



# Core Spring

Four Day Workshop

Building Enterprise Applications using Spring

Pivotal

# Copyright Notice

Copyright © 2014 Pivotal Software, Inc. All rights reserved. This manual and its accompanying materials are protected by U.S. and international copyright and intellectual property laws.

Pivotal products are covered by one or more patents listed at <http://www.pivotal.io/patents>.

Pivotal is a registered trademark or trademark of Pivotal Software, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies. The training material is provided “as is,” and all express or implied conditions, representations, and warranties, including any implied warranty of merchantability, fitness for a particular purpose or noninfringement, are disclaimed, even if Pivotal Software, Inc., has been advised of the possibility of such claims. This training material is designed to support an instructor-led training course and is intended to be used for reference purposes in conjunction with the instructor-led training course. The training material is not a standalone training tool. Use of the training material for self-study without class attendance is not recommended.

These materials and the computer programs to which it relates are the property of, and embody trade secrets and confidential information proprietary to, Pivotal Software, Inc., and may not be reproduced, copied, disclosed, transferred, adapted or modified without the express written approval of Pivotal Software, Inc.

# Welcome to Core Spring

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

# Logistics

- Participants list
- Self introduction
- MyLearn registration
- Courseware
- Internet access
- Working hours
- Lunch
- Toilets
- Fire alarms
- Emergency exits
- Other questions?



# How You will Benefit

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



# Covered in this section

- **Agenda**
- Spring and Pivotal

# Course Agenda: Day 1

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



# Course Agenda: Day 2

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

2

# Course Agenda: Day 3

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

3

# Course Agenda: Day 4

- Rapidly start new projects with Spring Boot
- Securing web applications with Spring Security
- Implementing REST with Spring MVC
- Simplifying message applications with Spring JMS



# Covered in this section

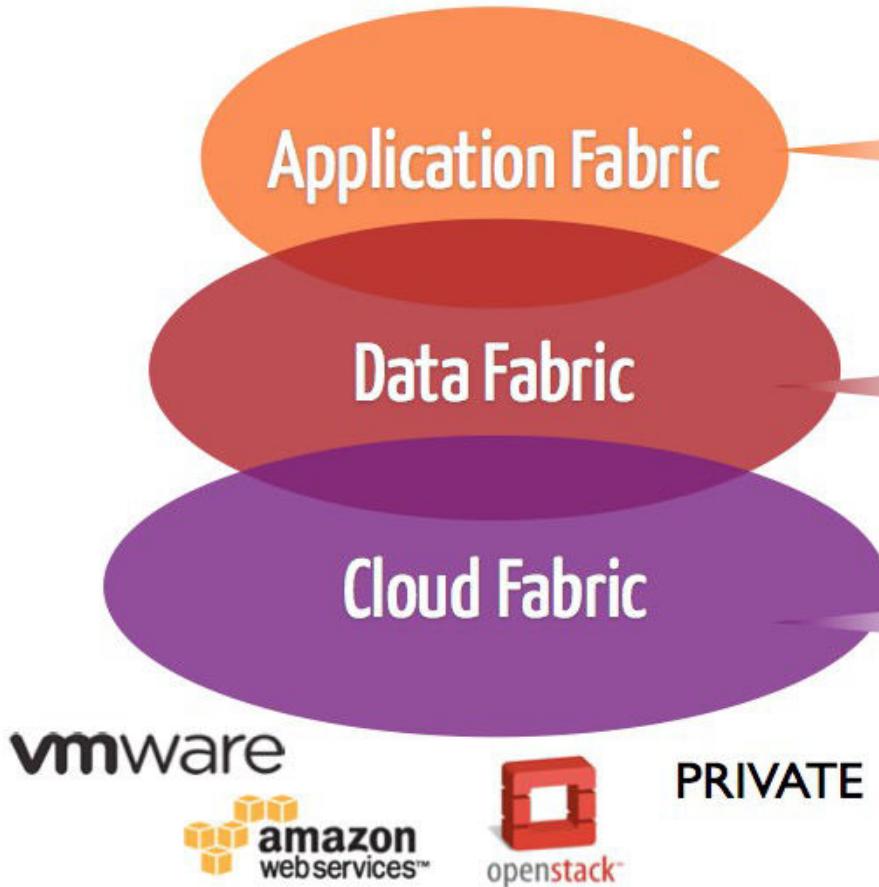
- Agenda
- **Spring and Pivotal**

# Spring and Pivotal

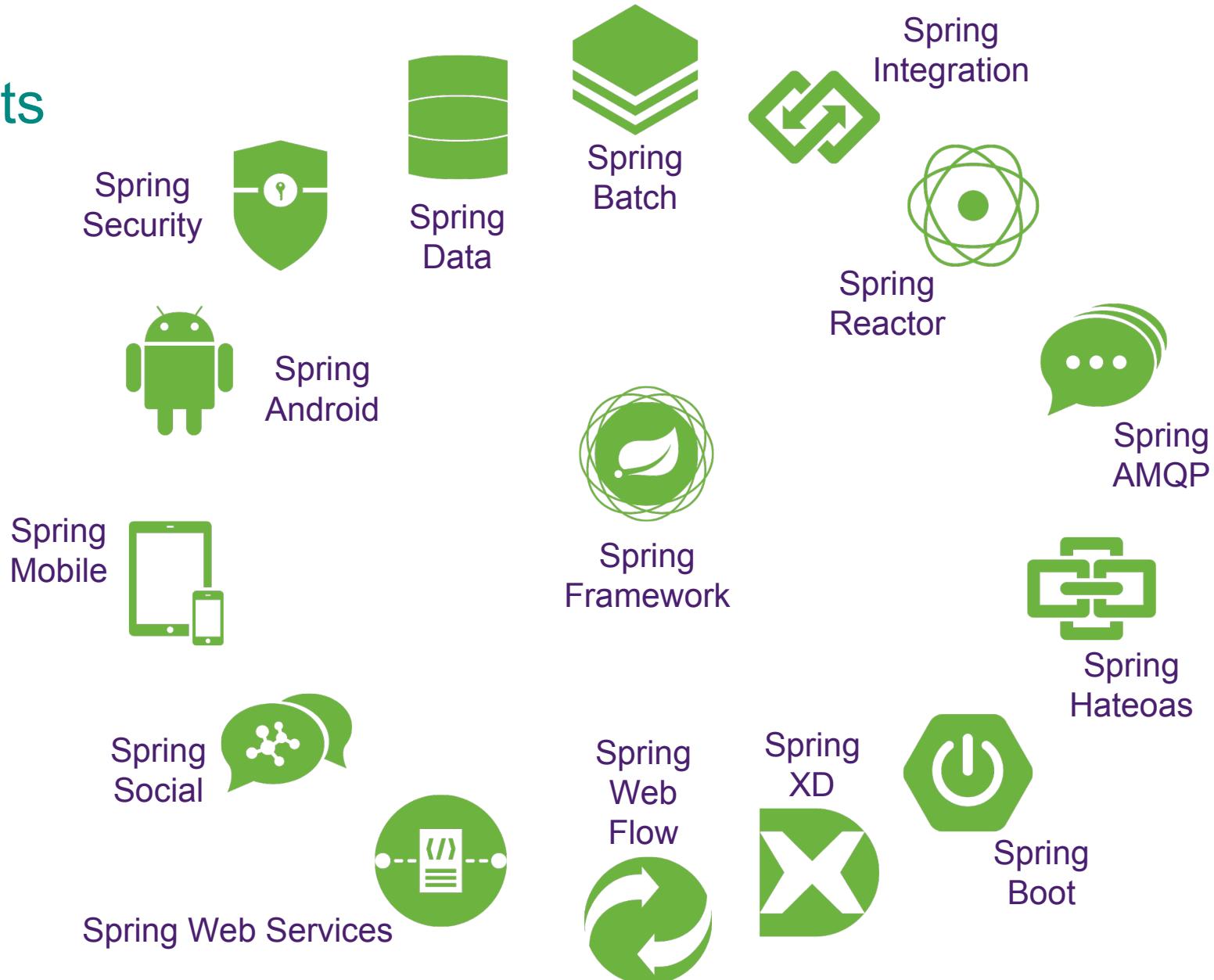
- SpringSource, the company behind Spring
  - acquired by VMware in 2009
  - transferred to Pivotal joint venture 2013
- Spring projects key to Pivotal's big-data and cloud strategies
  - Virtualize your Java Apps
    - Save license cost
    - Deploy to private, public, hybrid clouds
  - Real-time analytics
    - Spot trends as they happen
    - Spring Data, Spring Hadoop, Spring XD & Pivotal HD



# The Pivotal Platform



# Spring Projects



# Open Source contributions

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



# Covered in this section

- Agenda
- Spring and Pivotal

Let's get on with the course..!



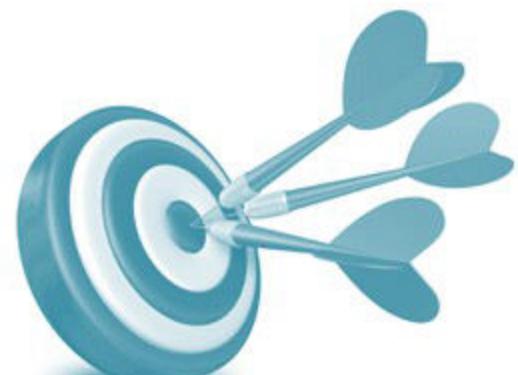
# Overview of the Spring Framework

Introducing Spring in the Context of Enterprise Application Architecture

What is Spring and why would you use it?

# Objectives

- After completing this lesson, you should be able to:
  - Define the Spring Framework
  - Understand what Spring is used for
  - Understand why Spring is successful
  - Explain where it fits in your world



# Topics in this session

- **What is the Spring Framework?**
- Spring is a Container
- Spring Framework history
- What is Spring Used For?

# What is the Spring Framework?

- Spring is an Open Source, Lightweight, Container and Framework for building Java enterprise applications



- Open Source
- Lightweight
- Container
- Framework

# What is the Spring Framework?

## Open Source



- Spring binary and source code is freely available
- Apache 2 license
- Code is available at:
  - <https://github.com/spring-projects/spring-framework>
- Binaries available at Maven Central
  - <http://mvnrepository.com/artifact/org.springframework>
- Documentation available at:
  - <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle>

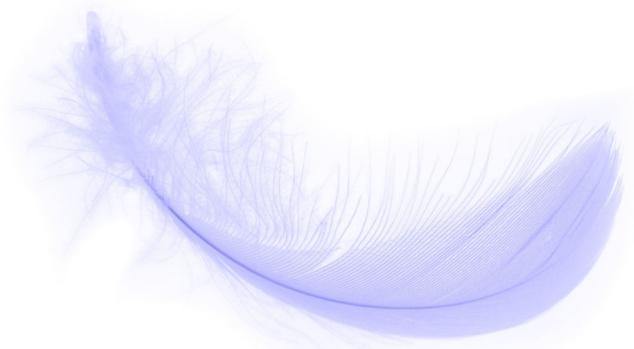


The use of a transitive dependency management system (Maven, Gradle, Ant/Ivy) is recommended for any Java application

# What is the Spring Framework?

## Lightweight

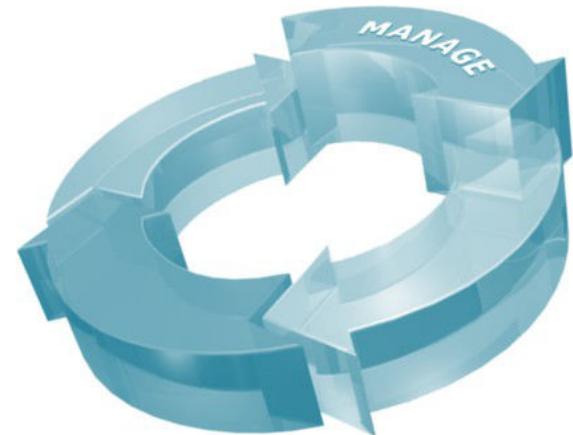
- Spring applications do not require a Java EE application server
  - But they can be deployed on one
- Spring is not *invasive*
  - Does not require you to extend framework classes or implement framework interfaces for most usage
  - You write your code as POJOs
- Spring jars are relatively small
  - Spring jars used in this course are < 8 MB



# What is the Spring Framework?

## Container

- Spring serves as a container for your application objects.
  - Your objects do not have to worry about finding / connecting to each other.
- Spring instantiates and dependency injects your objects
  - Serves as a lifecycle manager



# What is the Spring Framework?

## Framework

- Enterprise applications must deal with a wide variety of technologies / resources
  - JDBC, JMS, AMQP, Transactions, ORM / JPA, NoSQL, Security, Web, Tasks, Scheduling, Mail, Files, XML/JSON Marshalling, Remoting, REST services, SOAP services, Mobile, Social, ...
- Spring provides framework classes to simplify working with lower-level technologies

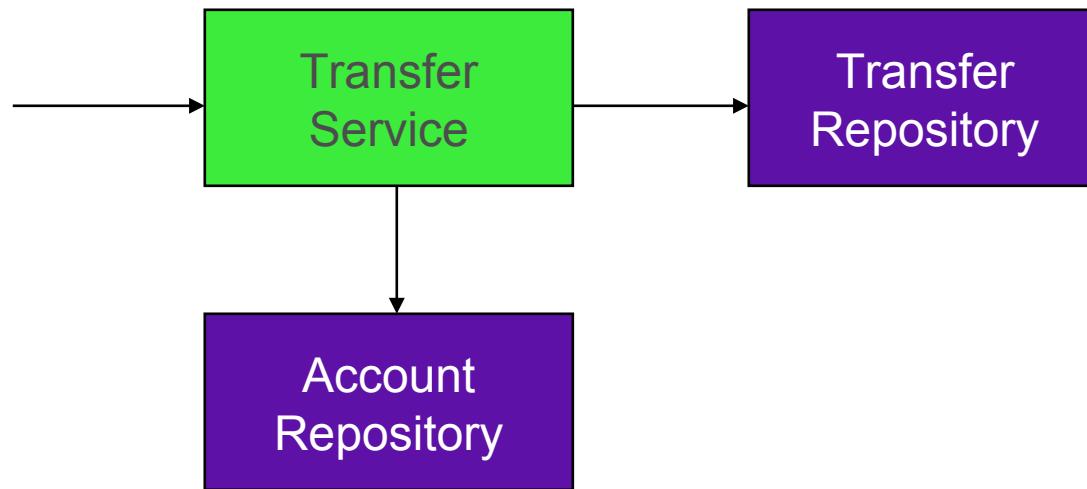


# Topics in this session

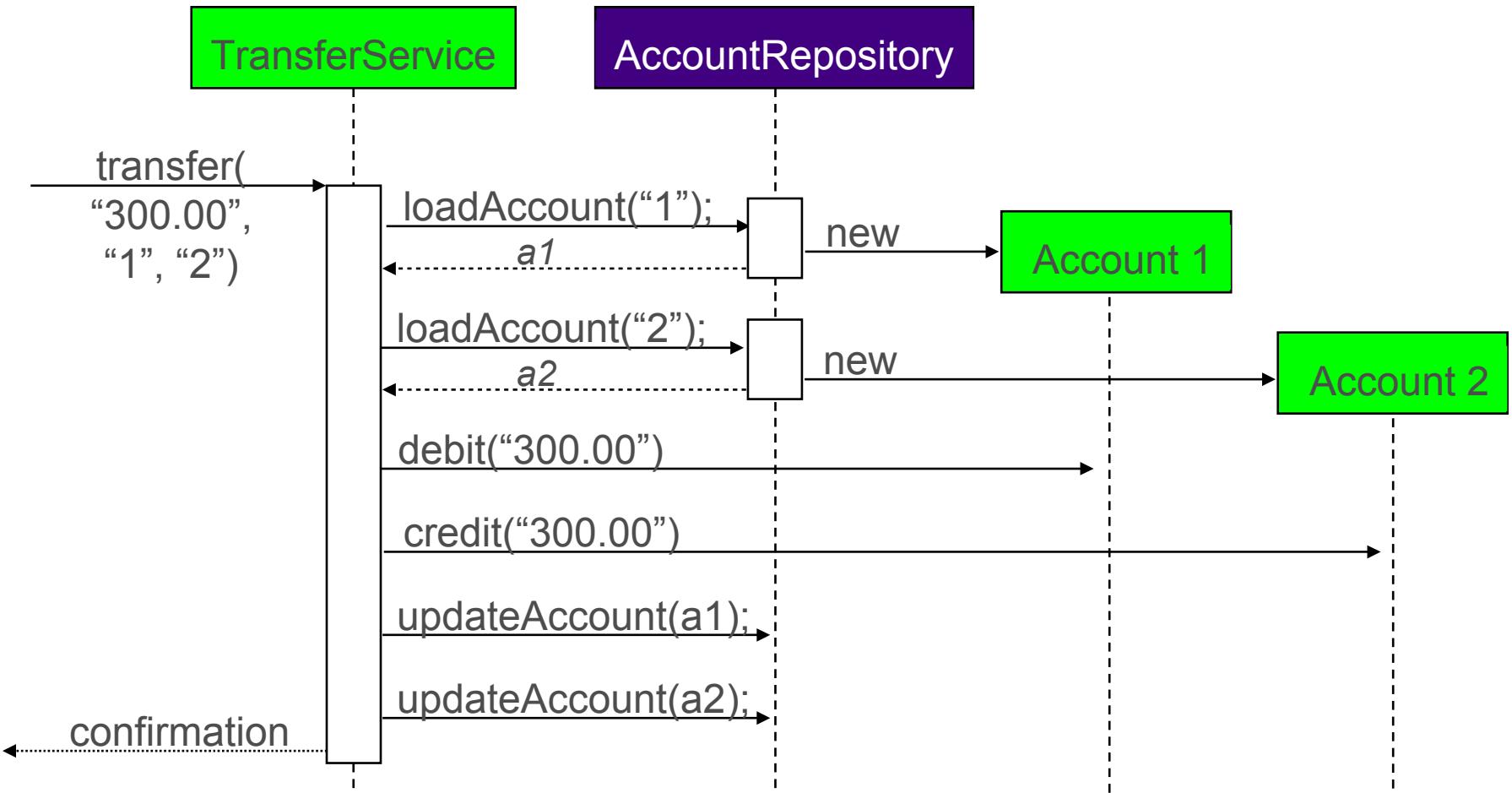
- What is the Spring Framework?
- **Spring is a Container**
- Spring Framework History
- What is Spring Used For?

# Application Configuration

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



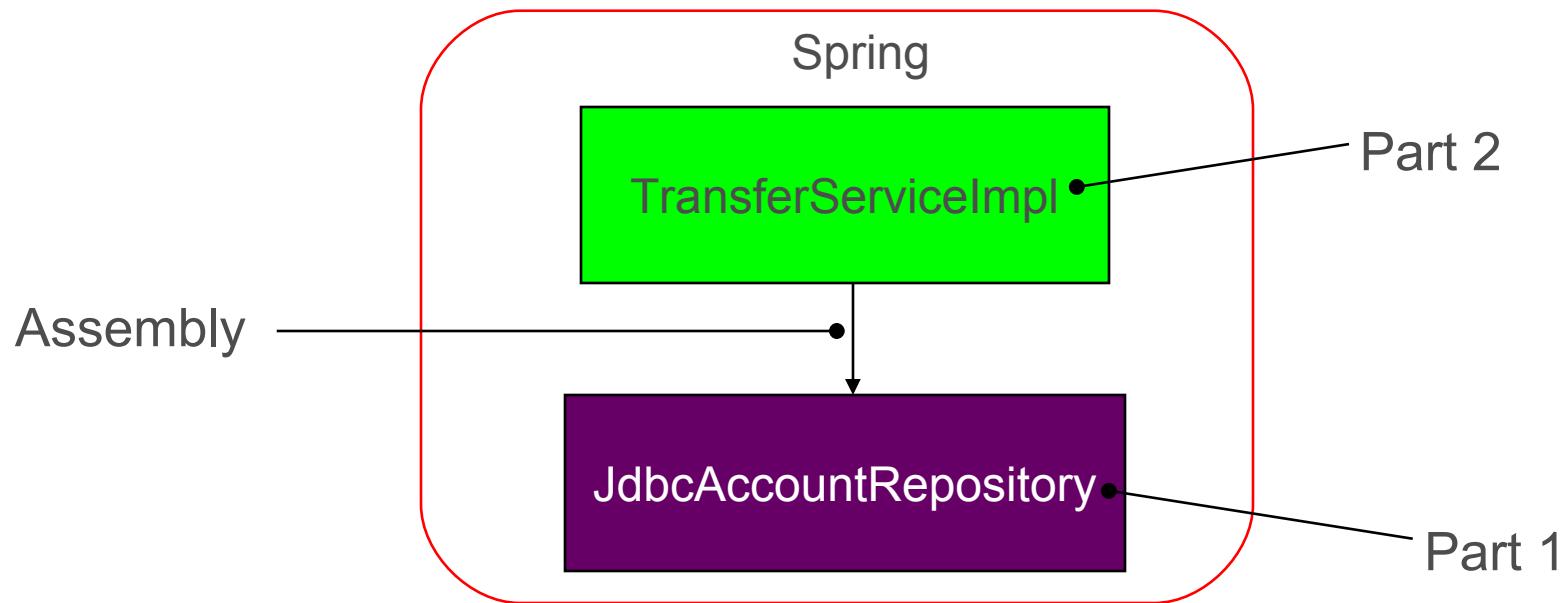
# Example: Money Transfer System



# Spring's Configuration Support

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

# Money Transfer System Assembly



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

# Parts are Just Plain Old Java Objects

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

Implements a service/business interface

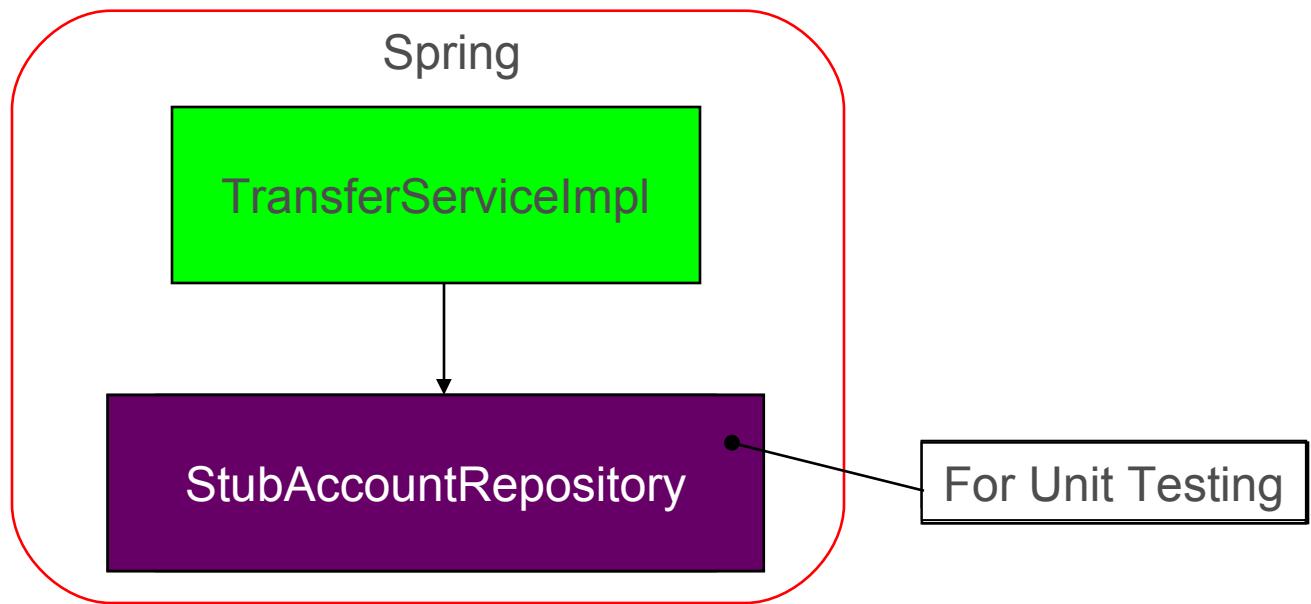
Part 1

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

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

Part 2

# Swapping Out Part Implementations



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

# Topics in this session

- What is the Spring Framework?
- Spring is a Container
- **Spring Framework History**
- What is Spring Used For?

# Why is Spring Successful?

## A brief history of Java

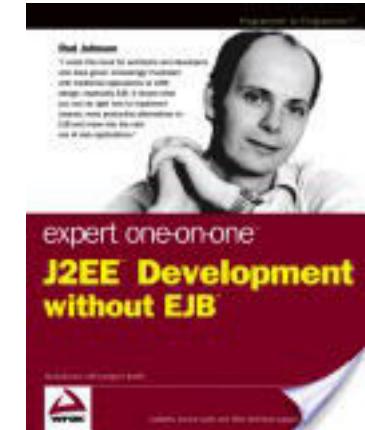
- The early years:
  - 1995 – Java introduced, Applets are popular
  - 1997 – Servlets introduced
    - Efficient, dynamic web pages become possible.
  - 1999 – JSP introduced
    - Efficient, dynamic web pages become easy.
- Questions arise regarding “Enterprise” applications
  - How should a Servlet / JSP application handle:
    - Persistence?
    - Transactions?
    - Security?
    - Business Logic?
    - Messaging?
    - Etc.?

