

CORE SPRING





Welcome to Core Spring

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

Course Introduction



- Core Spring covers the essentials of developing applications with the Spring Framework
- Course is 50% theory, 50% lab work
 - Theory highlights how to use Spring
 - Lab environment based on SpringSource Tool Suite
- Memory keys contain
 - The Spring Framework
 - Lab environment based on STS and Tomcat
 - Lab materials and documentation

Course Logistics



- Hours
- Lunch and breaks
- Other questions?

Covered in this section

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

Course Agenda: Day 1



-
- Introduction to Spring
 - Using Spring to configure an application
 - Understanding the bean life-cycle
 - Simplifying application configuration
 - Annotation-based dependency injection

Course Agenda: Day 2



- Testing a Spring-based application
- Adding behavior to an application using aspects
- Introducing data access with Spring
- Simplifying JDBC-based data access
- Driving database transactions in a Spring environment

Course Agenda: Day 3



- Introducing object-to-relational mapping (ORM)
- Getting started with Hibernate in a Spring environment
- Effective web application architecture
- Getting started with Spring MVC
- RESTful web services with Spring MVC

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

Covered in this section

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

Build

Spring, Grails
Tool Suite



Application
Frameworks
and Tools

*Build custom apps
for virtual and cloud
environments*

Run

tc server



Lightweight
Application
Runtime

*Run custom apps on a
platform ideally suited for
virtual environments*

Manage

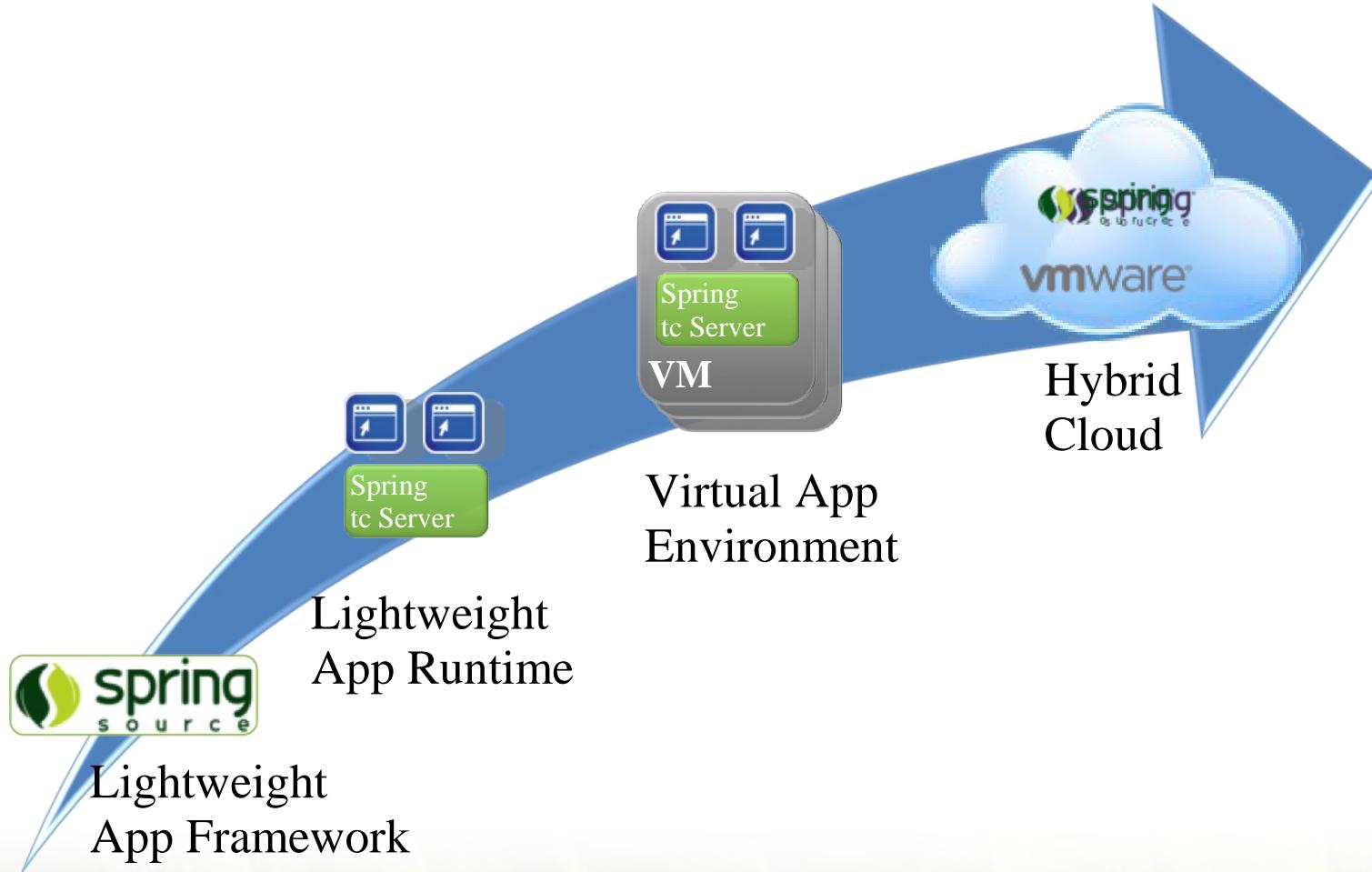
Hyperic



Application
Monitoring and
Management

*Run custom apps on a
platform ideally suited for
virtual environments*

A Division of VMware

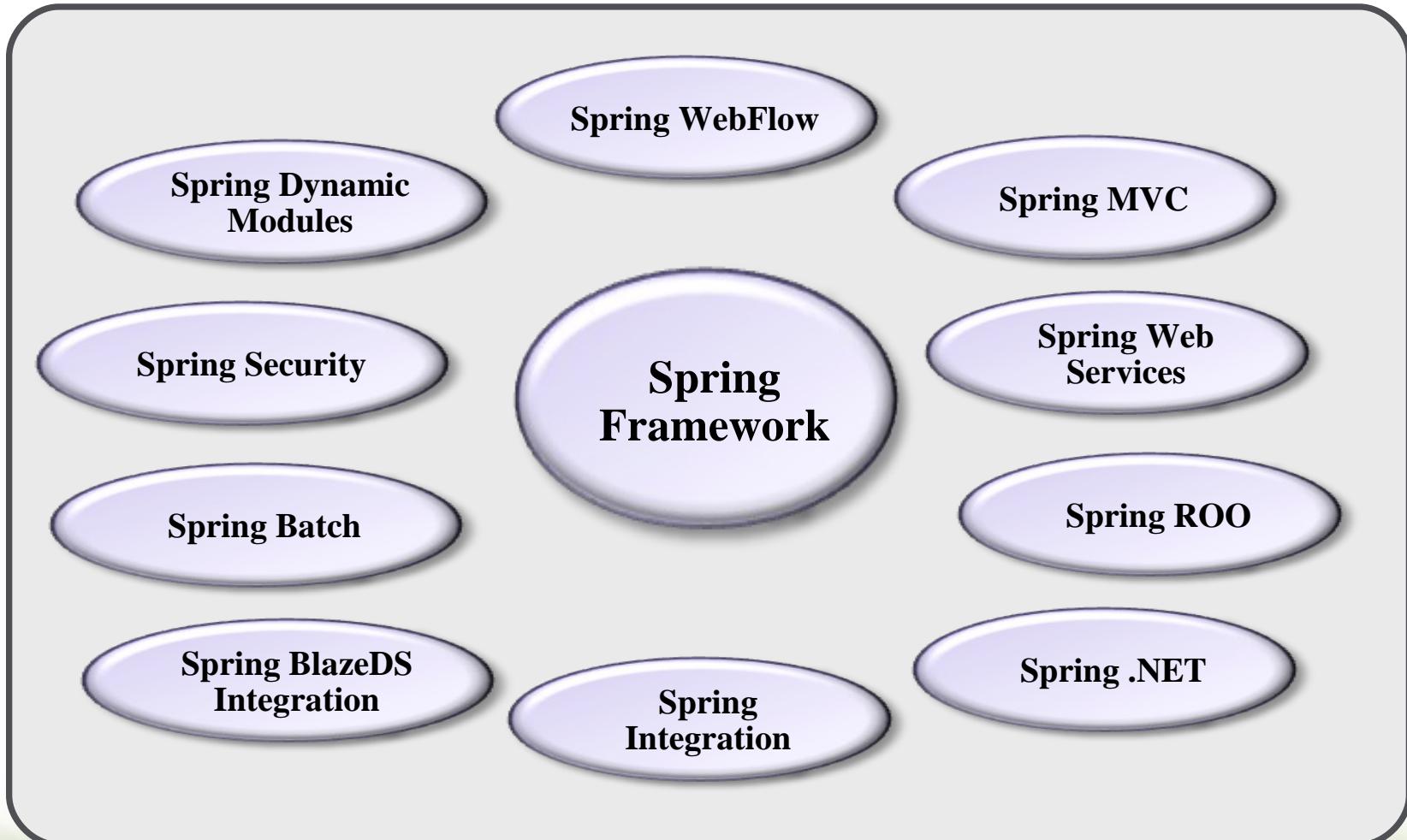


Opensource contributions



- Spring technologies
 - SpringSource develops 99% 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 (most active committers)
- Hyperic
- Groovy/Grails
- Rabbit MQ

The Spring projects





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
- The Spring triangle

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)

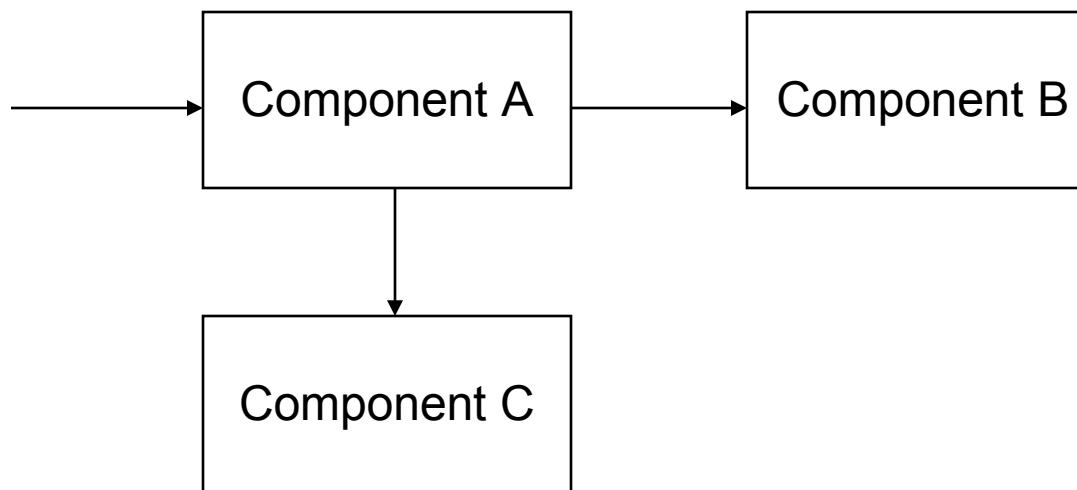


- Core support
 - Application Configuration
 - Enterprise Integration
 - Testing
 - Data Access

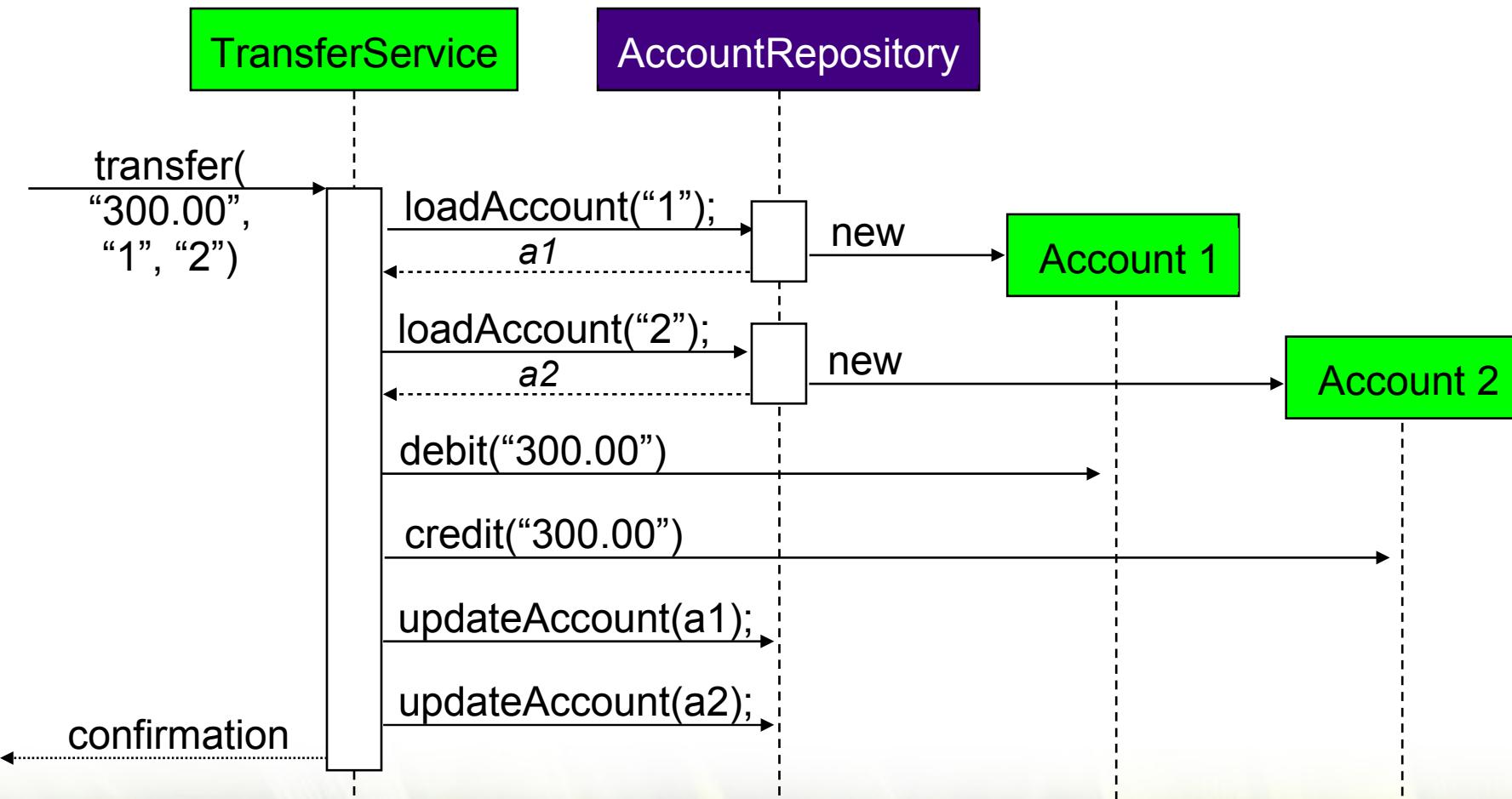
Application Configuration



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



Example: A Money Transfer System

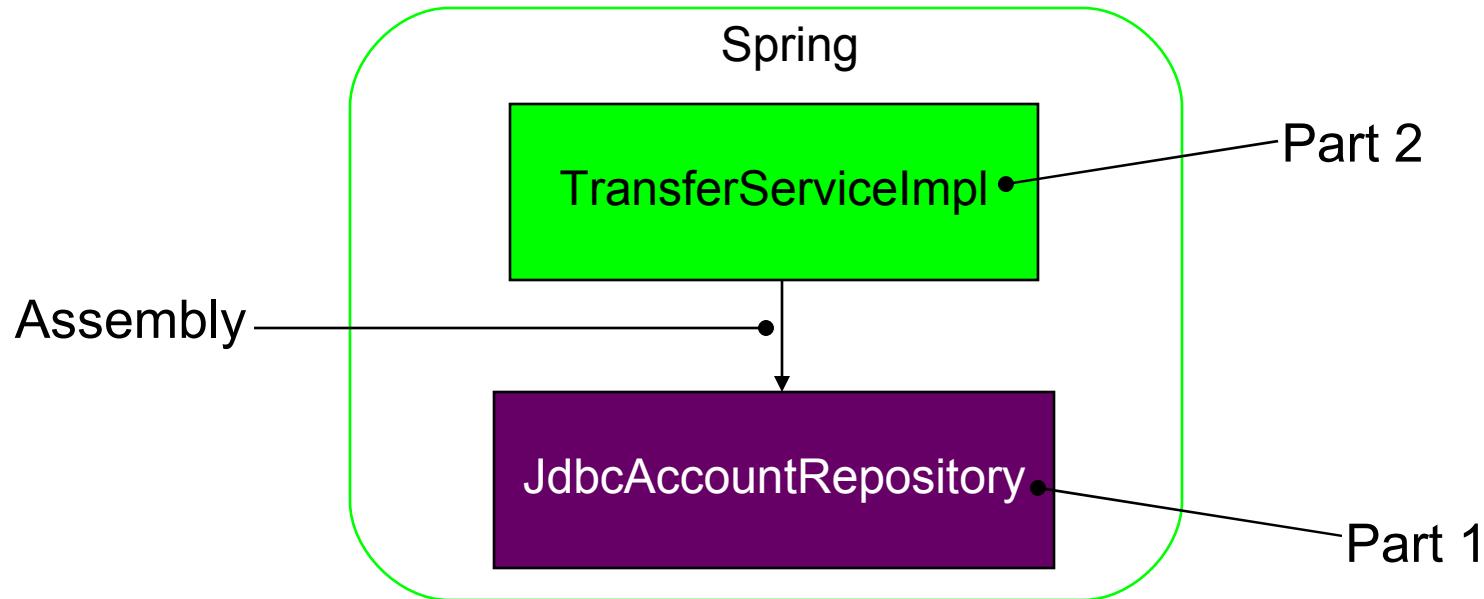


Spring's Configuration Support



- 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

Money Transfer System Assembly



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

Parts are Just Plain Java Objects



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

Implements a service interface

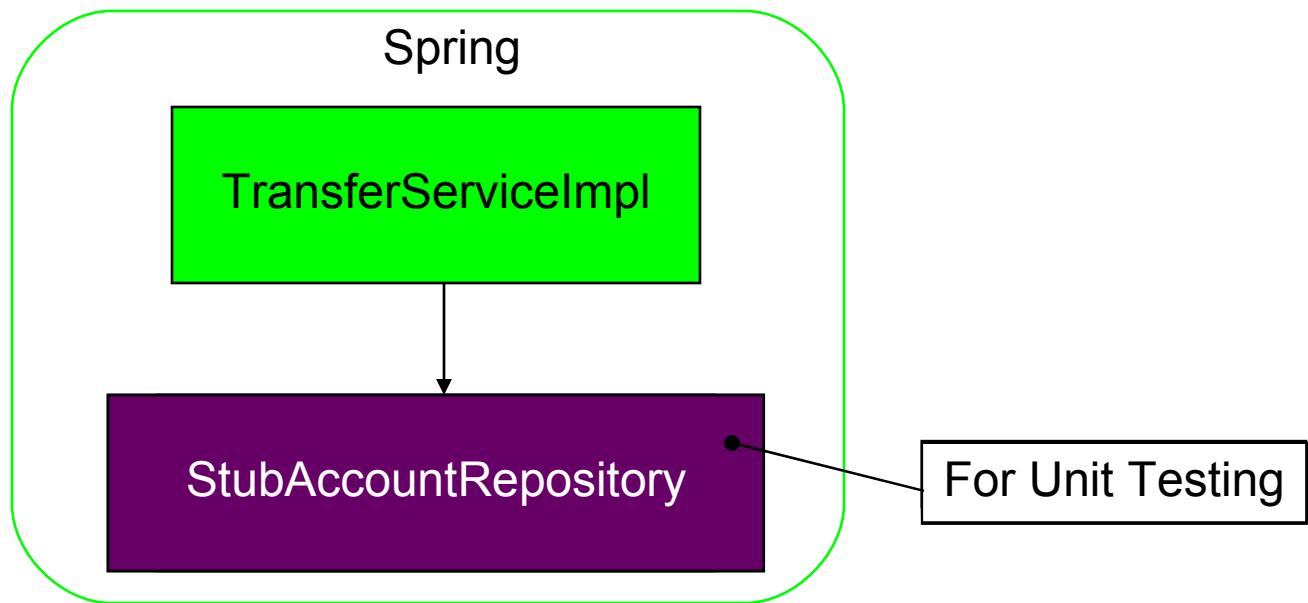
Part 1

```
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);
```

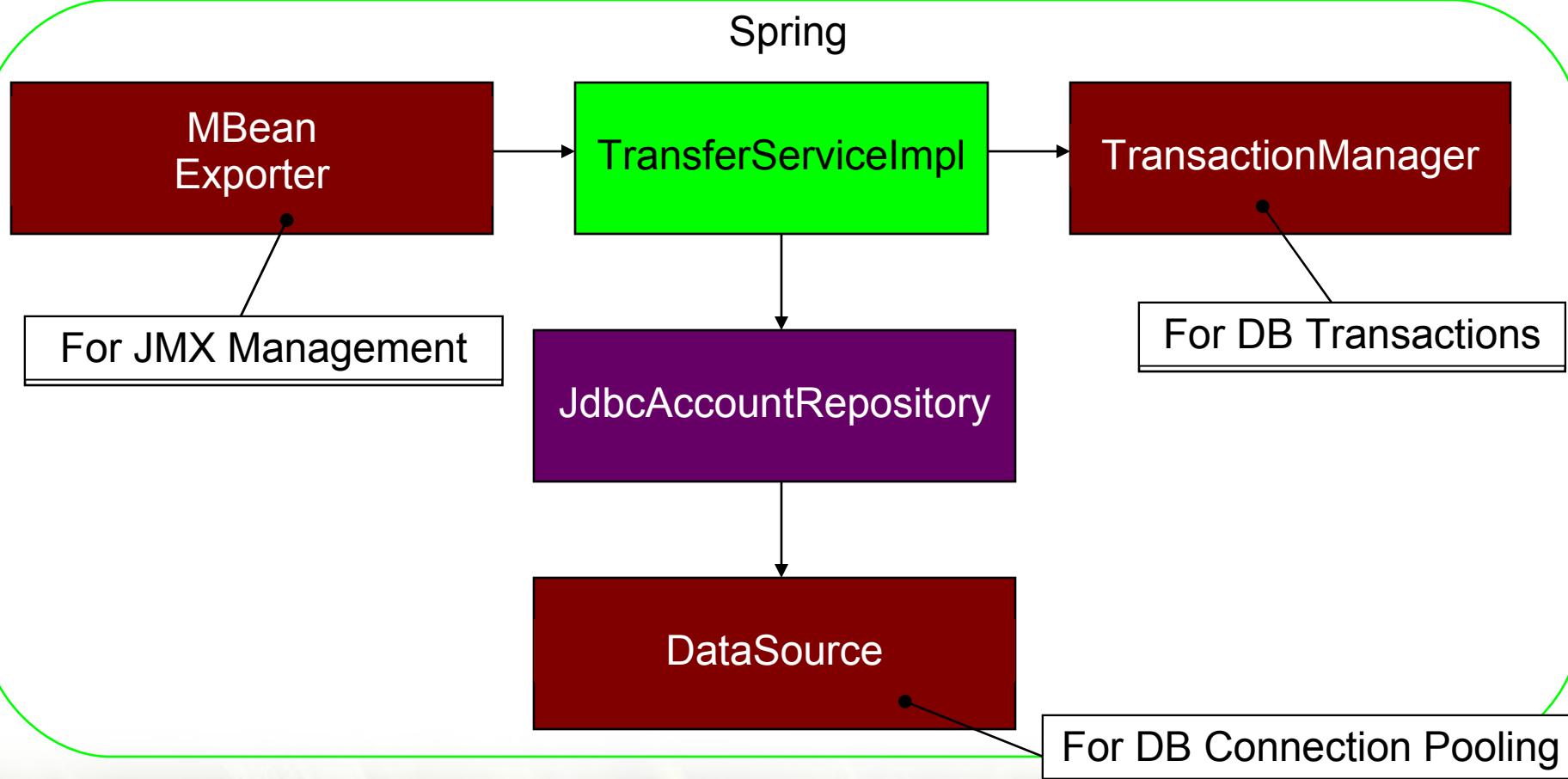
# Enterprise Integration



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

- 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

# Enterprise Money Transfer with Spring



# Simple Application Code



```
public class TransferServiceImpl implements TransferService {
 @Transactional
 public TransferConfirmation transfer(MonetaryAmount amount,
 String srcAccountId, String targetAccountId) {
 Account src = accountRepository.loadAccount(srcAccountId);
 Account target = accountRepository.loadAccount(targetAccountId);
 src.debit(amount);
 target.credit(amount);
 accountRepository.updateAccount(src);
 accountRepository.updateAccount(target);
 return new TransferConfirmation(...);
 }
}
```

Tells Spring to run this method  
in a database transaction

# 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

# 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
    - JDO
    - iBatis

# Spring JDBC 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 of any exception
- Release of the connection

Spring's JDBC helper class

All handled  
by Spring

# 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)

# Spring's Support (3)

---



- Enterprise application development support
  - Developing web services
  - Adding manageability
  - Integrating messaging infrastructures
  - Securing services and providing object access control
  - Scheduling jobs

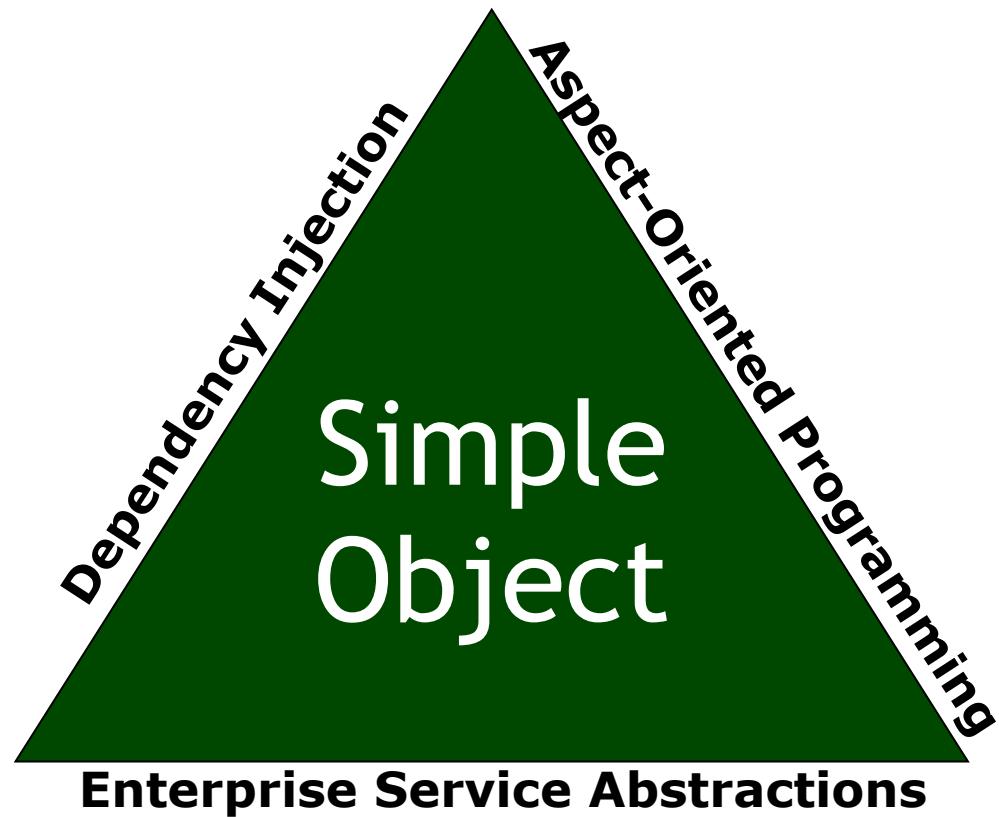
# Spring Framework History



- <http://www.springsource.org/download>
- Spring 3.0 (released 12/09, currently 3.0.0)
  - Requires Java 1.5+ and JUnit 4.7+
  - REST support, JavaConfig, SpEL, more annotations
- Spring 2.5 (released 11/07, currently 2.5.6)
  - Requires Java 1.4+ and supports JUnit 4
  - Annotation DI, @MVC controllers, XML namespaces
- Spring 2.0 (released 10/06, currently 2.0.8)
  - Compatible with Java 1.3+
  - XML simplification, async JMS, JPA, AspectJ support

---

## The Spring Triangle





# LAB

Developing an Application from Plain Java  
Objects



# Spring Quick Start

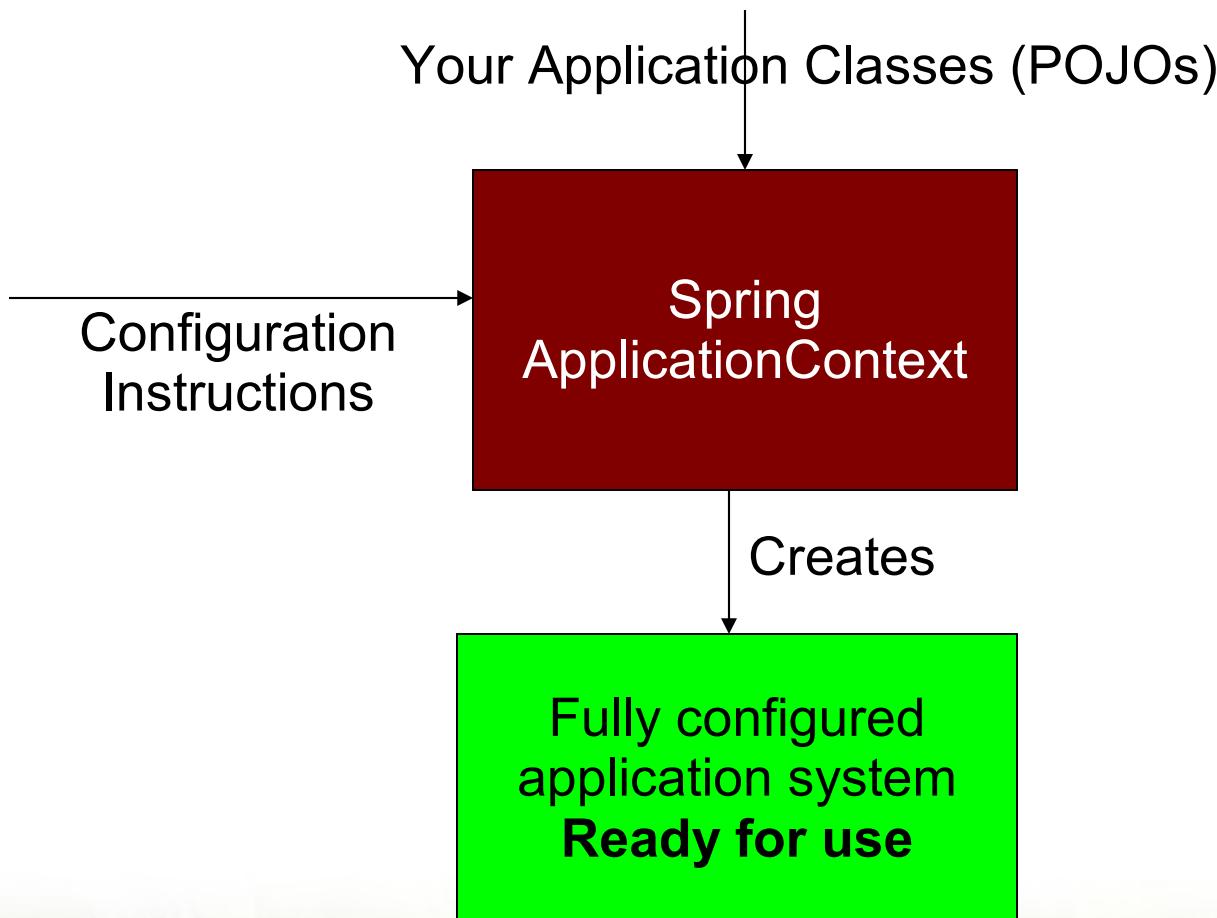
Introducing the Spring Application Context and  
Spring's XML-based configuration language

# Topics in this session

---

- **Spring quick start**
- Writing bean definitions
  - Configuring objects
  - Going beyond constructor instantiation
- Creating an application context
- Retrieving bean instances
- Summary

# 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="app.impl.TransferServiceImpl">
 <constructor-arg ref="accountRepository" />
 </bean>

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

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

</beans>
```

# Creating and Using the Application

---



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

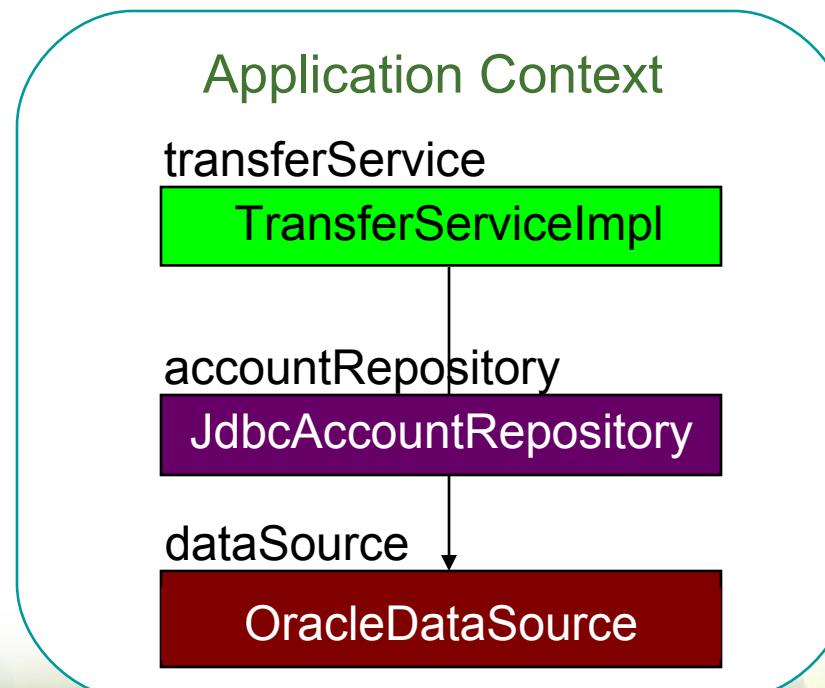
// Look up the application service interface
TransferService service =
 (TransferService) context.getBean("transferService");

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

# 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 they are used
- 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 the bean provides to clients
- The ApplicationContext works as a container, encapsulating the bean implementations chosen for a given deployment
  - Conceals implementation details

# Topics in this session

---

- Spring quick start
- **Writing bean definitions**
  - Configuring objects
  - Going beyond constructor instantiation
- Creating an application context
- Retrieving bean instances
- Summary

# Basic Spring XML Bean Definition Template

---



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="
 http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

 <!-- Add your bean definitions here -->

</beans>
```

# Simplest Possible Bean Definition

---



A class with no dependencies

```
<bean id="service" class="example.ServiceImpl"/>
```

Results in (via Reflection):

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

ApplicationContext

```
service -> instance of ServiceImpl
```

# Constructor Dependency Injection



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

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

A class with a single dependency expected by its constructor

Equivalent to:

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

ApplicationContext

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

# Setter Dependency Injection



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

A class with a single dependency expected by a setter method

```
<bean id="repository" class="example.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

# 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

# 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="example.ServiceImpl">
 <constructor-arg ref="required" />
 <property name="optional" ref="optional" />
</bean>

<bean id="required" class="example.RequiredImpl" />
<bean id="optional" class="example.Optionallmpl" />
```

Equivalent to:

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

# Injecting Scalar Values

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

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

Equivalent to:

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

ApplicationContext

service -> instance of ServiceImpl

# Automatic Value Type Conversion



```
<bean id="service" class="example.ServiceImpl">
 <property name="integerProperty" value="29" />
</bean>
```

```
public class ServiceImpl {
 public void setIntegerProperty(Integer i) { ... }
 // ...
}
```

Equivalent to:

```
ServiceImpl service = new ServiceImpl();
Integer value = 29;
service.setIntegerProperty(value);
```

ApplicationContext

service -> instance of ServiceImpl

# Injecting lists

```
<bean id="service"
 class="com.springsource.warehouse.InventoryManagerImpl">
<property name="warehouseList">
 <list>
 <ref bean="warehouse1"/>
 <ref bean="warehouse2"/>
 <ref bean="warehouse3"/>
 </list>
</property>
</bean>
```

Equivalent to:

```
InventoryManagerImpl manager = new InventoryManagerImpl();
manager.setWarehouseList(list); // create list with bean references
```

ApplicationContext

manager -> instance of InventoryManagerImpl

# Injecting other types of collection

---



- Similar support available for
  - Properties (through props / prop elements)
  - Set (through a set element, similar to list)
  - Map (through map / key elements)



Some more advanced features about collections will be seen later with the *util* namespace

# Topics in this session

---

- How Spring works
- Writing Spring bean definitions
  - Configuring objects
  - **Going beyond constructor instantiation**
- Creating a Spring application context
- Retrieving bean instances
- Summary

# The factory-method attribute



- Non-intrusive
  - Useful for existing Singletons or Factories

```
public class LegacySingleton {
 private static LegacySingleton inst = new LegacySingleton();
 private LegacySingleton() { ... }

 public static LegacySingleton getInstance() {
 return inst;
 }
}
```

```
<bean id="service" class="example.LegacySingleton"
 factory-method="getInstance" />
```



*getInstance()* should always be static

# POJOs as Factory Beans



- Any POJO can be used as a factory

```
public class ClientServiceFactory {
 ...
 public ClientService getNewInstance() {
 ...
 }
}
```

```
<bean id="clientServiceFactory" class="example.ClientServiceFactory"/>
```

```
<bean id="clientService" factory-bean="clientServiceFactory"
 factory-method="getNewInstance">
```

No need for class attribute

# The FactoryBean interface



- Part of the Spring framework
- Used by many Spring classes
  - can also choose to use this yourself

```
public class ClientServiceFactoryBean implements
 FactoryBean<ClientService> {
 //...
 public ClientService getObject() throws Exception {
 return clientService;
 }

 public Class<?> getObjectType() {
 return ClientService.class;
 }
 public boolean isSingleton() { return true; }
}
```

# The FactoryBean interface

---



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

```
<bean id="cpxService" class="example.ComplexServiceFactory" />

<bean id="controller" class="example.MyController">
 <property name="service" ref="cpxService" />
</bean>
```

# FactoryBeans in Spring

---



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

# Topics in this session

---

- Spring quick start
- Writing bean definitions
  - Configuring objects
  - Going beyond constructor instantiation
- **Creating an application context**
- Summary

# Creating a Spring Application Context

---



- Spring application contexts can be bootstrapped in any environment, including
  - JUnit system test
  - Web application
  - Enterprise Java Bean (EJB)
  - 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 TransferServiceTest {
 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**

# 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

# Creating a Spring Application Context from Multiple Files

---



- A context can be configured from multiple files
- Allows partitioning of bean definitions into logical groups
- Generally a best practice to separate out “application” beans from “infrastructure” beans
  - Infrastructure often changes between environments
  - Evolves at a different rate

# Mixed Configuration

```
<beans>
```

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

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

Coupled to a local Oracle environment

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

```
</beans>
```

# Partitioning Configuration



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

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

```
<beans>
 <!-- creates an in-memory database populated with test data for fast testing -->
 <bean id="dataSource"
 class="org.sfj.jdbc.datasource.embedded.EmbeddedDatabaseFactoryBean">
 <property name="databasePopulator" ref="populator"/>
 </bean>

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

Now substitutable for other environments

# 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 Oracle environment

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

---

- How Spring works
- Writing Spring bean definitions
  - Configuring objects
  - Going beyond constructor instantiation
- Creating a Spring application context
- **Retrieving bean instances**
- Summary

# Accessing a Bean in Spring 3



- Cast is not compulsory anymore

```
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);
```

```
// No need for bean id if type is unique
```

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

New in  
Spring 3.0

# Topics in this session

---

- How Spring works
- Writing Spring bean definitions
  - Configuring objects
  - Going beyond constructor instantiation
- Creating a Spring application context
- Retrieving bean instances
- **Summary**

# Benefits of Dependency Injection

---



- 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 the implementation details of each dependency
- Improves testability
  - Dependencies can be 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



# Understanding the Bean Lifecycle

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

# Topics in this session

---



- Introduction to Spring's XML namespaces
- JSR-250 annotations
- The lifecycle of a Spring application context
  - The initialization phase
  - The use phase
  - The destruction phase
- Bean scoping

# Introduction to Spring's Custom XML Namespaces

---



- Spring provides higher-level configuration languages that build on the generic <beans/> language
  - Provide abstraction and semantic validation
  - Simplify common configuration needs
- Each language is defined by its own XML namespace you may import
  - You can even write your own

# Importing a Custom XML Namespace



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

 http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context-3.0.xsd">

 <!-- You can now use <context:/> tags -->

</beans>
```

Assign namespace a prefix

Associate namespace with its XSD

# Benefits of Using Namespace Configuration



- Before

```
<beans>
 <bean class="org.springframework.beans.factory.config.
 PropertyPlaceholderConfigurer">
 <property name="location" value="/WEB-INF/datasource.properties" />
 </bean>
</beans>
```

Generic bean/property syntax is all that's available

Requires remembering framework classnames

- After

```
<beans ... >
 <context:property-placeholder location="/WEB-INF/datasource.properties" />
</beans>
```

Creates same bean as above, but hides framework details

Attributes are specific to the task, rather than generic name/value pairs

# Namespaces Summary



- Namespace configuration is about simplification and expressiveness
  - Generic bean/property syntax can be too low-level
  - Hides framework details
- Namespaces are defined for categories of related framework functionality\*
  - aop, beans, context, jee, jms, tx, util
- We will use namespaces to simplify configuration throughout this training

\* *an incomplete list. see <http://www.springframework.org/schema/> for details*

# Topics in this session

---



- Introduction to Spring's XML namespaces
- **JSR-250 annotations**
- The lifecycle of a Spring application context
  - The initialization phase
  - The use phase
  - The destruction phase
- Bean scoping

# A Primer on JSR-250

## “Common Annotations”

---



- JSR-250 provides a set of common annotations for use in Java EE applications
  - Helps avoid proliferation of framework-specific annotations
  - Facilitates portability
  - Used by EJB 3.0
  - Used by Spring, starting with version 2.5

- All annotations reside in the javax.annotation package
  - Included in the JDK starting in Java 6
  - Available as part of the 'javaee-api' jar for Java 5
- To name a few
  - @PostConstruct
  - @PreDestroy
  - @Resource
- Spring makes use of these annotations to trigger lifecycle events

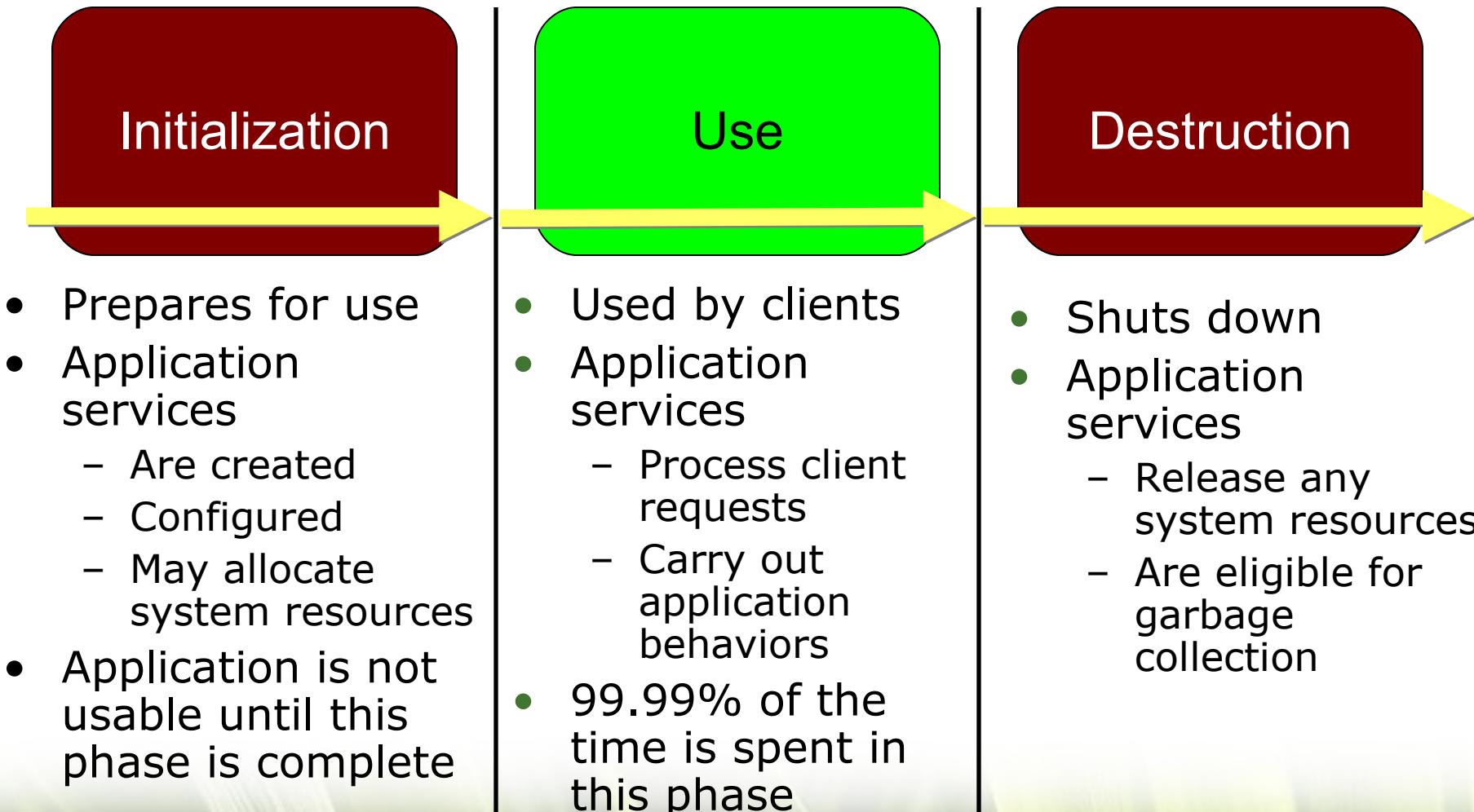
# Topics in this session

---



- Introduction to Spring's XML namespaces
- JSR-250 annotations
- **The lifecycle of a Spring application context**
  - The initialization phase
  - The use phase
  - The destruction phase
- Bean scoping

