

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



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

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



- Hours
- Lunch and breaks
- Other questions?



Evaluation and other questions

- We value your input
 - <http://www.springsource.com/training/evaluation>
- Send any additional feedback or questions to
 - training@springsource.com



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
 - Testing a Spring-based application



Course Agenda: Day 2



-
- 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
 - Securing web applications with Spring Security



7

Course Agenda: Day 4



-
- Understanding Spring's remoting framework
 - Getting started with Spring Web Services
 - Simplifying message applications with Spring JMS
 - Adding manageability to an application with Spring JMX



8

SpringSource Overview



- Center of thought leadership for Java, Spring, Apache, Groovy/Grails, Hyperic
 - Rod Johnson, CEO & father of Spring; develop 99%+ of Spring
 - Tomcat leaders: 80% of code commits, 95% of bug fixes
 - Groovy/Grails leaders: acquired G2One Nov 2008
 - Hyperic app management leaders: acquired Hyperic May 2009
- Forge open source innovation into enterprise products
 - Leading enterprise adoption of lean infrastructure
- Customers include most of the Global 2000
- Offices worldwide
 - US-based, UK, Netherlands, Germany, Australia, France, Canada

9

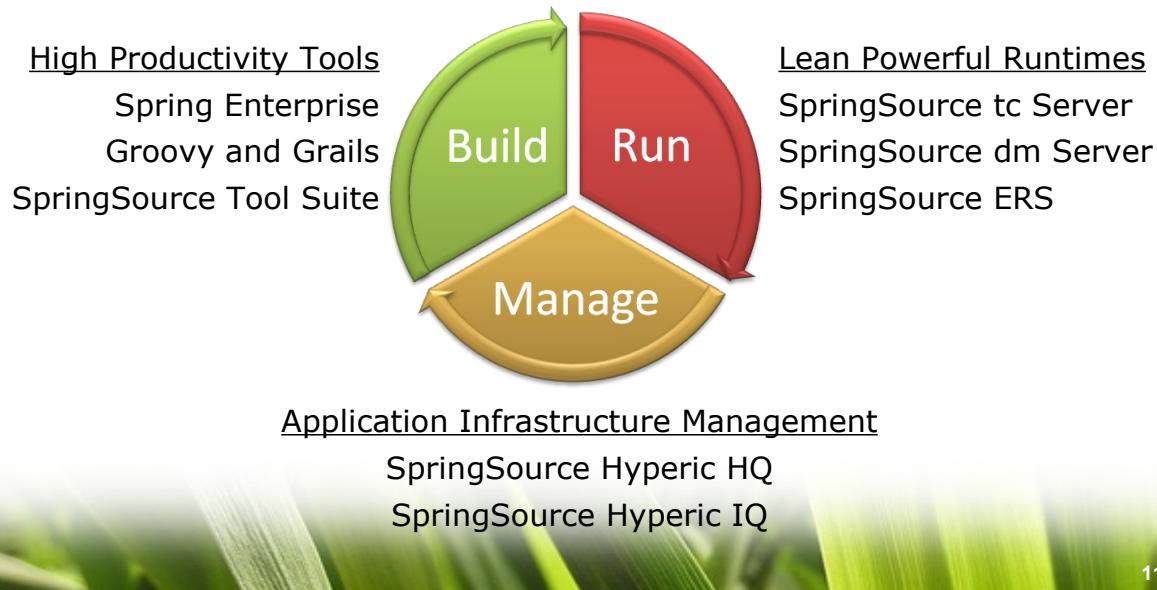
SpringSource: Open Source Leader in Enterprise Java



- **SpringSource = Spring**
 - #1 enterprise Java programming model
 - Over 6 million downloads
- **SpringSource = Tomcat**
 - #1 Java application server
- **SpringSource = Enterprise OSGi**
 - Next-generation of server modularity
- **SpringSource = Groovy & Grails**
 - Java's answer to Ruby on Rails
 - #1 dynamic language for JVM
- **SpringSource = Apache HTTP**
 - #1 Web server
- **SpringSource = Hyperic**
 - Leader in open source management

10

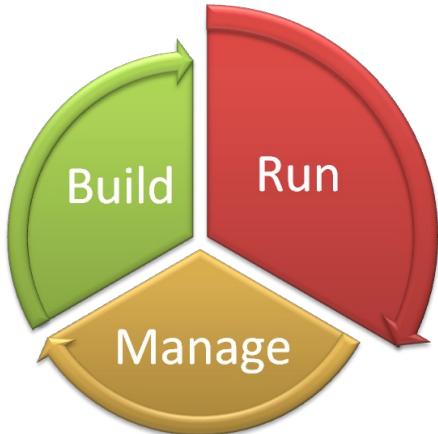
Unifying the Application Lifecycle: from Developer to Datacenter



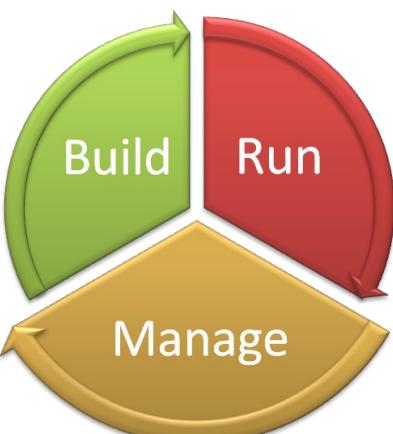
Build | Run | Manage



- Development solutions users love
 - De facto standard component model (Spring)
 - SpringSource Tool Suite (STS)
 - Groovy/Grails
 - Spring Roo
- Productivity
- Management Integration



- Enterprise grade server solutions built on open source
 - tc Server
 - dm Server
 - http Server
- High Performance
- Manageable



- Depth and Breath
 - Application metrics
 - Entire stack visibility
- Corrective Actions
- Operations Intelligence
 - BI for IT and Operations
- SLA Monitoring



- Subscriptions
 - Certified Software
 - Development and Production Support
 - Legal Indemnification
- Training
 - Public, on-site, certification program
- Professional Services
 - Client-driven engagements; packaged & custom

Delivered by Spring, Apache, Groovy/Grails and Hyperic Experts

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
 - 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



3

Spring's Support (1)



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

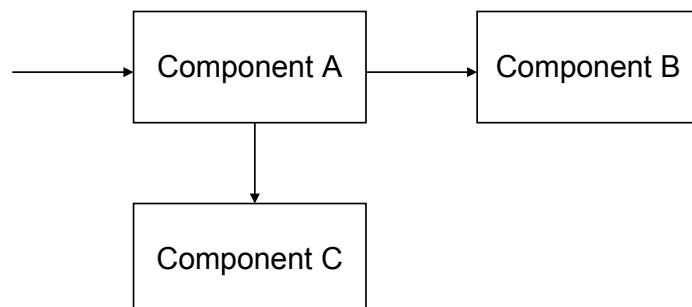


4

Application Configuration

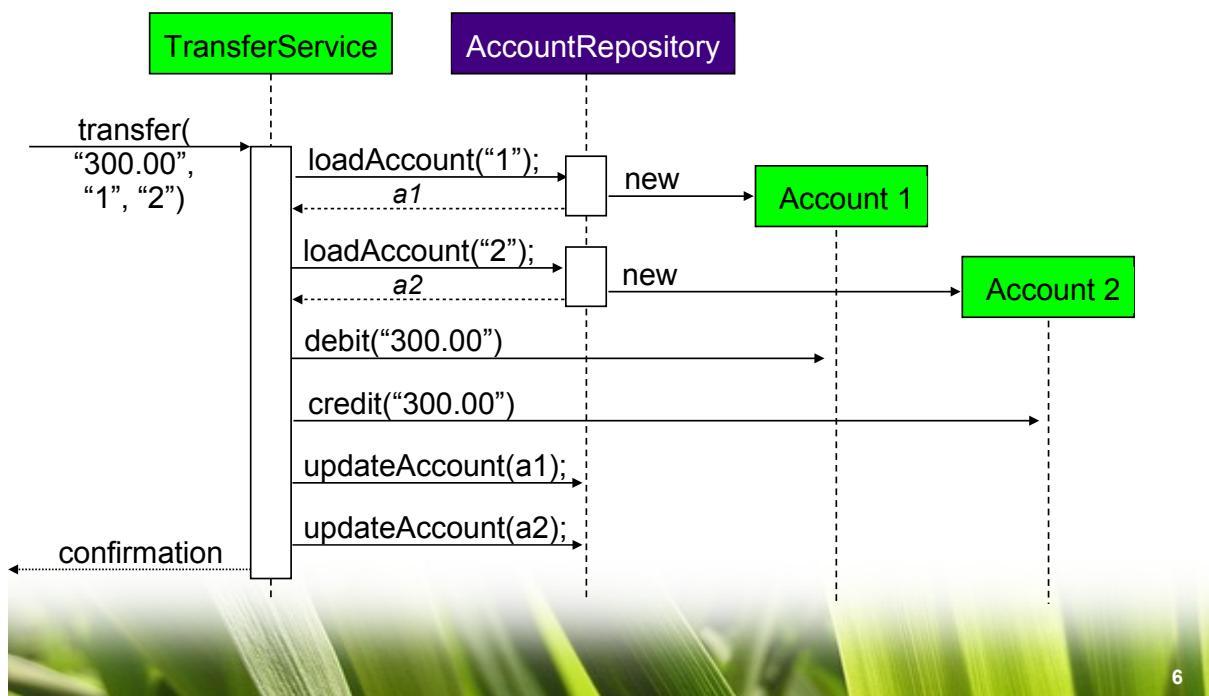


- A typical application system consists of several parts working together to carry out a use case



5

Example: A Money Transfer System



6

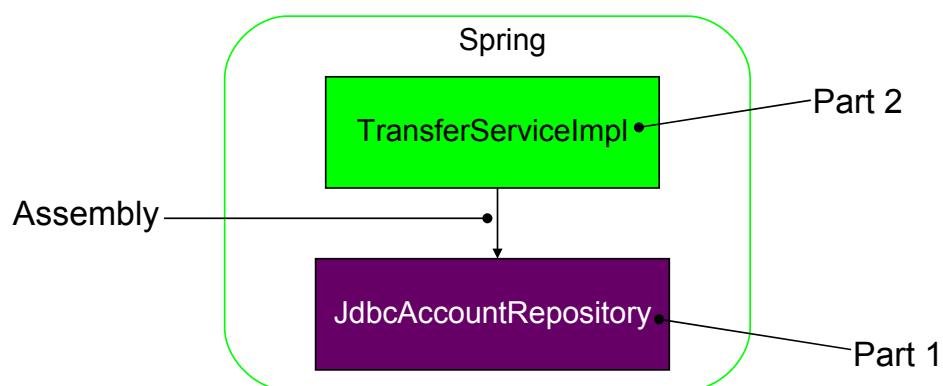
Spring's Configuration Support



- 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

7

Money Transfer System Assembly



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

8

Parts are Just Plain Java Objects



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

Implements a service interface

Part 1

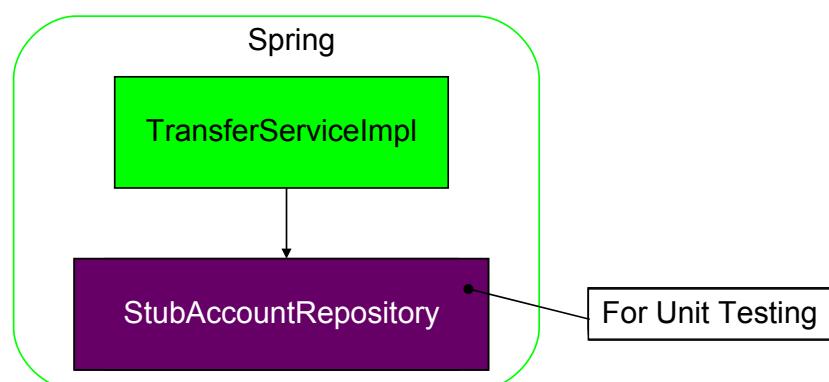
```
public class TransferServiceImpl implements TransferService {  
    private AccountRepository accountRepository;  
  
    public void setAccountRepository(AccountRepository ar) {  
        accountRepository = ar;  
    }  
    ...  
}
```

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

Part 2

9

Swapping Out Part Implementations



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

10

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

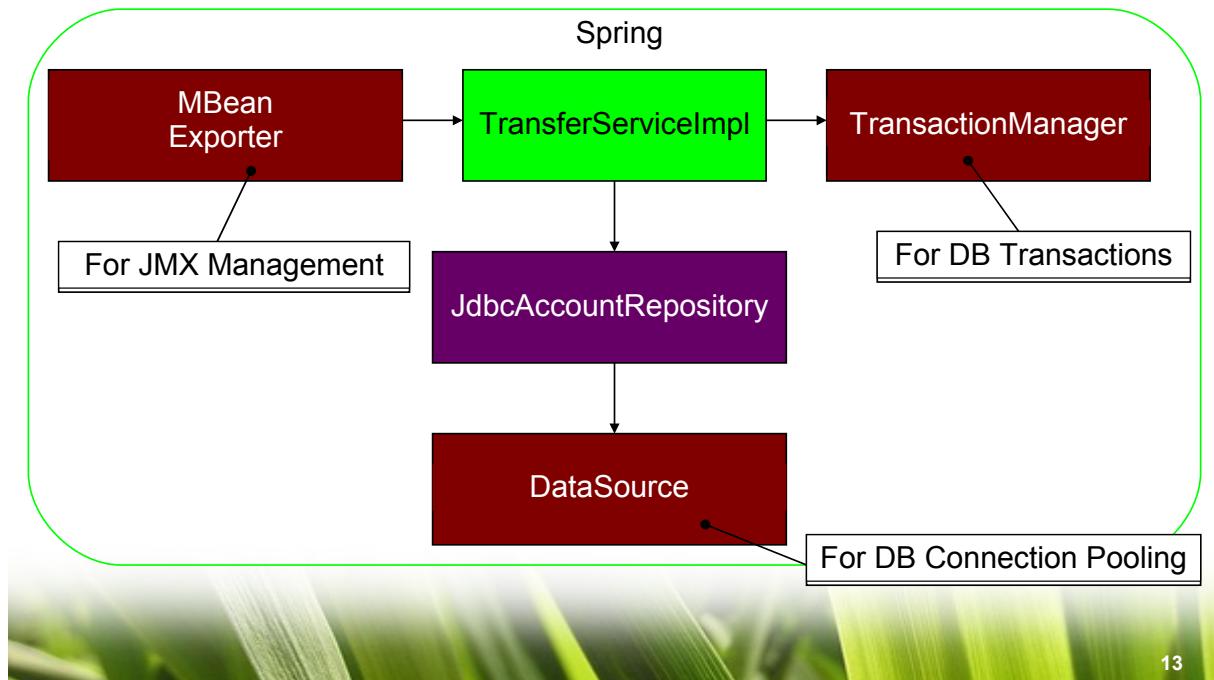


Spring Enterprise Integration

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



Enterprise Money Transfer with Spring



13

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

14

- 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



15

Enabling Unit Testing

```
public class TransferServiceImplTests {  
    private TransferServiceImpl transferService;  
  
    @Before public void setUp() {  
        AccountRepository repository = new StubAccountRepository();  
        transferService = new TransferServiceImpl(repository);  
    }  
  
    @Test public void transferMoney() {  
        TransferConfirmation confirmation =  
            transferService.transfer(new MonetaryAmount("50.00"), "1", "2");  
        assertEquals(new MonetaryAmount("100.00"),  
                    confirmation.getNewBalance());  
    }  
}
```

Minimizing dependencies increases testability

Testing logic in isolation measures unit design
and implementation correctness



16

Accessing Data



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



17

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



18

```
Spring's JDBC helper class  
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

All handled
by Spring



19

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)



20

Spring's Support (3)

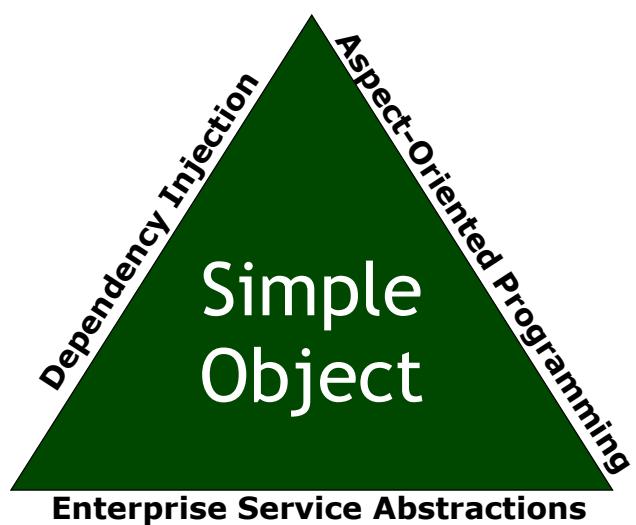


- Enterprise application development support
 - Developing web services
 - Adding manageability
 - Integrating messaging infrastructures
 - Securing services and providing object access control
 - Scheduling jobs

21



The Spring Triangle



22



LAB

Developing an Application from Plain Java Objects



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.



Spring Quick Start

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



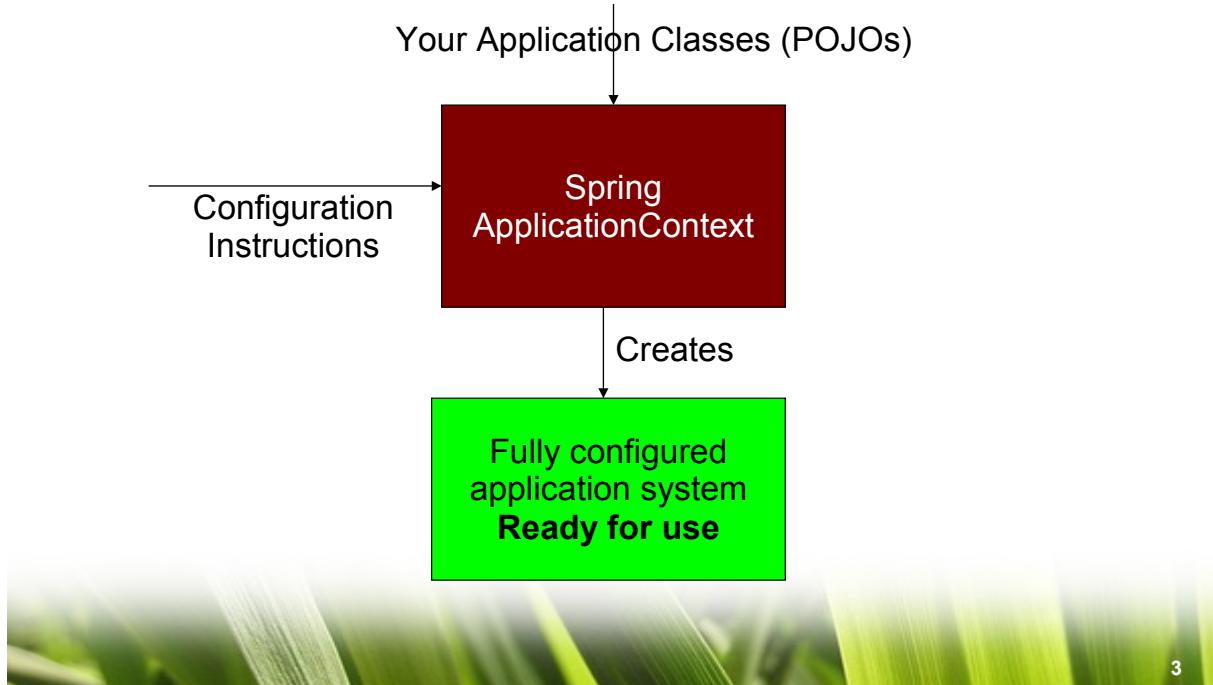
Topics in this session



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



How Spring Works



3

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

4

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>
```

5

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");

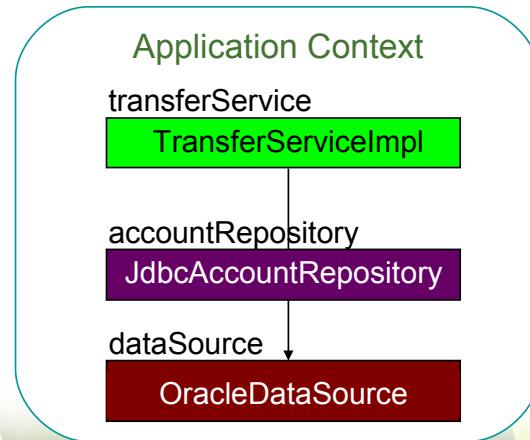
// Use the application
service.transfer(new MonetaryAmount("300.00"), "1", "2");
```

6

Inside the Spring Application Context



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



7

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
- Spring encapsulates the bean implementations chosen for a given application deployment
 - Conceals implementation details

8

Topics in this session



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



9

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-2.5.xsd">

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

</beans>
```



10

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>
```

A class with a single dependency expected by its constructor

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

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>

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

A class with a single dependency expected by a setter method

Equivalent to:

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

ApplicationContext

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

13

When to Use Constructors vs. Setters?



- Spring supports both
- You can mix and match

14

- 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
- Follow JavaBean™ conventions
- Inherited automatically



- 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



17

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.OptionalImpl" />
```

Equivalent to:

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



18

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

19

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

ApplicationContext

service -> instance of ServiceImpl

20

Injecting lists

```
<bean id="service" class="example.ServiceImpl">
  <property name="listProperty">
    <list>
      <ref bean="xyz"/>
      <value>1234</value>
      <null/>
    </list>
  </property>
</bean>
```

Equivalent to:

```
ServiceImpl service = new ServiceImpl();
// create list with bean reference, 1234 value and null element
service.setListProperty(list);
```

ApplicationContext

service -> instance of ServiceImpl

21

Injecting collections

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

22

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



23

Calling Factory Methods

- Some objects cannot be created by a constructor

```
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" />
```



24

Calling Factory Beans



- Sometimes you want Spring to always delegate to Java code to create another bean

```
public class ComplexServiceFactory implements FactoryBean {  
    Object getObject() throws Exception {  
        // create complex service impl in Java code  
    }  
}
```

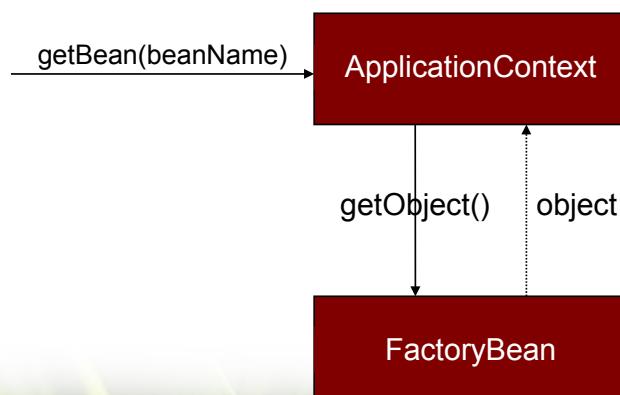
```
<bean id="service" class="example.ComplexServiceFactory">  
    <property name="neededToCreateService" value="..." />  
</bean>
```

25

The FactoryBean Contract



- Spring auto detects any bean that implements FactoryBean and returns the object created by getObject() to clients
 - The factory itself is concealed

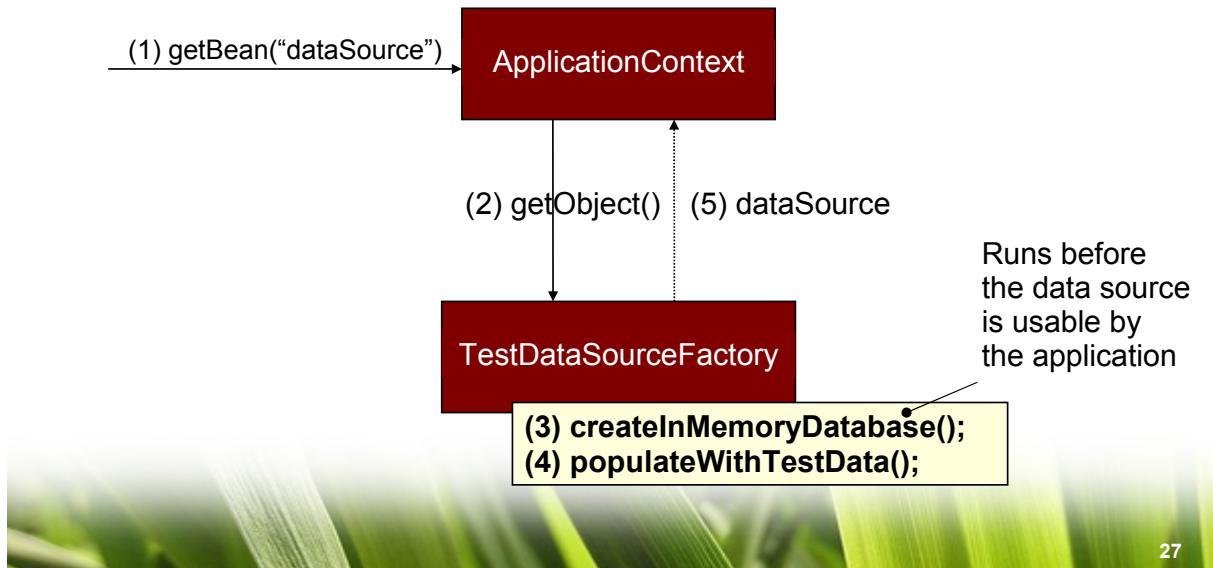


26

A Good Factory Bean Example



- Encapsulating the creation of a test DataSource



27

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

28

- 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() { ← Bootstraps the system to test  
        // 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() { ← Tests the system  
        Confirmation receipt =  
            service.transfer(new MonetaryAmount("300.00"), "1", "2"));  
        assertEquals(receipt.getNewBalance(), "500.00");  
    }  
    ...  
}
```

31

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");
```

C:\\etc\\application-config.xml

```
new XmlWebApplicationContext("WEB-INF/application-config.xml");
```

\$TOMCAT_ROOT/webapp/mywebapp/WEB-INF/application-config.xml

32

Spring's Flexible Resource Loading Mechanism (2)



- However, default rules can be overridden with *resource loading prefixes*

```
new XmlWebApplicationContext("classpath:com/acme/application-config.xml");
```

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

- These prefixes can be used *anywhere* Spring needs to deal with resources
 - Not just in constructor args to application context
- Various prefixes
 - classpath:
 - file:
 - http:

33

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

34

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

Coupled to an Oracle environment

35

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>
```

Now substitutable for other environments

36

Bootstrapping in Each Environment



- In the test environment

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

- In the production Oracle environment