# Introducing J2EE and EJB

- Java's answer: J2EE
  - 1999 – J2EE introduced
    - Featuring Enterprise Java Beans (EJB)
    - Answers the questions of persistence, transactions, business logic, security, etc
- However EJBs prove to be problematic:
  - Difficult to code.
    - Must extend / implement specific classes /interfaces
    - Complicated programming model required
  - Difficult to unit test
  - Expensive to run
    - Must have application server, resource intensive

# The Birth of Spring

- Rod Johnson publishes J2EE Development without EJB
- 2004 - Spring Framework 1.0 released
  - Champions dependency injection
  - Encourages POJOs
  - Uses XML files to describe application configuration
  - Becomes popular quickly as a EJB alternative



# Spring Framework History

- Spring 2.0 (2006):
  - XML simplification, async JMS, JPA, AspectJ support
- Spring 2.5 (2007, currently 2.5.6)
  - Requires Java 1.4+ and supports JUnit 4
  - Annotation DI, @MVC controllers, XML namespaces
- Spring 3.x (3.2 released Dec 2012)
  - Env. + Profiles, @Cacheable, @EnableXXX ...
  - Supports Java 7, Hibernate 4, Servlet 3
  - Requires Java 1.5+ and JUnit 4.7+
  - REST support, JavaConfig, SpEL, more annotations
- Spring 4 (released Dec 2013)
  - Support for Java 8, @Conditional, Web-sockets

# Topics in this session

- What is the Spring Framework?
- Spring is a Container
- Spring Framework History
- **What is Spring Used For?**

# What is Spring Used For?

- Spring provides comprehensive infrastructural support for developing enterprise Java™ applications
  - Spring deals with the plumbing
  - So you can focus on solving the domain problem
- Spring used to build enterprise applications dealing with:



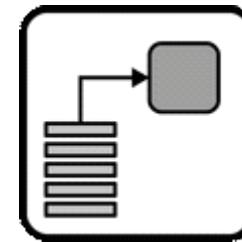
Web Interfaces



Messaging



Persistence



Batch

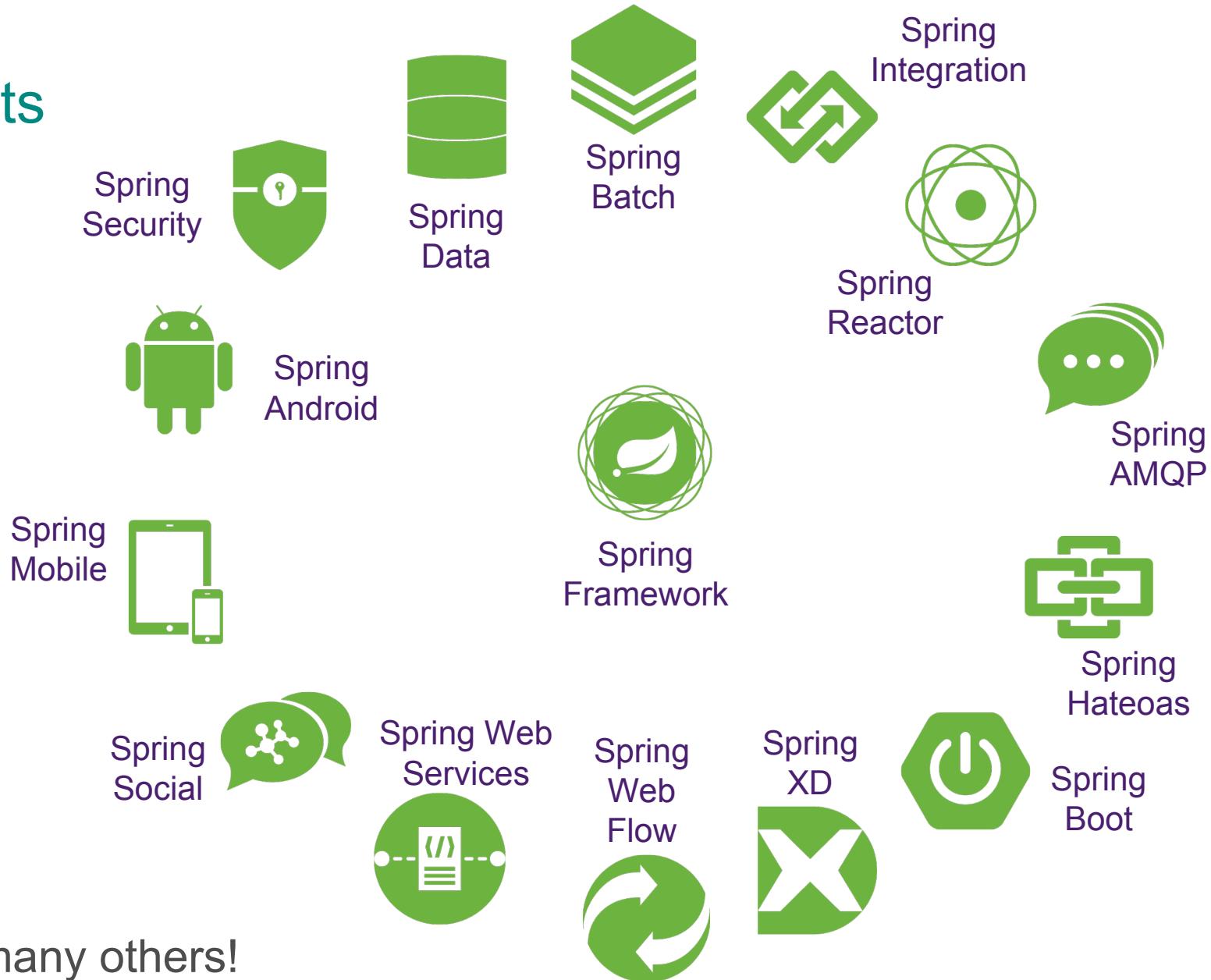


Integration

# The Current World

- Spring is not simply an alternative to J2EE / EJB
  - Modern application development challenges are different today than 2000
- Spring continues to innovate
  - Web
    - AJAX, WebSockets, REST, Mobile, Social
  - Data
    - NoSQL, Big Data, stream processing
  - Cloud
    - Distributed systems
  - Productivity
    - Spring Boot
  - Etc. etc.

# Spring Projects



# Lab

Developing an Application from Plain Java Objects

# Dependency Injection Using Spring

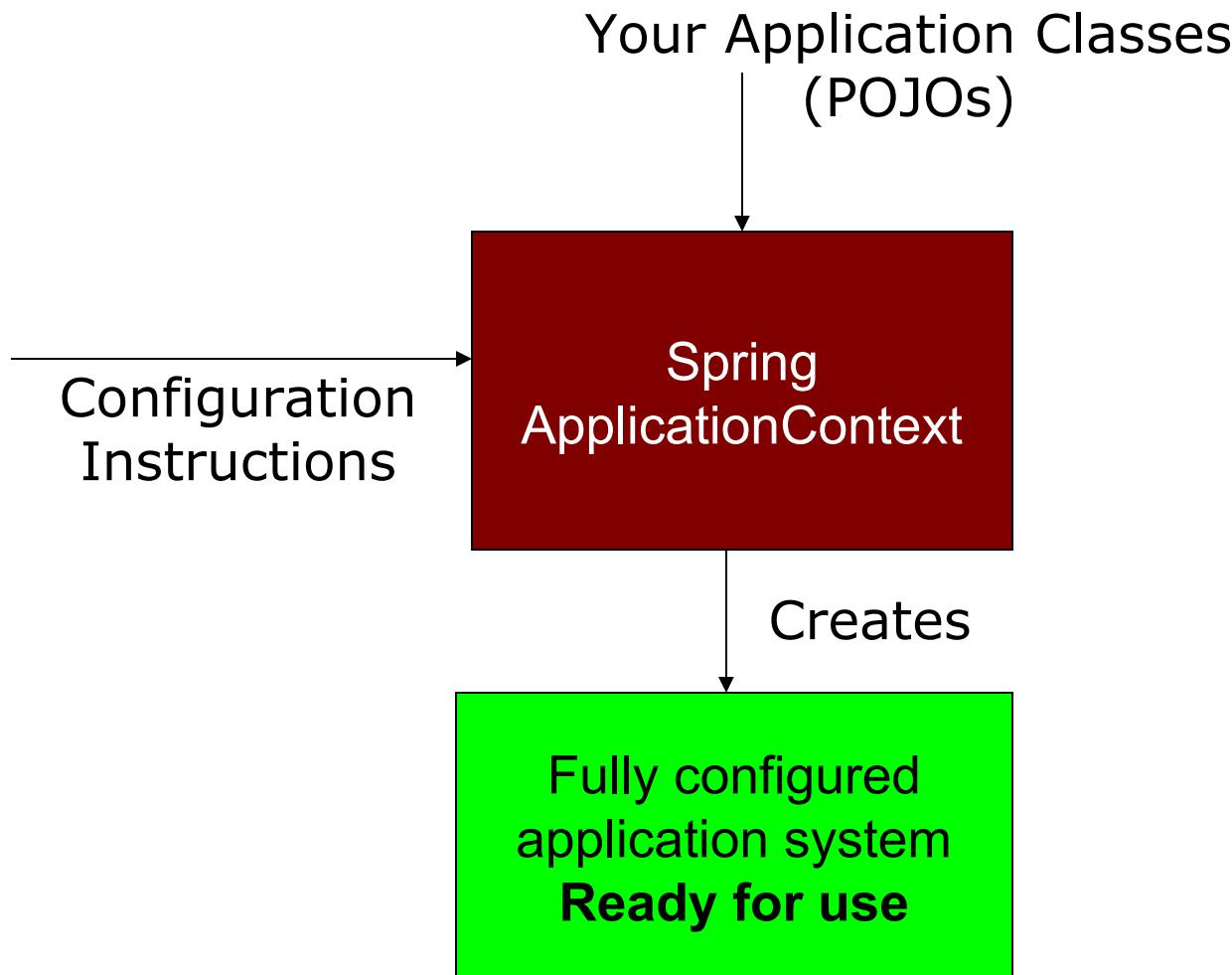
Introducing the Spring Application Context  
and Spring's Java Configuration capability

@Configuration and ApplicationContext

# Topics in this session

- **Spring quick start**
- Creating an application context
- Bean scope
- Lab

# How Spring Works



# Your Application Classes

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



Needed to perform money transfers between accounts

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



Needed to load accounts from the database

# Configuration Instructions

```
@Configuration  
public class ApplicationConfig {  
    @Bean public TransferService transferService() {  
        return new TransferServiceImpl( accountRepository() );  
    }  
    @Bean public AccountRepository accountRepository() {  
        return new JdbcAccountRepository( dataSource() );  
    }  
    @Bean public DataSource dataSource() {  
        DataSource dataSource = new BasicDataSource();  
        dataSource.setDriverClassName("org.postgresql.Driver");  
        dataSource.setUrl("jdbc:postgresql://localhost/transfer");  
        dataSource.setUser("transfer-app");  
        dataSource.setPassword("secret45");  
        return dataSource;  
    }  
}
```

# Creating and Using the Application

```
// Create the application from the configuration  
ApplicationContext context =  
    SpringApplication.run( AppConfig.class );
```

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

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

Bean ID  
Based on method name

# Accessing a Bean

- Multiple ways

```
ApplicationContext context = SpringApplication.run(...);
```

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

```
TransferService ts1 = (TransferService) context.getBean("transferService");
```

```
// Use typed method to avoid cast
```

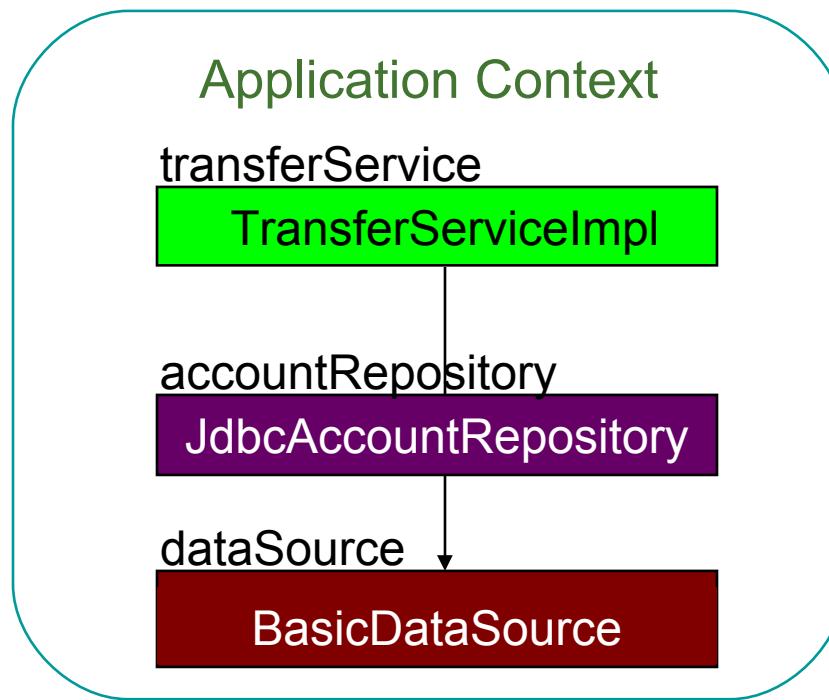
```
TransferService ts2 = context.getBean("transferService", TransferService.class);
```

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

```
TransferService ts3 = context.getBean(TransferService.class );
```

# Inside the Spring Application Context

```
// Create the application from the configuration  
ApplicationContext context =  
    SpringApplication.run( AppConfig.class )
```



# Quick Start Summary

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

# Topics in this session

- Spring quick start
- **Creating an application context**
- Multiple Configuration Files
- Bean scope
- Lab

# Creating a Spring Application Context

- Spring application contexts can be bootstrapped in any environment, including
  - JUnit system test
  - Web application
  - Standalone application

# Example: Using an Application Context Inside a JUnit System Test

```
public class TransferServiceTests {  
    private TransferService service;  
  
    @Before public void setUp() {  
        // Create the application from the configuration  
        ApplicationContext context =  
            SpringApplication.run( AppConfig.class );  
        // Look up the application service interface  
        service = context.getBean(TransferService.class);  
    }  
}
```

Bootstraps the system to test

```
@Test public void moneyTransfer() {  
    Confirmation receipt =  
        service.transfer(new MonetaryAmount("300.00"), "1", "2"));  
    Assert.assertEquals(receipt.getNewBalance(), "500.00");  
}
```

Tests the system

# Topics in this session

- Spring quick start
- Creating an application context
- **Multiple Configuration Files**
- Bean scope
- Lab

# Creating an Application Context from Multiple Files

- Your `@Configuration` class can get very long
  - Instead use *multiple* files combined with `@Import`
  - Defines a single Application Context
    - With beans sourced from multiple files

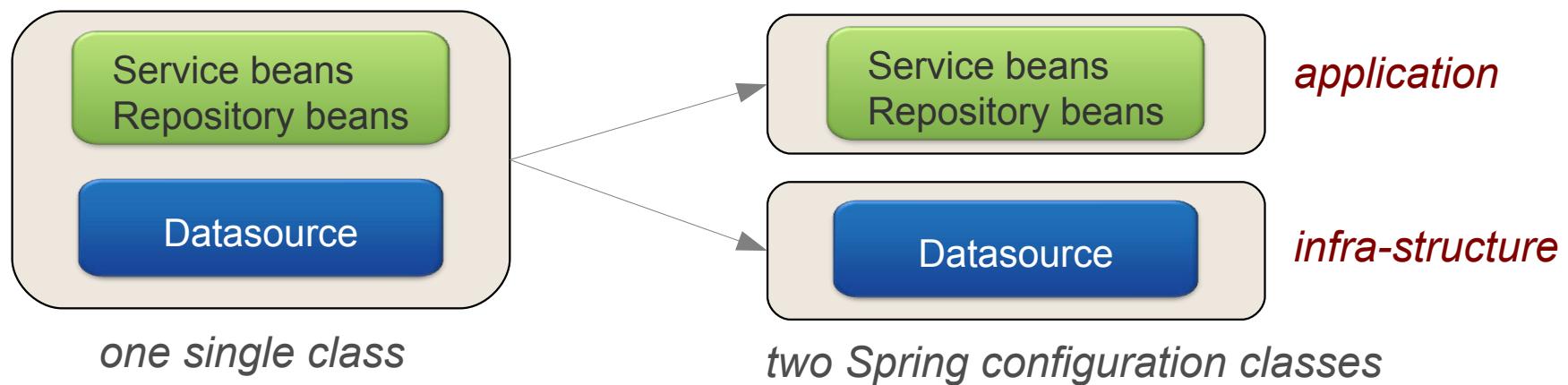
```
@Configuration  
@Import({InfrastructureConfig.class, WebConfig.class })  
public class ApplicationConfig {  
    ...  
}
```

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

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

# Creating an Application Context from Multiple Files

- Organize your `@Configuration` classes however you like
- Best practice: separate out “application” beans from “infrastructure” beans
  - Infrastructure often changes between environments



# Mixed Configuration

```
@Configuration  
public class ApplicationConfig {  
  
    @Bean public TransferService transferService()  
    { return new TransferServiceImpl( accountRepository() ); }  
  
    @Bean public AccountRepository accountRepository()  
    { return new JdbcAccountRepository( dataSource() ); }  
  
    @Bean public DataSource dataSource()  
    {  
        DataSource dataSource = new BasicDataSource();  
        dataSource.setDriverClassName("org.postgresql.Driver");  
        dataSource.setUrl("jdbc:postgresql://localhost/transfer" );  
        dataSource.setUser("transfer-app");  
        dataSource.setPassword("secret45" );  
        return dataSource;  
    }  
}
```

*application beans*

Coupled to a local Postgres environment

*infrastructure bean*

# Partitioning Configuration

```
@Configuration  
public class ApplicationConfig {  
    @Autowired DataSource dataSource;  
  
    @Bean public TransferService transferService() {  
        return new TransferServiceImpl( accountRepository() );  
    }  
  
    @Bean public AccountRepository accountRepository() {  
        return new JdbcAccountRepository( dataSource );  
    }  
}
```

*application beans*

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

*infrastructure bean*

# Referencing beans defined in another file

- Use `@Autowired` to reference bean defined in a separate configuration file:

```
@Configuration  
@Import( InfrastructureConfig.class )  
public class ApplicationConfig {
```

```
@Autowired  
DataSource dataSource;
```

```
@Configuration  
public class InfrastructureConfig {  
    @Bean  
    public DataSource dataSource() {  
        DataSource ds = new BasicDataSource();  
        ...  
        return ds;  
    }  
}
```

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

Or auto-wire a property setter, can't use a constructor

# Referencing beans defined in another file

- Alternative: Define @Bean method parameters
  - Spring will find bean that matches the type and populate the parameter

```
@Configuration  
@Import( InfrastructureConfig.class )  
public class ApplicationConfig {  
    @Bean  
    public AccountRepository accountRepository( DataSource dataSource ) {  
        return new JdbcAccountRepository( dataSource );  
    }  
}
```

```
@Configuration  
public class InfrastructureConfig {  
    @Bean  
    public DataSource dataSource() {  
        DataSource ds = new BasicDataSource();  
        ...  
        return ds;  
    }  
}
```

# Topics in this session

- Spring quick start
- Creating an application context
- Multiple Configuration Files
- **Bean scope**
- Lab

# Bean Scope: default

- Default scope is *singleton*

service1 == service2

```
@Bean  
public AccountService accountService() {  
    return ...  
}
```

```
@Bean  
@Scope("singleton")  
public AccountService accountService() {  
    return ...  
}
```

One single instance

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

# Bean Scope: prototype

service1 != service2

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

```
@Bean  
@Scope("prototype")  
public AccountService accountService() {  
    return ...  
}
```

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

2 instances

# Available Scopes

**singleton**

A single instance is used

**prototype**

A new instance is created each time the bean is referenced

**session**

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

**request**

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

**custom  
scope  
name**

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

# Dependency Injection Summary

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

# Lab

Using Spring to Configure an Application

# Dependency Injection Using Spring 2

Deeper Look into Spring's Java  
Configuration Capability

External Properties, Profiles and Proxies

# Topics in this session

- External Properties
- Profiles
- Proxying

# Setting property values

- Consider this bean definition from the last chapter:

```
@Bean  
public DataSource dataSource() {  
    DataSource ds = new BasicDataSource();  
    ds.setDriverClassName("org.postgresql.Driver");  
    ds.setUrl("jdbc:postgresql://localhost/transfer" );  
    ds.setUser("transfer-app");  
    ds.setPassword("secret45" );  
    return ds;  
}
```

- Unwise to hard-code DB connection parameters
  - “Externalize” these to a properties file

# Spring's Environment Abstraction – 1

- **Environment** object used to obtain properties from runtime environment
- Properties from many sources:
  - JVM System Properties
  - Java Properties Files
  - Servlet Context Parameters
  - System Environment Variables
  - JNDI

# Spring's Environment Abstraction – 2

```
@Configuration  
public class ApplicationConfig {  
  
    @Autowired public Environment env;  
  
    @Bean public DataSource dataSource() {  
        DataSource ds = new BasicDataSource();  
        ds.setDriverClassName( env.getProperty("db.driver") );  
        ds.setUrl( env.getProperty("db.url") );  
        ds.setUser( env.getProperty("db.user") );  
        ds.setPassword( env.getProperty("db.password") );  
        return ds;  
    }  
}
```

# Accessing Properties using @Value

```
@Configuration  
public class ApplicationConfig {  
  
    @Bean  
    public DataSource dataSource(  
        @Value("${db.driver}") String dbDriver,  
        @Value("${db.url}") String dbUrl,  
        @Value("${db.user}") String dbUser,  
        @Value("${db.password}") String dbPassword) {  
        DataSource ds = new BasicDataSource();  
        ds.setDriverClassName( dbDriver);  
        ds.setUrl( dbUrl);  
        ds.setUser( dbUser);  
        ds.setPassword( dbPassword ));  
        return ds;  
    }  
}
```

# Property Sources

- Environment obtains values from “property sources”
  - System properties & Environment variables populated automatically
  - Use **@PropertySources** to contribute additional properties
  - Available resource prefixes: classpath: file: http:

```
@Configuration  
@PropertySource ( "classpath:/com/organization/config/app.properties" )  
public class ApplicationConfig {  
    ...  
}
```

# `{} Placeholders`

- `{} placeholders` in a `@PropertySource` are resolved against existing properties
  - Such as System properties & Environment variables

`@PropertySource ( "classpath:/com/acme/config/app-${ENV}.properties" )`

```
db.driver=org.postgresql.Driver  
db.url=jdbc:postgresql://localhost/transfer  
db.user=transfer-app  
db.password=secret45
```

app-dev.properties

```
db.driver=org.postgresql.Driver  
db.url=jdbc:postgresql://qa/transfer  
db.user=transfer-app  
db.password=secret88
```

app-qa.properties

```
db.driver=org.postgresql.Driver  
db.url=jdbc:postgresql://prod/transfer  
db.user=transfer-app  
db.password=secret99
```

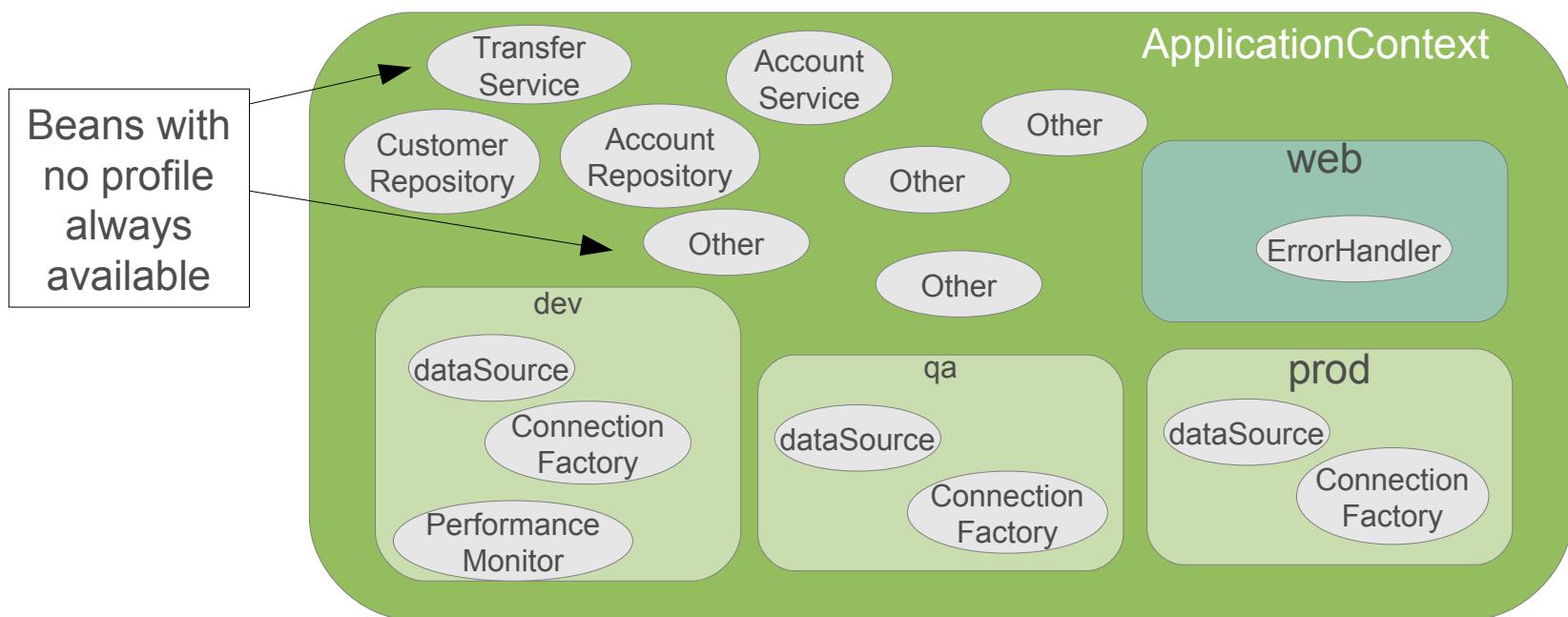
app-prod.properties

# Topics in this session

- External Properties
- **Profiles**
- Proxying

# Profiles

- Beans can be grouped into Profiles
    - Profiles can represent purpose: “web”, “offline”
    - Or environment: “dev”, “qa”, “uat”, “prod”
    - Beans included / excluded based on profile membership



# Defining Profiles – 1

- Using **@Profile** annotation on configuration class
  - All beans in Configuration belong to the profile

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

# Defining Profiles - 2

- Using **@Profile** annotation on **@Bean** methods

```
@Configuration  
public class DataSourceConfig {  
  
    @Bean(name="dataSource") ←  
    @Profile("dev")  
    public DataSource dataSourceForDev() {  
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();  
        return builder.setName("testdb") ...  
    }  
  
    @Bean(name="dataSource")  
    @Profile("prod")  
    public DataSource dataSourceForProd() {  
        DataSource dataSource = new BasicDataSource();  
        ...  
        return dataSource;  
    }  
}
```

Explicit bean-name  
overrides method name

Both profiles define  
*same* bean id, so only  
one profile should be  
activated at a time.