# Phases of the Application Lifecycle



# Spring's Role as a Lifecycle Manager

---



- For nearly any class of application
  - Standalone Java application
  - JUnit System Test
  - Java EE™ web application
  - Java EE™ enterprise application
- Spring fits in to manage the application lifecycle
  - Plays an important role in all phases

# Topics in this session



- Introduction to Spring's XML namespaces
- The lifecycle of a Spring application context
  - **The initialization phase**
  - The use phase
  - The destruction phase
- Bean scoping



# The 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?

# 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

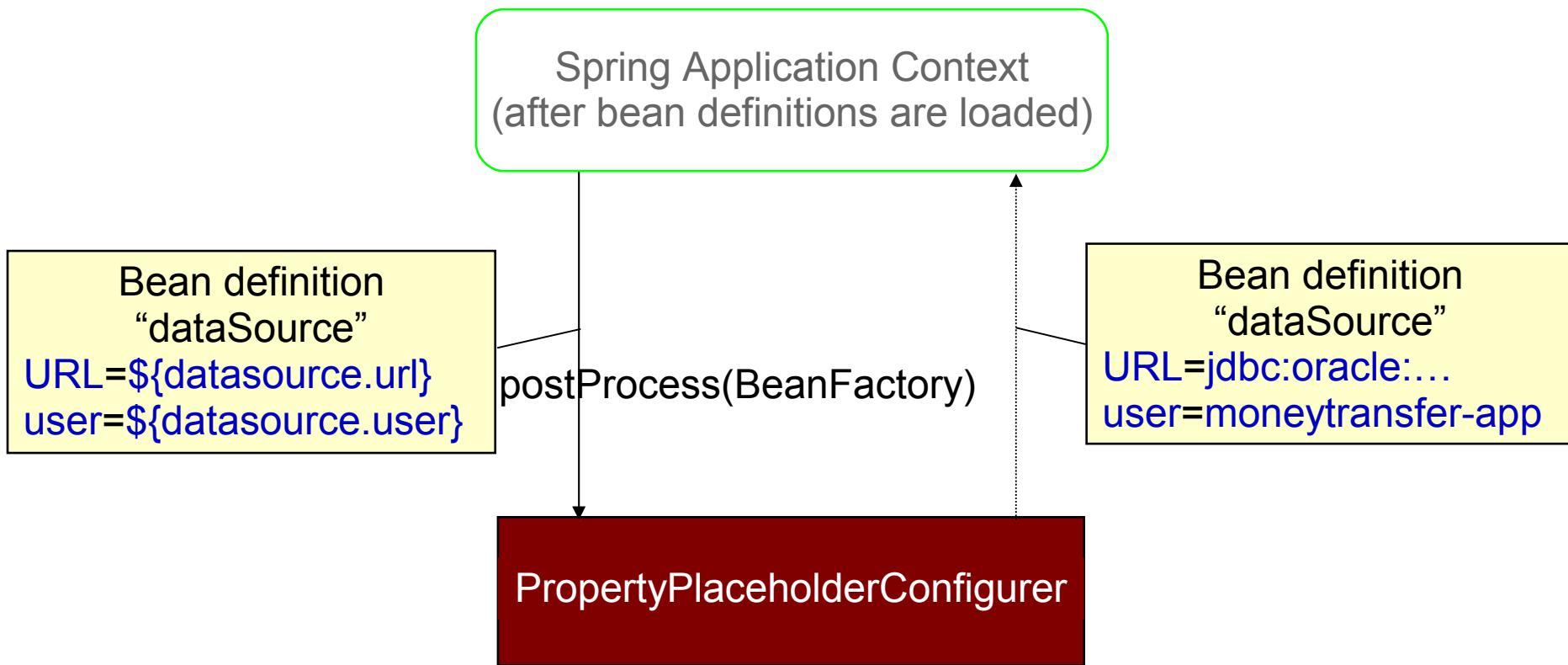
# An Example BeanFactoryPostProcessor

---



- **PropertyPlaceholderConfigurer** substitutes \${variables} in bean definitions with values from .properties files

# How PropertyPlaceholderConfigurer Works



# Using PropertyPlaceholderConfigurer



```
<beans>

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

 <bean class="org.springframework.beans.factory.config.
 PropertyPlaceholderConfigurer">
 <property name="location" value="/WEB-INF/datasource.properties" />
 </bean>

</beans>
```

Variables to replace

File where the variable values reside

datasource.properties

**datasource.url=jdbc:oracle:thin:@localhost:1521:BANK  
datasource.user=moneytransfer-app**

Easily editable

# Simplifying configuration with <context:property-placeholder>



```
<beans ... >

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

 <context:property-placeholder location="/WEB-INF/datasource.properties" />

</beans>
```

Creates the same PropertyPlaceholderConfigurer  
bean definition in a more concise fashion

# Inside the Application Context Initialization Lifecycle

---



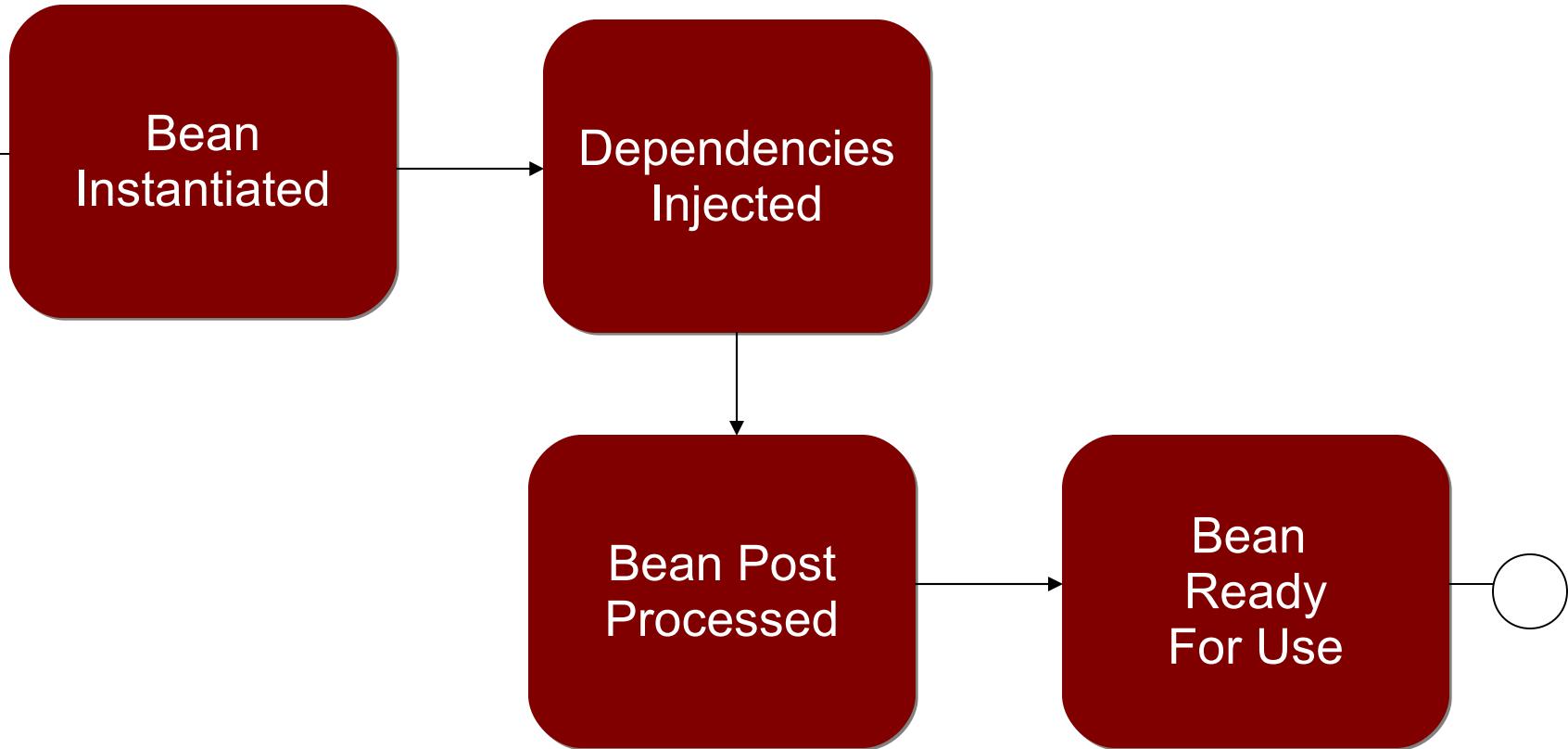
- Load bean definitions
- **Initialize bean instances**

# Initializing Bean Instances



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

# Bean Initialization Steps



# Bean Post Processing



- Bean post processing can be broken down into two steps
  - Initialize the bean if instructed
  - Call special BeanPostProcessors to perform additional configuration

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

# Initialize the Bean if Instructed

---



- A bean can optionally register for one or more initialization callbacks
  - Useful for executing custom initialization behaviors
- There's more than one way to do it
  - JSR-250 @PostConstruct
  - <bean/> init-method attribute
  - Implement Spring's InitializingBean interface

# Bean Initialization Techniques

---



- **@PostConstruct**
- init-method
- InitializingBean

# Registering for Initialization with @PostConstruct



```
public class TransferServiceImpl {
 @PostConstruct
 void init() {
 // your custom initialization code
 }
}
```

Tells Spring to call this method after dependency injection

No restrictions on method name or visibility

- This is the preferred initialization technique
  - Flexible, avoids depending on Spring APIs
  - Requires Spring 2.5 or better

# Enabling Common Annotation-Based Initialization



- `@PostConstruct` will be ignored unless Spring is instructed to detect and process it

```
<beans>
 <bean id="transferService" class="example.TransferServiceImpl" />

 <bean class="org.springframework.context.annotation.
 CommonAnnotationBeanPostProcessor" />
</beans>
```

Detects and invokes any methods annotated with `@PostConstruct`

# Simplifying Configuration with <context:annotation-config>

---



```
<beans>
 <bean id="transferService" class="example.TransferServiceImpl" />

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

Enables multiple BeanPostProcessors, including:

- RequiredAnnotationBeanPostProcessor
- CommonAnnotationBeanPostProcessor

# Bean Initialization Techniques

---



- @PostConstruct
- **init-method**
- InitializingBean

# Initializing a Bean You Do Not Control



- Use **init-method** to initialize a bean you do not control the implementation of

```
<bean id="broker"
 class="org.apache.activemq.broker.BrokerService"
 init-method="start">
 ...
</bean>
```

A callout box with a black border and white background contains the text "Class with no Spring dependency". Two arrows point from the text "init-method="start"" in the XML code above to this callout box: one from the start of the attribute and another from the end of the attribute.

Method can be any visibility, but must have no arguments

# Bean Initialization Techniques

---



- @PostConstruct
- init-method
- **InitializingBean**

# Registering for an Initialization Callback with InitializingBean

---



- Initialization technique used prior to Spring 2.5
  - Disadvantage: Ties your code to the framework
  - Advantage: The only way to *enforce* initialization code should run (not affected by configuration changes)

```
package org.springframework.beans.factory;

public interface InitializingBean {
 public void afterPropertiesSet();
}
```

# Registering for an Initialization Callback with InitializingBean



```
public class MessageReceiver implements InitializingBean {
 public void afterPropertiesSet() {
 // allocate message receiving resources
 }
}
```

Implements special Spring lifecycle interface

Spring invokes method automatically

```
<beans>
 <bean id="messageReceiver" class="example.MessageReceiver" />
</beans>
```

No other configuration required

# 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
- Several implementations are provided by the framework
- Not common to write your own

# An Example BeanPostProcessor

---



- RequiredAnnotationBeanPostProcessor
  - Provided by the framework
  - Enforces that **@Required** properties are set
  - Must be explicitly enabled via configuration

# Using RequiredAnnotationBeanPostProcessor



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

 @Required ← Indicates the dependency
 public void setAccountRepository(AccountRepository ar) {
 this.accountRepository = ar;
 }
}
```

Property 'accountRepository' is required for bean 'transferService'

```
<beans>
 <bean id="transferService" class="example.TransferServiceImpl" />

 <bean class="org.springframework.beans.factory.annotation.
 RequiredAnnotationBeanPostProcessor" />
</beans>
```

Automatically detected; no other configuration is necessary

# Simplifying Configuration with <context:annotation-config>

---



- Remember <context:annotation-config>?

```
<beans>
 <bean id="transferService" class="example.TransferServiceImpl" />

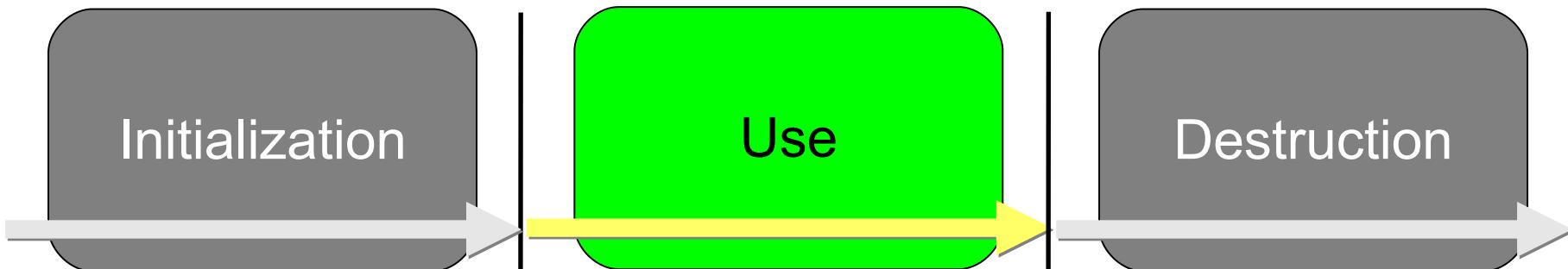
 <context:annotation-config/>
</beans>
```

Automatically enables RequiredAnnotationBeanPostProcessor

# Topics in this session



- Introduction to Spring's XML namespaces
- The lifecycle of a Spring application context
  - The initialization phase
  - **The use phase**
  - The destruction phase
- Bean scoping



# The 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");
```

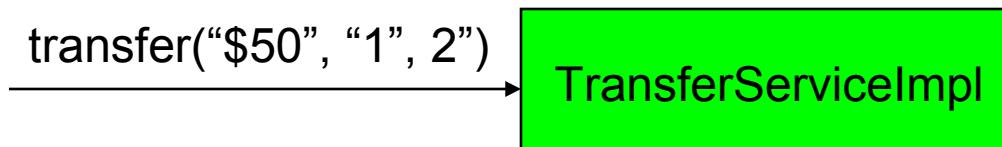
- But exactly what happens in this phase?

# Inside The Bean Request (Use) Lifecycle

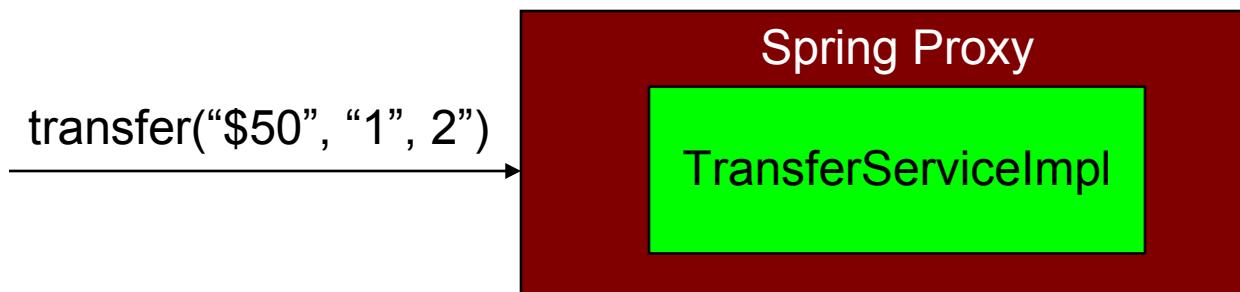
---



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

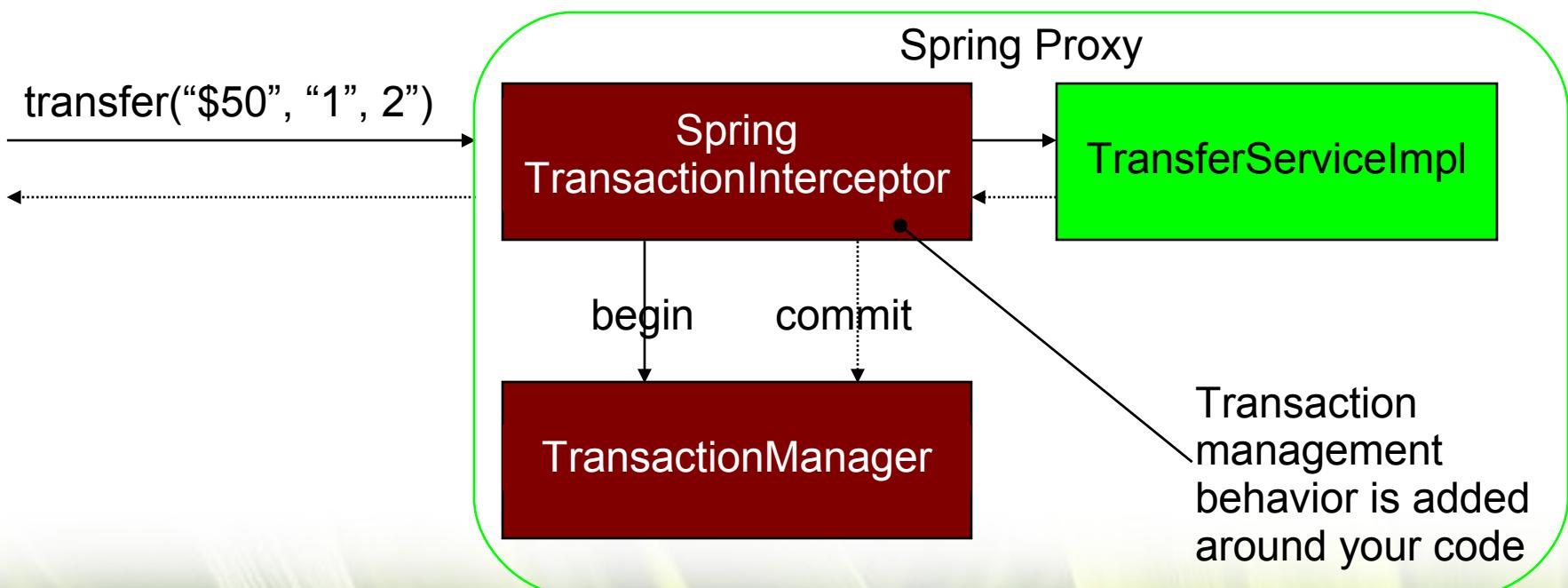


- If your bean has been wrapped in a proxy things get more interesting

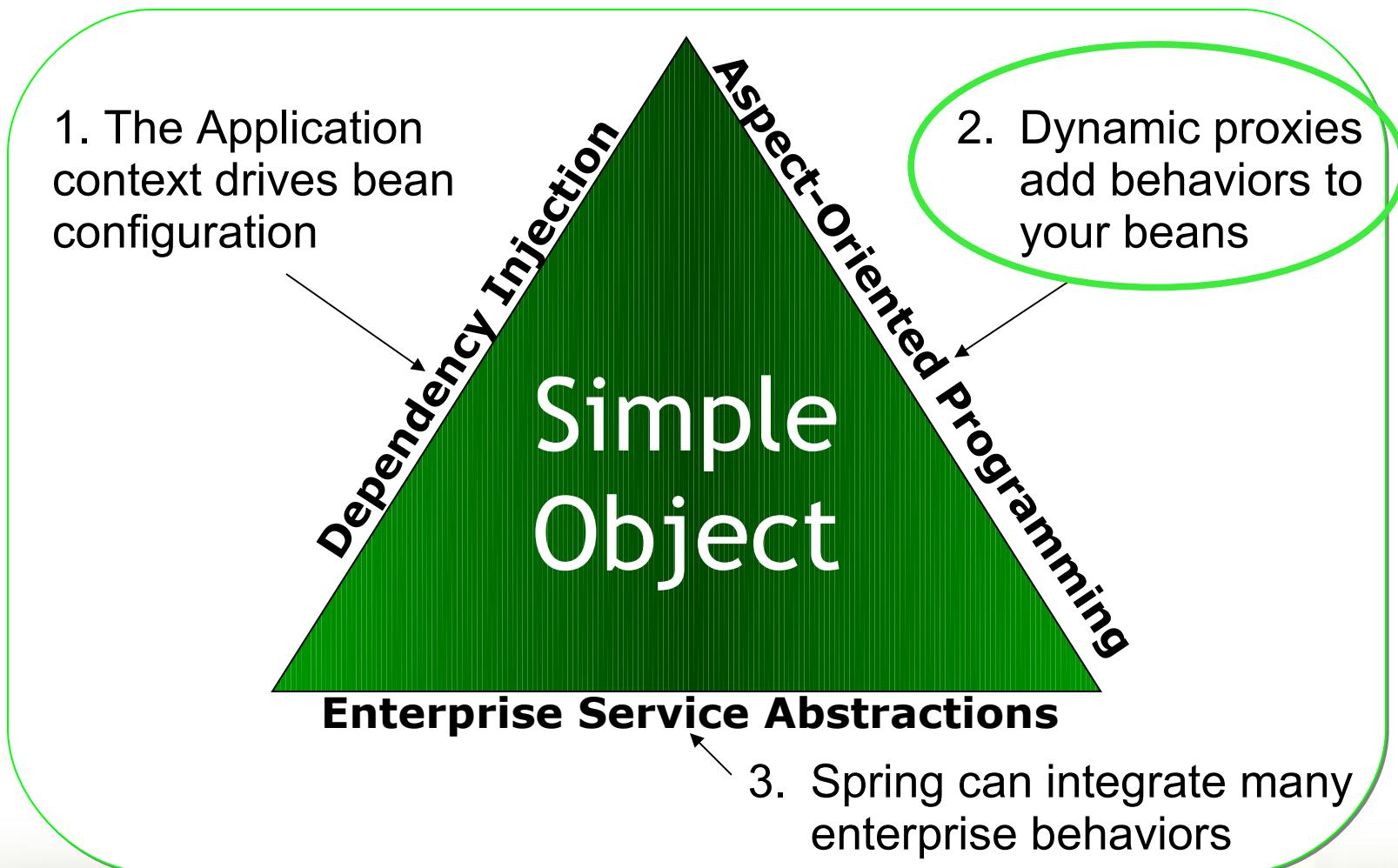


# Proxy Power

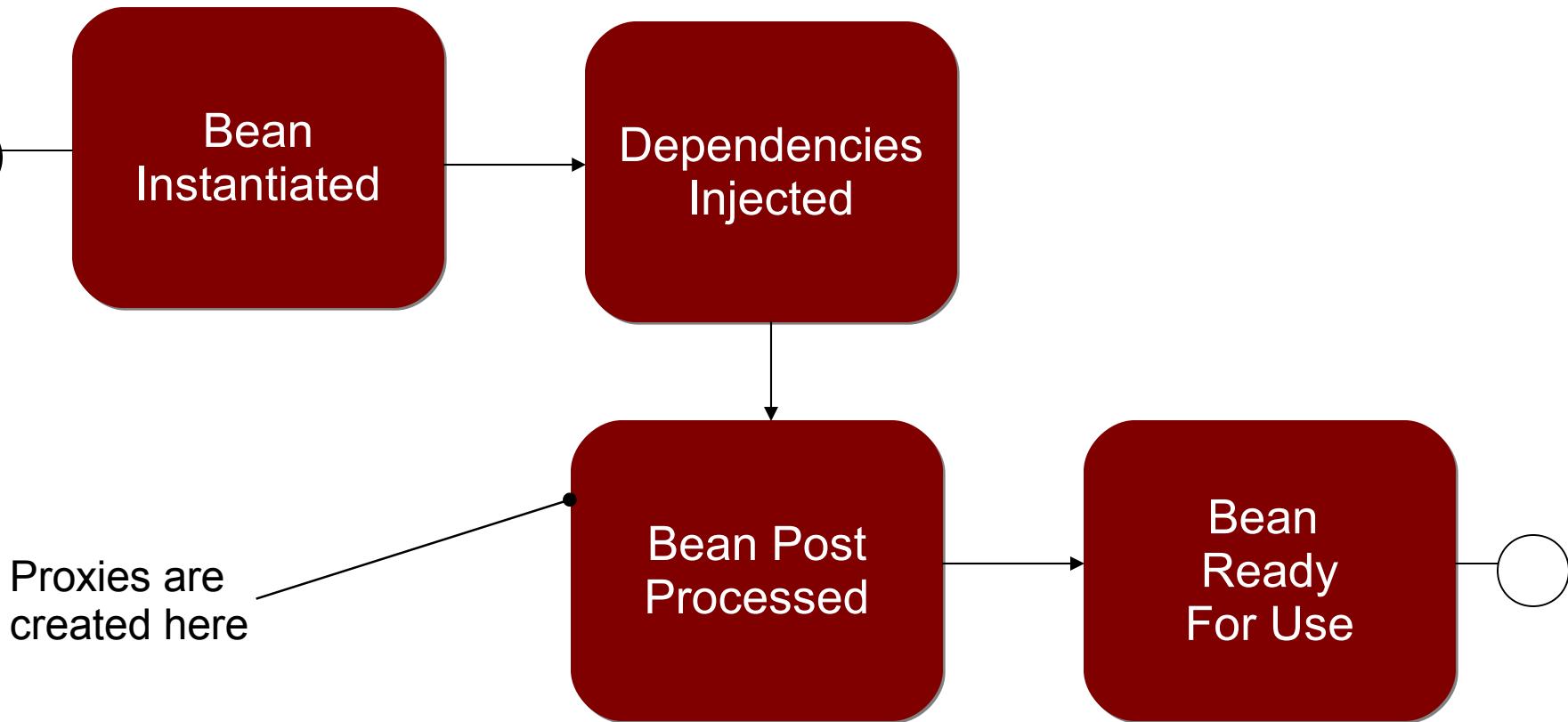
- A BeanPostProcessor may wrap your beans in a dynamic proxy and add behavior to your application logic *transparently*



# Proxies are Central to the Spring Triangle



# Bean Initialization Steps



# Topics in this session



- Introduction to Spring's XML namespaces
- The lifecycle of a Spring application context
  - The initialization phase
  - The use phase
  - **The destruction phase**
- Bean scoping



# 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

# Destroy Bean Instances if Instructed

---



- A bean can optionally register for one or more destruction callbacks
  - Useful for releasing resources and 'cleaning up'
- There's more than one way to do it
  - JSR-250 @PreDestroy
  - <bean/> destroy-method attribute
  - Implement Spring's DisposableBean interface

- 
- **@PreDestroy**
  - destroy-method
  - DisposableBean

# Registering for Destruction with @PreDestroy



```
public class TransferServiceImpl {
 @PreDestroy ←
 void releaseResources() {
 // your custom destruction code
 }
}
```

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

No restrictions on method name or visibility

```
<beans>
 <bean id="transferService" class="example.TransferServiceImpl" />

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

Enables common annotation processing

# Bean Destruction Techniques



- 
- @PreDestroy
  - **destroy-method**
  - DisposableBean

# Destroying a Bean You Do Not Control



- Use **destroy-method** to dispose a bean you do not control the implementation of

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

Class with no Spring dependency

Method can be any visibility, but must have no arguments

# Bean Destruction Techniques



- @PreDestroy
- destroy-method
- **DisposableBean**

# Registering for a Destruction Callback with DisposableBean

---



- Register for a bean destruction callback by implementing the **DisposableBean** interface

```
package org.springframework.beans.factory;

public interface DisposableBean {
 public void destroy();
}
```

- Spring invokes this method at the right time

# Topics in this session

---



- Introduction to Spring's XML namespaces
- JSR-250 annotations
- The lifecycle of a Spring application context
  - The initialization phase
  - The use phase
  - The destruction phase
- **Bean scoping**

# Bean Scoping



- Spring puts each bean instance in a scope
- Singleton scope is the default
  - The “single” instance is scoped by the context itself
  - By far the most common
- Other scopes can be used on a bean-by-bean basis

```
<bean id="myBean" class="example.MyBean"
 scope="...>
</bean>
```

Put this bean in some other scope

# Available Scopes

---



- **prototype** - A new instance is created each time the bean is referenced
- **session** - A new instance is created once per user session
- **request** - A new instance is created once per request
- **custom** - You define your own rules
  - Advanced feature

# Scoped Dependencies of Singletons

---



- Session-, request- or custom-scoped beans often act as dependency of singleton
- What will happen in this case?

```
<bean id="shoppingService" class="example.ShoppingServiceImpl">
 <property name="shoppingCart" ref="shoppingCart"/>
</bean>

<bean id="shoppingCart" class="example.ShoppingCart" scope="session"/>
```

# One-time Injection



```
<bean id="shoppingService" class="example.ShoppingServiceImpl">
 <property name="shoppingCart" ref="shoppingCart"/>
</bean>

<bean id="shoppingCart" class="example.ShoppingCart" scope="session"/>
```

- Singleton will only be injected *once*, on startup
- No current HTTP Session, so you'll get an error
  - Even if there was a Session, the service would use the same ShoppingCart every time
- Could fix by using lookup instead of injection
  - Inject the context and call ApplicationContext.getBean
  - Ugly: ties your code to Spring, hard to test

# Scoped Proxies



- Solution is to *proxy* the scoped dependency
- Proxy delegates to correct instance for current request or session or custom context
- Same proxy can be used by singleton for its entire lifecycle, so dependency injection works
- Built-in feature of Spring using aop namespace:

```
<bean id="shoppingService" class="example.ShoppingServiceImpl">
 <property name="shoppingCart" ref="shoppingCart"/>
</bean>

<bean id="shoppingCart" class="example.ShoppingCart" scope="session">
 <aop:scoped-proxy/>
</bean>
```



# LAB

## Understanding the Bean Lifecycle



# Simplifying Application Configuration

Techniques for Creating Reusable and Concise Bean Definitions

# Topics in this session

---

- **Bean Definition Inheritance**
- Inner Beans
- Property Editors
- Importing Configuration Files
- Bean Naming
- The p Namespace
- The util Namespace
- Spring Expression Language

# 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?

# With Bean Definition Inheritance



```
<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" />
 </bean>
</beans>
```

Will not be instantiated

Each pool inherits its parent configuration

# Overriding Parent Bean Configuration

```
<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>
 <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>
</beans>
```

Not abstract; acts as a concrete default

Overrides URL property

Overrides class as well

# Topics in this session

---

- Bean Definition Inheritance
- **Inner Beans**
- Property Editors
- Importing Configuration Files
- Bean Naming
- The p Namespace
- The util Namespace
- Spring Expression Language

- Sometimes a bean naturally scopes the definition of another bean
  - The “scoped” bean is not useful to anyone else
- Make the scoped bean an inner bean
  - More manageable
  - Avoids polluting the bean namespace

# 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>
```

# Topics in this session

---

- Bean Definition Inheritance
- Inner Beans
- **Property Editors**
- Importing Configuration Files
- Bean Naming
- The p Namespace
- The util Namespace
- Spring Expression Language

# Introducing PropertyEditor (1)

---



- In XML all literal values are text strings

```
<bean id="phoneValidator" class="sample.PhoneValidator">
 <property name="pattern" value="[0-9]{3}-[0-9]{3}-[0-9]{4}" />
</bean>
```

A text string

- In your code there are often rich object types

```
public class PhoneValidator {
 private Pattern pattern;

 public void setPattern(Pattern pattern) {
 this.pattern = pattern;
 }
}
```

A rich object with behavior

# Introducing PropertyEditor (2)

---



- Your code should not be burdened with the responsibility of type conversion
- Spring will convert for you using a **PropertyEditor**
  - Comes with many property editors pre-installed
  - You may write and install your own

# Built-In PropertyEditor implementations

---



- See the `org.springframework.beans.propertyeditor` package
- Some highlights
  - CustomNumberEditor converts to the Number types
  - CustomBooleanEditor converts to Boolean
  - CustomDateEditor converts to `java.util.Date`
  - ResourceBundleEditor converts to a Resource
  - PropertiesEditor converts to `java.util.Properties`
  - ByteArrayPropertyEditor converts to a `byte[]`
  - LocaleEditor converts to a Locale

# Built-In PropertyEditor Example



```
<bean id="sessionFactory"
 class="org.springframework.orm.hibernate3.support.
 LocalSessionFactoryBean">
 ...
 <property name="hibernateProperties">
 <value>
 hibernate.format_sql=true
 hibernate.jdbc_batch_size=30
 </value>
 </property>
</bean>
```

String-encoded properties

```
public class LocalSessionFactoryBean {
 public void setHibernateProperties(Properties p) { ...
}
```

Typed java.util.Properties object

# Implementing Your Own PropertyEditor



- Create a custom editor by extending PropertyEditorSupport
- Override setAsText(String) to convert from String
- Override getAsText() to format the value (optional)

```
public class PatternEditor extends PropertyEditorSupport {
 public void setAsText(String text) {
 setValue(text != null ? Pattern.compile(text) : null);
 }
 public String getAsText() {
 Pattern value = (Pattern) getValue();
 return (value != null ? value.pattern() : "");
 }
}
```

Converts from String

# Installing your Custom Property Editor



- Use a CustomEditorConfigurer to install your custom editor (a BeanFactoryPostProcessor)

```
<bean class="org.springframework.beans.factory.config.
 CustomEditorConfigurer">
 <property name="customEditors">
 <map>
 <entry key="javax.util.regex.Pattern"
 value="example.PatternEditor" />
 </map>
 </property>
</bean>
```

Fully-qualified convertible type

The PropertyEditor to perform conversions to that type

\*A *PatternEditor* implementation is built-in as of Spring 2.0.1

# Topics in this session

---

- Bean Definition Inheritance
- Inner Beans
- Property Editors
- **Importing Configuration Files**
- Bean Naming
- The p Namespace
- The util Namespace
- Spring Expression Language

# Importing Config Files (1)



- Use the import tag to encapsulate including another configuration file

application-config.xml

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

system-test-config.xml

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

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

Client only imports one file

# Importing Config Files (2)



- import tag uses relative path by default

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

- This can be overridden using prefixes

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

# Importing Configuration Files (3)

---



- Promotes splitting configuration into logical groups of bean definitions
- Enables reuse across modules within an application
- Complements the option of creating an ApplicationContext from multiple files

```
ApplicationContext context =
 new ClassPathXmlApplicationContext(
 new String[] {
 "classpath:com/module1/module-config.xml",
 "classpath:com/module2/module-config.xml"
 });
```

# Topics in this session

---

- Bean Definition Inheritance
- Inner Beans
- Property Editors
- Importing Configuration Files
- **Bean Naming**
- The p Namespace
- The util Namespace
- Spring Expression Language

# Bean Naming

- Most beans are assigned a unique name with the **id** attribute

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

- As an XML ID this attribute has several limitations
  - Can only hold a single value
  - Some special characters cannot be used (/ :)

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



'/' not allowed

# Assigning a Bean Multiple Names

---



- Use the name attribute when special characters are needed

```
<bean name="dataSource/primary" .../>
```

- Also consider it when you need to assign a bean multiple names

```
<bean name="dataSource, dataSource/primary" .../>
```

# Topics in this session

---

- Bean Definition Inheritance
- Inner Beans
- Property Editors
- Importing Configuration Files
- Bean Naming
- **The p Namespace**
- The util Namespace
- Spring Expression Language

# Shortcut to specifying bean properties

---



- The p namespace allows properties to be set as an attribute to the bean definition
- Before:

```
<bean name="example" class="com.foo.Example">
 <property name="email" value="foo@foo.com"/>
</bean>
```

- After:

```
<bean name="example" class="com.foo.Example"
 p:email="foo@foo.com"/>
```

# Shortcut to specifying bean references

---



- References to other beans require the format  
p:<property-name>-ref=<reference>
- Before:

```
<bean name="example" class="com.foo.Example">
 <property name="dataSource" ref="dataSource"/>
</bean>
```

- After:

```
<bean name="example" class="com.foo.Example"
 p:dataSource-ref="dataSource"/>
```

# Considerations of the p namespace

---



- Requires a namespace definition:

```
xmlns:p="http://www.springframework.org/schema/p"
```

- Namespace not specified in an XSD file, unlike other Spring namespaces
  - So no extra schemaLocation entry required

# Topics in this session

---

- Bean Definition Inheritance
- Inner Beans
- Property Editors
- Importing Configuration Files
- Bean Naming
- The p Namespace
- **The util Namespace**
- Spring Expression Language

# Spring's Util Namespace



- **<util:/>** is one of the “out-of-the-box” namespaces you can import
- Offers high-level configuration for common utility tasks
  - Populating collections
  - Accessing constants (static fields)
  - Loading properties
- Also possible to load collections using the default “beans” namespace
  - Collections in the “util” namespace propose some more advanced features

# Configuring a List

```
<bean id="inventoryManager" class="foo.MyInventoryManager">
 <property name="warehouses">
 <util:list> ←
 <ref bean="primaryWarehouse" /> List is an inner bean
 <ref bean="secondaryWarehouse" />
 <bean class="foo.Warehouse" .../>
 </util:list>
 </property> Elements are beans (one inner)
</bean>
```

```
<bean id="primaryWarehouse" .../>
<bean id="secondaryWarehouse" .../>
```

# Advanced configuration



```
<bean id="inventoryManager" class="foo.MyInventoryManager">
 <property name="warehouses">
 <util:list list-class="java.util.ArrayList"
 value-type="foo.Warehouse">
 <ref ... />
 </util:list>
 </property>
</bean>
```

List implementation  
instantiated by Spring

Default Java type for  
nested values

# Configuring a Set

```
<bean id="notificationService" class="foo.MyNotificationService">
 <property name="subscribers" ref="subscribers"/>
</bean>

<util:set id="subscribers">
 <value>larry@foo.com</value>
 <value>curly@foo.com</value>
 <value>moe@foo.com</value>
</util:set>
```

Set is a top-level bean.  
id is required

Elements are literal values

# Configuring a Map (1)

```
<bean id="inventoryManager" class="foo.MyInventoryManager">
 <property name="warehouses">
 <util:map>
 <entry key="primary" value-ref="primaryWarehouse" />
 <entry key="secondary">
 <bean class="foo.Warehouse" .../>
 </entry>
 </util:map>
 </property>
</bean>

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

Value is a top-level bean

Value is an inner bean

# Configuring a Map (2)

```
<bean id="inventoryManager" class="foo.MyInventoryManager">
 <property name="warehouses">
 <util:map>
 <entry>
 <key>
 <bean class="foo.WarehouseKey">
 <constructor-arg value="123456789" />
 </bean>
 </key>
 Key is an inner bean
 <bean class="foo.Warehouse" .../>
 </entry>
 </util:map>
 </property>
</bean>
```

# Accessing Constants

```
<bean id="notificationService"
 class="example.MyNotificationService">
 <property name="hostname">
 <util:constant static-field="example.MyConstants.HOST_NAME" />
 </property>
</bean>
```

Accesses static field in the MyConstants class

# Loading Properties

```
<bean id="notificationService"
 class="example.MyNotificationService">
 <property name="emailMappings">
 <util:properties location="classpath:mail.properties" />
 </property>
</bean>
```

Loads a java.util.Properties from the location

# Topics in this session

---

- Bean Definition Inheritance
- Inner Beans
- Property Editors
- Importing Configuration Files
- Bean Naming
- The p Namespace
- The util Namespace
- **Spring Expression Language**

# Spring Expression Language



- Spring 3.0 introduces new Expression Language
  - SpEL for short
- Use programmatically, with XML or annotations
- Inspired by WebFlow EL, superset of unified EL
  - Property & indexed collections access, incl. filtering and projection
  - Functions, operators, etc.
  - Can also call methods/constructors
- Pluggable/extendable by other Spring-based frameworks

# SpEL Context Attributes

---



- EL Attributes can be:
  - Named Spring beans
  - Implicit references
- Some global implicit references:
  - systemProperties
  - systemEnvironment
- More available in web applications
- Extensible through API
  - e.g. Spring Security adds 'principal'

# SpEL XML Example

```
<bean class="mycompany.RewardsTestDatabase">
 <property name="databaseName"
 value="#{systemProperties.databaseName} />
 <property name="username"
 value="#{dbProps.username}"/>
 <property name="keyGenerator"
 value="#{strategyBean.databaseKeyGenerator} />
</bean>

<util:properties id="dbProps"
 location="classpath:db-config.properties"/>

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

# Summary

---

- Spring offers many techniques to simplify configuration
  - We've seen just a few here
  - It's about expressiveness and elegance, just like code
- Don't repeat yourself!
  - Use **bean inheritance** if you're seeing repetitive XML
- Consider hiding low-level beans
  - Use **inner beans** where it's natural to do so
- Avoid monolithic configuration
  - Use **<import/>** to aid in modularizing your XML

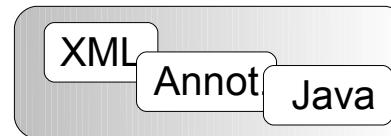
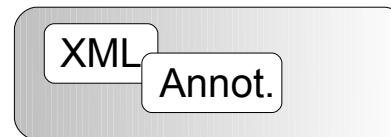
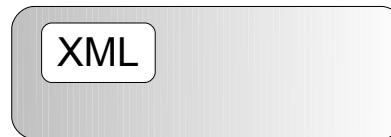


# Annotation- and Java-Based Configuration Styles

# History

---

- 2004: Spring 1.0
  - XML-based configuration
- 2007: Spring 2.5
  - Adds support for annotation-based configuration
- 2009: Spring 3.0
  - Adds support for Java-based configuration



# Topics in this Session

---

- **Annotation-based configuration**
- Component-scanning and best practices
- Standard annotations
- Java-based configuration
- Mixing configuration styles
- When to use what

# XML/Annotation DI



- XML-based dependency injection

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

External configuration

- Annotation-based dependency injection

```
@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

# Definitions: @Autowired



- **@Autowired**
  - Indicates that Spring should inject a dependency
    - Based on its type (default)
    - Based on its name if used with the **@Qualifier** annotation

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

```
@Autowired
public TransferServiceImpl(@Qualifier("accountRepository")
 AccountRepository repo) {
 this.accountRepository = repo;
}
```

# Definitions: Component scanning

---



- Component-scanning
  - Base-package scanned at startup
    - Sub-packages of the base-package also scanned
  - All classes annotated with @Component loaded as Spring beans
    - Some other annotations will also be discussed later

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

# Usage of @Autowired

- constructor-injection

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

Can also contain multiple params

- method-injection

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

Method name does not have to start with “set”

- field-injection

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

# 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

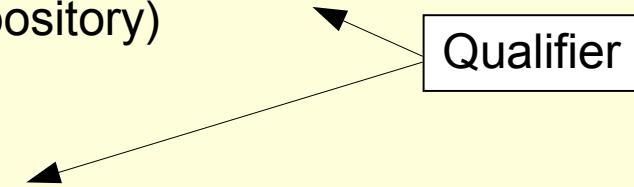


- Use of the @Qualifier annotation

```
@Component
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 can also be used with method injection and field injection

# Component names

---

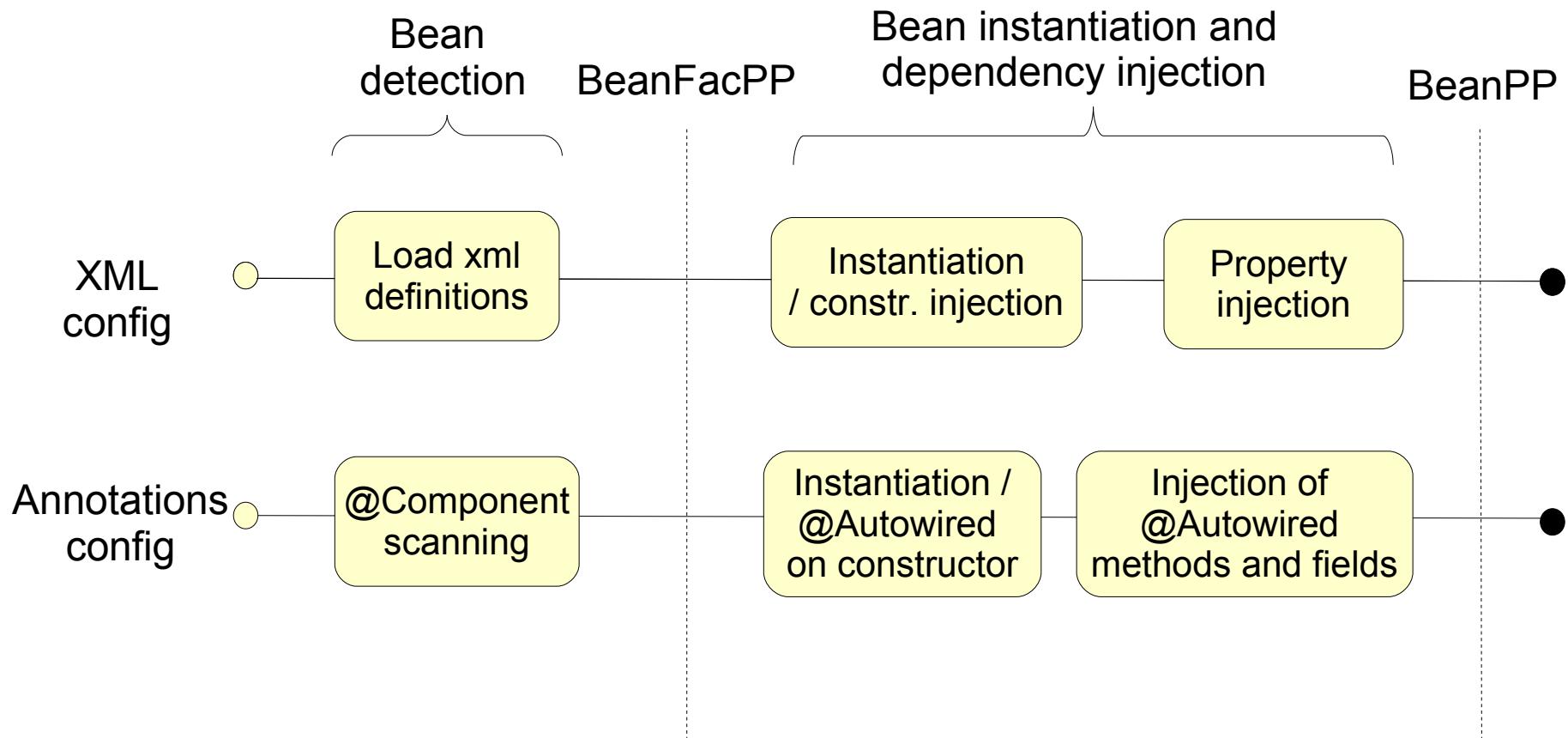
- When not specified
  - names are auto-generated
  - De-capitalized non-qualified names by default
  - Never rely on generated names!
- When specified
  - Less refactoring-friendly



Common strategy: avoid using names when possible.

*In many cases, it is not needed to have 2 implementations of the same bean in the ApplicationContext*

# Lifecycle



BeanFacPP →  
BeanPP →

BeanFactoryPostProcessor  
BeanPostProcessor

# XML vs Annotations syntax



- Same options are available