```
// Create the application from the configuration
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {
        "application-config.xml",
        "oracle-infrastructure-config.xml"
});
```

37

Topics in this session



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

38

Benefits of Dependency Injection



- Your object is handed what it needs to work
 - Frees your object 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



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

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



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

3

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-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

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

Assign namespace a prefix

Associate namespace with its XSD

4

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

5

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

6

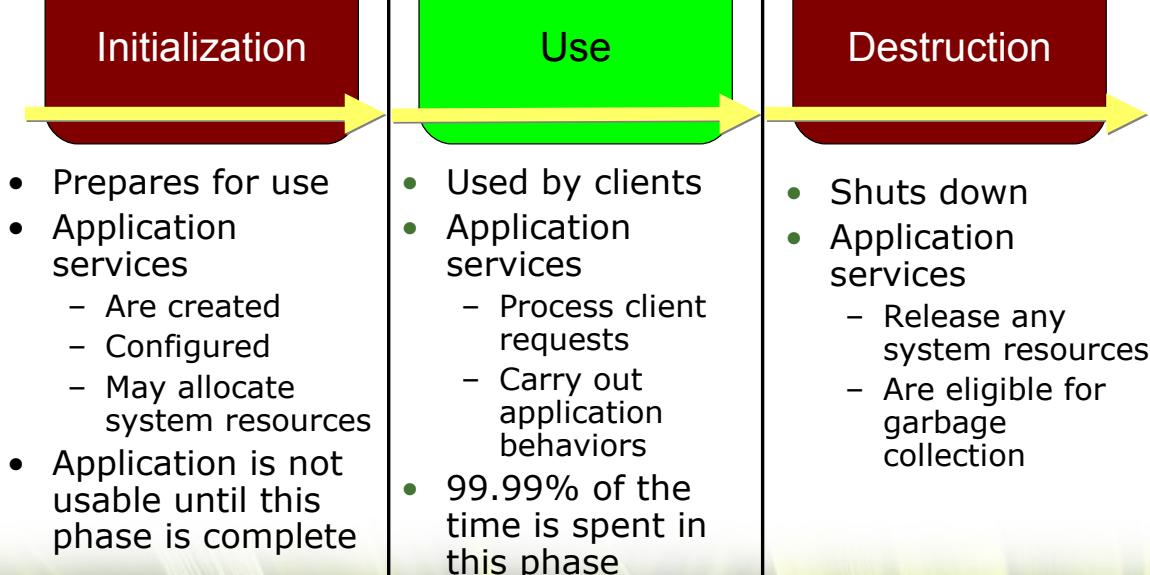
Topics in this session



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



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



- When you create a context the initialization phase completes

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

- But exactly what 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

13

Load Bean Definitions



application-config.xml

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

test-infrastructure-config.xml

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

Application Context

BeanFactory

transferService
accountRepository
dataSource

postProcess(BeanFactory)

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

BeanFactoryPostProcessors

14

- 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



15

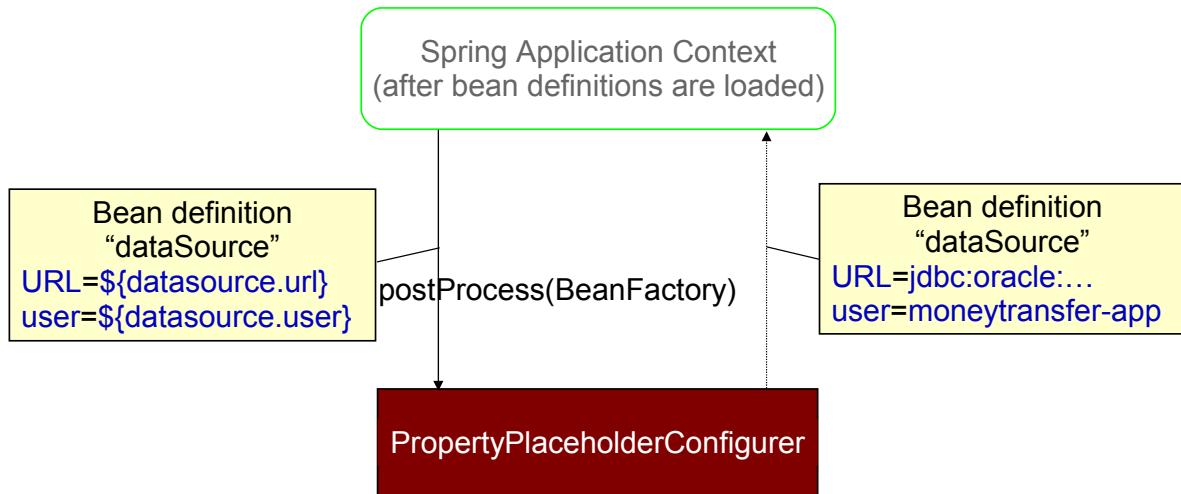
An Example BeanFactoryPostProcessor

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



16

How PropertyPlaceholderConfigurer Works



17

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

18

container-2 - 9

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

19

Inside the Application Context Initialization Lifecycle



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

20

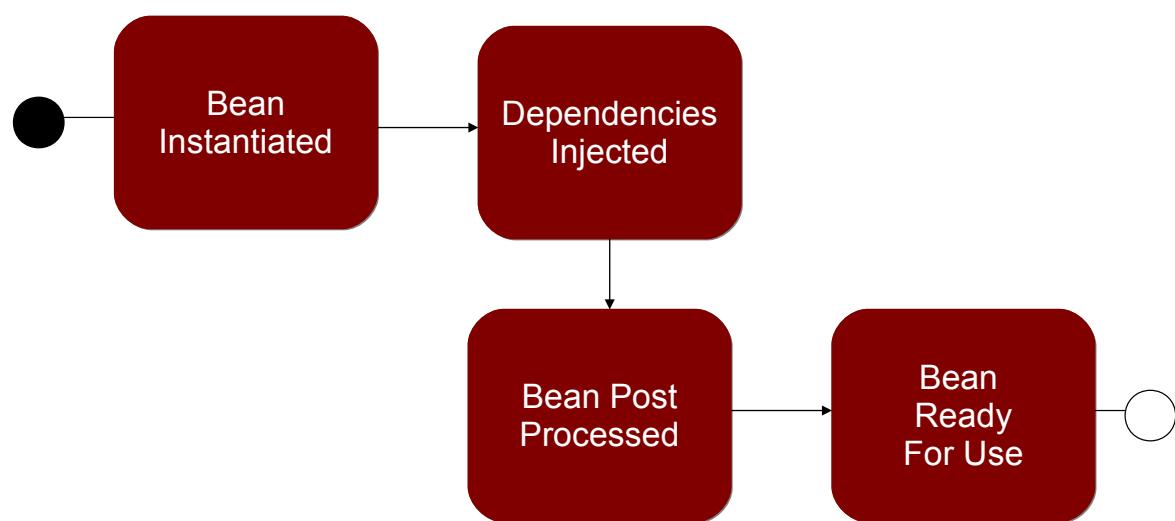
Initializing Bean Instances



- 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

21

Bean Initialization Steps



22

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



Bean Post Processing Steps

-
- **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



25

Bean Initialization Techniques



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



26

First, 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



27

JSR-250 Primer, cont.



-
- 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
 - *Which brings us back to... initialization!*
 - Spring makes use of these annotations to trigger lifecycle events



28

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

29

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

30

Simplifying Configuration with <context:annotation-config>



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

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

Enables multiple BeanPostProcessors, including:

- RequiredAnnotationBeanPostProcessor
- CommonAnnotationBeanPostProcessor



31

Bean Initialization Techniques



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



32

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>
```

Class with no Spring dependency

Method must be public with no arguments



A blurred background image showing green leaves, creating a natural and organic feel for the slide.

33

Bean Initialization Techniques



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



34

Registering for an Initialization Callback with InitializingBean



- Initialization technique used prior to Spring 2.5
 - Requires implementing a Spring-specific interface
 - Favor use of @PostConstruct if possible

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

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

35

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

36

-
- 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

39

Using RequiredAnnotationBeanPostProcessor



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

```
<beans>  
    <bean id="transferService" class="example.TransferServiceImpl" />  
  
    <bean class="org.springframework.beans.factory.annotation.  
            RequiredAnnotationBeanPostProcessor" />  
</beans>
```

Automatically detected; no other configuration is necessary

40

Simplifying Configuration with <context:annotation-config>



- Remember <context:annotation-config>?

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

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

Automatically enables RequiredAnnotationBeanPostProcessor

41

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



42

- 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");
```

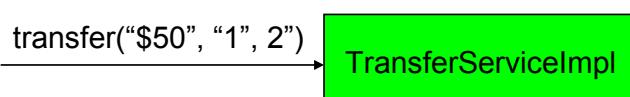
- But exactly what happens in this phase?



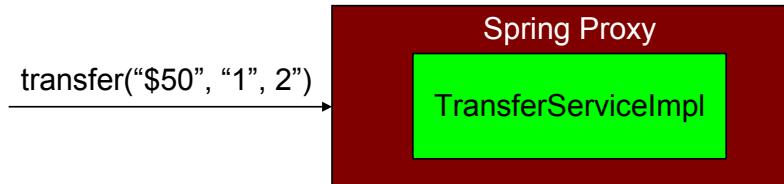
43

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

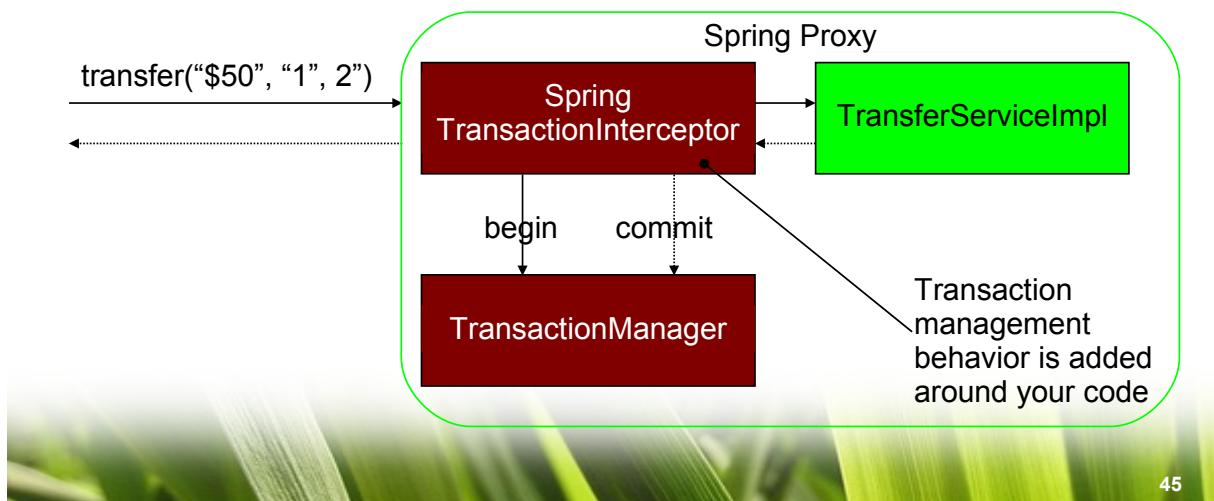


44

Proxy Power

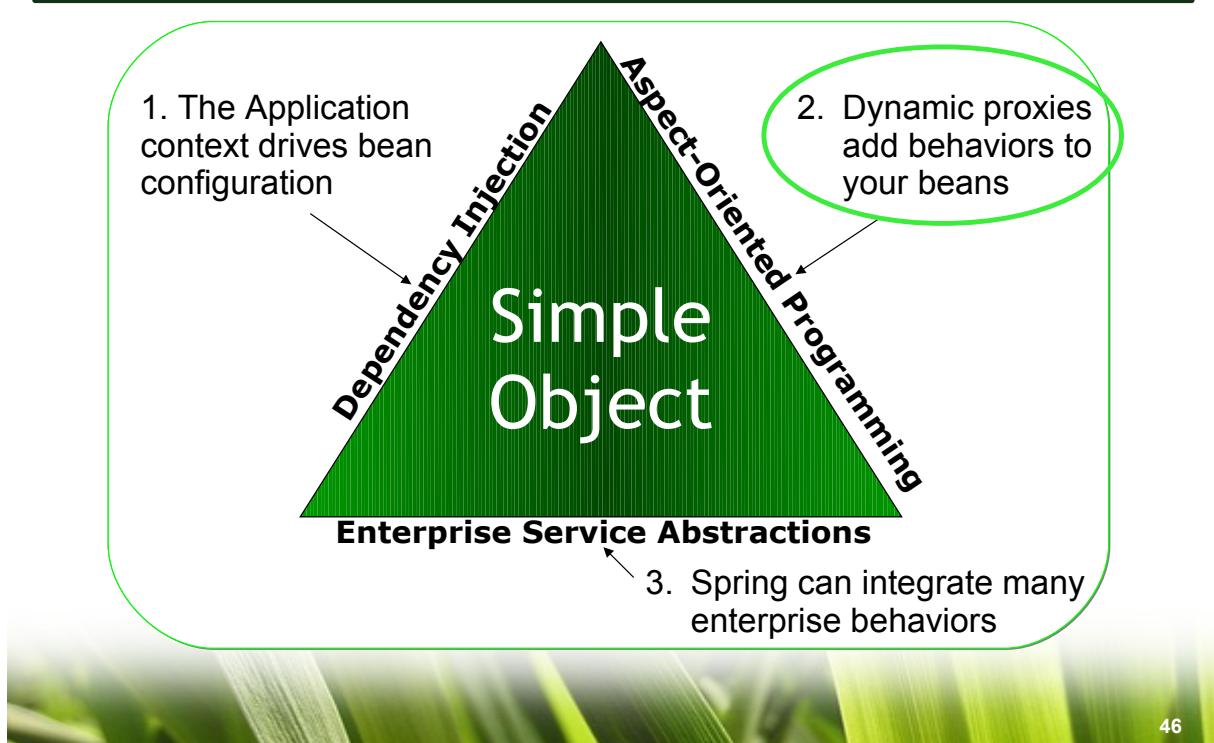


- A BeanPostProcessor may wrap your beans in a dynamic proxy and add behavior to your application logic *transparently*



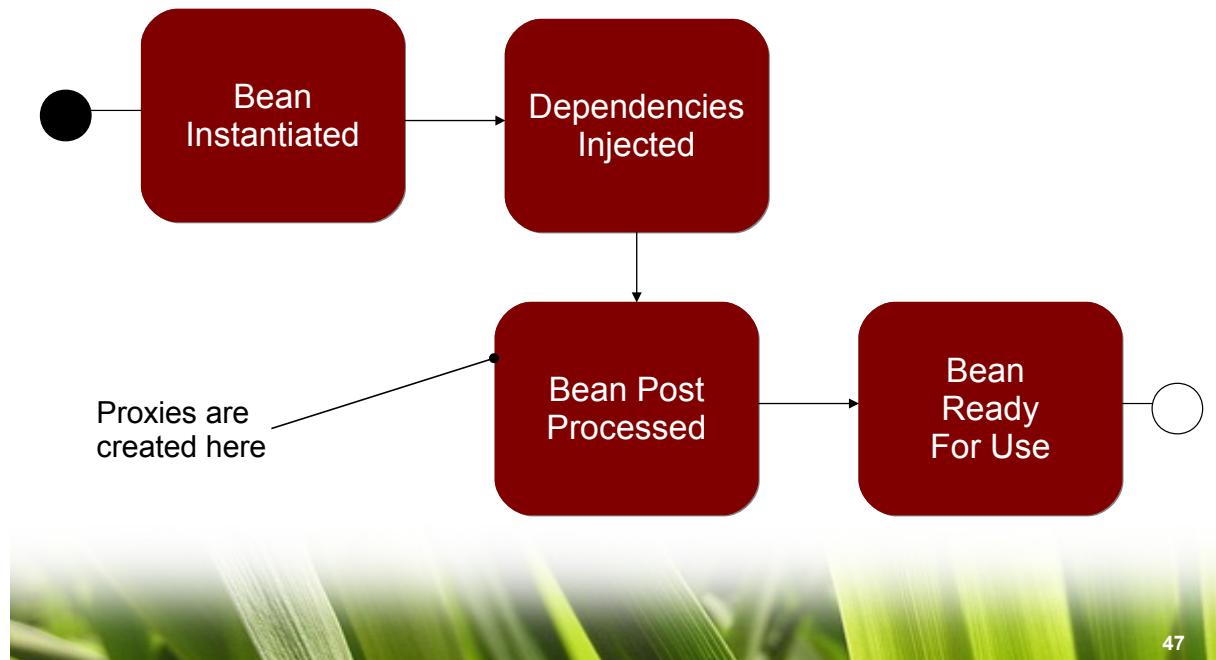
45

Proxies are Central to the Spring Triangle



46

Bean Initialization Steps



47

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



48

- 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



51

Bean Destruction Techniques



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



52

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

53

Bean Destruction Techniques



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

54

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 must be public with no arguments



55

Bean Destruction Techniques



-
- **@PreDestroy**
 - **destroy-method**
 - **DisposableBean**
- 
- 56

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



57

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**



58

- 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

59

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

60



LAB

Understanding the Bean Lifecycle



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

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



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



3

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?



4

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

5

Overriding Parent Bean Configuration



```
<beans>
  <bean id="defaultPool" 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" 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

6

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



Inner Beans

- 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)

9

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

10

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>
```

11

Topics in this session



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

12

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

13

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

14

Built-In PropertyEditor implementations



- See the `org.springframework.beans.propertyeditor` package
- Some highlights
 - NumberEditor converts to the Number types
 - BooleanEditor converts to Boolean
 - DateEditor converts to `java.util.Date`
 - ResourceEditor converts to a Resource
 - PropertiesEditor converts to `java.util.Properties`
 - ByteArrayEditor converts to a `byte[]`
 - LocaleEditor converts to a Locale

15

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

16

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

17

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

18

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



19

Importing Configuration Files (1)

- Use the import tag to encapsulate including another configuration file

application-config.xml

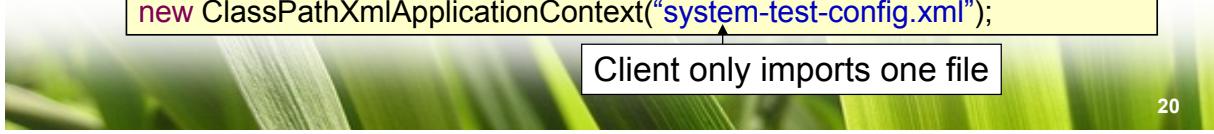
```
<beans>
    <import resource="accounts/account-config.xml" />
    <import resource="restaurant/restaurant-config.xml" />
    <import resource="rewards/rewards-config.xml" />
</beans>
```

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



20

Importing Configuration Files (2)



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

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

21



Topics in this session



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

22



- 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

23

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" .../>
```

24

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



25

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"/>
```



26

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"/>
```

27

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

28

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



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
- Before this namespace you wrote more to do the same thing
 - Using internal concepts like FieldRetrievingFactoryBean



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" .../>
```

31

Configuring a Set



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

<util:set id="subscribers"> ← Set is a top-level bean
    <value>larry@foo.com</value>
    <value>curly@foo.com</value>
    <value>moe@foo.com</value>
</util:set> ← Elements are literal values
```

32

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

33

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>
                <bean class="foo.Warehouse" .../>
            </entry>
        </util:map>
    </property>
</bean>
```

Key is an inner bean

34

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

35

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

36

- 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



37



Annotation-Based Dependency Injection

Introducing Spring's Annotations for
Dependency Injection



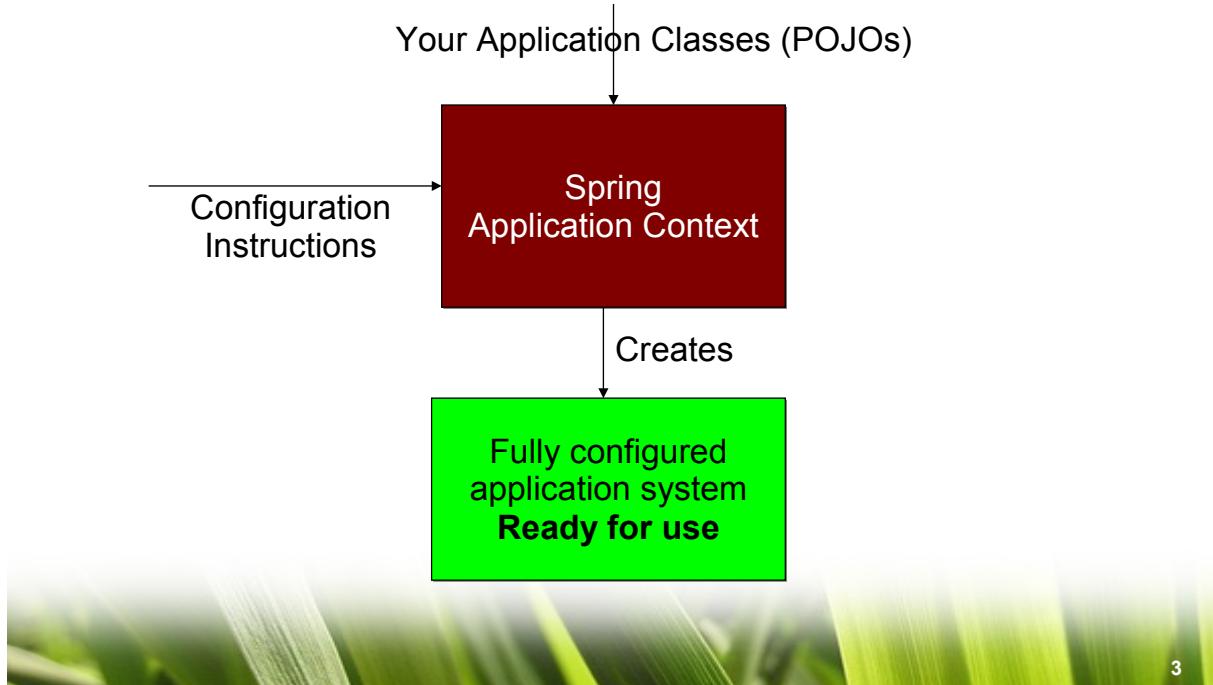
Topics in this Session



- Annotation Quick Start
- All Configuration Annotations
- Summary



How Spring Works



3

The Approach You're Used to



```
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

4

The Approach You're Used to



```
<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>
```

5

The Annotation-Based Approach



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

Spring: 'automatically try to wire this!'

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

6

container-4 - 3

The Annotation-Based Approach



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

No more need to specify constructor-args

7

Usage - Nothing Changes!



```
// 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");
```

8

- Spring container needs POJOs and wiring instructions to assemble your application
- Wiring instructions can be provided in XML...
- ...but also using annotations...
- ...or a combination of the two



9

Topics in this Session

- Annotation Quick Start
- **Configuration Annotations**
- Summary



10

- @Autowired
- @Resource
- @Component / @Repository
- @PostConstruct / @PreDestroy / @Required
 - As shown earlier



11

@Autowired

- Asks Spring to autowire fields, methods or constructors by using the type
 - Requires unique match!
- Removes the need to specify **constructor-arg** or **property** elements in XML

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



12

- Methods can have any name
- Methods and constructors can have multiple arguments that will all be injected

```
public class MyServiceImpl implements MyService
    @Autowired
    public void initialize(SomeDependency sd, OtherDependency od) {
        this.someDependency = sd;
        this.otherDependency = od;
    }
    ...
}
```

13

@Autowired Works For Collections Too!

- Just include a typed Collection to obtain a reference to all beans of a certain type

Injects all beans of type AccountRepository available in the ApplicationContext

```
public class TransferServiceImpl implements TransferService
    @Autowired
    public TransferServiceImpl(List<AccountRepository> repos) {
        this.repositories = repos;
    }
    ...
}
```

14

Optional @Autowiring



- Using the required element, specify whether or not Spring should fail if no instance was found
- Defaults to true

```
@Autowired(required=false)  
private AccountRepository accountRepository;
```

- Provides nice way of specifying defaults

```
@Autowired(required=false)  
private AccountRepository accountRepository =  
    new DefaultAccountRepository();
```

15

Disambiguation with @Autowired



- Although not often needed, @Autowired provides disambiguation features
- Needed in case multiple instances of the same type exist, one of which needs to be injected
- Using @Qualifier you can inject beans by name

Specify name of
bean to inject

```
@Autowired  
@Qualifier("primaryDataSource")  
private DataSource dataSource;
```

16

@Resource ("myBeanName")



- JSR-250 annotation (not Spring-specific)
- Injects a bean identified by the given name
- Useful when autowiring by type does not work
- Can also lookup JNDI references

```
public class TransferServiceImpl implements TransferService {  
  
    @Resource("myBackupRepository")  
    public void setBackupRepo(AccountRepository repo) {  
        this.backRepo = repo;  
    }  
}
```

17

@Resource in Spring



Some additional behavior over Java EE 5:

- By default, falls back to autowiring by type if no bean with given name exists
- Works for all *methods*, not just setters
 - But not for constructors!

18

For these Annotations to Work...



- We need to 'enable them'

```
<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-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd">
    <context:annotation-config/> ← Turns on checking of DI annotations for beans declared in this context
    <bean id="transferService" class="transfer.TransferService"/>
    <bean id="accountRepository" class="transfer.JdbcAccountRepository"/>
</beans>
```

19

@Component



- Identifies POJOs as Spring Beans
- Removes the need to specify *almost anything* in XML
- Optionally pass it a String, which will be the bean name
- Default bean name is de-capitalized non-qualified name

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

20

- @Component takes a String that names the bean
- Arguably not a best practice to put bean names in your Java code

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

21

For This Annotation To Work...

- We need to enable 'component scanning'
 - Auto-enables <context:annotation-config/>

```
<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-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

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

```

</beans>

No more need to mention the beans in the application context

22

@Scope



- @Scope allows you to customize the instantiation strategy to 'prototype', 'request', 'session' or any other (possibly custom) scope

```
@Scope("prototype") @Component
public class TransferServiceImpl implements TransferService
    @Autowired
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    ...
}
```

23

@Component Is A Meta-Annotation



- Allows introduction of custom annotation being picked up by the component scanner

```
package com.mycompany.util;

@Component
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomComponent { }
```

```
@MyCustomComponent
public class TransferServiceImpl implements TransferService
    ...
}
```

24

- Using filters, we can include or exclude beans, based on type or class-level annotation

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

25



When use what?

- Start using annotations for small isolated parts of your application (Spring @MVC controllers)
- Annotations are spread across your code base
- XML is centralized in one (or a few) places
- XML for infrastructure and more 'static' beans
- Annotations for frequently changing beans
- Spring IDE does not (yet) fully support @Autowired and @Component

26



- Annotation Quick Start
- All Configuration Annotations
- **Summary**



Summary

- Spring's configuration directives can be written in XML or using annotations
- You can mix and match XML and annotations as you please
- @Autowired and @Component allow for almost empty 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



5

TDD and Design



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



6

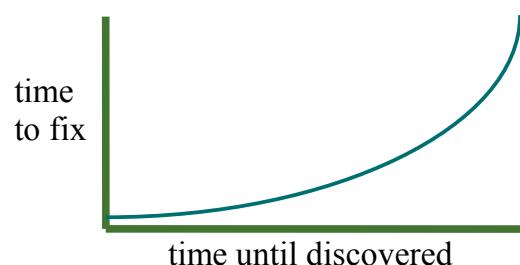
- A test case helps you focus on what matters
- It helps you not to write code that you don't need
- Find problems early



7

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



8

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



- 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

11

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)

12

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

Annotations and code snippets are highlighted in purple and blue respectively. Callouts with arrows point from the annotations to explanatory text boxes:

- An annotation `@Before` is annotated with a callout box containing the text "Stub is configured". An arrow points from the `setUp()` method to this box.
- The `@Test` annotations for both methods are annotated with a callout box containing the text "Scenarios are exercised based on the state of stub". Arrows point from both `@Test` annotations to this box.