# Ways to Activate Profiles

- Profiles must be activated at run-time
  - Integration Test: Use `@ActiveProfiles` (covered later)
  - System property

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

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

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

`web.xml`

- Programmatically on `ApplicationContext`
  - Simply set `spring.profiles.active` system property before instantiating `ApplicationContext`.

# Quiz:

## Which of the Following is/are Selected?

-Dspring.profiles.active=jpa

?

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

?

```
@Configuration  
@Profile("jpa")  
public class  
JpaConfig  
{ ...}
```

?

```
@Configuration  
@Profile("jdbc")  
public class  
JdbcConfig  
{ ...}
```

# Property Source selection

- `@Profile` can control which `@PropertySources` are included in the Environment

```
@Configuration  
@Profile("dev")  
@PropertySource ("dev.properties")
```

```
db.driver=org.postgresql.Driver  
db.url=jdbc:postgresql://localhost/transfer  
db.user=transfer-app  
db.password=secret45
```

dev.properties

```
@Configuration  
@Profile("prod")  
@PropertySource ("prod.properties")
```

```
db.driver=org.postgresql.Driver  
db.url=jdbc:postgresql://prod/transfer  
db.user=transfer-app  
db.password=secret99
```

prod.properties

# Topics in this session

- External Properties
- Profiles
- **Proxying**

# Quiz

Which is the best implementation?

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

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

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

Method call

New instance

**Prefer call to dedicated method.  
Let's discuss why ...**

# Working with Singletons

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

Singleton??

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

Method  
called twice  
more

```
@Bean  
public AccountService accountService() {  
    return new AccountServiceImpl( accountRepository() );  
}
```

HOW IS IT POSSIBLE?

# Inheritance-based Proxies

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

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



```
public class AppConfig$$EnhancerByCGLIB$ extends AppConfig {
```

```
    public AccountRepository accountRepository() { // ... }  
    public TransferService transferService() { // ... }
```

```
    ...
```

# Inheritance-based Proxies

- Child class is the entry point

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



Java Configuration uses *cglib* for inheritance-based proxies

# Summary

- Property values are easily externalized using Spring's Environment abstraction
- Profiles are used to group sets of beans
- Spring proxies your @Configuration classes to allow for scope control.

# Annotations in Spring

Annotations for Dependency Injection and  
Interception

Component scanning and auto-injection

# Topics in this Session

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

# Before – Explicit Bean Definition

- Configuration is external to bean-class
  - *Separation of concerns*
  - Java-based dependency injection

```
@Configuration  
public class TransferModuleConfig {  
    @Bean public TransferService transService() {  
        TransferServiceImpl service = new TransferServiceImpl();  
        service.setAccountRepository(accountRepository());  
        return service;  
    }  
}
```

# After - Implicit Configuration

- Annotation-based configuration *within* bean-class

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

Annotations embedded with POJOs

```
@Configuration  
@ComponentScan ( "com.bank" )  
public class AnnotationConfig {  
    // No bean definition needed any more  
}
```

Find **@Component** classes within designated (sub)packages

# Usage of @Autowired

Unique dependency of  
correct **type** must exist

- Constructor-injection

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

- Method-injection

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

- Field-injection

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

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

# @Autowired dependencies: required or not?

- Default behavior: required

Exception if no dependency found

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

- Use required attribute to override default behavior

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

Only inject *if* dependency exists

# Constructor vs Setter Dependency Injection

- Spring doesn't care – can use either
  - But which is best?
- Follow the same rules as standard Java
  - Be consistent across your project team
  - Many classes use both

Constructors	Setters
Mandatory dependencies	Optional / changeable dependencies
Immutable dependencies	Circular dependencies
Concise (pass several params at once)	Inherited automatically

# Autowiring and Disambiguation

- What happens here?

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

```
@Component  
public class JpaAccountRepository implements AccountRepository {}
```

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

Which one should get injected?

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

# Autowiring and Disambiguation

- Use of the @Qualifier annotation

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

**qualifier**

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

**bean ID**

```
@Component("jpaAccountRepository")
public class JpaAccountRepository implements AccountRepository {..}
```



@Qualifier also available with method injection and field injection  
Component names *should not* show implementation details unless there are 2 implementations of the same interface (as here)

# Component names

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



Common strategy: avoid using qualifiers when possible.  
*Usually rare to have 2 beans of same type in ApplicationContext*

# Java Config vs Annotations syntax

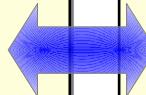
- Similar options are available

```
@Component("transferService")
@Scope("prototype")
@Profile("dev")
@Lazy(false)
public class TransferServiceImpl
    implements TransferService {
    @Autowired
    public TransferServiceImpl
        (AccountRepository accRep) ...
}
```

*Annotations*

```
@Bean
@Scope("prototype")
@Profile("dev")
@Lazy(false)
public TransferService transferService() {
    return
        new TransferServiceImpl(
            accountRepository());
}
```

*Java Configuration*



# Topics in this Session

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

# Component scanning

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

# Best practices

- Really bad:

```
@ComponentScan ( { "org", "com" } )
```

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

- Still bad:

```
@ComponentScan ( "com" )
```

- OK:

```
@ComponentScan ( "com.bank.app" )
```

- Optimized:

```
@ComponentScan ( { "com.bank.app.repository",  
    "com.bank.app.service", "com.bank.app.controller" } )
```

# Topics in this Session

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

# When to use what?

Java

## Java Configuration

- Pros:
  - Is centralized in one (or a few) places
  - Write any Java code you need
  - Strong type checking enforced by compiler (and IDE)
  - Can be used for all classes (not just your own)
- Cons:
  - More verbose than annotations

# When to use what?



## Annotations

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

# Topics in this Session

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

# Mixing Java Config and annotations

- You can mix and match in many ways. A few options:
  - Use annotations for Spring MVC beans
    - Beans that are not referenced elsewhere
  - Use Java Configuration for Service and Repository beans
  - Use annotations when possible, but still use Java Configuration for legacy code that can't be changed

# Topics in this Session

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

# @PostConstruct and @PreDestroy

- Add behavior at startup and shutdown

```
public class JdbcAccountRepository {
```

```
    @PostConstruct
```

```
    void populateCache() { }
```

Method called at startup *after* dependency injection

```
    @PreDestroy
```

```
    void clearCache() { }
```

Method called at shutdown prior to destroying the bean instance

```
}
```



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

# @PostConstruct / @PreDestroy configuration

- @ComponentScan required.

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

JSR-250 annotations  
– EJB3 also uses them

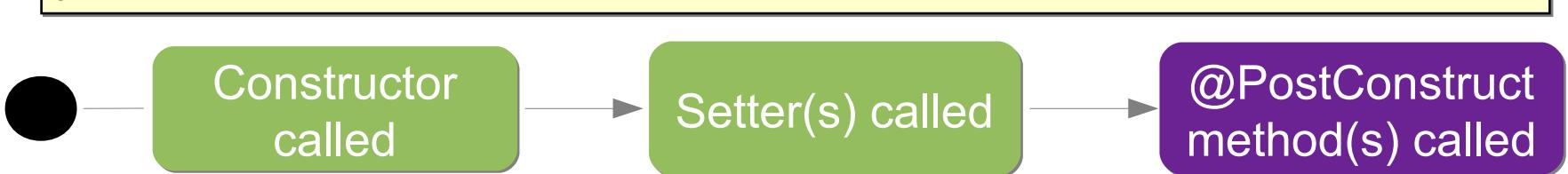
```
@Configuration  
@ComponentScan ( "... " )  
public class AnnotationConfig {  
    ...  
}
```

Automatically enables processing of  
@PostConstruct and @PreDestroy

# @PostConstruct

- Called after setter methods are called

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



# @PreDestroy

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

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

```
// Destroy the application
```

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

Triggers call of **all**  
@PreDestroy annotated methods

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

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



- Called only when ApplicationContext / JVM exit *normally*
- Not called for prototype beans

# Lifecycle Methods via @Bean

- Alternatively, **@Bean** has options to define these life-cycle methods

```
@Bean (initMethod="populateCache", destroyMethod="clearCache")
public AccountRepository accountRepository() {
    // ...
}
```

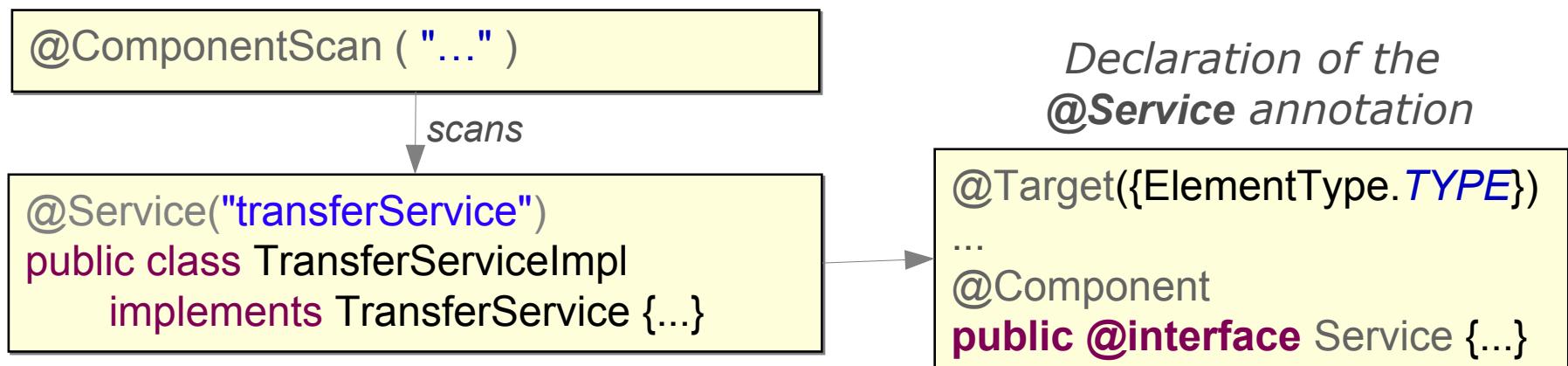
- Common Usage:
  - Use **@PostConstruct/@PreDestroy** for your own classes
  - Use **@Bean** properties for classes you didn't write and can't annotate

# Topics in this Session

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

# Stereotype annotations

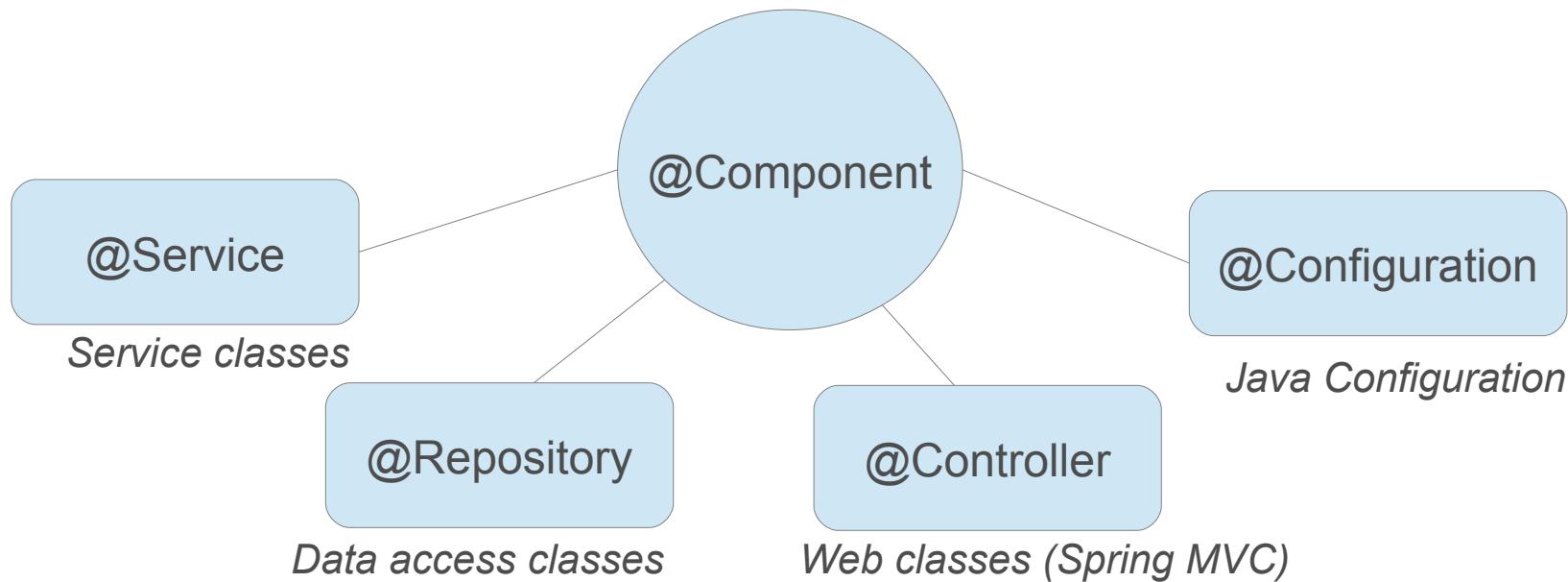
- Component scanning also checks for annotations that are themselves annotated with @Component
  - So-called *stereotype annotations*



@Service annotation is part of the Spring framework

# Stereotype annotations

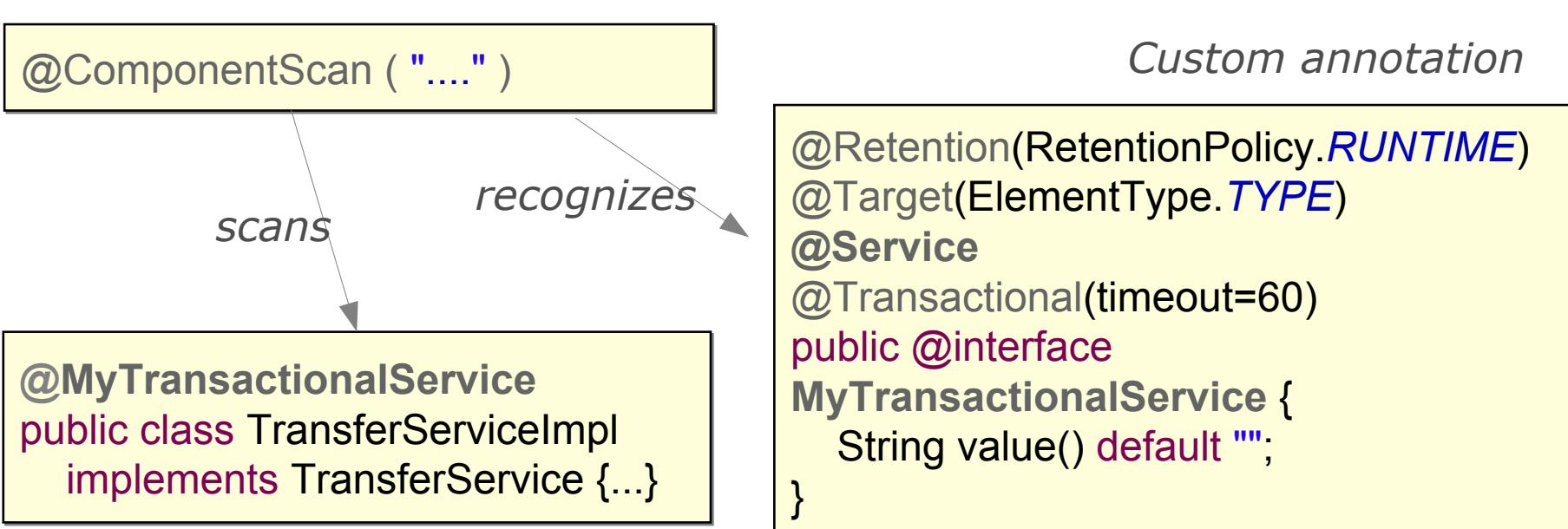
- Spring framework stereotype annotations



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

# Meta-annotations

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



# Lab

Using Spring Annotations  
To Configure and Test an application

# Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices for component-scanning
  - Java Config versus annotations: when to use what?
  - Mixing Java Config and annotations for Dep. Injection
  - `@PostConstruct` and `@PreDestroy`
  - Stereotypes and meta annotations
- Lab
- Advanced features
  - `@Resource`
  - Standard annotations (JSR 330)

# Using @Resource

- From JSR-250, supported by EJB 3.0 and Spring
  - Identifies dependencies by name, not by type
    - Name is Spring bean-name*
    - @Autowired matches by type
  - Supports setter and field injection *only*

```
@Resource(name="jdbcAccountRepository")
public void setAccountRepository(AccountRepository repo) {
    this.accountRepository = repo;
}
```

Setter  
Injection

```
@Resource(name="jdbcAccountRepository")
private AccountRepository accountRepository;
```

Field  
injection

# Qualifying @Resource

- When no name is supplied
  - Inferred from property/field name
  - Or falls back on injection by type
- Example
  - Looks for bean called *accountRepository*
    - because method is **setAccountRepository**
  - Then looks for bean of type *AccountRepository*

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

# Topics in this Session

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

# JSR 330

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

# JSR 330 annotations

Also scans JSR-330 annotations

```
@ComponentScan ( "...." )
```

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

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

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

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

# From @Autowired to @Inject

Spring	JSR 330	Comments
@Autowired	@Inject	@Inject always mandatory, has no required option
@Component	@Named	Spring also scans for @Named
@Scope	@Scope	JSR 330 Scope for meta-annotation and injection points only
@Scope ("singleton")	@Singleton	JSR 330 default scope is like Spring's ' <i>prototype</i> '
@Qualifier	@Named	
@Value	No equivalent	SpEL specific
@Required	Redundant	@Inject <i>always</i> required
@Lazy	No equivalent	Useful when needed, often abused

# Summary

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

# Dependency Injection Using XML

Spring's XML Configuration Language

Introducing Spring's Application Context

# Topics in this session

- **Writing bean definitions in XML**
- Creating an application context
- Controlling Bean Behavior
- Namespaces
- Lab
- Advanced Topics

# XML Configuration

- Original form of Configuration / Dependency Injection
  - Dating back to before 2004
  - Still fully supported
- Most commonly seen in existing applications
  - ...and in older blogs, books, etc.
- External configuration as with Java Config
  - Uses custom XML instead of Java

# @Configuration Comparison

## Java configuration class

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

## XML configuration file

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

# Constructor Injection Configuration

- One parameter

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

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

- Multiple parameters

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

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

Parameters injected according to their type

# Constructor Injection 'Under the Hood'

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

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

Equivalent to:

```
@Bean public Repository repository() {
    return new RepositoryImpl();
}

@Bean public Service service() {
    return new ServiceImpl( repository() );
}
```

# Setter Injection

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

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

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

Equivalent to:

```
@Bean public Repository repository() {  
    return new RepositoryImpl();  
}
```

```
@Bean public Service service() {  
    Service svc = new ServiceImpl();  
    svc.setRepository( repository() );  
    return svc;  
}
```

# Combining Constructor and Setter Injection

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

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

Equivalent to:

```
@Bean public RequiredImpl required() { ... }
@Bean public OptionallImpl optional() { ... }
@Bean public Service service() {
    Service svc = new ServiceImpl( required() );
    svc.setOptional( optional() );
    return svc;
}
```

# Injecting Scalar Values

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

Equivalent

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

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

Equivalent to:

```
@Bean  
public Service service() {  
    Service svc = new ServiceImpl();  
    svc.setStringProperty("foo");  
    return svc;  
}
```

# Automatic Value Type Conversion

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

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

Equivalent to:

```
@Bean public Service service() {
    Service svc = new ServiceImpl();
    int val = // Integer parsing logic, 29.
    svc.setIntProperty( val );
    return svc;
}
```

Spring can convert:

- Numeric types
- BigDecimal,
- boolean: "true", "false"
- Date
- Locale
- Resource

# Topics in this session

- Writing bean definitions in XML
- **Creating an application context**
- Controlling Bean Behavior
- Namespaces
- Lab
- Advanced Topics

# Creating an ApplicationContext using XML

- Use a Java Configuration class, then use `@ImportResource` to specify XML files:

```
SpringApplication.run(MainConfig.class);
```

```
@Configuration
```

```
@ImportResource( {
```

```
    "classpath:com/acme/application-config.xml",
```

```
    "file:C:/Users/alex/application-config.xml" } )
```

```
@Import(DatabaseConfig.class) ←
```

```
public class MainConfig { ... }
```

Multiple files possible

Prefixes allowed

(classpath: (default), file:, http:)

Combine with  
@Configuration  
imports



Older applications may use `ClassPathXmlApplicationContext` or `FileSystemXmlApplicationContext`, still valid, see advanced topics.

# Remember @Import?

```
@Configuration  
@Import(DatabaseConfig.class)  
public class MainConfig {  
    ...  
}
```

- Use <import /> to import other XML configuration files

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

- Uses relative path by default
  - Same prefixes available (file, classpath, http)

# Topics in this session

- Writing bean definitions
- Creating an application context
- **Controlling Bean Behavior**
- Namespaces
- Lab
- Advanced Topics

# Remember @PostConstruct?

```
@PostConstruct
```

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

- Same option available in XML
  - But called “init-method”:

```
<bean id="accountService" class="com.acme.ServiceImpl" init-method="setup">  
    ...  
</bean>
```



Same rules: method can have any visibility, *must take no* parameters, must return *void*. Called after dependency injection.

# Remember @PreDestroy?

```
@PreDestroy
```

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

- Same option available in XML
  - But called “destroy-method”:

```
<bean id="Service" class="com.acme.ServiceImpl" destroy-method="teardown">  
    ...  
</bean>
```



Same rules: method can have any visibility, *must take no* parameters, must return *void*.