```
@Component("transferService")
@Component("transferService")
@Scope("prototype")
public class TransferServiceImpl
 implements TransferService {
```

```
@PostConstruct
public void init() { //... }

@PreDestroy
public void destroy() { //... }
```

*annotations*

<bean

```
id="transferService"
scope="prototype"
class="TransferServiceImpl"
```

init-method="init"

destroy-method="destroy" />

*xml config*

# Stereotype annotations

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

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

scans

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

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

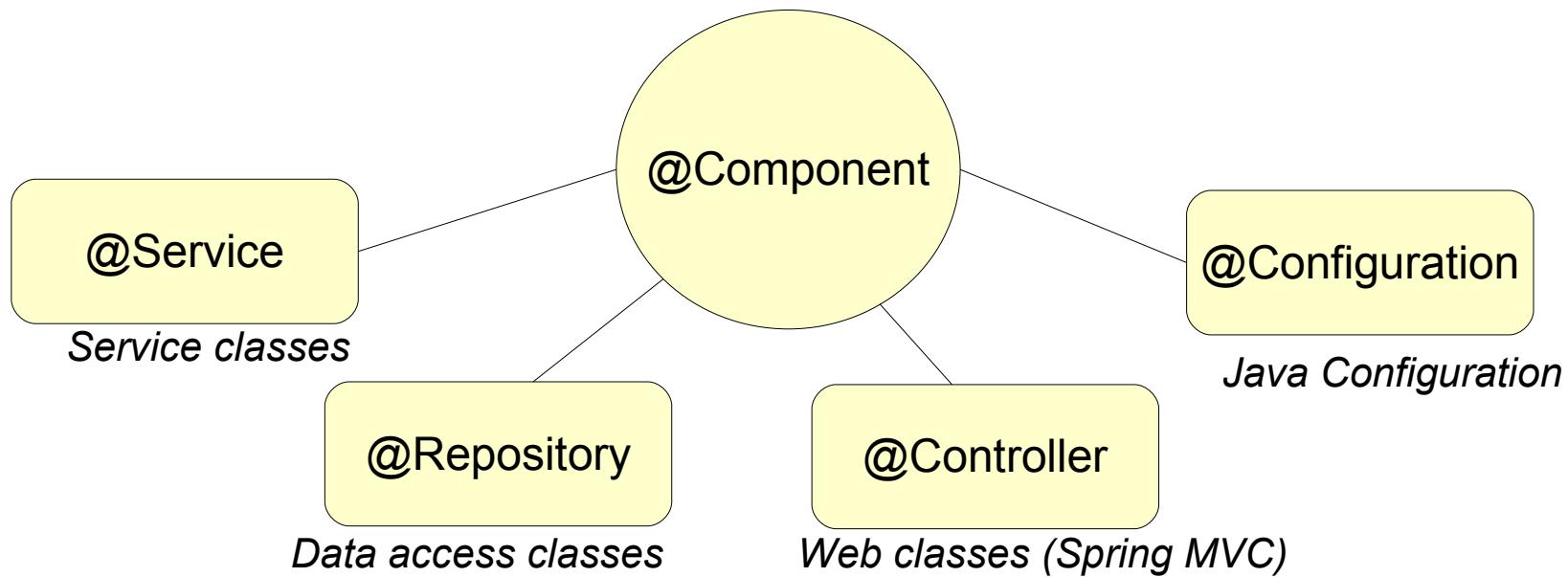
*Declaration of the @Service annotation*



@Service annotation is part of the Spring framework

# Stereotype annotations

- Spring framework stereotype annotations



Other Spring projects provide their own stereotype annotations  
(Spring WS, 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="com.springsource"/>
```

scans

recognizes

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

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

*Custom annotation*

# @Value and systemProperties



- Use @Value for SpEL expressions in Java code
  - On a field or method or on parameter of autowired method or constructor

```
@Value("#{ systemProperties['user.region'] }")
private String defaultLocale;

@Value("#{ systemProperties['user.region'] }")
public void setDefaultLocale(String defaultLocale) { ... }

@Autowired
public void configure(AccountRepository accountRepository,
 @Value("#{ systemProperties['user.region'] }") String defaultLocale) {
 ...
}
```



SpEL: Spring Expression Language (available since Spring 3.0)

# Topics in this Session

---

- Annotation-based configuration
- **Component-scanning**
- Standard annotations
- Java-based configuration
- Mixing configuration styles
- When to use what

# 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!!

- 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

---

- Annotation-based configuration
- Component scanning
- **Standard annotations**
- Java-based configuration
- Mixing configuration styles
- When to use what

- 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="com.springsource"/>
```

```
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	javax.inject.*	javax.inject restrictions
@Autowired	@Inject	@Inject has no 'required' attribute
@Component	@Named	Spring supports scanning for @Named
@Scope	@Scope	for meta-annotation and injection points only
@Scope ("singleton")	@Singleton	jsr-330 default scope is like Spring's 'prototype'
@Qualifier	@Named	
@Value	<i>no equivalent</i>	
@Required	<i>no equivalent</i>	
@Lazy	<i>no equivalent</i>	

# Topics in this Session

---

- Annotation-based configuration
- Component-scanning
- Standard annotations
- **Java-based configuration**
- Mixing configuration styles
- When to use what

# The Java-based Approach

---



- Spring 3.0 introduces Java-based configuration
  - Used to be a separate project called Spring JavaConfig
  - @Configuration-annotated classes are the key artifact
  - @Bean methods instead of <bean/> elements
- Use Java code to define and configure beans
  - Configuration still external to POJO classes to be configured
  - More control over bean instantiation and configuration
  - More type-safety

# @Configuration Example



```
@Configuration
```

```
public class RewardsConfig {
```

This method produces a bean definition

```
@Bean
```

```
public TransferService transferService() {
```

```
 TransferServiceImpl service = new TransferServiceImpl();
```

```
 service.setAccountRepository(accountRepository());
```

```
 return service;
```

```
}
```

bean id

Dependency injection

```
@Bean
```

```
public AccountRepository accountRepository() {
```

```
 //...
```

```
}
```

```
}
```

# Enabling @Configuration class processing



- 2 ways:

- Use component scanning

scans @Configuration annotated classes

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

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

- Use AnnotationConfigApplicationContext
    - no XML required *at all*

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

# Implementing @Bean Methods



- Configuration in Java: call any code you want
  - Not just constructors and property setters
- May not be private
- Method name becomes bean id
- Call @Bean method to refer to bean in same class
  - We'll show how to refer to external beans later
- Same defaults as XML: beans are singletons
  - Use @Scope annotation to change scope

```
@Bean @Scope("prototype")
public StatefulService service() {
```

```
...
```

# Implementing @Configuration Classes

---



- Must have default constructor
  - Implicit or not
- Multiple @Configuration classes may exist

```
@Configuration
public class RewardsConfig {
 //..
}
```

```
@Configuration
public class SecurityConfig { //.. }
```

# @Configuration internals (1)



- By default, Spring creates singletons out of @Bean annotated methods
  - How is it possible?

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

@Bean
public AccountRepository accountRepository() {
 AccountRepositoryImpl repo = new AccountRepositoryImpl();
 //...
 return repo;
}
```

This is a singleton!!

# @Configuration internals (2)

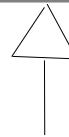


- Proxy created at startup time
  - Does all the magic
    - Stores beans in ApplicationContext, creates singletons...

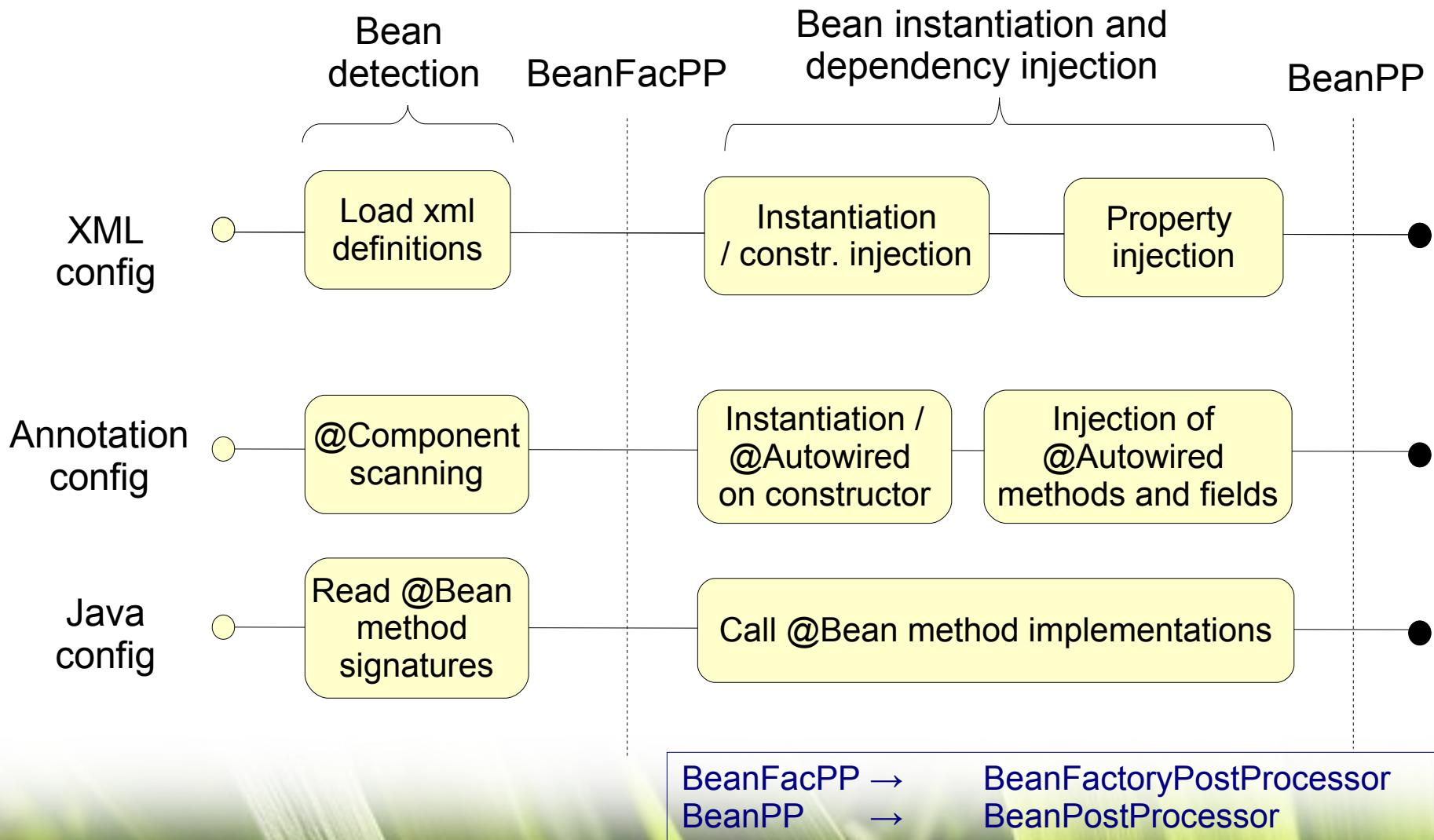
```
@Configuration
public class RewardConfig {
 @Bean
 public TransferService transferService() { //... }
}
```

```
public class ProxyName extends RewardConfig {
 public TransferService transferService() {
 // check if bean is in the applicationContext
 // if not, call super.transferService() and store bean in application Context
 }
}
```

Proxy generation uses  
CGLib by default



# Lifecycle



# Topics in this Session

---

- Annotation-based configuration
- Component-scanning and best practices
- Standard annotations
- Java-based configuration
- **Mixing configuration styles**
- When to use what

# Mixing configuration styles: Why?

---



- One single application can mix and match injection styles
  - XML-based
    - No major technical limitation but more verbose
  - Annotation-based
    - Can only annotate the classes you write
  - Java-based
    - does not have namespaces (context, tx, security...)

# Using annotations 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/>

 <bean id="transferService" class="transfer.TransferService"/>
 <bean id="accountRepository" class="transfer.JdbcAccountRepository"/>
</beans>
```



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

# Using annotations without component scanning (2)



- No @Component annotation anymore
  - Without Qualifier

```
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")
 AccountRepository accountRepository) { ... }
}
```

Refers to  
declared bean id

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

# Using annotations without component scanning (3)

---



- Java-based configuration: use of @Import
  - Equivalent to <import/> in XML
  - Means included classes don't have to be defined as Spring beans any more

```
@Configuration
@Import({SecurityConfig.class, JmxConfig.class})
public class RewardsConfig {
 ...
}
```

```
@Configuration
public class SecurityConfig {
 ...
}
```

# Mixing configuration styles: @Configuration example



- Bean datasource declared in XML

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

Declaration

- Used in @Configuration class

```
@Configuration
public class ApplicationConfig {
 @Autowired
 private DataSource datasource;

 @Bean
 public AccountRepository accountRepository() {
 AccountRepositoryImpl repository = new AccountRepositoryImpl();
 repository.setDataSource(this.datasource);
 return service;
 }
}
```

Dependency Injection

# Topics in this Session

---

- Annotation-based configuration
- Component scanning
- Standard annotations
- Java-based configuration
- Mixing configuration styles
- **When to use what**

# When to use what?

---

## Annotations:

- Nice for frequently changing beans
  - Start using for small isolated parts of your application (e.g. Spring @MVC controllers)
- Pros:
  - Single place to edit (just the class)
  - Allows for very rapid development
- Cons:
  - Configuration spread across your code base
  - Only works for your own code

# When to use what?

---

## 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:
  - Limited configuration options
  - Some people just don't like XML!

# When to use what?

---

## @Configuration classes:

- For full control over instantiation and configuration
  - As alternative to implementing FactoryBean
- Pros:
  - Configuration still external to code
  - Integrates well with legacy code
    - Can call methods such as *addDataSource(DataSource d)*
- Cons:
  - Wiring won't show up in STS
  - No equivalent to the XML namespaces

# 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` or `@Configuration` allows for almost empty XML configuration files



# LAB

Using Spring's annotations to Configure and  
test an application



# Testing Spring Applications

Unit Testing without Spring,  
and Integration Testing with Spring

# Topics in this session

---



- **Test Driven Development**
- Unit Testing vs. Integration Testing
- Unit Testing with Stubs
- Unit Testing with Mocks
- Integration Testing with Spring

# 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

# TDD and Design

---



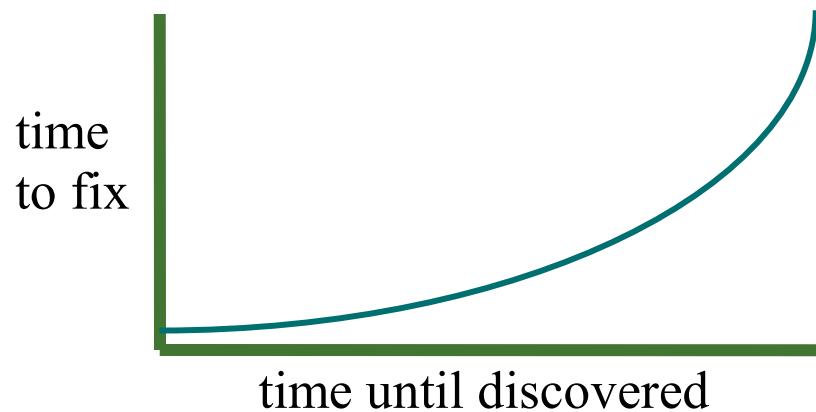
- Testing makes you think about your design
- If your code is hard to test then the design should be reconsidered

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

# 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**
- Unit Testing with Stubs
- Unit Testing with Mocks
- Integration Testing with Spring

# 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

---



- Verify a unit works in *isolation*
  - If A depends on B, A's tests should not fail because of a bug in B
  - A's test should not pass because of a bug in B
- Stub or mock out dependencies if needed
- Have each test exercise a single scenario
  - A *path* through the unit

# Example Unit to be Tested



```
public class AuthenticatorImpl implements Authenticator {
 private AccountDao accountDao;

 public AuthenticatorImpl(AccountDao accountDao) {
 this.accountDao = accountDao;
 }

 public boolean authenticate(String username, String password) {
 Account account = accountDao.getAccount(username);
 if (account.getPassword().equals(password)) {
 return true;
 } else {
 return false;
 }
 }
}
```

External dependency

Unit business logic (2 paths)

# Topics in this session

---



- Test Driven Development
- Unit Testing vs. Integration Testing
- **Unit Testing with Stubs**
- Unit Testing with Mocks
- Integration Testing with Spring

# Implementing a Stub



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

Simple state

# Testing with a Stub



```
public class AuthenticatorImplTests {

 private AuthenticatorImpl authenticator;

 @Before public void setUp() {
 authenticator = new AuthenticatorImpl(new StubAccountDao());
 }

 @Test public void successfulAuthentication() {
 assertTrue(authenticator.authenticate("lisa", "secret"));
 }

 @Test public void invalidPassword() {
 assertFalse(authenticator.authenticate("lisa", "invalid"));
 }
}
```

**Stub is configured**

**Scenarios are exercised based on the state of stub**

# Stub Considerations

---



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

# Topics in this session

---



- Test Driven Development
- Unit Testing vs. Integration Testing
- Unit Testing with Stubs
- **Unit Testing with Mocks**
- Integration Testing with Spring

# 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

# Mock Libraries



- EasyMock
  - very popular
  - used extensively in Spring
- Jmock
  - nice API, very suitable for complex stateful logic
- Mockito
  - uses a Test Spy instead of a true mock, making it easier to use most of the time

# Example - Testing with EasyMock



```
import static org.easymock.classextensions.EasyMock.*;

public class AuthenticatorImplTests {
 private AccountDao accountDao
 = createMock(AccountDao.class);
 private AuthenticatorImpl authenticator
 = new AuthenticatorImpl(accountDao);

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

On setup

- Create mock(s)
- Create unit

For each test

- Record expected method calls and set mock return values
- Call replay
- Exercise test scenario
- Call verify

# Mock Considerations

---



- 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
- 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
- Unit Testing with Stubs
- Unit Testing with Mocks
- **Integration Testing with Spring**

# 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

# Spring's Integration Test Support

---



- Packaged as a separate module
  - spring-test.jar
- Consists of several JUnit test support classes
- Central support class is SpringJUnit4TestRunner
  - Caches a shared ApplicationContext across test methods

# Annotations for Integration Testing

---



- Annotate the test with `@ContextConfiguration`
- Optionally pass a String-array of config file locations
  - by default Spring loads \${classname}-context.xml from the same package
- Use `@Autowired` annotations as before

```
@ContextConfiguration("/transfer-config.xml")
@RunWith(SpringJUnit4ClassRunner.class)
public class AuthenticatorTests {
 @Autowired
 private Authenticator authenticator;
}
```

# Using Spring's test support



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:system-test-config.xml")
public final class AuthenticatorTests {
 @Autowired
 private Authenticator authenticator;

 @Test
 public void successfulAuthentication() {
 authenticator.authenticate(...);
 }
}
```

Run with Spring support

Point to system test configuration file

Define a dependency

Test the system as normal

# Multiple test methods



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:system-test-config.xml")
public final class AuthenticatorTests {
 @Autowired
 private Authenticator authenticator;

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

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

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 Integration 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

# Summary

---



- Testing is an essential part of *any* development
- Unit testing tests a class in isolation where 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



# LAB

## Testing Spring Applications



# 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

# 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 to the line of code that throws an `AccessDeniedException`.

# Symptom #2: Scattering

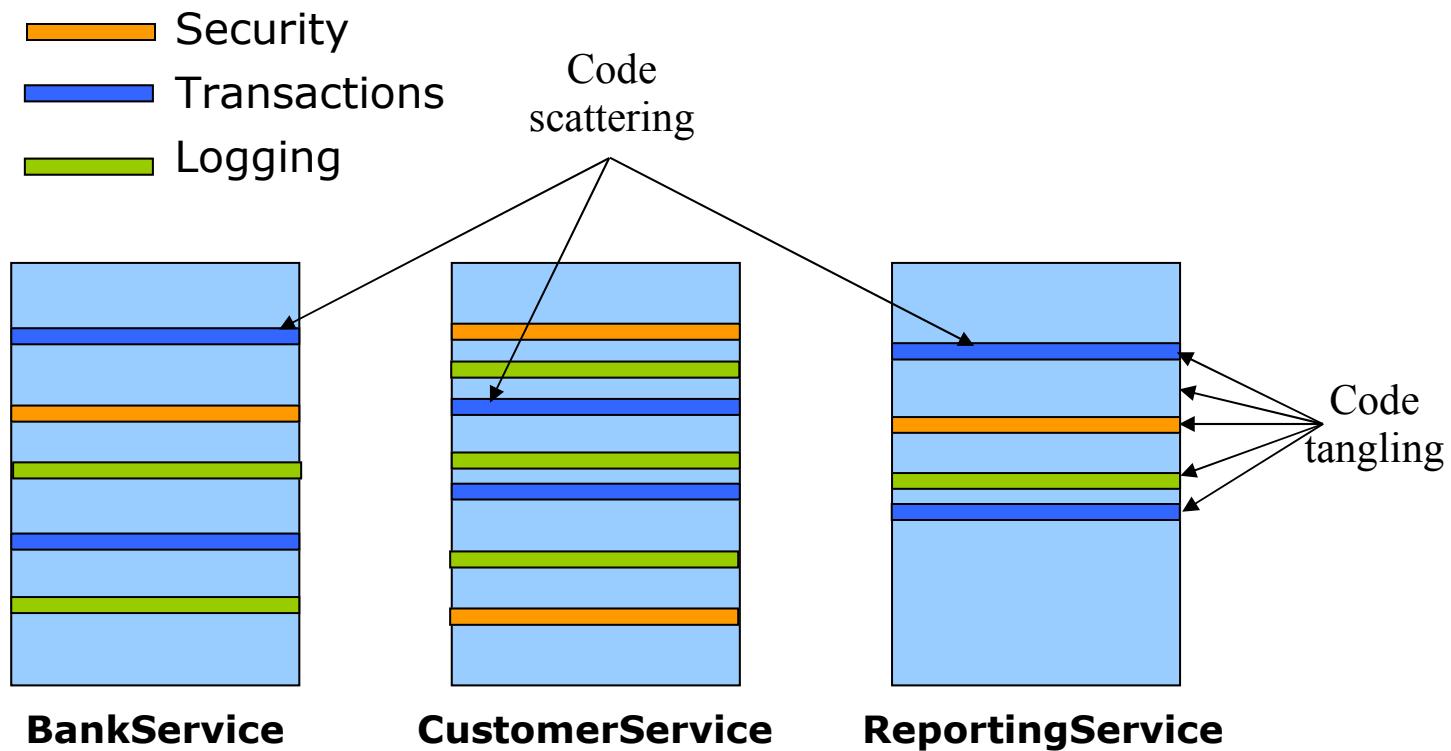


```
public class HibernateAccountManager implements AccountManager {
 public Account getAccountForEditing(Long id) {
 if (!hasPermission(SecurityContext.getPrincipal())) {
 throw new AccessDeniedException();
 }
 ...
 }
```

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

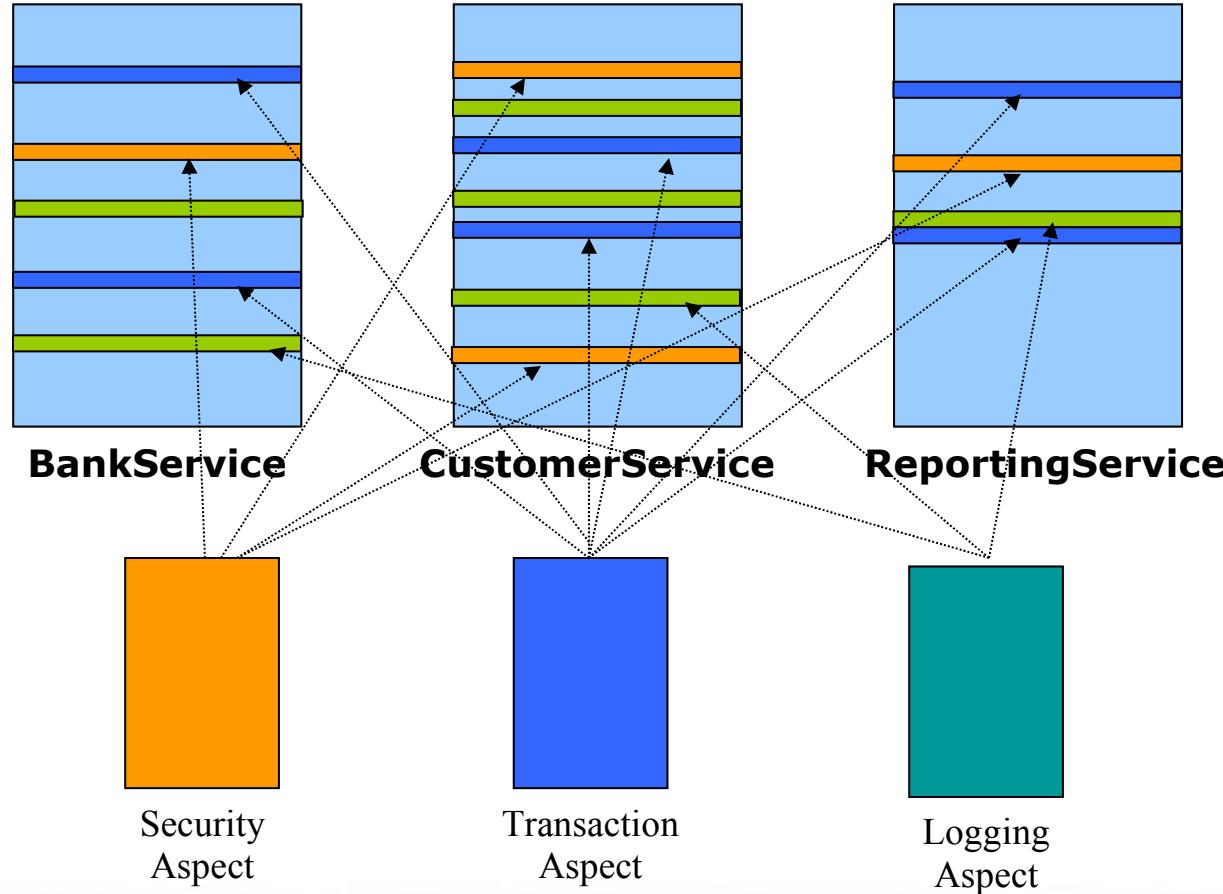
# How AOP Works

---



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



- 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; }

 public void setBeanName(String beanName) { name = beanName; }

 public String toString() { return name; }

 ...
}
```

# 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>
```

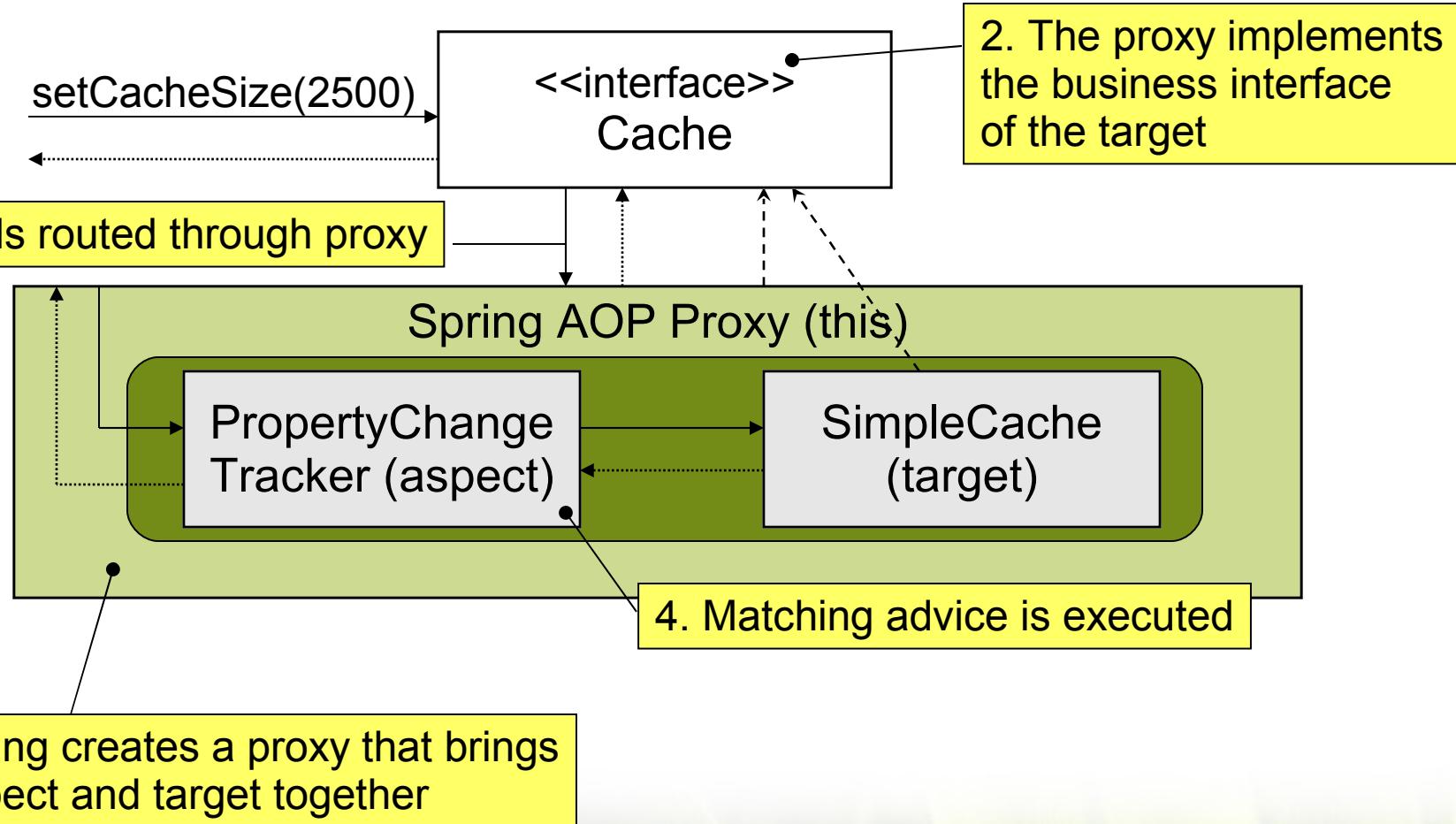
# Test the Application



```
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 is provided by the *JoinPoint* method 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

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

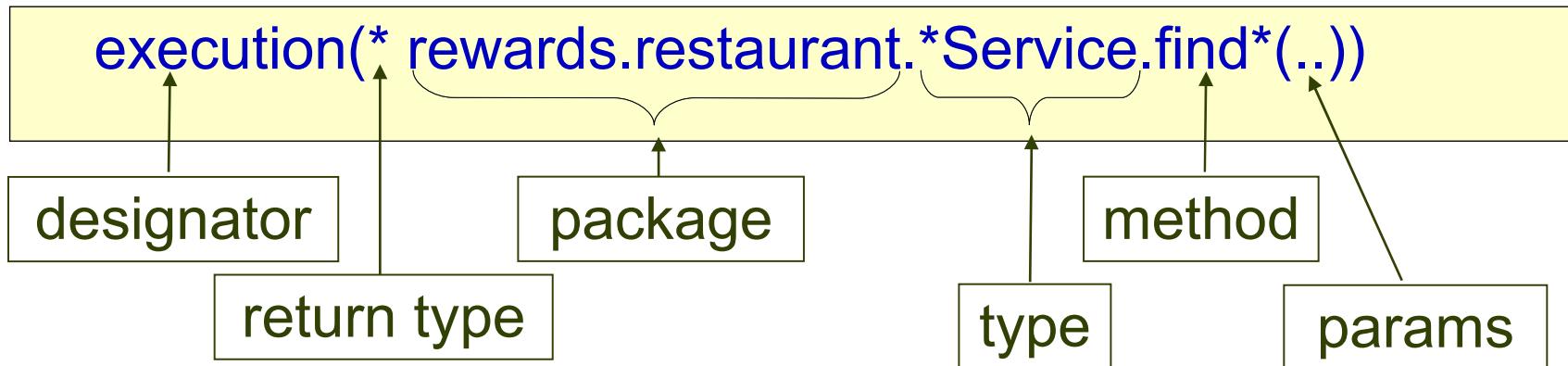
---



- 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>

- 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*

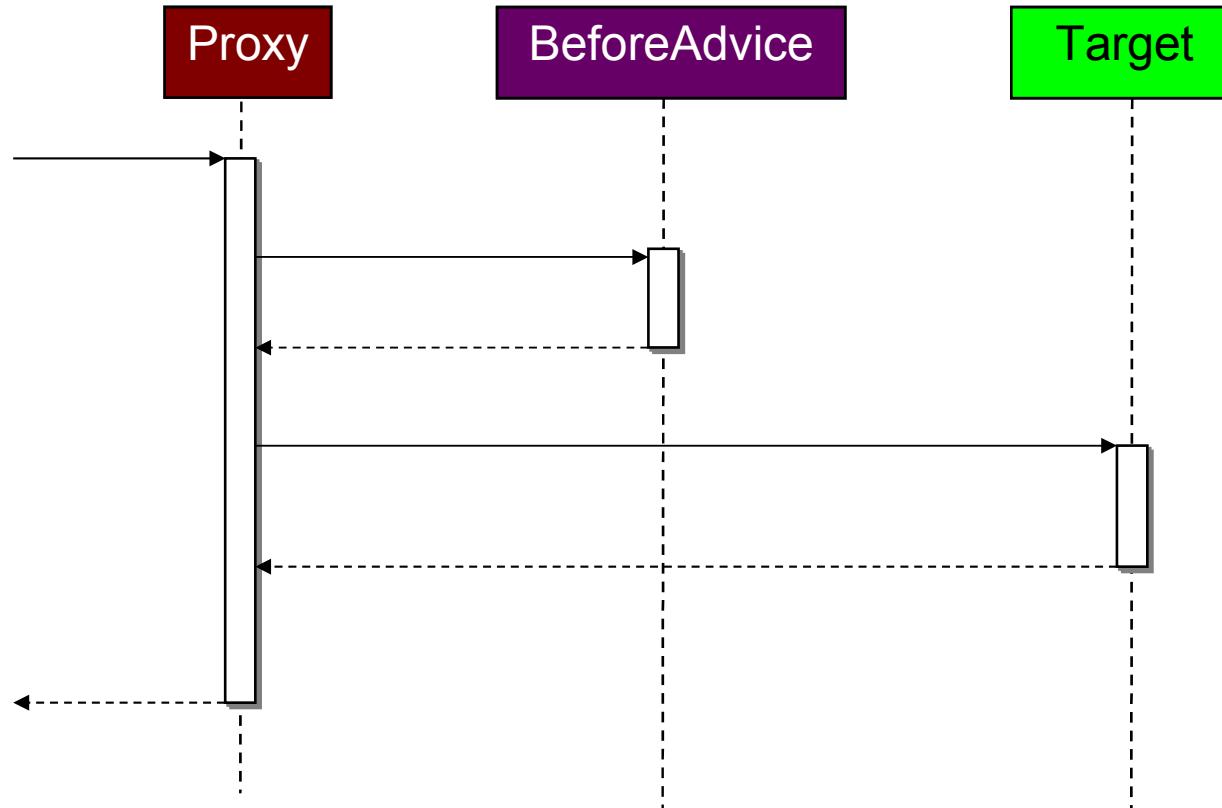
# Topics in this session

---



- 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



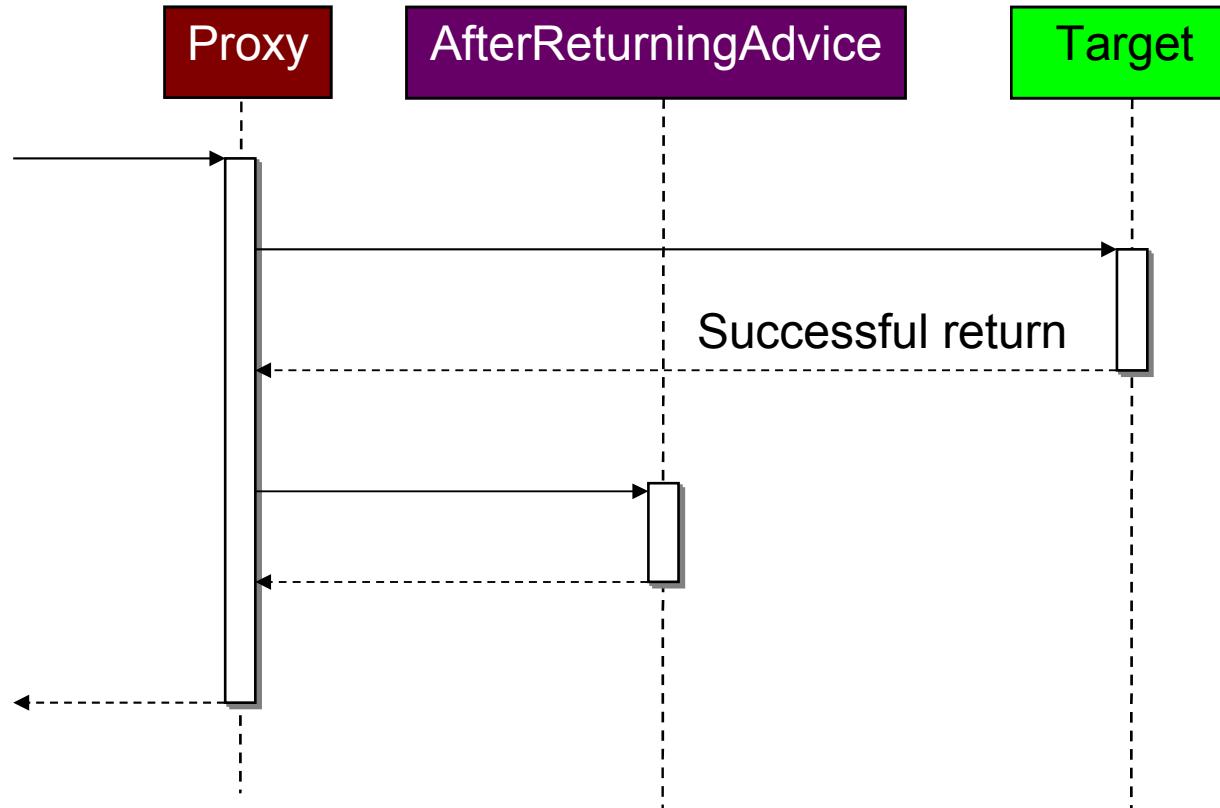
- Use `@Before` annotation
  - If the advice throws an exception, target will not be called

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

---

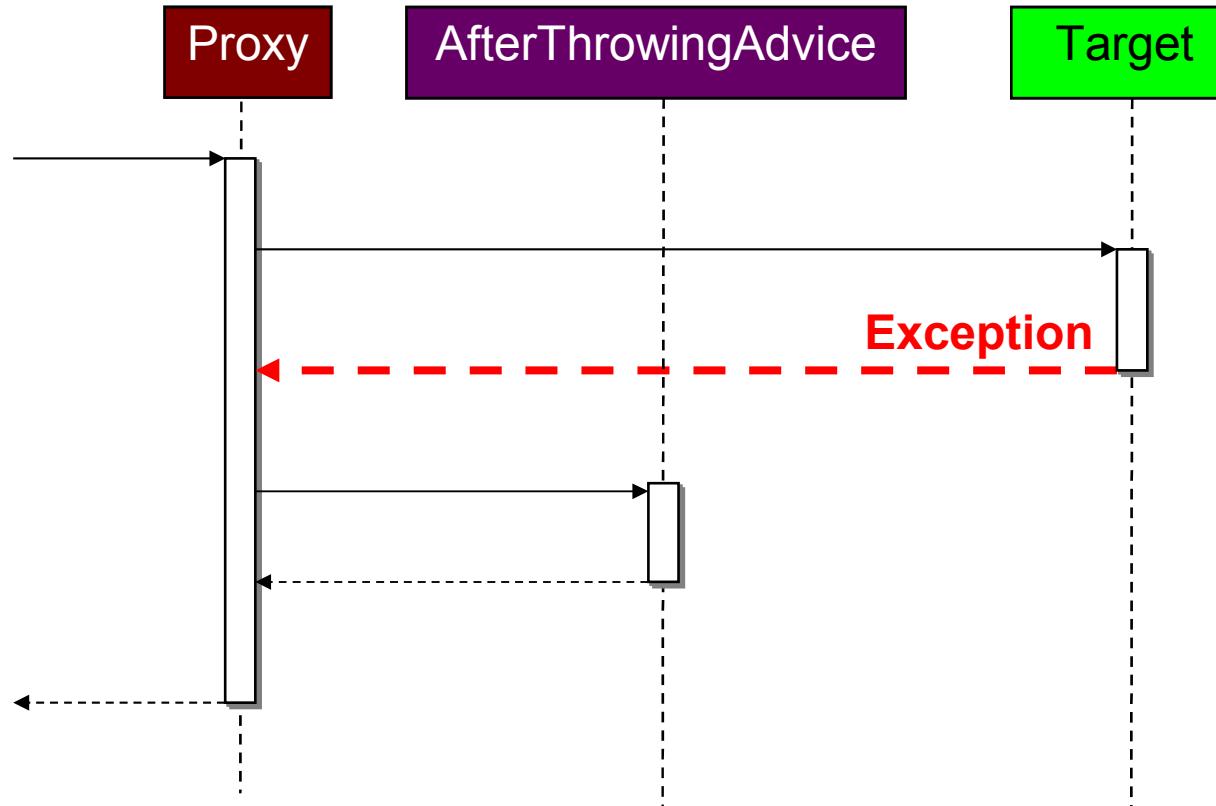


- Use `@AfterReturning` annotation with the *returning* attribute

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) {
 audit(jp.getSignature() + " returns a reward object ");
}
```

# 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

---



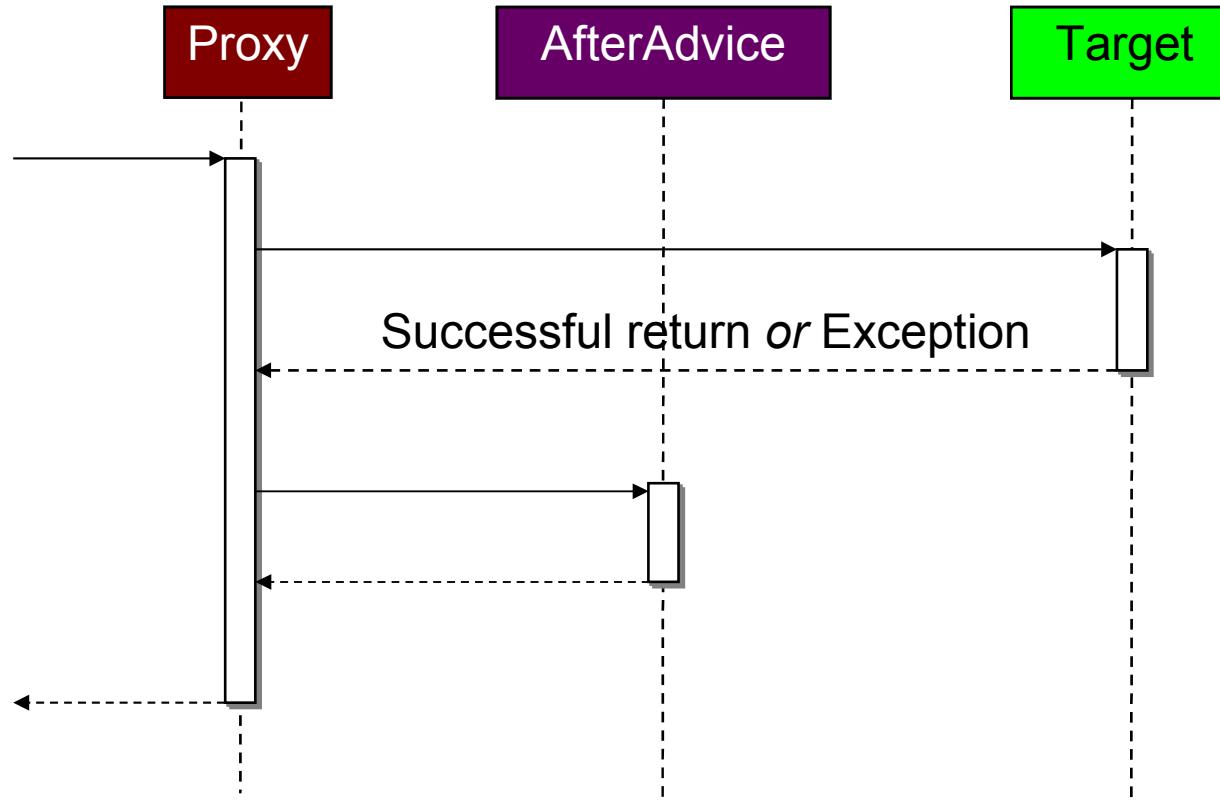
- The @AfterThrowing advice will not stop the exception from propagating
  - However it can throw a different type of exception

```
@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



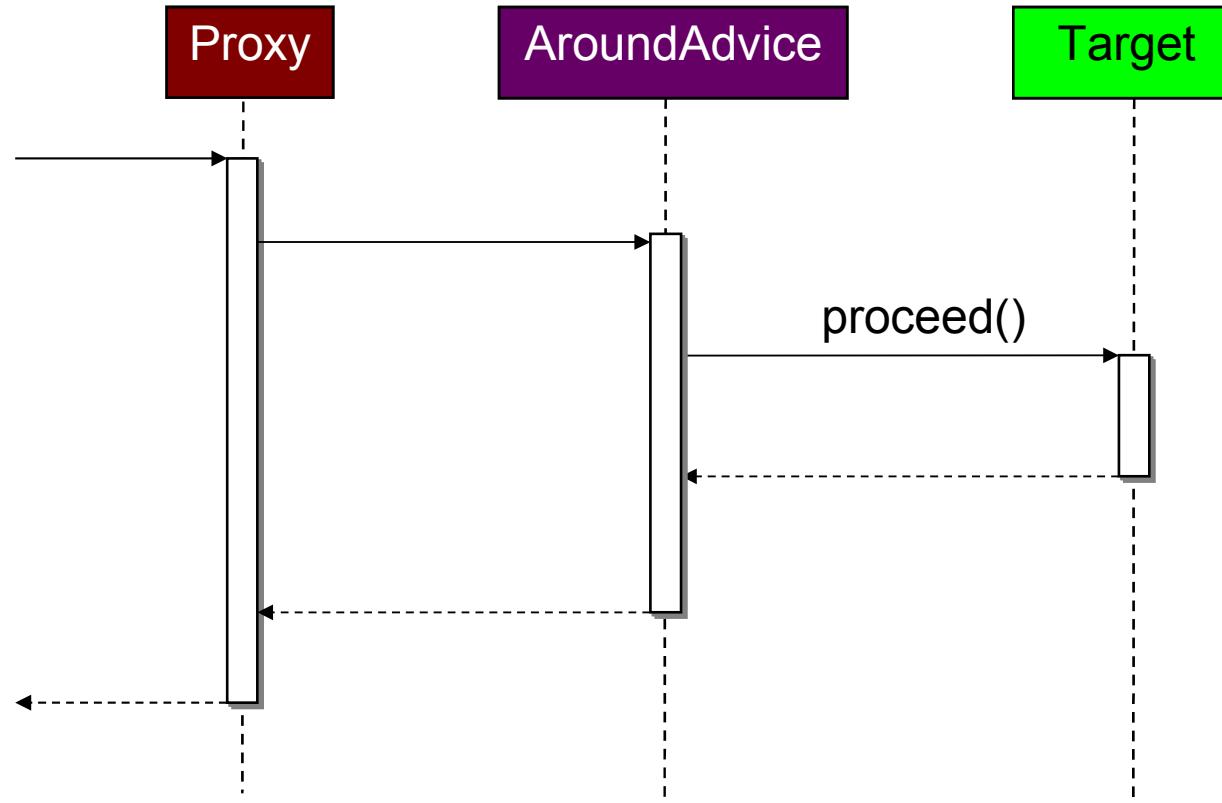
- Use `@After` annotation
  - Called regardless of whether an exception has been thrown by the target or not

Track calls to all update methods