15

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

16

Topics in this session



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



17

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



18

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

19

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

20

- 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



21

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>



22

Topics in this session



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



23

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



24

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

25

Annotations for Integration Testing

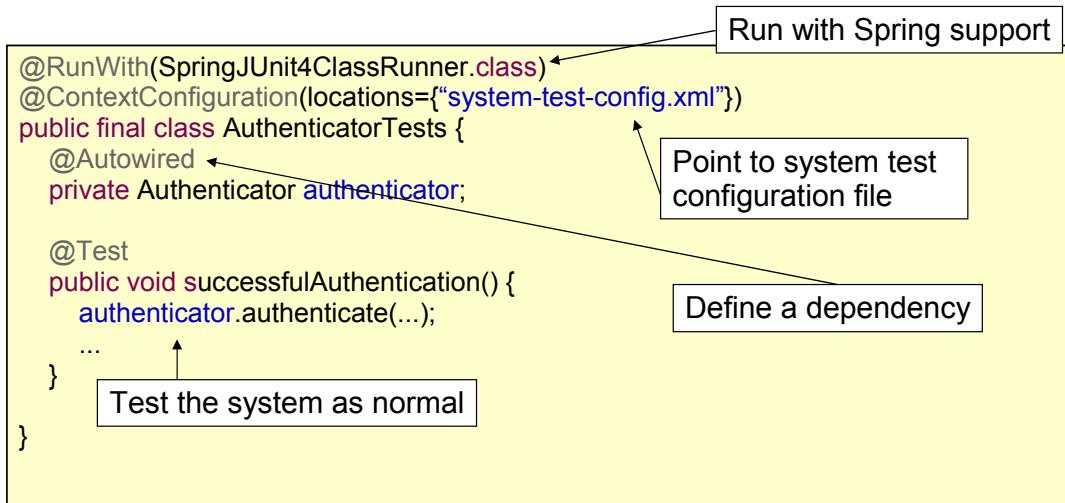


- Annotate the test with @ContextConfiguration
- Optionally pass a String-array of config locations using the locations property (by default Spring loads config.xml from the same package)
- Use @Autowired annotations as before

```
@ContextConfiguration(locations={"/transfer-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class TransferServiceTest {
    @Autowired TransferService serviceToTest;
}
```

26

Using Spring's test support



27

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

28

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

29



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



5

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



6

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);  
        ...  
    }  
}
```

Mixing of concerns

7

Symptom #2: Scattering



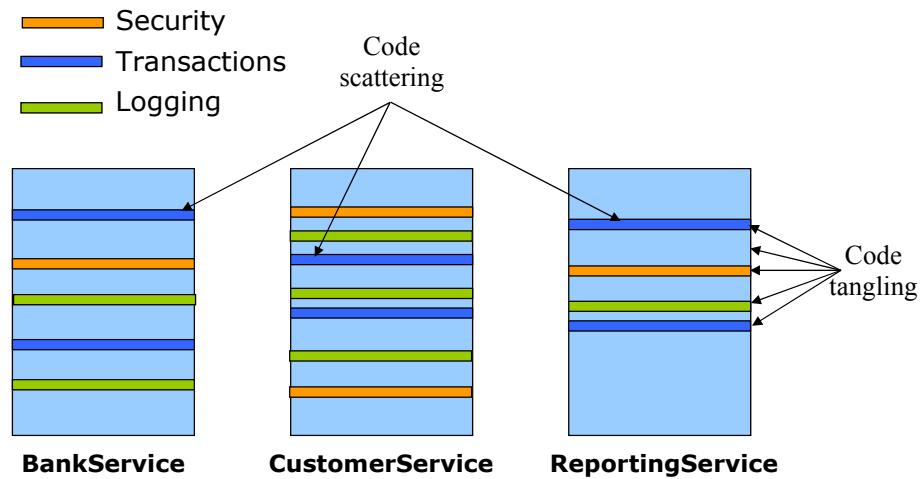
```
public class HibernateAccountManager implements AccountManager {  
    public Account getAccountForEditing(Long id) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        ...  
    }  
}
```

Duplication

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

8

System Evolution Without Modularization



9

Aspect Oriented Programming (AOP)



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

10

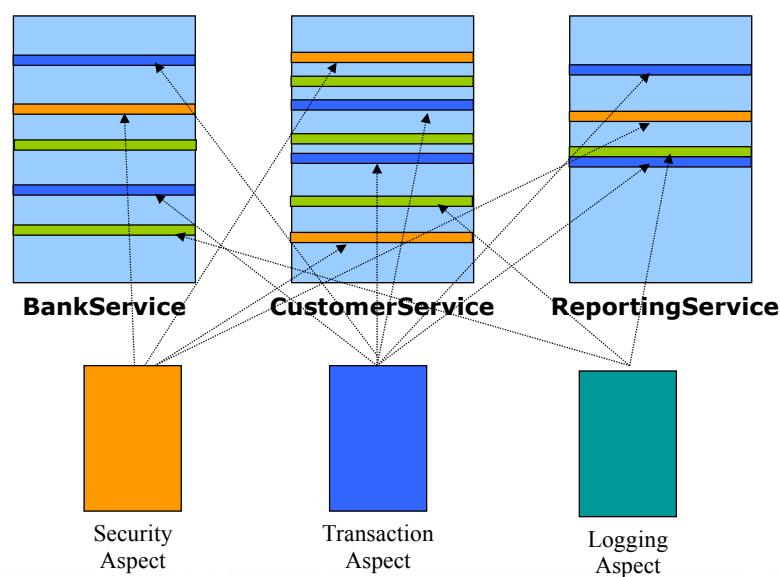
How AOP Works



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

11

System Evolution: AOP based



12

- 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



- 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



15

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



16

- 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...");  
    }  
}
```

19

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

20

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>
```

A blurred background image showing green leaves and stems of a plant.

21

Test the Application



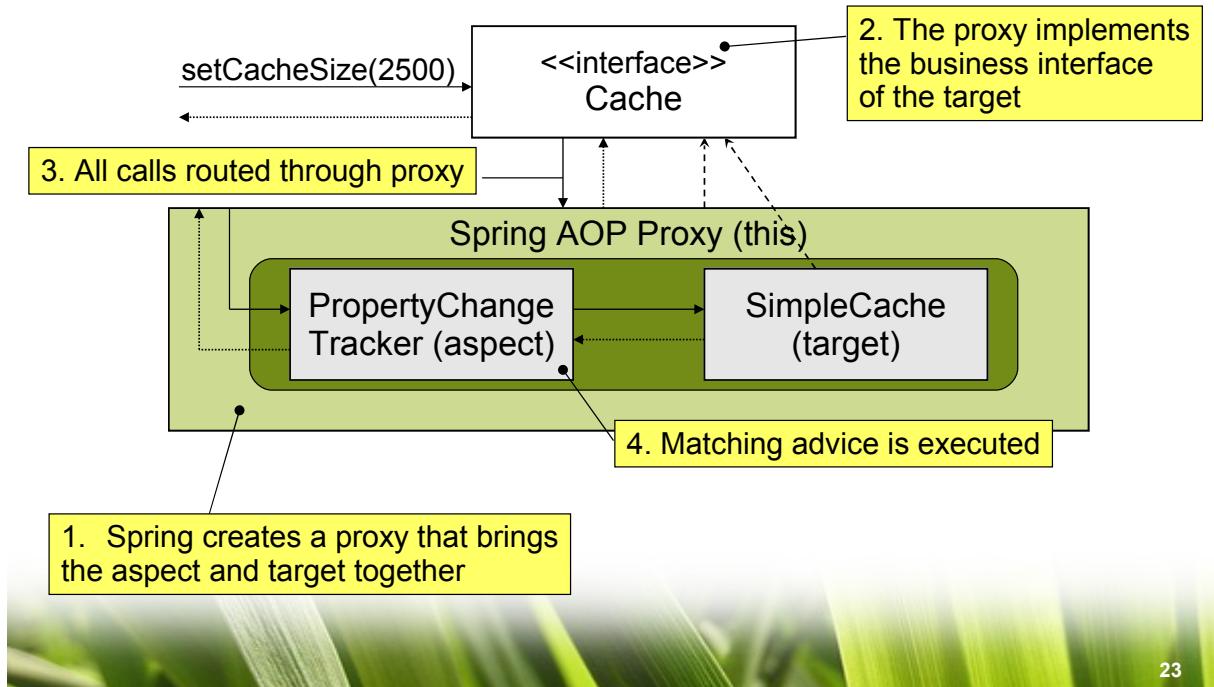
```
ApplicationContext context =
    new ClassPathXmlApplicationContext(“application-config.xml”);
Cache cache = (Cache) context.getBean(“cache-A”);
cache.setCacheSize(2500);
```

INFO: Property about to change...

A blurred background image showing green leaves and stems of a plant.

22

How Aspects are Applied



23

Tracking Property Changes – With Context



- Context is provided by the `JoinPoint` method parameter

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

Context about the intercepted point

```
@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());
```

```
}
```

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

24

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



25

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>



26

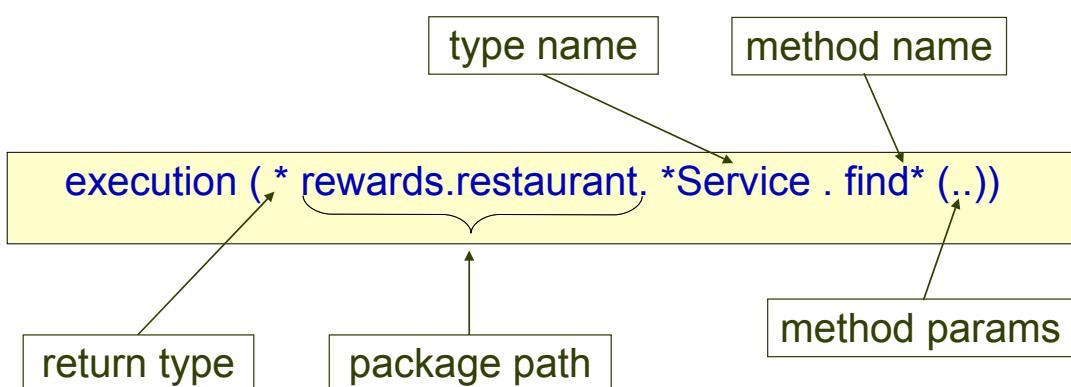
Common Pointcut Designator



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

27

Writing expressions



28

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)



29

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 takes a single parameter

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

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



30

`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*



31

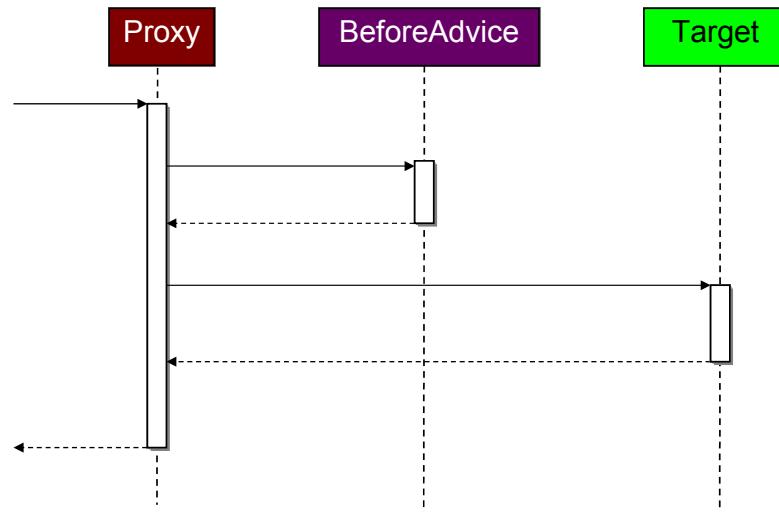
Topics in this session

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



32

Advice Types: Before



33

Before Advice Example

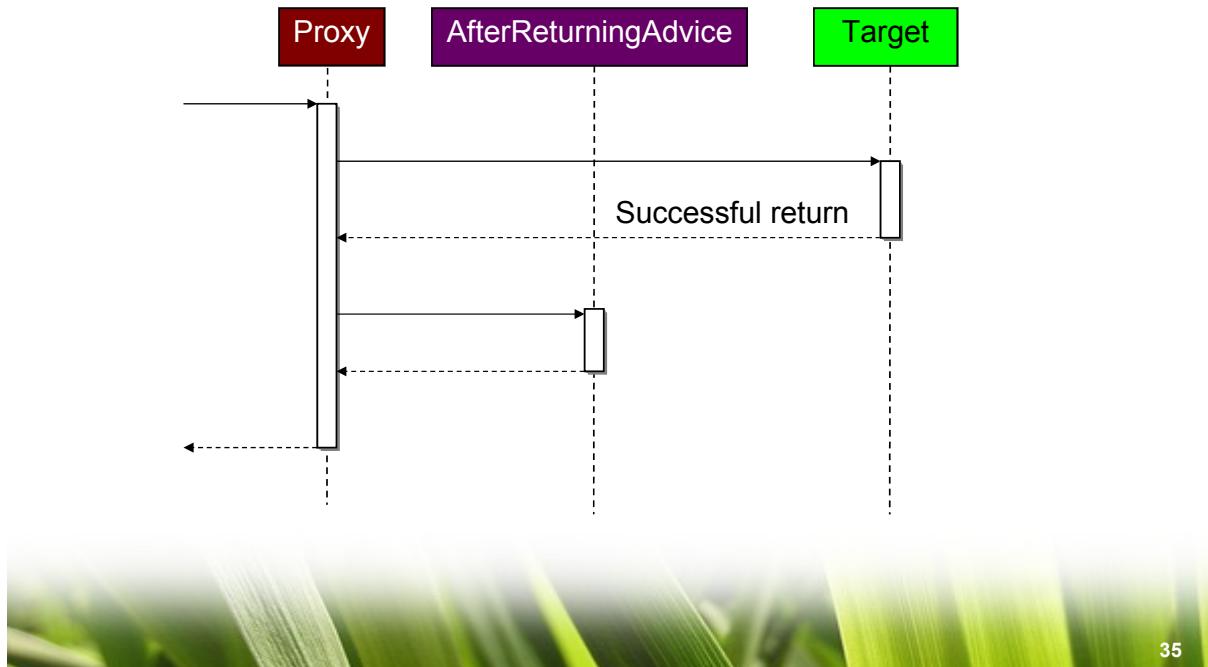


- Use `@Before` annotation
 - If the advice throws an exception, target will not be called

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...");  
    }  
}
```

34



35

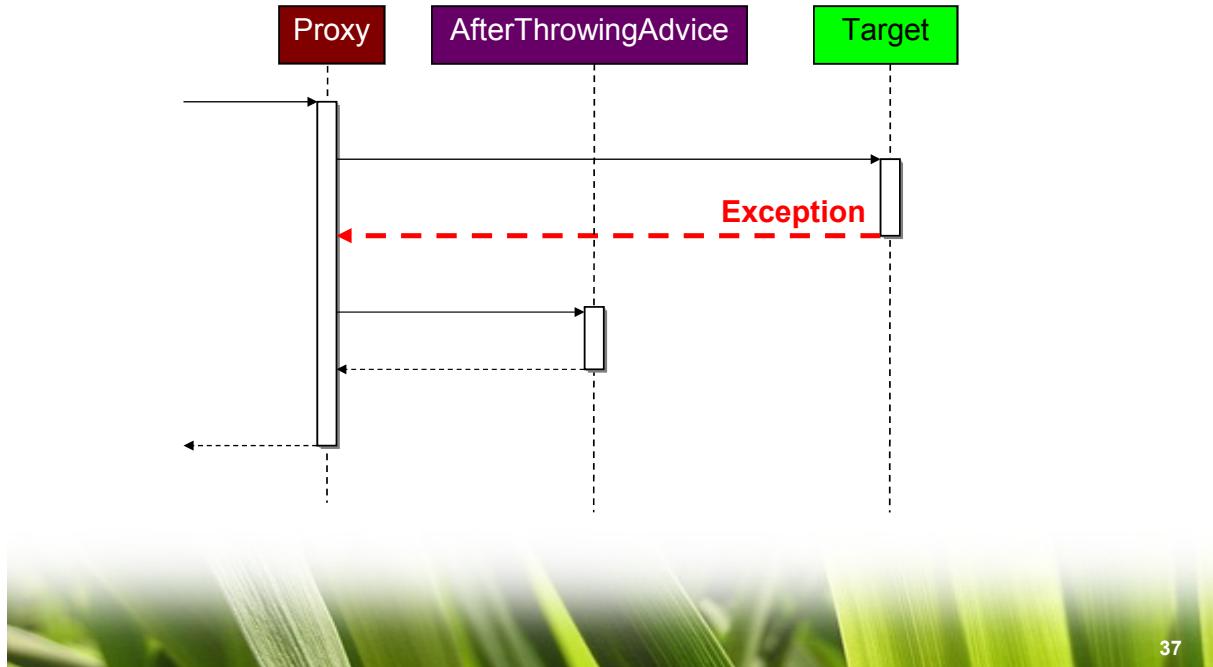
After Returning Advice - Example

- 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 ");
}
```

36



37

After Throwing Advice - Example

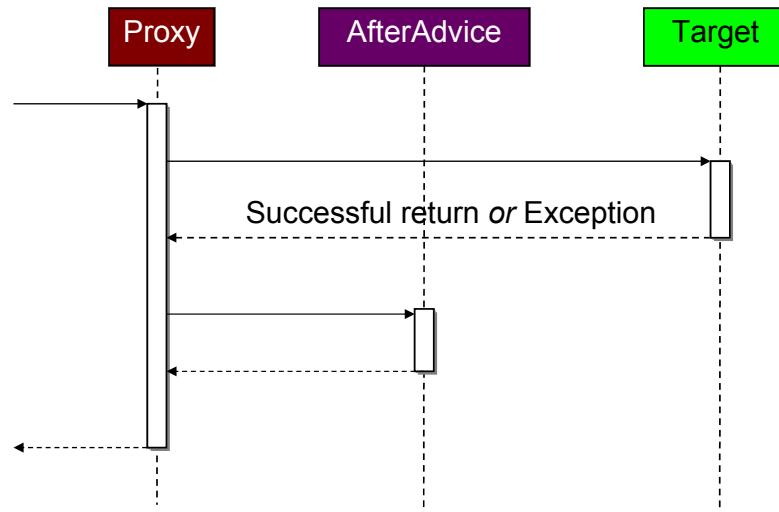
- Use `@AfterThrowing` annotation with the `throwing` attribute

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);
}
```

38

Advice Types: After



39

After Advice Example

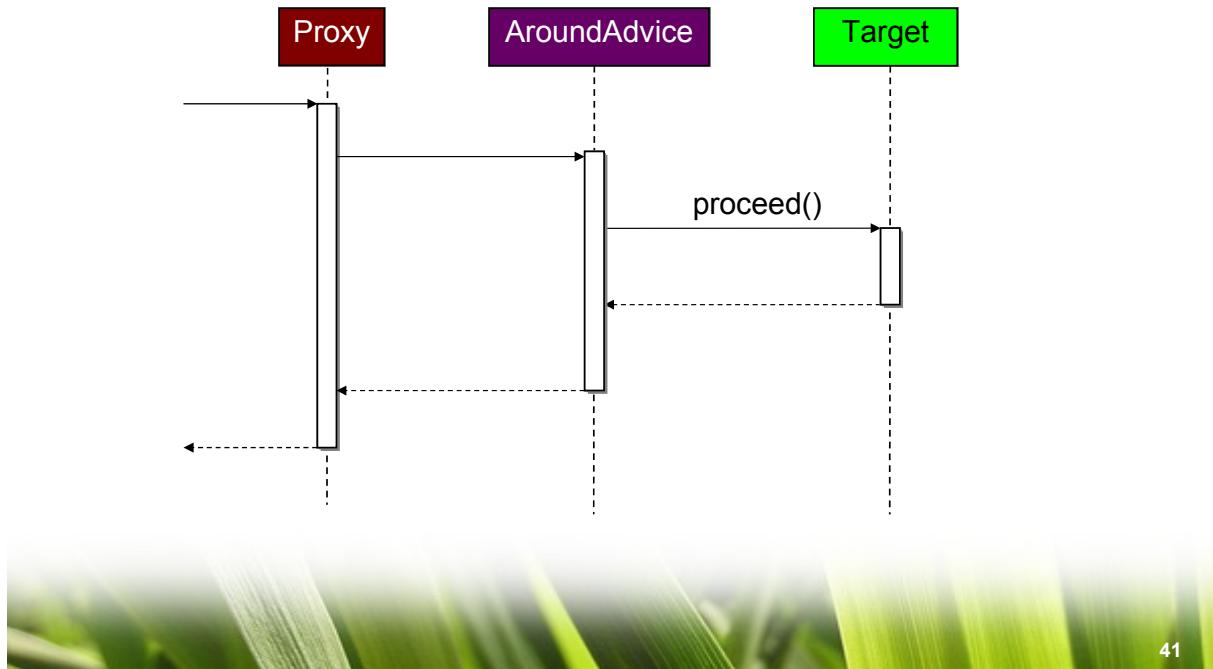


- Use `@After` annotation
 - Called regardless of whether an exception has been thrown by the target or not

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...");  
    }  
}
```

40



41

Around Advice Example

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

Cache values returned by services

```
@Around("execution(* rewards.service..*.*(..))")
public Object cache(ProceedingJoinPoint point) {
    Object value = cacheStore.get(cacheKey(point));
    if (value == null) {
        value = point.proceed();
        cacheStore.put(cacheKey(point), value);
    }
    return value;
}
```

Proceed only if not already cached

42

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
 - Easy to learn once annotation syntax is understood

43

Tracking Property Changes - Java Code



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

Aspect is a Plain Java Class with no Java 5 annotations

44

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" />
```

45



LAB

Developing Aspects with Spring AOP

- 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



47

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() {}  
}
```



48

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

49

Named Pointcuts - Summary



- Give the possibility to break one complicated expression into several sub-expressions
- Allow pointcut expression reusability
- Best practice: externalize pointcut expressions into one dedicated class

50

- 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



51

Context Selecting Pointcuts

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



52

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?

53

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];  
    ...  
}
```

54

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

55

Context Selection - Named Pointcut



```
@Before("serverStartMethod(server, input)")  
public void logServerStartup(Server server, Map input) {  
    ...  
}  
    'target' binds the server starting up  
    'args' binds the argument value
```



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

56

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

57

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

58

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 () {
    ...
}
```

59

AOP and annotations - Example



- 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());
}
```

60

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) {}
```

61

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**

62

- 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



63

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
 - The three advice types are before, after, and around
- Aspects are defined in Java with annotations or XML configuration



64



Introduction to Data Access with Spring

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



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

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



3

Making Data Access Easier



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

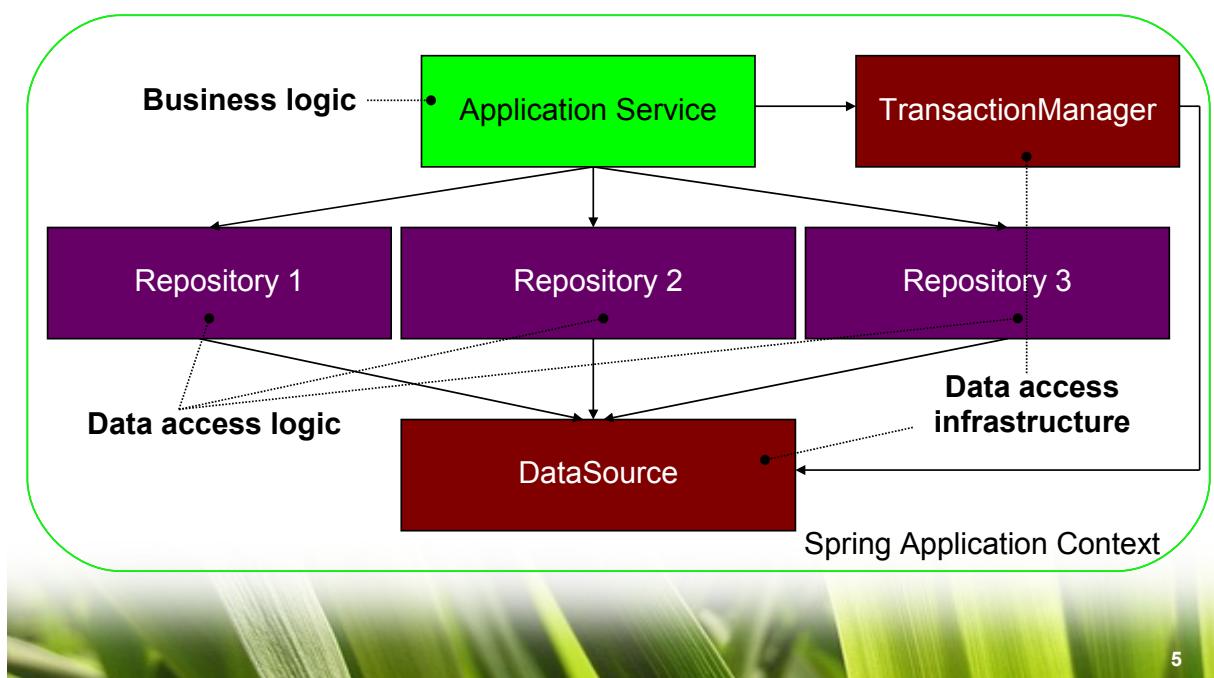
- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling any exceptions
- Release of the connection

All handled
by Spring



4

Enabling a Layered Architecture



5

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

6

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



7

The Resource Management Problem

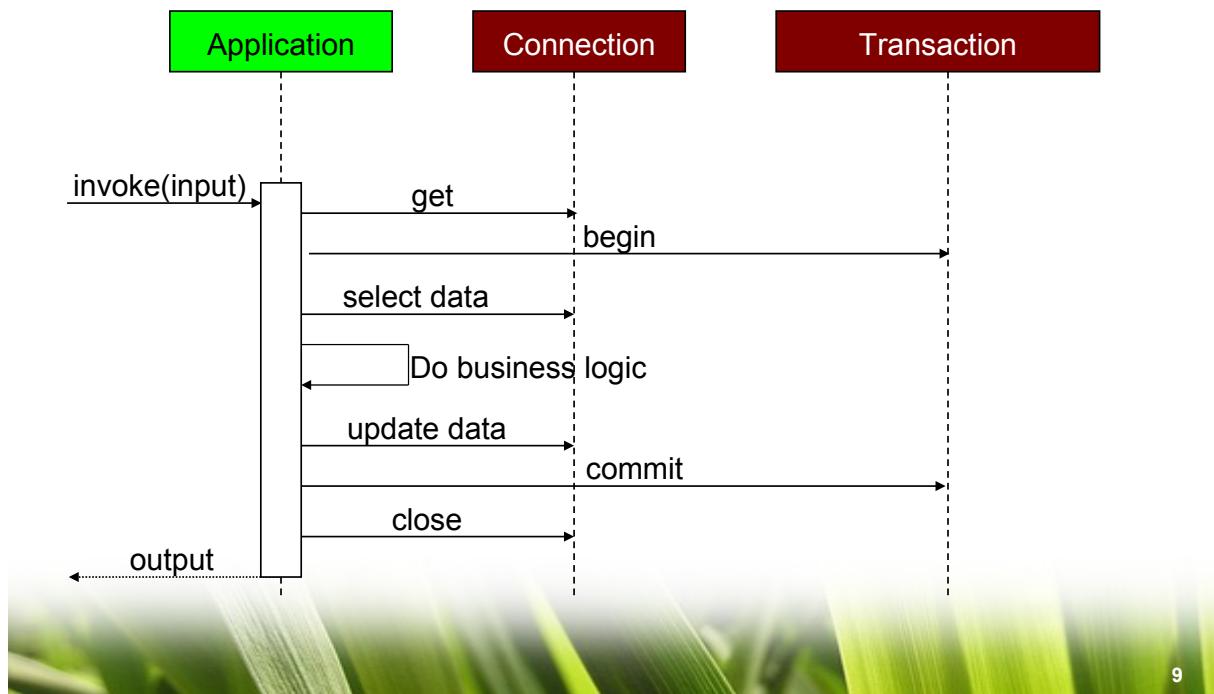


- To access a data source an application must
 - Establish a connection
- To start its work the application must
 - Begin a transaction
- When done with its work the application must
 - Commit or rollback the transaction
 - Close the connection



8

Effective Resource Management



9

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

10

Not Effective - Copy & Pasting Code



Copied around everywhere

```
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 {}  
    }  
}
```

Unique

11

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

12

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



13

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



14

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 {}  
    }  
}
```

15

Spring Resource Management



■ AFTER

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

16

Spring Resource Management Works Everywhere



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

17



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

18



- 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)
 - Java Data Objects (JDO)



