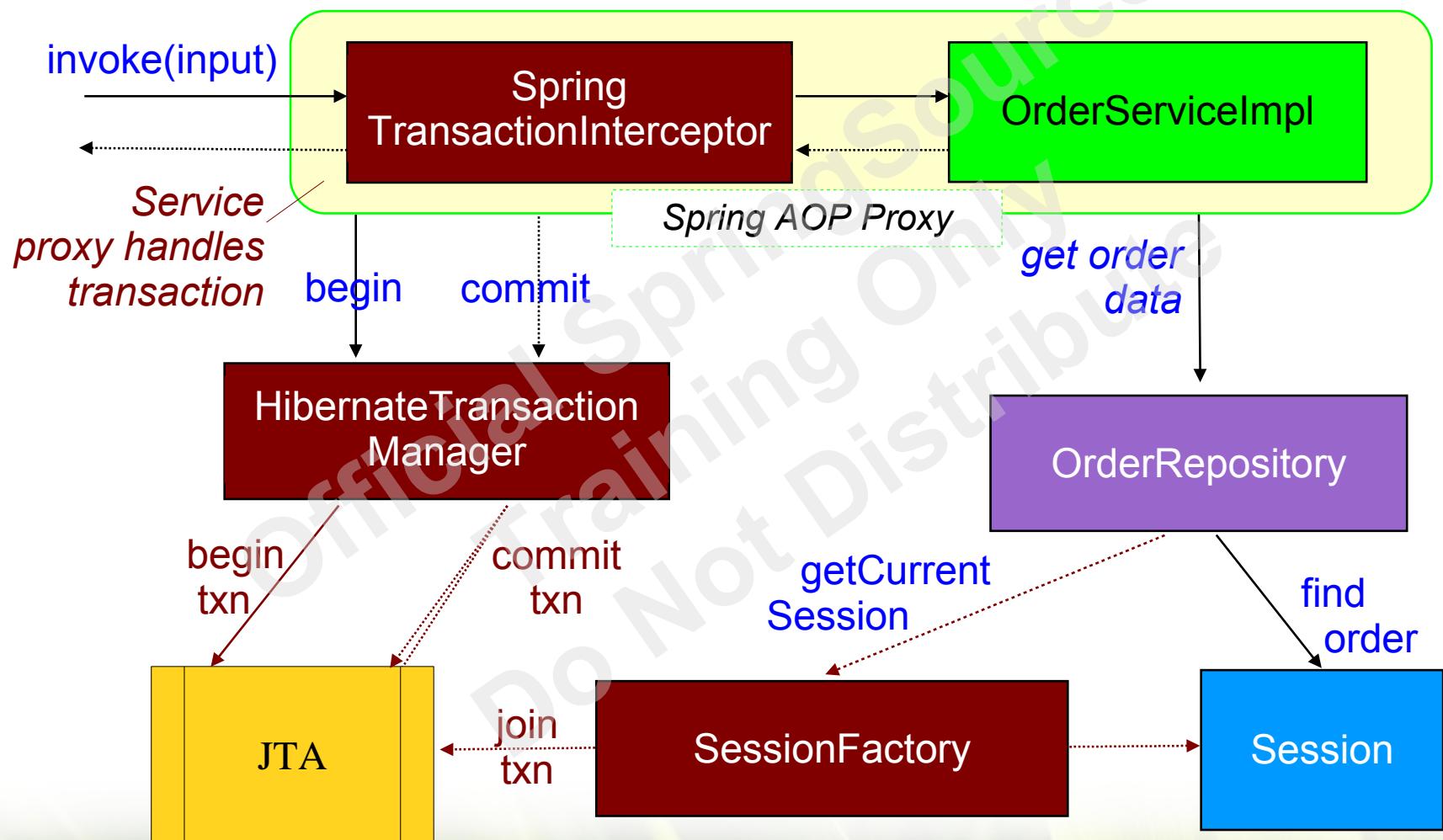
could use **JtaTransactionManager** if needed

A diagram consisting of two arrows pointing upwards from the bottom right towards the 'ref="sessionFactory"' attribute in the transactionManager bean's configuration. This visualizes the dependency relationship between the two beans.

How it Works (Hibernate)



How it Works (JTA)



Spring-managed Transactions (4)



- Best practice: use read-only transactions when you *don't* write anything to the database
- Prevents Hibernate from flushing its session
 - possibly dramatic performance improvement
- Also marks JDBC Connection as read-only
 - provides additional safeguards with some databases
 - for example: Oracle only accepts SELECT statements

```
@Transactional(readOnly=true)
public List<RewardConfirmation> listRewardsFrom(Date d) {
    // read-only, atomic unit-of-work
}
```

Topics in this session



- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- **Exception Mapping**

Transparent Exception Translation



- Used as-is, the previous DAO implementation can throw an access specific **HibernateException**
 - Propagate up to the caller
 - service layer or other users of the DAOs
 - Introduces dependency on specific persistence solution that should not exist
 - Service layer “knows” you are using Hibernate
- Can use AOP to convert them to Spring’s rich **DataAccessException** hierarchy
 - Hides access technology used

Exception Translation

– Using @Repository



Spring provides this capability out of the box

- Annotate with **@Repository**
- Define a Spring-provided BeanPostProcessor

```
@Repository  
public class HibernateOrderRepository implements OrderRepository {  
    ...  
}
```

```
<bean class="org.springframework.dao.annotation.  
PersistenceExceptionTranslationPostProcessor"/>
```

Exception Translation - XML



- Can't always use annotations
 - For example with a third-party repository
 - Use XML to configure instead

```
public class HibernateOrderRepository implements OrderRepository {  
    ...  
}
```

No annotations

```
<bean id="persistenceExceptionInterceptor"  
      class="org.springframework.dao.support.  
          PersistenceExceptionTranslationInterceptor"/>
```

```
<aop:config>  
    <aop:advisor pointcut="execution(* *..OrderRepository+.*(..))"  
                  advice-ref="persistenceExceptionInterceptor" />  
</aop:config>
```

Summary

- Use Hibernate and/or JPA to define entities
 - Repositories have no Spring dependency
- Use Spring to configure Hibernate session factory
 - Works with Spring-driven transactions
 - Optional translation to DataAccessExceptions