# Remember Bean Scope?

```
@Bean  
@Scope("prototype")  
public AccountService accountService() {  
    return ...  
}
```

```
@Component  
@Scope("prototype")  
public class AccountServiceImpl {  
    ...  
}
```

- Same options available in XML
  - singleton, prototype, request, session, (custom)

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

# Remember @Lazy?

```
@Bean  
@Lazy("true")  
public AccountService accountService() {  
    return ...  
}
```

```
@Component  
@Lazy("true")  
public class AccountServiceImpl {  
    ...  
}
```

- Same option available in XML
  - Still not recommended, often misused

```
<bean id="accountService" class="com.acme.ServiceImpl" lazy="true">  
    ...  
</bean>
```

# Remember @Profile?

```
@Configuration  
@Profile("dev")  
public class MainConfig {  
    ...  
}
```

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

- Set profile attribute on the <beans> element
  - Or child element:

The diagram illustrates two nested XML structures. The outer structure is a single horizontal bar containing the opening tag <beans profile="dev">. The inner structure is a vertical stack of bars, each starting with <beans> and ending with </beans>. The first inner bar contains another <beans profile="dev"> tag, which is circled in green. The entire outer bar and its first inner bar are also circled in green.

```
<beans profile="dev">  
    ...  
</beans>  
    <beans>  
        <beans profile="dev">  
            ...  
        </beans>  
    </beans>  
    </beans>
```

# Topics in this session

- Writing bean definitions in XML
- Creating an application context
- Controlling Bean Behavior
- **Namespaces**
- Lab
- Advanced Topics

# Default namespace

- The default namespace in a Spring configuration file is typically the “beans” 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"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd>
<!-- ... -->
</beans>
```



dozens of other namespaces are available!

# Other namespaces

- Defined for subsets of framework functionality<sup>\*</sup>
  - aop (Aspect Oriented Programming)
  - tx (transactions)
  - util
  - jms
  - context
  - ...
- They allow hiding of actual bean definitions
  - Greatly reduce size of bean files (see next slides)

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

# Namespace provides shortcut

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

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



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

- Same functionality, less typing

# property-placeholder example

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

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



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



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

# Power of Namespaces

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

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

hides 1 bean definition

AOP configuration: hides 5+ bean definitions

Transactions configuration: hides more than 15 bean definitions!



tx and aop namespaces will be discussed later

# XML Profiles and Properties

```
<import resource="classpath:config/${current.env}-config.xml"/>  
<context:property-placeholder properties-ref="configProps"/>  
  
{ <beans profile="dev">  
    <util:properties id="configProps" location="config/app-dev.properties">  
  </beans>  
  
<beans profile="prod">  
    <util:properties id="configProps" location="config/app-prod.properties">  
  </beans>
```

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

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

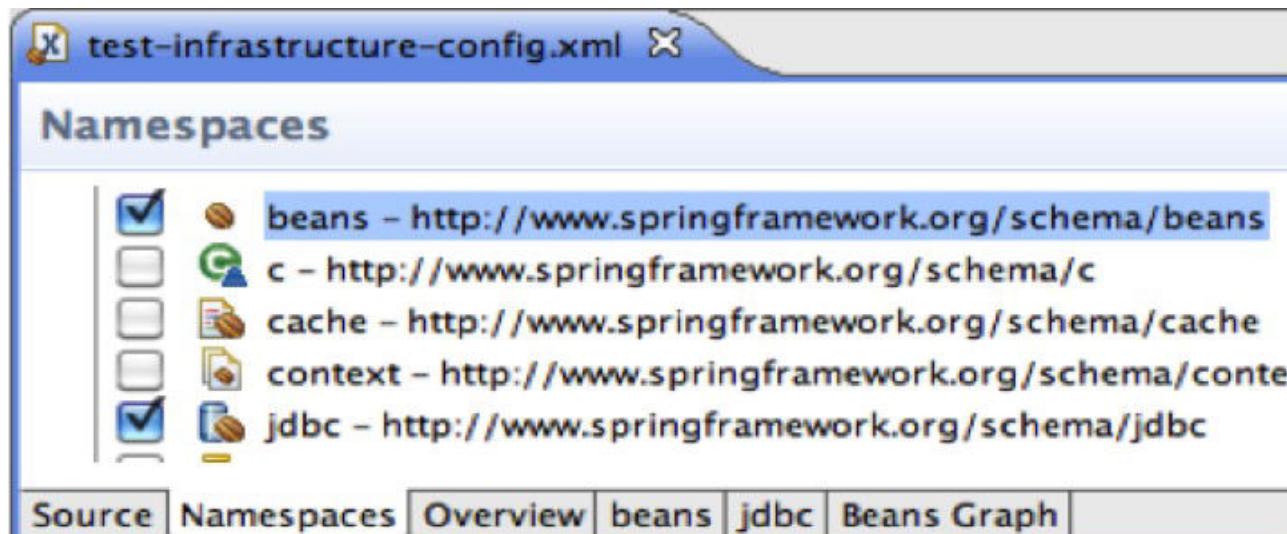
Equivalent to ...

@PropertySource ( "classpath:/com/acme/config/app-\${ENV}.properties" )

# Adding namespace declaration

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

xsi:schemaLocation="..."



Click here and select appropriate namespaces

# Schema version numbers

spring-beans-3.1.xsd

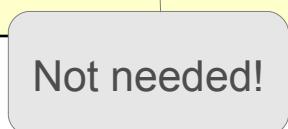
OR

spring-beans.xsd

?

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

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



# Remember @ComponentScan?

```
@Configuration  
@ComponentScan ( { "com.acme.app.repository",  
    com.acme.app.service", "com.acme.app.controller" } )  
public class MainConfig {  
    ...  
}
```

Array of Strings

- Available in the context namespace

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

Single String

# Lab (optional)

Using XML to Configure an Application

# Topics in this session

- Writing bean definitions in XML
- Creating an application context
- Controlling Bean Behavior
- Namespaces
- Lab
- **Advanced Topics**
  - Creating ApplicationContext
  - Constructor Arguments

# Creating the ApplicationContext

- So far, you have seen the ApplicationContext created like this:

```
ApplicationContext context = SpringApplication.run(MainConfig.class);
```

- Older “classic” techniques available as well:

```
new AnnotationConfigApplicationContext(MainConfig.class);
```

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

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

```
// Load from absolute path: C:/Users/alex/application-config.xml
```

```
new FileSystemXmlApplicationContext("C:/Users/alex/application-config.xml");
```

```
// path relative to the JVM working directory
```

```
new FileSystemXmlApplicationContext("./application-config.xml");
```

- Older techniques not intended to work with Spring Boot

- (covered later)

# 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:/Users/alex/application-config.xml");  
// absolute path: C:/Users/alex/application-config.xml
```

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



XmlWebApplicationContext is also available

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

# More on Constructor Args

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

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

Both look like Strings to XML

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

Index from zero

# Using Constructor Names

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

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

*Specify arg names in order*

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

# XML Dependency Injection

## Advanced Features & Best Practices

Techniques for Creating Reusable and Concise  
Bean Definitions

# Topics in this session

- **Factory Beans / Factory Method**
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- Advanced Features
  - Inner beans, collections, SpEL

# Advanced XML Bean Instantiation

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

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

# Advanced XML Bean Instantiation

- 3 techniques:
  - `@Bean` method in `@Configuration` class
    - 100% Java code available
  - XML *factory-method* attribute
    - No need to update Java class
  - Implement *FactoryBean* interface
    - Instantiation logic coded within *FactoryBean*
    - Spring auto-detects *FactoryBean* implementations

# The factory-method Attribute

- Non-intrusive
  - Useful for existing Singletons or Factories

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

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

*Spring configuration*

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

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

*Test class*

# The FactoryBean interface

- Used when factory-method unavailable
  - Or other complex cases inexpressible in XML
  - Used before @Bean methods introduced

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

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

# The FactoryBean interface

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

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

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

getObject()  
called by Spring  
automatically

# EmbeddedDatabaseFactoryBean

- Common example of a *FactoryBean*
  - Spring framework class for creating in-memory databases

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.  
                           embedded.EmbeddedDatabaseFactoryBean">
```

```
    <property name="databasePopulator" ref="populator"/>  
</bean>
```

*FactoryBean*

```
<bean id="populator"  
      class="org.springframework.jdbc.datasource.init.ResourceDatabasePopulator">
```

```
    <property name="scripts">  
        <list>
```

```
            <value>classpath:testdb/setup.sql</value>
```

```
        </list>
```

```
    </property>
```

```
</bean>
```

*Populate with test-data*

# FactoryBeans in Spring

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

# Topics in this session

- Factory Beans / Factory Method
- **'p' and 'c' namespaces**
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- Advanced Features
  - Inner beans, collections, SpEL

# The *c* and *p* namespaces

- Before

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

- After

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

Use camel case or hyphens



c namespace was introduced in Spring 3.1

# The c and p namespaces

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

Namespace declaration

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

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

Inject value for property 'url'

Inject reference for bean 'service'

# No schemaLocation needed

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

    <!-- ... -->

</beans>
```

*p and c*  
namespace  
definitions

no extra *schemaLocation*  
entry required (no xsd)

# 'c' and 'p' Pros and Cons

- Pros
  - More concise
  - Well supported in STS
    - CTRL+space works well
- Cons
  - Less widely known than the usual XML configuration syntax



# Topics in this session

- Factory Beans / Factory Method
- 'p' and 'c' namespaces
- **Profiles**
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- Advanced Features
  - Inner beans, collections, SpEL

# Profile Configuration XML

- All bean definitions

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

Profile applies to all beans

- Subset of bean definitions

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

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

# Sample XML Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <beans profile="dev">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource" />
    </beans>
</beans>
```

# Topics in this session

- Factory Beans / Factory Method
- 'p' and 'c' namespaces
- Profiles
- **Externalizing values into properties files**
- Using Bean definition inheritance
- Lab
- Advanced Features
  - Inner beans, collections, SpEL

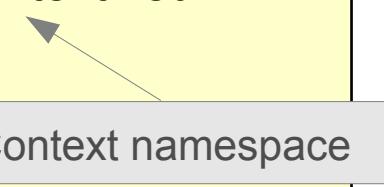
# Externalizing values to a properties file (1)

- Namespace declaration

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

    <context:property-placeholder ... />

</beans>
```



Context namespace

# Externalizing values to a properties file (2)

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

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="URL" value="${db.url}" />
    <property name="user" value="${db.user.name}" />
  </bean>
</beans>
```



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

*db-config.properties*



```
<bean id="dataSource"
  class="org.apache.commons.dbcp.BasicDataSource">
  <property name="URL" value="jdbc:postgresql:..." />
  <property name="user" value="moneytransfer-app" />
</bean>
```

Resolved at  
startup time

# Topics in this session

- Factory Beans / Factory Method
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- **Using Bean definition inheritance**
- Lab
- Advanced Features
  - Inner beans, collections, SpEL

# Bean Definition Inheritance (1)

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

# Without Bean Definition Inheritance

```
<beans>
  <bean id="pool-A" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="URL" value="jdbc:postgresql://server-a/transfer" />
    <property name="user" value="moneytransfer-app" />
  </bean>

  <bean id="pool-B" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="URL" value="jdbc:postgresql://server-b/transfer" />
    <property name="user" value="moneytransfer-app" />
  </bean>

  <bean id="pool-C" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="URL" value="jdbc:postgresql://server-c/transfer" />
    <property name="user" value="moneytransfer-app" />
  </bean>
</beans>
```

Can you find the duplication?

# Abstract Parent bean

```
<beans>
  <bean id="abstractPool"
    class="org.apache.commons.dbcp.BasicDataSource" abstract="true">
    <property name="user" value="moneytransfer-app" />
  </bean>
<bean id="pool-A" parent="abstractPool">
  <property name="URL" value="jdbc:postgresql://server-a/transfer" />
</bean>
<bean id="pool-B" parent="abstractPool">
  <property name="URL" value="jdbc:postgresql://server-b/transfer" />
</bean>
<bean id="pool-C" parent="abstractPool">
  <property name="URL" value="jdbc:postgresql://server-c/transfer" />
  <property name="user" value="bank-app" />
</bean>
</beans>
```

Will not be instantiated

Can override

Each pool inherits its *parent* configuration

# Default Parent Bean

```
<beans>

    <bean id="defaultPool" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="URL" value="jdbc:postgresql://server-a/transfer" />
        <property name="user" value="moneytransfer-app" />
    </bean>

    <bean id="pool-B" parent="defaultPool">
        <property name="URL" value="jdbc:postgresql://server-b/transfer" />
    </bean>

    <bean id="pool-C" parent="defaultPool" class="example.SomeOtherPool">
        <property name="URL"
            value="jdbc:postgresql://server-c/transfer" />
    </bean>

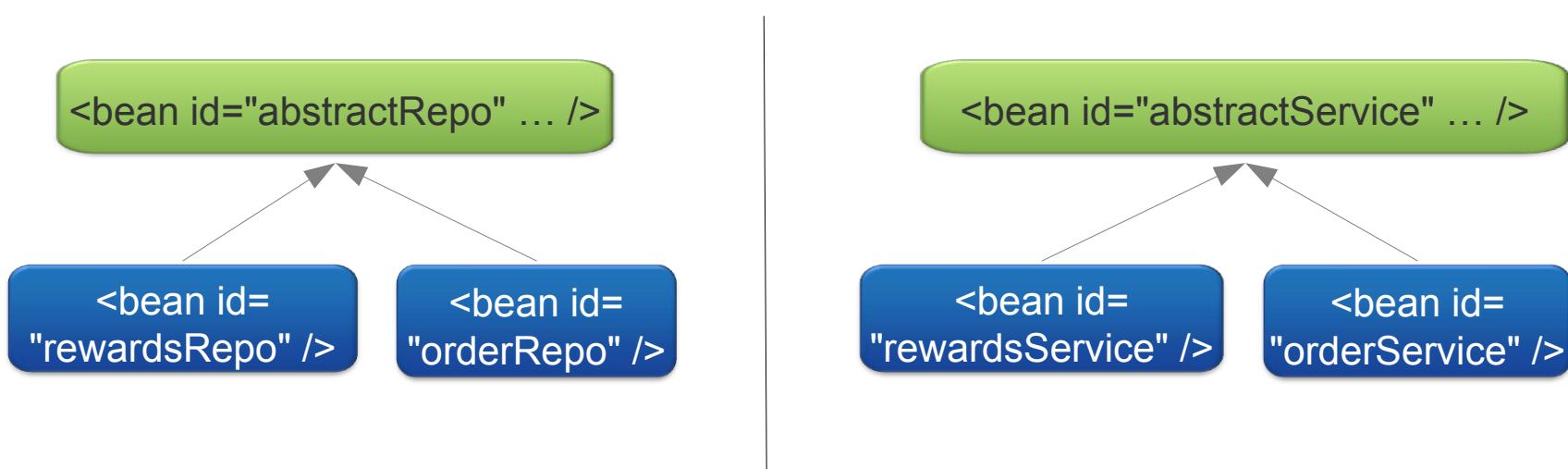
</beans>
```

Overrides URL property

Overrides class as well

# Inheritance for service and repository beans

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



# Lab (Optional)

Using Bean Definition Inheritance and  
Property Placeholders

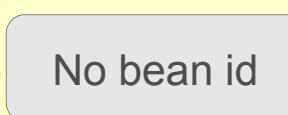
# Topics in this session

- Factory Beans / Factory Method
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- Advanced Features
  - Inner beans
  - Collections
  - SpEL

# Inner beans

- Inner bean only visible from surrounding bean

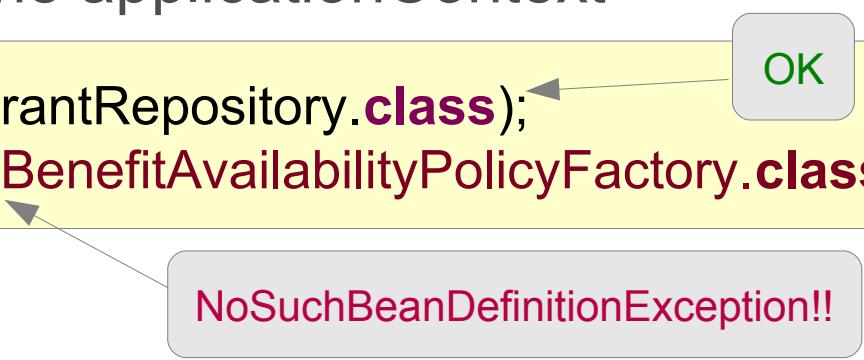
```
<bean id="restaurantRepository"
      class="rewards.internal.restaurant.JdbcRestaurantRepository">
    <property name="benefitAvailabilityPolicy">
      <bean class="rewards...DefaultBenefitAvailabilityPolicyFactory" />
    </property>
</bean>
```



No bean id

- Cannot be accessed from the applicationContext

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



OK

NoSuchBeanDefinitionException!!

# Without an Inner Bean

```
<beans>

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

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

    ...
</beans>
```

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

# With an Inner Bean

```
<beans>

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

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

# Multiple Levels of Nesting

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

# Inner Beans: pros and cons

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



# Topics in this session

- Factory Beans / Factory Method
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  - Inner beans
  - **Collections**
  - SpEL

# *beans* and *util* collections

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



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

# Using the *beans* namespace

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

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

Equivalent to:

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

ApplicationContext

service -> instance of TransferServiceImpl

# beans collections limitation

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

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

OK

NoSuchBeanDefinitionException!!

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

# Injecting a Set or Map

- Similar support available for Set

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

- Map (through map / entry / key elements)

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

Key can use primitive type or ref to bean

# Injecting a collection of type *Properties*

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

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

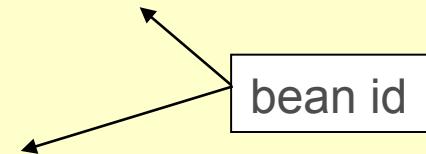
```
<property name="config">  
  <props>  
    <prop key="server.host">mailer</prop>  
    <prop key="server.port">1010</prop>  
  </props>  
</property>
```

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

# util collections

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

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



Implementation class

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

# *beans* or *util* collections?

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

# Topics in this session

- Factory Beans / Factory Method
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  - Inner beans
  - Collections
  - **SpEL**

# Spring Expression Language

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



SpEL was introduced in Spring 3.0

# SpEL examples – XML

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

Can refer a nested property

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

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

Equivalent to System.getProperty(...)

# SpEL examples – Using @Value

```
<bean id="rewardsDb" class="com.acme.RewardsTestDatabase">
    <property name="databaseName" value="#{systemProperties['database.name']}"/>
    <property name="keyGenerator" value="#{strategyBean.databaseKeyGenerator}"/>
</bean>
```

```
@Repository
public class RewardsTestDatabase {

    @Value("#{systemProperties.databaseName}")
    public void setDatabaseName(String dbName) { ... }

    @Value("#{strategyBean.databaseKeyGenerator}")
    public void setKeyGenerator(KeyGenerator kg) { ... }
}
```

```
@Configuration
class RewardConfig {
    @Bean public RewardsDatabase rewardsDatabase
        (@Value("#{systemProperties.databaseName}") String databaseName, ...) {
            ...
        }
}
```

# SpEL

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

# Using SpEL functions

- In Spring Security

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

- In Spring Batch

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



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

# Summary

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

# Understanding the Bean Lifecycle

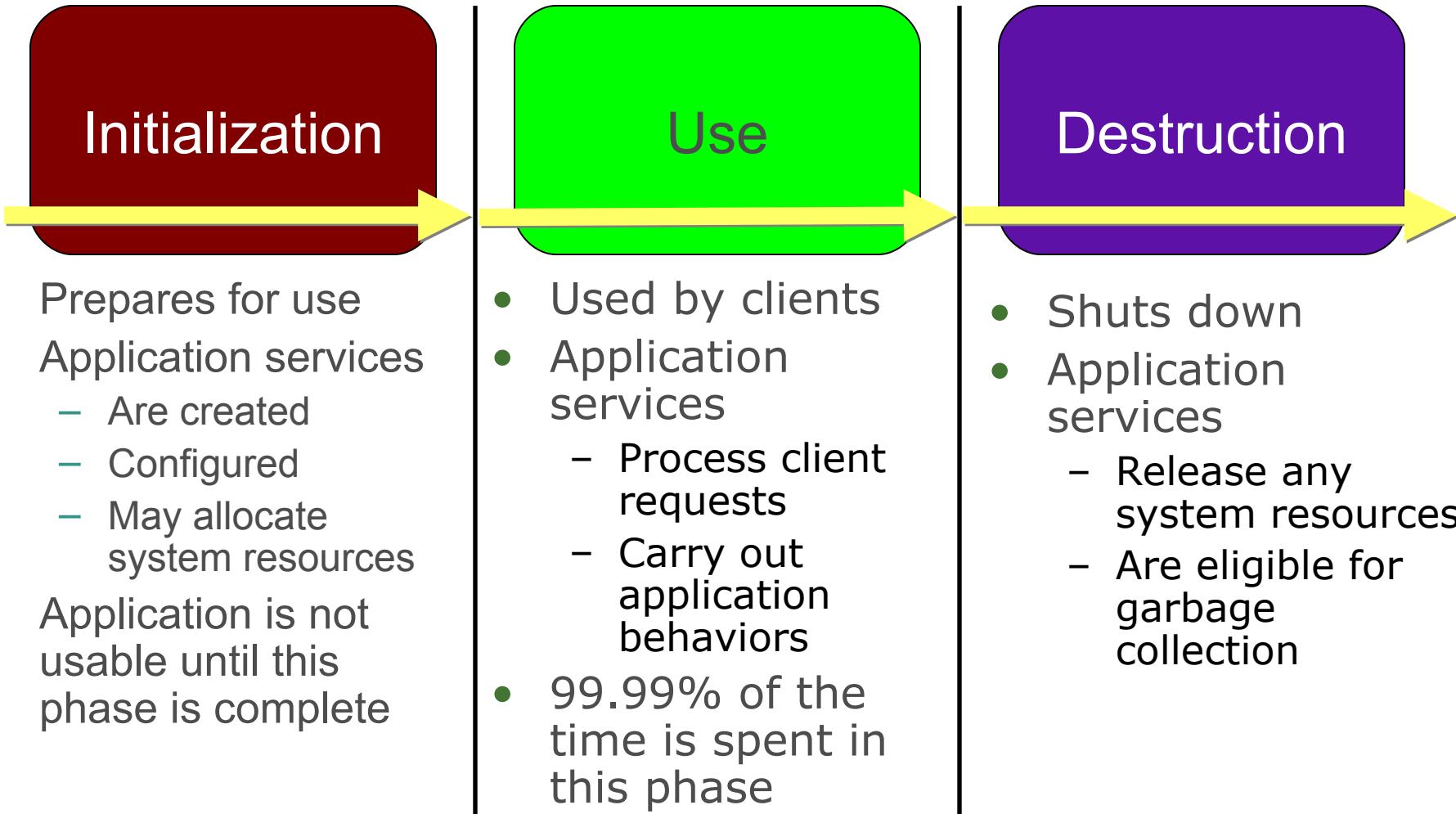
An In-Depth Look “Under the Hood”

Using Bean Pre- and Post-Processors

# Topics in this session

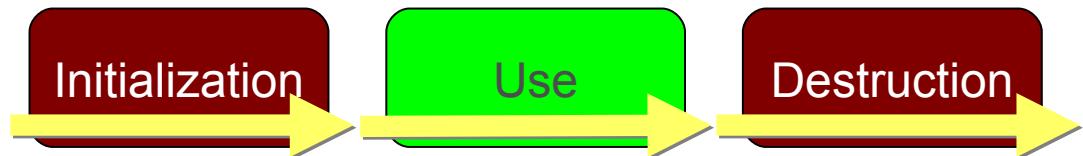
- **Introduction**
- The initialization phase
- The use phase
- The destruction phase

# Phases of the Application Lifecycle



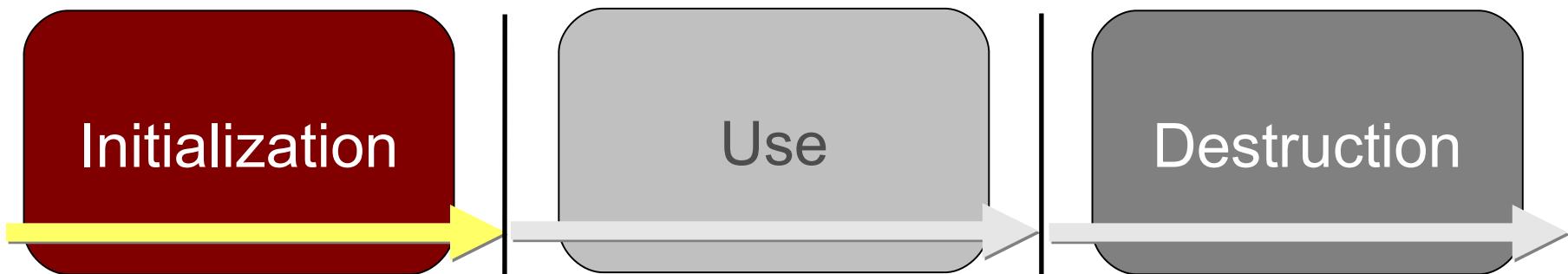
# Spring's Role as a Lifecycle Manager

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



# Topics in this session

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



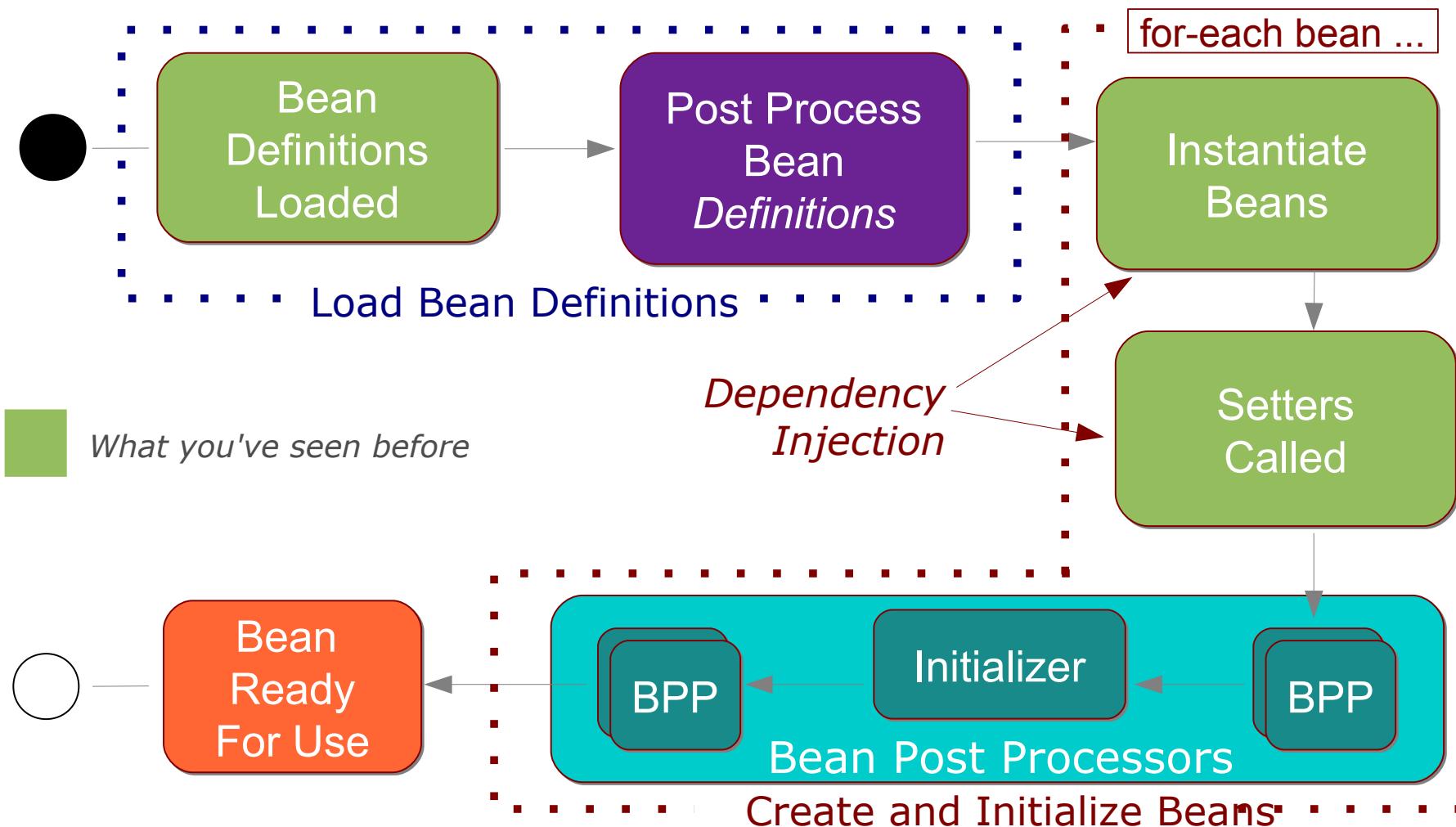
# Lifecycle of a Spring Application Context (1) - The Initialization Phase

- When a context is created the initialization phase completes

```
// Create the application from the configuration
ApplicationContext context =
    new SpringApplication.run(AppConfig.class);
```

- But what exactly happens in this phase?