Spring Data Access Support - **spring** source

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

Support class provides DAO configuration assistance

Data access helper simplifies use of API

21

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

22

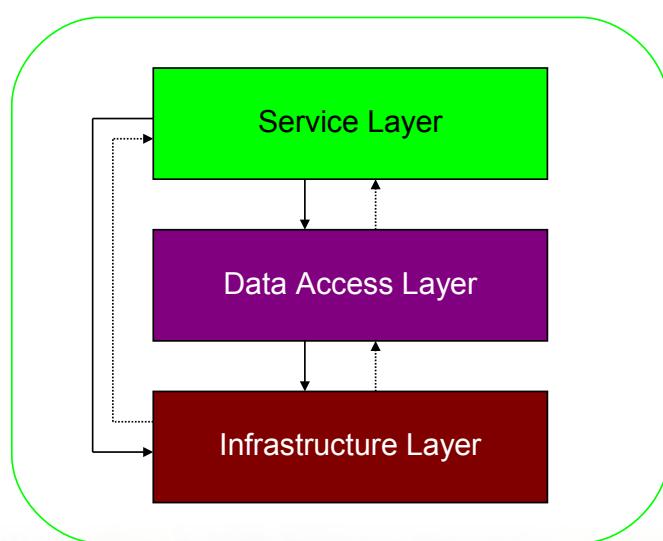
Data Access in a Layered Architecture



- 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

23

Layered Application Architecture



24

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



25

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



26

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



27

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)



28

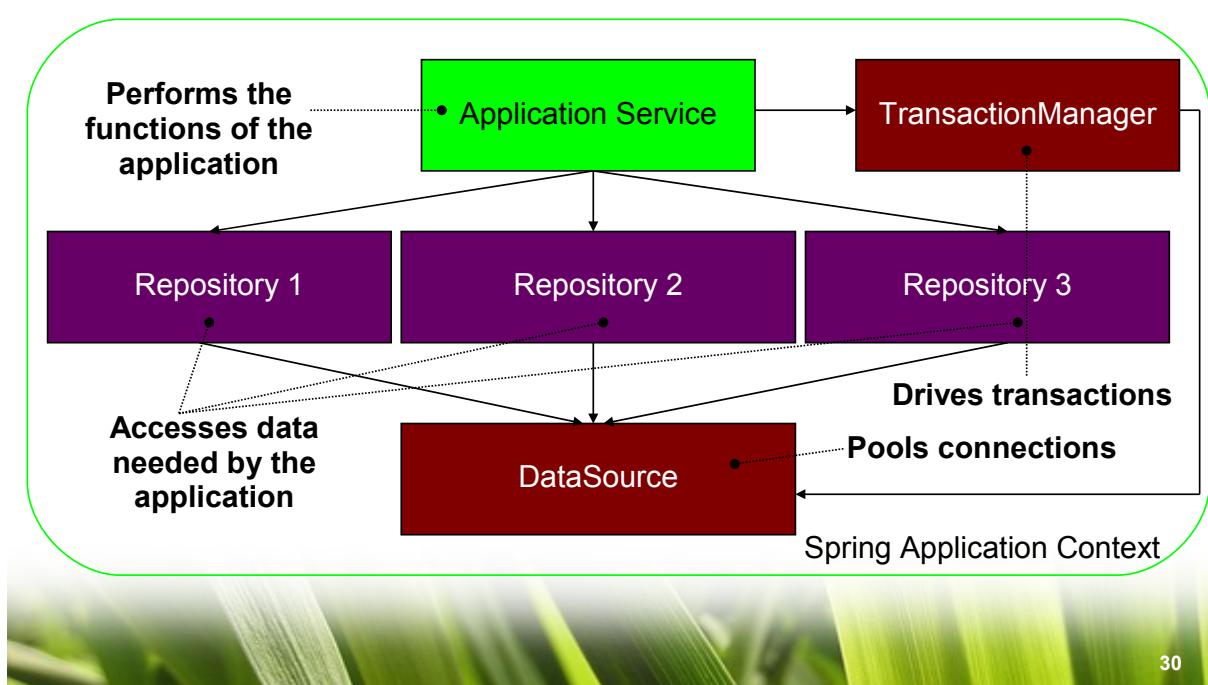
Data Access Infrastructure Services - Data Source



- 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

29

Spring Putting the Layers Together at Configuration Time



30

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



31

The Layers Working Together at Use Time (2)

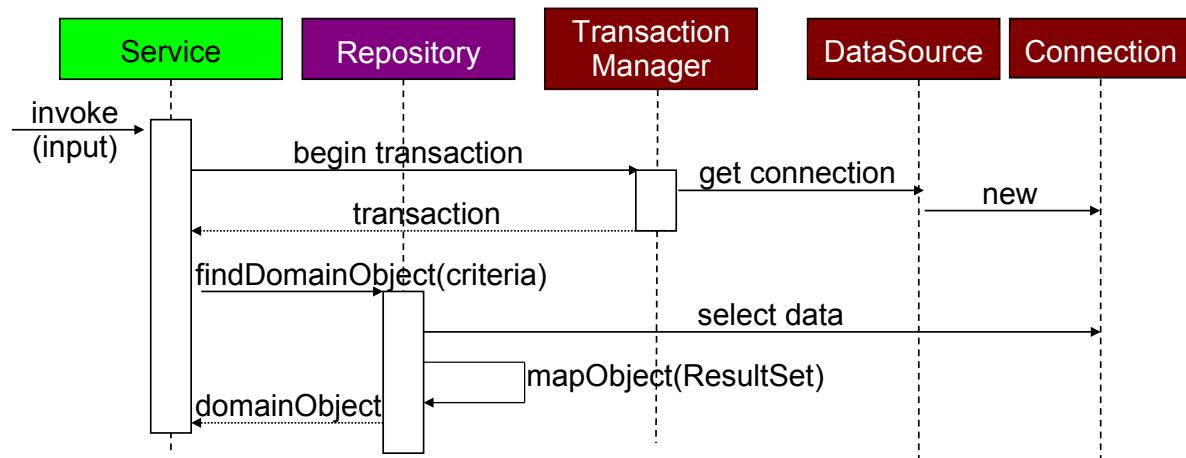


- A service continues processing
 - By executing business logic
 - Often by coordinating between domain objects loaded by repositories
- And finally, completes processing
 - By updating changed data and committing the transaction
 - Some technologies apply repository updates for you (ORMs)



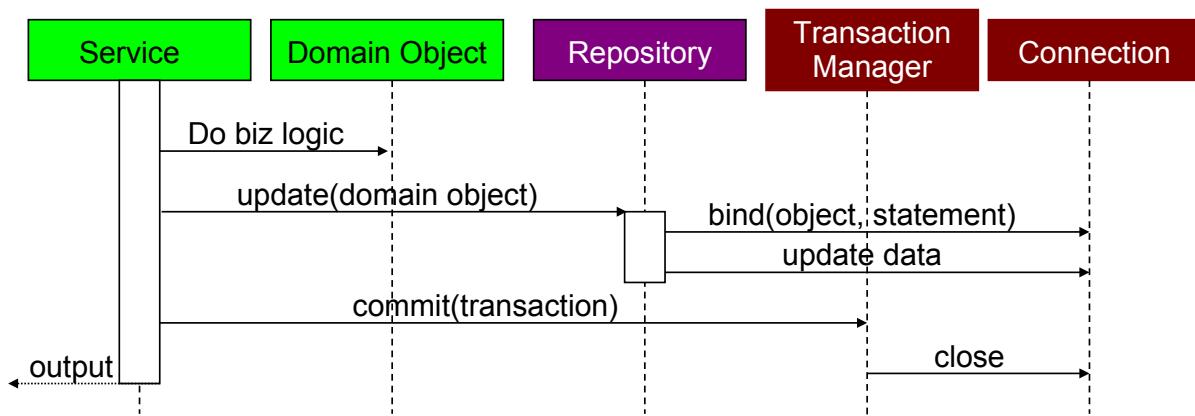
32

Application Service - Sequence (1)



33

Application Service - Sequence (2)



34

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

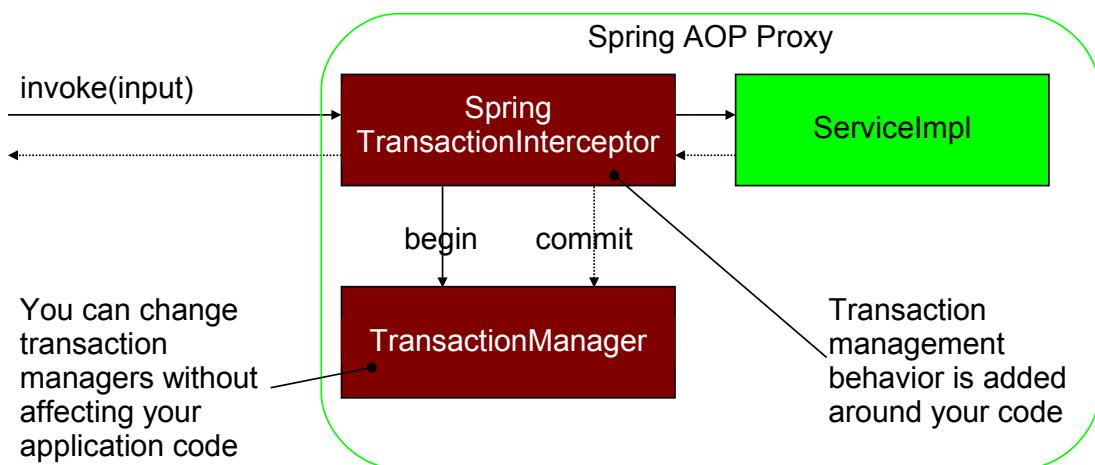


```
public class ServiceImpl implements ServiceInterface {  
    @Transactional  
    public void invoke(...) {  
        // your application logic  
    }  
}
```

Tells Spring to always run this method in a database transaction



Enforcing Declarative Transactional Policies



37

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

38

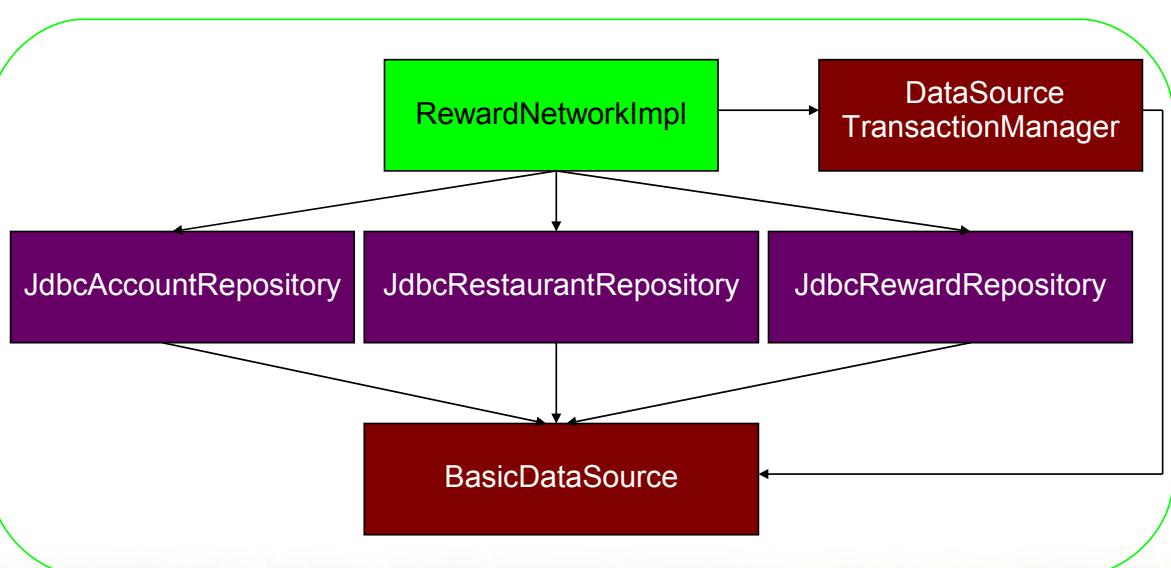
Topics in this Session



- 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**

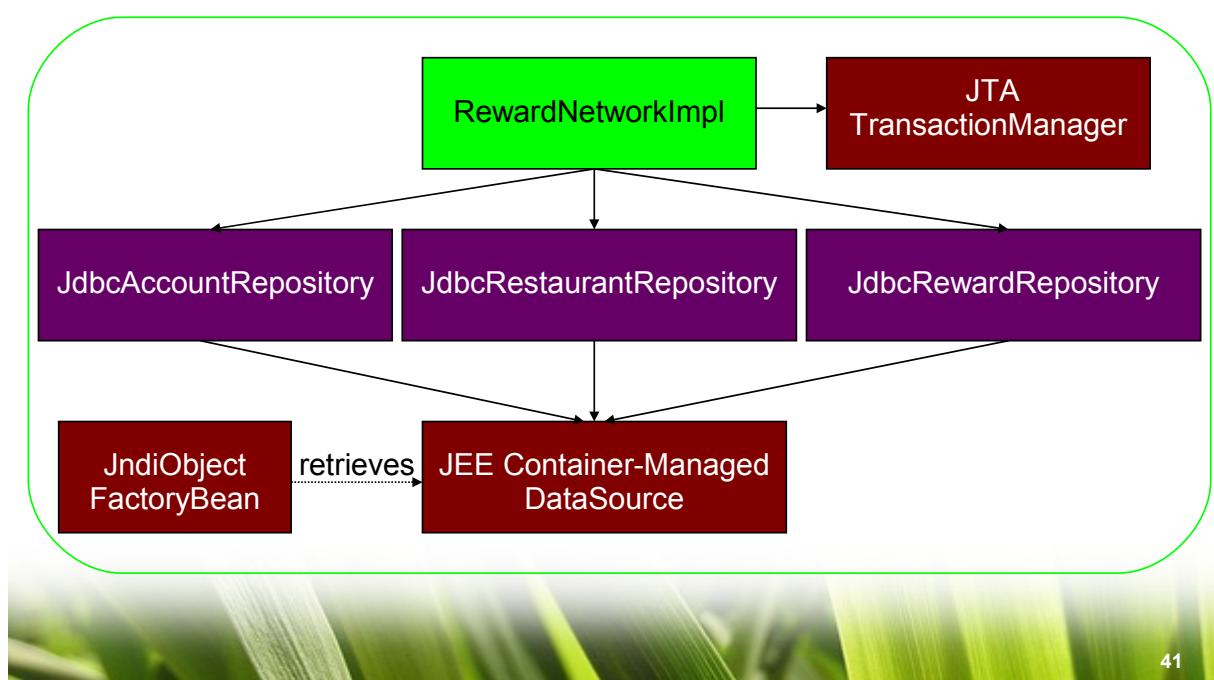
39

Local JDBC Configuration



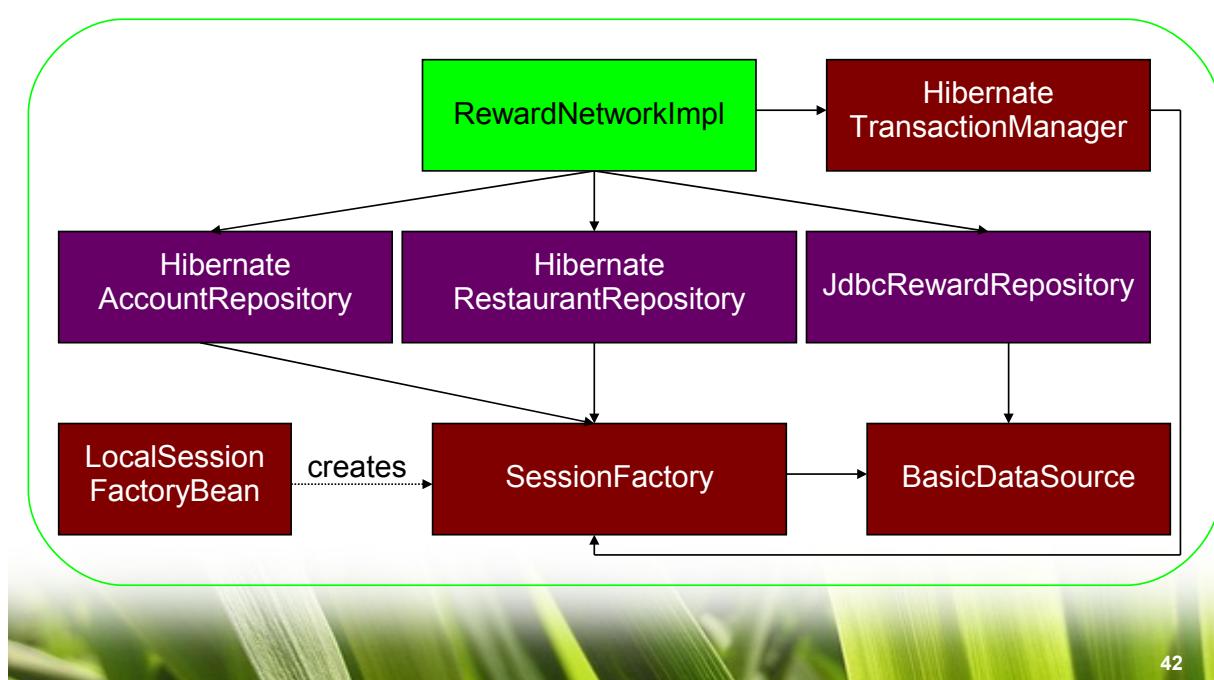
40

JDBC Java EE Configuration



41

Local Hibernate Configuration



42

Spring Data Access Support Matrix



Purpose	JDBC	Hibernate	TopLink
Technology configuration support	Several local DataSource implementations; JNDI lookup support	LocalSessionFactoryBean; JNDI lookup support	SessionFactoryBean; JNDI lookup support
DAO implementation support	JdbcDaoSupport	HibernateDaoSupport	TopLinkDaoSupport
API Helper	JdbcTemplate	HibernateTemplate	TopLinkTemplate
Transaction management	DataSource Transaction Manager or JTA	HibernateTransaction Manager or JTA	TopLinkTransaction Manager or JTA

43

Introduction to Spring JDBC

Simplifying JDBC-based data access with Spring JDBC



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

Topics in this Session

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



Redundant, Error Prone Code



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

3

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

4

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?

5

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

6

- 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(someSql,  
    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  
        }  
    }  
}
```

9

Creating a JdbcTemplate



- Requires a DataSource

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```
- For Java 5 or higher, use SimpleJdbcTemplate
 - Uses generics and varargs

```
SimpleJdbcTemplate template =  
    new SimpleJdbcTemplate(dataSource);
```
- Create a template once and re-use it
 - Do not create one for each use
 - Thread safe after construction

10

When to use (Simple)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)



11

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);  
    }  
}
```



12

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" />
```

13

Creating a JDBC-based Repository class



- Spring provides two support classes you may extend from
 - Extend JdbcDaoSupport on Java 1.4
 - Extend SimpleJdbcDaoSupport on Java 5.0 or greater
- You inherit a JdbcTemplate configured by injecting a DataSource
 - Get access to the template using getJdbcTemplate()
- Code your own class using constructor-injection
 - Support class uses setter injection: less concise
 - Support class does not work well with @Autowired

14

- SimpleJdbcTemplate is not just for compatibility with Java 5 and above
 - Some infrequently used functionality has also been removed
- If you need more control, use `getJdbcOperations()`
 - gets the underlying JdbcTemplate
- Stick to using SimpleJdbcTemplate where possible
- Note: In Spring 3.0 JdbcTemplate uses Java 5 as well
 - Less need for the SimpleJdbcTemplate

15



Querying with JdbcTemplate

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

16



Querying for Simple Java Types (1)



- Query with no bind variables

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



17

Querying for Simple Java Types (2)



- Query with a bind variable on Java 1.4

```
private JdbcTemplate jdbcTemplate;  
  
public int getCountOfPersonsOlderThan(int age) {  
    return jdbcTemplate.queryForInt(  
        "select count(*) from PERSON where age > ?",
        new Object[] { new Integer(age) });  
}
```



18

Querying With SimpleJdbcTemplate



- Query with a bind variable on Java 5 or greater
 - Java 5 supports a variable argument list

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

No need for `new Object[] { new Integer(age) }`

19

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

20

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

21

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 java.util.List of Map of [Column Name | Field Value] pairs

22

- 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



23

Querying for Domain Objects (1)

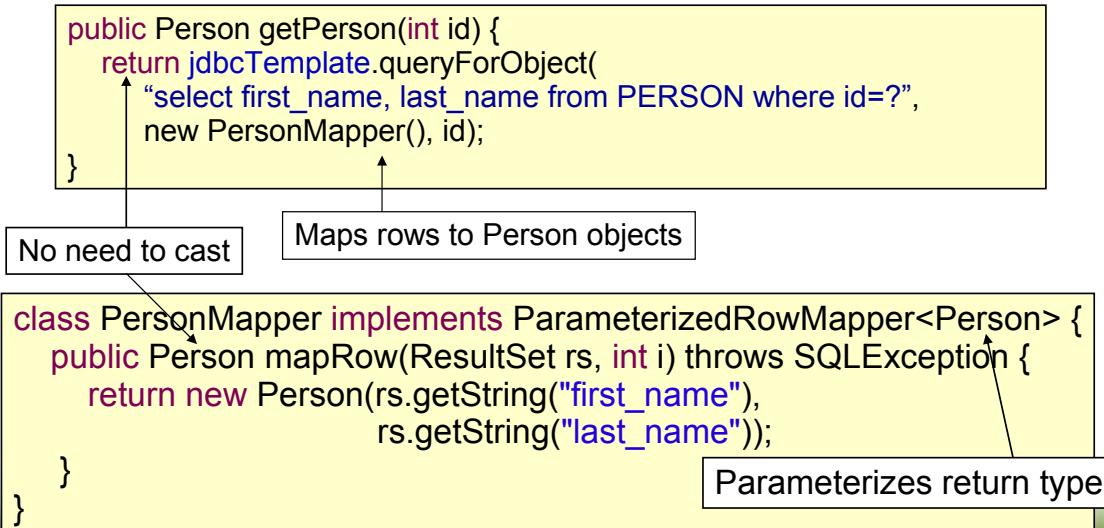
- Query for single row with SimpleJdbcTemplate

```
public Person getPerson(int id) {  
    return jdbcTemplate.queryForObject(  
        "select first_name, last_name from PERSON where id=?",  
        new PersonMapper(), id);  
}  
  
class PersonMapper implements ParameterizedRowMapper<Person> {  
    public Person mapRow(ResultSet rs, int i) throws SQLException {  
        return new Person(rs.getString("first_name"),  
                         rs.getString("last_name"));  
    }  
}
```

No need to cast

Maps rows to Person objects

Parameterizes return type



24

- Query for multiple rows

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

Same row mapper can be used

No need to cast

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

25

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
 - Only available in JdbcTemplate

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

26

Using a RowCallbackHandler



```
public class JdbcOrderRepository {  
    public void generateReport(Writer out) {  
        // select all orders for a full report  
        jdbcTemplate.query("select...from order o, item i",  
            new OrderReportWriter(out));  
    }  
}
```

```
class OrderReportWriter implements RowCallbackHandler {  
    public void processRow(ResultSet rs) throws SQLException {  
        // stream row to output  
    }  
}
```

27

ResultSetExtractor



- Spring provides a ResultSetExtractor interface for mapping an entire ResultSet to an object
 - You are responsible for iterating the ResultSet
 - Only available in JdbcTemplate

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

28

Using a ResultSetExtractor



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

```
class OrderExtractor implements ResultSetExtractor {  
    public Object extractData(ResultSet rs) throws SQLException {  
        // create an Order object from multiple rows  
    }  
}
```

29

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
- ResultSetExtractor
 - Best choice when *multiple* rows of a ResultSet map to a domain object

30

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());  
}
```