```
@Aspect
public class PropertyChangeTracker {
 private Logger logger = Logger.getLogger(getClass());

 @After("execution(void update*(*))")
 public void trackChange() {
 logger.info("An update has been made...");
 }
}
```

# 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

# 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



- Allow 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

- Give the possibility to 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

- 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

```
@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

```
@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)")
```

The code above illustrates the use of named pointcuts. The first part shows a method definition with annotations. The second part shows a pointcut declaration. Two callout boxes with arrows point from specific parts of the code to their descriptions:

- A box labeled "'target' binds the server starting up" points to the `target(server)` part of the pointcut declaration.
- A box labeled "'args' binds the argument value" points to the `args(input)` part of the pointcut declaration.

```
public void serverStartMethod (Server server, Map input) {}
```

# 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

---



- `execution(@org.springframework.transaction.annotation.Transactional void *(..))`
  - Any method marked with the @Transactional annotation
- `execution( @example.Tracked *) *(..))`
  - Any method that returns a value marked with the @Tracked annotation

# Example



## Requirements:

Run a security check before any @Secured service operation

Security logic depends on:

- annotation attribute value – what roles to check?
- method name – what method is protected?
- the ‘this’ object – Spring AOP proxy being protected

```
@Secured(allowedRoles={"teller","manager"})
public void waiveFee () {
 ...
}
```

- Use of the *annotation()* designator

Run a security check before any @Secured service operation

```
@Before("execution(* service..*.*(..)) && target(object) &&
 @annotation(secured)")
public void runCheck(JoinPoint jp, Object object, Secured secured) {
 checkPermission(jp, object, secured.allowedRoles());
}
```

# AOP and annotations

## – Named pointcuts

---



Run a security check before any @Secured service operation

```
@Before("securedMethod(object, secured)")
public void runCheck(JoinPoint jp, Object object, Secured secured) {
 checkPermission(jp, object, secured.allowedRoles());
}
```

```
@Pointcut("execution(* service..*.*(..)) && target(object) &&
@annotation(secured)")
public void securedMethod(Object object, Secured secured) {}
```

# 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
- Some limitations of weaving with proxies
  - Spring will add 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 will revert to using CGLIB for weaving
  - When using proxies, if 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 are defined in Java with annotations or XML configuration



# Introduction to Data Access with Spring

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

# Topics in this Session

---



- **The role of Spring in enterprise data access**
- Spring resource management
- Spring data access support
- Data access in a layered architecture
- Common data access configurations

# The Role of Spring in Enterprise Data Access

---



- Provide comprehensive data access support
  - To make data access easier to do effectively
- Enable a layered application architecture
  - To isolate an application's business logic from the complexity of data access

# Making Data Access Easier

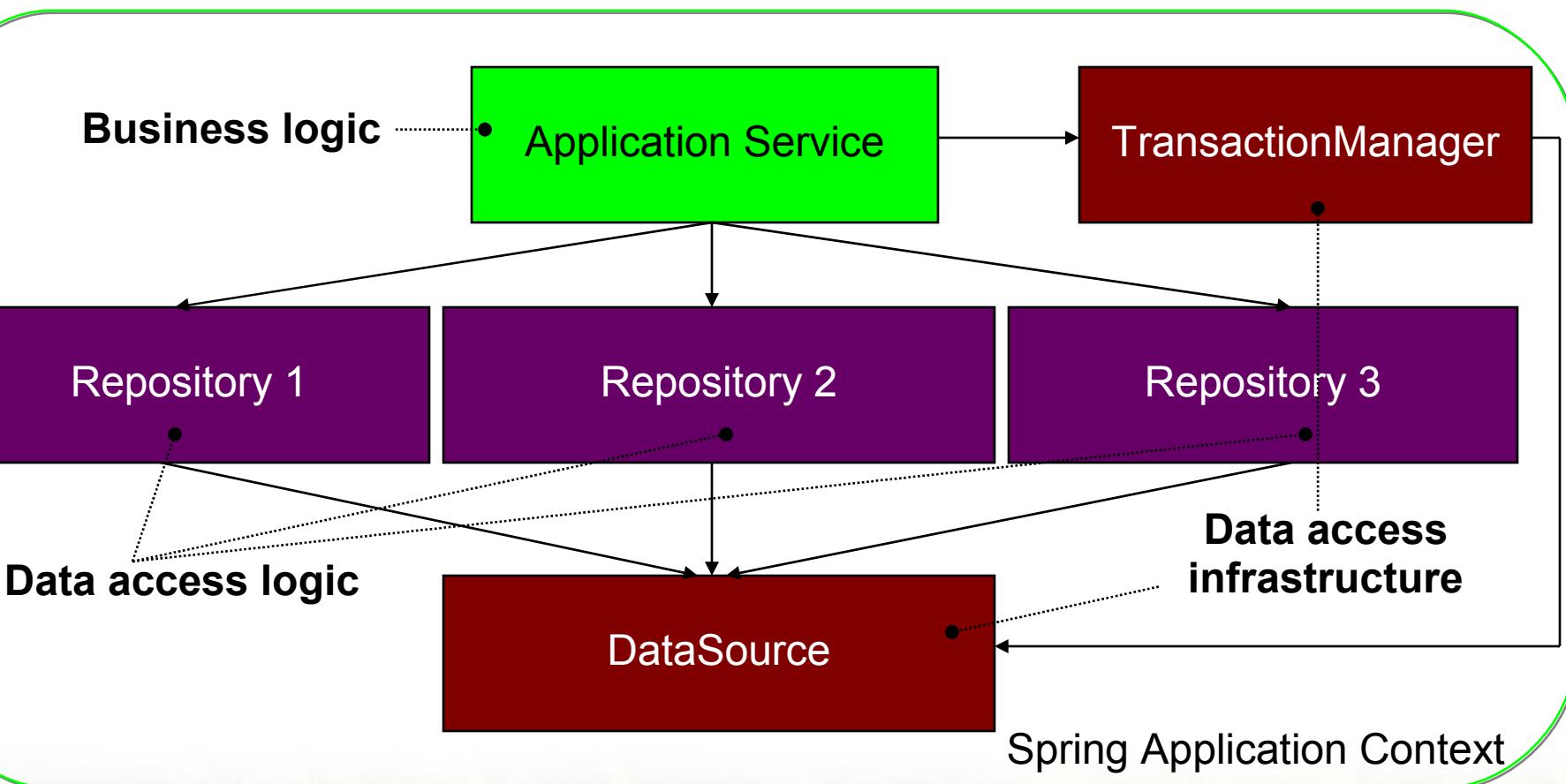


```
int count = jdbcHelper.queryForInt(
 "SELECT COUNT(*) FROM CUSTOMER");
```

- 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

# Enabling a Layered Architecture



# Topics in this Session

---



- The role of Spring in enterprise data access
- **Spring resource management**
- Spring data access support
- Data access in a layered architecture
- Common data access configurations

# Spring Resource Management

---



- Accessing external systems like a database requires resources
  - Limited resources must be managed properly
- Putting the resource management burden on the application developer is unnecessary
- Spring provides a comprehensive resource management solution

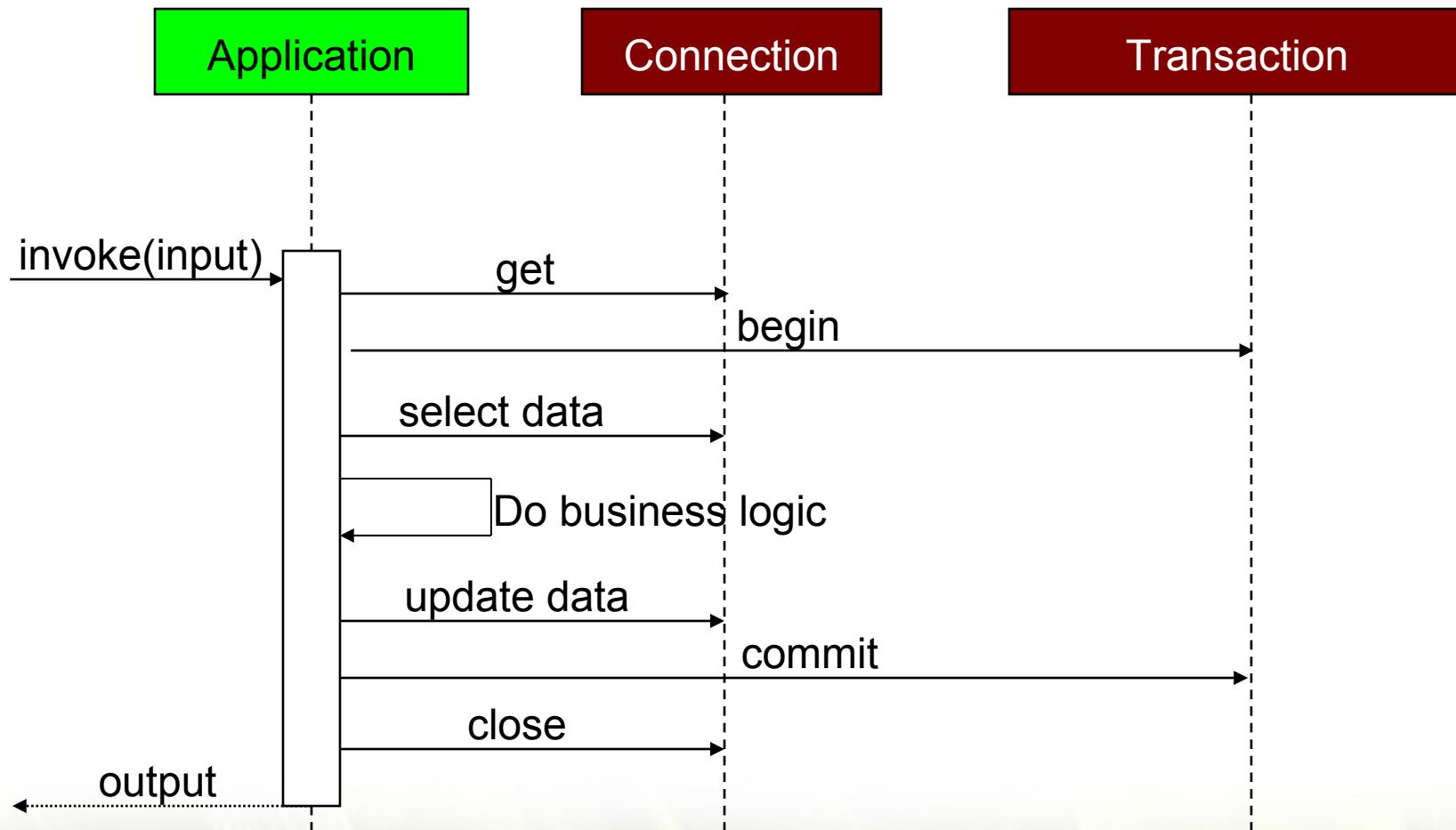
# The Resource Management Problem

---



- 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

# Effective Resource Management



# Managing Resources Manually

---



- In many applications resource management is a manual process
  - The responsibility of the application developer
- This is not an effective approach
  - Copy and pasting code
  - Introducing subtle, potentially critical bugs

# Not Effective - Copy & Pasting Code



```
doSomeWork(...) {
 try {
 get connection
 begin transaction
 execute SQL
 process the result set
 commit
 }
 catch (SQLException e) {
 rollback
 throw application exception
 } finally {
 try {
 close connection
 }
 catch {}
 }
}
```

Copied around everywhere

Unique

# Not Effective - Classic Memory Leak



```
doSomeWork(...) {
 try {
 get connection
 begin transaction
 execute SQL
 execute more SQL
 commit
 close connection
 }
 catch (SQLException) {
 rollback
 log exception
 }
}
```

FAILS

close never called

# Spring Resource Management

---



- Spring manages resources for you
  - Eliminates boilerplate code
  - Reduces likelihood of bugs
- Spring frees application developers from mundane responsibility
  - So they can get on with the real work

# Key Resource Management Features

---



- 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

# Spring Resource Management



## ■ BEFORE

```
doSomeWork(..) {
 try {
 establish connection
 begin transaction
 execute SQL
 process the result set
 commit
 }
 catch (SQLException) {
 rollback
 throw application exception
 } finally {
 try {
 close connection
 }
 catch {}
 }
}
```

# Spring Resource Management

---



■ **AFTER**

```
@Transactional
doSomeWork(..) {
 execute SQL
 process the result set
}
```

# Spring Resource Management Works Everywhere

---



- Works *consistently* with all leading data access technologies
  - Java Database Connectivity (JDBC)
  - JBoss Hibernate
  - Java Persistence API (JPA)
  - EclipseLink
  - Apache iBatis
- In any environment
  - Local (standalone app, web app)
  - Full-blown JEE container

# Topics in this Session

---



- The role of Spring in enterprise data access
- Spring resource management
- **Spring data access support**
- Data access in a layered architecture
- Common data access configurations

- Spring provides support libraries that simplify writing data access code
  - Promote consistency and code savings
- Support is provided for all the major data access technologies
  - Java Database Connectivity (JDBC)
  - Apache iBatis
  - JBoss Hibernate
  - Java Persistence Architecture (JPA)

# Spring Data Access Support - Scope

---



- For each data access technology, Spring's support consists of
  - A factory to help you configure the technology
  - A base support class to help you implement DAOs
  - A helper called a *data access template* to help you use the public API effectively

# JDBC DAO Support Example



```
public class JdbcCustomerRepository extends SimpleJdbcDaoSupport
implements CustomerRepository {
```

Support class provides  
DAO configuration assistance

```
 public int getCountOfCustomersOlderThan(int age) {
 String sql = "select count(*) from customer where age > ?";
 return getSimpleJdbcTemplate().queryForInt(sql, age);
 }
}
```

Data access helper simplifies use of API

# Topics in this Session

---



- The role of Spring in enterprise data access
- Spring resource management
- Spring data access support
- **Data access in a layered architecture**
- Common data access configurations

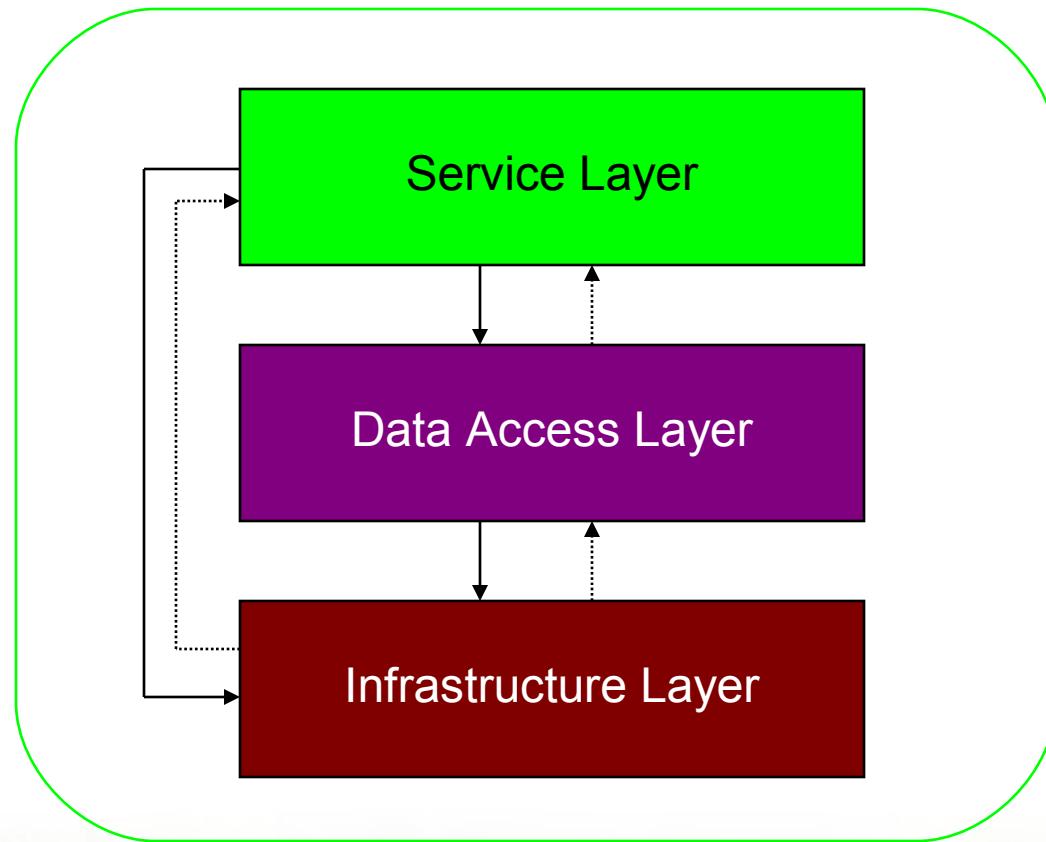
# Data Access in a Layered Architecture

---



- Spring enables layered application architecture
- Most enterprise applications consist of three logical layers
  - Service layer (or application layer)
    - Exposes high-level application functions
  - 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



# The Service Layer (Green)



- Defines the public functions of the application
  - Clients call into the application through this layer
- Encapsulates the logic to carry out each application function
  - Delegates to the infrastructure layer to manage transactions
  - Delegates to the data access layer to map persistent data into a form needed to execute the business logic

# The Data Access Layer (Purple)

---



- Used by the service layer to access data needed by the business logic
- Encapsulates the complexity of data access
  - The use of a data access API
    - JDBC, Hibernate, etc
  - The details of data access statements
    - SQL, HQL, etc
  - The mapping of data into a form suitable for business logic
    - A JDBC ResultSet to a domain object graph

# The Infrastructure Layer (Red)

---



- Exposes low-level services needed by other layers
  - Infrastructure services are used by the application
  - Application developers typically do not write them
  - Often provided by a framework like Spring
- Likely to vary between environments
  - Production vs. test

# Data Access Infrastructure Services - Transaction Manager

---



- Spring provides a transaction manager abstraction for driving transactions
- Includes implementations for all major data access technologies
  - JDBC
  - Java Persistence API (JPA)
  - Java Data Objects (JDO)
  - Hibernate
  - Toplink
- And the standard Java Transaction API (JTA)

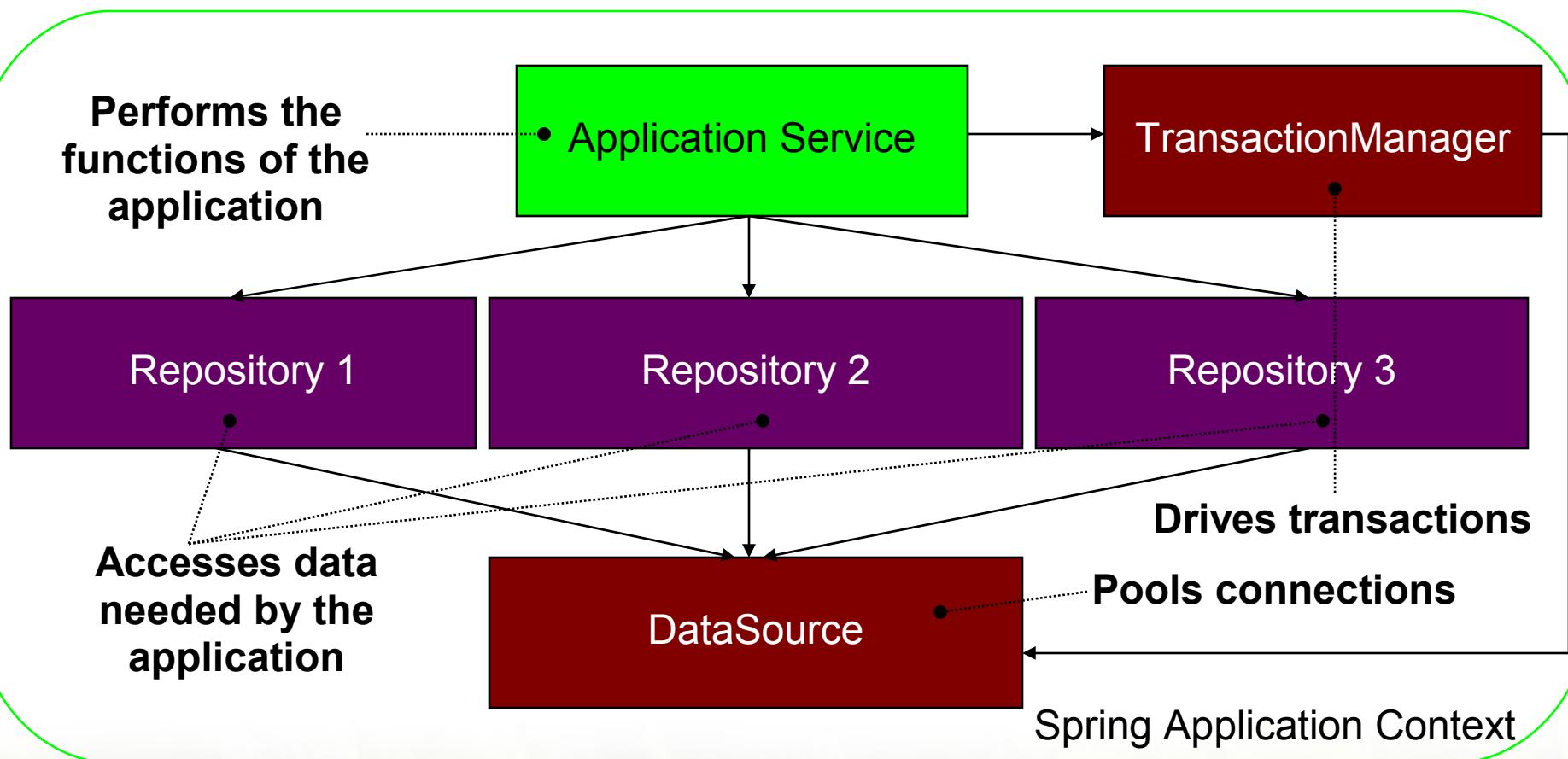
# Data Access Infrastructure Services - Data Source

---



- Spring uses a standard JDBC data source for acquiring connections
  - Provides several basic implementations
  - Integrates popular open source implementations
    - Apache DBCP, c3p0
  - Can integrate JEE container-managed data sources

# Spring Putting the Layers Together at Configuration Time



# The Layers Working Together at Use Time (1)

---



- A service initiates a function of the application
  - By delegating to a transaction manager to begin a transaction
  - By delegating to repositories to load data for processing
    - All data access calls participate in the transaction
    - Repositories often return domain objects that encapsulate domain behaviors

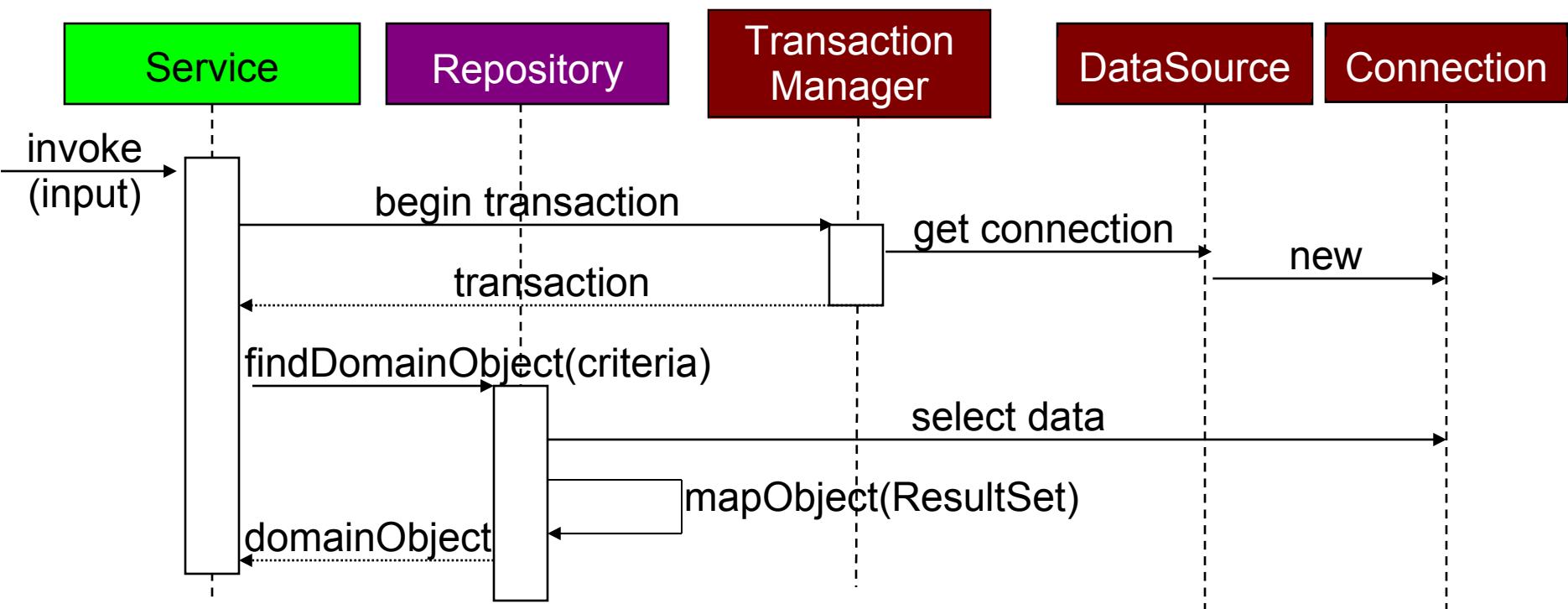
# The Layers Working Together at Use Time (2)

---

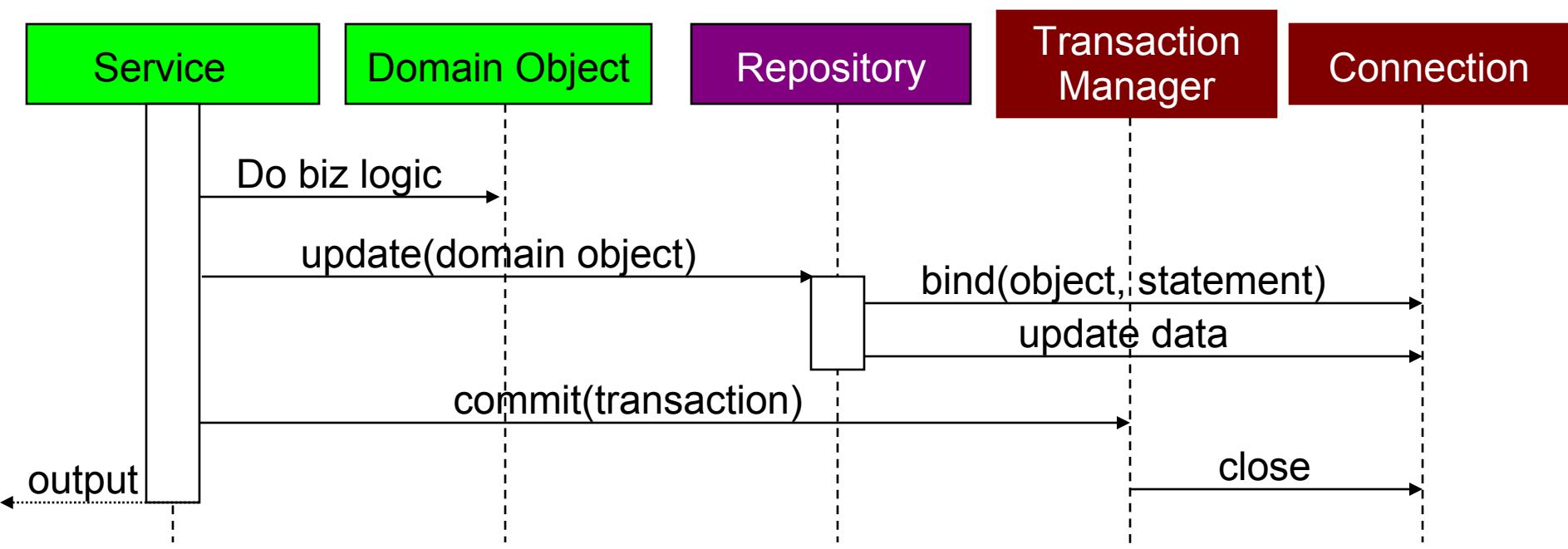


- A service continues processing
  - By executing business logic
  - Often by coordinating between domain objects loaded by repositories
- And finally, completes processing
  - By updating changed data and committing the transaction
  - Some technologies apply repository updates for you (ORMs)

# Application Service - Sequence (1)



# Application Service - Sequence (2)



# Declarative Transaction Management

---



- Spring adds the transaction management calls *around* your service logic
- You simply declare the transactional policies and Spring enforces them

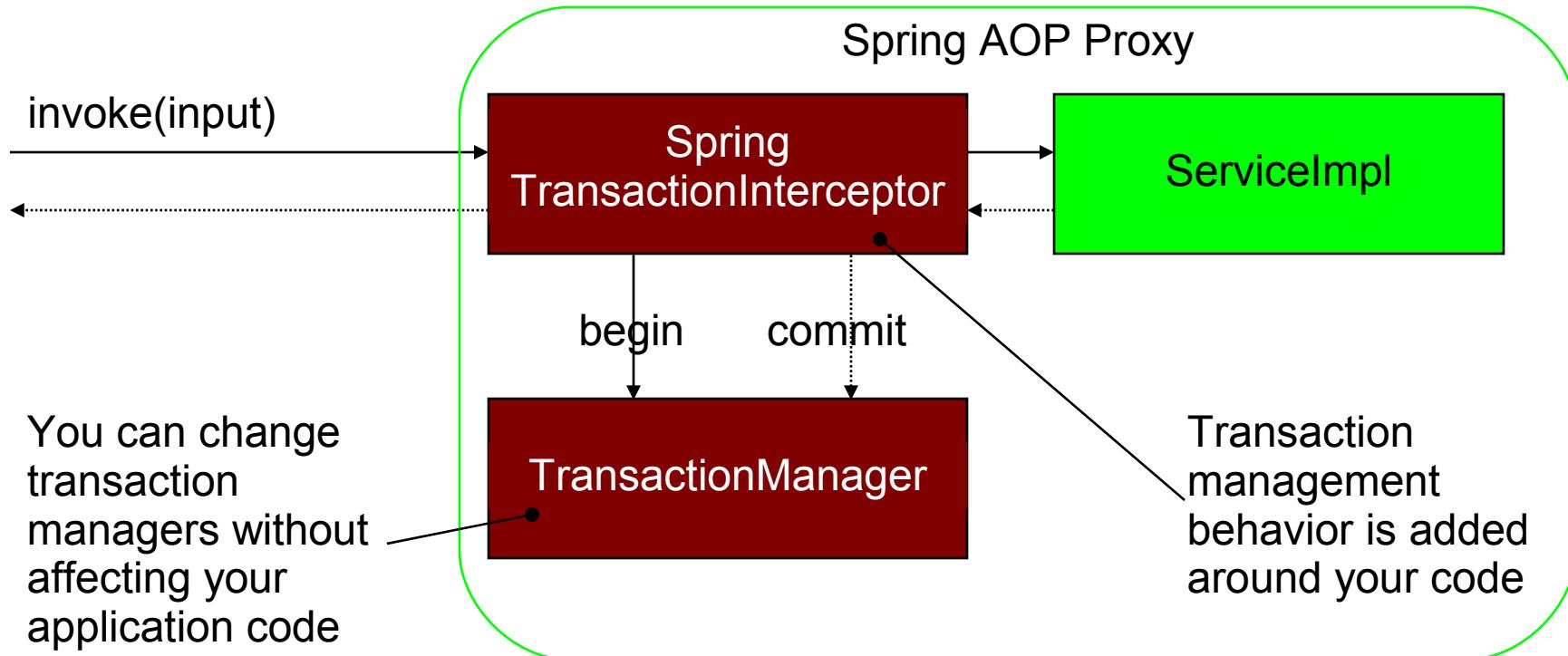
# Setting Transactional Policies Declaratively



```
public class ServiceImpl implements ServiceInterface {
 @Transactional
 public void invoke(...) {
 // your application logic
 }
}
```

Tells Spring to always run this method  
in a database transaction

# Enforcing Declarative Transactional Policies



# Effects on Connection Management

---



- Connections are managed for you
  - Driven by transactional boundaries
  - Not the responsibility of application code
  - No possibility of connection leak
- Repositories always use the transactional connection for data access
  - Spring provides several ways to ensure this

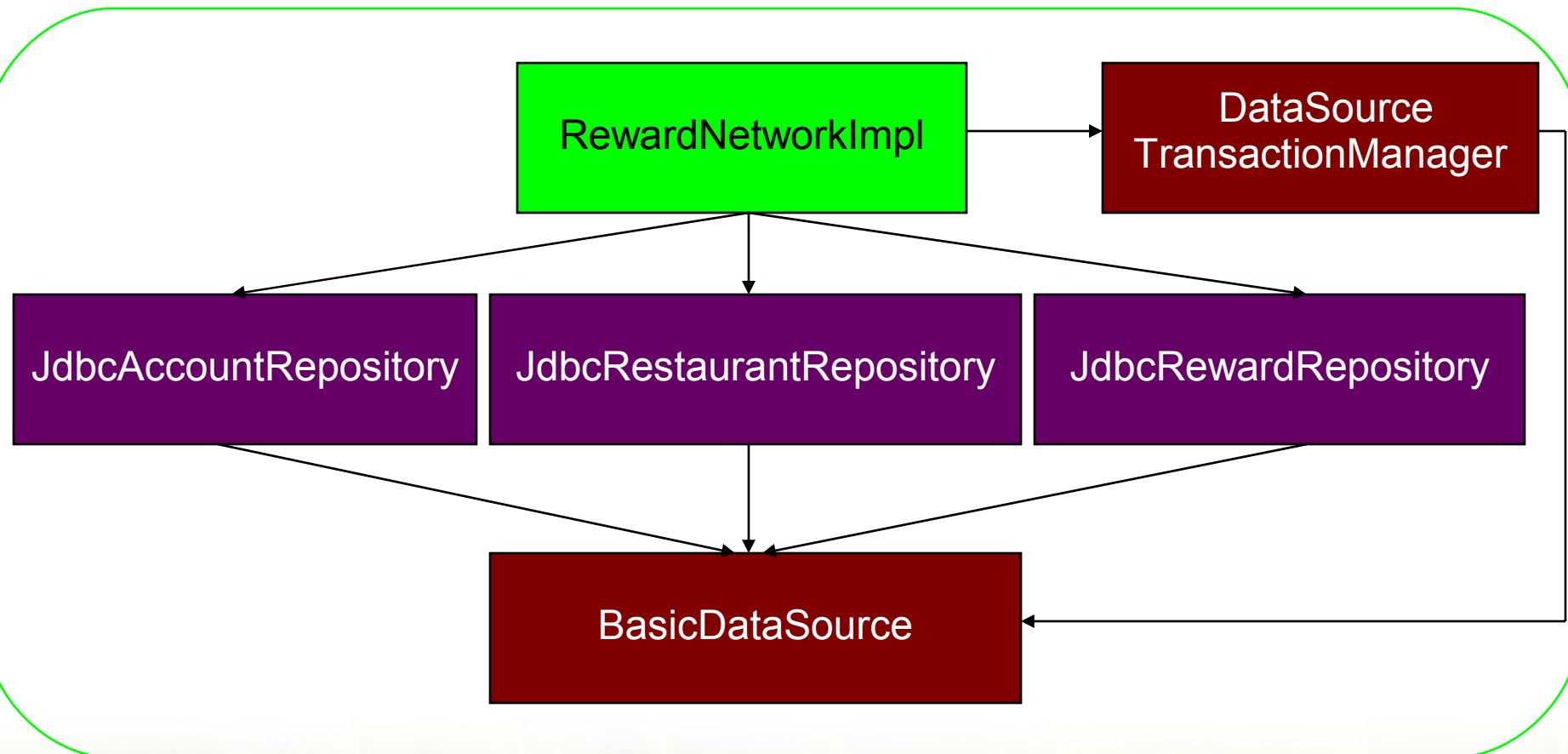
# Topics in this Session

---

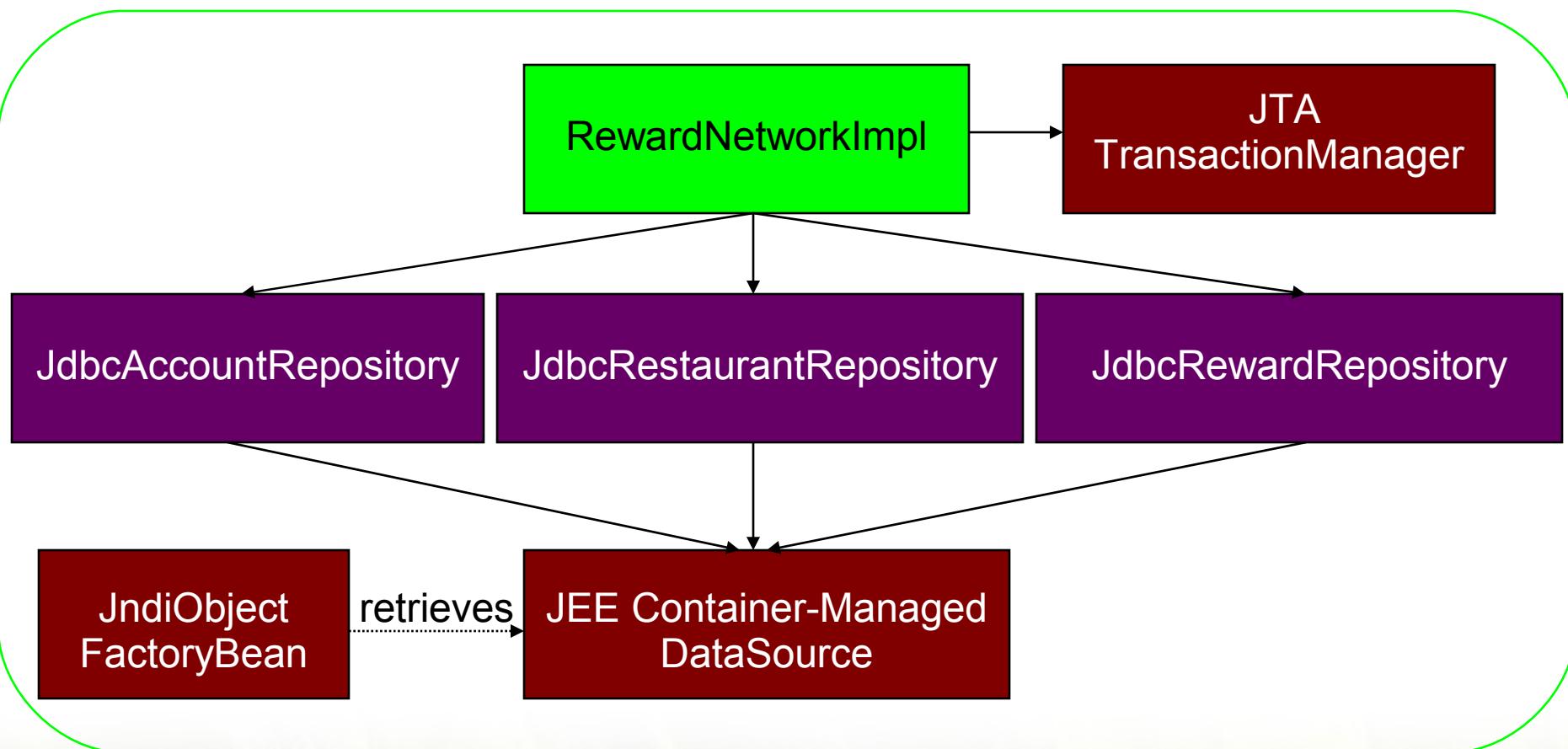


- The role of Spring in enterprise data access
- Spring resource management
- Spring data access support
- Data access in a layered architecture
- **Common data access configurations**

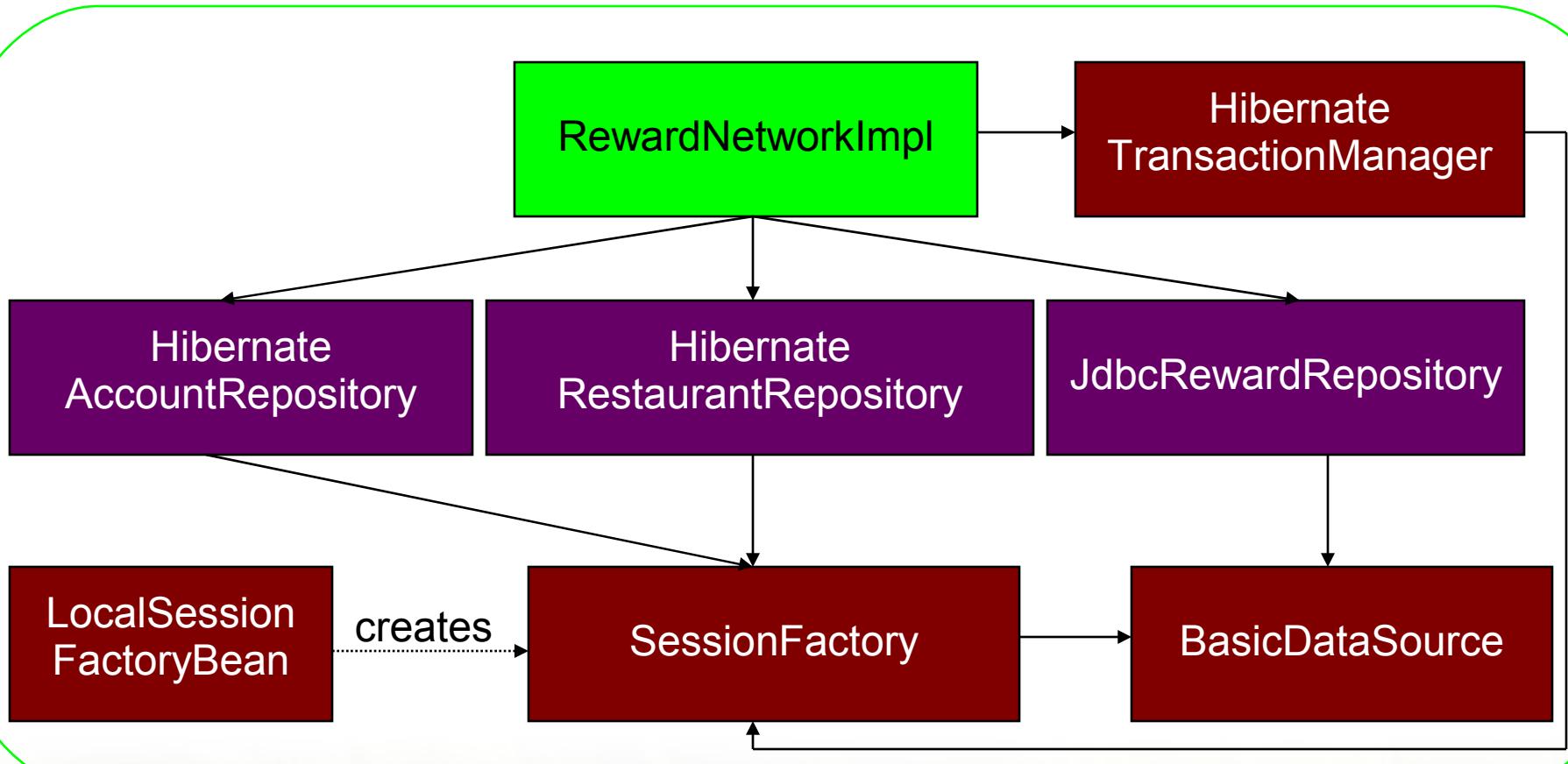
# Local JDBC Configuration



# JDBC Java EE Configuration



# Local Hibernate Configuration





# 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
- JDBC Namespace

# 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
- JDBC Namespace

# 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() {
 public Object mapRow(ResultSet rs, int row) throws SQLException {
 // map the current row to an object
 }
 });
```

```
class JdbcTemplate {
 public List 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);
 }
}
```

# 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" />
```

# 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

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

# 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

# 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

# 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

```
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"));
 }
}
```

Parameterizes return type

# 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

```
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 {
 // create an Order object with Items from multiple rows
 }
}
```