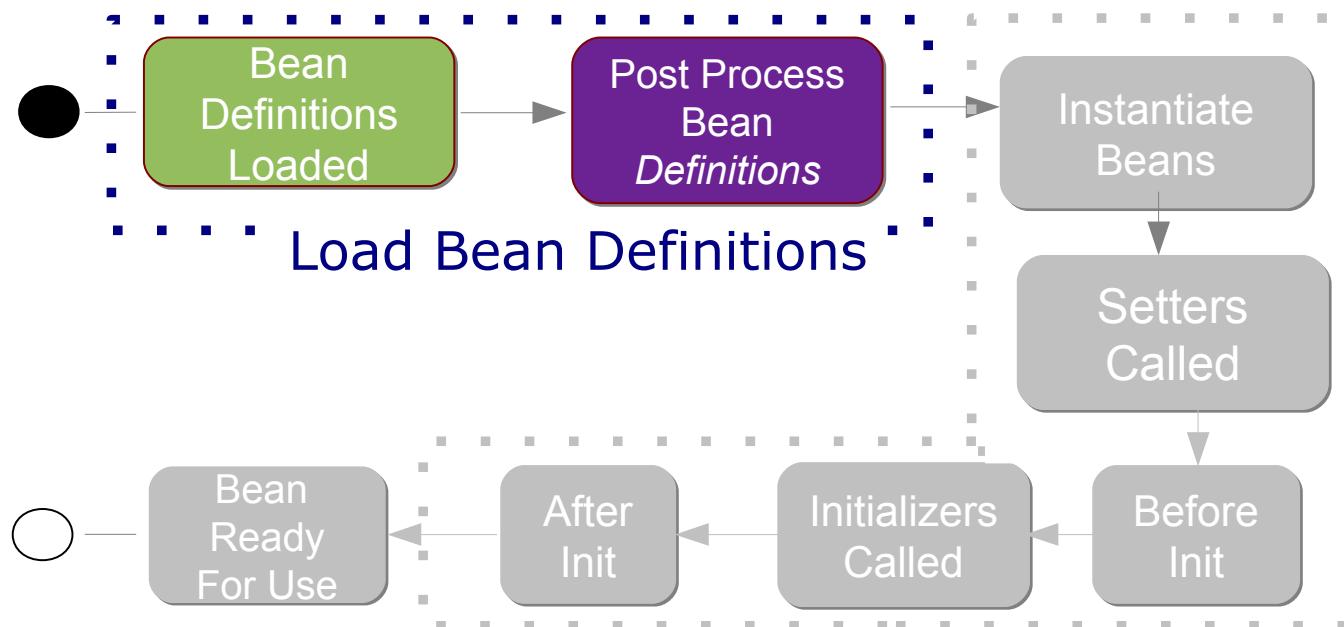
# Bean Initialization Steps



# Inside The Application Context

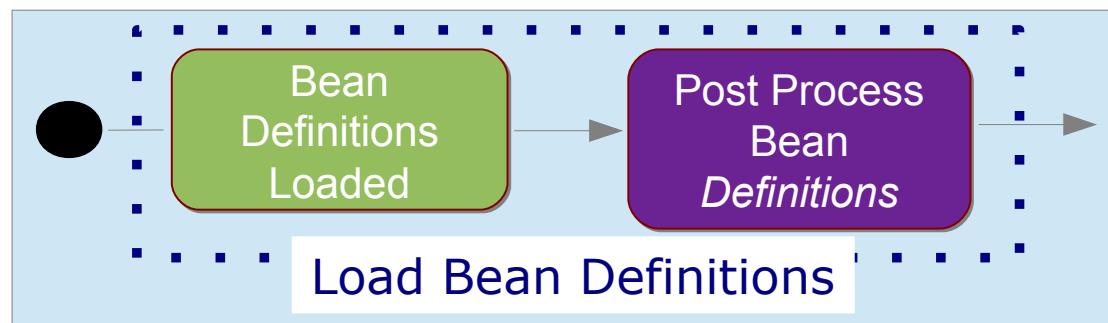
## - Initialization Lifecycle

- Load bean definitions
- Initialize bean instances



# Load Bean Definitions

- The @Configuration classes are processed
  - And/or @Components are scanned for
  - And/or XML files are parsed
- Bean definitions added to BeanFactory
  - Each indexed under its id
- Special BeanFactoryPostProcessor beans invoked
  - Can modify the definition of any bean



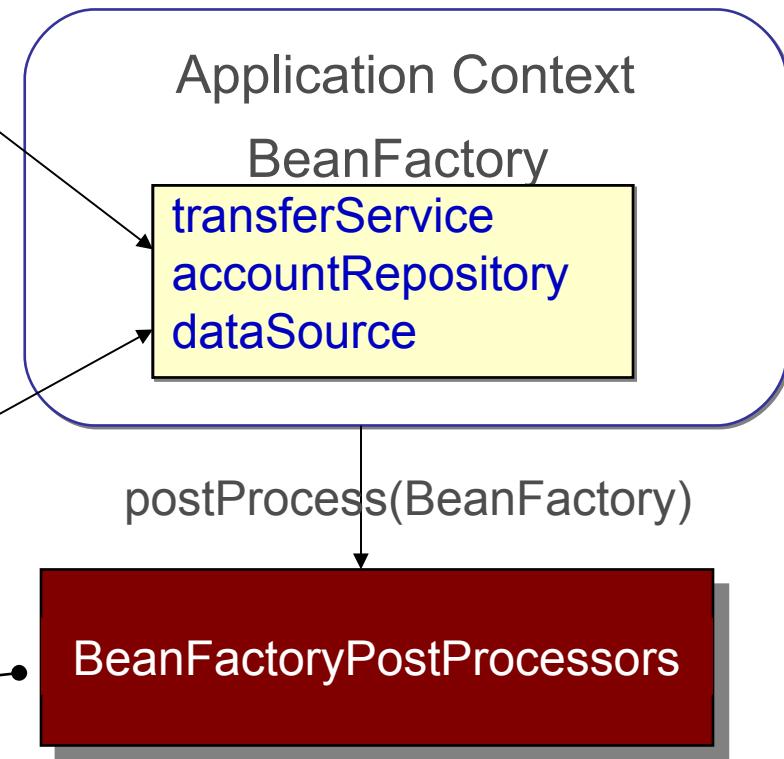
# Load Bean Definitions

## AppConfig.java

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

## TestInfrastructureConfig.java

```
@Bean  
public DataSource dataSource () { ... }
```



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

# BeanFactoryPostProcessor Extension Point

- Applies transformations to bean *definitions*
  - Before objects are actually created
- Several useful implementations provided in Spring
  - You can write your own (not common)
  - Implement **BeanFactoryPostProcessor** interface

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

# Most Common Example of BeanFactoryPostProcessor

Remember the  
*property-placeholder*?

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

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



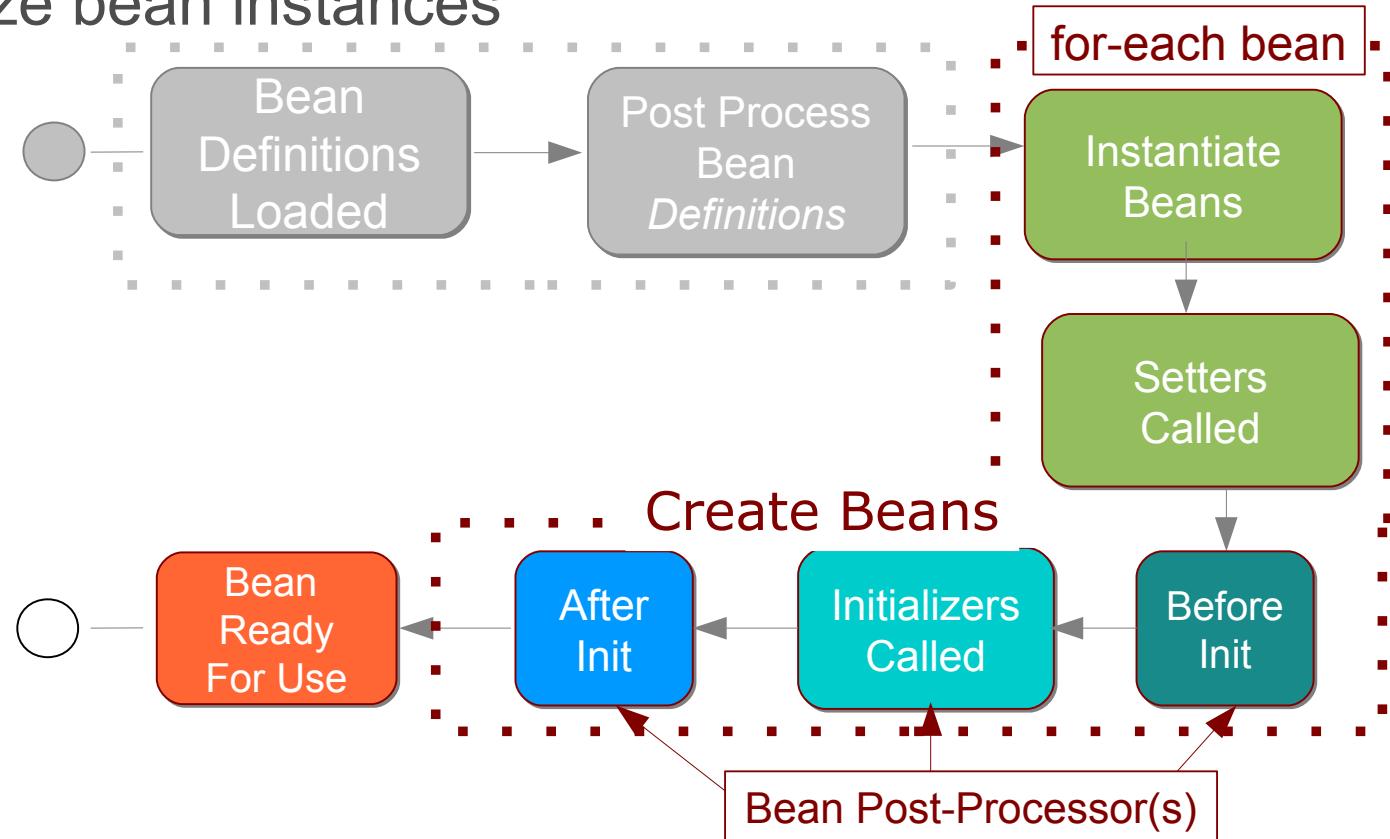
dbUrl=jdbc:oracle:...  
dbUserName=moneytransfer-app



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

# Inside the Application Context Initialization Lifecycle

- Load bean definitions
- Initialize bean instances



# Initializing Bean Instances

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



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

# Bean Post Processing

- There are two types of bean post processors
  - Initializers
    - Initialize the bean if instructed (i.e. `@PostConstruct`)
  - All the rest!
    - Allow for additional configuration
    - May run before or after the initialize step



# The Initializer Extension Point

- All *init* methods are called



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

*By Annotation*

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

*Using XML only*

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

Declares several BPPs *including*  
CommonAnnotationBeanPostProcessor

# The BeanPostProcessor Extension Point



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

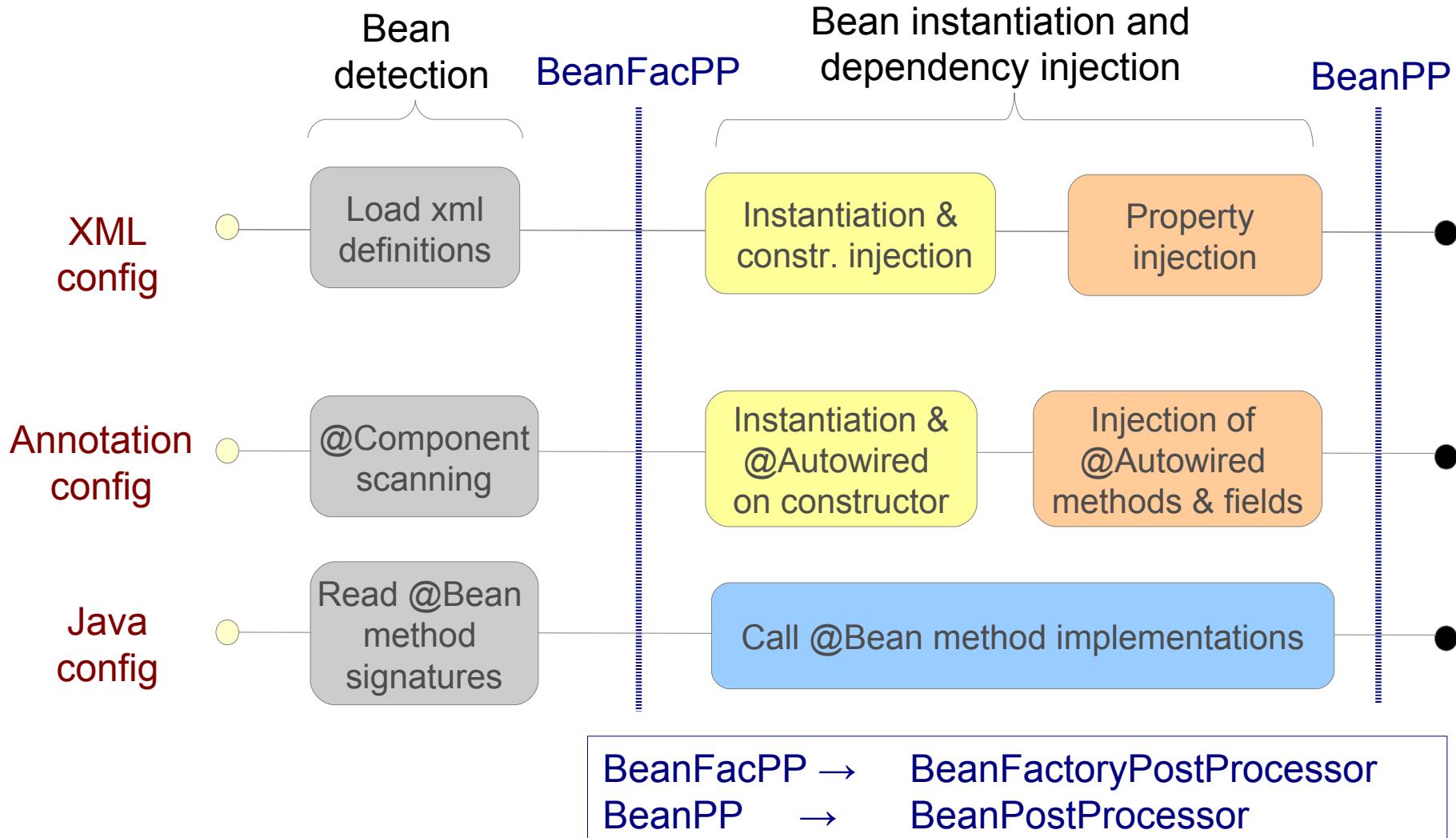
Course  
will show  
several  
BPPs

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

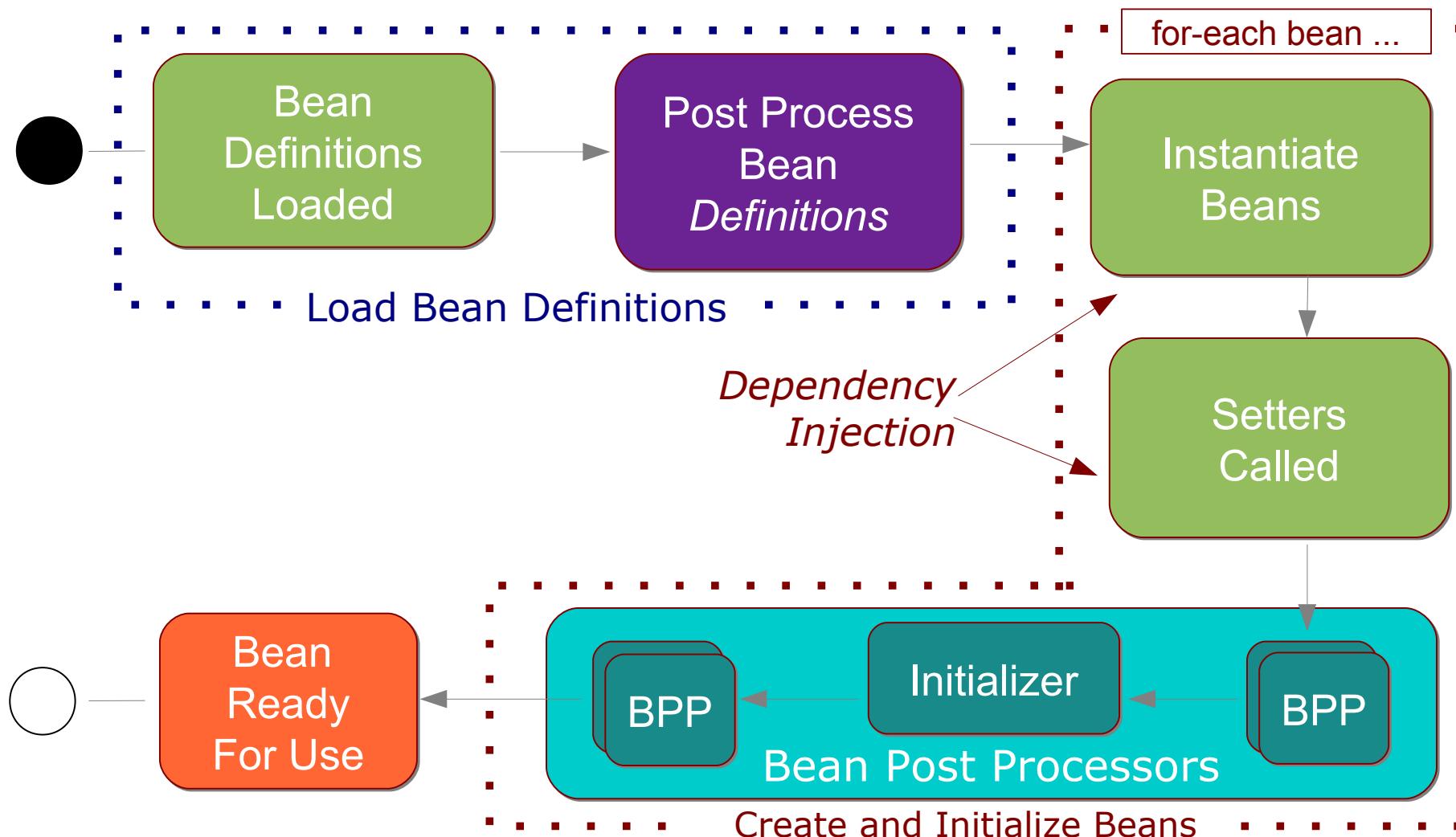
Post-processed bean

Original bean

# Configuration Lifecycle

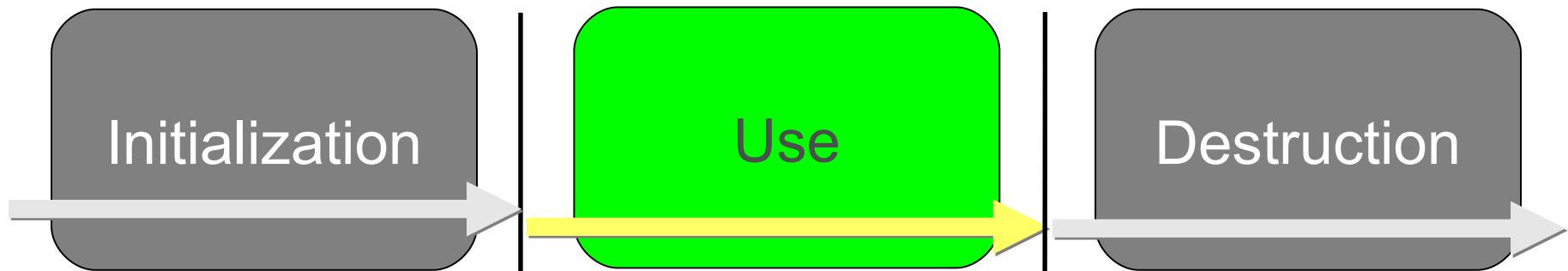


# The Full Initialization Lifecycle



# Topics in this session

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



# Lifecycle of a Spring Application Context (2)

## - The Use Phase

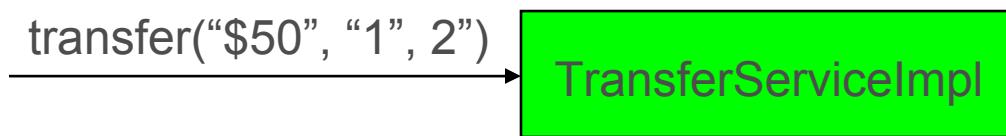
- When you invoke a bean obtained from the context the application is used

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

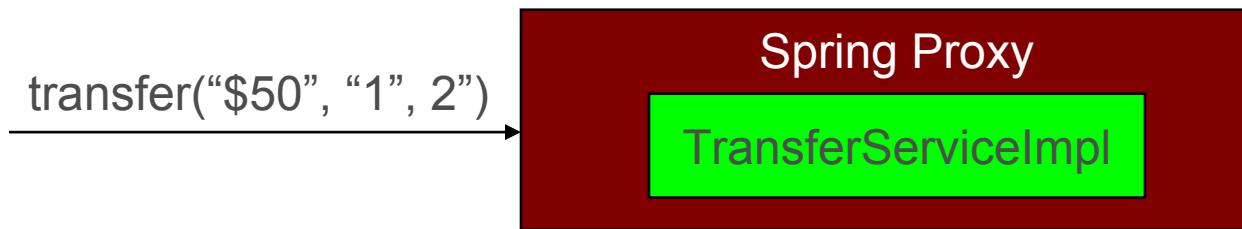
- But exactly what happens in this phase?