31

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());  
}
```

32

- 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



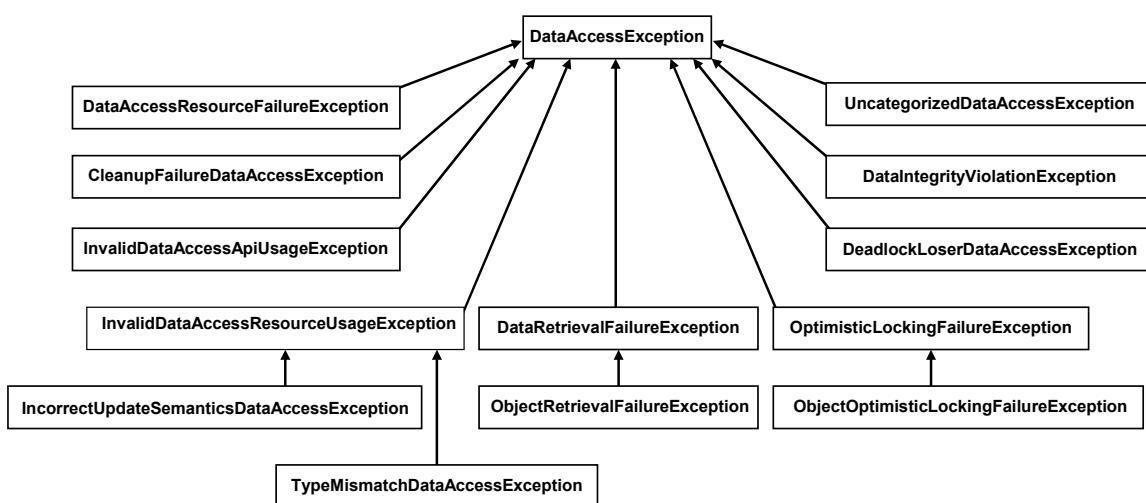
Spring SQLException Handling



- 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

35

Spring DataAccessException Hierarchy (subset)



36



LAB

Introduction to Spring JDBC



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

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
 - Programmatic transactions
 - Read-only transactions
 - Distributed transactions



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



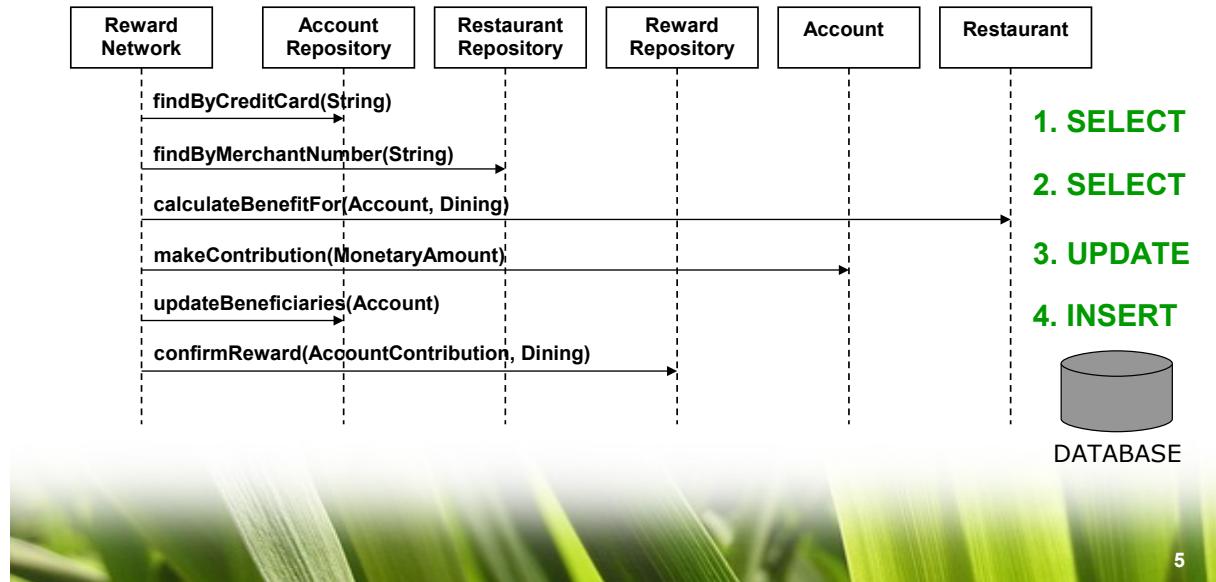
- The rewardAccountFor(Dining) method represents a unit-of-work that should be atomic



RewardNetwork Atomicity



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



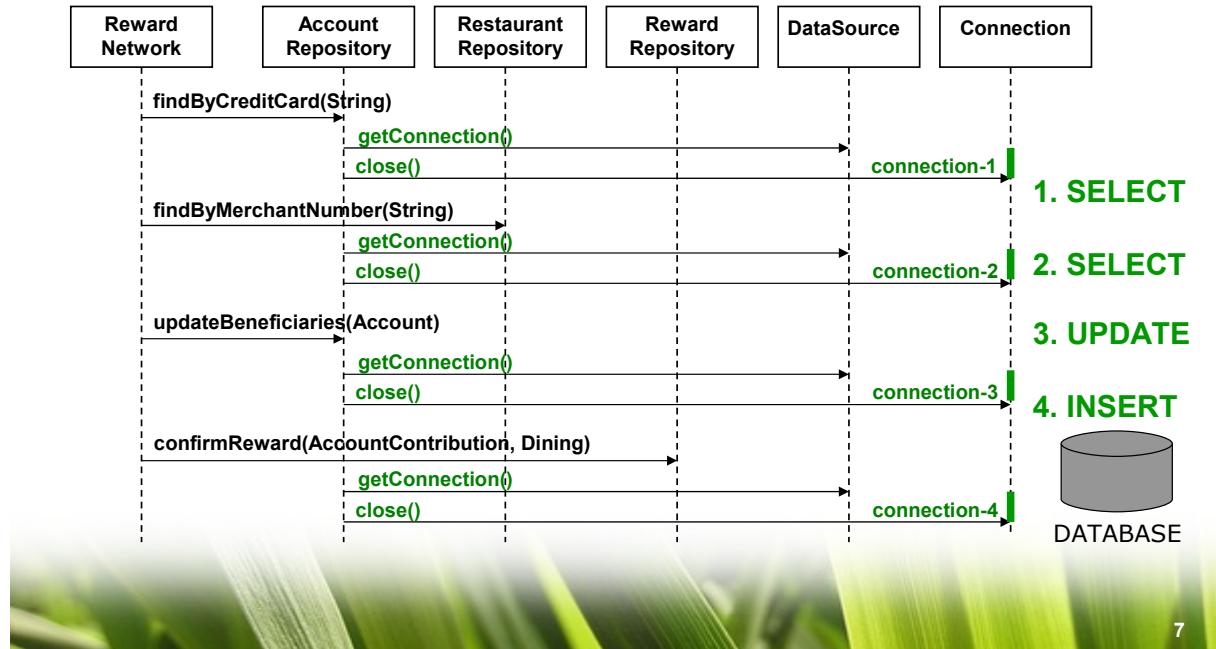
Naïve Approach:
Connection per Data Access Operation



- 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

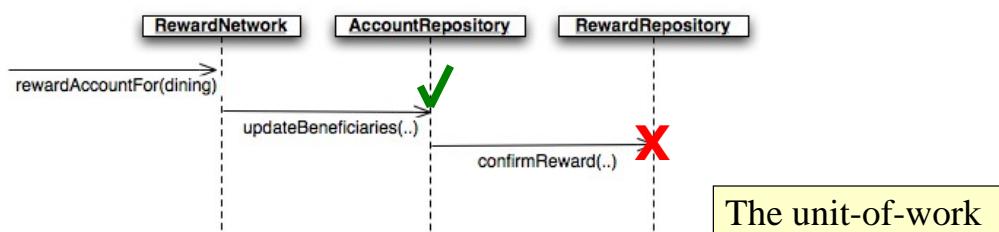


7

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!

8

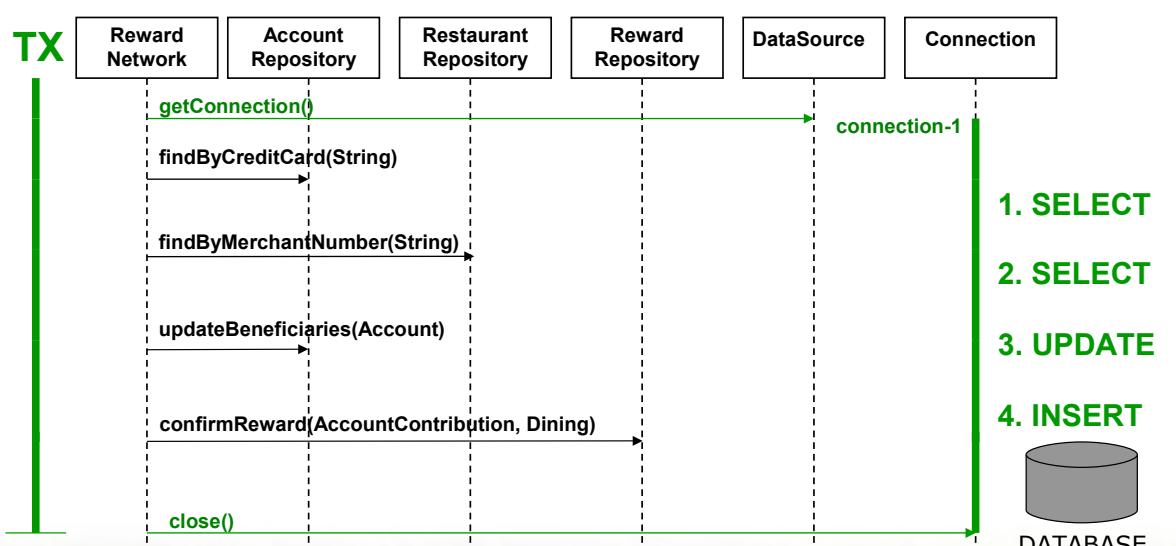
Correct Approach: Connection per Unit-of-Work



- 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***

9

Running in a Transaction



10

tx-1 - 5

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
 - Distributed transactions



11

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



12

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);  
    }  
}
```

13

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

14

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
 - Distributed transactions

15

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

16

- 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

17

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

18

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



19

@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>
```



20

@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)



21

@Transactional at the class level



- 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(RewardConfirmation rc) {  
        // atomic unit-of-work  
    }  
}
```



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



22

@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
    }
}
```



23

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



24

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(..)

25

Topics in this session



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

26

Isolation levels



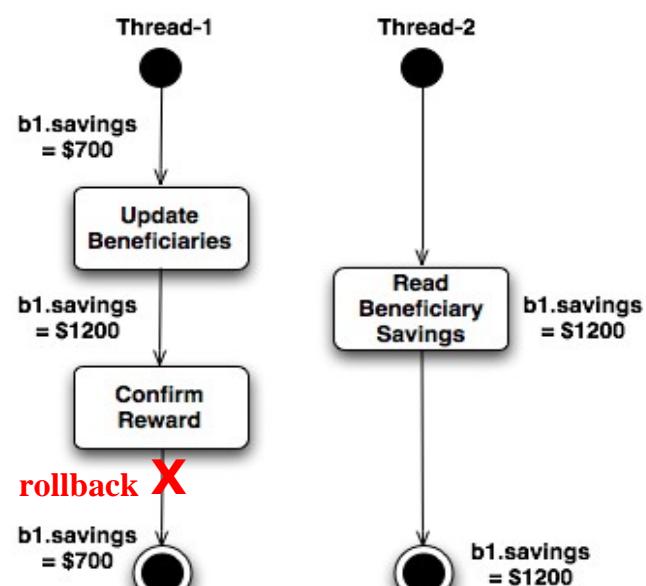
- 4 isolation levels can be used:
 - READ_UNCOMMITTED
 - READ_COMMITTED
 - REPEATABLE_READ
 - SERIALIZABLE
- Some DBMSs do not support all isolation levels
- Isolation is a complicated subject
 - DBMS all have differences in the way their isolation policies have been implemented
 - We just provide general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

27

Dirty Reads



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



28

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  
    }  
}
```

29

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  
    }  
}
```

30

- 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



31

Topics in this session

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



32

Understanding Transaction Propagation: Example

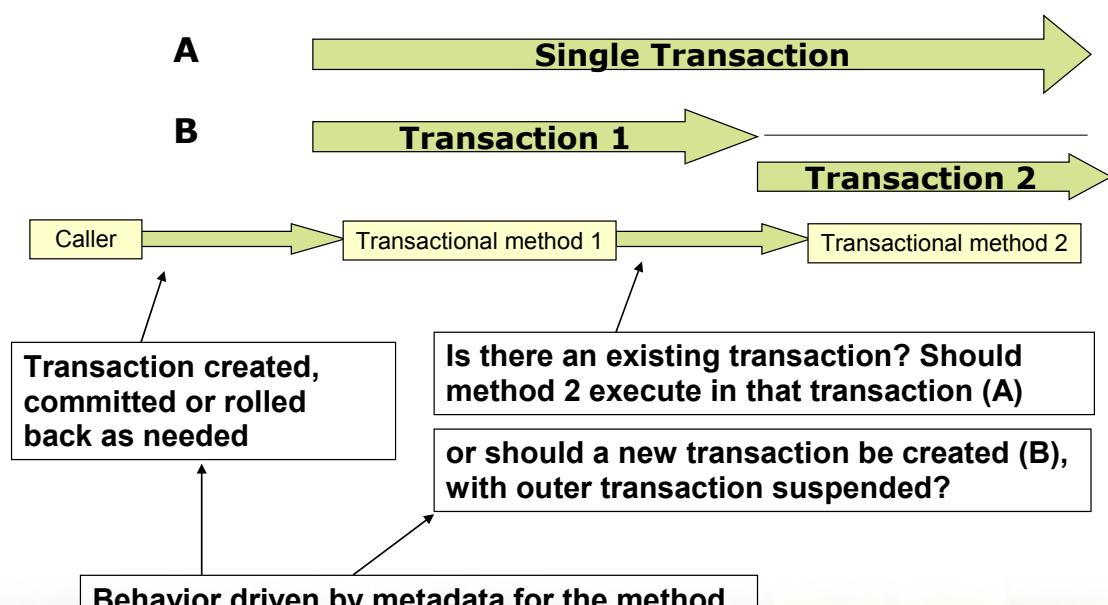


- Consider the sample below. What should happen if ClientServiceImpl calls AccountServiceImpl?
 - Should everything run into a single transaction?
 - Should each service have its own transaction?

```
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)  
    { // ...  
    }  
}
```

33

Understanding Transaction Propagation



34

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:

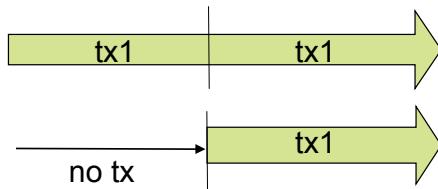
```
@Transactional(propagation=Propagation.REQUIRES_NEW)
```



REQUIRED



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



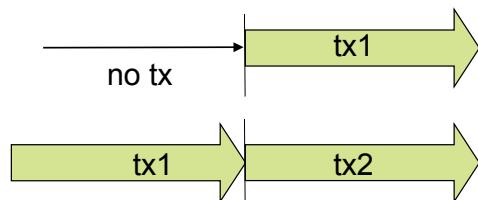
```
@Transactional(propagation=Propagation.REQUIRED)
```



REQUIRES_NEW



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



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

37

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
 - Distributed transactions

38

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

39

rollbackFor and noRollbackFor



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

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

not `RuntimeException(s)`

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

40

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
 - Distributed transactions

41

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

42

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

Transactions does a *commit* at the end by default

Overrides default *rollback* settings



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

43

@Before vs @BeforeTransaction



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

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

Run before transaction is started

Within the transaction



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

44

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**
 - **Distributed transactions**



- 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

47

Programmatic Transactions: example

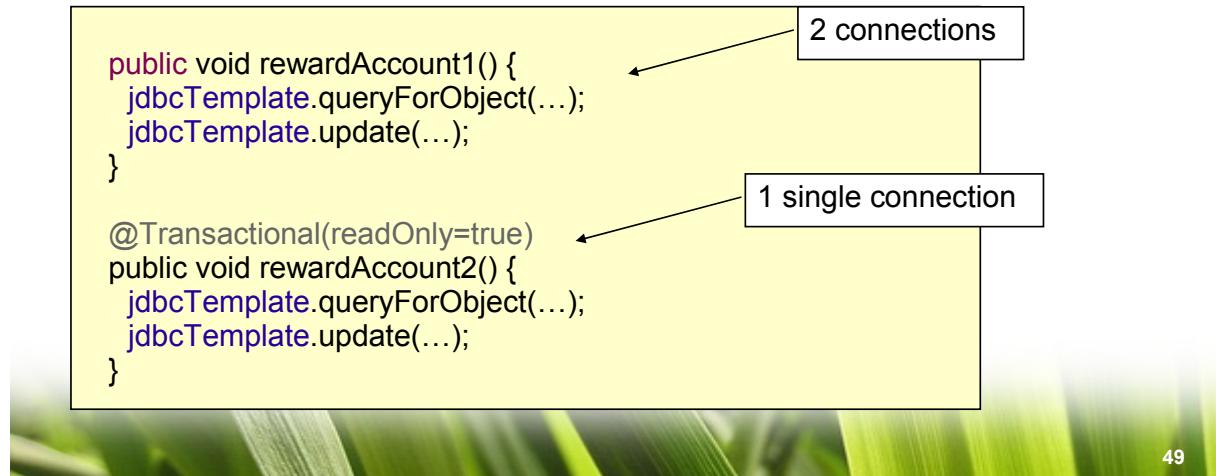
```
public RewardConfirmation rewardAccountFor(Dining dining) {  
    ...  
    txTemplate = new TransactionTemplate(txManager);  
    return txTemplate.execute(new TransactionCallback() {  
        public Object doInTransaction(TransactionStatus status) {  
            try {  
                ...  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

48

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



49

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

50

- 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 (ORM) engines exist to mitigate the mismatch
 - Spring supports all of the major ones:
 - Hibernate
 - Toplink
 - JDO
 - Java Persistence API (JPA)
 - This session will focus on Hibernate



Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

5

Topics in this session

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



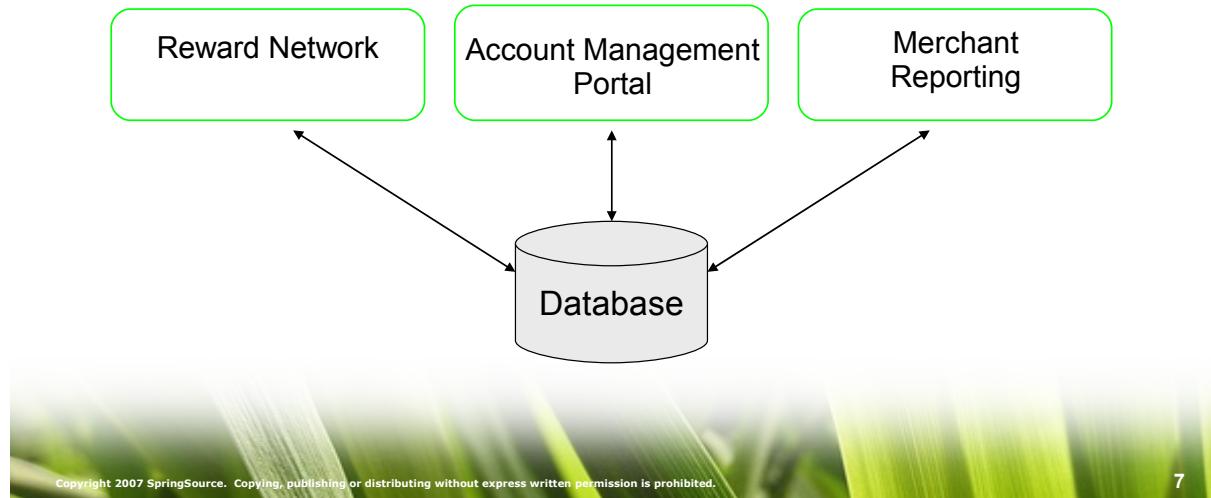
Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

6

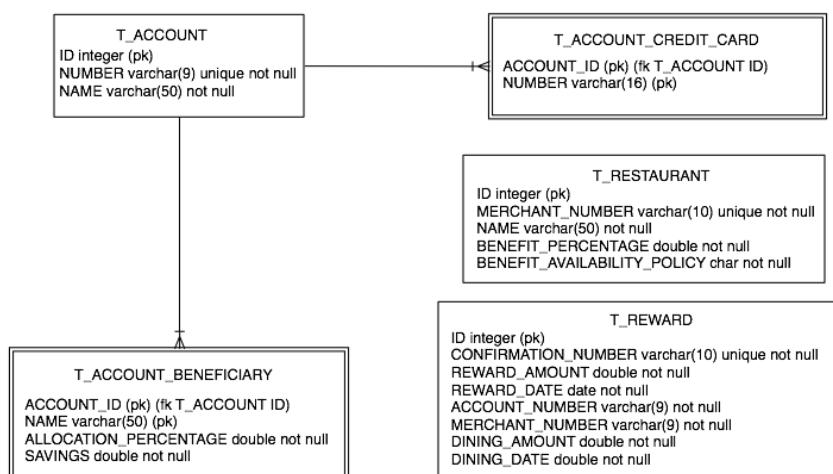
ORM in context



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



The integration database schema



Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

8

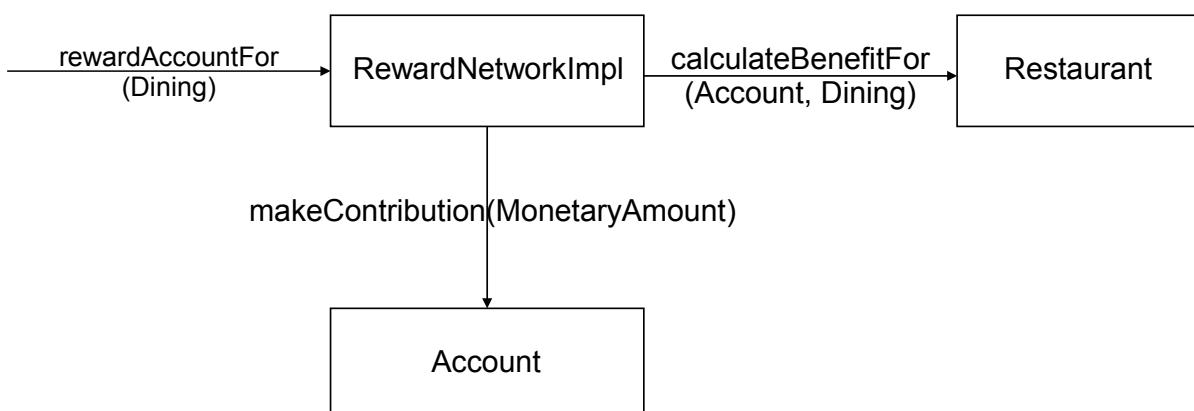
The RewardNetwork application



- 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



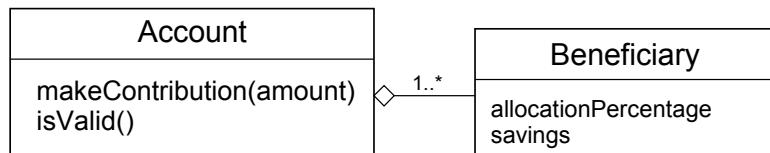
Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

11

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%



Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

12

-
- 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?



Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

13

O/R Mismatch: Granularity (1)

-
- In an object-oriented language, cohesive fine-grained classes provide encapsulation and express the domain naturally
 - In a database schema, granularity is typically driven by normalization and performance considerations



Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

14

just one example...

Domain Model in Java

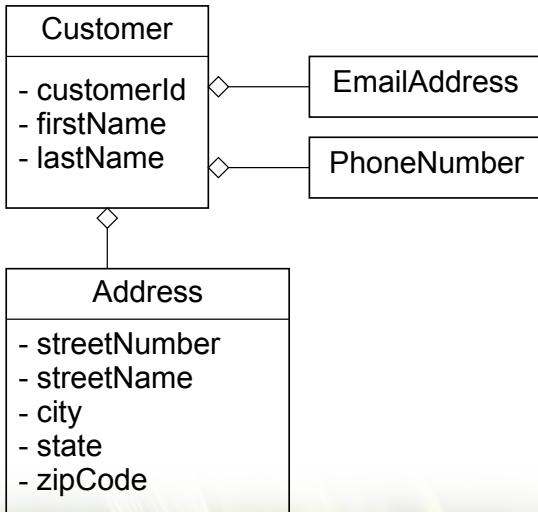


Table in Database

CUSTOMER
CUST_ID <>PK>>
FIRST_NAME
LAST_NAME
EMAIL
PHONE
STREET_NUMBER
STREET_NAME
CITY
STATE
ZIP_CODE

Copyright © 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.
18

O/R Mismatch: Identity (1)

- In Java, there is a difference between Object identity and Object equivalence:
 - `x == y` *identity* (same memory address)
 - `x.equals(y)` *equivalence*
- In a database, identity is based solely on primary keys:
 - `x.getEntityId().equals(y.getEntityId())`

Copyright © 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.
19

- 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**



Copyright © 2009 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

- Object Query Language
- Automatic Change Detection
- Persistence by Reachability
- Caching
 - Per-Transaction (1st Level)
 - Per-DataSource (2nd Level)

Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

21

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"
```

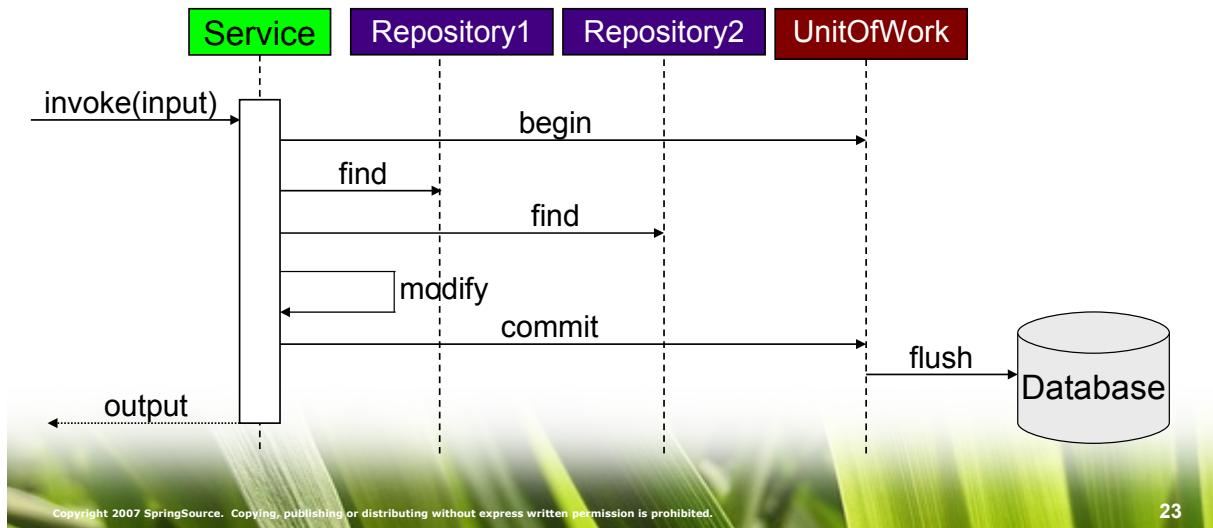
Copyright 2007 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

22

Automatic Change Detection



- When a unit-of-work completes, all modified state will be synchronized with the database.



Persistence by Reachability (1)



- 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.findByConfirmationId(cid);  
// order is now a managed object – retrieved via ORM
```

```
List<LineItem> items = order.getLineItems();  
for (LineItem item : items) {  
    if (item.isCancelled()) { order.removeItem(item); }  
    // the database row for this item will be deleted  
}  
  
if (order.isCancelled()) {  
    orderRepository.remove(order);  
    // all item rows for the order will be deleted  
}
```

Caching

- The first-level cache (1LC) is scoped at the level of a unit-of-work
 - When an object is first loaded from the database within a unit-of-work it is stored in this cache
 - Subsequent requests to load that same entity from the database will hit this cache first
- The second-level cache (2LC) is scoped at the level of the SessionFactory
 - Reduce trips to database for read-heavy data
 - Especially useful when a single application has exclusive access to the database

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



- 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)

5

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

6

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) {    'property' access uses setter  
        this.firstName = firstName;  
    }  
    ...}
```

7

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

8

Mapping collections with annotations



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

Propagate all operations to the underlying objects

9

Mapping collections with XML



```
public class Customer {  
    private Long id;  
    private Set addresses;  
...}
```

```
<class name="Customer" table="T_CUSTOMER">  
    ...  
    <set name="addresses" table="T_ADDRESS"  
          cascade="all" access="field">  
        <key column="CUST_ID"/>  
        <one-to-many class="Address"/>  
    </set>  
</class>
```

10

- 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



- 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();
```

13

Topics in this session

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

14

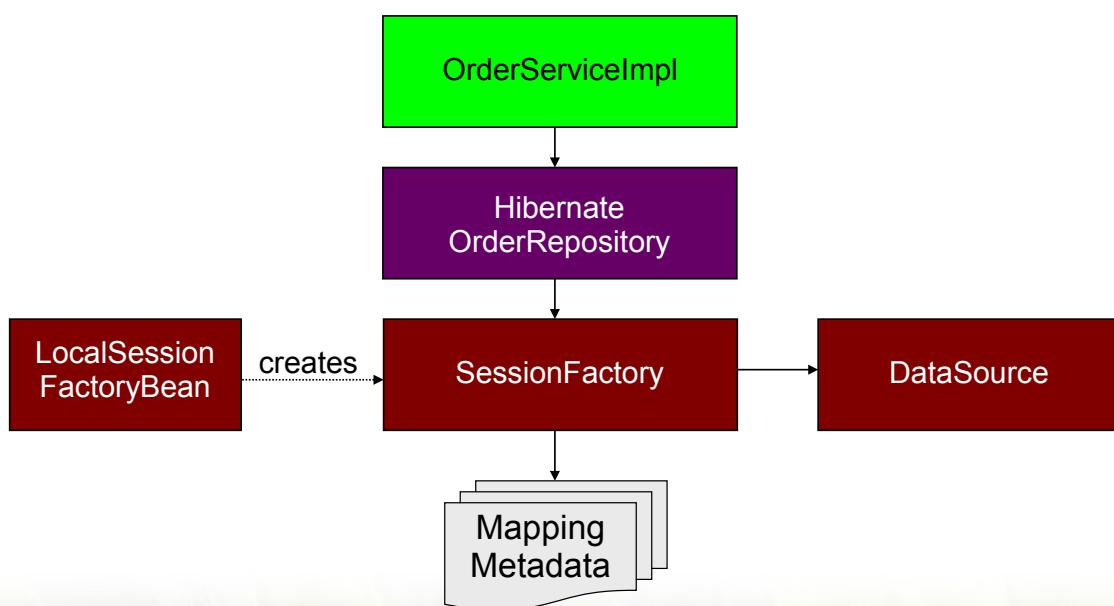
Configuring a SessionFactory (1)



- 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

15

Configuring a SessionFactory (2)



16

Configuring a SessionFactory with annotation mappings



```
<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="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"/>
```

17

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"/>
```

18

Topics in this session



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



19

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



20

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

21

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 find.byId(long orderId) {  
        // use the Spring-managed Session  
        Session session = this.sessionFactory.getCurrentSession();  
        return (Order) session.get(Order.class, orderId);  
    }  
}
```

22

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

23

Spring-managed Transactions (4)