# 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());
}
```

# SQLException Handling

---



- SQLExceptions are not explicitly caught in most cases
- But, when they are, they can create a *leaky abstraction*
  - Portability suffers
  - Testability does too
- Using the JdbcTemplate ensures that SQLExceptions are handled in a consistent, portable fashion

# SQLException Handling Issues

---



- Hard to determine cause of failure
  - Must read a vendor-specific error code
- SQLException is a leaky abstraction
  - As a checked exception a SQLException must propagate if it is not caught
  - Leads to one of two things:
    - “Catch and wrap”
    - Being swallowed

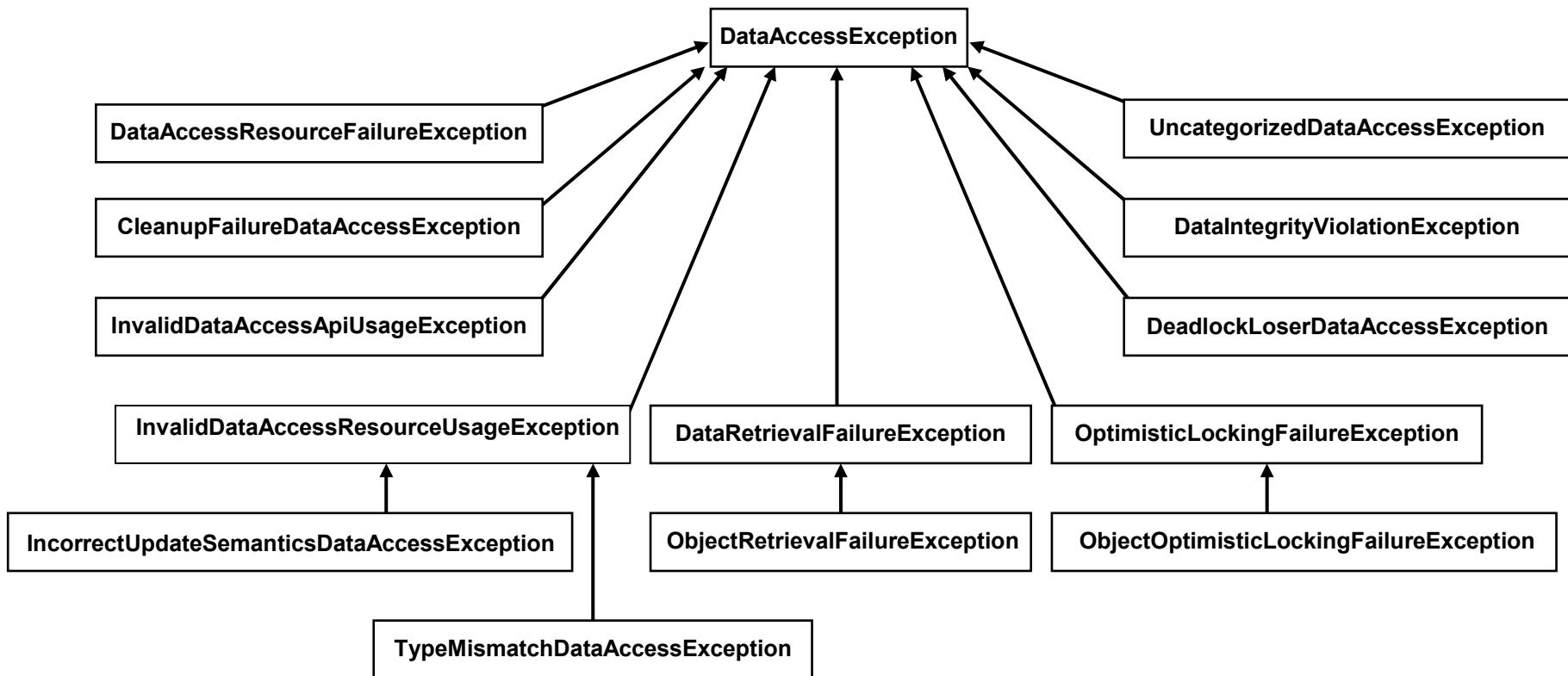
# Spring SQLException Handling

---



- SQLExceptions are handled consistently
  - Resources are always released properly
- Generic SQLExceptions are translated to root cause DataAccessExceptions
  - Provides consistency across all database vendors
  - Frees you from your own catch-and-wrap approach
  - Enables selective handling by type of failure
  - Exceptions always propagate if not caught

# Spring DataAccessException Hierarchy (subset)



# 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
- **JDBC Namespace**

# 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>
```

- 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>
```



# 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
- 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

---

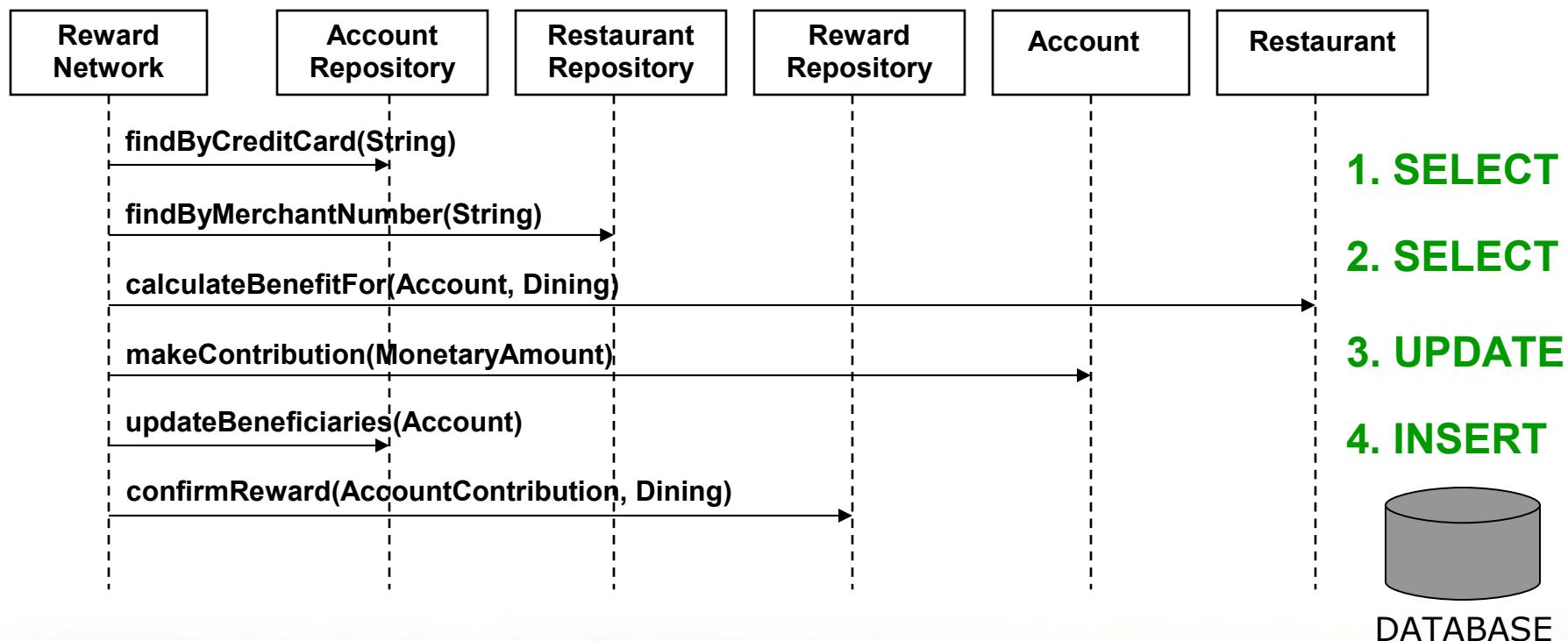


- The rewardAccountFor(Dining) method represents a unit-of-work that should be atomic

# RewardNetwork Atomicity



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



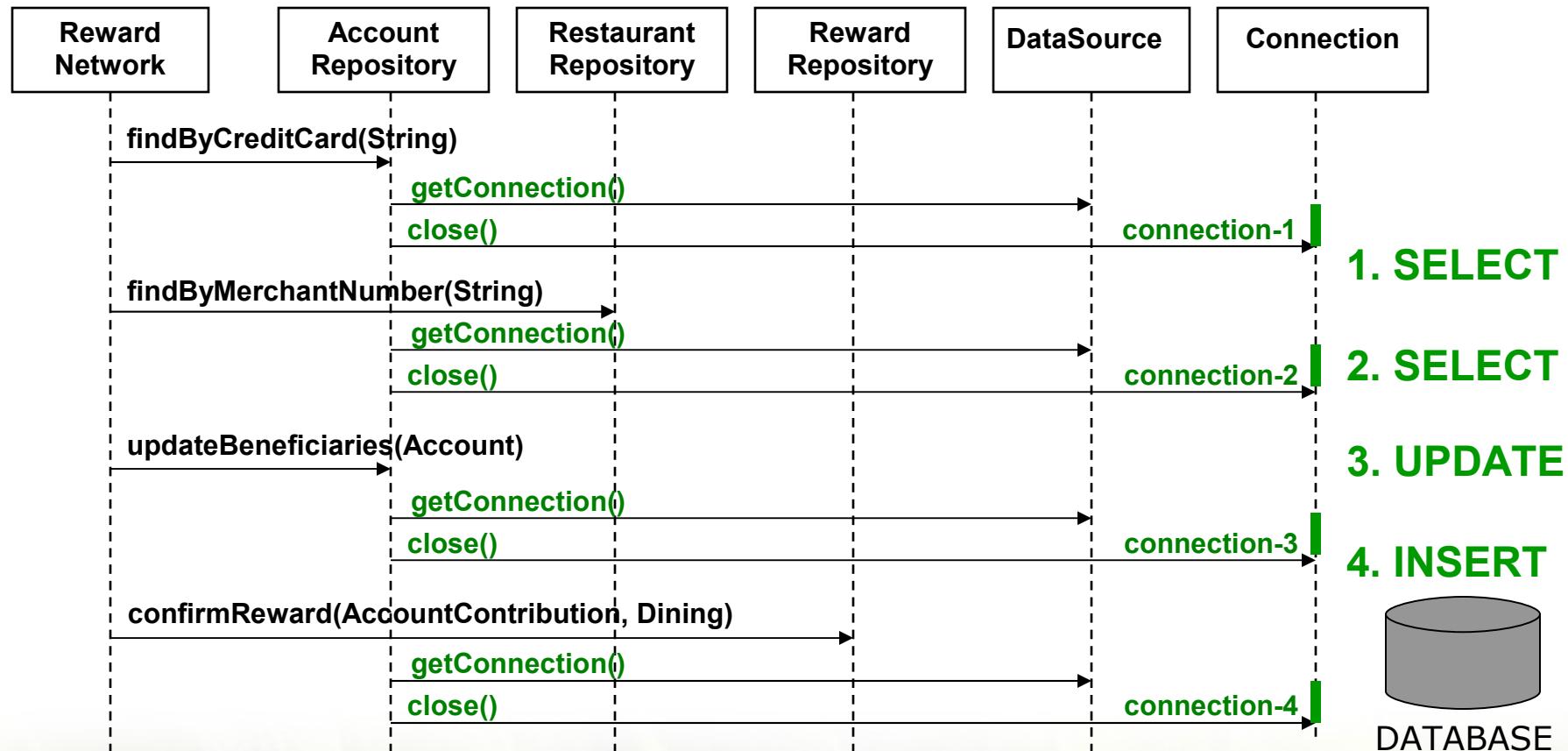
# Naïve Approach: Connection per Data Access Operation

---



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

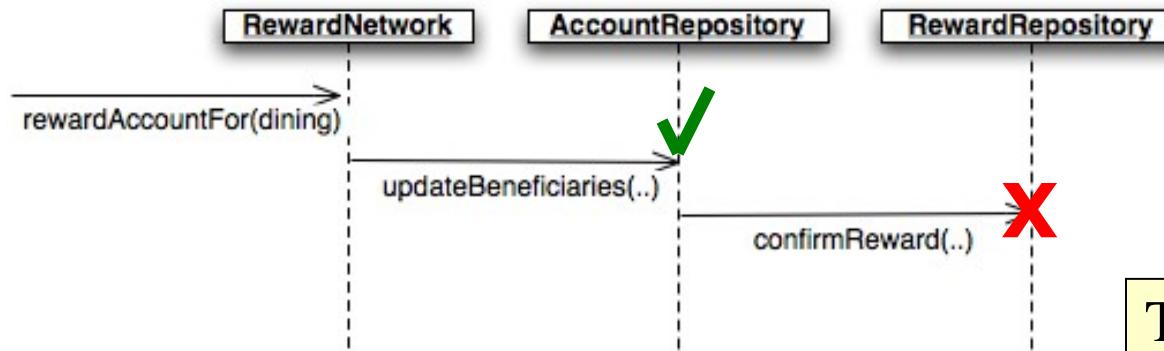
# Running non-Transactionally



# 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*

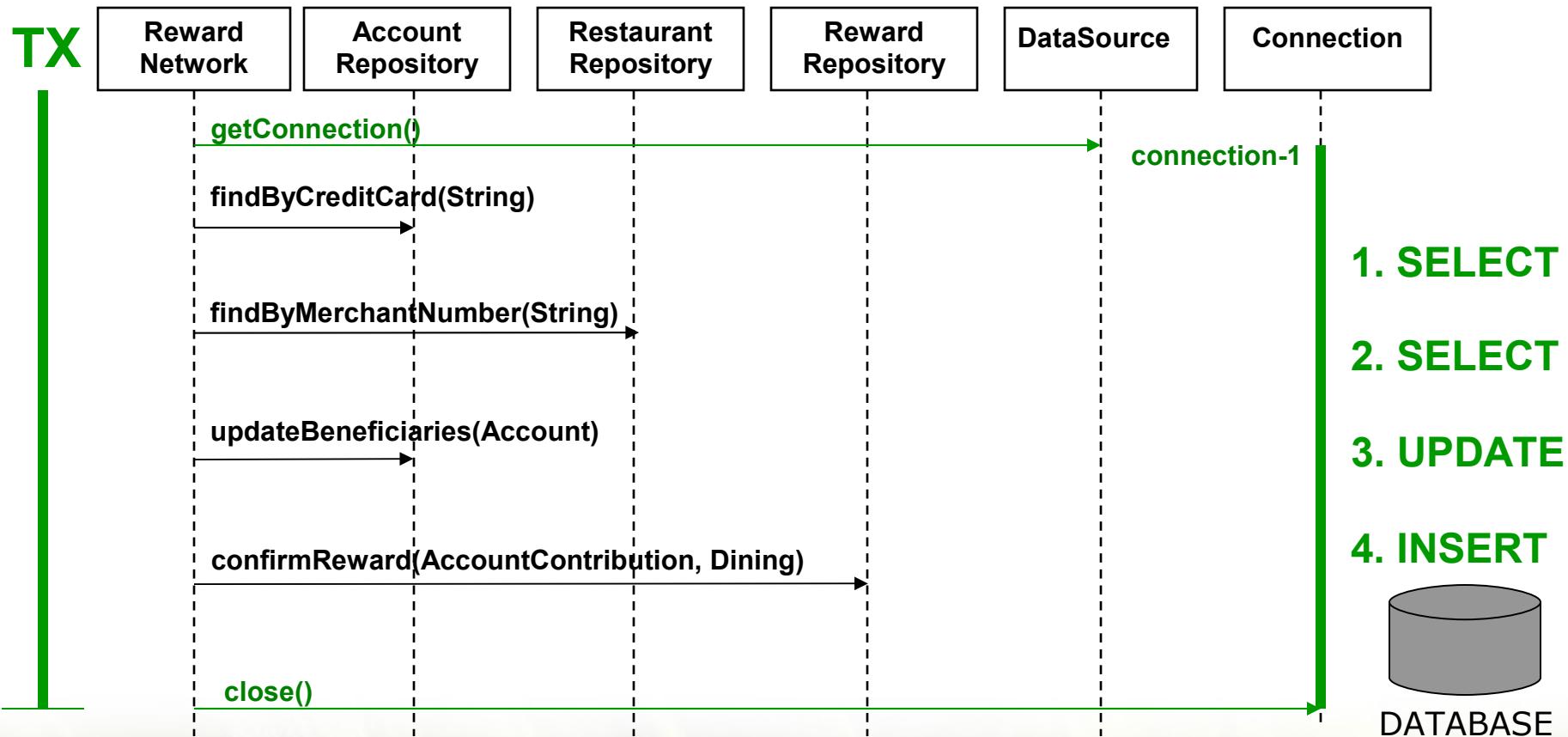
# Correct Approach: Connection per Unit-of-Work

---



- 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
- 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**
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

# Spring Transaction Management

---



- Provides a flexible and powerful abstraction layer for transaction management
- Several ways (can be mixed and matched)
  - XML
  - Annotations
  - Programmatic
- Optimization for local transactions
  - Database connection automatically bound to the current thread

- 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

# Declarative Transactions with Spring

---



- Declarative transactions are the recommended approach
- There are only 2 steps
  - Define a TransactionManager
  - Declare the transactional methods
- Spring supports both Annotation-driven and XML-based configuration

# @Transactional configuration



- 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>
```

# @Transactional: what happens exactly?

---



- 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)

- 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 at the 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(public * rewards..RewardNetwork+.*(..))"/>
 <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 pointcut expression

Method-level configuration for transactional advice

Includes rewardsAccountFor(..) and updateConfirmation(..)

# Topics in this session

---



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

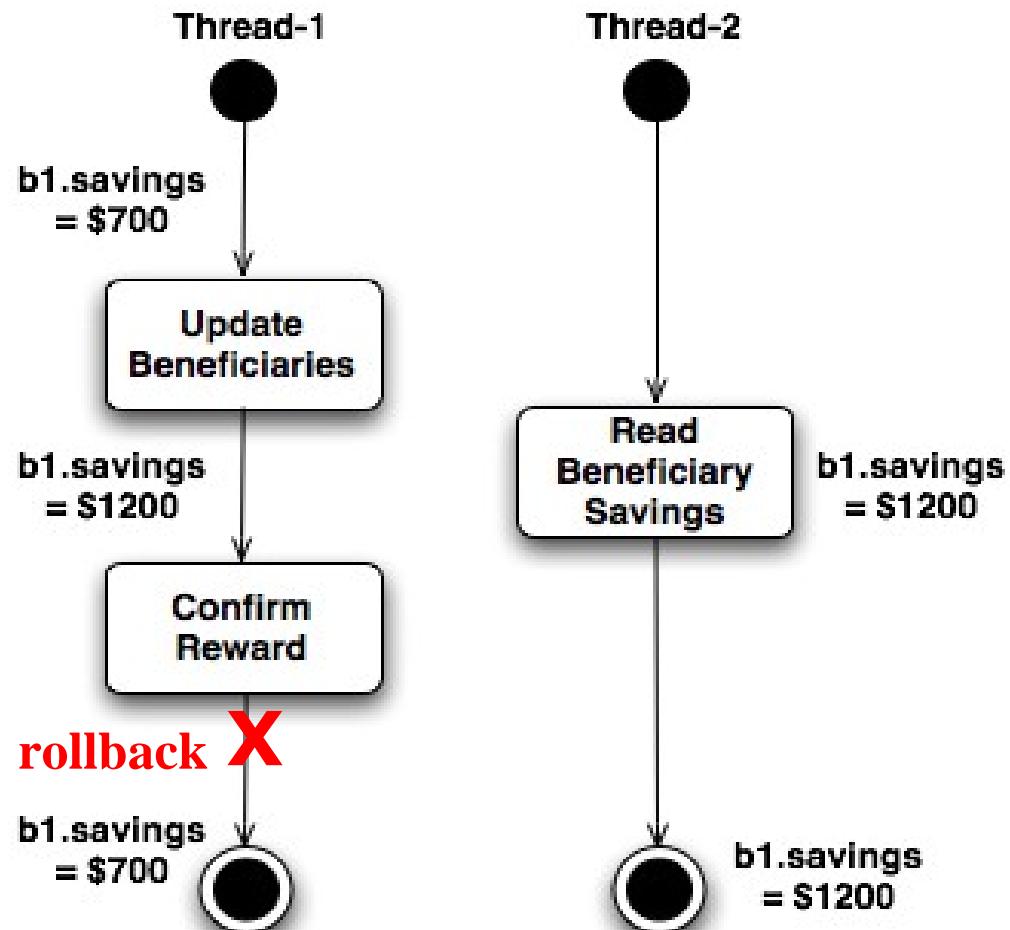
# Isolation levels

---

- 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
- 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?

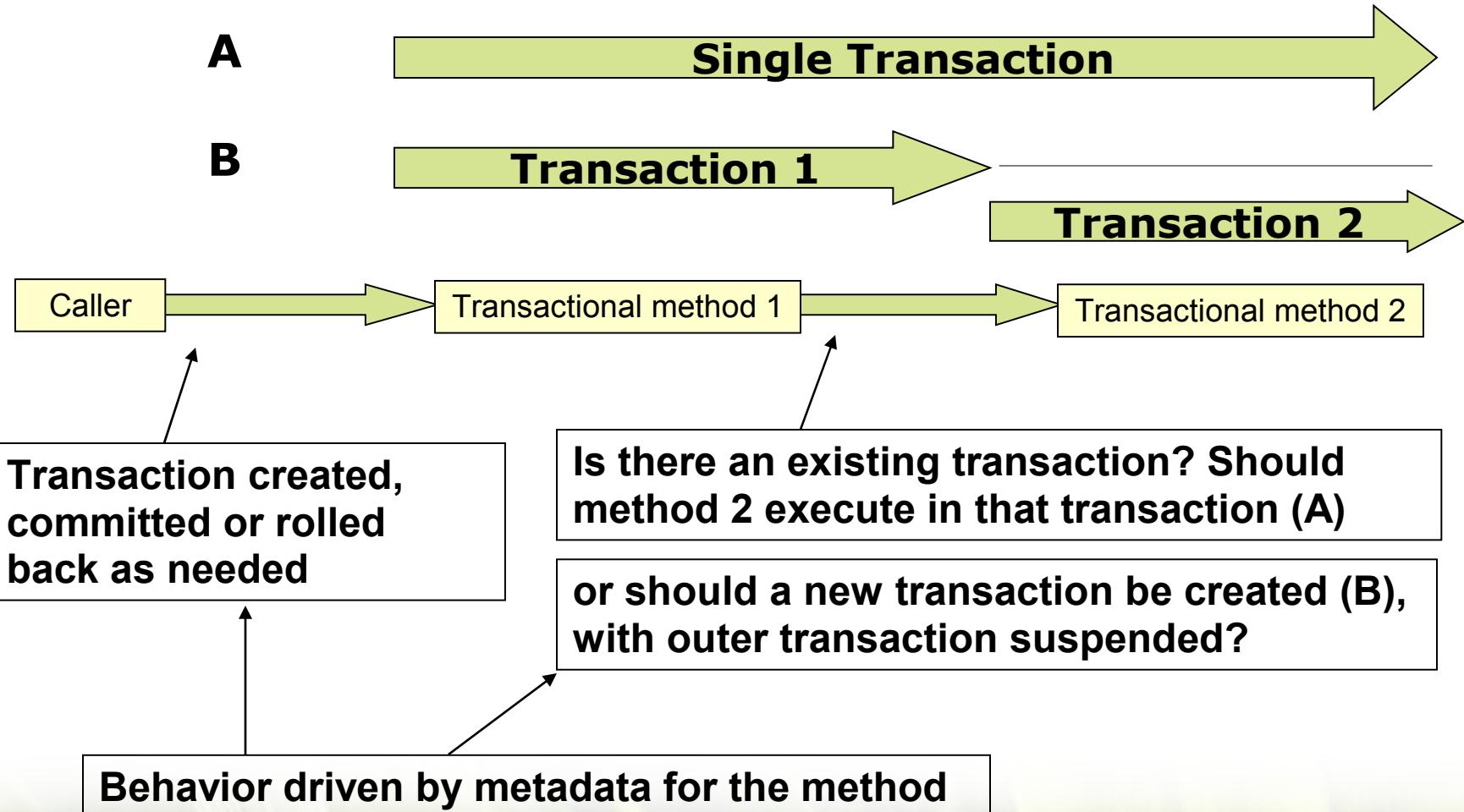
```
public class ClientServiceImpl
 implements ClientService {

 @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

---



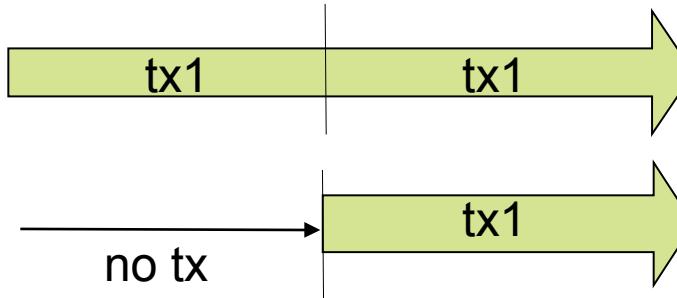
- 7 levels of propagation
- The following examples show *REQUIRED* and *REQUIRES\_NEW*
  - *Check the documentation for other levels*
- Can be used as follows:

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

# REQUIRED



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

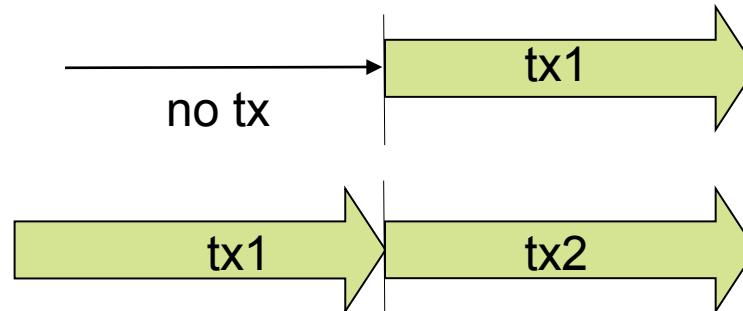


```
@Transactional(propagation=Propagation.REQUIRED)
```

# REQUIRES\_NEW



- REQUIRES\_NEW
  - Create a new transaction, suspending the current transaction if one exists



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

# Topics in this session

---

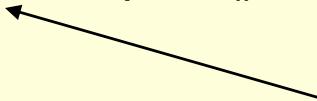


- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- Transaction Propagation
- **Rollback rules**
- Testing
- Advanced topics

# 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();
 }
}
```



Triggers a rollback

# rollbackFor and noRollbackFor



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

```
public class RewardNetworkImpl implements RewardNetwork {
```

not RuntimeException(s)

```
@Transactional(rollbackFor={RemoteException.class,MyOwnException.class})
public RewardConfirmation rewardAccountFor(Dining d) throws SQLException {
 // ...
}
```

```
@Transactional(noRollbackFor=RuntimeException.class)
public RewardConfirmation rewardAccountFor(Dining d) {
 // ...
}
```

Overrides default settings: there should not be a rollback in case of this specific Exception

# Topics in this session

---



- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- 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)
@TransactionalConfiguration(defaultRollback=false, transactionManager="txMgr")
@Transactional
public class RewardNetworkTest {
```

Transactions does a *commit* at the end by default

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

Overrides default *rollback* settings

 Inside *@TransactionalConfiguration*, 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() {...}
```

Run before transaction is started

```
 @Before
```

```
 public void setUpTestDataInTransaction() {...}
```

Within the transaction

```
 @Test @Transactional
```

```
 public void testRewardAccountFor() {
```

```
 ...
```

```
}
```

```
}
```



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



# LAB

Managing Transactions Declaratively with Spring  
Annotations

# Topics in this session

---



- Why use Transactions?
- Local Transaction Management
- Spring Transaction Management
- Transaction Propagation
- Rollback rules
- Testing
- **Advanced topics**
  - Programmatic transactions
  - Read-only transactions
  - Multiple transaction managers
  - Distributed transactions

# 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();
 }
 return confirmation;
 }
 });
}
```

# 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.queryForObject(...);
 jdbcTemplate.queryForInt(...);
}
```

2 connections

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

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

# Multiple Transaction Managers



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

```
@Transactional("myOtherTransactionManager")
public void rewardAccount1() {
```

Uses the bean id  
"myOtherTransactionManager"

```
 jdbcTemplate.queryForObject(...);
 jdbcTemplate.queryForInt(...);
}
```

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

Uses the bean id  
"transactionManager" 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
- 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>



# 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

---

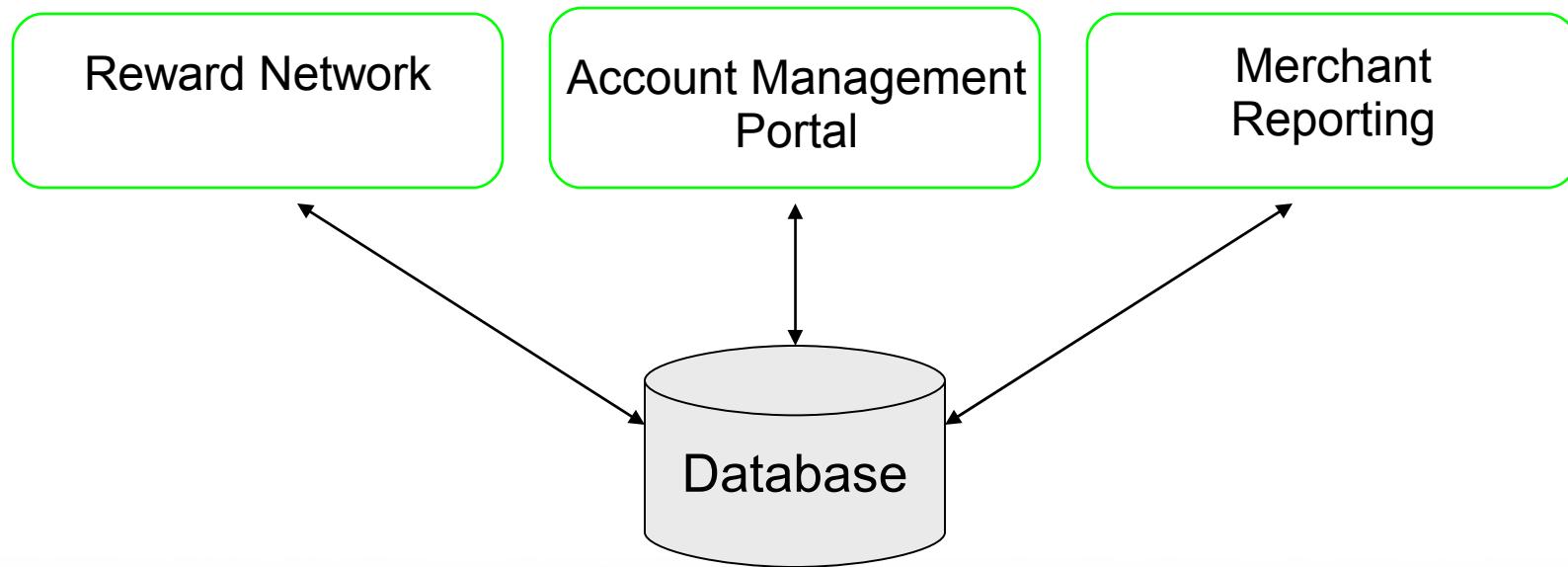


- The Object/Relational Mismatch
- **ORM in context**
- Benefits of modern-day ORM engines

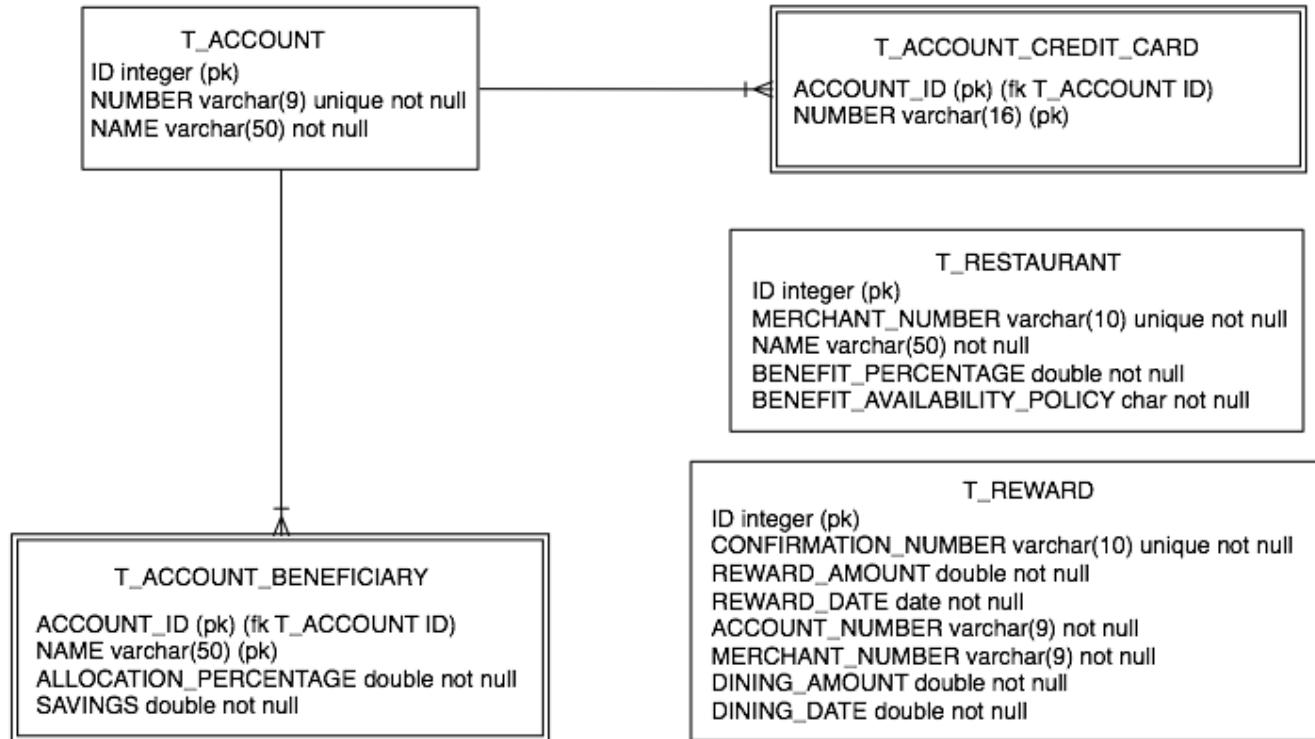
# ORM in context



- For the **Reward Dining** domain
  - The database schema already exists
  - Several applications share the data



# The integration database schema



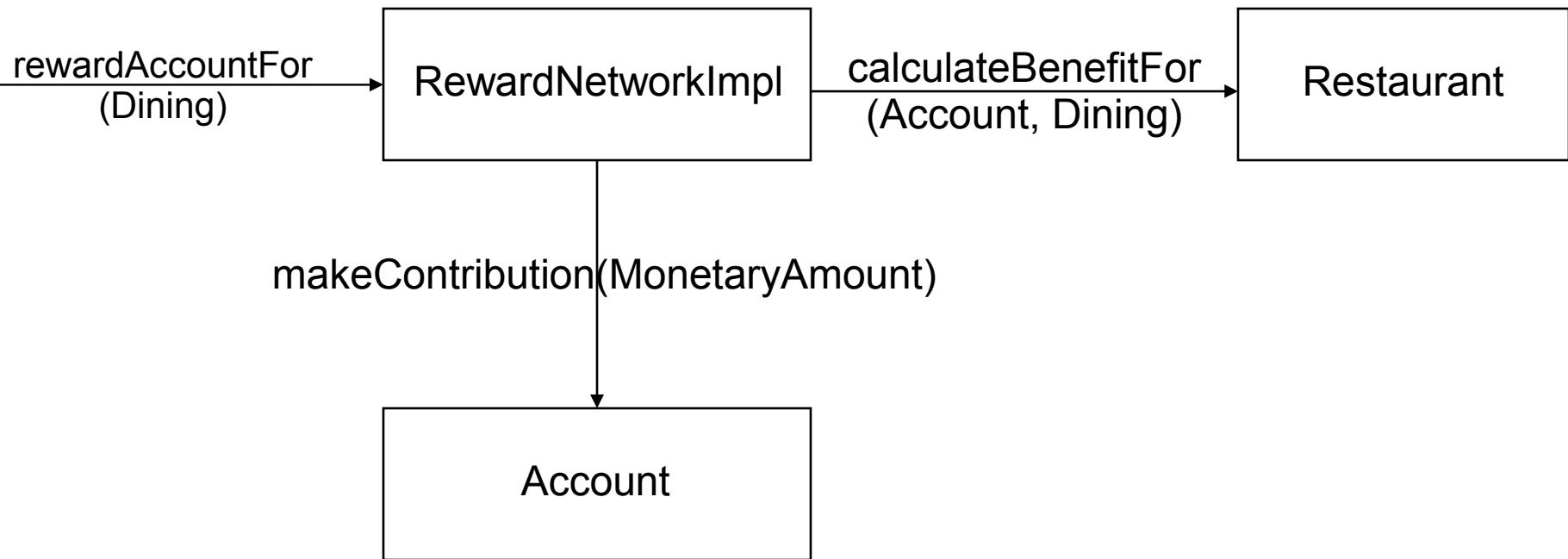
# The RewardNetwork application

---



- Rewards an account for dining
  - By asking the **Restaurant** where the dining occurred to calculate the benefit amount
  - Then, by making a contribution to the **Account** that initiated the dining

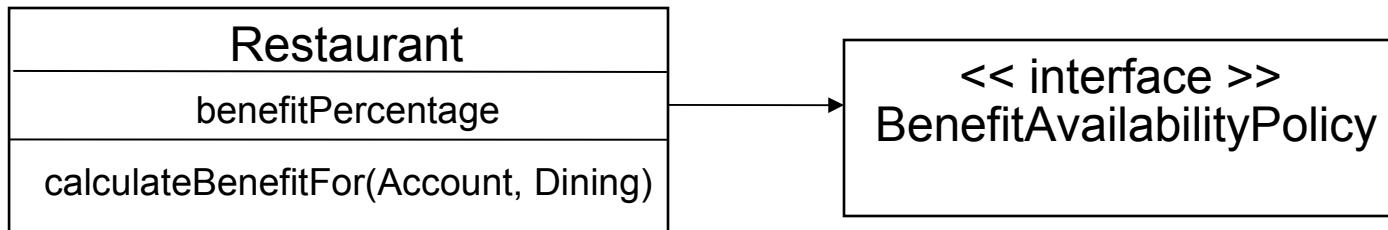
# The RewardNetwork application



# The Restaurant module



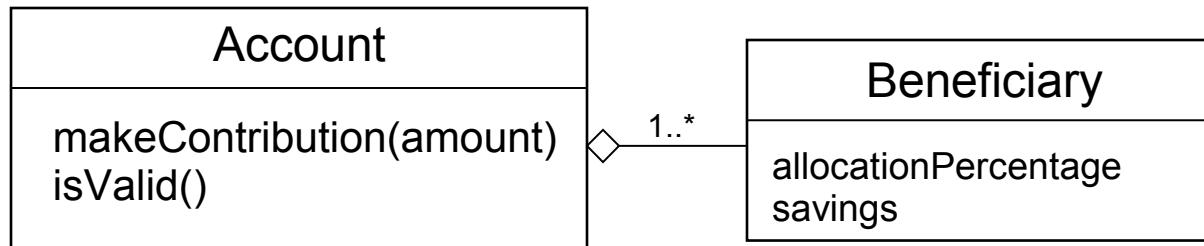
- Restaurant benefit calculations are based on a:
  - Benefit percentage
    - e.g. 4% of the dining bill
  - Benefit availability policy
    - e.g. some Restaurants only reward benefit on certain days



# The Account module



- Account contributions are distributed among the account's beneficiaries
  - Each beneficiary has an allocation percentage
  - The total beneficiary allocation must add up to 100%

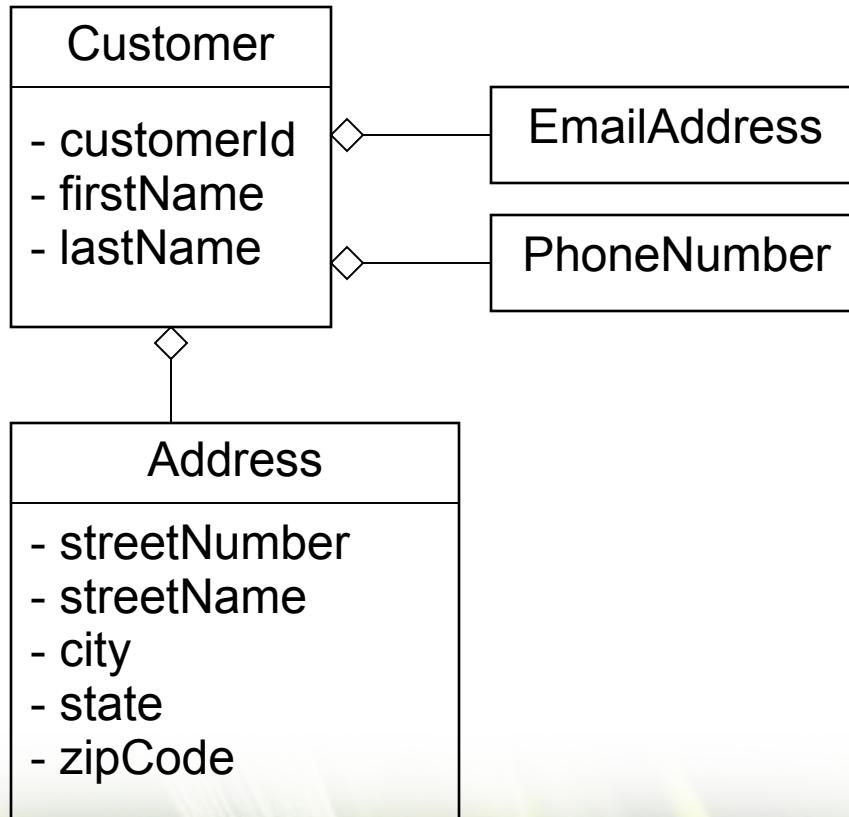


- All account and restaurant data is stored in the database
  - Data is needed by several applications
- What are the major challenges in mapping this object model to the relational model?

- 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

*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

- 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
- 2 Objects may be logically equivalent, but only one has a value set for its primary key field
- 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 (1<sup>st</sup> Level)
  - Per-DataSource (2<sup>nd</sup> 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
```

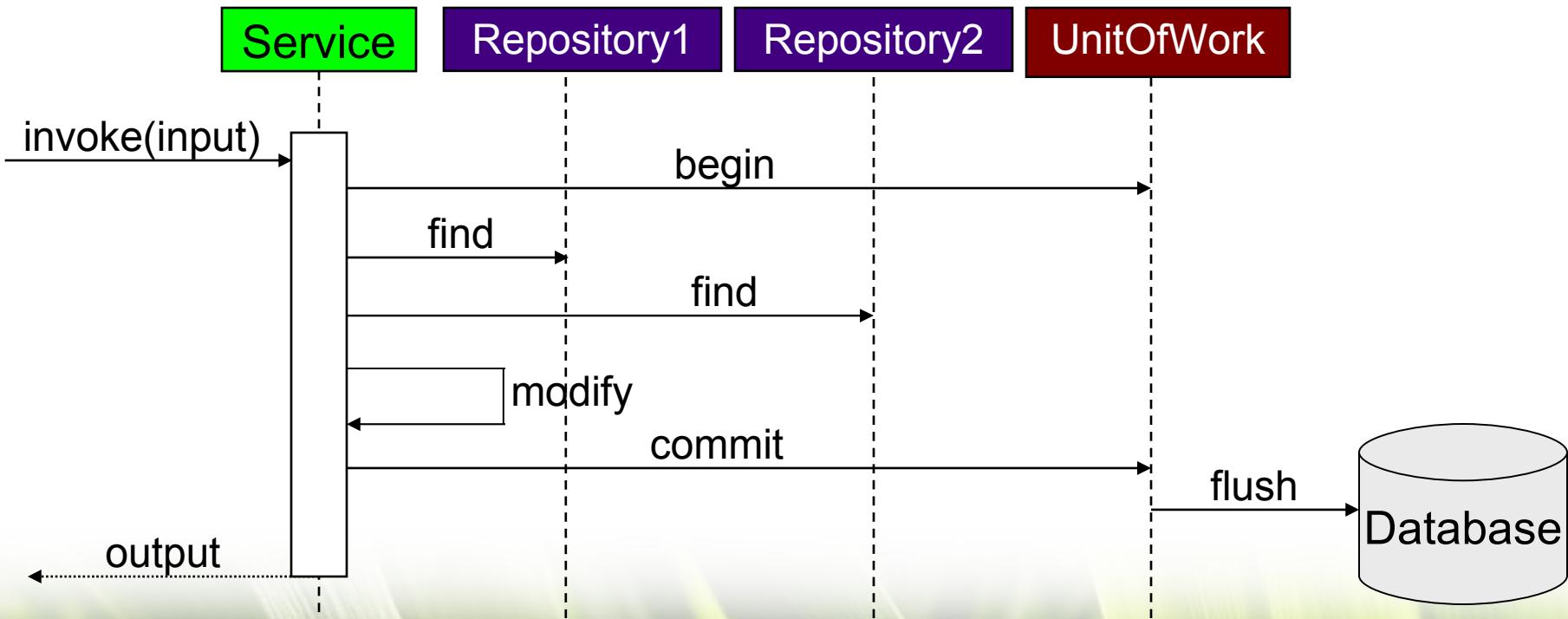
- Query with object properties and associations:

```
"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.