Spring-Hibernate – 3 day in-depth Hibernate course



LAB

Using Hibernate with Spring

Topics in this session

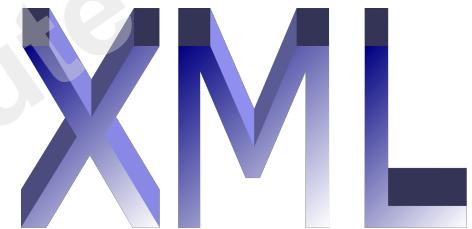


- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- Exception Mapping
- **Appendix**
 - **XML Metadata**
 - Configuration
 - HibernateTemplate

Metadata in XML



- JPA XML
 - JPA defines an XML syntax (orm.xml)
 - In practice rarely used
 - Most JPA users, use annotations
- Hibernate XML
 - Predates annotations and JPA
 - Still extensively used in older applications
 - Will be shown here



Mapping simple properties with Hibernate XML



```
public class Customer {  
    → private Long id;  
    private String firstName;  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
    ...  
}
```

Different defaults:

1. Property access is the default
2. Data-members not in XML are *transient*

‘property’ access (default) uses setter

```
<hibernate-mapping package="org.xyz.customer">  
    <class name="Customer" table="T_CUSTOMER">  
        → <id name="id" column="cust_id" access="field"/>  
        <property name="firstName" column="first_name"/>  
    </class>  
</hibernate-mapping>
```

‘field’ access

Suggestion: put default-access="field" in <class>

Mapping collections with XML



```
public class Customer {  
    private Long id;  
    → private Set addresses;  
    ...
```

```
<class name="Customer" table="T_CUSTOMER">  
    ...  
    → <set name="addresses" cascade="all" access="field">  
        <key column="CUST_ID"/> ← Foreign key in  
        <one-to-many class="Address"/>  
    </set>  
</class>
```

Foreign key in
Address table

Topics in this session



- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- **Appendix**
 - XML Metadata
 - **Configuration**
 - HibernateTemplate

Configuring a SessionFactory with XML Metadata



- Configure using **LocalSessionFactoryBean**
 - Define XML mapping-locations

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingLocations">
        <list>
            <value>classpath:example/Customer.hbm.xml</value>
            <value>classpath:example/Address.hbm.xml</value>
        </list>
    </property>
</bean>
```

Wildcards are supported

Prior to Spring 3.1, use class in hibernate3 package

Mixing Metadata

- Can use annotations and XML together
- Why do this?
 - Migration
 - annotate new classes, old classes still use XML
 - Some features better defined using XML
 - filters, named-queries
- LocalSessionFactoryBean supports both
 - Use the **AnnotationSessionFactoryBean** prior to Spring 3.1
 - Sub-classes **LocalSessionFactory**
 - Functionality now moved to **LocalSessionFactory**

Configuring a SessionFactory with Both



Spring 3.1

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.annotation.LocalSessionFactoryBean">

    <property name="dataSource" ref="dataSource"/>
    <property name="annotatedClasses">
        <list>
            <value>example.Customer</value>
            <value>example.Address</value>
        </list>
    </property>
    <property name="mappingLocations">
        <list>
            <value>classpath:example/Suppliers.hbm.xml</value>
            <value>classpath:example/Filters.hbm.xml</value>
            <value>classpath:example/NamedQueries.hbm.xml</value>
        </list>
    </property>
</bean>
```

Can use
annotations *and*
XML in same
application

Prior to Spring 3.1, use AnnotationSessionFactoryBean

Topics in this session



- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- **Appendix**
 - XML Metadata
 - Configuration
 - **HibernateTemplate**

- Consistent with Spring's general DAO support
 - Manages resources (acquiring/releasing Sessions)
 - Provides convenience methods
 - `findBy*`, `getSession`, ...
 - Throws `DataAccessExceptions`
- Native approach recommended since
Hibernate 3.1+
 - Preferred by most Hibernate & Spring developers
 - No Spring dependencies in DAO
 - *HibernateTemplate no longer recommended*

Topics Covered



- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- Appendix: XML

Spring-Hibernate – 3 day in-depth Hibernate course



Spring Web Services

Implementing Loosely Coupled Communication
with Spring Web Services

Topics in this Session

- **Introduction to Web Services**
 - Why use or build a web service?
 - Best practices for implementing a web service
- Spring Web Services
- Client access

Web Services enable *Loose Coupling*



*"Loosely coupled systems are considered useful when either the **source** or the **destination** computer systems are subject to frequent **changes**"*

Wikipedia (July 2007)

**Loose coupling increases tolerance...
changes should not cause incompatibility**

Web Services enable *Interoperability*



- XML is the lingua franca in the world of interoperability
- XML is understood by all major platforms
 - SAX, StAX or DOM in Java
 - System.XML or .NET XML Parser in .NET
 - REXML or XmlSimple in Ruby
 - Perl-XML or XML::Simple in Perl

Best practices for implementing web services



- Remember:
 - web services != SOAP
 - web services != RPC
- Design contract independently from service interface
- Refrain from using stubs and skeletons
- Consider skipping validation for incoming requests...
- ... and using Xpath to only extract what you need

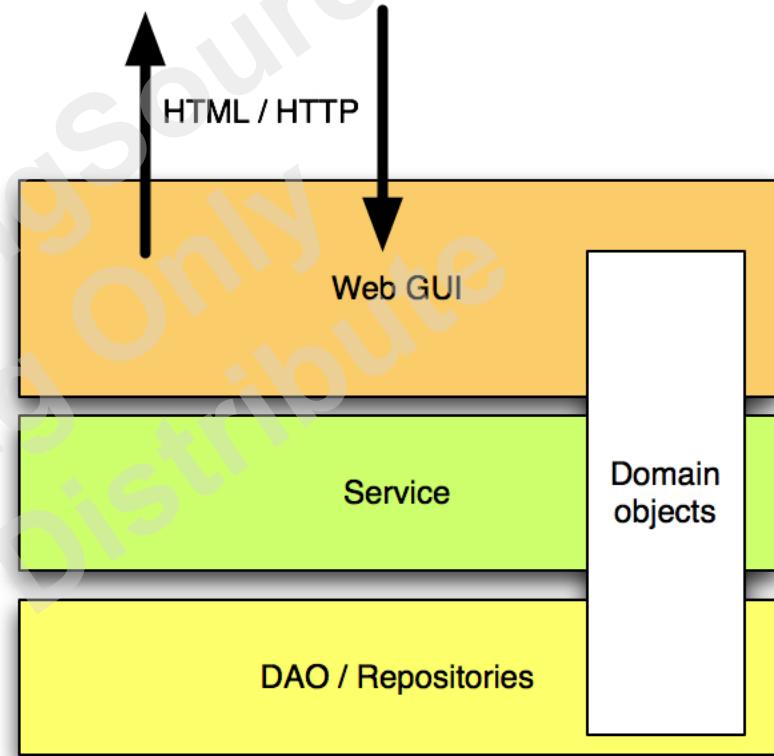
↑
Postel's law:

“Be conservative in what you do;
be liberal in what you accept from others.”