- Best practice : use read-only transactions when you don't write anything in the database
- Prevents Hibernate from flushing its session, which can result in a dramatic performance improvement
- Can provide an additional safeguard with some databases : Oracle will only accept SELECT statements

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

24

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



25

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



26

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**

27

Transparent Exception Translation (3): Java 5+ Environment



-
- the code

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

- the configuration

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

28

- 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>
```

29

Topics in this session

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

30

- 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+



31

Configuring HibernateTemplate

- Simply provide a reference to the SessionFactory

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



32

- Spring also provides this convenience base class that data access objects can extend if they need a `HibernateTemplate`
 - Inject it with a reference to the `SessionFactory`
 - It creates the `HibernateTemplate` for you
 - No value for Native DAOs



33

Extending HibernateDaoSupport

```
public class HibernateCustomerDao extends HibernateDaoSupport
    implements CustomerDao {

    public List findAllCustomers() {
        String hql = "from Customer";
        return getHibernateTemplate().find(hql);
    }

    // other data access methods
}
```



34

Configuring a subclass of HibernateDaoSupport



```
<bean id="customerDao"
      class="example.customer.HibernateCustomerDao" >
  <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/Customer.hbm.xml</value>
    </list>
  </property>
</bean>
```

35



LAB

Using Hibernate with Spring

Overview of Spring Web

Developing modern web applications
with Spring



Topics in this Session

- **Goals of effective web application architecture**
- Overview of Spring Web
 - Spring Web MVC
 - Spring JavaScript
 - Spring Web Flow
 - Spring Faces
- Using Spring in Web Applications



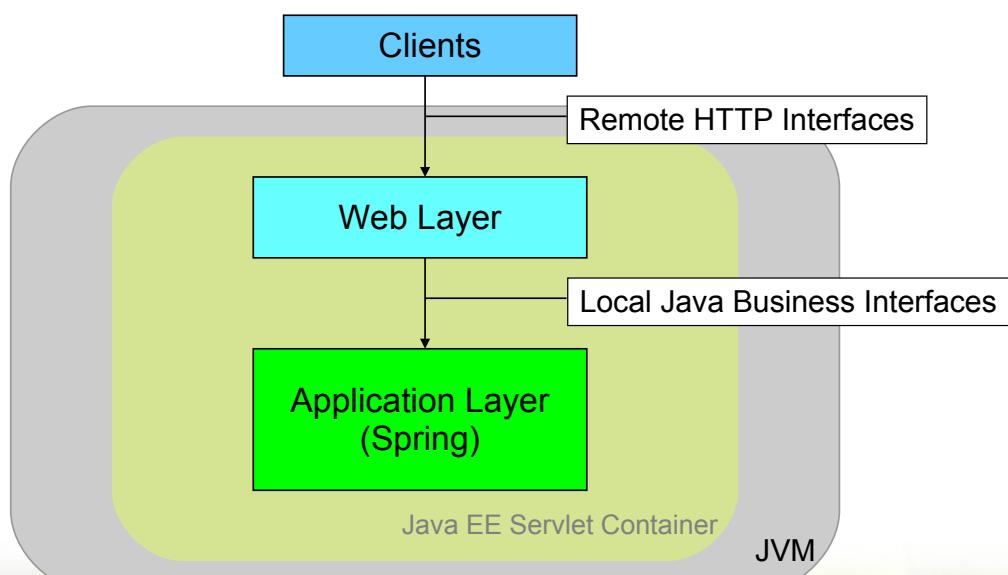
Goals of Effective Web Application Architecture



- Be thin
 - An effective web architecture is thin
 - The web layer plays the role of an adapter responsible for mapping HTTP requests to application functionality
 - The application layer encapsulates the business logic and domain complexity
- Be clean
 - An effective web architecture allows for a separation of concerns
 - Designers can focus on markup and style
 - Java developers can focus on request processing logic
- Good modern web frameworks help meet these goals

3

Effective Web Application Architecture



4

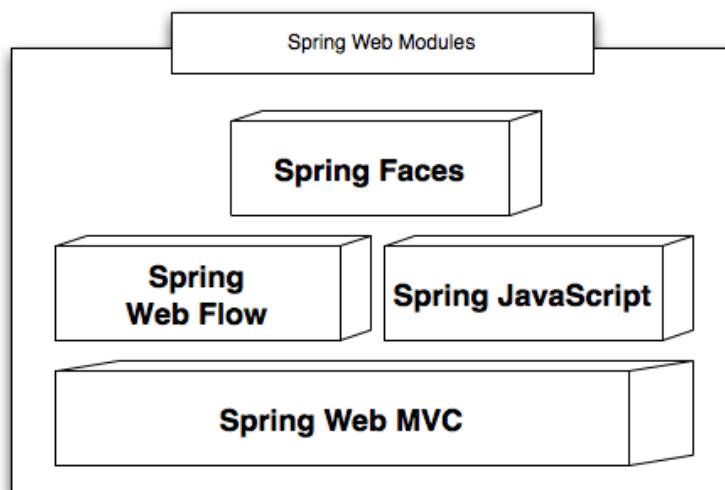
Topics in this Session



- Goals of effective web application architecture
- **Overview of Spring Web**
 - Spring Web MVC
 - Spring JavaScript
 - Spring Web Flow
 - Spring Faces
- Using Spring in Web Applications

5

Spring Web: The Big Picture



6

- Spring's web framework
 - Optional component
 - Uses Spring for its own configuration
- Builds on the Java Servlet API
 - A parallel Portlet version is also provided
- Comparable to Struts 1.x
 - provides a stronger architectural foundation
- The core platform for developing web applications with Spring
 - All higher-level modules build on it



7

Spring Web MVC Strengths

- Easy to learn
 - Few external dependencies
- Flexible
 - Smart extension points put you in control
 - Layered architecture facilitates reuse
- Versatile
 - Integrates popular view and controller technologies
- Uses Spring for its configuration
 - testable artifacts
 - benefits from dependency injection



8

- Supported View Technologies
 - JSP / JSTL
 - Apache Velocity
 - Freemarker
 - Adobe PDF
 - Microsoft Excel
 - Jasper Reports
 - XML / XSLT



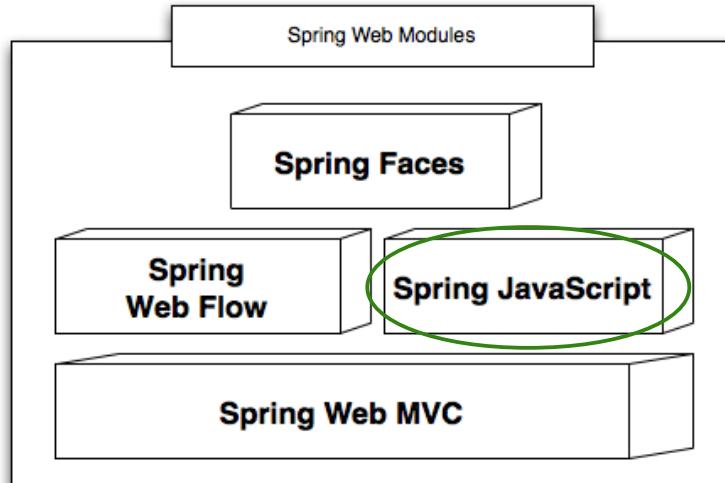
9

Other Key Features

- Data Binding Framework
- Validation Support
- Internationalization Support
- Tag Library



10



-
- JavaScript abstraction framework
 - builds on existing JS frameworks
 - Dojo is currently supported
 - Simplifies use for common enterprise scenarios
 - avoid proliferation of JavaScript code
 - Aims to promotes best practices
 - Progressive enhancement



- Decorate form input fields
- Client-side validation
- Ajax events
- Fragment rendering
- Ajax modal dialogs (pop-up)



13

More Spring JS Features

- Graceful degradation
 - Applies progressive enhancement techniques
 - Makes your web application more accessible
- CSS framework
- Performance optimizations
 - Our reference applications receive a “A” Yahoo UI Performance Grade
 - Custom Dojo Build
 - Reduces overall amount of JavaScript transferred to client
 - ResourceServlet
 - Serves up cached, compressed JavaScript and CSS resources
- Ajax error handling



14

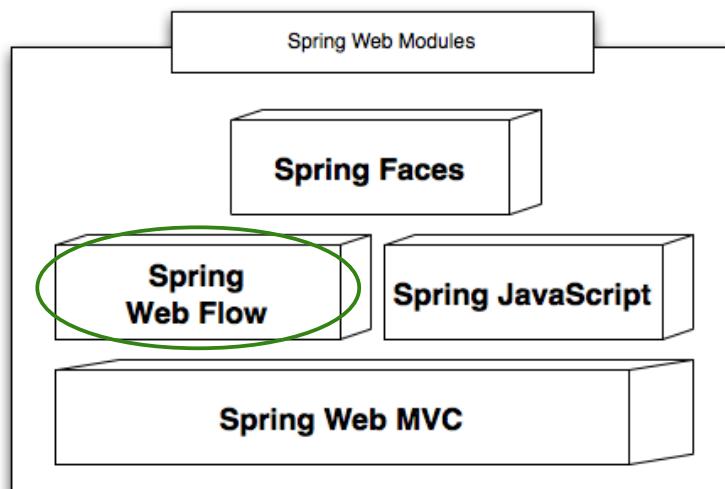
See Spring JS In Action



- SpringSource bundle repository application
 - <http://www.springsource.com/repository>
- Built on Spring MVC and Spring JavaScript

15

Spring Web Flow

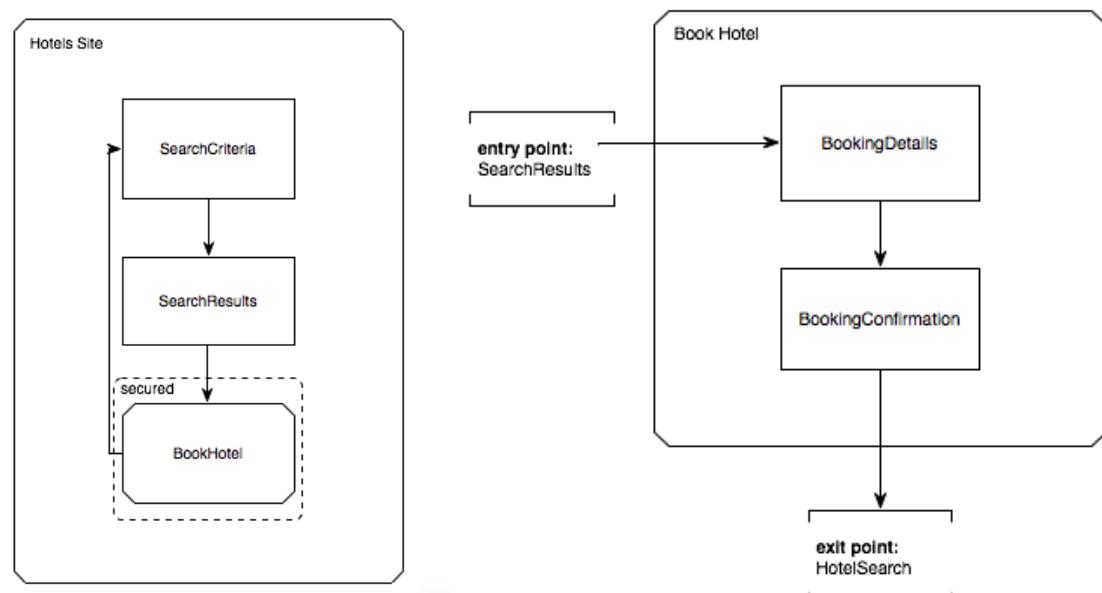


16

- Plugs into Spring Web MVC as a Controller technology for implementing stateful “flows”
 - *“A flow encapsulates a reusable sequence of steps that can execute in different contexts”*
- Raises the abstraction level by providing a dedicated *flow definition language* (DSL)

17

Web Flow Sweet Spot



18

Example Flow Definition



```
<flow>
    <view-state id="step1">
        <transition on="eventA" to="step2"/>
    </view-state>

    <view-state id="step2">
        <transition on="eventB" to="done"/>
    </view-state>

    <end-state id="done"/>
</flow>
```

19

Flow Design



- Flows model stateful operations that take a series of requests to complete
 - BookHotel
 - FlightCheckIn
 - EditAccount
 - GetInsuranceQuote
 - ApplyForLoan
 - RegisterNewUser
- Can be used for simple “single-request” operations, but not its focus or “sweet spot”

20

- Flows are addressable web resources
 - Address them to begin new operations
- Examples
 - <http://localhost/accounts/register>
 - Begins a new account registration operation
 - <http://localhost/flights/checkin>
 - Begins a new flight check-in operation

21

Flow Executions are Isolated

- Flows execute in isolation from other users
 - Provide a local context for operations currently “in progress” but not yet completed
- The execution context for each operation is *session-specific*
 - <http://localhost/flights/checkin?execution=e1s1>

Session-specific flow execution key



22

Spring Security Integration



```
<flow>
    <secured attributes="ROLE_USER" />
</flow>
```

Login Required

Valid username/passwords are:

- keith/melbourne
- erwin/leuven
- jeremy/atlanta
- scott/rochester



A screenshot of a login form. It has two input fields: 'User:' containing 'erwin' and 'Password:' containing '*****'. Below the fields is a checkbox labeled 'Don't ask for my password for two weeks:'. At the bottom right is a 'Login' button.

- Can also secure states and transitions
- Can refer to user Principal using EL
 - Use the **currentUser** implicit variable
 - Resolvable within flow and view templates

23

Flow-Managed Persistence

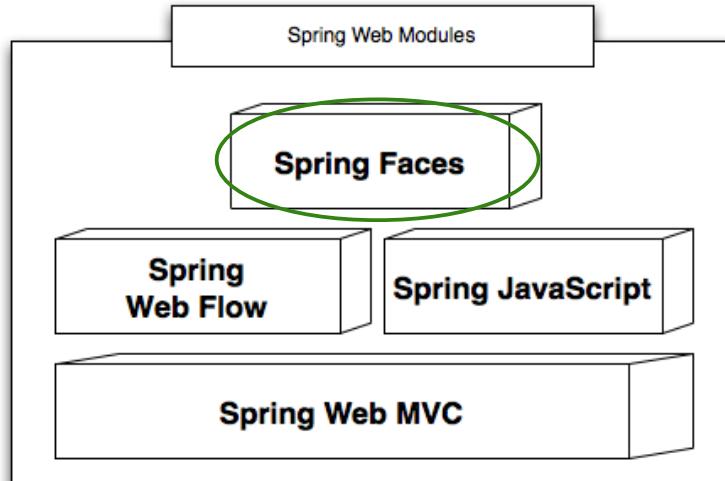


```
<flow>
    <persistence-context/>
</flow>
```

- Scopes a persistence context with this flow
- Automatically used by your data access objects
- Support for committing changes made by the flow upon completion
 - `<end-state id="bookingConfirmed" commit="true" />`

24

Spring Faces



25

Spring Faces



- Combines the JSF UI component model with Web Flow navigation/state management
 - All in a native Spring MVC environment
- Spring Faces also includes a lightweight JSF component library
 - Designed for the 80%
 - Includes Ajax support, client-side form validation
 - Built on a new Spring JavaScript
 - Integrates Dojo as the primary UI toolkit
 - Applies progressive enhancement techniques



26

-
- Enhanced page navigation rules
 - Better exception handling policies
 - Easier client and server-side (model) validation
 - Fine-grained state management
 - flow scope, view scope, flash scope
 - Ajax support with graceful degradation



27

Spring Travel Reference Application Comparison

	Plain JSF	Spring Faces
Number of controller artifacts	4	2
Lines of code	333	95



- Makes JSF easier for the Spring community
- Pioneering approach to JSF integration
- Compliant with JSF 1.1 and 1.2 specifications
- Major component libraries are all usable in a Spring Faces environment
 - JBoss Rich Faces, Apache Trinidad, IceSoft IceFaces



29

Topics in this Session

- Goals of effective web application architecture
- Overview of Spring Web
 - Spring Web MVC
 - Spring JavaScript
 - Spring Web Flow
 - Spring Faces
- **Using Spring in Web Applications**



30

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

31

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>
```

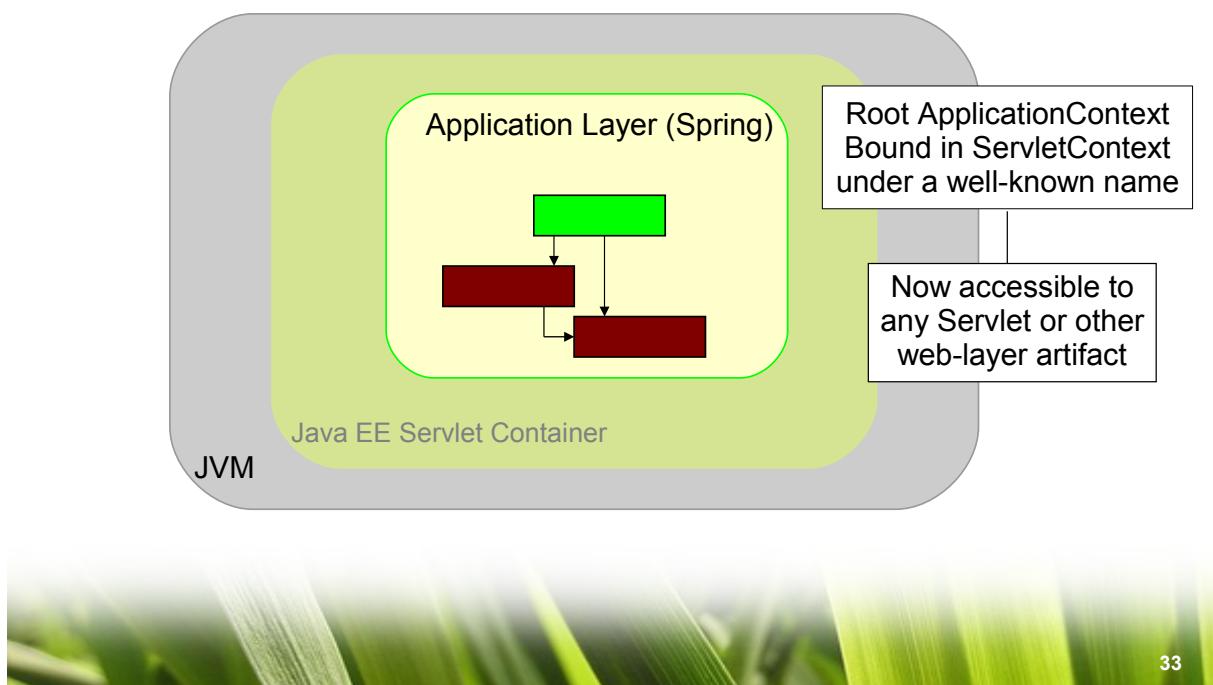
The application context's configuration file(s)

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

Loads the ApplicationContext into the ServletContext
before any Servlets are initialized

32

Servlet Container After Starting Up



33

Access To Spring-Managed Objects



- **WebApplicationContextUtils**
 - static helper
 - provides access to the Spring Application Context
- **Built-in framework integration**
 - JSF
- **Spring Web MVC**
 - via dependency injection

34

"Plain" Servlet Access To Spring



```
public class TopSpendersReportGenerator extends HttpServlet {  
    private MerchantReportingService reportingService;  
  
    @Override  
    public void init() {  
        ApplicationContext context = WebApplicationContextUtils.  
            getRequiredWebApplicationContext(getServletContext());  
        reportingService = (MerchantReportingService)  
            context.getBean("merchantReportingService");  
    }  
    ...  
}
```

Saves the reference to the application entry-point for invocation during request handling

Looks up ApplicationContext under its special name in the ServletContext

35

Spring Web MVC Controllers



```
@Controller  
public class TopSpendersReportController {  
    private MerchantReportingService reportingService;  
  
    @Autowired  
    public TopSpendersReportController(MerchantReportingService service) {  
        this.reportingService = service;  
    }  
    ...  
}
```

Dependency is automatically injected by type

36

- JSF-centric integration
 - Spring plugs in as a JSF managed bean provider
 - configure a VariableResolver in faces-config.xml
 - replace all managed beans in faces-config.xml
- Spring-centric integration
 - JSF managed beans typically declared in flow definition in one of several scopes



37

Summary



- Spring Web has a set of modules for rich web application development
- Application-layer, Spring-managed objects can be used in any web application
- Separate 4-day SpringSource course available:
 - “Rich Web Applications with Spring”



38

Spring Web MVC Essentials

Getting started with Spring Web MVC



Topics in this Session

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



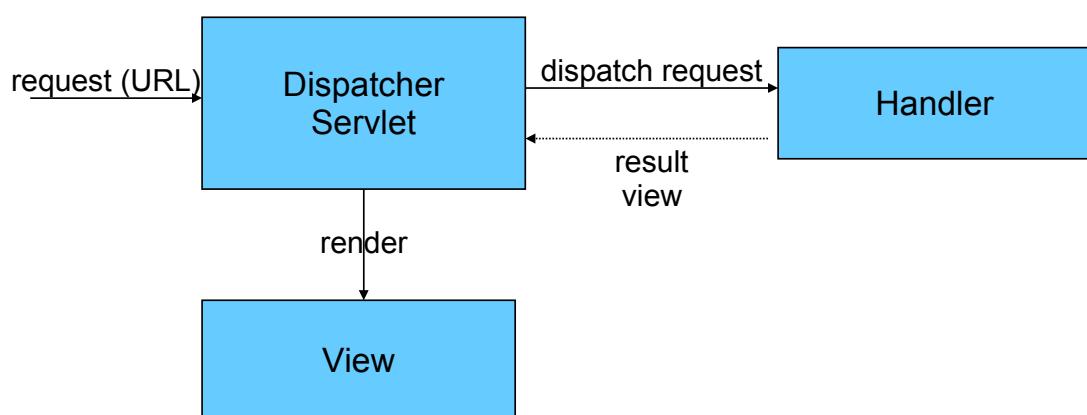
Web Request Handling Overview



- 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



- 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



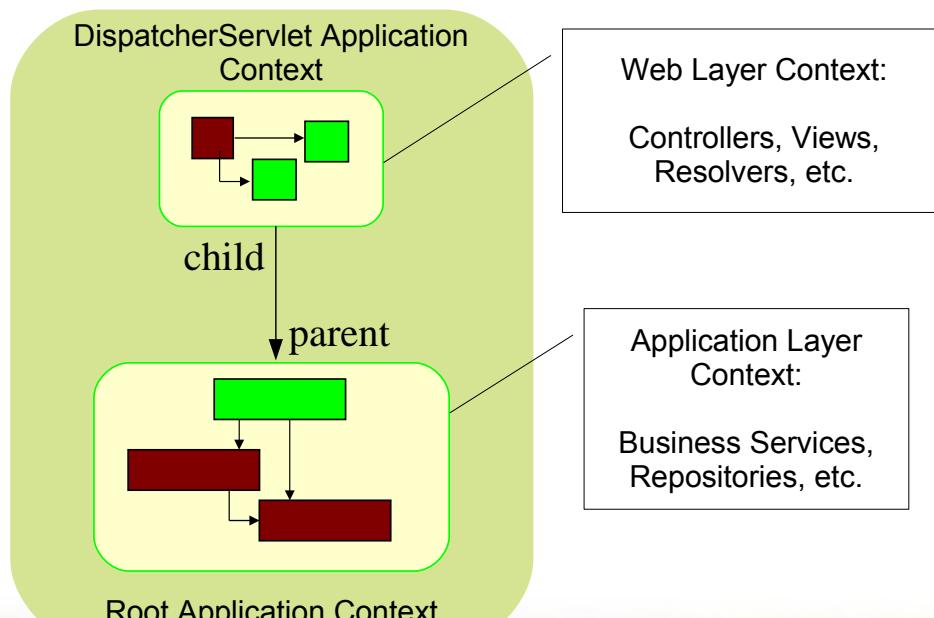
DispatcherServlet Configuration



- 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

7

Servlet Container After Starting Up



8

Topics in this Session



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

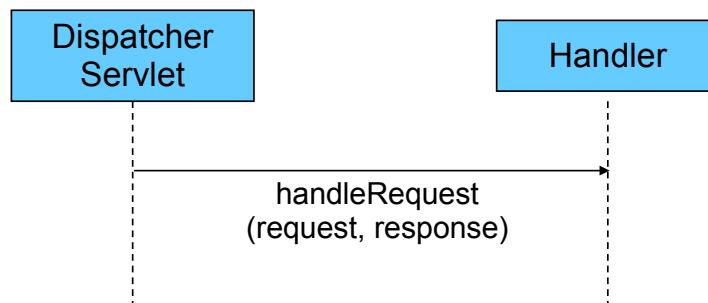


9

Request Handlers



-
- Incoming requests are dispatched to handlers
 - There are potentially many handlers per DispatcherServlet



10

- 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/>”
- This training will focus on Spring @MVC

11

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”

12

- Mapping rules you define are 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



13

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) {  
        ...  
    }  
}
```



14

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

15

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

16

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

Not very common!



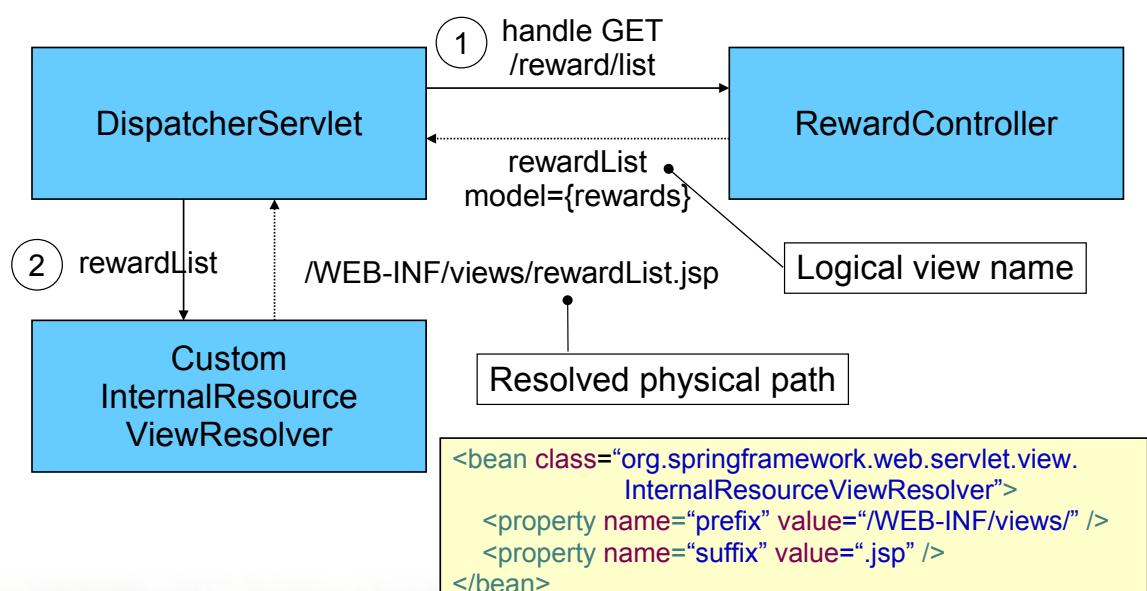
View Resolvers



- 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

19

Custom Internal Resource View Resolver Example



20

Topics in this Session



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



21

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



22

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

23

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
```

24

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>
```

25

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(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

26

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

27

4. Register the Controller



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

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

28

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

29

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) { ... }  
}
```

30



LAB

Spring Web MVC Essentials



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.