- 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
```

- The same concept applies for deletion:

```
Order order = orderRepository.findById(confirmationId);
// 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
}

if (order.isCancelled()) {
 orderRepository.remove(order);
 // all item rows for the order will be deleted
}
```

- 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



# Object/Relational Mapping with Spring and Hibernate

# Topics in this session

---



- **Introduction to Hibernate**
  - **Mapping**
  - **Querying**
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- Spring's HibernateTemplate

# Introduction to Hibernate



- Hibernate's **Session** is a stateful object representing a unit-of-work
  - Corresponds at a higher-level to a Connection
  - Manages persistent objects within the unit-of-work
  - Acts as a transaction-scoped cache (1LC)
- A **SessionFactory** is a thread-safe, shareable object that represents a single data source
  - Provides access to a transactional Session
  - Manages the globally-scoped cache (2LC)

# Hibernate Mapping



- Hibernate requires metadata for mapping classes and their properties to database tables and their columns
- Metadata can be provided as XML or annotations
- This session will present both approaches
  
- Carefully consider pros and cons of each approach, as well as hybrid approach

# Annotations support in Hibernate

---



- Hibernate supports two sets of annotations
  - javax.persistence.\* (standard, defined by JPA)
  - Native Hibernate annotations (extensions to JPA annotations)
- Best practice:
  - Use javax.persistence.\* annotations for typical features
  - Use Hibernate annotations for extensions
    - For behavior not supported by JPA
    - Performance enhancements specific to Hibernate
- Developers can still base their applications on the Hibernate API and HQL (it's not a JPA application)

# What can you annotate?

---

- Classes
  - Metadata which applies to the entire class (e.g. table)
- Either fields or properties
  - Metadata which applies to a single field or property (e.g. mapped column)
  - Defines the access mode (one default per class), most typically field, but not necessary
  - Based on default, all fields or properties will be treated as persistent (mappings will be defaulted)
  - Can be annotated with @Transient (non-persistent)
  - In Hibernate you can override the default access mode by using @org.hibernate.annotations.AccessType

# Mapping simple properties with annotations



```
@Entity
@Table(name= "T_CUSTOMER")
public class Customer { defines field as entity id, default access mode is 'field'
 @Id
 @Column (name="cust_id")
 private Long id;

 @Column (name="first_name")
 @org.hibernate.annotations.AccessType ("property")
 private String firstName;

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

Annotations and their descriptions:

- `@Entity`: defines field as entity id, default access mode is 'field'
- `@Table(name= "T_CUSTOMER")`: defines field as entity id, default access mode is 'field'
- `@Id`: defines field as entity id, default access mode is 'field'
- `@Column (name="cust_id")`: defines field as entity id, default access mode is 'field'
- `@Column (name="first_name")`: 'property' access uses setter
- `@org.hibernate.annotations.AccessType ("property")`: 'property' access uses setter
- `private String firstName;`: 'property' access uses setter
- `public void setFirstName(String firstName) {`: 'property' access uses setter
- `this.firstName = firstName;`: 'property' access uses setter
- `}`: 'property' access uses setter
- `...`: 'property' access uses setter

# Mapping simple properties with XML



```
public class Customer {
 private Long id;
 private String firstName;
 public void setFirstName(String firstName) {
 this.firstName = firstName;
 }
 ...
}
```

‘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>
```

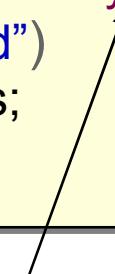
‘field’ access

# 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

# 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"/>
 <one-to-many class="Address"/>
 </set>
</class>
```

# 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
- This will first check the transactional cache

```
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

```
String city = "Chicago";
Query query = session.createQuery(
 "from Customer c where c.address.city = :city");
query.setString("city", city);
List customers = query.list();

// or if expecting a single result
Customer customer = (Customer) query.uniqueResult();
```

# Topics in this session

---



- Introduction to Hibernate
  - Mapping
  - Querying
- **Configuring a Hibernate SessionFactory**
- Spring's HibernateTemplate
- Implementing Native Hibernate DAOs

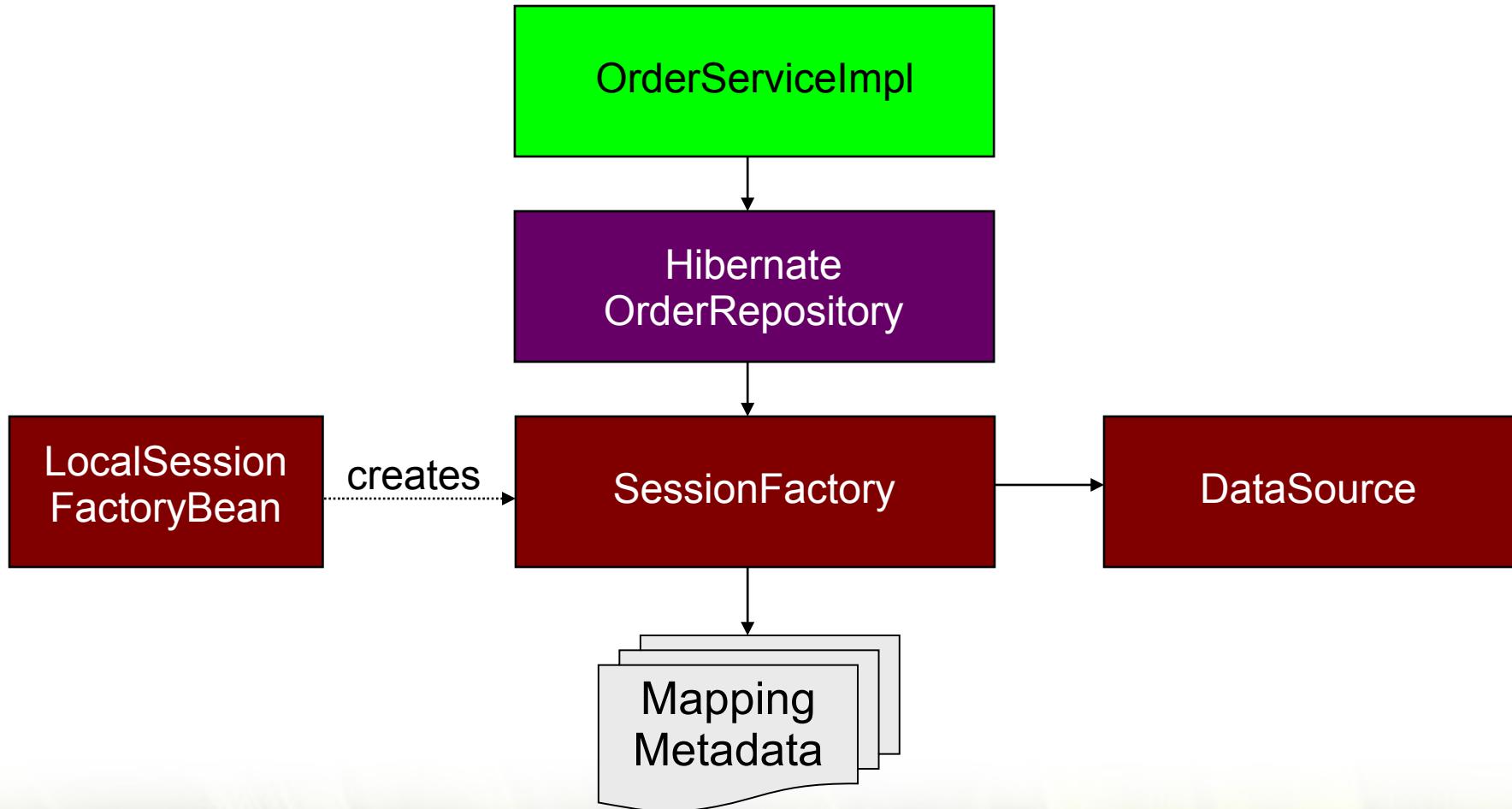
# Configuring a SessionFactory (1)

---



- 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

# Configuring a SessionFactory (2)



# SessionFactory and annotated classes



- annotatedClasses attribute
  - Entity classes should be listed one by one

```
<bean id="orderRepository"
 class="example.HibernateOrderRepository">
 <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="sessionFactory"
 class="o.s.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
 <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"/>
```

Wildcards are not supported here

# SessionFactory and scanned packages



- `packagesToScan` attribute
  - Wildcards are supported
  - Also scans all subpackages

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
 <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"/>
```

# Configuring a SessionFactory with XML metadata



```
<bean id="orderService" class="example.OrderServiceImpl">
 <constructor-arg ref="orderRepository"/>
</bean>
<bean id="orderRepository"
 class="example.HibernateOrderRepository">
 <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="sessionFactory"
 class="org.springframework.orm.hibernate3.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>
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/orders"/>
```

Wildcards are supported

# Topics in this session

---



- Introduction to Hibernate
  - Mapping
  - Querying
- Configuring a Hibernate SessionFactory
- **Implementing Native Hibernate DAOs**
- Spring's HibernateTemplate

# Implementing Native Hibernate DAOs (1)

---



- For 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
- It is now possible to remove any dependency on Spring in your DAO implementations

# Spring-managed Transactions and Sessions (1)

---



- To transparently participate in Spring-driven transactions, simply use one of Spring's FactoryBeans for building the SessionFactory
- Provide it to the data access code via dependency injection
  - Avoiding static access is a good thing anyway
- 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 void setSessionFactory(SessionFactory sessionFactory) {
 this.sessionFactory = sessionFactory;
 }
 ...
 public Order findById(long orderId) {
 // use the Spring-managed Session
 Session session = this.sessionFactory.getCurrentSession();
 return (Order) session.get(Order.class, orderId);
 }
}
```

# Spring-managed Transactions (3)



- The configuration

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

could use JtaTransactionManager if needed

# 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:  
e.g. Oracle will only accept SELECT statements

```
@Transactional(readOnly=true)
public List<RewardConfirmation> listRewardsFrom(Date d) {
 // read-only, atomic unit-of-work
}
```

# Transparent Exception Translation (1)

---



- Used as-is, the previous DAO implementation will throw native, vendor-specific **HibernateExceptions**
  - Not desirable to let these propagate up to the service layer or other users of the DAOs
    - Introduces dependency on specific persistence solution where it should not exist
- AOP allows translation to Spring's rich, vendor-neutral **DataAccessException** hierarchy

# Transparent Exception Translation (2)

---



Spring provides this capability out of the box

- Java 5+
  - Annotate with **@Repository** or your own custom annotation
  - Define a Spring-provided BeanPostProcessor
- Java 1.4+
  - Use pointcut matching your own (base or marker) repository interface to apply Spring-provided **PersistenceExceptionTranslationInterceptor**

# Transparent Exception Translation (3): Java 5+ Environment



- the code

```
@Repository
public class HibernateOrderRepository implements OrderRepository {
 ...
}
```

- the configuration

```
<bean class="org.springframework.dao.annotation.
PersistenceExceptionTranslationPostProcessor"/>
```

# Transparent Exception Translation (4): Java 1.4+ Environment



- the code

```
public class HibernateOrderRepository implements MyRepository {
 ...
}
```

- the configuration

```
<bean id="persistenceExceptionInterceptor"
 class="org.springframework.dao.support.
 PersistenceExceptionTranslationInterceptor"/>

<aop:config>
 <aop:advisor pointcut="execution(* *..MyRepository+.*(..))"
 advice-ref="persistenceExceptionInterceptor" />
</aop:config>
```

# Topics in this session

---



- Introduction to Hibernate
  - Mapping
  - Querying
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- **Spring's HibernateTemplate**

# HibernateTemplate



- Consistent with Spring's general DAO support
  - Manages resources (acquiring/releasing Sessions)
  - Translates Exceptions to the DataAccessException hierarchy
  - Participates in Spring-managed transactions automatically
  - Provides convenience methods
- Most prefer native approach, instead of HibernateTemplate usage, for Hibernate 3.1+

# Configuring HibernateTemplate

---



- Simply provide a reference to the SessionFactory

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



# LAB

## Using Hibernate with Spring



# Overview of Spring Web

Developing modern web applications  
with Spring

# Topics in this Session

---

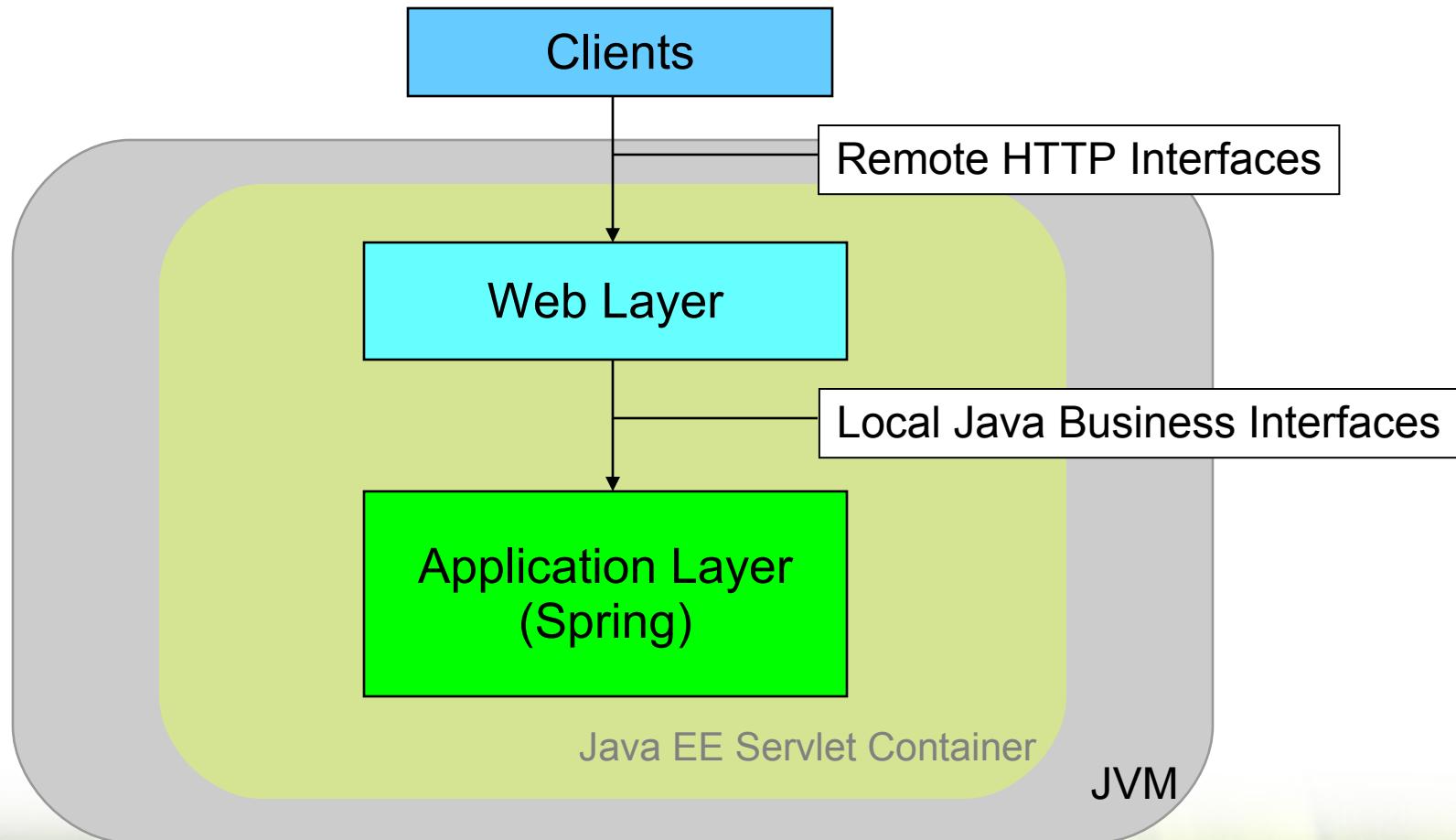
- **Introduction**
- Using Spring in Web Applications
- Overview of Spring Web
- Spring and other Web frameworks

# 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

# 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

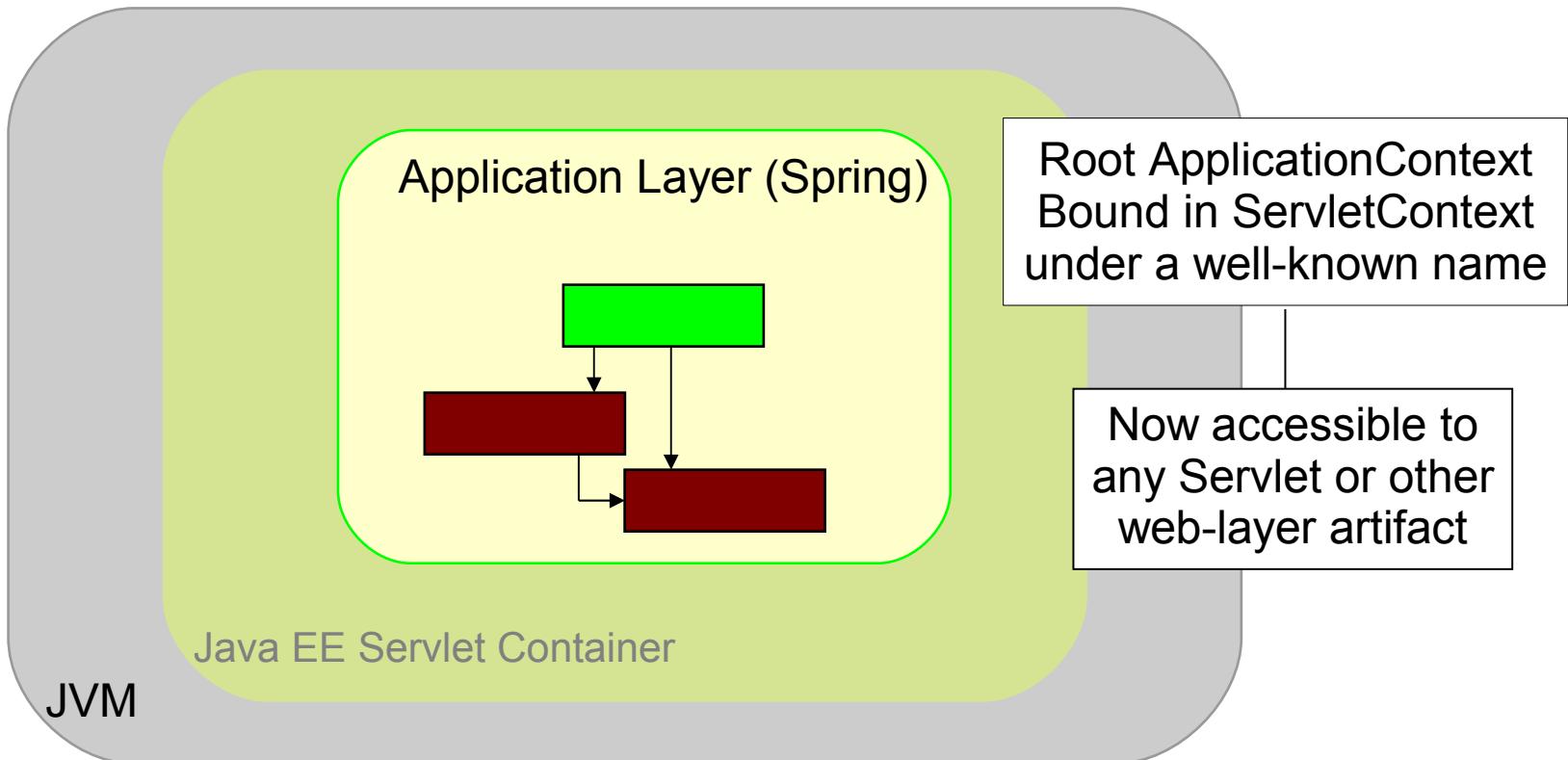
```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
 /WEB-INF/merchant-reporting-webapp-config.xml
 </param-value>
</context-param>
```

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

# 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 MerchantReportingService reportingService;

 @Autowired
 public TopSpendersReportController(MerchantReportingService service) {
 this.reportingService = 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

- 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 *conversation* scope
  - Addresses the double-submit problem elegantly

# Example Flow Definition

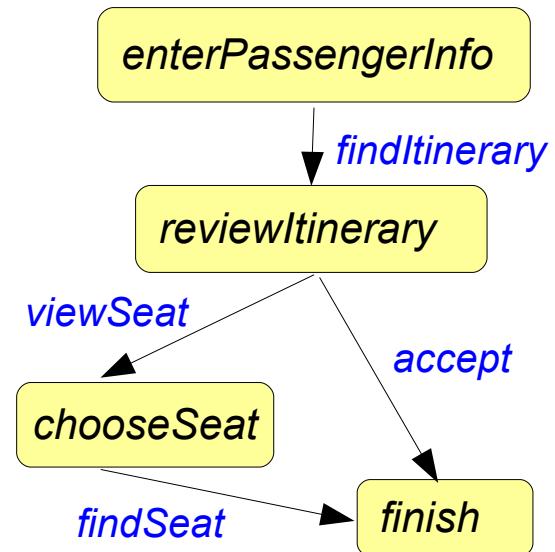
- Flows are declared in Xml

```
<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>



- WebFlow also comes with a set of sub-projects
  - Spring Faces
    - Allows to use WebFlow with JSF on the view layer
    - Provides enhancements to standard JSF facilities
    - scoped state management
  - Spring Javascript
    - Graceful degradation for clients without Javascript
    - Partial page rendering
    - Client-side validation
    - A ResourceServlet for serving & caching static web resource



There will be some important changes on Spring JS and Spring Faces in WebFlow 3.x. Stay tuned!

# Topics in this Session

---

- Introduction
- Using Spring in Web Applications
- Overview of Spring Web
- **Spring and other Web frameworks**

# 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

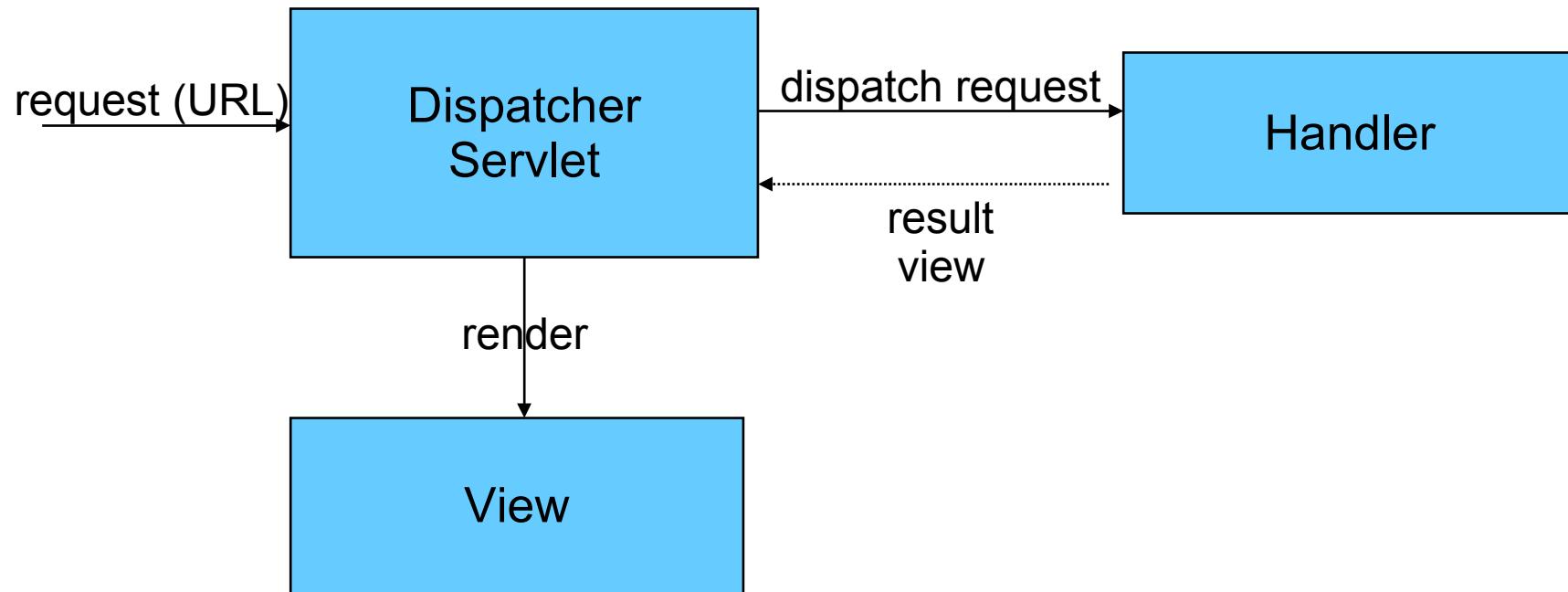
# Web Request Handling Overview

---



- 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



# 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

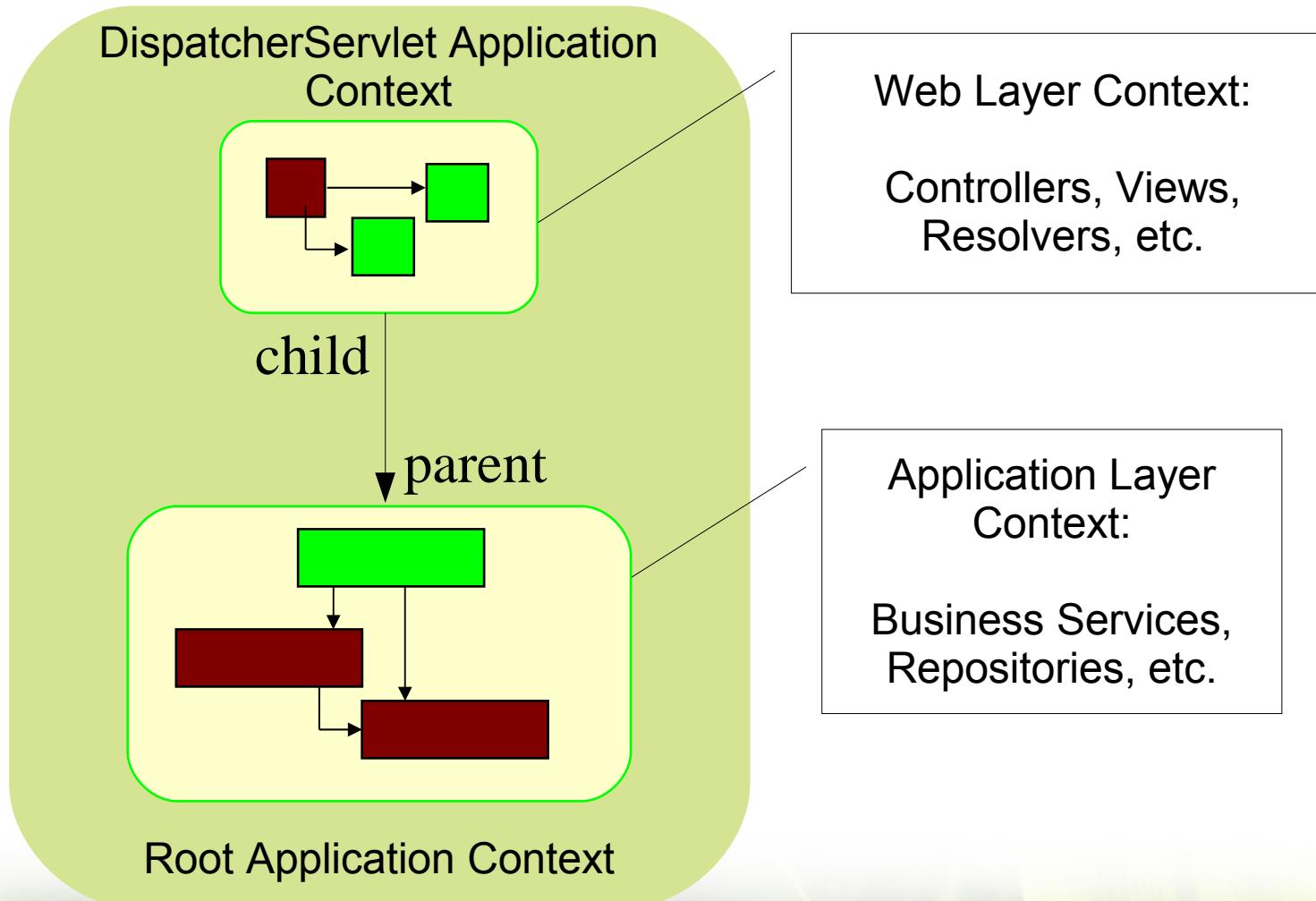
# DispatcherServlet Configuration

---



- 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

# Servlet Container After Starting Up



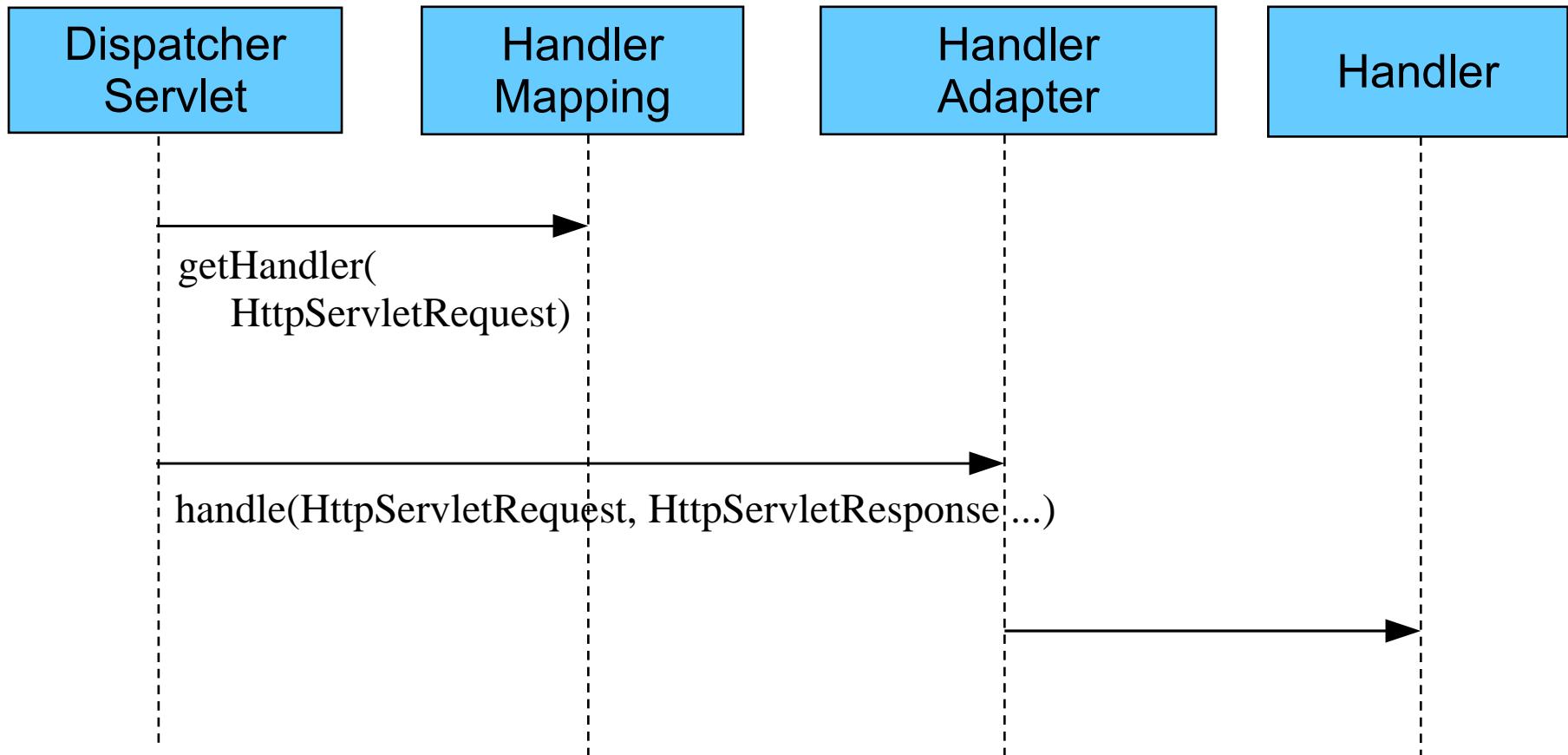
# Topics in this Session

---



- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - **Handlers**
  - Views
- Quick Start

# Request Processing



- Spring 2.5 introduces a simplified, annotation-based model for developing Spring MVC applications
  - Informally referred to as “Spring @MVC”
- Earlier versions rely more heavily on XML for configuration
  - “Spring <MVC/>”
  - Controller class hierarchy deprecated in Spring 3.0
- This training will focus on Spring @MVC

# Controllers as Request Handlers



- Handlers you define 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 ExampleController {

 @RequestMapping("/listAccounts.htm")
 public String list(Model model) {
 ...
 }
}
```

Literally, “execute this method to process requests for /listAccounts.htm”

# 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



- Handler methods will likely need context about the current request
- Spring MVC will 'fill in' declared parameters
  - allows for very flexible method signatures

```
@Controller
public class ExampleController {

 @RequestMapping("/listAccounts")
 public String list(Model model) {
 ...
 }
}
```

# Extracting Request Parameters



- Use `@RequestParam` annotation
  - Extracts parameter from the request
  - Performs type conversion

```
@Controller
public class ExampleController {

 @RequestMapping("/listAccounts")
 public String show(@RequestParam("entityId") long id) {
 ...
 }
}
```

See JavaDoc of `@RequestMapping`  
for all possible argument types

# URI Templates

- Values can also be extracted from request URLs since Spring 3.0
- 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 ExampleController {

 @RequestMapping("/accounts/{accountId}")
 public String show(@PathVariable("accountId") long id) {
 ...
 }
}
```

# Example Controllers

---

- An AccountController
  - creates, shows, updates, and deletes Accounts
- A LoginController
  - logs users in
- A TopPerformingAccountsController
  - generates an Excel spreadsheet with top 20 accounts

# 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 view name is interpreted as path to JSP page
- Controllers may return **null** (or **void**)
  - DispatcherServlet then selects a default view based on the request URL
- Controllers may also return a concrete View
  - `new JstlView("/WEB-INF/reward/list.jsp")`
  - `new RewardListingPdf()`

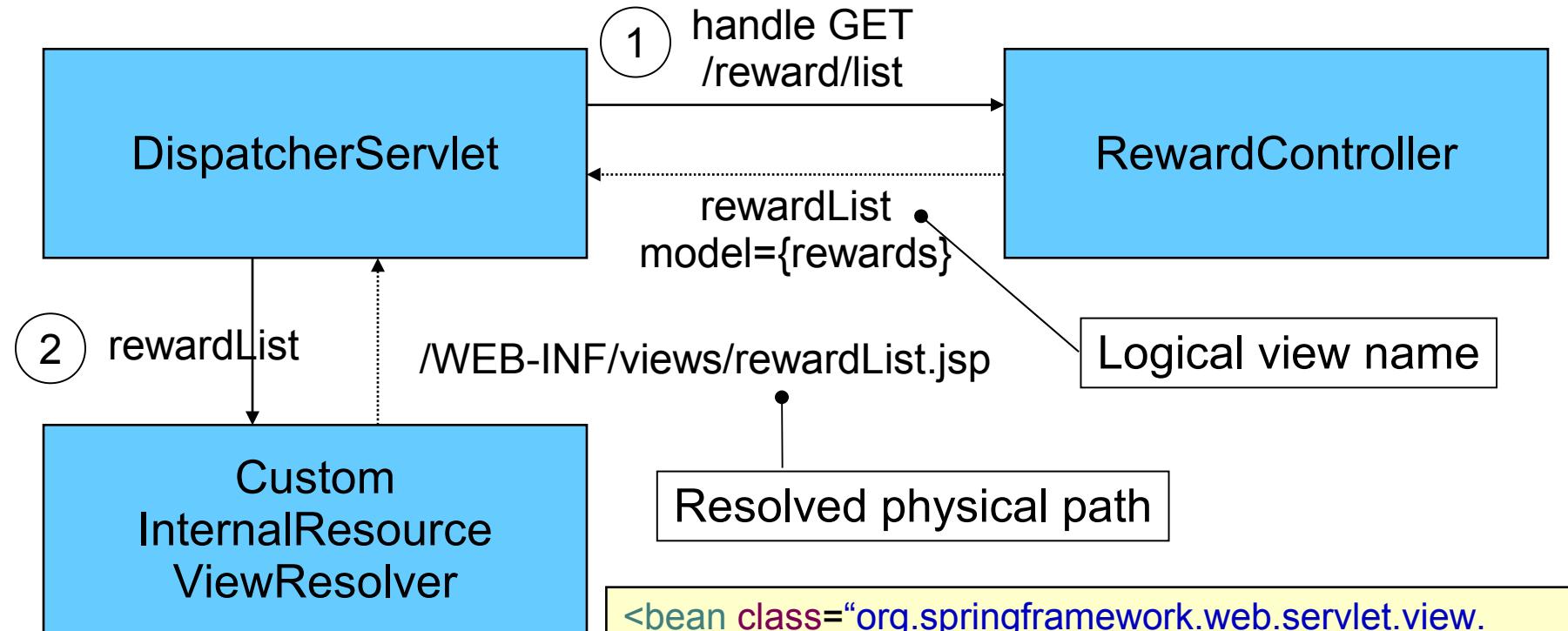
Not very common!

# 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



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

# 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



- Define inside <webapp> within web.xml

```
<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/rewardsadmin-servlet-config.xml
 </param-value>
 </init-param>
</servlet>
```

↑

Contains the Servlet's configuration

# 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/rewardsadmin/reward/list
http://localhost:8080/rewardsadmin/reward/new
http://localhost:8080/rewardsadmin/reward/show?id=1
```

# Initial DispatcherServlet Configuration

---



/WEB-INF/rewardsadmin-servlet-config.xml

```
<beans>

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

</beans>
```

## 2. Implement the Controller



```
@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) {
 Reward reward = lookupService.lookupReward(id);
 model.addAttribute(reward);
 return "rewardView";
 }
}
```

Depends on application service

Selects the “rewardView” to render the reward

Automatically filled in by Spring

### 3. Implement the View

/WEB-INF/views/rewardView.jsp

```
<html>
 <head><title>Your Reward</title></head>
 <body>
 Amount=${reward.amount}
 Date=${reward.date}
 Account Number=${reward.account}
 Merchant Number=${reward.merchant}
 </body>
</html>
```

↑  
References result model object by name

# 4. Register the Controller

---

/WEB-INF/rewardsadmin-servlet-config.xml

```
<bean id="rewardController" class="rewardsadmin.RewardController">
 <constructor-arg ref="rewardLookupService" />
</bean>
```

## 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

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

```
@Controller
public class RewardController {
 private RewardLookupService lookupService;

 @Autowired
 public RewardController(RewardLookupService svc) {
 this.lookupService = svc;
 }

 @RequestMapping("/reward/show")
 public String show(long id, Model model) { ... }
}
```

# MVC Additions in Spring 3.0

---



- @MVC model is enabled by default
  - Appropriate HandlerMapping and ~Adapter registered without any additional configuration
- However, Spring 3.0 introduces new features *not* enabled by default
  - Stateless converter framework for binding and formatting
  - Support for JSR-303 declarative validation
  - HttpMessageConverters (for RESTful web services)
- How do you use these features?

# MVC Namespace

- Use new MVC namespace to easily 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
 HttpMessageConverters -->
 <mvc:annotation-driven/>

```

- Registers HandlerMapping and ~Adapter for @MVC
  - Means you lose the other default mappings and adapters!
- Allows custom conversion service and validator
  - and defining HandlerInterceptors applicable to all HandlerMappings or specific URI paths
- Goes beyond the scope of this course



# LAB

Spring Web MVC Essentials



# 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
- REST is an architectural style that describes best practices to expose web services over HTTP
  - REpresentational S<sub>tate</sub> T<sub>ransfer</sub>, term by Roy Fielding
  - HTTP as *application* protocol, not just transport
  - Emphasizes scalability

# REST Principles (1)

---

- Expose *resources* through URIs
  - Model nouns, not verbs
- Resources support limited set of operations
  - GET, PUT, POST, DELETE in case of HTTP
  - All have well-defined semantics
- to update an existing 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
  - Looser coupling between client and server
  - Makes clients keep track of state
  - MIME types contract for interacting with a resource

# REST Principles (3)

---

- Stateless architecture
  - No HttpSession usage
  - GETs can be cached on URL
  - Part of what makes it scalable
- HTTP headers and status codes communicate result to clients
  - All well-defined in HTTP Specification

# REST Clients

---

One benefit of REST: every platform/language supports HTTP

- Unlike for example SOAP + WS-\* specs
- Easy to support many different clients

# 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

- 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
  - Content Negotiation
  - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

- Some features have been covered already
  - URI templates to parse 'RESTful' URLs
  - Views for multiple resource representations
  - Apply to all web clients, incl. browsers
- This module covers additional features for building web services for programmatic clients
  - Support for RESTful web applications targeting browser-based clients is also offered
  - but is outside of this course's scope

# 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

# 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
}
```

# HTTP Status Code Support

---



- Web apps just use a handful of status codes
  - 200 OK, 404 Not Found, 302/303 for redirects and 500 Internal Server Error for unhandled Exceptions
- RESTful applications use many additional codes to communicate with their clients
- Use `@ResponseStatus` on controller method so you don't have to set this on `HttpServletResponse` yourself

# Common Response Codes

---



- Some common HTTP response codes:
  - 200: after a successful GET where content is returned
  - 201: when new resource was created on POST or PUT
    - Location header should contain URI of new resource
  - 204: when the response is empty
    - e.g. after successful update with PUT or DELETE
  - 404: when requested resource was not found
  - 405: when HTTP method is not supported by resource
  - 409: when a conflict occurs while making changes
    - e.g. when POSTing unique data that already exists
  - 500: internal server error

# @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()));
}
```

- When using `@ResponseStatus`, `void` methods no longer imply a default view name!
  - There will be no View at all
  - 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 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**
  - Content Negotiation
  - 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 3<sup>rd</sup> party libraries on the classpath

# Why Use HttpMessageConverters (Input)

---



- Web service applications often process request content differently than regular web applications
  - Not just binding request parameters
    - e.g. a PUT or POST containing XML or JSON document
  - Still preferable to not process HttpServletRequest directly in controller
  - Need something else to map request to method parameter

# Why Use HttpMessageConverters (Output)

---



- Writing to response often different from web application as well
  - Controller methods might not want to use separate View to render result
  - nor write representation to HttpServletResponse directly
  - Need something else to map return value to response

# @RequestBody

- Use converters for request data by annotating method parameter with `@RequestBody`

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.PUT)
@ResponseBody(HttpStatus.NO_CONTENT) // 204
public void updateOrder(@RequestBody Order updatedOrder,
 @PathVariable long id) {
 // process updated order data and return empty response
 orderManager.updateOrder(id, updatedOrder);
}
```

- 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

# @ResponseBody

---

- Use converters for response data by annotating method with `@ResponseBody`

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseBody(HttpStatus.OK) // 200
public Order getOrder(@PathVariable long id) {
 // Order class is annotated with JAXB2's @XmlRootElement
 Order order= orderRepository.findOrderById(id);
 // results in XML response containing marshalled order:
 return order;
}
```

- Converter handles rendering to response
  - No ViewResolver and View involved anymore!