Web GUI on top of your services



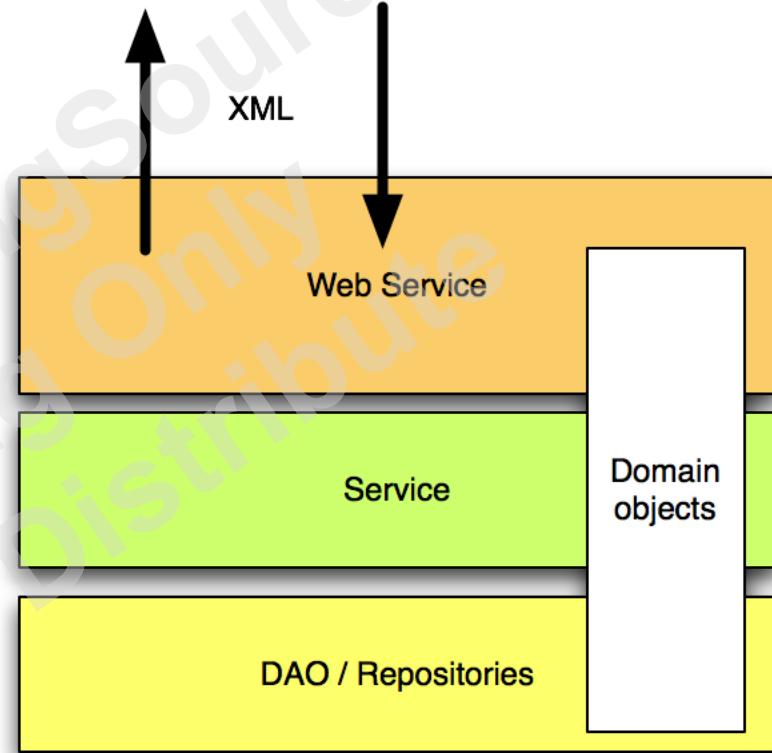
The **Web GUI layer** provides **compatibility** between **HTML-based** world of the user (the browser) and the **OO-based** world of your service



Web Service on top of your services



Web Service layer
provides **compatibility**
between **XML-based**
world of the user and
the **OO-based** world of
your service



Topics in this Session

- Introduction to Web Services
 - Why use or build a web service?
 - Best practices for implementing a web service
- **Spring Web Services**
- Client access

Define the contract

- Spring-WS uses Contract-first
 - Start with XSD/WSDL
- Widely considered a Best Practice
 - Solves many interoperability issues
- Also considered difficult
 - But isn't

Contract-first in 3 simple steps



- Create sample messages
- Infer a contract
 - Trang
 - Microsoft XML to Schema
 - XML Spy
- Tweak resulting contract

Sample Message



Namespace for this message

```
<transferRequest xmlns="http://mybank.com/schemas/tr"
    amount="1205.15">
    <credit>S123</credit>
    <debit>C456</debit>
</transferRequest>
```

Define a schema for the web service message



```
<xs:schema  
    xmlns:xs="http://www.w3.org/2001/XMLSchema"  
    xmlns:tr="http://mybank.com/schemas/tr"  
    elementFormDefault="qualified"  
    targetNamespace="http://mybank.com/schemas/tr">  
    <xs:element name="transferRequest">  
        <xs:complexType>  
            <xs:sequence>  
                <xs:element name="credit" type="xs:string"/>  
                <xs:element name="debit" type="xs:string"/>  
            </xs:sequence>  
            <xs:attribute name="amount" type="xs:decimal"/>  
        </xs:complexType>  
    </xs:element>  
</xs:schema>
```

```
<transferRequest  
    amount="1205.15">  
    <credit>S123</credit>  
    <debit>C456</debit>  
</transferRequest>
```

Type constraints

```
<xs:element name="credit" >  
  <xs:simpleType>  
    <xs:restriction base="xs:string">  
      <xs:pattern value="\w\d{3}"/>  
    </xs:restriction>  
  </xs:simpleType>  
</xs:element>
```