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



5

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



6

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



7

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



8

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



9

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

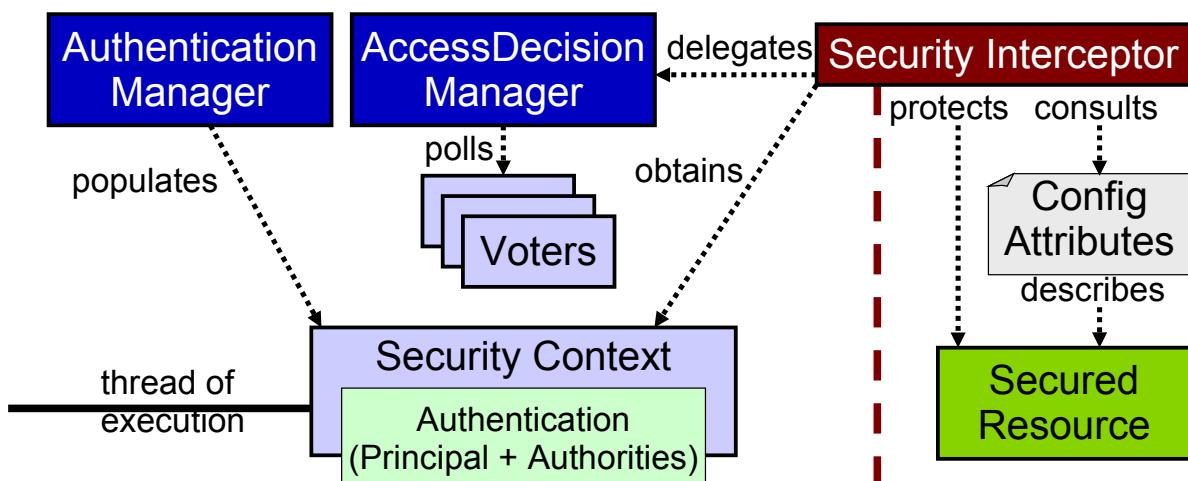


10

- 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

11

Spring Security: the Big Picture



12

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

13

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)

14

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>
```

15

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" />
```

16

Topics in this Session



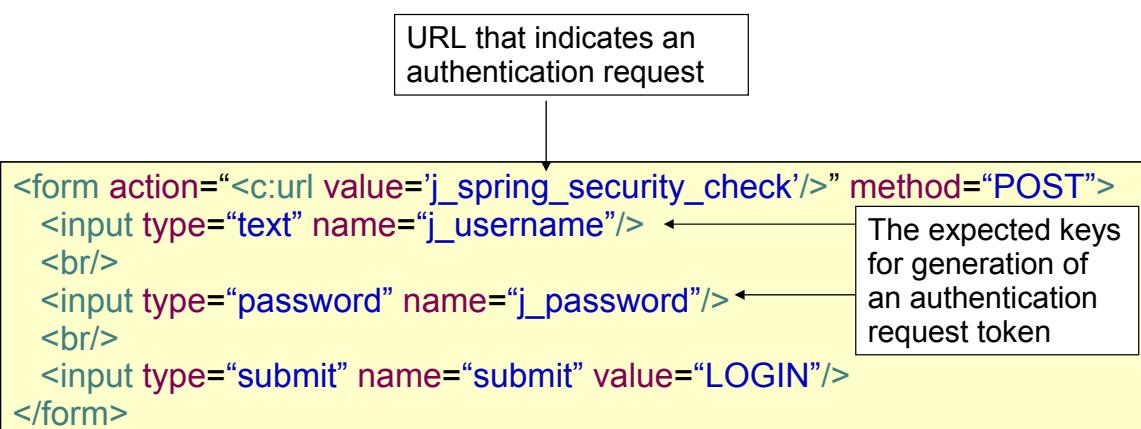
-
- High-Level Security Overview
 - Motivations of Spring Security
 - Spring Security in a Web Environment
 - **Configuring Web Authentication**
 - Using Spring Security's Tag Libraries
 - Method security
 - Advanced security: working with filters



17

A blurred background image showing green leaves.

An Example Login Page



18

A blurred background image showing green leaves.

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

```
<security:authentication-provider>
    ... Configure specific UserDetailsService
</security:authentication-provider>
```



19

The JDBC UserDetailsService (1/2)

Queries RDBMS for users and their authorities

- Provides default queries
 - SELECT username, password, enabled FROM users WHERE username = ?
 - SELECT username, authority FROM authorities WHERE username = ?



20

The JDBC UserDetailsService (2/2)



- Configuration:

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

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

21

The In-Memory UserDetailsService



- Useful for development and testing

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

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

22

- Implement a custom `UserDetailsService`
 - Delegate to an existing User repository or DAO
- LDAP
- X.509 Certificates
- JAAS Login Module
- Single-Sign-On
 - SiteMinder
 - JA-SIG Central Authentication Service

Authorization is not affected by changes to Authentication!

23

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

24

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

25

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 <a href="/admin/deleteAll">HERE</a> to delete all records.
</security:authorize>
```

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

26

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



27

Method security



-
- Spring Security 2.0 uses AOP for security at the method level
 - xml configuration with the Spring Security namespace
 - annotations based on Spring annotations or JSR-250 annotations



28

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.springframework..*Service.*(..))"
    access="ROLE_USER" />
</security:global-method-security>
```

29

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) {
        ...
    }
}
```

30

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) {  
        ...  
    }  
}
```

31

Topics in this Session



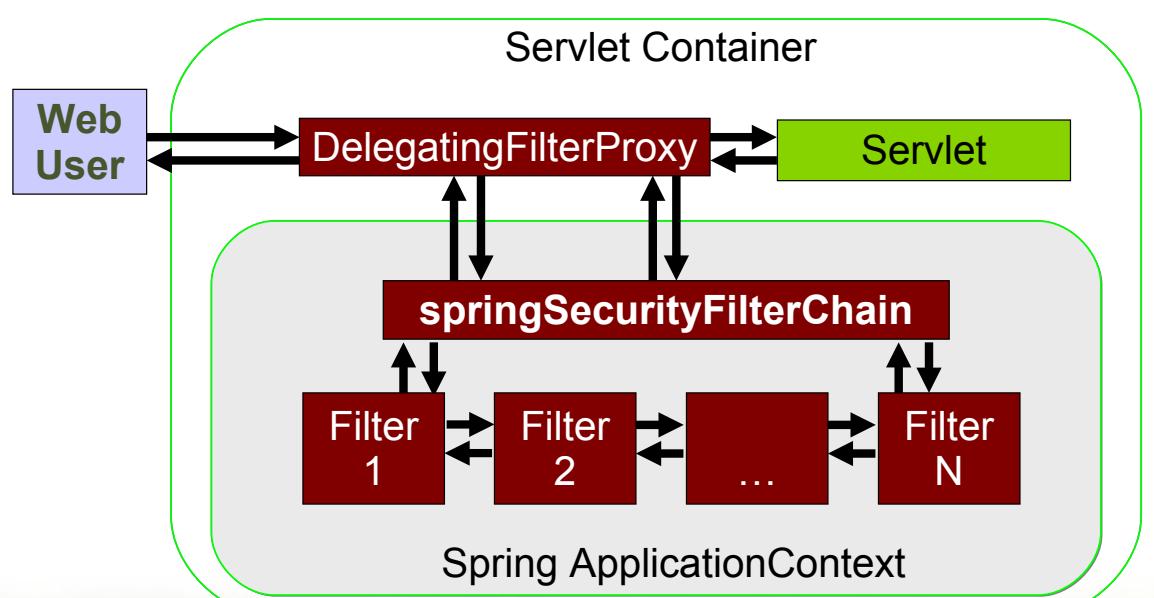
- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- **Advanced security: working with filters**

32

- `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*

33

Web Security Filter Configuration



34

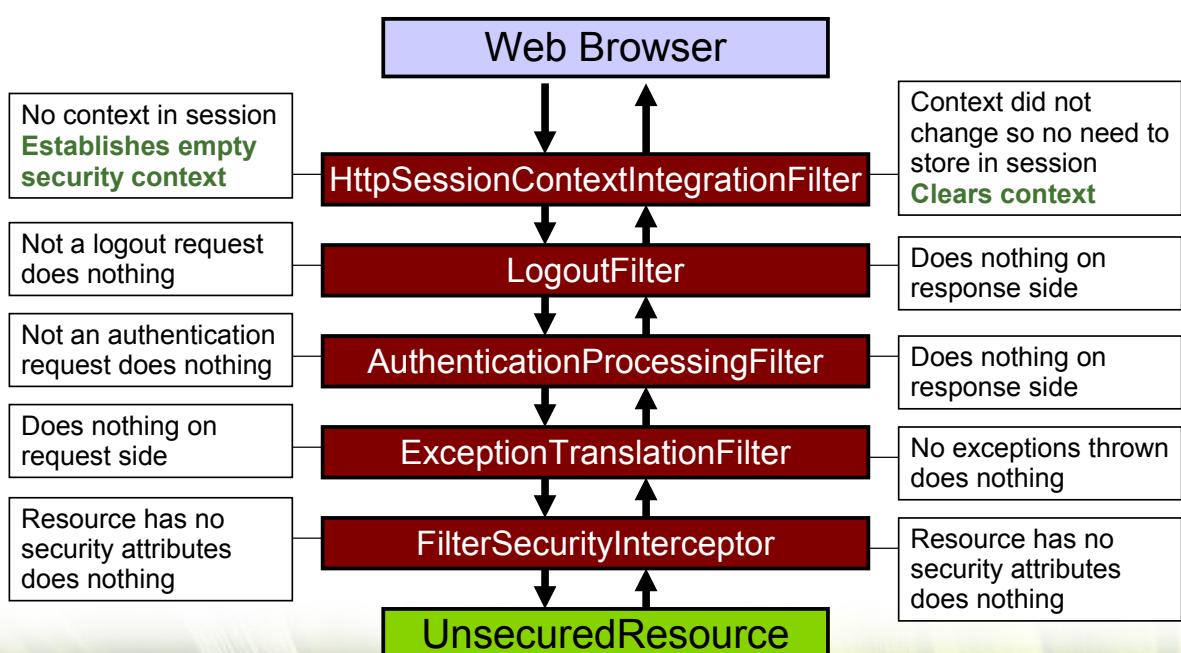
The Filter chain



- 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

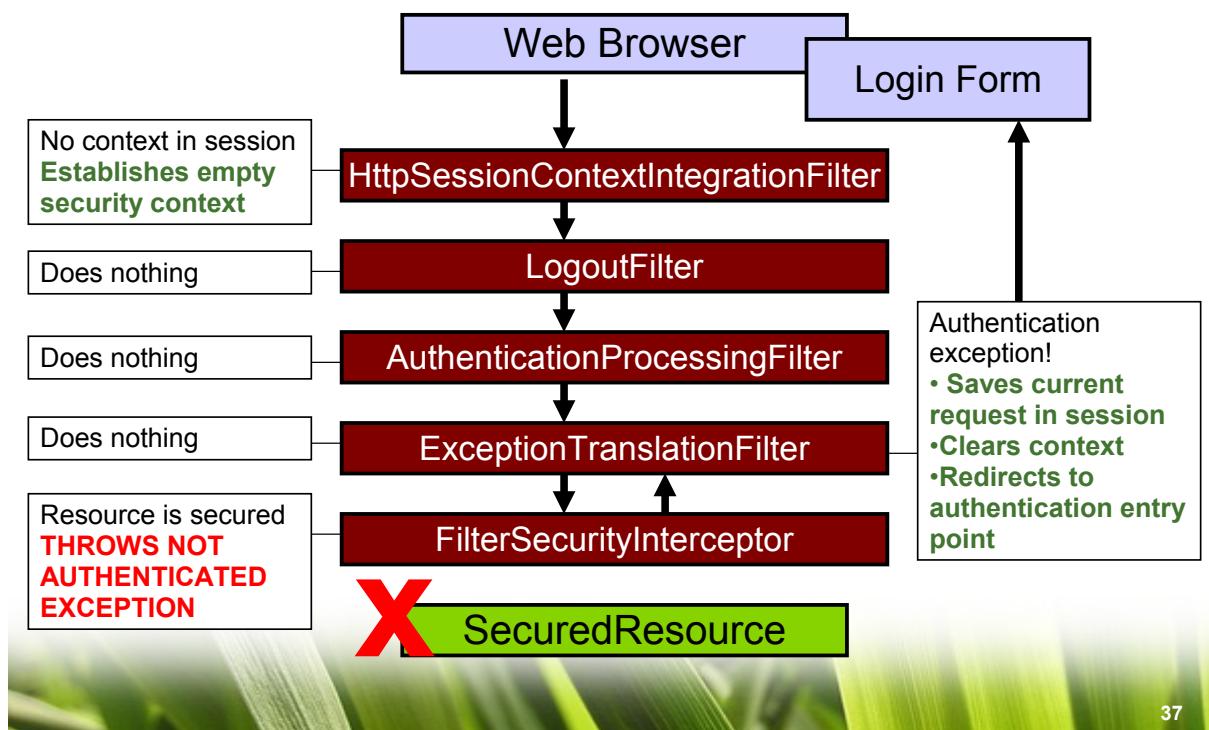
35

Access Unsecured Resource Prior to Login



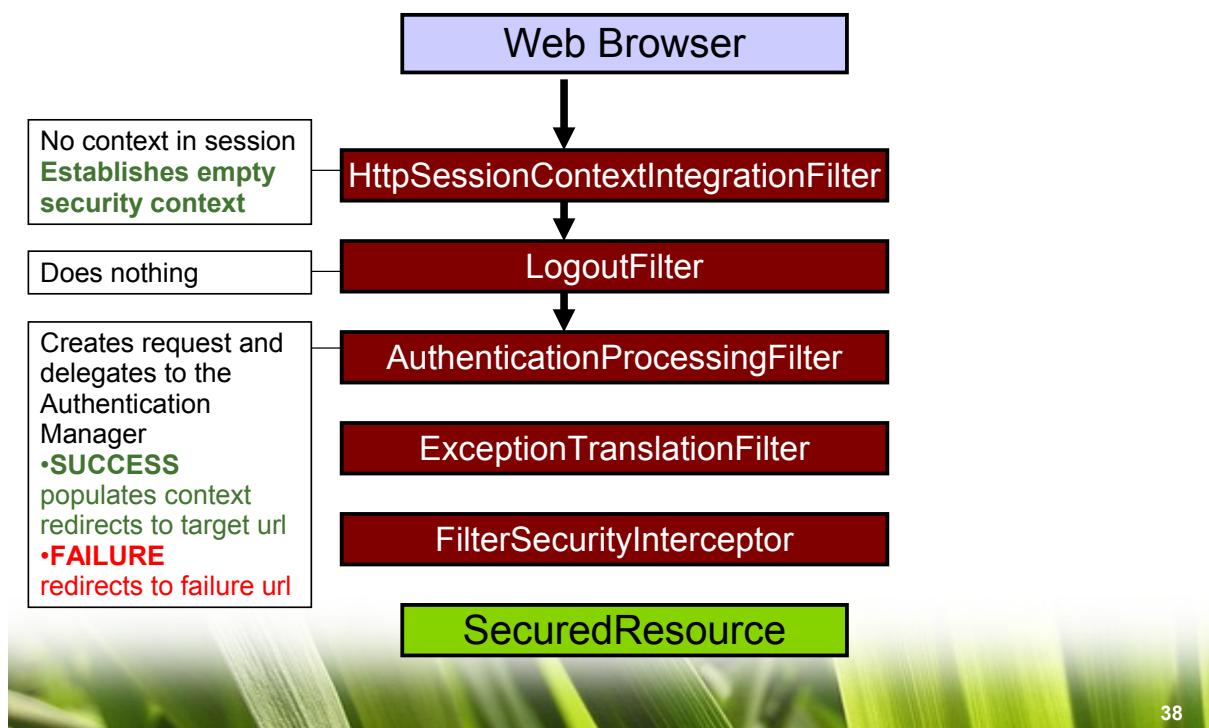
36

Access Secured Resource Prior to Login



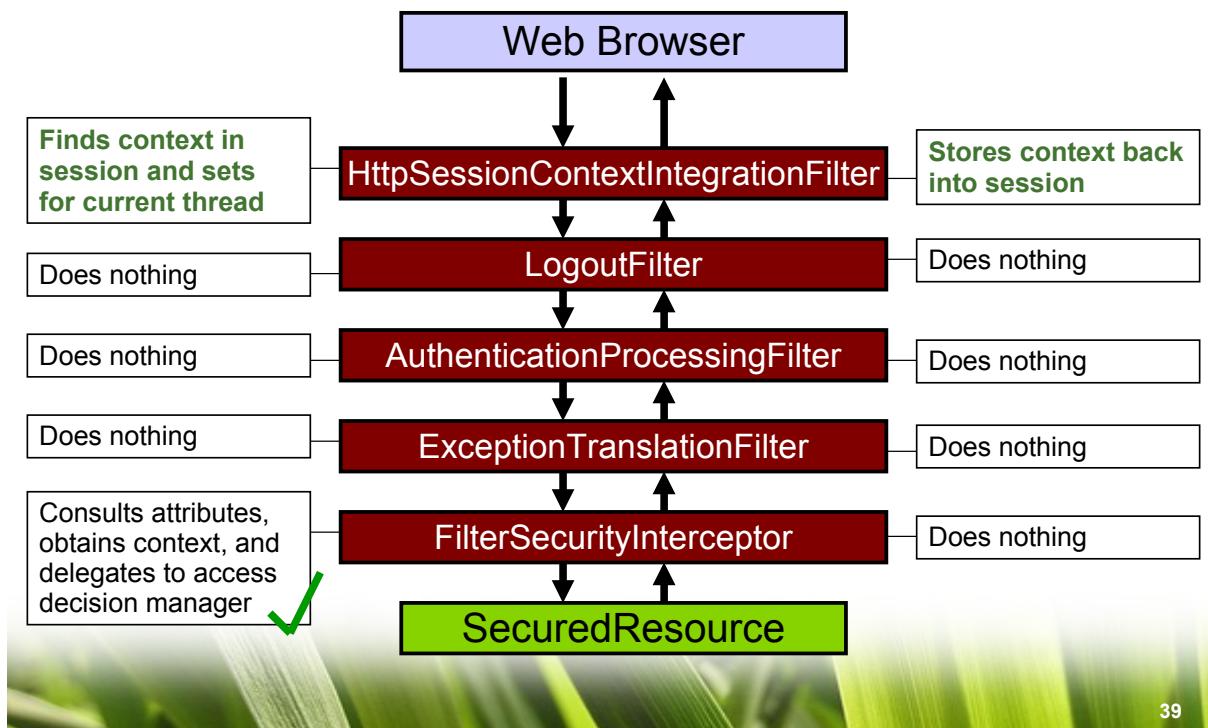
37

Submit Login Request



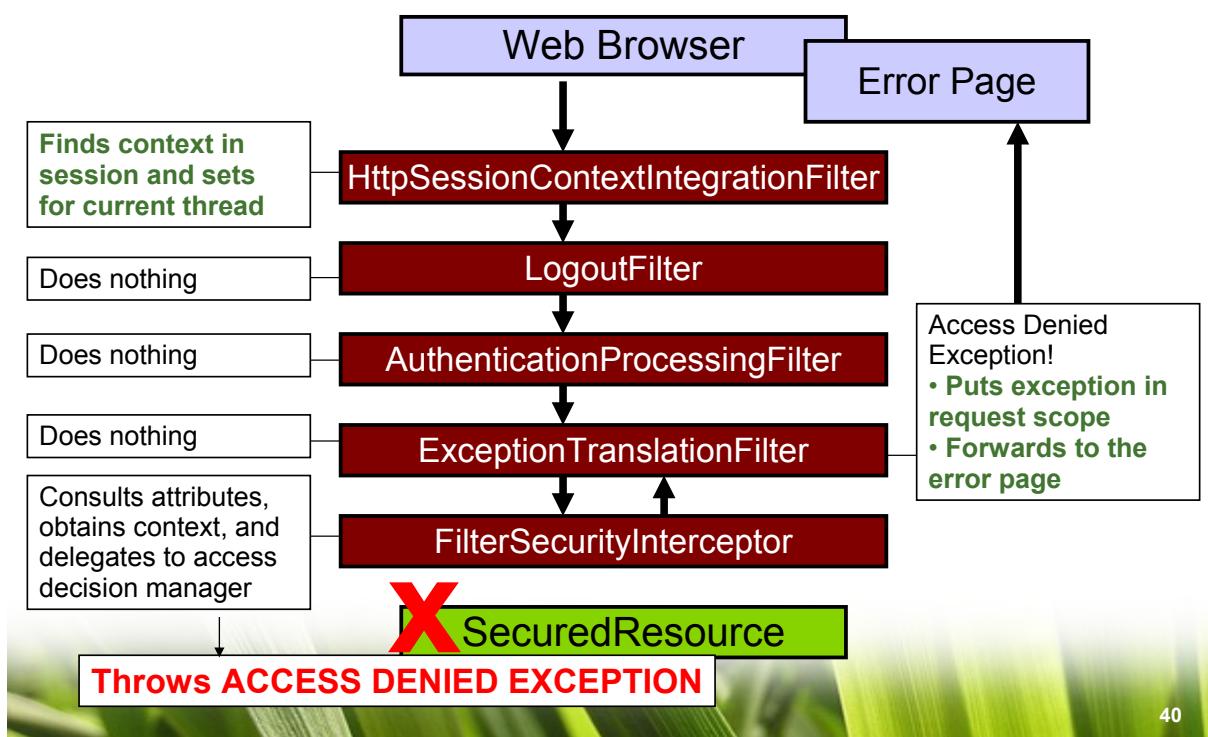
38

Access Resource With Required Role



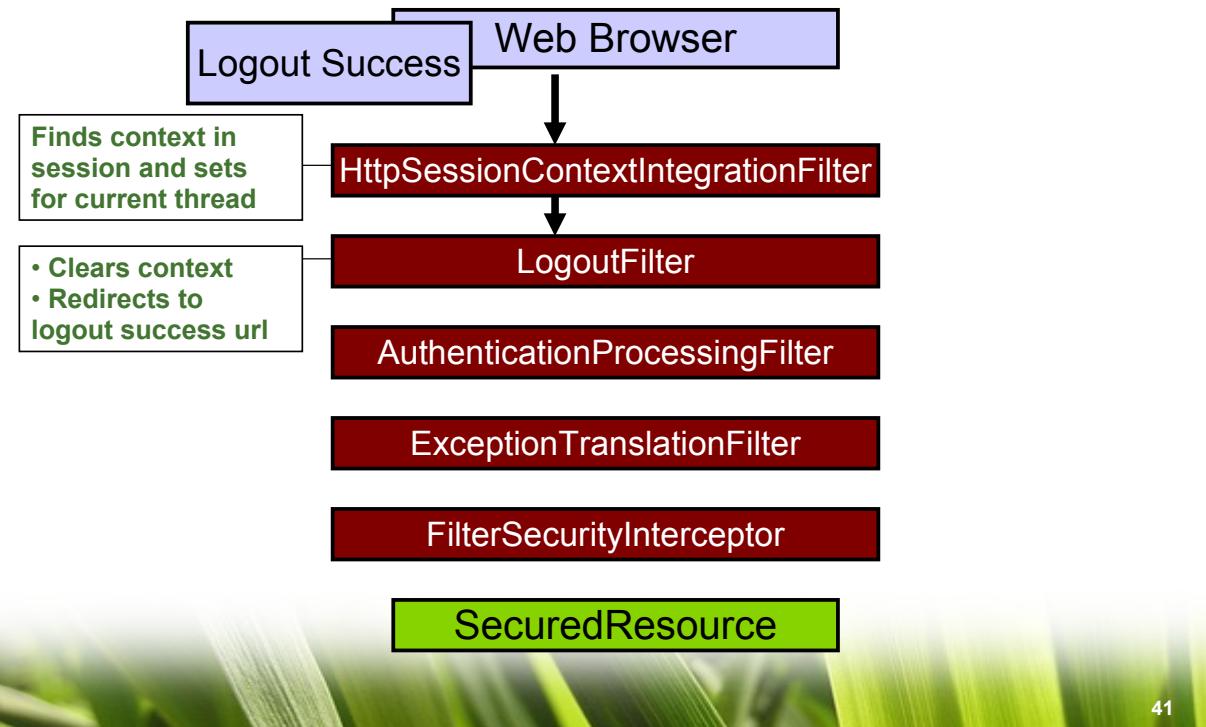
39

Access Resource Without Required Role



40

Submit Logout Request



41

The Filter Chain: Summary



#	Filter Name	Main Purpose
1	HttpSessionContext IntegrationFilter	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 URL patterns

42

Custom filter chain (1/2)



- One filter on the stack may be **replaced** by a custom filter

```
<bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter">
    <security:custom-filter
        position="AUTHENTICATION_PROCESSING_FILTER" />
</bean>
```

- One filter can be **added** to the chain

```
<bean id="myFilter"
      class="com.mycompany.MySpecialFilterWithSpecialFeatures">
    <security:custom-filter
        after="AUTHENTICATION_PROCESSING_FILTER" />
</bean>
```

43

Custom filter chain (2/2)



- Writing the Java class
 - The easiest way is to extend SpringSecurityFilter

```
public class MySpecialAuthenticationFilter extends SpringSecurityFilter {

    public void doFilterHttp(HttpServletRequest request, HttpServletResponse
        response, FilterChain chain) throws IOException, ServletException {
        ...
    }
}
```