# 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

# Content Negotiation

---

- HTTP clients can request a particular resource representation through media / MIME types
  - Using Accept header in HTTP Request
    - `Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`
- Best available representation will be returned
- Java object backing resource typically the same
  - Single model, only representation change

# Content Negotiation & @ResponseBody

---



- @ResponseBody method returns just the model
- How to choose correct HttpMessageConverter for requested representation?
  - Not controller's responsibility
  - No ViewResolver involved either



*Remember to use <mvc:annotation-driven/> or to register custom converters to have HttpMessageConverters defined at all!*

# Automatic Content Negotiation

---



- `HttpMessageConverter` selected automatically for `@ResponseBody`-annotated methods
  - Based on `Accept` header in request
  - 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 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": [ ... ], ... }  
}

# Topics in this Session

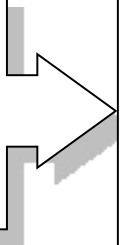
---

- 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

# 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 long id) {
 return orderRepository.findOrderById(id);
}
```

# Creating a new Resource: POST



```
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 orderId,
 @RequestBody Item newItem,
 HttpServletRequest request,
 HttpServletResponse response) {
 orderRepository.findOrderById(id).addItem(newItem); // adds id to Item
 response.addHeader("Location",
 getLocationForChildResource(request, newItem.getId()));
}
```

# 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(id).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(id).deleteItem(itemId);
}
```

# 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

# 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="jsonRestTemplate" 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";

// 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);
```



*Spring 3.0.2 introduces `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
  - Content Negotiation
  - 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



# 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

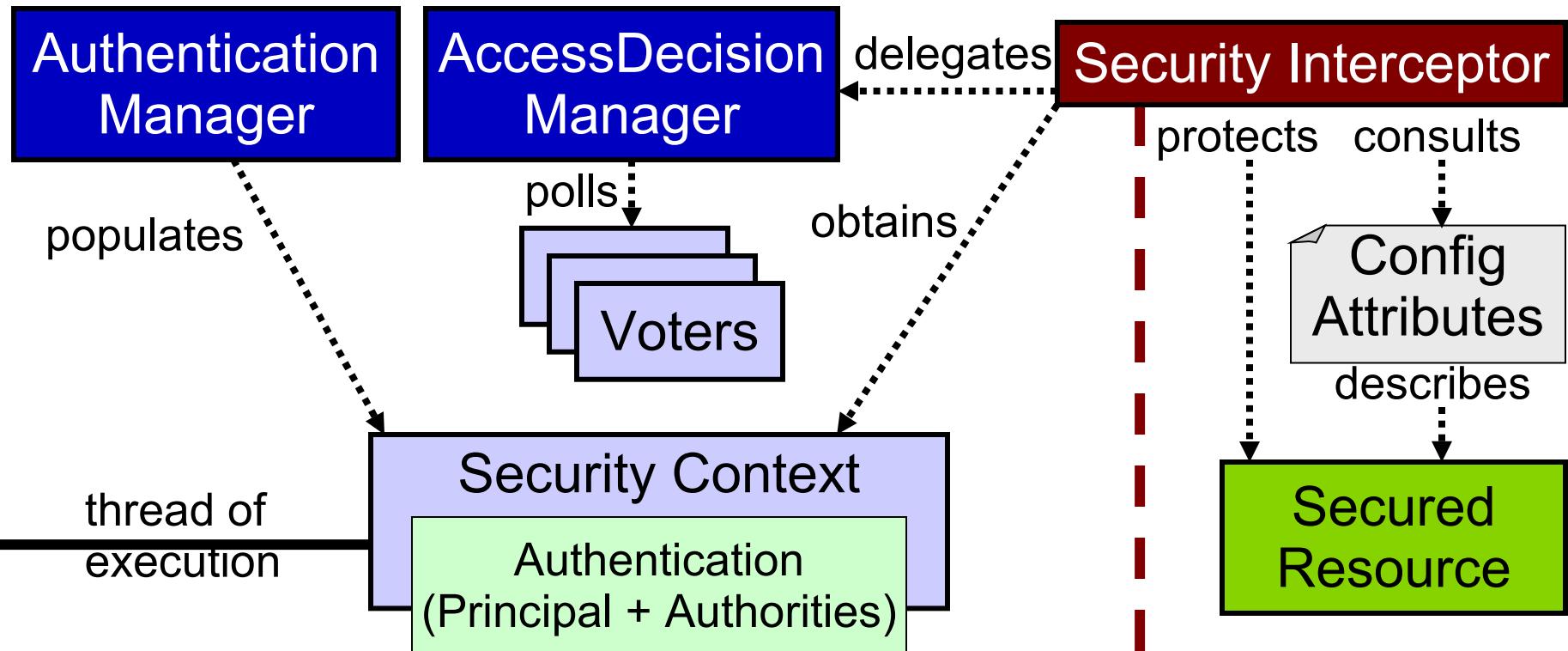
# Motivations: Consistency

---



- 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**
- 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 />

 </security:http>
</beans>
```

Match all URLs starting with /accounts/ (ANT-style path)

# Configuration in web.xml



- Define the single proxy filter
  - `springSecurityFilterChain` is a mandatory name
  - It refers to an existing Spring bean with the 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>
```

# intercept-url

---

- intercept-urls are evaluated in the order listed
  - The first match will be used
  - specific matches should be put on top

```
<security:intercept-url pattern="/accounts/login.htm" filters="none" />

<security:intercept-url pattern="/accounts/edit*"
 access="ROLE_USER, ROLE_ADMIN" />

<security:intercept-url pattern="/accounts/account*" access="ROLE_USER" />

<security:intercept-url pattern="/accounts/**"
 access="IS_AUTHENTICATED_FULLY" />
```

# 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

# An Example Login Page



URL that indicates an authentication request

```
<form action="
```

The expected keys for generation of an authentication request token

# Configure Authentication



- DAO Authentication provider is default
- Plug-in specific UserDetailsService implementation to provide credentials and authorities
  - Built-in: JDBC, in-memory
  - Custom

```
<security:authentication-manager>
 <security:authentication-provider>
 ...
 </security:authentication-provider>
<security:authentication-manager>
```

# The In-Memory user service (1/2)



- Useful for development and testing
  - Without encoding

```
<security:authentication-manager>
 <security:authentication-provider>
 <security:user-service properties="/WEB-INF/users.properties" />
 </security:authentication-provider>
<security:authentication-manager>
```

admin=secret,ROLE\_ADMIN  
testuser1=pass,ROLE\_MEMBER

- With encoding

```
<security:authentication-manager>
 <security:authentication-provider>
 <security:password-encoder hash="md5" />
 <security:user-service properties="/WEB-INF/users.properties" />
 </security:authentication-provider>
<security:authentication-manager>
```

# The In-Memory UserDetailsService (2/2)

---



- The properties file

```
admin=secret,ROLE_ADMIN,ROLE_MEMBER
testuser1=pass,ROLE_MEMBER
testuser2=pass,ROLE_MEMBER
guest=guest,ROLE_GUEST,ROLE_MEMBER
```

login

password

List of roles separated by commas

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 = ?

# The JDBC user service (2/2)



- Configuration:

```
<security:authentication-manager>
 <security:authentication-provider>
 <security:jdbc-user-service data-source-ref="myDatasource" />
 </security:authentication-provider>
<security:authentication-manager>
```



possibility to customize queries using  
attributes such as  
authorities-by-username-query

# 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 can be declared as follows

```
<%@ taglib prefix="security"
 uri="http://www.springframework.org/security/tags" %>
```

available since Spring Security 2.0



# Spring Security's Tag Library



- Display properties of the Authentication object

You are logged in as:

```
<security:authentication property="principal.username"/>
```

- Hide sections of output based on ROLE

```
<security:authorize ifAnyGranted="ROLE_ADMIN">
 TOP-SECRET INFORMATION
 Click HERE to delete all records.
</security:authorize>
```

This URL *should* be protected by  
the intercept-url tag also!

# 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

# Method security with XML config

---



- Allows to apply security to many beans with only a simple declaration

```
<security:global-method-security>
 <security:protect-pointcut
 expression="execution(* com.springsource..*Service.*(..))"
 access="ROLE_USER" />
</security:global-method-security>
```

# Method security with Spring annotations



- Spring Security annotations should be enabled

```
<security:global-method-security secured-annotations="enabled" />
```

- on the Java level:

```
import org.springframework.security.annotation.Secured;

public class ItemManager {
 @Secured("ROLE_MEMBER")
 public Item findItem(long itemNumber) {
 ...
 }
}
```

# Method security with JSR 250 annotations



- JSR-250 annotations should be enabled

```
<security:global-method-security jsr250-annotations="enabled" />
```

- on the Java level:

```
import javax.annotation.security.RolesAllowed;
```

```
public class ItemManager {
 @RolesAllowed("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**

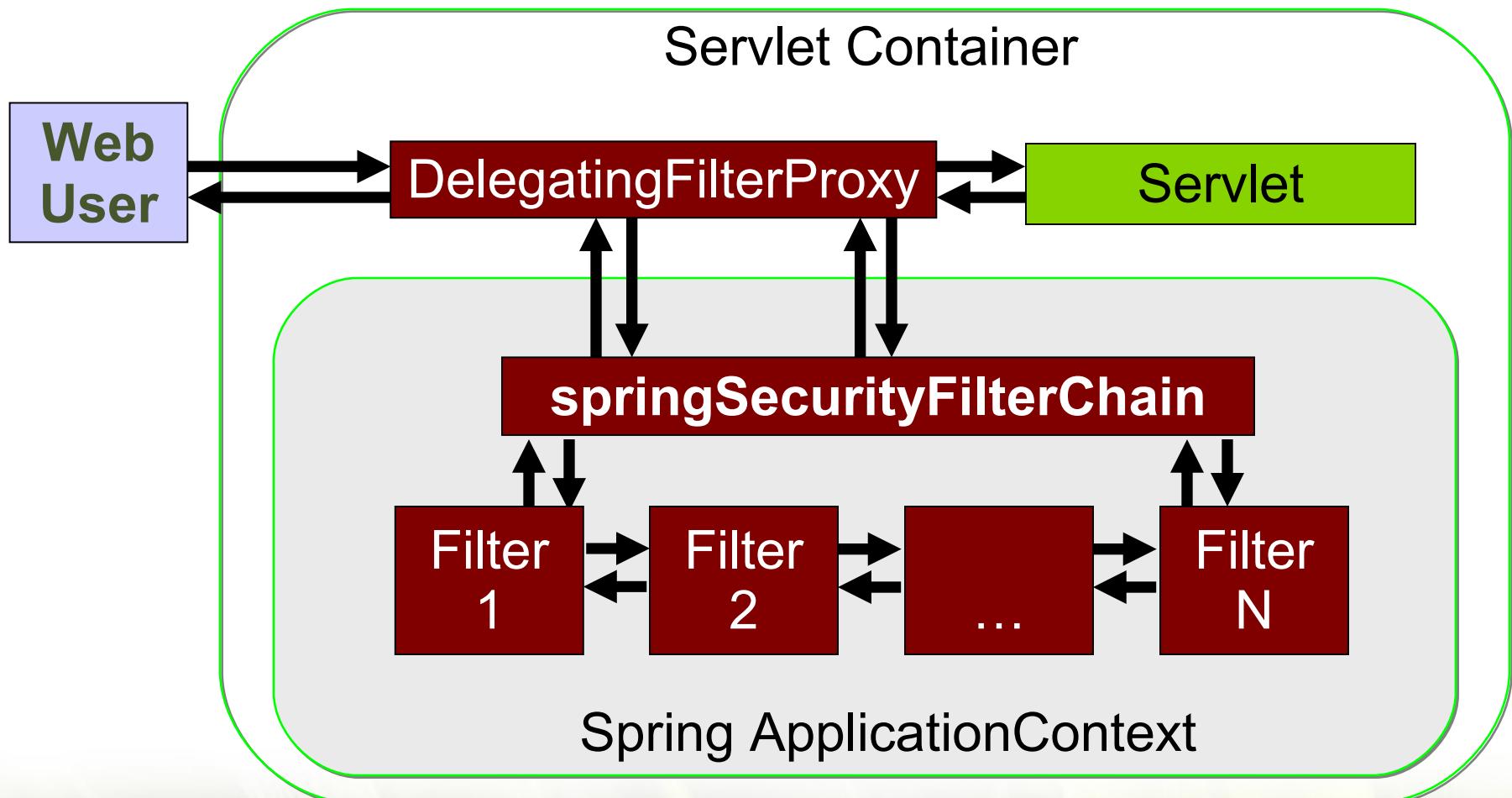
# Spring Security in a Web Environment

---



- `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

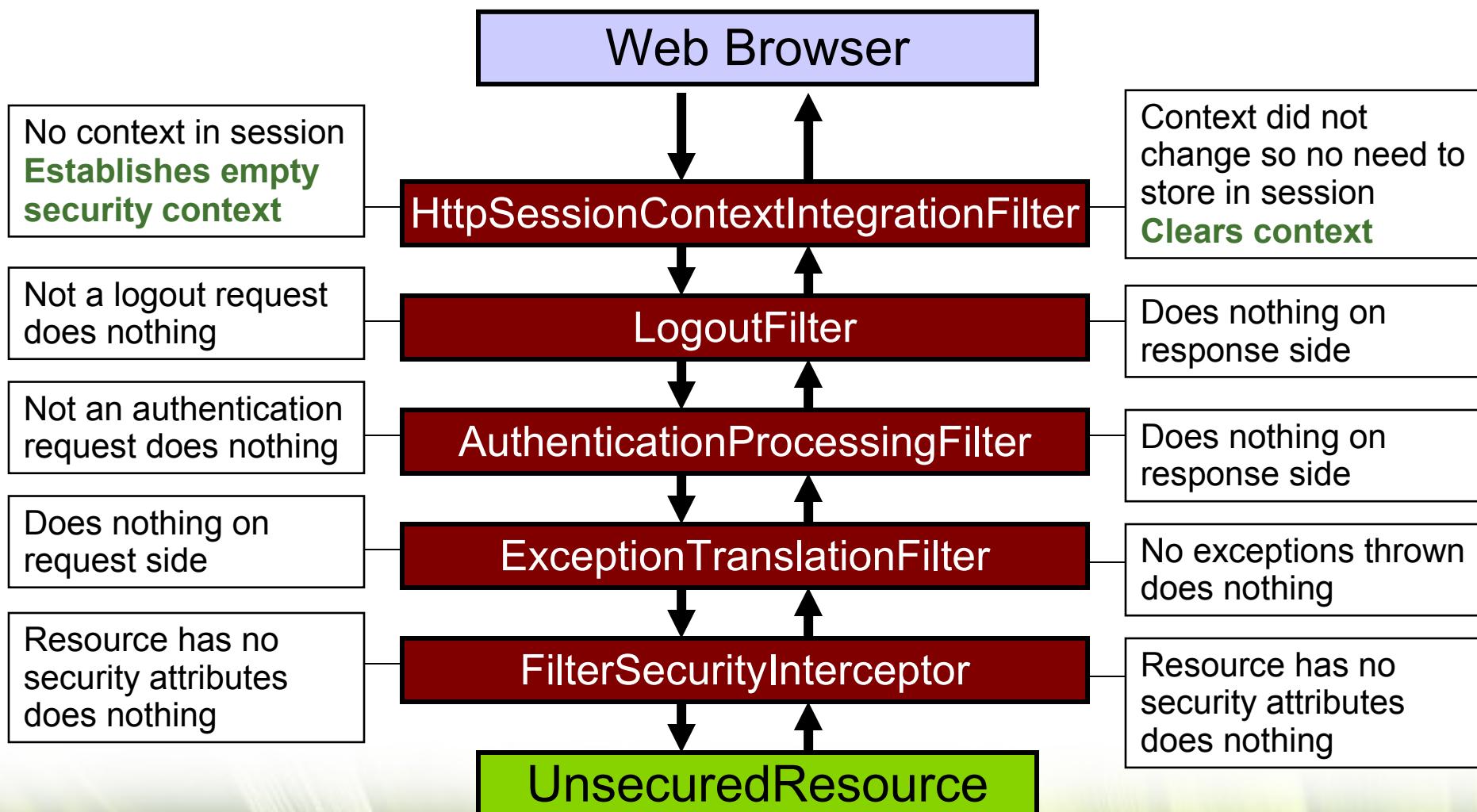


# The Filter chain

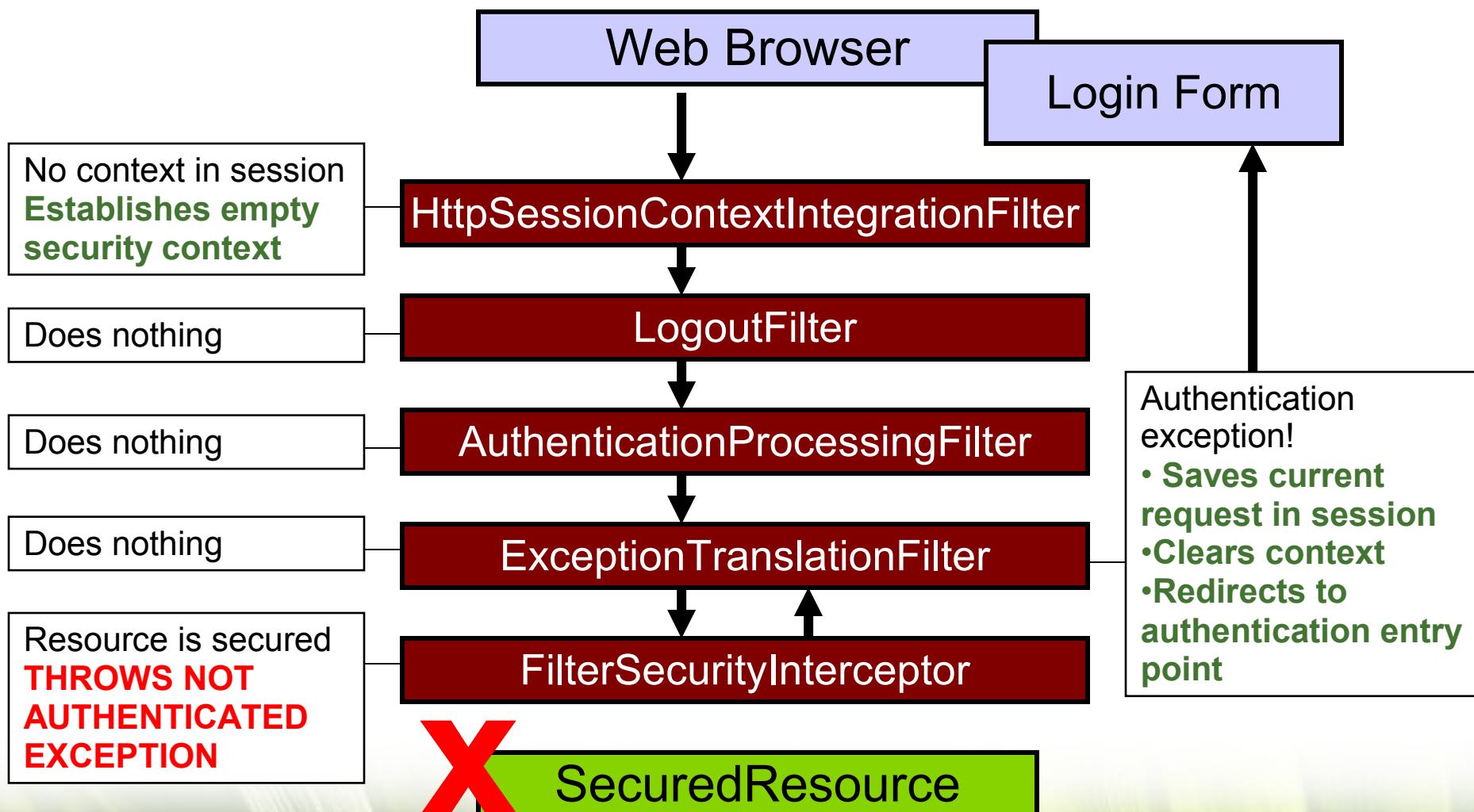


- With ACEGI Security 1.x
  - Filters were manually configured as individual <bean> elements
  - Led to verbose and error-prone XML
- Since Spring Security 2.0
  - Filters are initialized with correct values by default
  - Manual configuration is not required **unless you want to customize Spring Security's behavior**
  - It is still important to understand how they work underneath

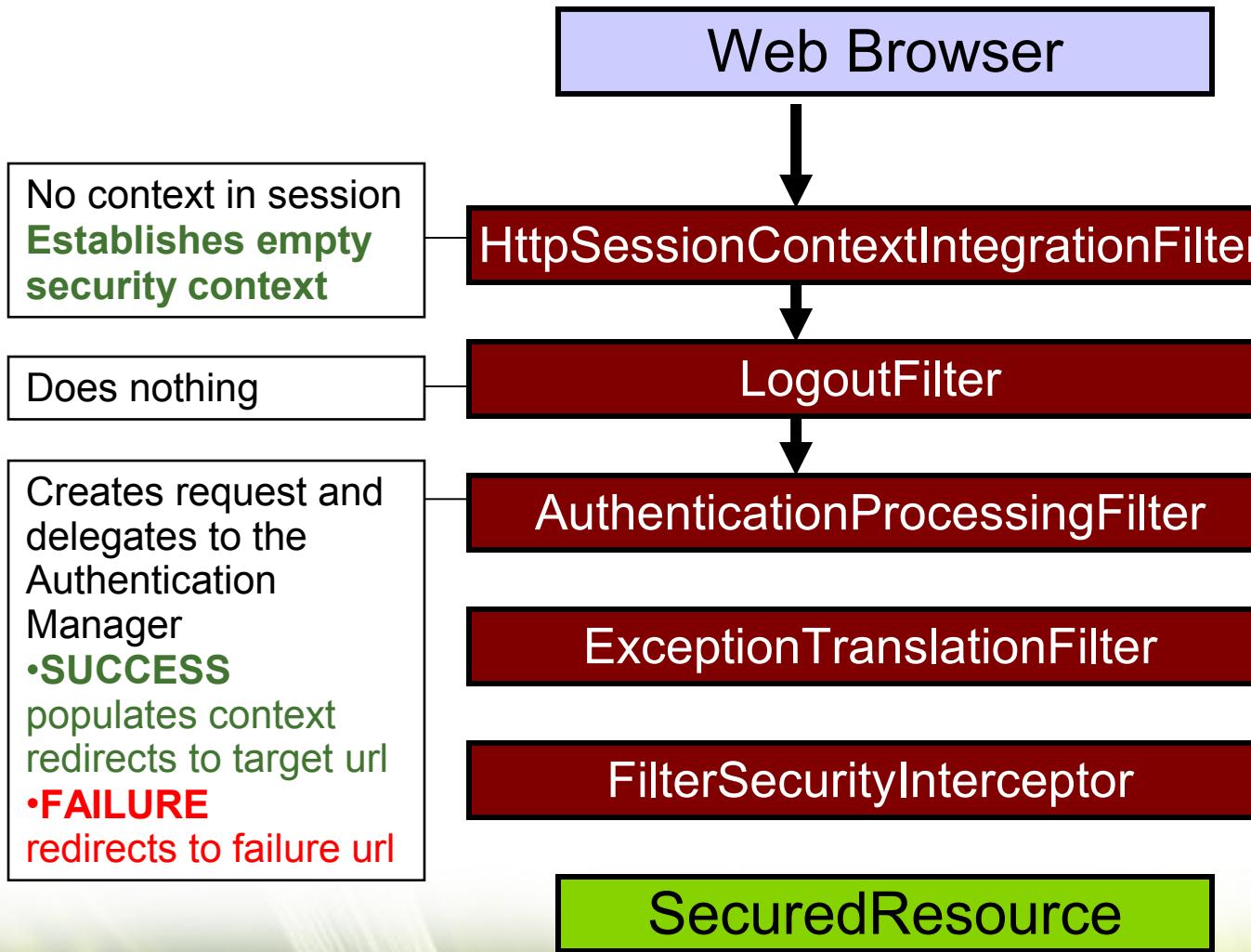
# Access Unsecured Resource Prior to Login



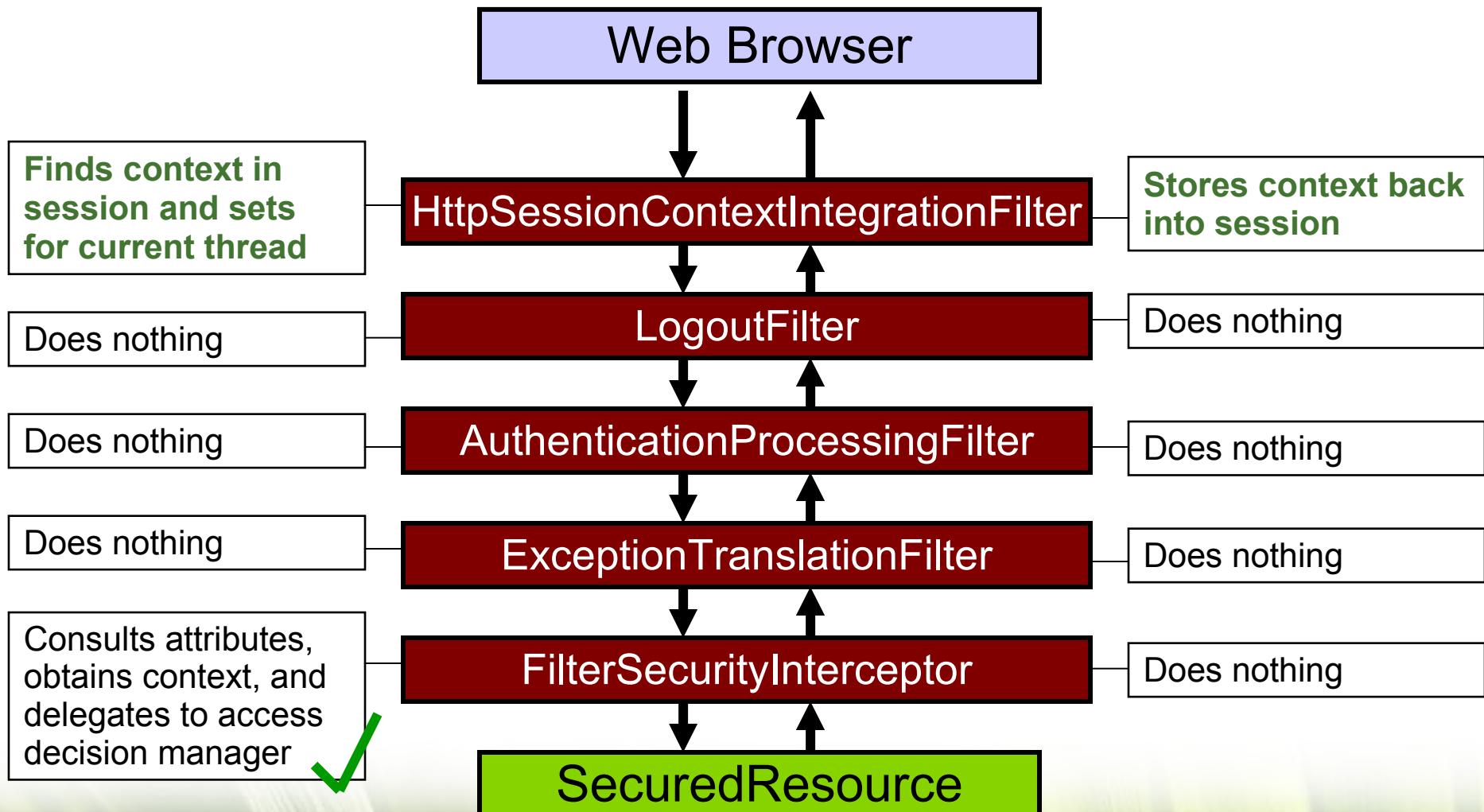
# Access Secured Resource Prior to Login



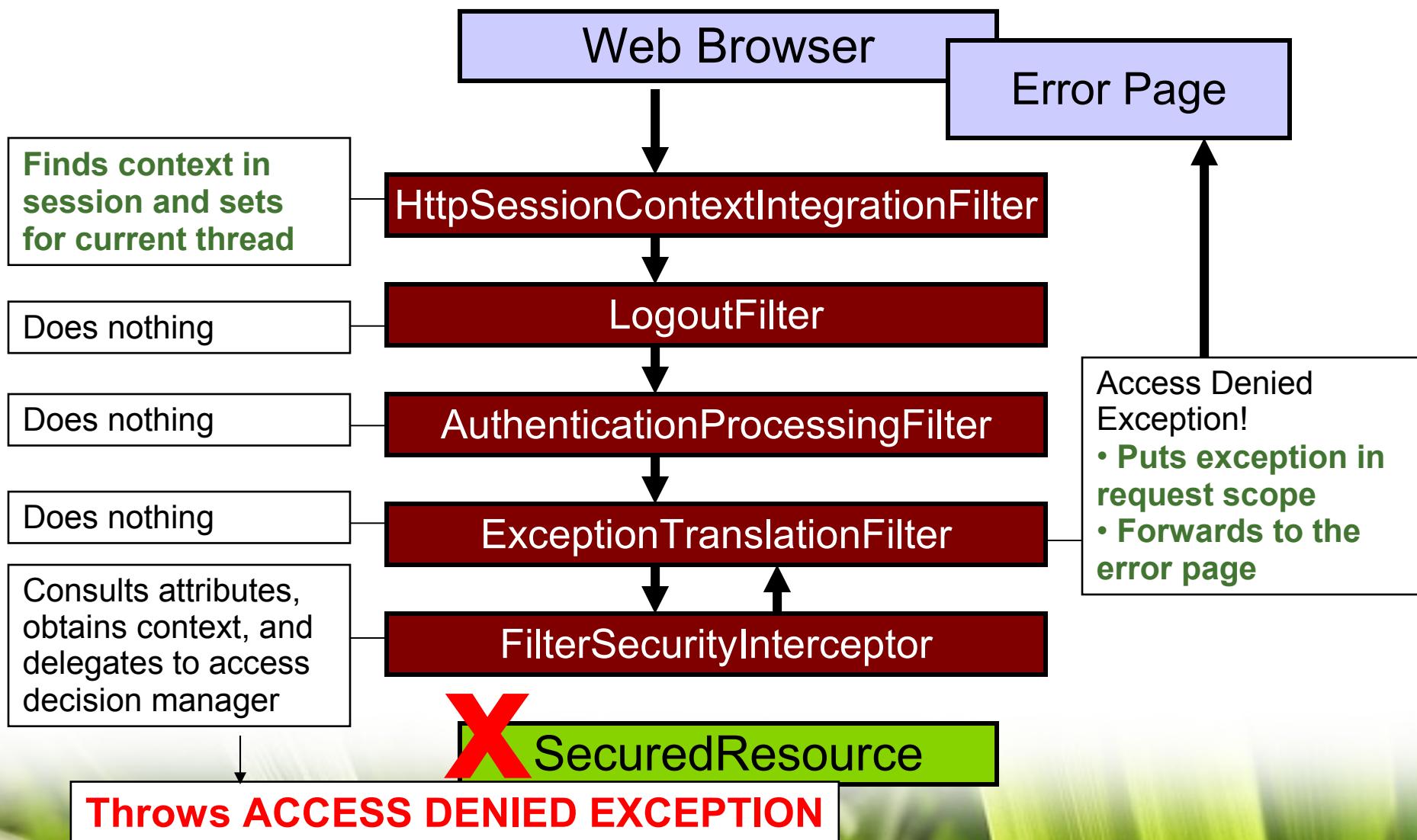
# Submit Login Request



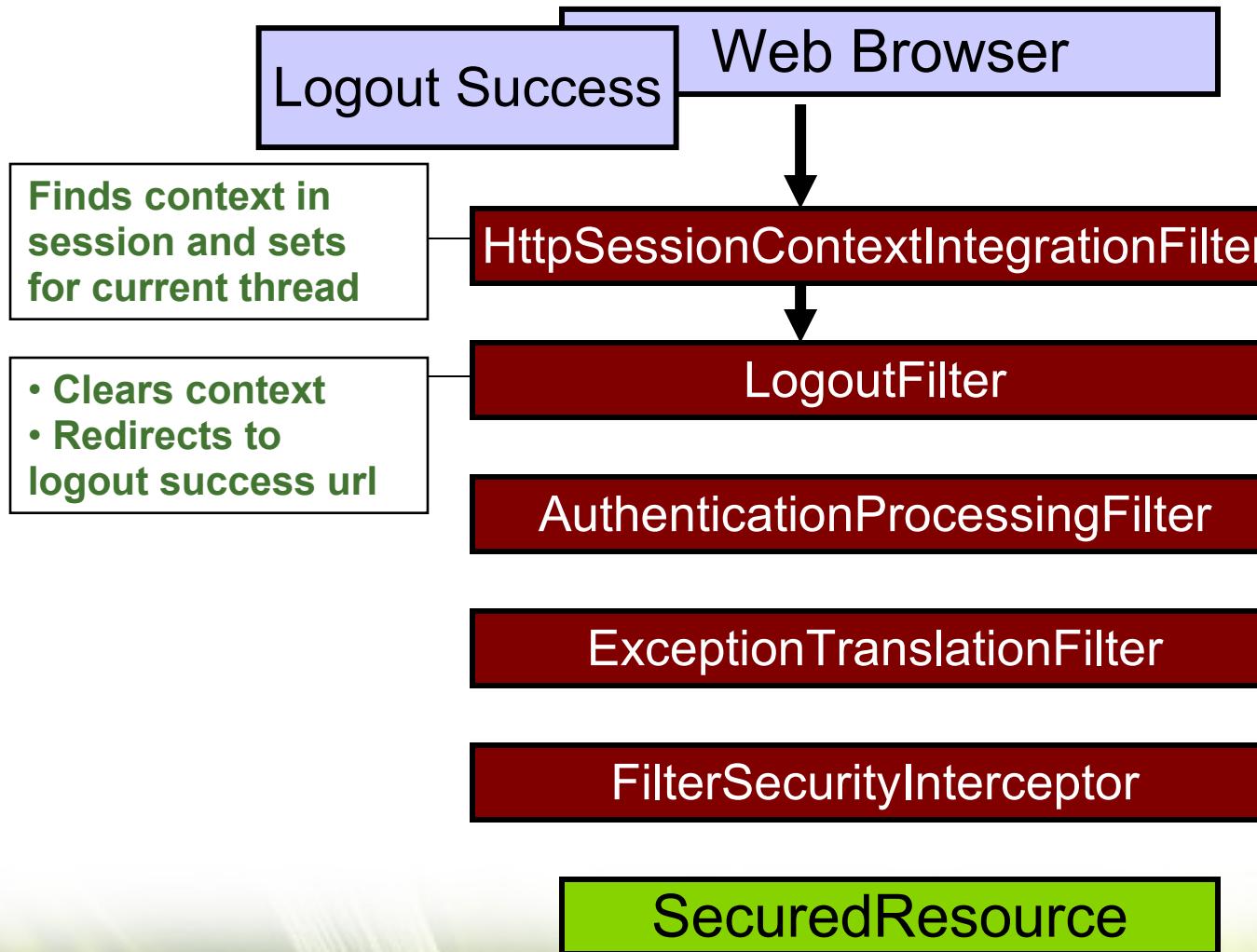
# Access Resource With Required Role



# Access Resource Without Required Role



# Submit Logout Request



# The Filter Chain: Summary



#	Filter Name	Main Purpose
1	HttpSessionContextIntegrationFilter	Establishes SecurityContext and maintains between HTTP requests
2	LogoutFilter	Clears SecurityContextHolder when logout requested
3	Authentication Processing Filter	Puts Authentication into the SecurityContext on login request
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>
 <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>
 <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

---



- **Goals of Spring Remoting**
- Spring Remoting Overview
- Supported Protocols
  - RMI
  - HttpInvoker
  - Hessian/Burlap

# Goals of Spring Remoting



- 
- Hide “plumbing” code
  - Configure and expose services declaratively
  - Support multiple protocols in a consistent way

# The Problem with Plumbing Code



- Remoting mechanisms provide an abstraction over transport details
- These abstractions are often leaky
  - The code must conform to a particular model
- For example, with RMI:
  - Service interface extends **Remote**
  - Service class extends **UnicastRemoteObject**
  - Client must catch **RemoteExceptions**

**Violates a separation of concerns**  
**Couples business logic to remoting infrastructure**

# Hiding the Plumbing with Spring



- 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

# The Declarative Approach



- Spring's abstraction uses a configuration-based approach
- 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

# 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
  - Migrate between remote vs. local deployments

# Topics in this Session

---

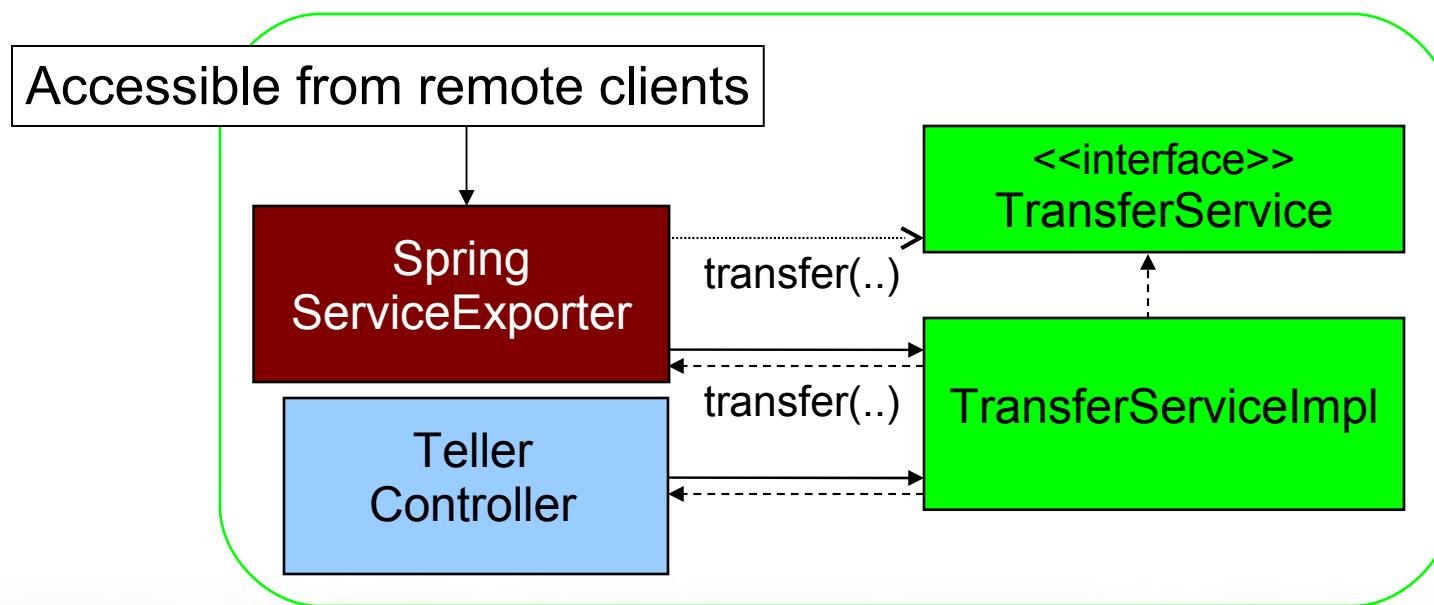


- Goals of Spring Remoting
- **Spring Remoting Overview**
- Supported Protocols
  - RMI
  - HttpInvoker
  - Hessian/Burlap

# Service Exporters



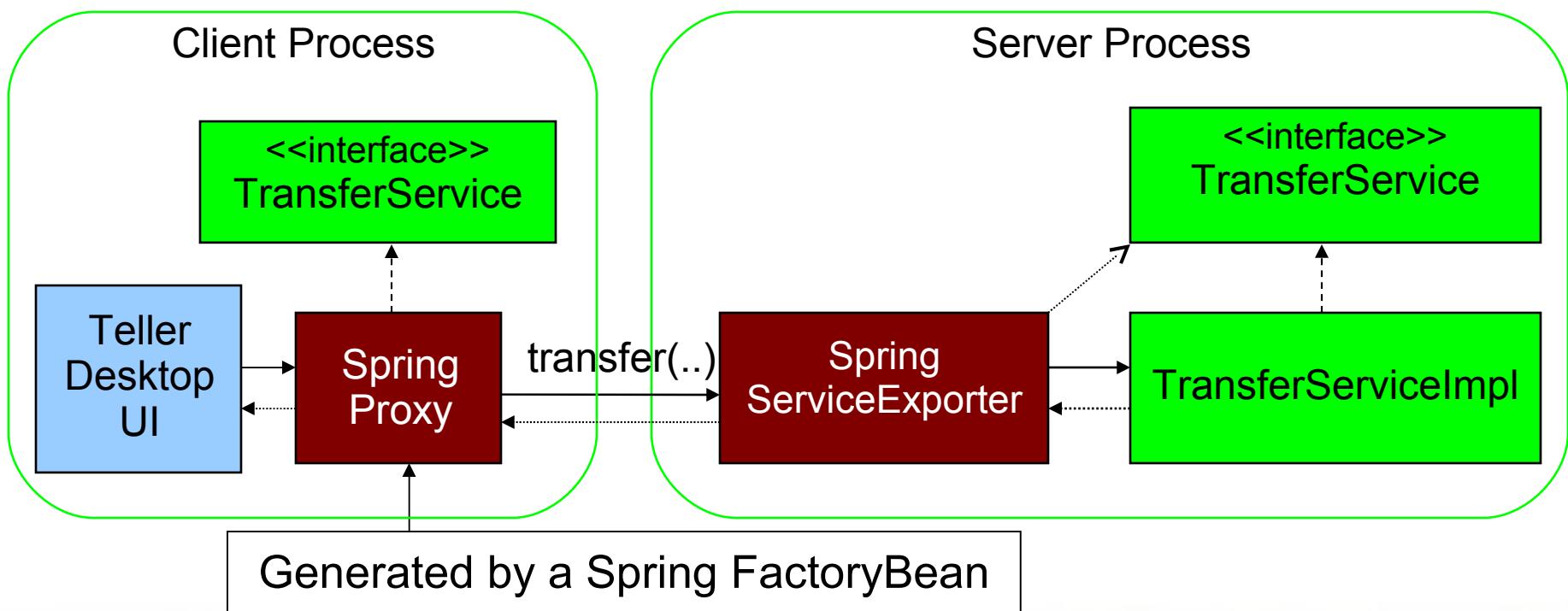
- Spring provides service exporters to enable declarative exposing of existing services



# Client Proxies



- Dynamic proxies generated by Spring communicate with the service exporter



# Topics in this Session

---



- Goals of Spring Remoting
- Spring Remoting Overview
- **Supported Protocols**
  - RMI
  - HttpInvoker
  - Hessian/Burlap

# The RMI Protocol

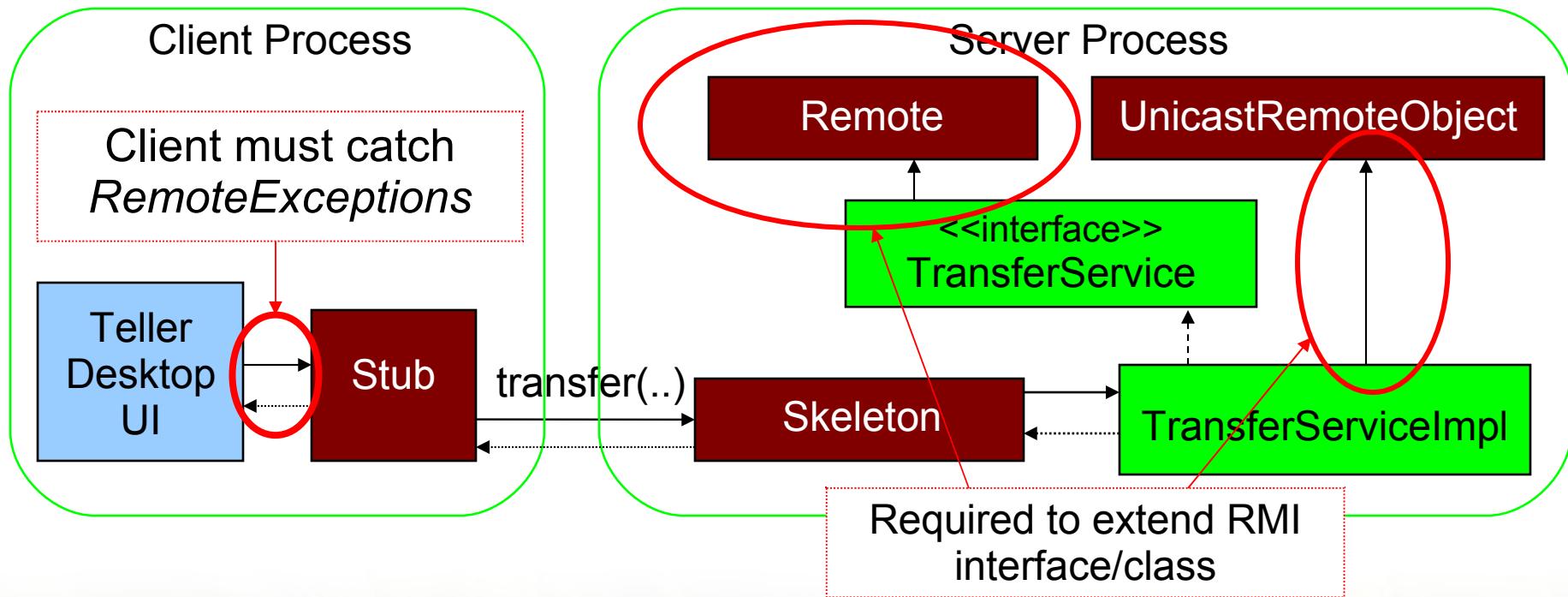


- Standard Java remoting protocol
- Server-side exposes a *skeleton*
- Client-side invokes methods on a *stub* (proxy)
- Java serialization is used for marshalling

# Traditional RMI



- The RMI model is invasive - server *and* client code is coupled to the framework



# 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

# Configuring the RMI Service Exporter



- Start with an existing POJO service

```
<bean id="transferService" class="foo.TransferServiceImpl">
 <property name="accountRepository" ref="accountRepository"/>
</bean>
```

- Define a bean to export it

Binds to rmiRegistry as “transferService”

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
 <property name="serviceName" value="transferService"/> ←
 <property name="serviceInterface" value="foo.TransferService"/>
 <property name="service" ref="transferService"/>
</bean>
```

Can also specify ‘registryPort’ (default is 1099)

```
<property name="registryPort" value="1096"/>
```

# Spring's RMI Proxy Generator



- Spring provides a FactoryBean implementation that generates an RMI client-side proxy
- It is simpler to use than a traditional RMI stub
  - Converts checked RemoteExceptions into Spring's *runtime* hierarchy of RemoteAccessExceptions
  - Dynamically implements the business interface

**Proxy is a drop-in replacement for a local implementation  
(especially convenient with dependency injection)**

# Configuring the RMI Proxy



- Define a factory bean to generate the proxy

```
<bean id="transferService"
 class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
 <property name="serviceInterface" value="foo.TransferService"/>
 <property name="serviceUrl" value="rmi://foo:1099/transferService"/>
</bean>
```

TellerDesktopUI only depends on the TransferService interface

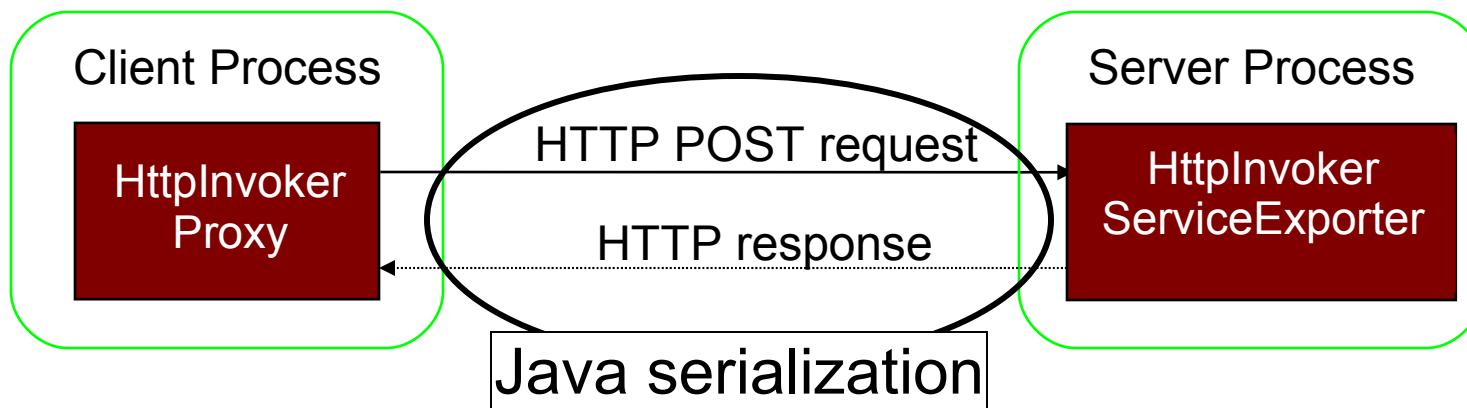
- Inject it into the client

```
<bean id="tellerDesktopUI" class="foo.TellerDesktopUI">
 <property name="transferService" ref="transferService"/>
</bean>
```

# Spring's HttpInvoker



- A lightweight HTTP-based remoting protocol
  - Method invocation is converted to an HTTP POST
  - Method result is returned as an HTTP response
  - Method parameters and return values are marshalled with standard Java serialization



# Configuring the HttpInvoker Service Exporter



- Start with an existing POJO service

```
<bean id="transferService" class="foo.TransferServiceImpl">
 <property name="accountRepository" ref="accountRepository"/>
</bean>
```

- Define a bean to export it

```
<bean name="/transfer" <-- endpoint for HTTP request handling
 class="org.springframework.remoting.httpinvoker.
 HttpInvokerServiceExporter">
 <property name="serviceInterface" value="foo.TransferService"/>
 <property name="service" ref="transferService"/>
</bean>
```

# Configuring the HttpInvoker Proxy



- Define a factory bean to generate the proxy

```
<bean id="transferService"
 class="org.springframework.remoting.httpinvoker.
 HttpInvokerProxyFactoryBean">
 <property name="serviceInterface" value="foo.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="foo.TellerDesktopUI">
 <property name="transferService" ref="transferService"/>
</bean>
```

# Hessian and Burlap



- Cauchy created these two lightweight protocols for sending XML over HTTP
  - Hessian uses binary XML (more efficient)
    - Implementations for many languages
  - Burlap uses textual XML (human readable)
- The Object/XML serialization/deserialization relies on a proprietary mechanism
  - Better performance than Java serialization
  - Less predictable when working with complex types

# Hessian and Burlap Configuration

---



- Service exporter configuration is identical to `HttpInvokerServiceExporter` except class names
  - `org.springframework.remoting.caucho.HessianServiceExporter`
  - `org.springframework.remoting.caucho.BurlapServiceExporter`
- Proxy configuration is identical to `HttpInvokerProxyFactoryBean` except class names
  - `org.springframework.remoting.caucho.HessianProxyFactoryBean`
  - `org.springframework.remoting.caucho.BurlapProxyFactoryBean`

# Choosing a Remoting Protocol (1)

---



- Spring on server and client?
  - HttpInvoker
- Java environment but no web server?
  - RMI
- Interop with other languages using HTTP?
  - Hessian (workable, but not ideal)
- Interop with other languages without HTTP?
  - RMI-IIOP (CORBA)

# Choosing a Remoting Protocol (2)

---



- Also consider the relationship between server and client
- All of the protocols discussed here are based upon Remote Procedure Calls (RPC)
  - Clients need to know details of method invocation
    - Name, parameters, and return value
  - When using Java serialization
    - Classes/interfaces must be available on client
    - Versions must match
- If serving public clients beyond your control, Web Services are usually a better option
  - Document-based messaging promotes loose coupling



# 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

# Java Message Service (JMS)



- The JMS API provides an abstraction for accessing Message Oriented Middleware
  - Avoid vendor lock-in
  - Increase portability

# JMS Core Components



- 
- Message
  - Destination
  - Connection
  - Session
  - MessageProducer
  - MessageConsumer

# 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

# The JMS Connection



- A JMS Connection is obtained from a factory