1 Character + 3 Digits

SOAP Message



Envelope

Header

Security

Routing

Body

TransferRequest

Simple SOAP 1.1 Message Example

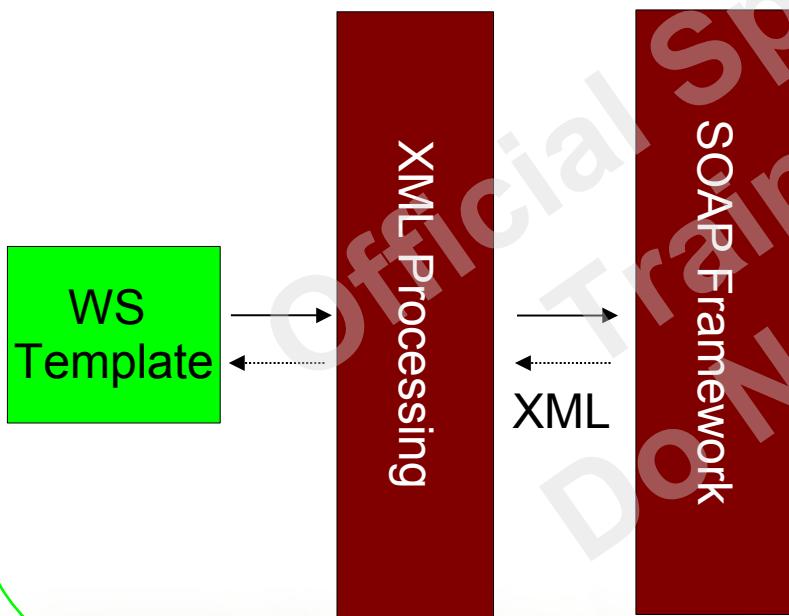


```
<SOAP-ENV:Envelope xmlns:SOAP-  
    ENV="http://schemas.xmlsoap.org/soap/envelope/">  
    <SOAP-ENV:Body>  
        <tr:transferRequest xmlns:tr="http://mybank.com/schemas/tr"  
            tr:amount="1205.15">  
            <tr:credit>S123</tr:credit>  
            <tr:debit>C456</tr:debit>  
        </tr:transferRequest>  
    </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

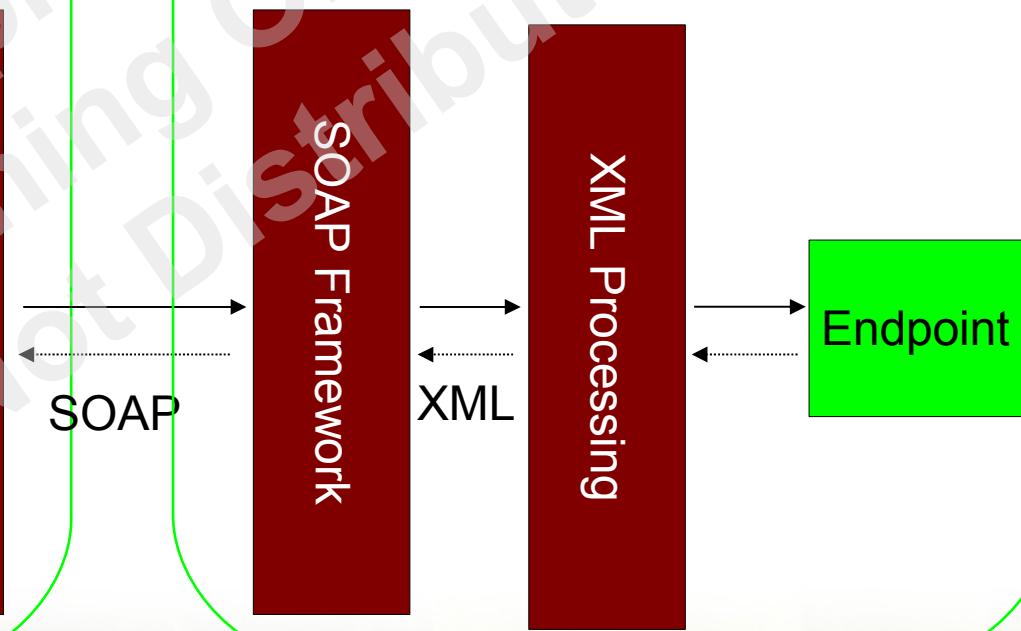
Spring Web Services



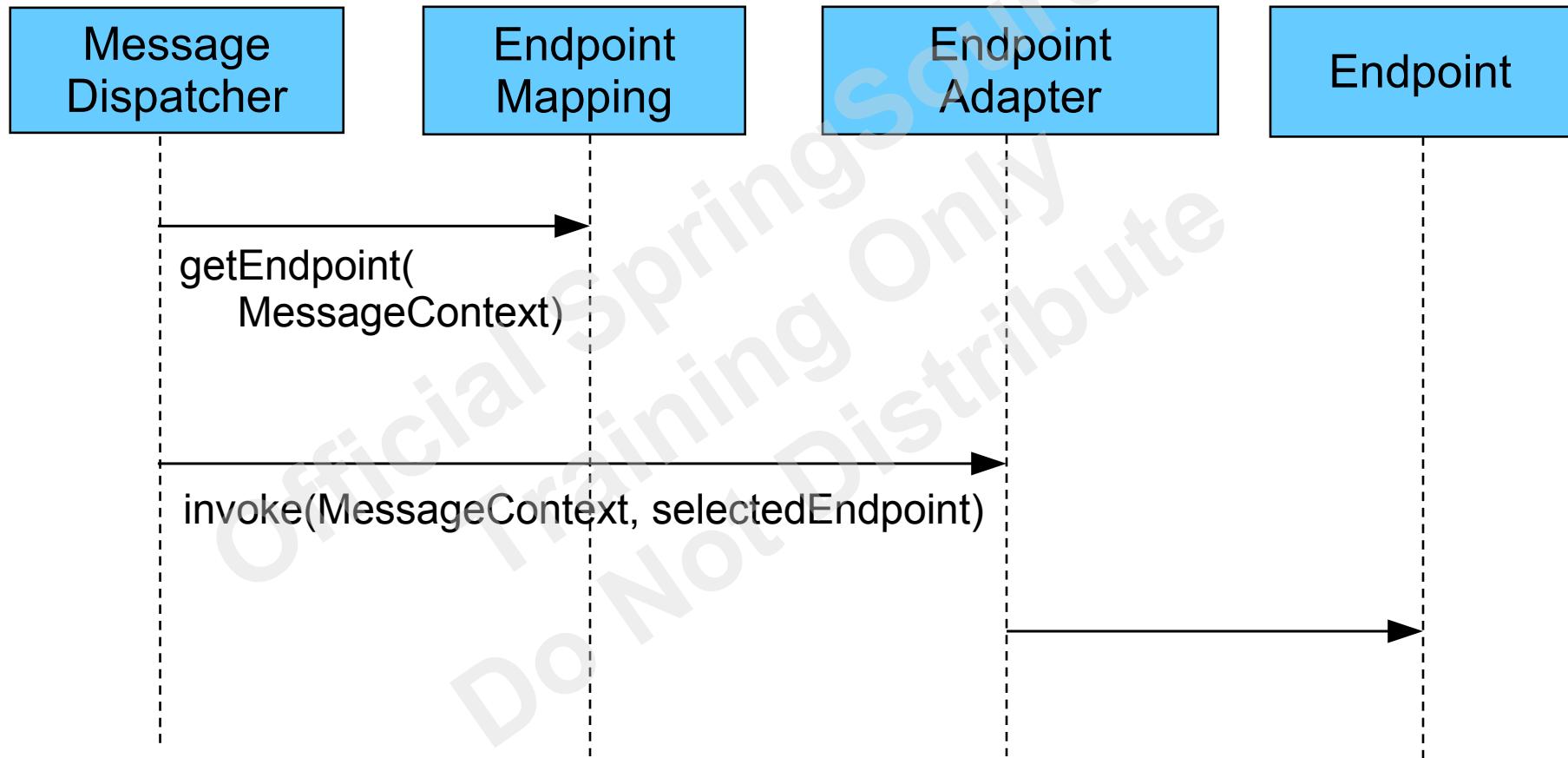
Client Process



Server Process



Request Processing



Bootstrap the application tier



- Inside <webapp/> within web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/transfer-app-cfg.xml
    </param-value>
</context-param>
```