44



LAB

Applying Security to a Web Application



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

Introduction to Spring Remoting

Simplifying Distributed Applications



Topics in this Session

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



Goals of Spring Remoting



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



3

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



4

-
- 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



7

Topics in this Session



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

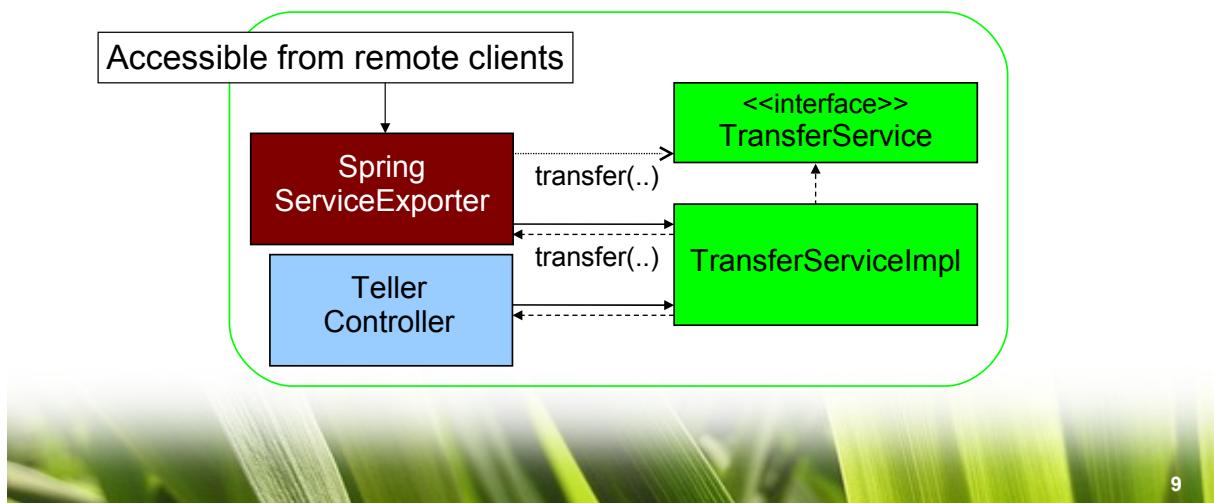


8

Service Exporters



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

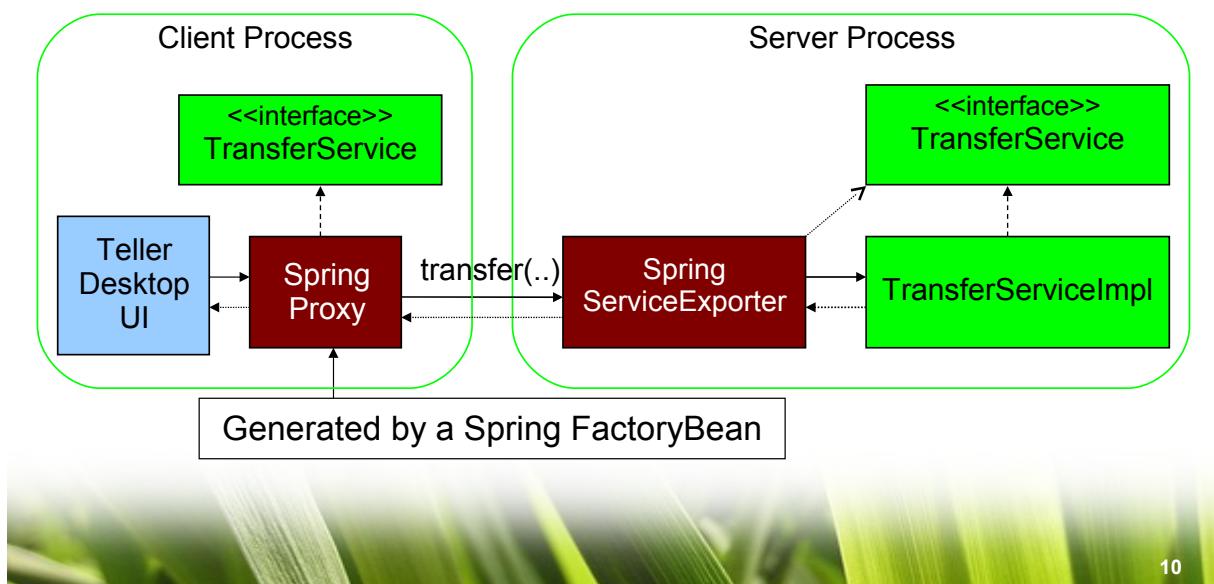


9

Client Proxies



- Dynamic proxies generated by Spring communicate with the service exporter



10

Topics in this Session



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



The RMI Protocol



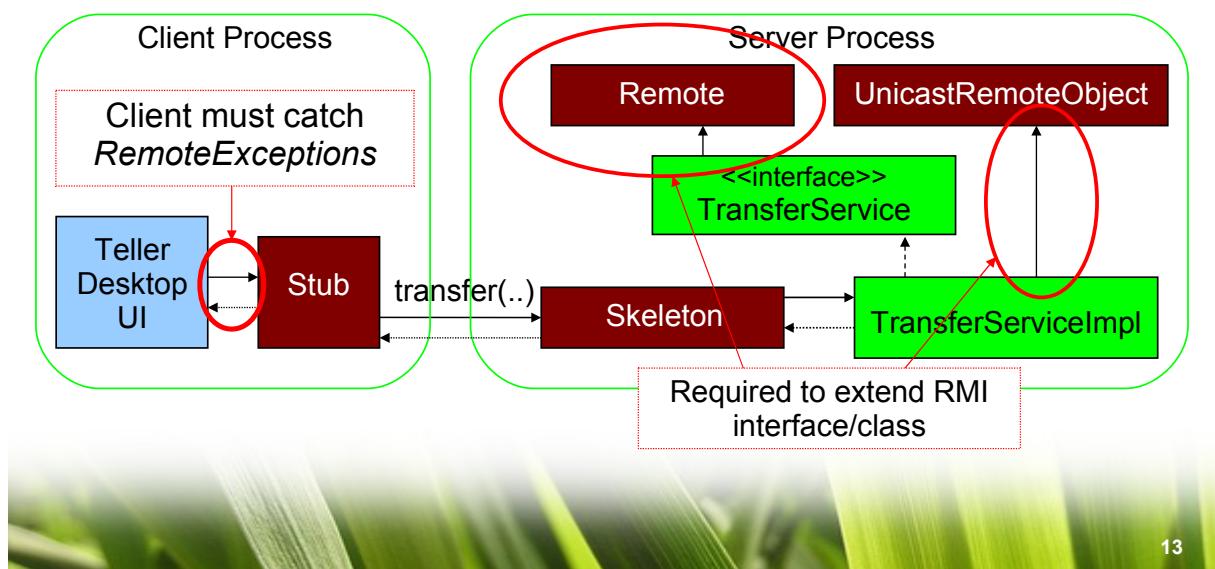
-
- Standard Java remoting protocol
 - Server-side exposes a *skeleton*
 - Client-side invokes methods on a *stub* (proxy)
 - Java serialization is used for marshalling



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

- 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"/>
```

15

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)**

16

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>
```

17

RMI-IIOP for Interoperability: Service



- Generate IIOP stubs and ties
 - Use RMI compiler with IIOP flag: **rmic -iiop**
- Define the service interface
 - Must extend the RMI Remote interface
 - All methods must throw RemoteException
- Implement the service class
 - All methods must throw RemoteException
- Leverage Spring to expose the service to a CORBA Object Request Broker via JNDI
 - Use Spring's **JndiRmiServiceExporter**
 - Provide jndiName and jndiEnvironment properties

18

- Spring also provides a JndiRmiProxyFactoryBean
- The generated proxy can implement any interface
 - No need to throw RemoteExceptions
 - The proxy will convert to the runtime hierarchy
- Client-side code is decoupled from RMI even when accessing an IIOP-based CORBA service

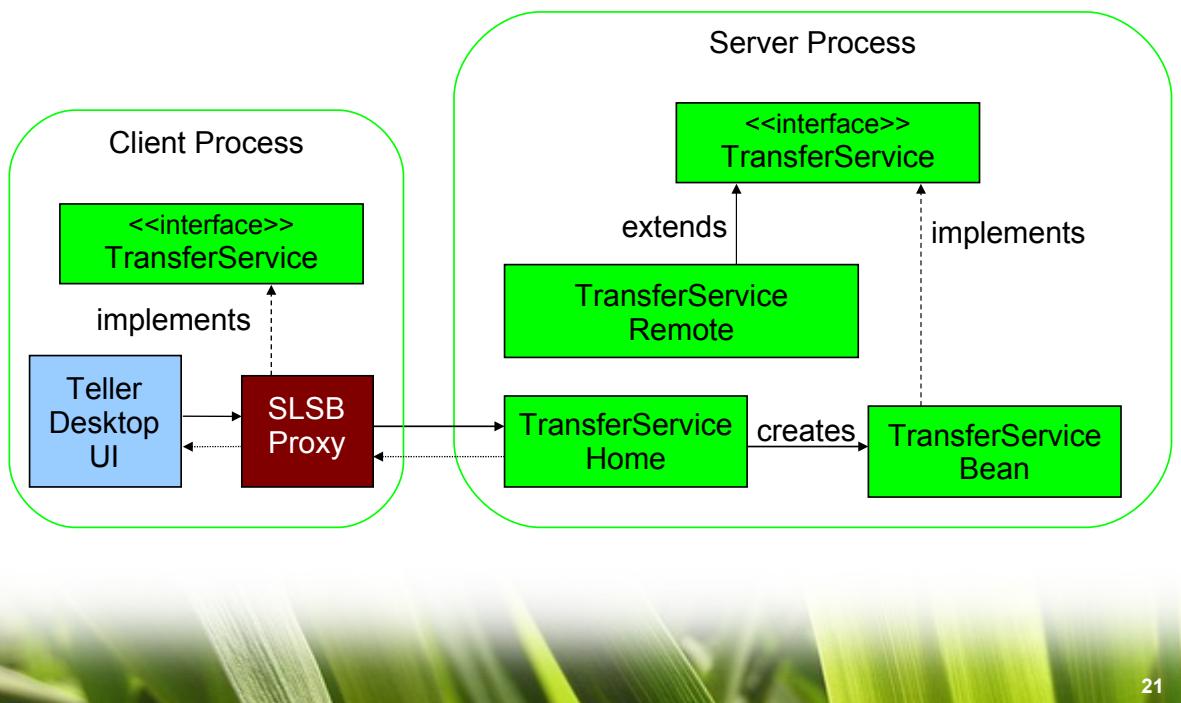


Accessing EJBs

-
- Spring generates proxies to access Stateless Session Beans
 - Conceals JNDI lookup (and retry logic)
 - Caches the EJB home
 - Converts RemoteExceptions to runtime hierarchy
 - Bean definitions can use the jee namespace
 - <jee:local-slsb>
 - <jee:remote-slsb>



Accessing an EJB with a Spring Proxy



21

Configuring the EJB Proxy



- Use the `jee` namespace to generate the proxy

```
<jee:remote-slsb id="transferService"
    jndi-name="java:comp/env/ejb/transferService"
    business-interface="foo.TradeService">
<jee:environment>
    java.naming.provider.url=t3://remoteserver:7001
    java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
</jee:environment>
</jee:remote-slsb>
```

InitialContext settings are not necessary if co-located

- Inject it into the client

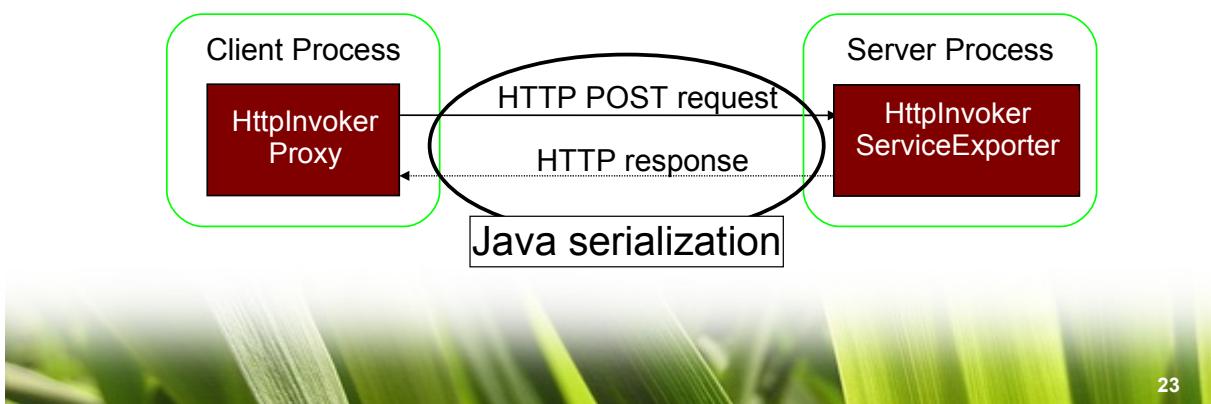
```
<bean id="tellerDesktopUI" class="foo.TellerDesktopUI">
    <property name="transferService" ref="transferService"/>
</bean>
```

22

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



23

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>
```

24

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>
```

25

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

26

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`



27

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)`



28

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



29



LAB

Simplifying Distributed Applications with Spring Remoting



Copyright 2005-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

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.”

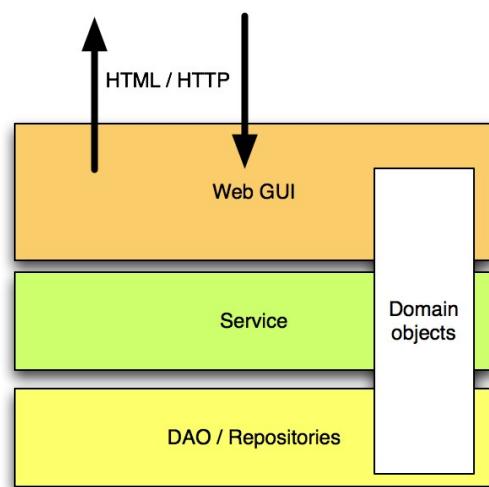
<http://blog.springframework.com/arjen/archives/2007/03/27/ws-duck-typing/>

5

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

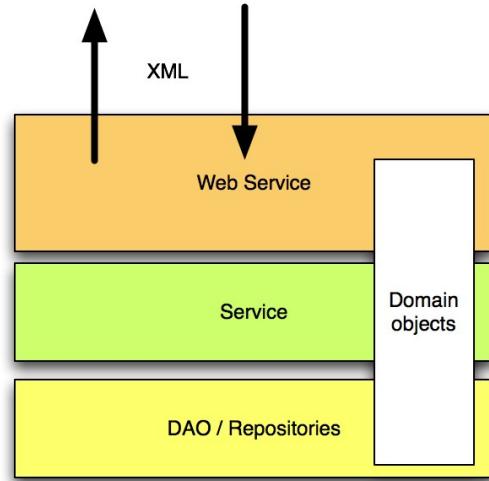


6

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



7

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

8

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>
```



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

13

SOAP Message



14

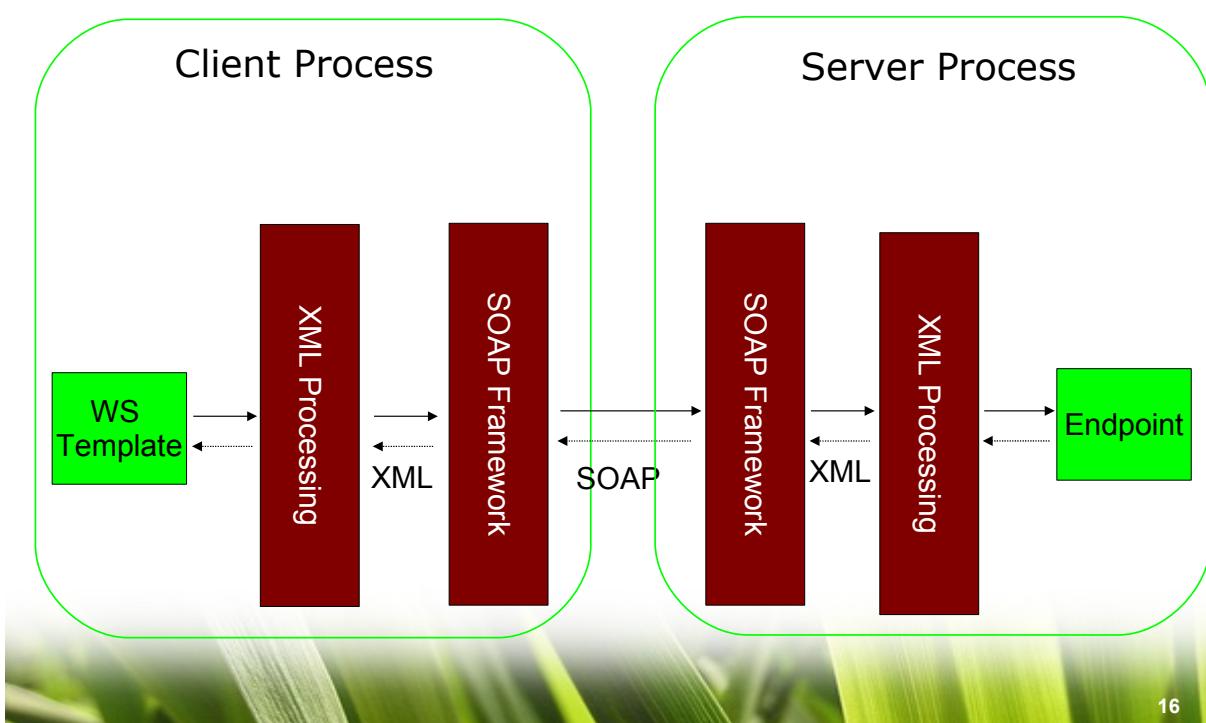
Simple SOAP 1.1 Message Example



```
<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>
```

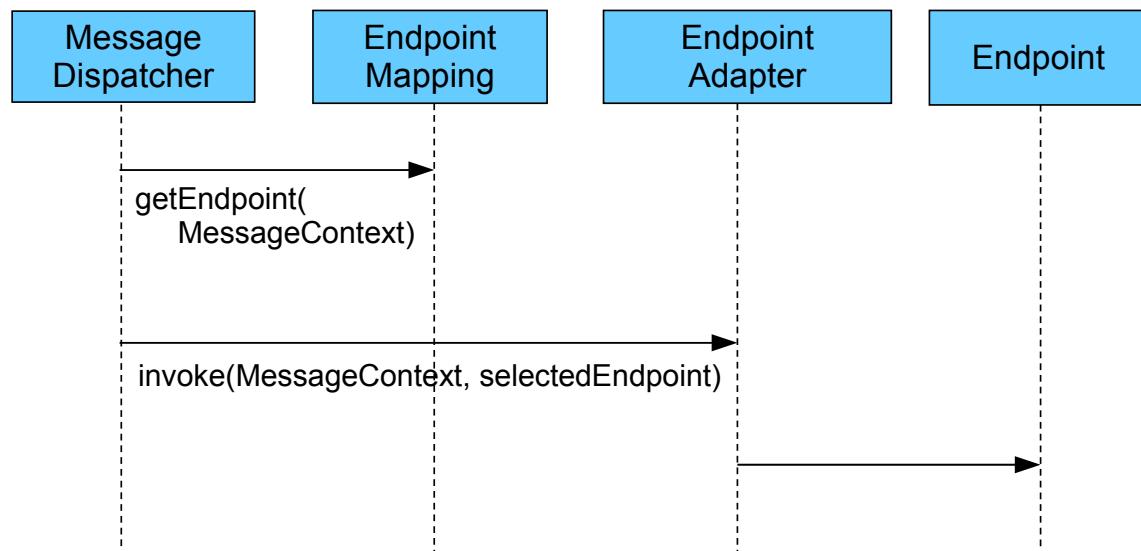
15

Spring Web Services



16

Request Processing



17

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

The application context's configuration file(s)

Loads the ApplicationContext into the ServletContext
before any Servlets are initialized

18

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

19

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

20

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



21

XML Handling techniques



- Low-level techniques
 - DOM (JDOM, dom4j, XOM)
 - SAX
 - StAX
- Marshalling
 - JAXB (1 and 2)
 - Castor
 - XMLBeans
- XPath argument binding



22

- 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>
```

23

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

24

Configure the Endpoint



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

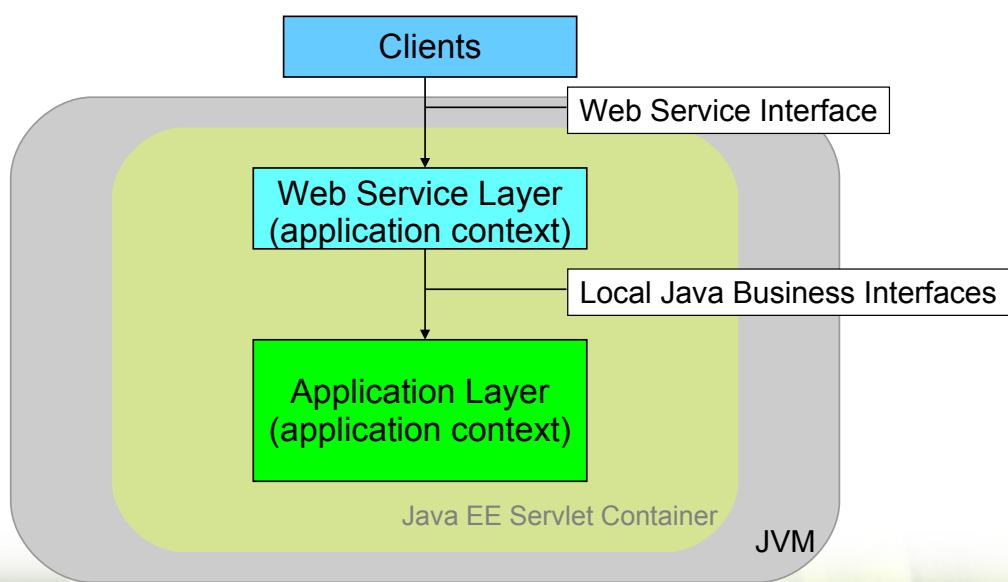
↑
‘transferService’ is defined in the application tier


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

↑
Searching for @Endpoint

25

Architecture of our application exposed using a web service



26

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

29

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.marshalSendAndReceive(request);
```

30



LAB

Exposing SOAP Endpoints using
Spring Web Services



Copyright 2007-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.



Spring JMS

Simplifying Messaging Applications



Topics in this Session



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



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

A blurred background image of green grass blades.

3

JMS Core Components

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

A blurred background image of green grass blades.

4

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



- 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");
```

7

- 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();
}
```

8

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);
```

9

Producers and Consumers



- The Session is also responsible for creating instances of MessageProducer and MessageConsumer

```
producer = session.createProducer(someDestination);  
  
consumer = session.createConsumer(someDestination);
```

10

Topics in this Session



- Introduction to JMS
- **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"/>
```



13

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"/>
```



14

Topics in this Session



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



15

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



16

-
- 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



19

DestinationResolver



-
- 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 JMSException;
```



20

- 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>
```

21

Topics in this Session

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

22

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

23

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);
```

24

Sending Messages with Callbacks



- When more control is needed, use callbacks

```
public void send(MessageCreator messageCreator);  
public Object execute(ProducerCallback action);  
public Object execute(SessionCallback action);  
  
Message createMessage(Session session) {...}
```

25

Creating the queue reference yourself using a callback



```
jmsTemplate.execute(new SessionCallback() {  
  
    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;  
    }  
});
```

26

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



Synchronous Message Reception

-
- The JmsTemplate can receive messages also, but these are blocking methods
 - 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  
}
```

29

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

30

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 need to implement MessageListener or SessionAwareMessageListener
- jms:listener-container allows for tweaking of task execution strategy, concurrency, container type, transaction manager and more

31

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"
    method="order"
    destination="queue.orders"
    response-destination="queue.confirmation"/>
```

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

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

32



LAB

Sending and Receiving Messages in a Spring Environment



Copyright 2006-2008 SpringSource. Copying, publishing or distributing without express written permission is prohibited.

Spring JMX

Management and Monitoring of Java
Applications



Topics in this session

- **Introduction to JMX**
- JMX Architecture
- Introducing Spring JMX
- Explicitly exporting beans with Spring
- Automatically exporting existing MBeans



Goals of 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
 - A Java Bean with a management interface



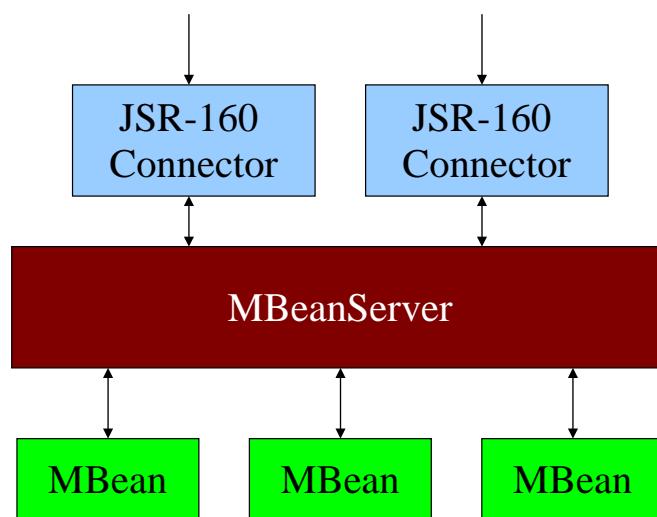
Topics in this session



- Introduction to JMX
- **JMX Architecture**
- Introducing Spring JMX
- Explicitly exporting beans with Spring
- Automatically exporting existing MBeans

5

JMX Architecture



6

- An MBean is a standard Java Bean with an additional management interface
 - Attributes
 - where getters and setters define readability and writeability
 - Operations
- The management interface can be defined statically with a Java interface or defined dynamically at runtime



- 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
 - This key is referred to as an ObjectName



- External communication with an MBeanServer is accomplished using a JSR-160 connector
- This specification defines a generic API for communicating with an MBeanServer
- The only protocol required by the spec is RMI
 - But there are additional protocols including:
 - SOAP
 - Hessian
 - IPC



Topics in this session

- Introduction to JMX
- JMX Architecture
- **Introducing Spring JMX**
- Explicitly exporting beans with Spring
- Automatically exporting existing MBeans



- 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



- Exposing existing Spring beans as MBeans
 - Transparently without changing any Java code
 - Declaratively using Spring bean definitions
- Configuring JMX infrastructure
 - Declaratively using FactoryBeans
- Consuming JMX managed beans
 - Transparently using a proxy-based mechanism



Topics in this session



- Introduction to JMX
- JMX Architecture
- Introducing Spring JMX
- **Explicitly exporting beans with Spring**
- Automatically exporting existing MBeans



Creating an MBeanServer



-
- To create an MBeanServer declaratively, define a bean

```
<bean id="mBeanServer"
      class="org.springframework.jmx.support.
      MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true"/>
</bean>
```



- 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



15

Exporting a bean as an MBean

- Start with an existing POJO bean

```
<bean id="messageService" class="MessageService"/>
```

- Use the MBeanExporter to export it

```
<bean class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <util:map>
            <entry key="service:name=messageService"
                  value-ref="messageService"/>
        </util:map>
    </property>
</bean>
```



16

- In many cases the default strategies for creating the management interface and setting the ObjectName are not sufficient
- The MBeanExporter allows alternative strategies to be injected
 - ObjectNamingStrategy
 - MBeanInfoAssembler



17

ObjectNamingStrategy

- ObjectNames are controlled by implementations of the ObjectNamingStrategy interface
- Three included implementations
 - KeyNamingStrategy
 - Default implementation
 - Uses the key passed to the MBeanExporter by default
 - Can be configured to look in a properties file
 - IdentityNamingStrategy
 - Generates an ObjectName dynamically based on the JVM identity of a bean
 - MetadataNamingStrategy
 - Reads an ObjectName from source-level metadata



18

ObjectNamingStrategy



```
<bean class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <util:map>
            <entry key="not.important"
                  value-ref="messageService"/>
        </util:map>
    </property>
    <property name="namingStrategy">
        <bean class="org.springframework.jmx.export.naming.
                    IdentityNamingStrategy"/>
    </property>
</bean>
```

19

MBeanInfoAssembler



- Management interfaces are controlled by implementations of the MBeanInfoAssembler interface
 - Builds on the JMX standard ModelMBean infrastructure
 - Allows for the full feature set of JMX to be exposed
 - Easily extensible for your own requirements

20

- Four included implementations
 - SimpleReflectiveMBeanInfoAssembler
 - Default implementation
 - Exposes all public properties and methods
 - MethodNameBasedMBeanInfoAssembler
 - Supports declarative selection of methods in a Spring configuration file
 - Extremely easy to configure
 - Allows per instance configuration



21

- InterfaceBasedMBeanInfoAssembler
 - Supports management interface definition using Java interfaces
 - Extends the Standard MBean interface semantics
 - Use multiple interfaces per bean
 - Beans do not need to implement the management interface directly
- MetadataMBeanInfoAssembler
 - Define management interfaces using source-level metadata
 - JDK 5.0 Annotations
 - Supports autodetection
 - Any bean marked with the @ManagedResource annotation will be automatically registered with an MBeanServer



22

```
<bean class="org.springframework.jmx.export.MBeanExporter">
...
<property name="assembler">
<bean class="org.springframework.jmx.export assembler.
InterfaceBasedMBeanInfoAssembler">
<property name="interfaceMappings">
<util:map>
<entry key="service:name=messageService"
value="infoassembler.MessageCapable" />
</util:map>
</property>
</bean>
</property>
</bean>
```

23

Topics in this session

- Introduction to JMX
- JMX Architecture
- Introducing Spring JMX
- Explicitly exporting beans with Spring
- **Automatically exporting existing MBeans**

24

Automatically exporting pre-existing MBeans



- Some beans are Mbeans themselves
 - 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>
```

25

A close-up photograph of green, serrated leaves, likely from a plant like a palm or banana, creating a natural and organic background for the slide.

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 MbeanInfoAssembler to customize the management interface
- Using <context:mbean-export> to automatically export pre-existing MBeans

26

A close-up photograph of green, serrated leaves, likely from a plant like a palm or banana, continuing the natural theme from the previous slide.



LAB

Adding Management and Monitoring with Spring JMX