```
Connection conn = connectionFactory.createConnection();
```

- In a typical enterprise application, the ConnectionFactory is a managed resource and 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

- 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

## 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

# Configuring JMS Resources with Spring

---



- Spring enables decoupling of your application code from the underlying infrastructure
  - The container provides the resources
  - The application is simply coded against the API
- This provides deployment flexibility
  - use a standalone JMS provider
  - use an ApplicationServer to manage JMS resources

# Configuring a ConnectionFactory

---



- The ConnectionFactory may be standalone

```
<bean id="connectionFactory"
 class="org.apache.activemq.ActiveMQConnectionFactory">
 <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

- Or it may be retrieved from JNDI

```
<jee:jndi-lookup id="connectionFactory"
 jndi-name="jms/ConnectionFactory"/>
```

# Configuring Destinations



- The Destinations may be standalone

```
<bean id="orderQueue"
 class="org.apache.activemq.command.ActiveMQQueue">
 <constructor-arg value="queue.order"/>
</bean>
```

- Or they may be 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

# Spring's JmsTemplate



- The template simplifies usage of the API
  - Reduces boilerplate code
  - Manages resources transparently
  - Handles exceptions properly
  - Converts checked exceptions to runtime equivalents
  - Provides convenience methods and callbacks

# JmsTemplate Strategies



- The JmsTemplate delegates to collaborators to handle some of the work
  - MessageConverter
  - DestinationResolver

# MessageConverter



- The JmsTemplate uses a MessageConverter to convert between objects and messages
- The default SimpleMessageConverter handles basic types
  - String to TextMessage
  - Serializable to ObjectMessage
  - Map to MapMessage
  - byte[] to BytesMessage

# Implementing MessageConverter

---



- It is sometimes desirable to provide your own conversion strategy
  - Reuse existing code
  - Delegate to an object-to-XML translator
- Implement the two necessary methods

```
Message toMessage(Object o, Session session)
Object fromMessage(Message message)
```

- Provide the implementation to JmsTemplate via dependency injection

- It is often necessary to resolve destination names at runtime
- JmsTemplate uses DynamicDestinationResolver as a default
- The JndiDestinationResolver is also available
- The interface only requires one method

```
Destination resolveDestinationName(Session session,
 String destinationName,
 boolean pubSubDomain)
throws JMSEException;
```

# Defining a JmsTemplate Bean



- Provide a reference to the ConnectionFactory
- Optionally provide other references
  - MessageConverter
  - DestinationResolver
  - Default Destination (or default Destination name)

```
<bean id="jmsTemplate"
 class="org.springframework.jms.core.JmsTemplate">
 <property name="connectionFactory" ref="connectionFactory"/>
 <property name="defaultDestination" ref="orderQueue"/>
</bean>
```

# CachingConnectionFactory



- JmsTemplate aggressively closes and reopens JMS resources like Sessions and Connections
  - Assumes these are cached by ConnectionFactory
- Without caching this causes lots of overhead
  - Resulting in poor performance
- Use 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>
```

# Topics in this Session

---



- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- **Sending Messages**
- Receiving Messages

# Sending Messages



- The template provides options
  - 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 Messages with Conversion

---



- Leveraging the template's MessageConverter

```
public void convertAndSend(Object message);
```

```
public void convertAndSend(Destination destination,
 Object message);
```

```
public void convertAndSend(String destinationName
 Object message);
```

# Sending Messages with Callbacks

---



- When more control is needed, use callbacks

```
public void send(MessageCreator messageCreator);
```

```
public Object execute(ProducerCallback<T> action);
```

```
public Object execute(SessionCallback<T> action);
```



```
Message createMessage(Session session) {...}
```

# Creating the queue reference yourself using a callback



```
jmsTemplate.execute(new SessionCallback<Object>() {

 public Object doInJms(Session session) throws JMSException {

 Queue queue = session.createQueue("someQueue");
 MessageProducer producer =
 session.createProducer(queue);
 Message message =
 session.createTextMessage("Hello Queue!");
 producer.send(message);
 return null;
 }

});
```

# Topics in this Session

---



- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- **Receiving Messages**

# Synchronous Message Reception

---



- JmsTemplate can also receive messages, but methods are blocking (with optional timeout)
  - receive()
  - receive(Destination destination)
  - receive(String destinationName)
- The MessageConverter can be leveraged for message reception as well

```
Object someSerializable =
 jmsTemplate.receiveAndConvert(someDestination);
```

# The JMS MessageListener



- The JMS API defines this interface for asynchronous reception of messages

```
public void onMessage(Message) {
 // handle the message
}
```

# Spring's MessageListener Containers

---



- Traditionally, use of MessageListener implementations required an EJB container
- 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 allows for tweaking of task execution strategy, concurrency, container type, transaction manager and more

# Spring's message-driven objects



- Spring also allows you to specify a plain Java object that can serve as a listener

```
<jms:listener ref="mySimpleObject" ①
 method="order" ②
 destination="queue.orders"
 response-destination="queue.confirmation"/> ③
```

```
public class OrderService { ①
 public OrderConfirmation order(Order o) { ②
 } ③
```

- Parameter automatically converted using a MessageConverter
- Return value sent to response-destination



# LAB

Sending and Receiving Messages in a Spring Environment



# 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

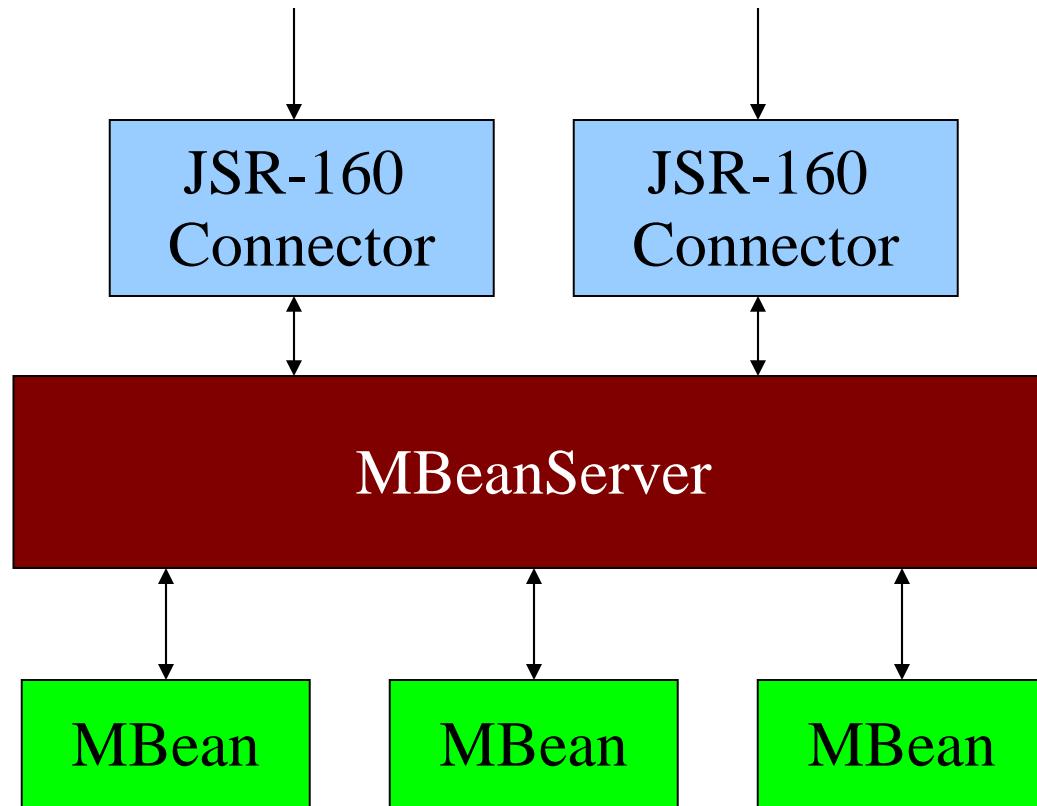
# How JMX Works

---



- 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

---



- An MBeanServer acts as a broker for communication between
  - Multiple local MBeans
  - Remote clients and MBeans
- An MBeanServer maintains a keyed reference to all MBeans registered with it
- Many generic clients available
  - jconsole, jvisualvm, Hyperic

# JMX Architecture



- 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



- Annotate your class and methods to be exposed

```
@ManagedResource(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() {...}
}
```

- Activate exporter

```
<context:mbean-export/>
```

- ObjectName derived from fully qualified class name
  - or passed as attribute to @ManagedResource

# 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

---



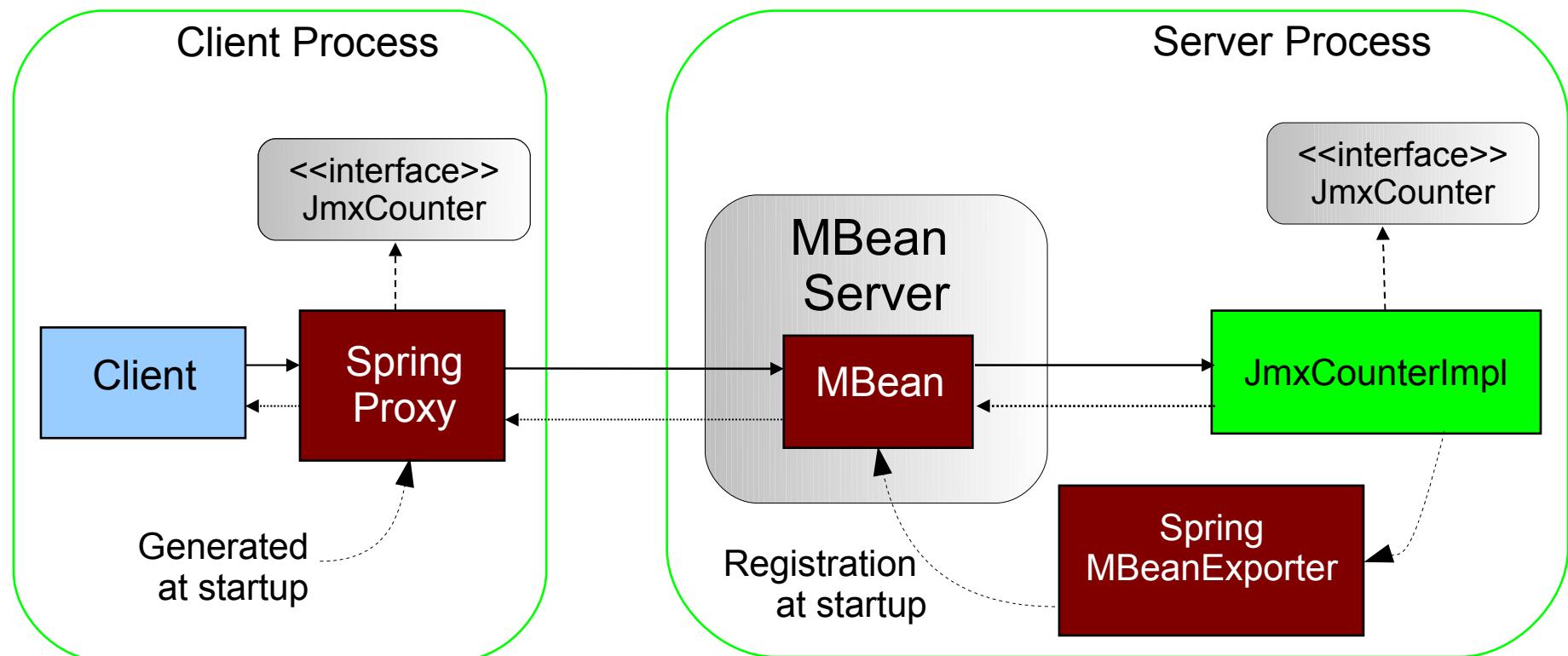
- Start with an existing POJO bean

```
<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
  - e.g. Hibernate StatisticsService
- Spring can easily autodetect those and export them for you

```
<context:mbean-export>

<bean id="statisticsService"
 class="org.hibernate.jmx.StatisticsService">
 <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

# Automatically exporting pre-existing MBeans

---



- Log4j - LoggerDynamicMBean

```
<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>
```

# Summary

---

- Spring JMX allows you to easily export Spring-managed beans to a JMX MBeanServer
  - Simple value-add now that your beans are managed
- Use <context:mbean-server> to create MBean server
- Use Spring annotations to declare JMX metadata
- Use <context:mbean-export> to automatically export pre-existing MBeans

# 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 into tc Server
- Focuses on what's relevant
  - esp. performance related parts of the application
- Detects performance issues during development

# Spring Insight Overview



The screenshot shows the Spring Insight Dashboard - Recent Activity page. The interface includes:

- Application Selector:** A dropdown menu on the right side labeled "swf-booking-mvc".
- Trace History:** A bar chart showing response times over time. The Y-axis ranges from 0 ms to 2.0 s. The X-axis shows dates: 10/13/2009 3:11 PM and 10/13/2009 3:14 PM. Several bars are visible between these two points.
- Traces:** A list of requests made between 3:14:10 PM and 3:14:13 PM. The list includes:
  - 115 ms 3:14:11 PM GET /swf-booking-mvc/spring/hotels/index
  - 109 ms 3:14:10 PM GET /swf-booking-mvc/spring/hotels/index
  - 108 ms 3:14:12 PM GET /swf-booking-mvc/spring/hotels/booking?execution=e7s1
  - 104 ms 3:14:13 PM GET /swf-booking-mvc/spring/hotels/index
  - 103 ms 3:14:13 PM GET /swf-booking-mvc/spring/hotels/search?searchString=&pageSize=5
  - 102 ms 3:14:11 PM GET /swf-booking-mvc/spring/hotels/index
  - 100 ms 3:14:13 PM GET /swf-booking-mvc/spring/hotels/search?searchString=&pageSize=5
  - 100 ms 3:14:11 PM POST /swf-booking-mvc/spring/hotels/booking?execution=e4s1
- Trace Detail:** A detailed view of the trace for the request at 3:14:13 PM (100 ms). It shows the call stack:
  - GET /swf-booking-mvc/spring/hotels/search?searchString=&pageSize=5 (100 ms)
  - Spring Web Dispatch (77 ms)
    - @ModelAttribute org.springframework.webflow.samples.booking.HotelsCont (3 ms)
    - Transaction [BookingService.findHotels] (2 ms)
      - JpaBookingService#findHotels(SearchCriteria) (2 ms)
      - JpaBookingService#getSearchPattern(SearchCriteria) (0 ms)
      - JDBC SELECT (0 ms)
  - Resolve view "hotels/search" (0 ms)
  - Render view "hotels/search" (2 ms)

# 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)
- Where?
  - In any Pearson VUE Test Center
    - There are some in most big or medium-sized cities
    - See <http://www.pearsonvue.com/vtclocator/>

# Certification (2)

---

- How to prepare for the exam?
  - See the Core Spring 3.x certification guide here:  
<http://www.springsource.com/training/certification/springprofessional>
  - Review all the slides
  - Redo the labs
- How to register?
  - At the end of the class, you will receive a voucher by email
  - For any further inquiry, you can write to  
[springsourceuniversity@vmware.com](mailto:springsourceuniversity@vmware.com)

# Other courses

---

- Many courses available
  - Rich 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>

- 4-day workshop
- Making the most of Spring in the web layer
  - Spring MVC and Spring Web Flow
  - Rich User Interfaces, Spring JavaScript, Dojo
  - JSF with Spring Faces
  - Flex clients with Spring BlazeDS
  - Productivity with Roo and Grails
- Spring Web Application Developer certification



- 4 day course that will give you the opportunity to:
  - Discover Spring Batch
  - Discover Spring Integration
  - Work with advanced transaction management
  - Learn how to work with concurrency
  - Learn Web Services best practices

# Hibernate with Spring

---



- 3 day course that will give you the opportunity to:
  - Configure Hibernate applications with Spring and Spring Transactions (overlaps this course)
  - 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 to tc Server Migration
- Tomcat to tc Server Migration
  - Migrate your application and production environment to tc Server
- And custom consulting engagements that fits 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 via RSS
  - <https://fisheye.springsource.org/browse/spring-framework/trunk>

# Thank You!

---

We hope you enjoyed the course

Please fill out the evaluation form at

**<http://www.springsource.com/training/evaluation>**





# Object/Relational Mapping with Spring and Java Persistence API

# Topics in this session

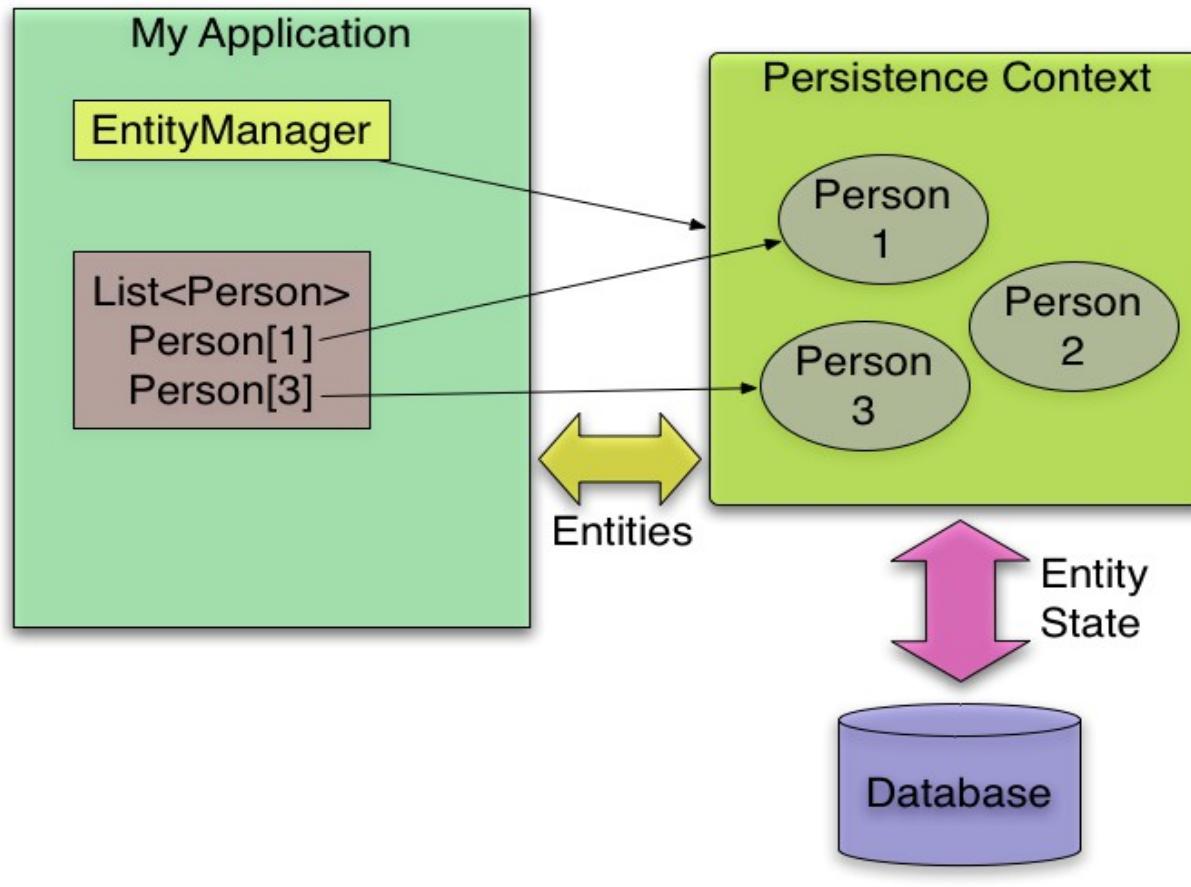
---

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

- The **EntityManager** represents a unit of work
  - Also known as *PersistenceContext*
  - Corresponds at a higher-level to a Connection
  - Manages persistent objects within the unit-of-work

- An **EntityManagerFactory** is a thread-safe, shareable object that represents a single data source
  - Provides access to a transactional EntityManager
- A **PersistenceUnit** describes a group of persistent classes
  - Defines providers
  - Defines transactional types (local vs JTA)
  - Multiple Units per application are allowed

# 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 entity state to be written to database immediately

Plus other methods ...

# JPA Providers

---

- Several major implementations of JPA spec
  - EclipseLink/Toplink (RI)
  - Apache OpenJPA
  - Hibernate EntityManager

- Hibernate implements the JPA Standard through an additional library
  - The *Hibernate EntityManager*
  - Hibernate still used behind the JPA interfaces
  - Custom Annotations available for Hibernate specific extensions not covered by JPA

- JPA requires metadata for mapping classes/properties to database tables/columns
  - Usually provided as annotations
  - Can also be provided as XML (*not recommended*)
- JPA metadata relies on defaults
  - No need to provide metadata for the obvious

# What can you annotate?

---



- Classes
  - Metadata which applies to the entire class (e.g. table)
- Either fields or properties
  - Metadata which applies to a single field or property (e.g. mapped column)
  - Based on default, all fields or properties will be treated as persistent (mappings will be defaulted)
  - Can be annotated with @Transient (non-persistent)

# Mapping simple properties with annotations



```
@Entity
@Table(name= "T_CUSTOMER")
public class Customer {
 @Id ← Defines field as entity ID
 @Column (name="cust_id")
 private Long id;

 @Column (name="first_name")
 private String firstName;
 ...
```

# Relationships

- Common relationship mappings supported
  - Single entities and collections of entities both supported

```
@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

# JPA Querying

---



- JPA provides several options for querying entities
  - Retrieve an object by primary key
  - Retrieve objects with the Java Persistence Query Language (JPQL)
  - Retrieve objects using the underlying “native” query language (typically SQL)

# 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 entity does not exist

# JPA Querying: JPQL



- To query for objects based on properties or associations use JPQL
- Method calls may be chained (as in Hibernate)

```
public Customer findByLastName(String lastName) {
 return (Customer) entityManager.createQuery(
 "select c from Customer c where c.name = :lastName")
 .setParameter("lastName", lastName)
 .getSingleResult();
}
```

If multiple customers are expected, use **getResultList()**

# Topics in this session

---

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

# Setting up an EntityManagerFactory

---



- Three ways to set up an EntityManagerFactory:
  - LocalEntityManagerFactoryBean
  - LocalContainerEntityManagerFactoryBean
  - Use a JNDI lookup
- All approaches require a **persistence.xml** for configuration

- Always stored in META-INF
- Specifies:
  - persistence unit
  - 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>
```

- Useful for standalone apps, integration tests
- Cannot specify a DataSource
  - Useful when only data access is via JPA
  - Uses standard JPA service location 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 (1)

---



- Provides full JPA capabilities
- Integrates with existing DataSources
- Useful when fine-grained customization needed
  - Can specify vendor-specific configuration

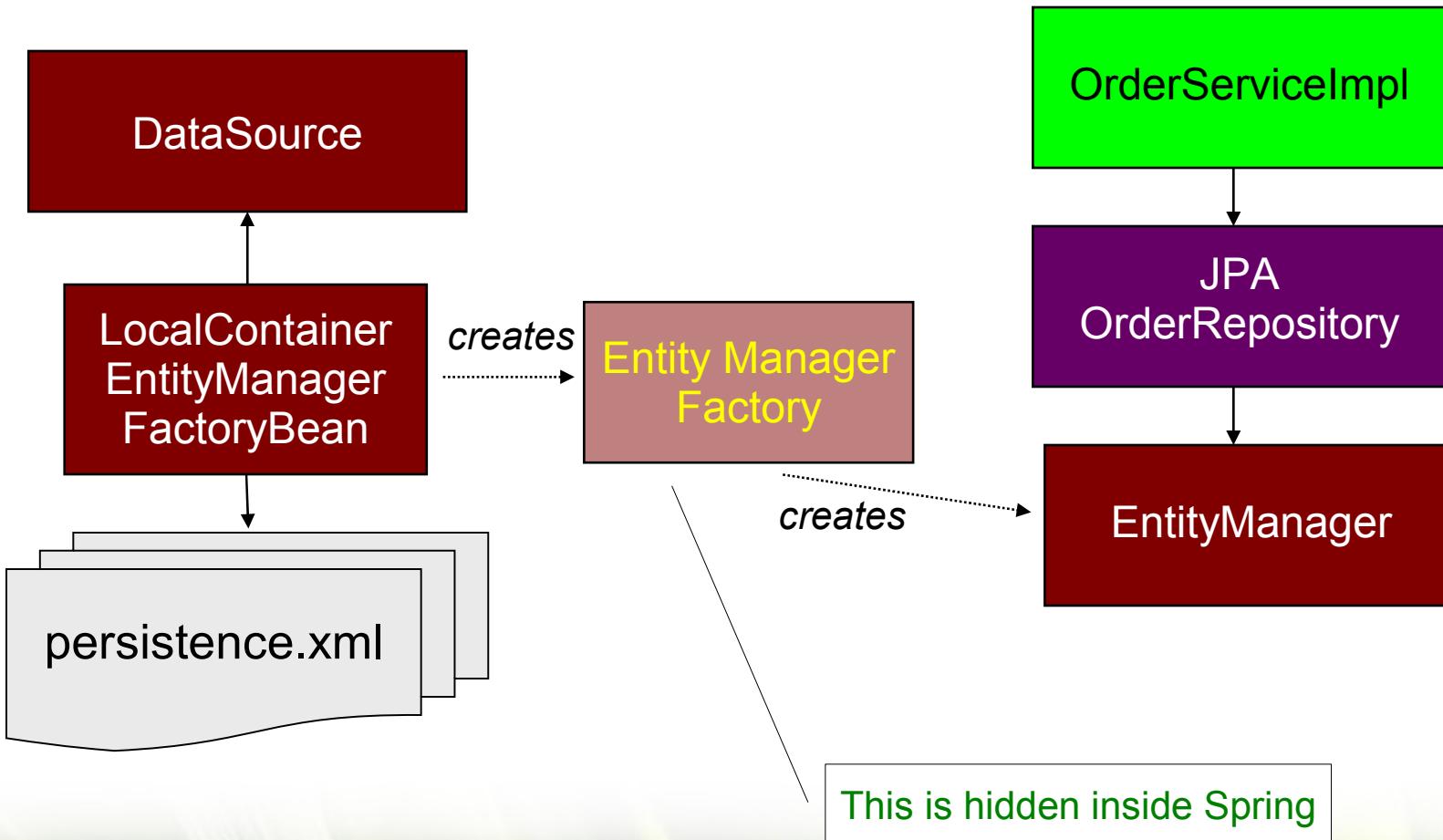


# LocalContainer EntityManagerFactoryBean (2)



```
<bean id="entityManagerFactory"
 class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="dataSource" ref="dataSource"/>
 <property name="persistenceUnitName" value="rewardNetwork"/>
 <property name="jpaVendorAdapter">
 <bean class="o.s.orm.jpa.vendor.HibernateJpaVendorAdapter">
 <property name="showSql" value="true"/>
 <property name="generateDdl" value="true"/>
 <property name="database" value="HSQL"/>
 </bean>
 </property>
 <property name="jpaProperties">
 <value>
 hibernate.format_sql=true
 </value>
 </property>
</bean>
```

# EntityManagerFactoryBeans



# 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**

# 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 – with no Spring dependencies

```
public class JpaOrderRepository implements OrderRepository {
 @PersistenceContext
 private EntityManager entityManager; ← Setter Injection also possible

 ...
 public void someDaoOperation() {
 ... // now use the already prepared entityManager
 }
}
```

# Spring-managed Transactions and EntityManager (3)



- The configuration

```
<beans>
 <bean id="entityManagerFactory"
 class="org.springframework.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>
```

Provides injection for @PersistenceContext

# 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
  - Can change persistence provider transparently

# 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 third-party code
  - Use XML to configure instead

```
public class JpaOrderRepository implements MyRepository {
 ...
}
```

No annotations

```
<bean id="persistenceExceptionInterceptor"
 class="org.springframework.dao.support.
 PersistenceExceptionTranslationInterceptor"/>

<aop:config>
 <aop:advisor pointcut="execution(* *..MyRepository+.*(..))"
 advice-ref="persistenceExceptionInterceptor" />
</aop:config>
```



# LAB

Using JPA with Spring



# 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
- Don't use validation for incoming requests
- Use XPath

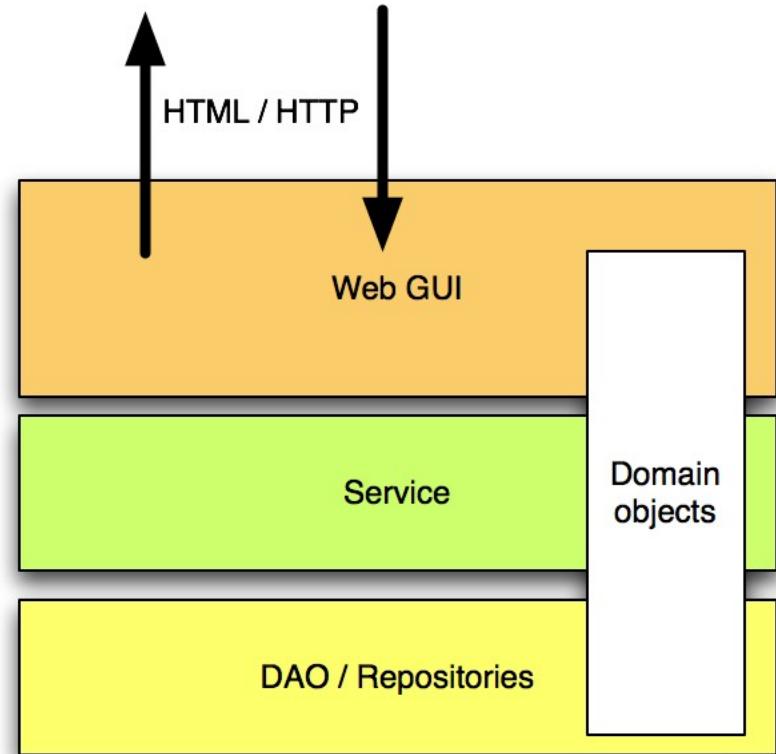
↑  
Postel's law:

“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

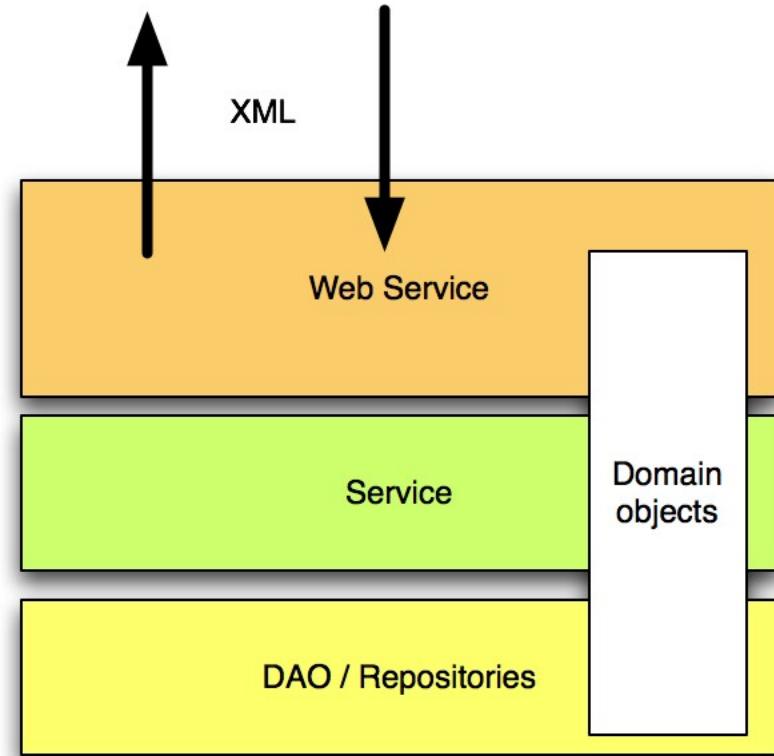


# Web Service on top of your services

---



**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

---

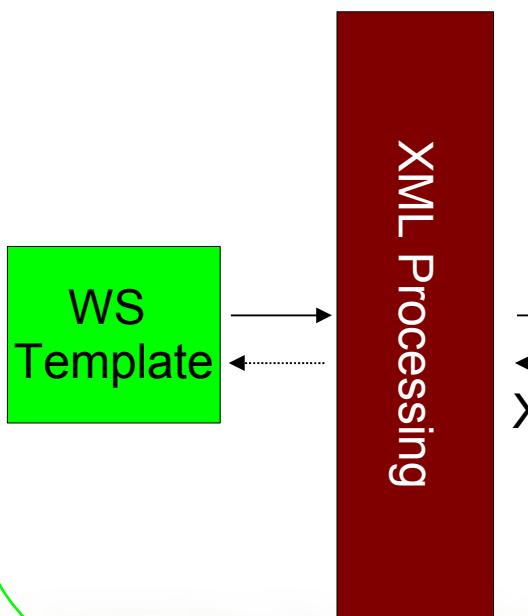


```
<SOAP-ENV:Envelope xmlns:SOAP-
 ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Body>
 <tr:transferRequest xmlns:tr="http://mybank.com/schemas/tr"
 tr:amount="1205.15">
 <tr:credit>S123</tr:credit>
 <tr:debit>C456</tr:debit>
 </tr:transferRequest>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

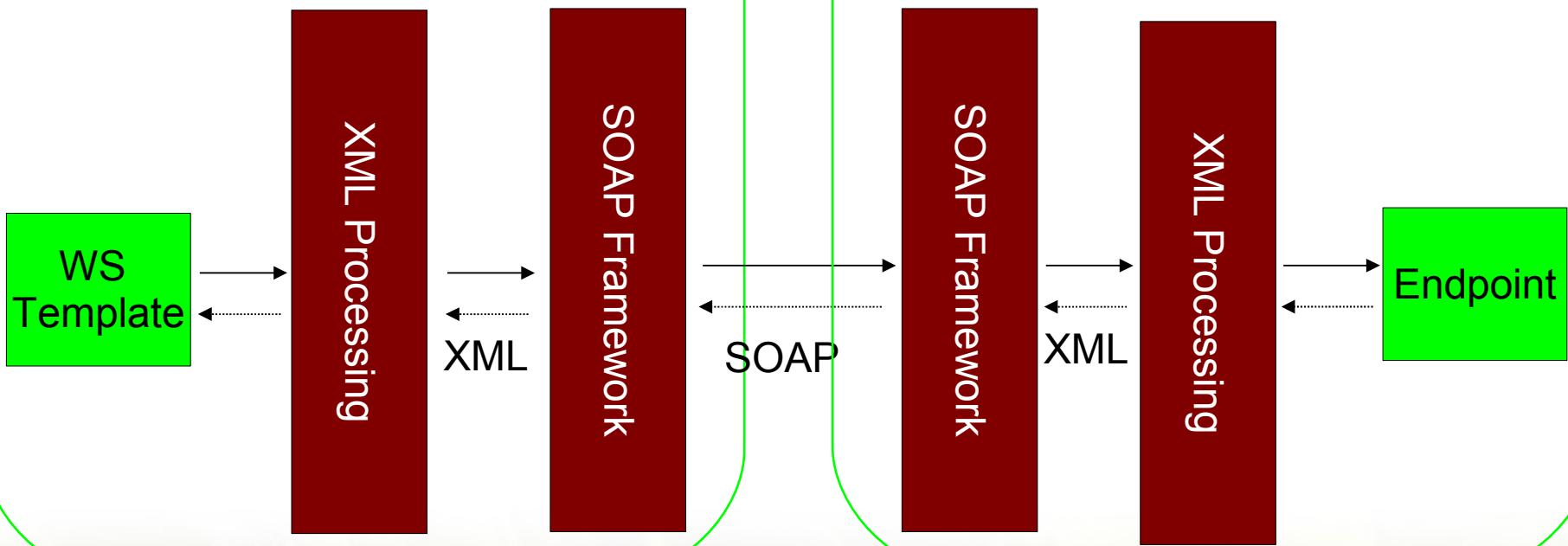
# Spring Web Services



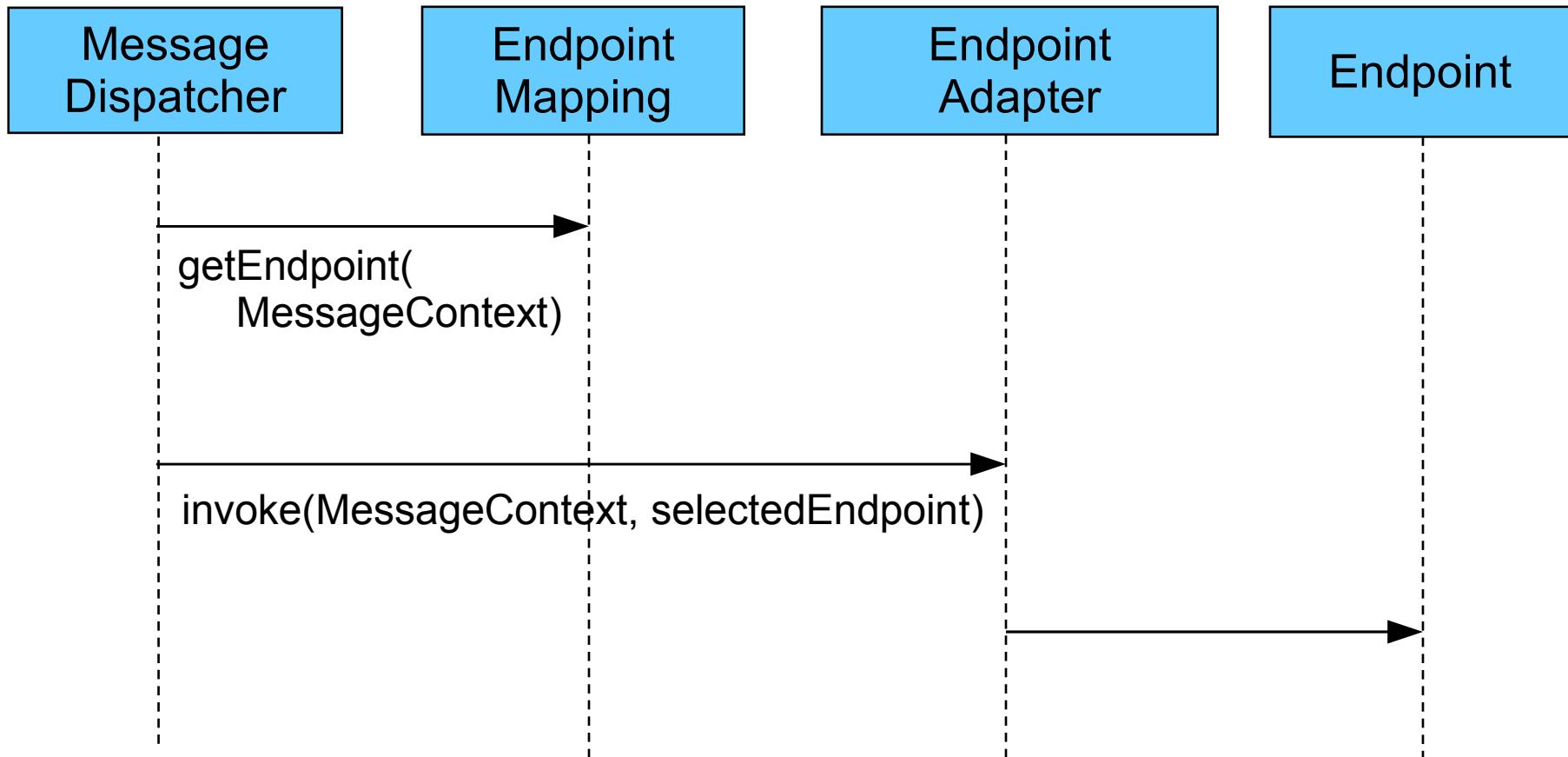
Client Process



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/transfer-ws-cfg.xml</param-value>
 </init-param>
</servlet>
```

The application context's configuration file(s)  
containing the web service infrastructure beans

# Map the Front Controller

- 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
- Generates classes from a schema
  - Xjc
- Also generates schema from classes

```
<bean id="marshaller" class="...Jaxb2Marshaller">
 <property name="contextPath" value="transfer.ws.types"/>
</bean>

<bean class="...GenericMarshallingMethodEndpointAdapter">
 <constructor-arg ref="marshaller" />
</bean>
```

# Implement the Endpoint



```
@Endpoint ← Spring WS Endpoint
public class TransferServiceEndpoint {
 private TransferService transferService;
 public TransferServiceEndpoint(TransferService transferService) {
 this.transferService = transferService;
 }
 @PayloadRoot(localPart="transferRequest",
 namespace="http://mybank.com/schemas/tr")
 public TransferResponse newTransfer(TransferRequest request) {
 // extract necessary info from request and invoke service
 }
}
```

Mapping

Converted with JAXB2

The diagram illustrates the flow of data in the Java code. A box labeled "Spring WS Endpoint" has an arrow pointing to the "@Endpoint" annotation. Another box labeled "Mapping" has arrows pointing to both the "@PayloadRoot" annotation and the "newTransfer" method. A box labeled "Converted with JAXB2" has an arrow pointing to the "newTransfer" method. The code itself shows the implementation of the TransferServiceEndpoint class, which includes a constructor that takes a TransferService, a PayloadRoot annotation for the newTransfer method, and the actual implementation of the newTransfer method which performs some processing and invokes a service.

# Configure the Endpoint

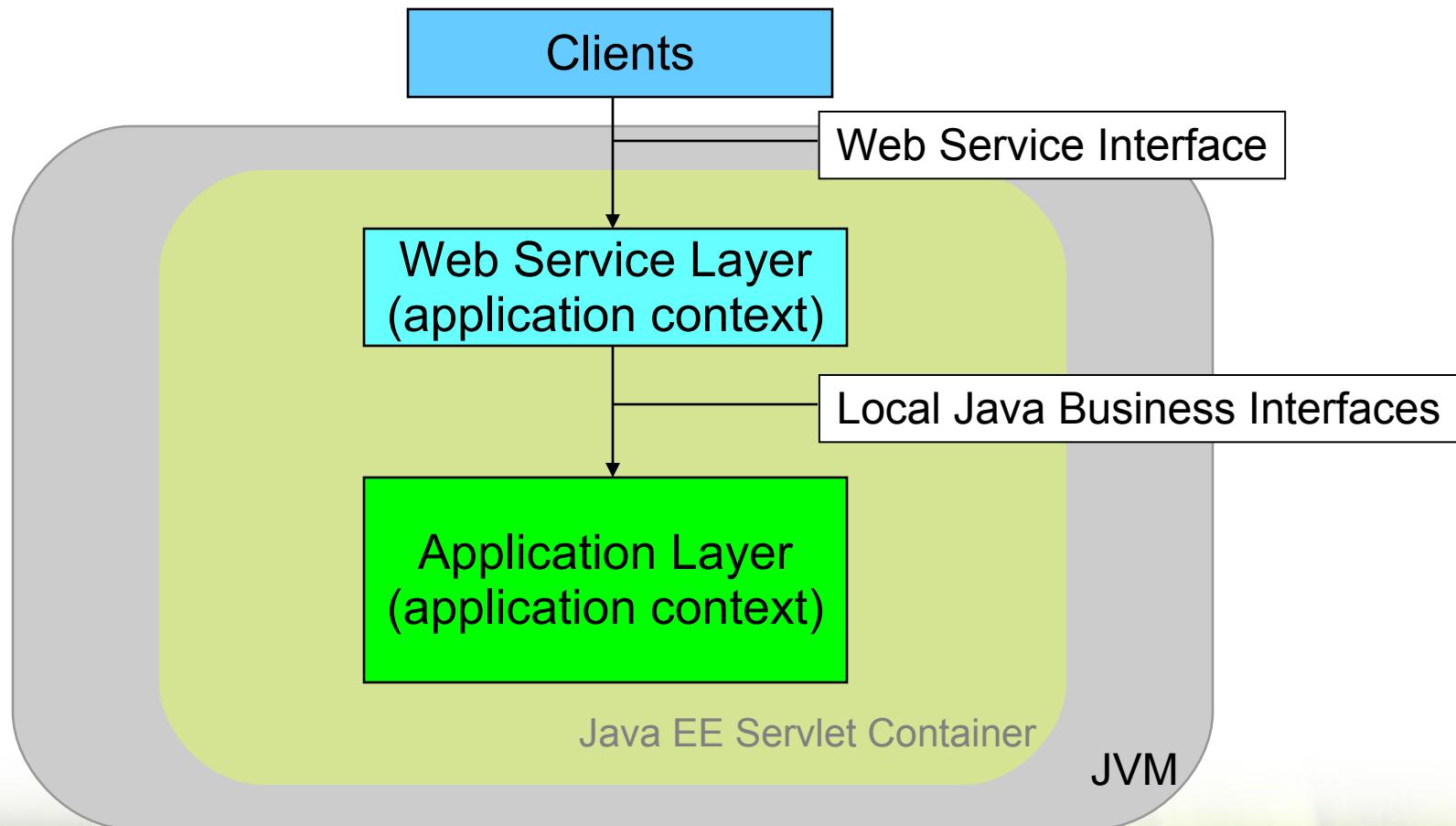
```
<bean id="transferEndpoint"
 class="example.ws.TransferServiceEndpoint">
 <constructor-arg ref="transferService"/>
</bean>
```

‘transferService’ is defined in the application tier

```
<bean class="...PayloadRootAnnotationMethodEndpointMapping" />
```

Searching for @Endpoint

# 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

# 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 template =
 (WebServiceTemplate) context.getBean("webServiceTemplate");
TransferRequest request = new TransferRequest("S123", "C456", "85.00");
Receipt receipt = (Receipt) template.marshallSendAndReceive(request);
```



# LAB

Exposing SOAP Endpoints using  
Spring Web Services