The application context's configuration file(s)

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

Loads the ApplicationContext into the ServletContext
before any Servlets are initialized

Wire up the Front Controller (MessageDispatcher)



- Inside <webapp> within web.xml

```
<servlet>
    <servlet-name>transfer-ws</servlet-name>
    <servlet-class>..ws..MessageDispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/ws-config.xml</param-value>
    </init-param>
</servlet>
```

The application context's configuration file(s)
containing the web service infrastructure beans

Map the MessageDispatcherServlet



- Inside <webapp/> within web.xml

```
<servlet-mapping>
  <servlet-name>transfer-ws</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

There might also be a web interface (GUI) that is mapped to another path

Endpoint

- Endpoints handle SOAP messages
- Similar to MVC Controllers
 - Handle input message
 - Call method on business service
 - Create response message
- With Spring-WS you can focus on the Payload
- Switching from SOAP to POX without code change

XML Handling techniques



- Low-level techniques
 - DOM (JDOM, dom4j, XOM)
 - SAX
 - StAX
- Marshalling
 - JAXB (1 and 2)
 - Castor
 - XMLBeans
- XPath argument binding

- JAXB 2 is part of Java EE 5 and JDK 6
- Uses annotations for mapping metadata
- Generates classes from a schema and vice-versa
- Supported as part of Spring-OXM

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="rewards.ws.types"/>
```

- No explicit marshaller bean definition necessary to be used in Spring-WS 2.0 endpoints

```
<!-- registers all infrastructure beans needed for annotation-based  
     endpoints, incl. JAXB2 (un)marshalling: -->  
<ws:annotation-driven/>
```

Implement the Endpoint



```
@Endpoint ← Spring WS Endpoint
public class TransferServiceEndpoint {
    private TransferService transferService;
    @Autowired
    public TransferServiceEndpoint(TransferService transferService) {
        this.transferService = transferService;
    }
    @PayloadRoot(localPart="transferRequest",
                namespace="http://mybank.com/schemas/tr")
    public @ResponsePayload TransferResponse newTransfer(
        @RequestPayload TransferRequest request) {
        // extract necessary info from request and invoke service
    }
}
```

The diagram illustrates the flow of data through the endpoint implementation. It starts with the `@Endpoint` annotation, which is associated with the `Spring WS Endpoint`. An arrow points from the `@PayloadRoot` annotation to a box labeled `Mapping`. Another arrow points from the `TransferResponse` return type of the method to a box labeled `Converted with JAXB2`. A third arrow points from the `TransferRequest` parameter to the same `Converted with JAXB2` box, indicating that the request is converted to XML and mapped to the response object.