# Inside The Bean Request (Use) Lifecycle

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



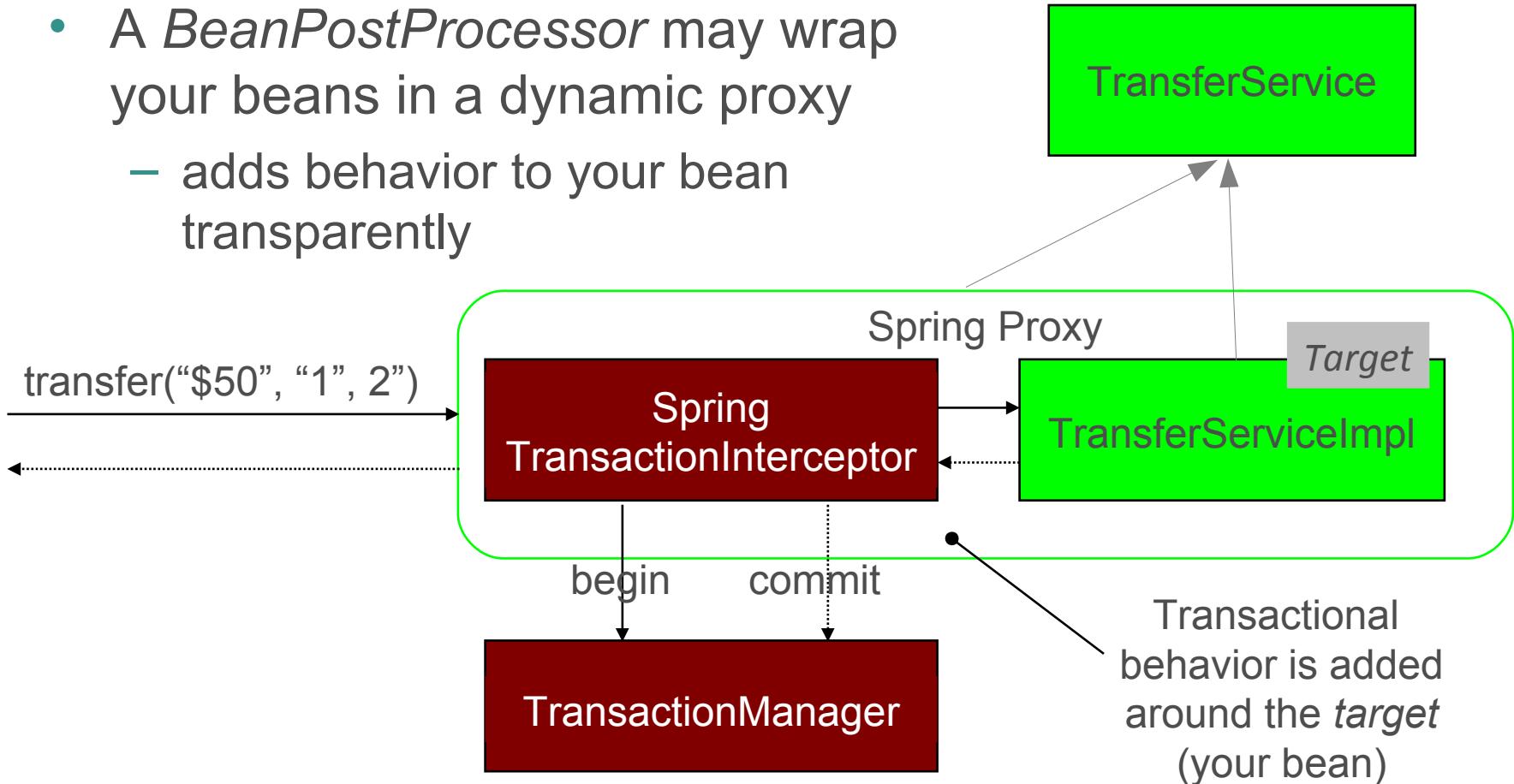
- Your bean has been wrapped in a *proxy*
  - things become more interesting



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

# Proxy Power

- A *BeanPostProcessor* may wrap your beans in a dynamic proxy
  - adds behavior to your bean transparently



# Kinds of Proxies

- Spring will create either JDK or CGLib proxies

## JDK Proxy

- Also called dynamic proxies
- API is built into the JDK
- Requirements: Java interface(s)

## CGLib Proxy

- NOT built into JDK
- Included in Spring jars
- Used when interface not available
- Cannot be applied to final classes or methods

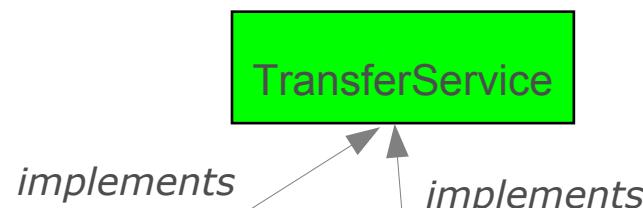


Recommendation: Code to interfaces / Use JDK proxies (default)

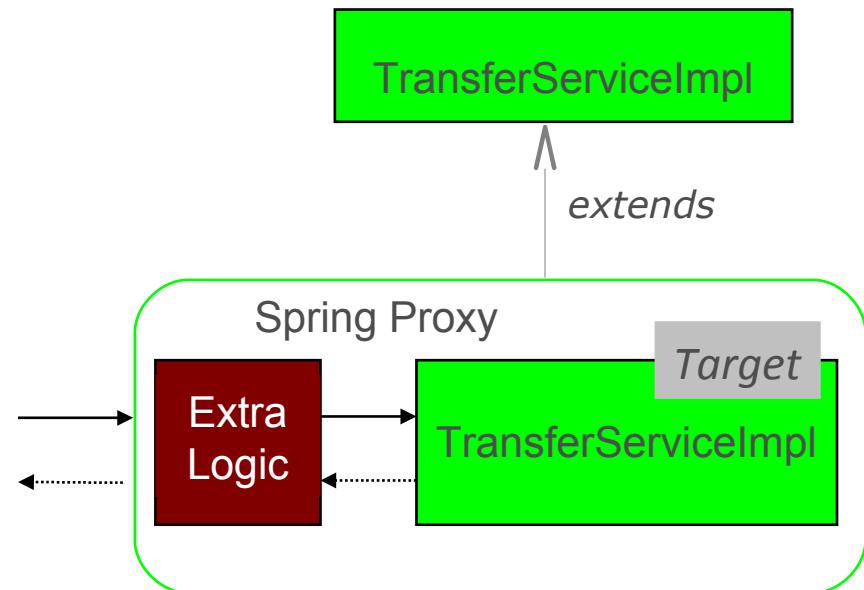
**See Spring Reference - 10.5.3 JDK- and CGLIB-based proxies**

# JDK vs CGLib Proxies

- JDK Proxy
  - Interface based

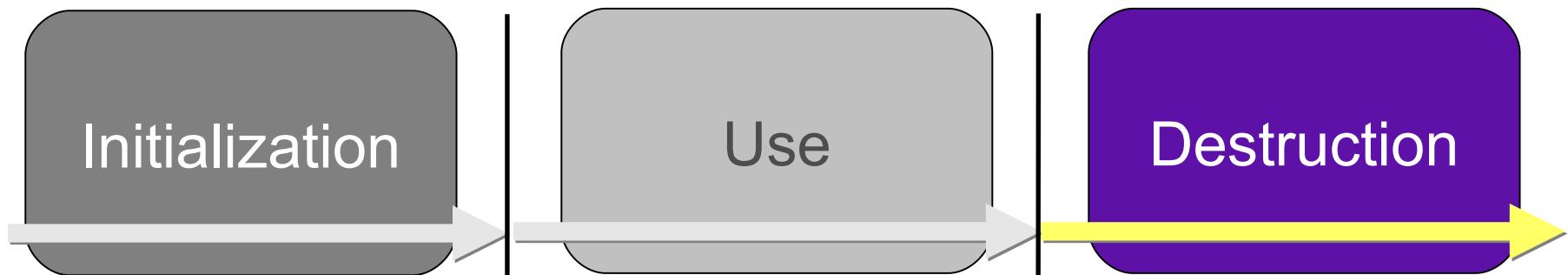


- CGLib Proxy
  - subclass based



# Topics in this session

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



# The Lifecycle of a Spring Application Context (3)

## - The Destruction Phase

- When you close a context the destruction phase completes

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

- But exactly what happens in this phase?

# ApplicationContext Destruction Lifecycle (1)

- Destroy bean instances if instructed
  - Call their destroy (clean-up) methods
  - Beans must have a *destroy method* defined
    - A no-arg method returning void
- Context then destroys (cleans-up) itself
  - The context is not usable again

**Remember:**  
only GC actually  
destroys objects

```
@Bean (destroyMethod="clearCache")
public AccountRepository accountRepository() {
    // ...
}
```

A method on the  
AccountRepository

JavaConfig



- Called only when ApplicationContext / JVM exit *normally*
- Not called for prototype beans

# ApplicationContext Destruction Lifecycle (2)

- Can do the same using XML or annotations
  - Annotations require *annotation-driven* or the component scanner to be activated

## Using XML

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

## By Annotation

```
public class AccountRepository {

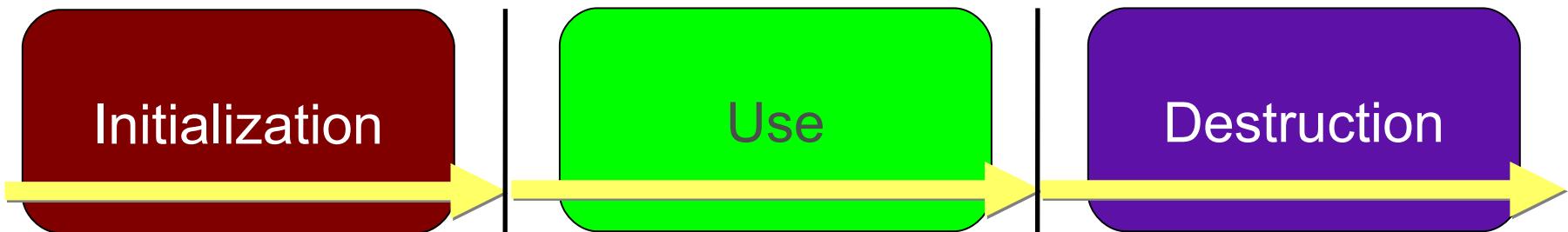
    @PreDestroy
    void clearCache() {
        // close files, connections ...
        // remove external resources ...
    }
}
```

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

```
<context:component-scan ... />
```

# Topics Covered

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



# Testing Spring Applications

Unit Testing without Spring  
Integration Testing with Spring

Testing in General, Spring and JUnit, Profiles

# Topics in this Session

- **Test Driven Development**
- Unit Testing vs. Integration Testing
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases

# What is TDD

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

# “But I Don’t Have Time to Write Tests!”

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

# TDD and Agility

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

# TDD and Design

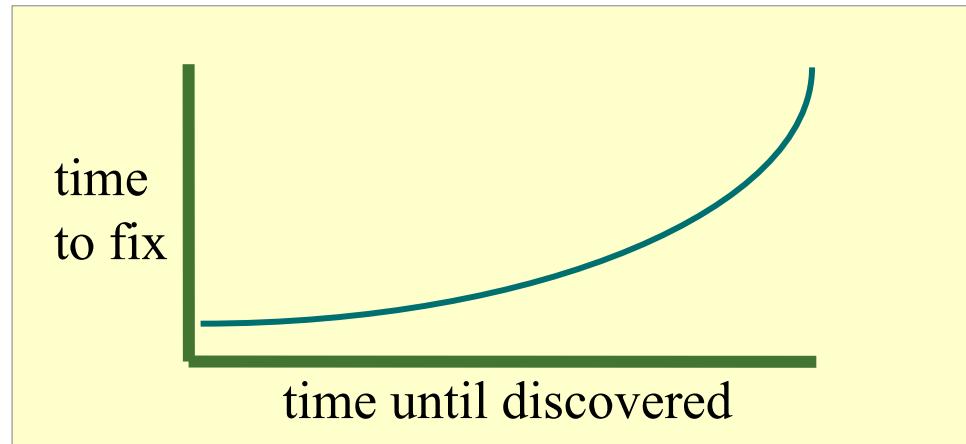
- Testing makes you think about your design
- If your code is hard to test then the design should be reconsidered

# TDD and Focus

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

# Benefits of Continuous Integration

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



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

# Topics in this Session

- Test Driven Development
- **Unit Testing vs. Integration Testing**
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases

# Unit Testing vs. Integration Testing

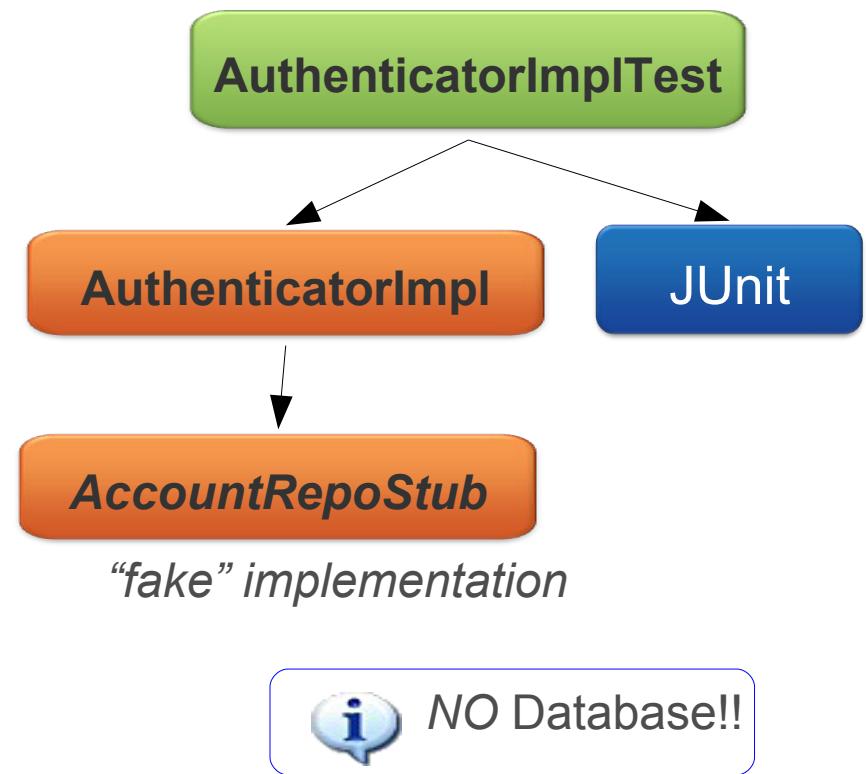
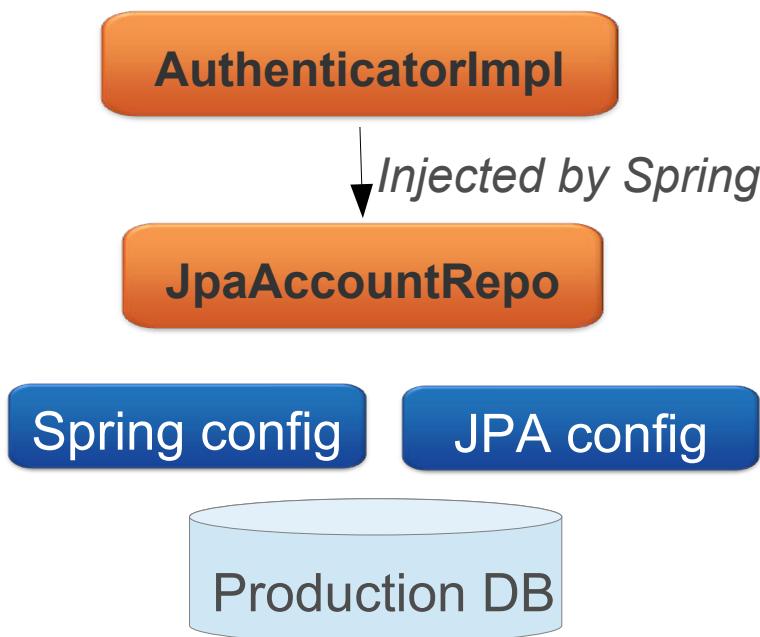
- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
- Integration Testing
  - Tests the interaction of multiple units working together
  - Integrates infrastructure

# Unit Testing

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

# Unit Testing example

- Production mode
- Unit test with Stubs



# Example Unit to be Tested

```
public class AuthenticatorImpl implements Authenticator {  
    private AccountRepository accountRepository;  
  
    public AuthenticatorImpl(AccountRepository accountRepository) {  
        this.accountRepository = accountRepository; ← External dependency  
    }  
  
    public boolean authenticate(String username, String password) {  
        Account account = accountRepository.getAccount(username);  
  
        return account.getPassword().equals(password); ← Unit business logic  
– 2 paths: success or fail  
    }  
}
```

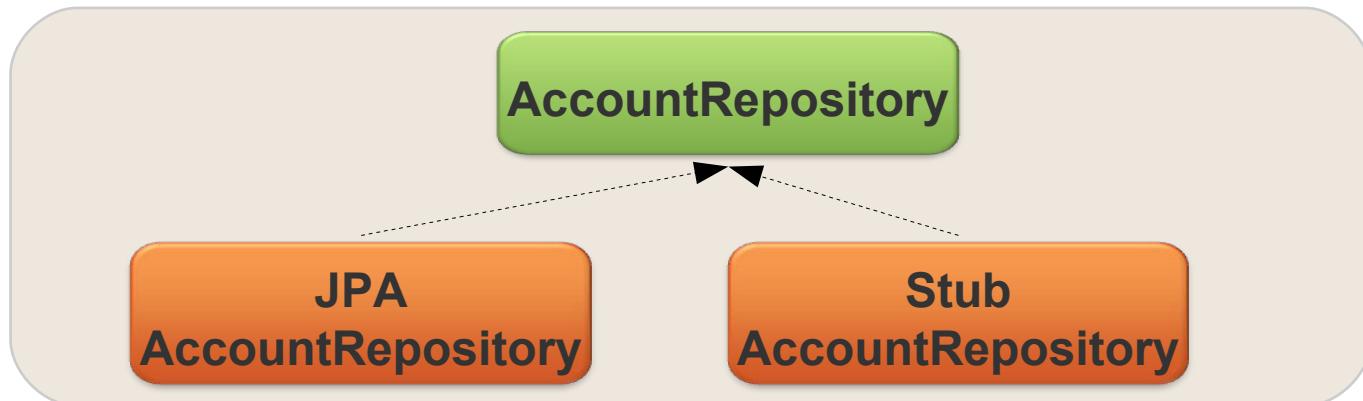
Note: Validation failure paths ignored for simplicity

# Implementing a Stub

- Class created manually
  - Implements Business interface

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

Simple state



# Unit Test using a Stub

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

```
public class AuthenticatorImplTests {
```

```
    private AuthenticatorImpl authenticator;
```

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

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

```
}
```

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

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

```
}
```

Spring **not** in charge of  
injecting dependencies

OK scenario

KO scenario

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

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

```
}
```

# Unit Testing with Stubs

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

# Steps to Testing with a Mock

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

# Example: Using a Mock - I

- Setup
  - A Mock class is created at startup time

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

static import

Implementation of interface  
AccountRepository is created

# Example: Using a Mock - II

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

```
@Test public void validUserWithCorrectPassword() {  
    expect(accountRepository.getAccount("lisa")).  
        andReturn(new Account("lisa", "secret"));
```

```
replay(accountRepository);
```

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

```
    verify(accountRepository);  
}  
}
```

Recording

What behavior to  
expect?

Recording      Playback

“playback”  
mode

Mock now fully available

Verification

No planned method call  
has been omitted

# Mock Considerations

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

# Mocks or Stubs?

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

# Topics in this Session

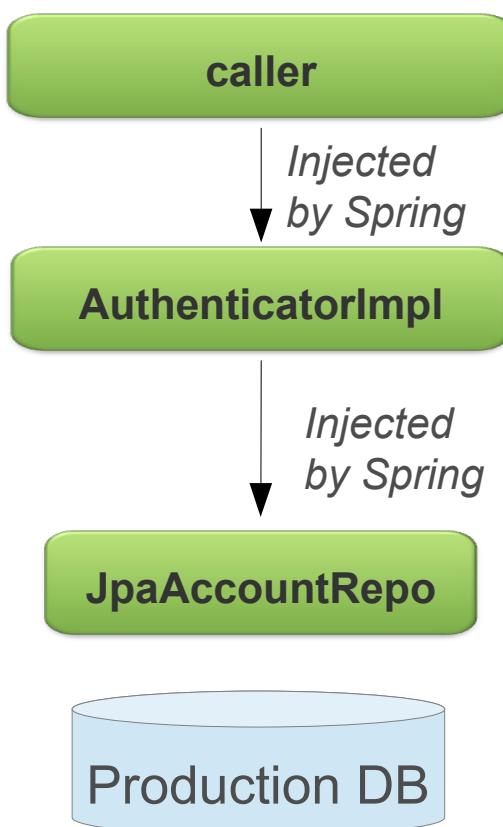
- Test Driven Development
- Unit Testing vs. Integration Testing
- **Integration Testing with Spring**
- Testing with Profiles
- Testing with Databases

# Integration Testing

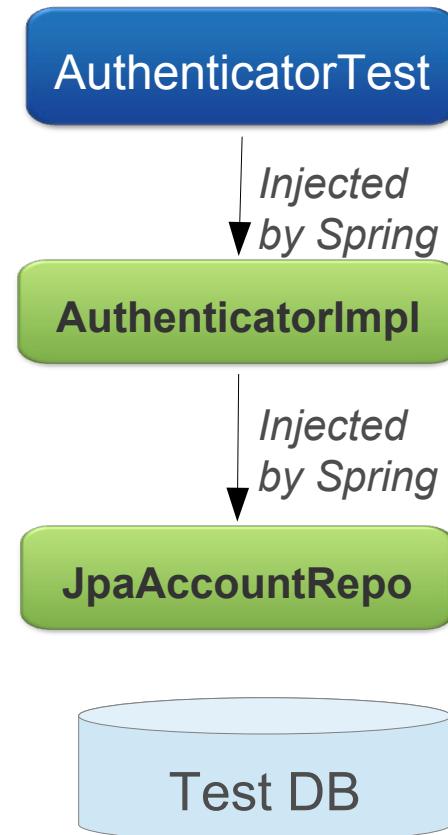
- Tests the interaction of multiple units
- Tests application classes in the context of their surrounding infrastructure
  - Infrastructure may be “scaled down”
  - e.g. use Apache DBCP connection pool instead of container-provider pool obtained through JNDI

# Integration test example

- Production mode



- Integration test



# Spring's Integration Test Support

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



See: Spring Framework Reference – Integration Testing

<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#integration-testing>

# Using Spring's Test Support

Run with Spring support

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@ContextConfiguration(classes=SystemTestConfig.class)
```

```
public final class TransferServiceTests {
```

```
    @Autowired
```

```
    private TransferService transferService;
```

```
    @Test
```

```
    public void successfulTransfer() {
```

```
        TransferConfirmation conf = transferService.transfer(...);
```

```
    }
```

Test the system as normal

```
}
```

Point to system test configuration file

Inject bean to test

No need for @Before method

# @ContextConfiguration and JavaConfig

- Testing Java Configuration defined beans

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

For @Configuration classes

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class MyTest {
    @Configuration
    public static class TestConfiguration {
        @Bean public DataSource dataSource() { ... }
    }
}
```

Inner class *must* be **static**

Inner @Configuration  
automatically detected

# @ContextConfiguration XML Examples

- Tests when using XML based configuration

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

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

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

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

Loads TransferServiceTests-context.xml

# Multiple test methods

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)

public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

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

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

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



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

# Topics in this Session

- Test Driven Development
- Unit Testing vs. Integration Testing
- Integration Testing with Spring
- **Testing with Profiles**
- Testing with Databases