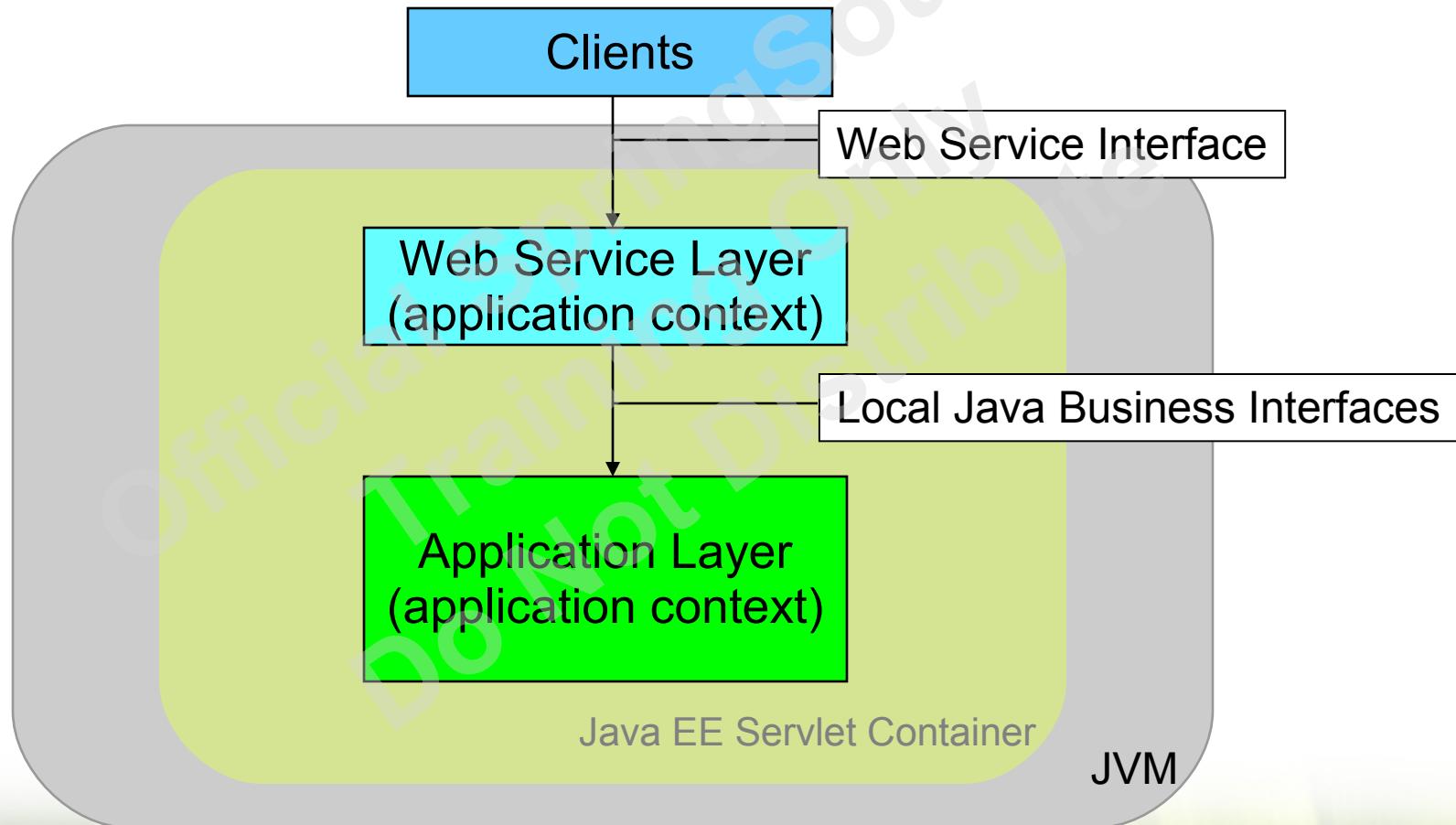
Configure Spring-WS and Endpoint beans



- @Endpoint annotated classes can be component-scanned
 - Saves writing lots of trivial XML bean definitions
 - Just make sure not to scan classes that belong in the root context instead

```
<!-- instead of defining explicit endpoints beans  
    we rely on component scanning: -->  
<context:component-scan base-package="transfers.ws"/>  
  
<!-- registers all infrastructure beans needed for  
    annotation-based endpoints, incl. JAXB2 (un)marshalling:  
-->  
<ws:annotation-driven/>
```

Architecture of our application exposed using a web service



Further Mappings

- You can also map your Endpoints in XML by
 - Message Payload
 - SOAP Action Header
 - WS-Addressing
 - XPath

Publishing the WSDL



http://somehost:8080/transferService/transferDefinition.wsdl

```
<ws:dynamic-wsdl id="transferDefinition"
    portTypeName="Transfers"
    locationUri="http://somehost:8080/transferService/"
    <ws:xsd location="/WEB-INF/transfer.xsd"/>
</ws:dynamic-wsdl>
```

- Generates WSDL based on given XSD
 - id becomes part of WSDL URL
- Most useful during development
- Just use static WSDL in production
 - Contract shouldn't change unless YOU change it

Topics in this Session

- Introduction to Web Services
 - Why use or build a web service?
 - Best practices for implementing a web service
- Spring Web Services
- **Client access**

Spring Web Services on the Client



- **WebServiceTemplate**
 - Simplifies web service access
 - Works directly with the XML payload
 - Extracts *body* of a SOAP message
 - Also works with POX (Plain Old XML)
 - Can use marshallers/unmarshallers
 - Provides convenience methods for sending and receiving web service messages
 - Provides callbacks for more sophisticated usage

Marshalling with WebServiceTemplate



```
<bean id="webServiceTemplate"
      class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="defaultUri" value="http://mybank.com/transfer"/>
    <property name="marshaller" ref="marshaller"/>
    <property name="unmarshaller" ref="marshaller"/>
</bean>

<bean id="marshaller" class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation" value="classpath:castor-mapping.xml"/>
</bean>
```

```
WebServiceTemplate ws = context.getBean(WebServiceTemplate.class);
TransferRequest request = new TransferRequest("S123", "C456", "85.00");
Receipt receipt = (Receipt) ws.marshalingSendAndReceive(request);
```



LAB

Exposing SOAP Endpoints using
Spring Web Services



Practical REST services with Spring MVC

Using Spring MVC to build RESTful web
services

Topics in this Session

- **REST introduction**
- REST and Java
- Spring MVC support for RESTful applications
 - Request/Response Processing
 - Using MessageConverters
 - Content Negotiation
 - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

REST Introduction

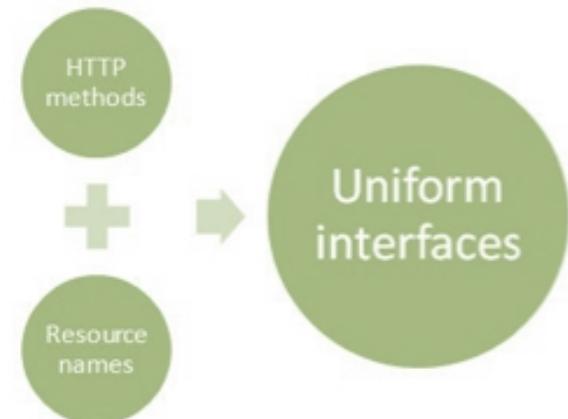


- Web apps not just usable by browser clients
 - Programmatic clients can also connect via HTTP
 - Such as: mobile applications, AJAX enabled web-pages
- REST is an *architectural style* that describes best practices to expose web services over HTTP
 - REpresentational State Transfer, term by Roy Fielding
 - HTTP as *application* protocol, not just transport
 - Emphasizes scalability
 - *Not* a framework or specification

REST Principles (1)



- Expose *resources* through URIs
 - Model nouns, not verbs
 - <http://mybank.com/banking/accounts/123456789>
- Resources support limited set of operations
 - GET, PUT, POST, DELETE in case of HTTP
 - All have well-defined semantics
- Example: update an order
 - PUT to </orders/123>
 - don't POST to </order/edit?id=123>



REST Principles (2)



- Clients can request particular representation
 - Resources can support multiple representations
 - HTML, XML, JSON, ...
- Representations can link to other resources
 - Allows for extensions and discovery, like with web sites
- Hypermedia As The Engine of Application State
 - *HATEOAS*: Probably the world's worst acronym!
 - RESTful responses contain the links you need – just like HTML pages do

REST Principles (3)



- Stateless architecture
 - No HttpSession usage
 - GETs can be cached on URL
 - Requires clients to keep track of state
 - Part of what makes it scalable
 - Looser coupling between client and server
- HTTP headers and status codes communicate result to clients
 - All well-defined in HTTP Specification

Why REST?

- Benefits of REST
 - Every platform/language supports HTTP
 - Unlike for example SOAP + WS-* specs
 - Easy to support many different clients
 - Scripts, Browsers, Applications
 - Scalability
 - Support for redirect, caching, different representations, resource identification, ...
 - Support for XML, but also other formats
 - JSON and Atom are popular choices



Topics in this Session

- REST introduction
- **REST and Java**
- Spring MVC support for RESTful applications
- RESTful clients with the RestTemplate
- Conclusion

- JAX-RS is a Java EE 6 standard for building RESTful applications
 - Focuses on programmatic clients, not browsers
- Various implementations
 - Jersey (RI), RESTEasy, Restlet, CXF
 - All implementations provide Spring support
- Good option for full REST support using a standard
- No support for building clients in standard
 - Although some implementations do offer it

- Spring-MVC provides REST support as well
 - Since version 3.0
 - Using familiar and consistent programming model
 - Spring MVC does not implement JAX-RS
- Offers both programmatic client support (HTTP-based web services) and browser support (RESTful web applications)
- Includes RestTemplate for building programmatic clients in Java

Topics in this Session

- REST introduction
- REST and Java
- **Spring MVC support for RESTful applications**
 - Request/Response Processing
 - Using MessageConverters
 - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

Spring-MVC and REST



- Some features have been covered already
 - Map requests based on HTTP method
 - URI templates to parse 'RESTful' URLs
- Will now extend Spring MVC to support REST
 - More Annotations
 - Message Converters
- Support for RESTful web applications targeting browser-based clients is also offered
 - See HttpMethodFilter

Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
 - **Request/Response Processing**
 - Using MessageConverters
 - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

Request mapping based on HTTP method



- Can map HTTP requests based on method
 - Allows same URL to be mapped to multiple methods
 - Often used for form-based controllers (GET & POST)
 - Essential to support RESTful resource URLs
 - incl. PUT and DELETE

```
@RequestMapping(value="/orders", method=RequestMethod.GET)
public void listOrders(Model model) {
    // find all Orders and add them to the model
}
```

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
public void createOrder(HttpServletRequest request, Model model) {
    // process the order data from the request
}
```

- Web apps just use a handful of status codes
 - Success: 200 OK
 - Redirect: 302/303 for Redirects
 - Client Error: 404 Not Found
 - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many *additional* codes to communicate with their clients
- Add `@ResponseStatus` to controller method
 - don't have to set status on `HttpServletResponse` manually

Common Response Codes

200

- ◆ After a successful GET returning content

201

- ◆ New resource was created on POST or PUT
- ◆ Location header contains its URI

204

- ◆ The response is empty – such as after successful PUT or DELETE

404

- ◆ Requested resource was not found

405

- ◆ HTTP method is not supported by resource

409

- ◆ Conflict while making changes – such as POSTing unique data that already exists

415

- ◆ Request body type not supported

@ResponseStatus

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public void createOrder(HttpServletRequest request,
                        HttpServletResponse response) {
    Order order = createOrder(request);
    // determine full URI for newly created Order based on request
    response.addHeader("Location",
                       getLocationForChildResource(request, order.getId()));
}
```

- **IMPORTANT:** When using `@ResponseStatus`
 - `void` methods *no longer* imply a default view name!
 - There will be no View at all
 - This example gives a response with an *empty* body

Determining Location Header



- Location header value must be full URL
 - Determine based on request URL
 - Controller shouldn't know host name or servlet path
- URL of created child resource usually a sub-path
 - POST to <http://www.myshop.com/store/orders> gives <http://www.myshop.com/store/orders/123>
 - Can use Spring's UriTemplate for encoding where needed

```
private String getLocationForChildResource(HttpServletRequest request,  
                                         Object childIdentifier) {  
    StringBuffer url = request.getRequestURL();  
    UriTemplate template = new UriTemplate(url.append("/{childId}").toString());  
    return template.expand(childIdentifier).toASCIIString();  
}
```

- Can also annotate exception classes with this
 - Given status code used when exception is thrown from controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
public class OrderNotFoundException extends RuntimeException {
    ...
}
```

```
@RequestMapping(value="/orders/{id}", method=GET)
public String showOrder(@PathVariable("id") long id, Model model) {
    Order order = orderRepository.findOrderById(id);
    if (order == null) throw new OrderNotFoundException(id);
    model.addAttribute(order);
    return "orderDetail";
}
```

@ExceptionHandler

- For existing exceptions you cannot annotate, use @ExceptionHandler method in controller
 - Method signature similar to request handling method
 - Also supports @ResponseStatus

```
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler({DataIntegrityViolationException.class})
public void conflict() {
    // could add the exception, response, etc. as method params
}
```

Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
 - Request/Response Processing
 - **Using MessageConverters**
 - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

HttpMessageConverter

- Converts between HTTP request/response and object
- Various implementations registered by default when using `<mvc:annotation-driven/>`
 - XML (using JAXP Source or JAXB2 mapped object*)
 - Feed data*, i.e. Atom/RSS
 - Form-based data
 - JSON*
 - Byte[], String, BufferedImage
- Define HandlerAdapter explicitly to register other HttpMessageConverters

* Requires 3rd party libraries on classpath



You must use `<mvc:annotation-driven/>` or register custom converters to have any HttpMessageConverters defined at all!

@RequestBody

- Annotate method parameter with **@RequestBody**
 - Enables converters for *request* data
 - Right converter chosen automatically
 - Based on content type of request
 - Order could be mapped from XML with JAXB2 or from JSON with Jackson, for example

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.PUT)
@ResponseBody(HttpStatus.NO_CONTENT) // 204
public void updateOrder(@RequestBody Order updatedOrder,
                       @PathVariable("id") long id) {
    // process updated order data and return empty response
    orderManager.updateOrder(id, updatedOrder);
}
```

@ResponseBody

- Use converters for *response* data by annotating method with **@ResponseBody**
 - Converter handles rendering to response
 - No ViewResolver and View involved anymore!

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseBody Order getOrder(@PathVariable("id") long id) {
    // Order class is annotated with JAXB2's @XmlRootElement
    Order order= orderRepository.findOrderById(id);
    // results in XML response containing marshalled order:
    return order;
}
```

Automatic Content Negotiation



- `HttpMessageConverter` selected automatically for `@ResponseBody`-annotated methods
 - Based on `Accept` header in request
`Accept: application/xml;`
 - Each converter has list of supported media types
- Allows multiple representations for a single controller method
 - Without affecting controller implementation
 - Alternative for content-based View selection as used without `@ResponseBody`

Content Negotiation Sample



```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseBody @ResponseStatus(HttpStatus.OK) // 200
public Order getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}
```

GET /store/orders/123
Host: www.myshop.com
Accept: application/xml, ...
...

HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type: application/xml
<order id="123">
...
</order>

GET /store/orders/123
Host: www.myshop.com
Accept: application/json, ...
...

HTTP/1.1 200 OK
Date: ...
Content-Length: 756
Content-Type: application/json
{
 "order": {"id": 123, "items": [...], ... }
}

- REST methods cannot return HTML, PDF, ...
 - No message converter
 - Views better for presentation-rich representations
- Need two methods on controller *for same URL*
 - One uses a converter, the other a view
 - Identify using **produces** attribute
- Use **consumes** to differentiate RESTful POST from an HTML form submission

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,  
    produces = {"application/json"})
```