# Activating Profiles For a Test

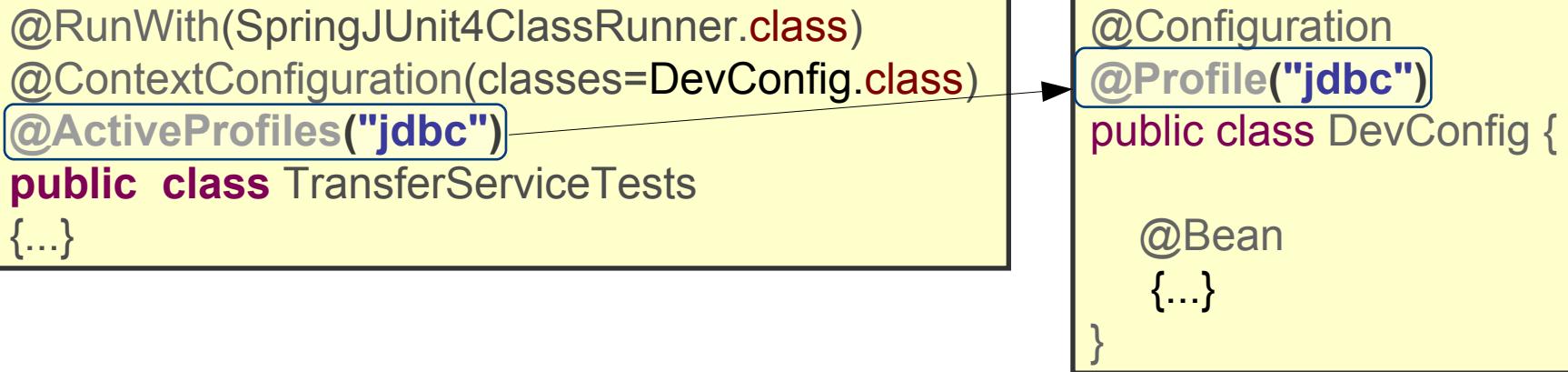
- **@ActiveProfiles** inside the test class
  - Define one or more profiles
  - Beans associated with that profile are instantiated
  - Also beans not associated with *any* profile
- Example: Two profiles activated – *jdbc* **and** *dev*

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=DevConfig.class)
@ActiveProfiles( { "jdbc", "dev" } )
```

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

# Profiles Activation with JavaConfig

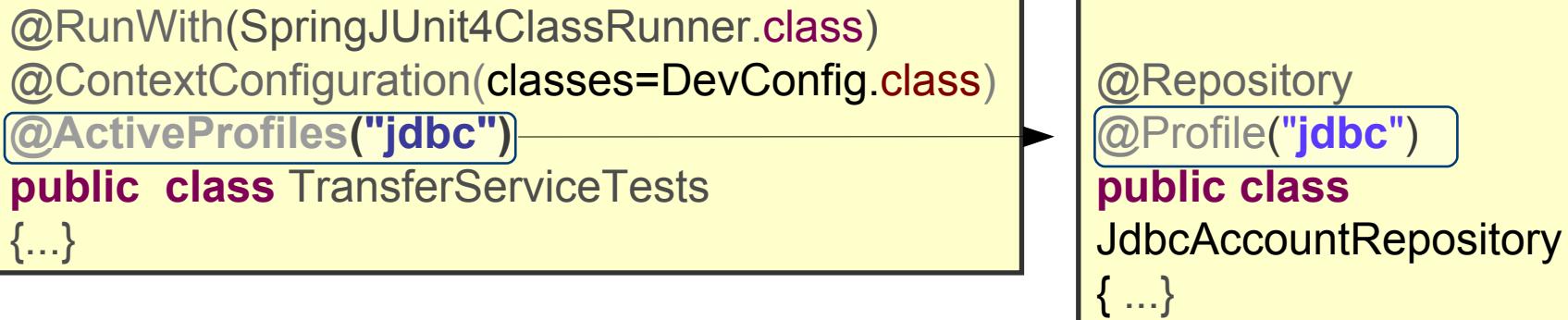
- **@ActiveProfiles** inside the test class
- **@Profile** inside the **@Configuration** class



**Remember:** only `@Configurations` matching an active profile or with *no profile* are loaded

# Profiles Activation with Annotations

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



Only beans with current profile / no profile are component-scanned

# Profiles Activation with XML

- **@ActiveProfiles** inside the test class
- **profile** attribute inside **<bean>** tag

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

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

```
<beans xmlns=...>
  <!-- Available to all profiles →
  <bean id="rewardNetwork" ... />
  ...
  <b><beans profile="jdbc"> ... </beans></b>
  <b><beans profile="jpa"> ... </beans></b>
</beans>
```



Only beans with current profile / no profile are loaded

# Topics in this Session

- Test Driven Development
- Unit Testing vs. Integration Testing
- Integration Testing with Spring
- Testing with Profiles
- **Testing with Databases**

# Testing with Databases

- Integration testing against SQL database is common.
- In-memory databases useful for this kind of testing
  - No prior install needed
- Common requirement: populate DB before test runs
  - Use the `@Sql` annotation:

```
@Test  
@Sql ( "/testfiles/test-data.sql" )  
public void successfulTransfer() {  
    ...  
}
```

Run this SQL command  
Before this test method executes.



See: Spring Framework Reference, Executing SQL Scripts

<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#testcontext-executing-sql>

# @Sql Examples

```
@RunWith(SpringJUnit4ClassRunner.class)
```

Run these scripts before  
each **@Test** method

```
@ContextConfiguration(...)
```

```
@Sql({ "/testfiles/schema.sql", "/testfiles/general-data.sql" } )
```

```
public final class MainTests {
```

```
    @Test
```

```
    @Sql
```

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

Run script named  
**MainTests.success.sql**  
in same package

```
    @Test
```

```
    @Sql ( "/testfiles/error.sql" )
```

```
    @Sql ( scripts="/testfiles/cleanup.sql",  
           executionPhase=Sql.ExecutionPhase.AFTER_TEST_METHOD )
```

Run before **@Test** method...

...run after **@Test** method

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

```
}
```

# Benefits of Testing with Spring

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

# Summary

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

# Lab

## Testing Spring Applications

# Developing Aspects with Spring AOP

Aspect Oriented Programming For  
Declarative Enterprise Services

Using and Implementing Spring Proxies

# Topics in this session

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

# What Problem Does AOP Solve?

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

# What are Cross-Cutting Concerns?

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

# An Example Requirement

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



A sign this requirement is a cross-cutting concern

# Implementing Cross Cutting Concerns Without Modularization

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

# Symptom #1: Tangling

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

 Mixing of concerns

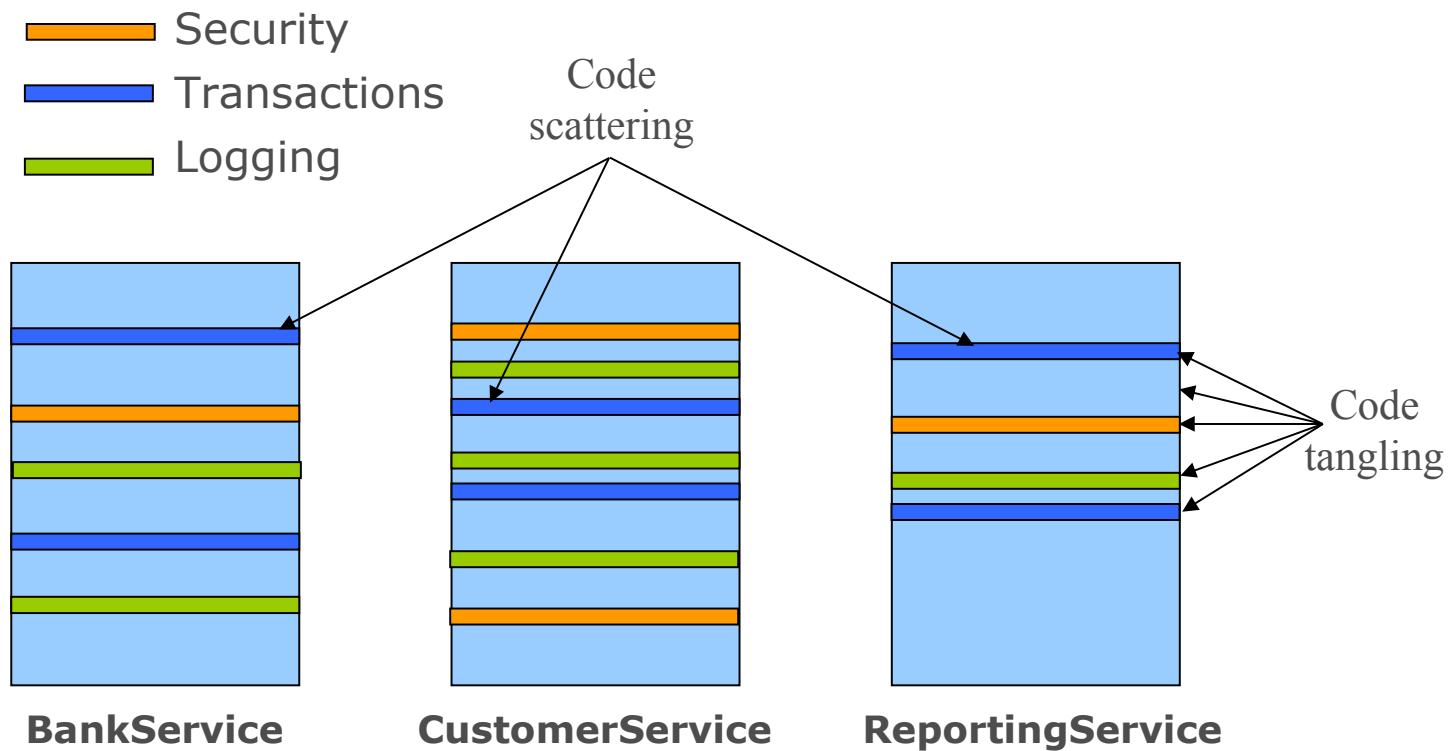
## Symptom #2: Scattering

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

Duplication

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

# System Evolution Without Modularization



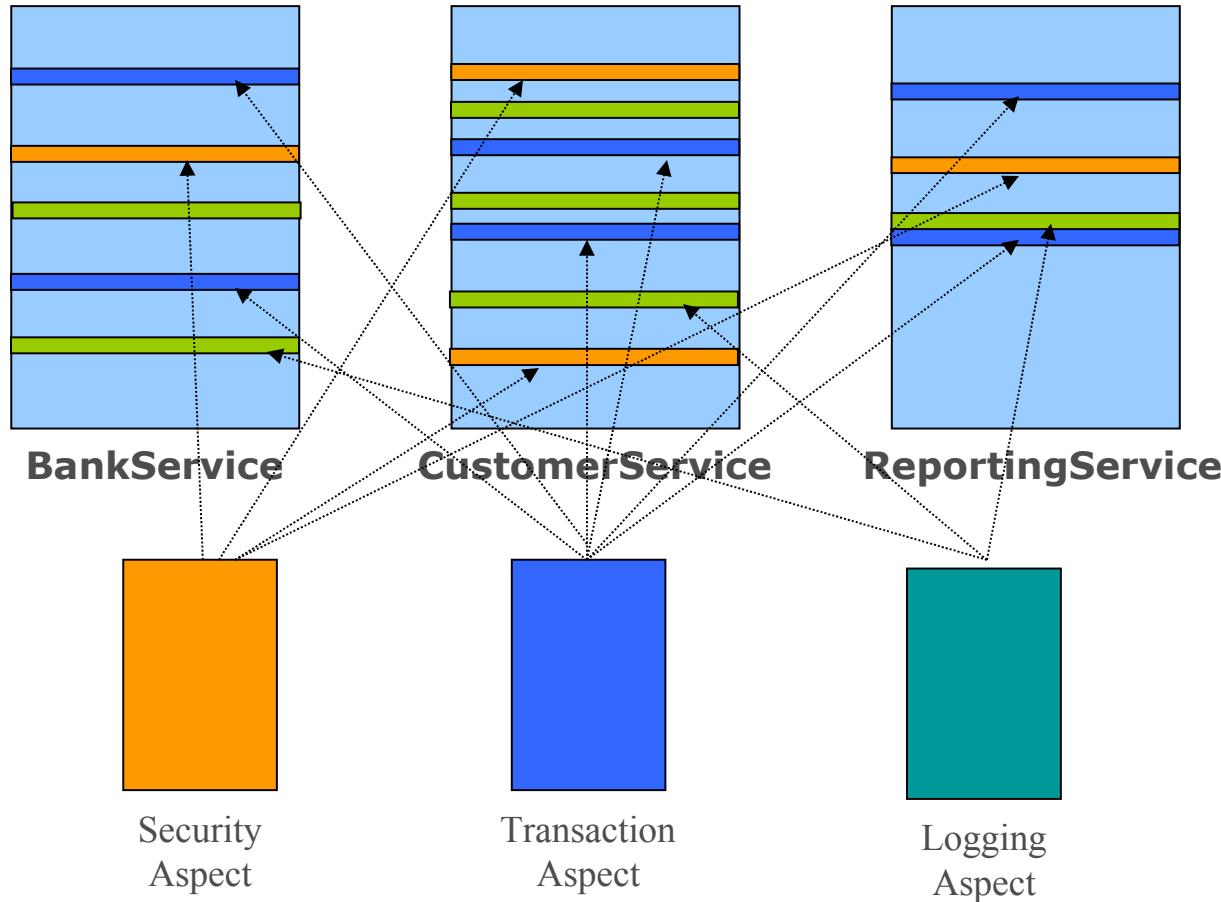
# Aspect Oriented Programming (AOP)

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

# How AOP Works

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

# System Evolution: AOP based



# Leading AOP Technologies

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



See: **Spring Framework Reference – Aspect Oriented Programming**  
<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#aop>

# Topics in this session

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

# Core AOP Concepts

- Join Point
  - A point in the execution of a program such as a method call or exception thrown
- Pointcut
  - An expression that selects one or more Join Points
- Advice
  - Code to be executed at each selected Join Point
- Aspect
  - A module that encapsulates pointcuts and advice
- Weaving
  - Technique by which aspects are combined with main code

# Topics in this session

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

# AOP Quick Start

- Consider this basic requirement

Log a message every time a property is about to change

- How can you use AOP to meet it?

# An Application Object Whose Properties Could Change

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

    public SimpleCache(String beanName) { name = beanName; }

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

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

    ...
    public String toString() { return name; }      // For convenience later
}
```

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

# Implement the Aspect

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

# Configure Aspect as a Bean

Configures Spring  
to apply `@Aspect`  
to your beans

*Using Java*

```
@Configuration  
@EnableAspectJAutoProxy  
@ComponentScan(basePackages="com.example")  
public class AspectConfig {  
    ...  
}
```

OR

*Using XML*

```
<beans>  
    <aop:aspectj-autoproxy />  
  
    <context:component-scan base-package="com.example" />  
</beans>
```

# Include the Aspect Configuration

```
@Configuration  
@Import(AspectConfig.class) ←  
public class MainConfig {  
  
    @Bean  
    public Cache cacheA() { return new SimpleCache("cacheA"); }  
  
    @Bean  
    public Cache cacheB() { return new SimpleCache("cacheB"); }  
  
    @Bean  
    public Cache cacheC() { return new SimpleCache("cacheC"); }  
}
```

Include aspect configuration

Could also use XML to create SimpleCache beans

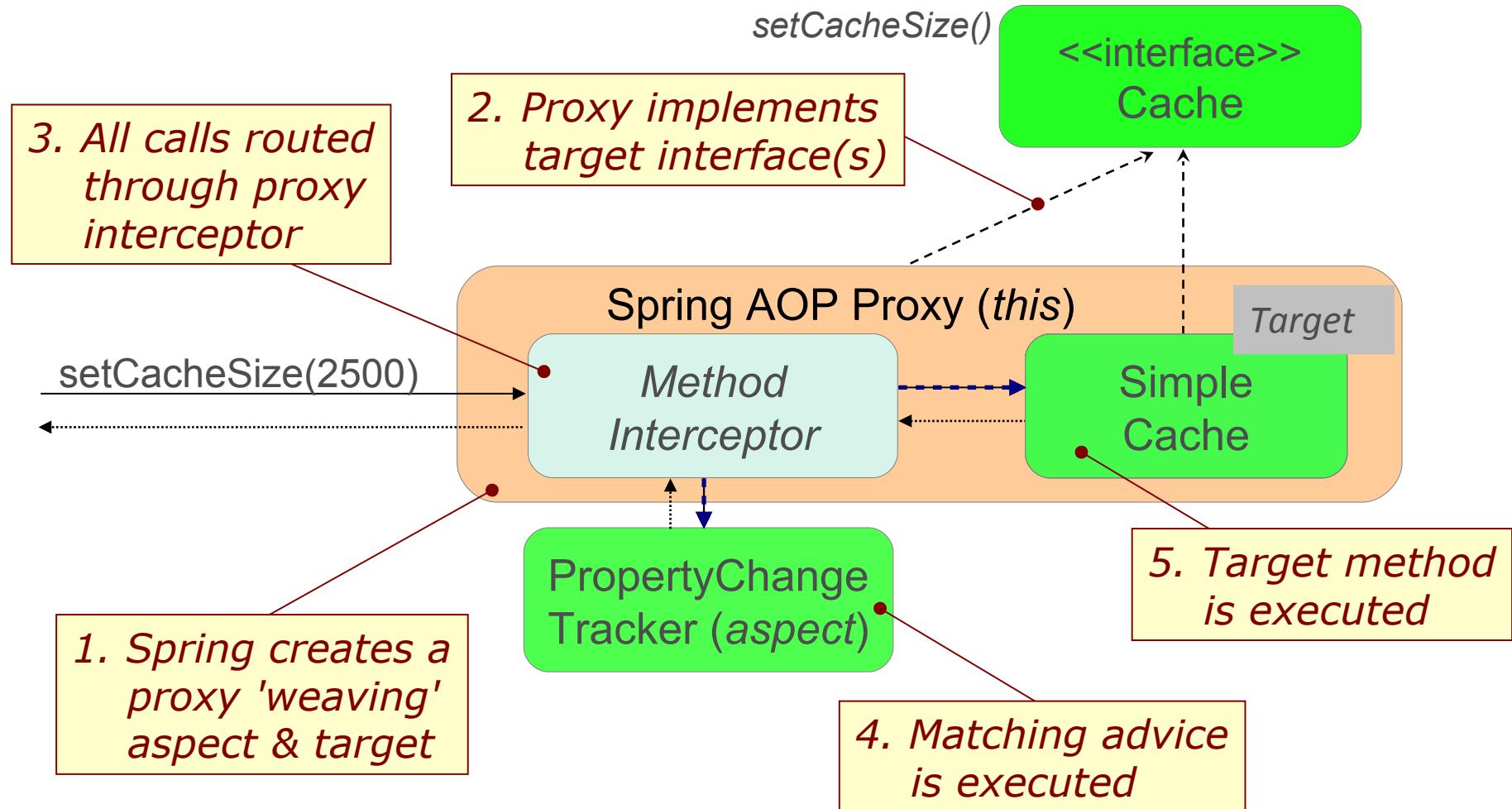
# Test the Application

```
ApplicationContext context = SpringApplication.run(MainConfig.class);
```

```
@Autowired @Qualifier("cacheA");
private Cache cache;
...
cache.setCacheSize(2500);
```

INFO: Property about to change...

# How Aspects are Applied



# Tracking Property Changes – With Context

- Context provided by the *JoinPoint* parameter

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

Context about the intercepted point

toString() returns bean-name

INFO: setCacheSize about to change to 2500 on cacheA

# Topics in this session

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

# Defining Pointcuts

- Spring AOP uses AspectJ's pointcut expression language
  - For selecting where to apply advice
- Complete expression language reference available at
  - <http://www.eclipse.org/aspectj/docs.php>
- Spring AOP supports a practical subset



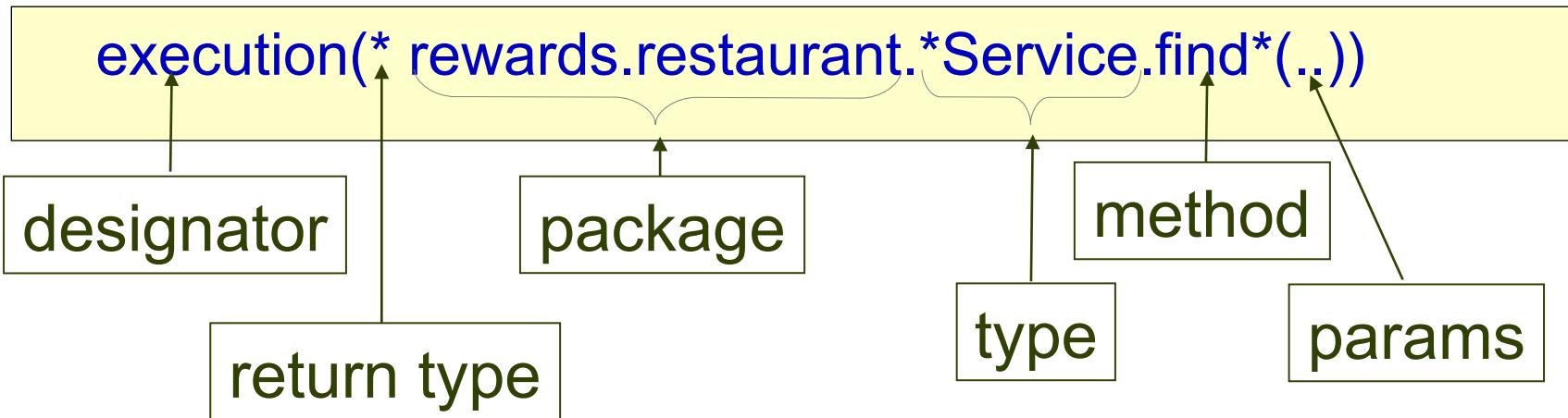
See: Spring Framework Reference – Declaring a Pointcut