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.POST,  
    consumes = {"application/json"})
```

Using Views and Annotations - 2



- Recommendation
 - Identify RESTful method with *produces*
 - Call RESTful method from View method
 - Implement all logic *once* in RESTful method

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,
    produces = {"application/json", "application/xml"})
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}

@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
public String getOrder(Model model, @PathVariable("id") long id) {
    model.addAttribute(getOrder(id)); // Use RESTful method
    return "orderDetails";
}
```

Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
 - Request/Response Processing
 - Using MessageConverters
 - **Putting it all together**
- RESTful clients with the RestTemplate
- Conclusion

Retrieving a Representation: GET



```
GET /store/orders/123  
Host: www.myshop.com  
Accept: application/xml, ...  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 1456  
Content-Type: application/xml  
<order id="123">  
...  
</order>
```

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)  
@ResponseStatus(HttpStatus.OK) // 200: this is the default  
public @ResponseBody Order getOrder(@PathVariable("id") long id) {  
    return orderRepository.findOrderById(id);  
}
```

Creating a new Resource: POST - 1



```
POST /store/orders/123/items  
Host: www.myshop.com  
Content-Type: application/xml  
<item>  
...  
</item>
```

```
HTTP/1.1 201 Created  
Date: ...  
Content-Length: 0  
Location:  
http://www.myshop.com/store/orders/123/items/abc
```

```
@RequestMapping(value="/orders/{id}/items", method=RequestMethod.POST)  
@ResponseStatus(HttpStatus.CREATED) // 201  
public void createItem(@PathVariable("id") long id,  
                        @RequestBody Item newItem,  
                        HttpServletRequest request,  
                        HttpServletResponse response) {  
    orderRepository.findOrderById(id).addItem(newItem); // adds id to Item  
    response.addHeader("Location",  
                      getLocationForChildResource(request, newItem.getId()));  
}
```

Creating a new Resource: POST - 2



- Simplify POST
 - Use @Value for request properties
 - Use HttpEntity instead of HttpServletResponse

```
@RequestMapping(value="/orders/{id}/items", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public HttpEntity<?> createItem(@PathVariable("id") long id,
    @RequestBody Item newItem,
    @Value("#{request.requestURL}") StringBuffer originalUrl) {
    orderRepository.findOrderById(id).addNewItem(newItem); // adds id to Item

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setLocation(getLocationForChildResource(originalUrl, newItem.getId()));
    return new HttpEntity<String>(responseHeaders);
}
```

Controller is a POJO again – no HttpServletResponse* parameters

Updating existing Resource: PUT



```
PUT /store/orders/123/items/abc  
Host: www.myshop.com  
Content-Type: application/xml  
<item>  
...  
</item>
```

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

```
@RequestMapping(value="/orders/{orderId}/items/{itemId}",  
    method=RequestMethod.PUT)  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
public void updateItem(@PathVariable("orderId") long orderId,  
    @PathVariable("itemId") String itemId  
    @RequestBody Item item) {  
    orderRepository.findOrderById(orderId).updateItem(itemId, item);  
}
```

Deleting a Resource: DELETE



```
DELETE /store/orders/123/items/abc  
Host: www.myshop.com
```

...

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

```
@RequestMapping(value="/orders/{orderId}/items/{itemId}",  
    method=RequestMethod.DELETE)  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
public void deleteItem(@PathVariable("orderId") long orderId,  
    @PathVariable("itemId") String itemId) {  
    orderRepository.findOrderById(orderId).deleteItem(itemId);  
}
```

Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
- **RESTful clients with the RestTemplate**
- Conclusion

RestTemplate Introduction



- Provides access to RESTful services
 - Supports URI templates, HttpMessageConverters and custom execute() with callbacks
 - Map or String... for vars, java.net.URI or String for URL

HTTP Method	RestTemplate Method
DELETE	delete(String url, String... urlVariables)
GET	getForObject(String url, Class<T> responseType, String... urlVariables)
HEAD	headForHeaders(String url, String... urlVariables)
OPTIONS	optionsForAllow(String url, String... urlVariables)
POST	postForLocation(String url, Object request, String... urlVariables)
	postForObject(String url, Object request, Class<T> responseType, String... urlVariables)
PUT	put(String url, Object request, String... urlVariables)

Defining a RestTemplate



- Just call constructor in your code

```
RestTemplate template = new RestTemplate();
```

- Has default HttpMessageConverters
 - Same as on the server, depending on classpath
- Or use external configuration
 - To use Apache Commons HTTP Client, for example

```
<bean id="restTemplate" class="org.sfw.web.client.RestTemplate">
  <property name="requestFactory">
    <bean class="org.sfw.http.client.CommonsClientHttpRequestFactory"/>
  </property>
</bean>
```

RestTemplate Usage Examples



```
RestTemplate template = new RestTemplate();
String uri = "http://example.com/store/orders/{id}/items";
```

{id} = 1

```
// GET all order items for an existing order with ID 1:
```

```
OrderItem[] items = template.getForObject(uri, OrderItem[].class, "1");
```

```
// POST to create a new item
```

```
OrderItem item = // create item object
```

```
URI itemLocation = template.postForLocation(uri, item, "1");
```

```
// PUT to update the item
```

```
item.setAmount(2);
```

```
template.put(itemLocation, item);
```

```
// DELETE to remove that item again
```

```
template.delete(itemLocation);
```



Also supports `HttpEntity`, which makes adding headers to the HTTP request very easy as well

Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
 - Request/Response Processing
 - Using MessageConverters
 - Putting it all together
- RESTful clients with the RestTemplate
- **Conclusion**

Conclusion

- REST is an architectural style that can be applied to HTTP-based applications
 - Useful for supporting diverse clients and building highly scalable systems
 - Java provides JAX-RS as standard specification
- Spring-MVC adds REST support using a familiar programming model
 - Extended by @Request-/@ResponseBody
- Use RestTemplate for accessing RESTful apps



LAB

RESTful applications with Spring MVC