<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#aop-pointcuts>

# Common Pointcut Designator

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

# Writing Expressions



# Execution Expression Examples

`execution(void send*(String))`

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

`execution(* send(*)`

- Any method named send that takes a single parameter

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

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

# Execution Expression Examples

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

- Any visible void method in the MessageServiceImpl class

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

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

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

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

# Execution Expression Examples

## – Working with Packages

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

- There is one directory between rewards and restaurant

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

- There may be several directories between rewards and restaurant

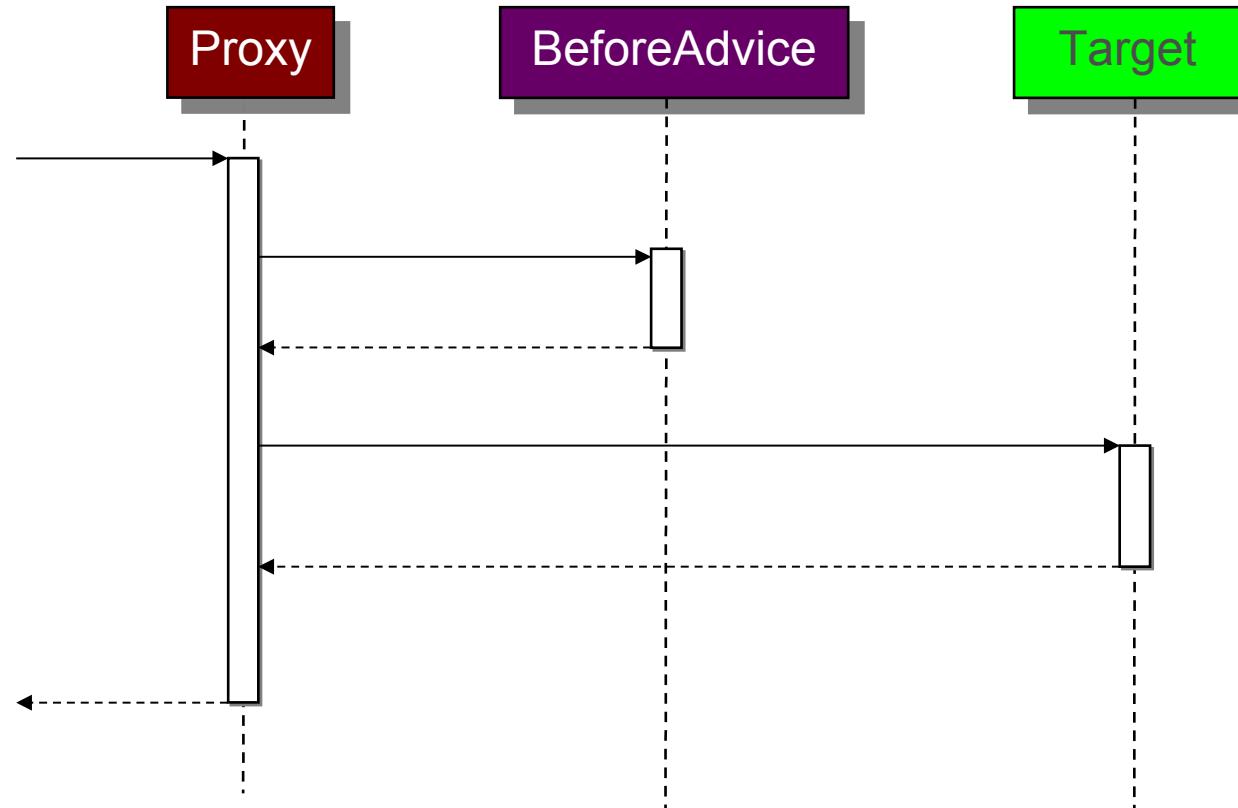
`execution(* *..restaurant.*.*(..))`

- Any sub-package called restaurant

# Topics in this session

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

# Advice Types: Before



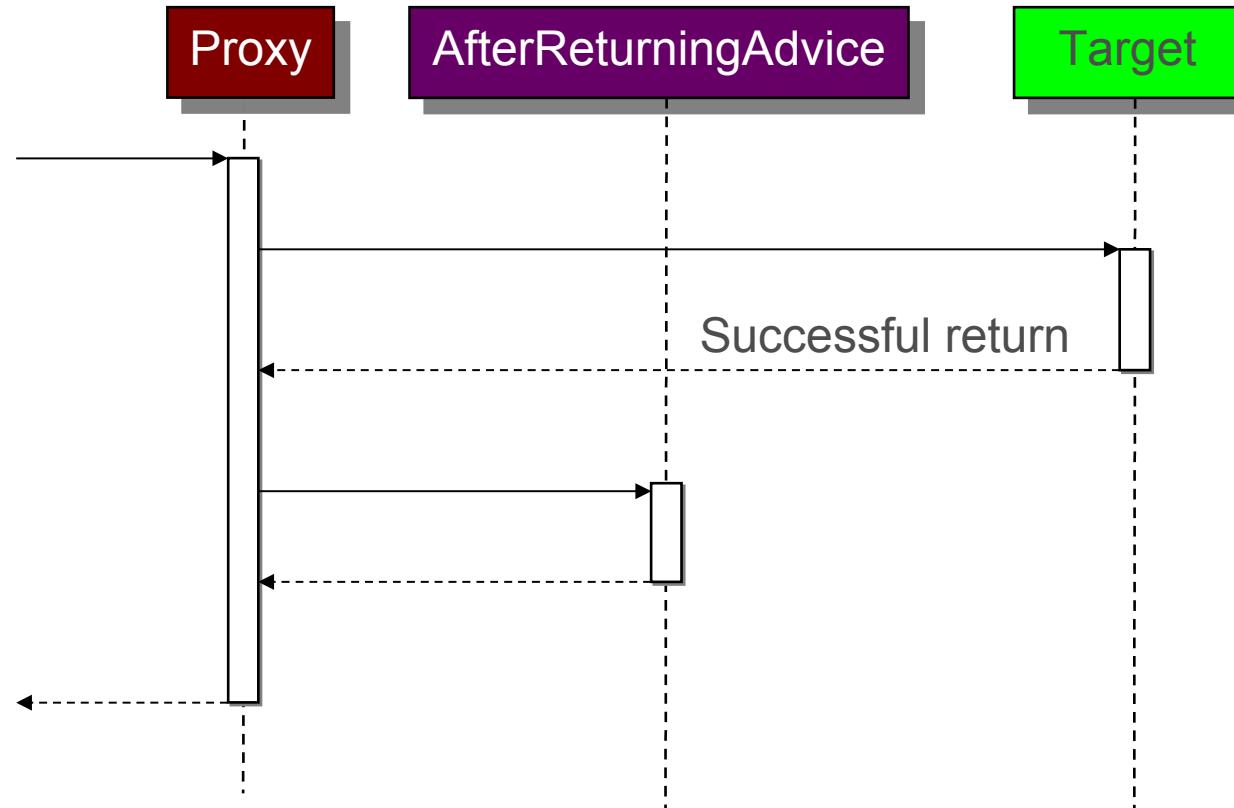
# Before Advice Example

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

Track calls to all setter methods

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

# Advice Types: After Returning



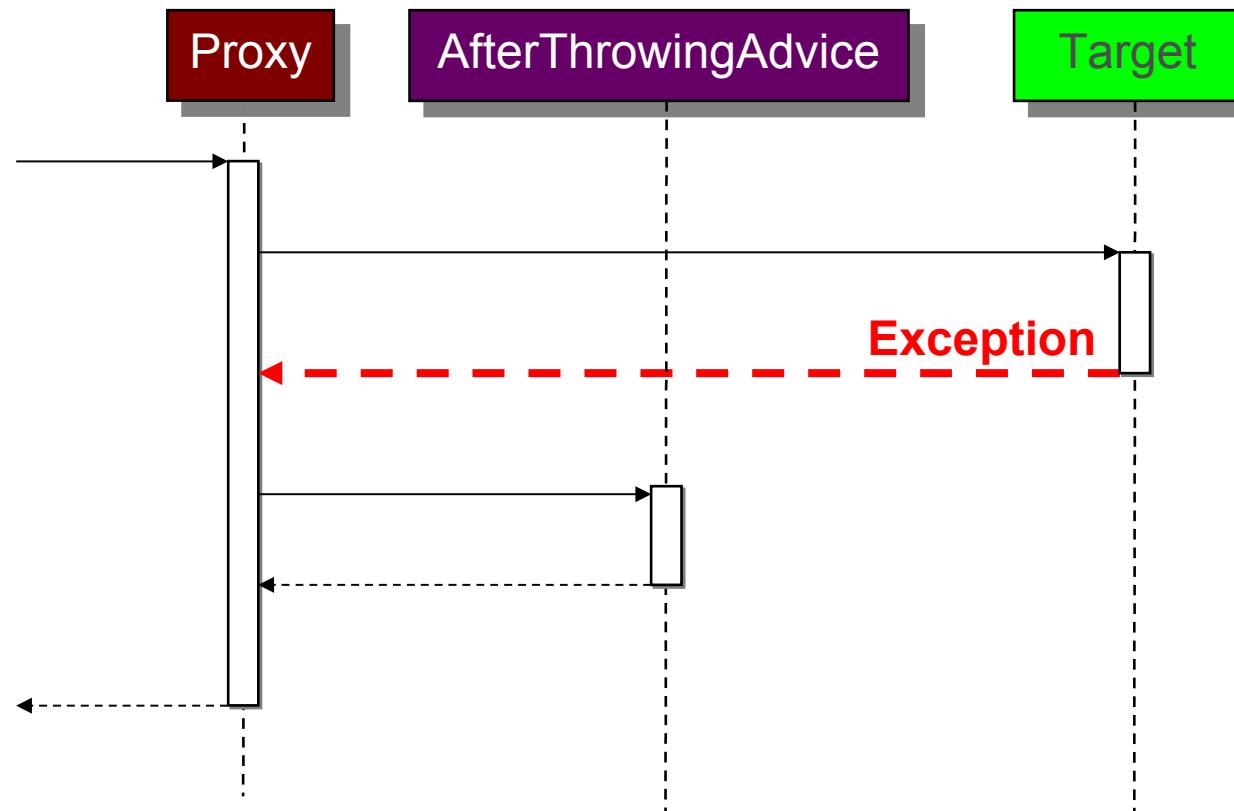
# After Returning Advice - Example

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

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

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

# Advice Types: After Throwing



# After Throwing Advice - Example

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

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

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

# After Throwing Advice - Propagation

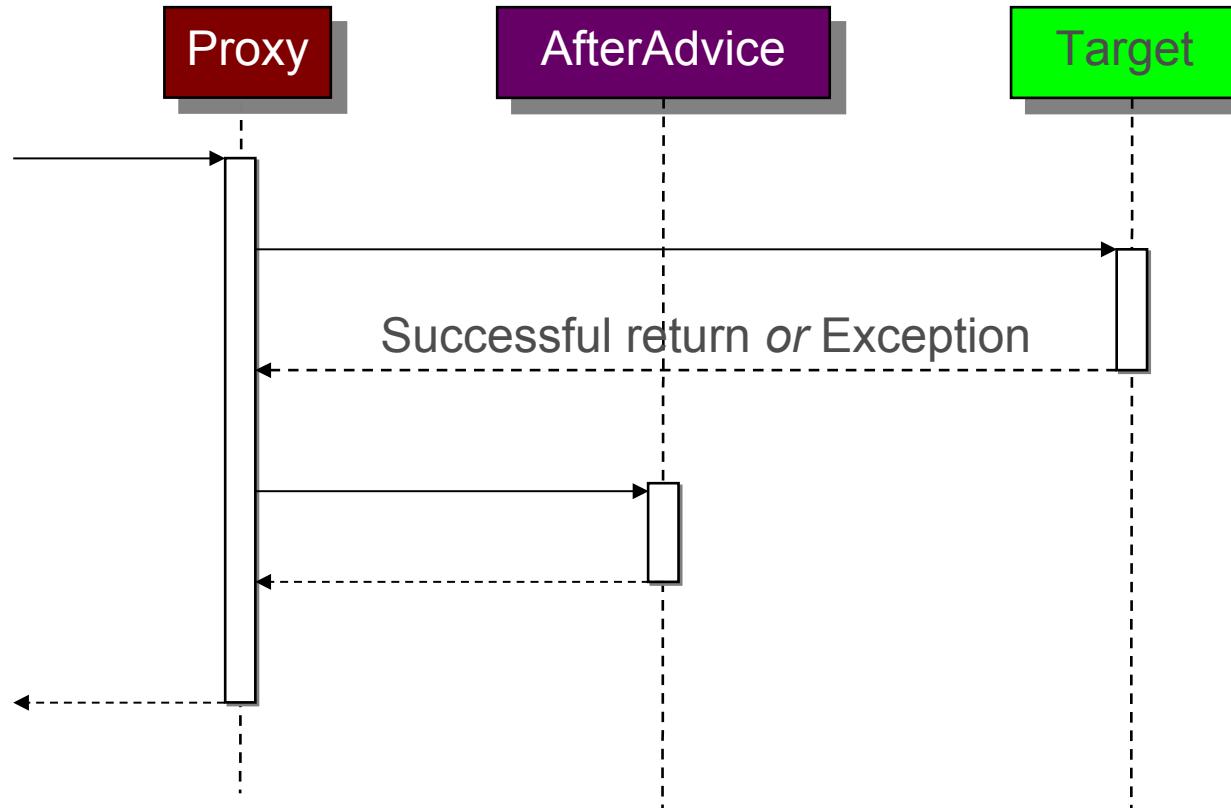
- The @AfterThrowing advice will not stop the exception from propagating
  - However it can throw a different type of exception

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



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

# Advice Types: After



# After Advice Example

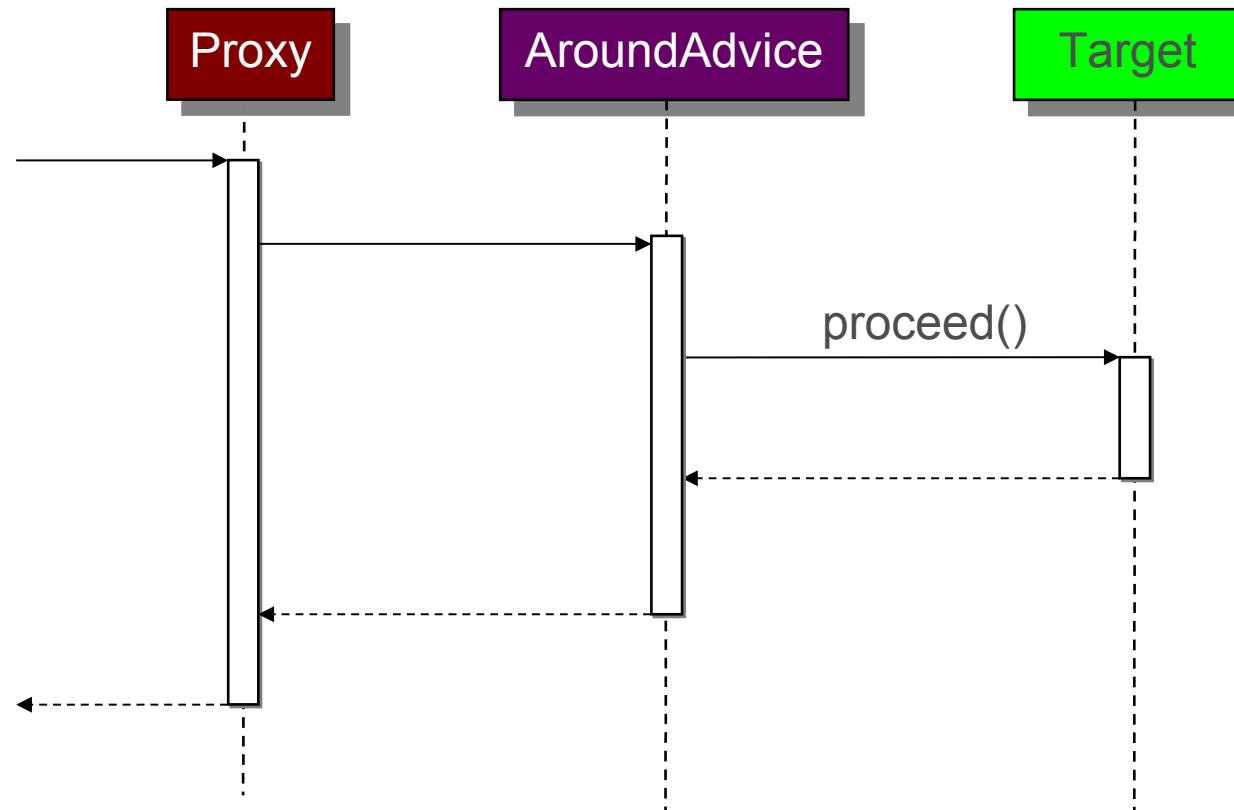
- Use `@After` annotation
  - Called regardless of whether an exception has been thrown by the target or not

Track calls to all update methods

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

We don't know how the method terminated

# Advice Types: Around



# Around Advice Example

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

Cache values returned by cacheable services

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

# Alternative Spring AOP Syntax - XML

- XML Based Alternative to @Annotations
  - More centralized configuration
- Approach
  - Aspect logic defined Java
  - Aspect configuration in XML
    - Uses the `aop` namespace

# Tracking Property Changes - Java Code

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



Aspect is a Plain Java Class with no annotations

# Tracking Property Changes - XML Configuration

- XML configuration uses the `aop` namespace

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

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

# Limitations of Spring AOP

- Can only advise *non-private* methods
- Can only apply aspects to *Spring Beans*
- Limitations of weaving with proxies
  - When using proxies, suppose method a() calls method b() on the *same* class/interface
    - advice will *never* be executed for method b()

# Lab

Developing Aspects using Spring AOP

# Topics in this session

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

# Named Pointcuts in XML

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

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

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

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

# Named Pointcut Annotation

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

The method *name* becomes the pointcut ID.  
The method is *not* executed.

# Named Pointcuts

- Expressions can be externalized

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

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

Fully-qualified pointcut name

# Named pointcuts - Summary

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

# Topics in this session

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

# Context Selecting Pointcuts

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

# Context Selecting Example

- Consider this basic requirement

Log a message every time Server is about to start

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

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

# Without Context Selection

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

```
@Before("execution(void example.Server.start(java.util.Map))")
public void logServerStartup(JoinPoint jp) {
    // A 'safe' implementation would check target type
    Server server = (Server) jp.getTarget();
    // Don't assume args[0] exists
    Object[] args= jp.getArgs();
    Map map = args.length > 0 ? (Map) args[0] : new HashMap();
    logger.info( server + " starting – params: " + map);
}
```

# With Context Selection

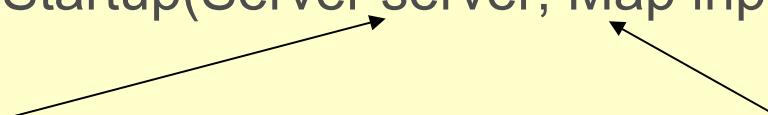
- Best practice: use context selection
  - Method attributes are bound automatically
  - Types must match or advice skipped

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

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

# Context Selection - Named Pointcut

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

...  
}  


'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) {}
```

# 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**

# Pointcut Expression Examples using Annotations

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



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

# AOP and annotations - Example

- Use of the *annotation()* designator

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

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

# AOP and Annotations – Named pointcuts

- Same example using a named-pointcut

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

# Summary

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

# Introduction to Data Management with Spring

## Implementing Data Access and Caching

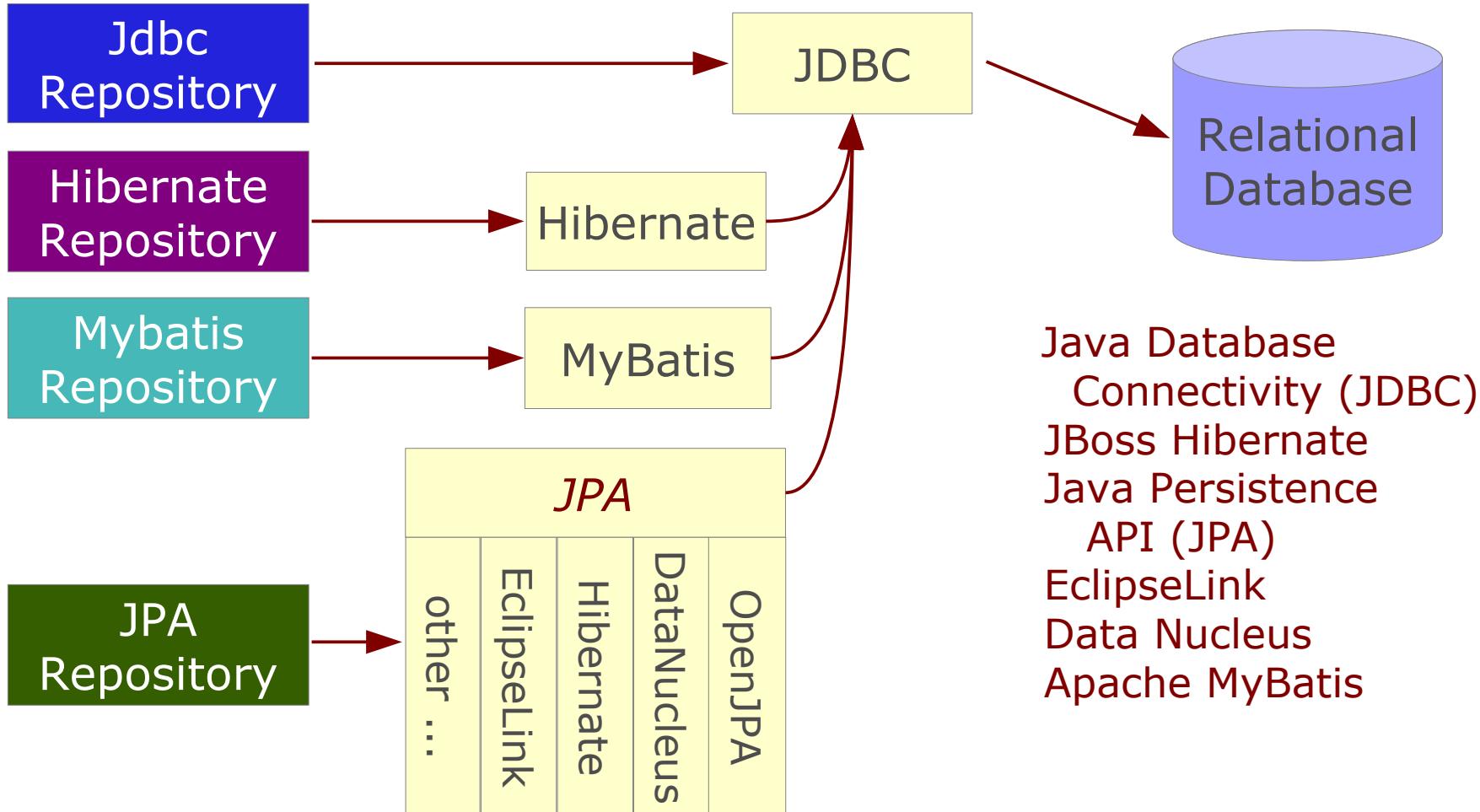
Spring's Role in Supporting Data Access in an Enterprise Application

# Topics in this Session

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

# Spring Resource Management Works Everywhere

Works consistently with leading data access technologies

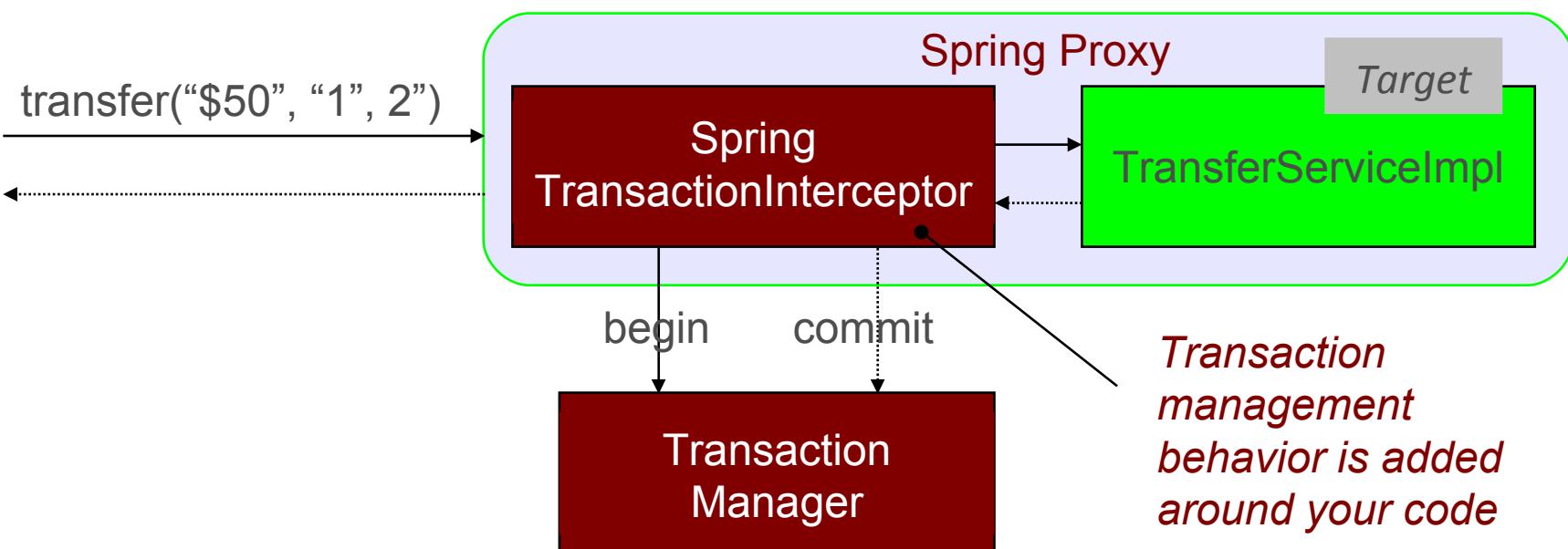


# Key Resource Management Features

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

# Declarative Transaction Management

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



# The Resource Management Problem

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

# Template Design Pattern

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

# Where are my Transactions?

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

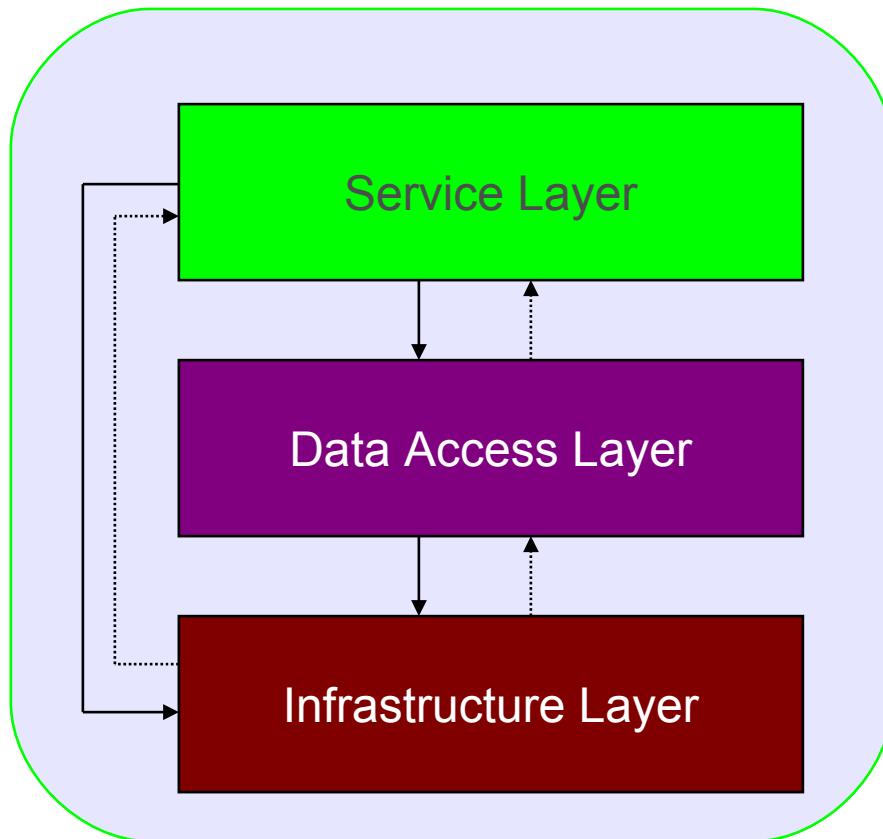
```
DataSourceUtils.getConnection(dataSource)
```
  - Hibernate sessions, JTA (Java EE) work similarly



# Data Access in a Layered Architecture

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

# Layered Application Architecture



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

# Topics in this Session

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

# Exception Handling

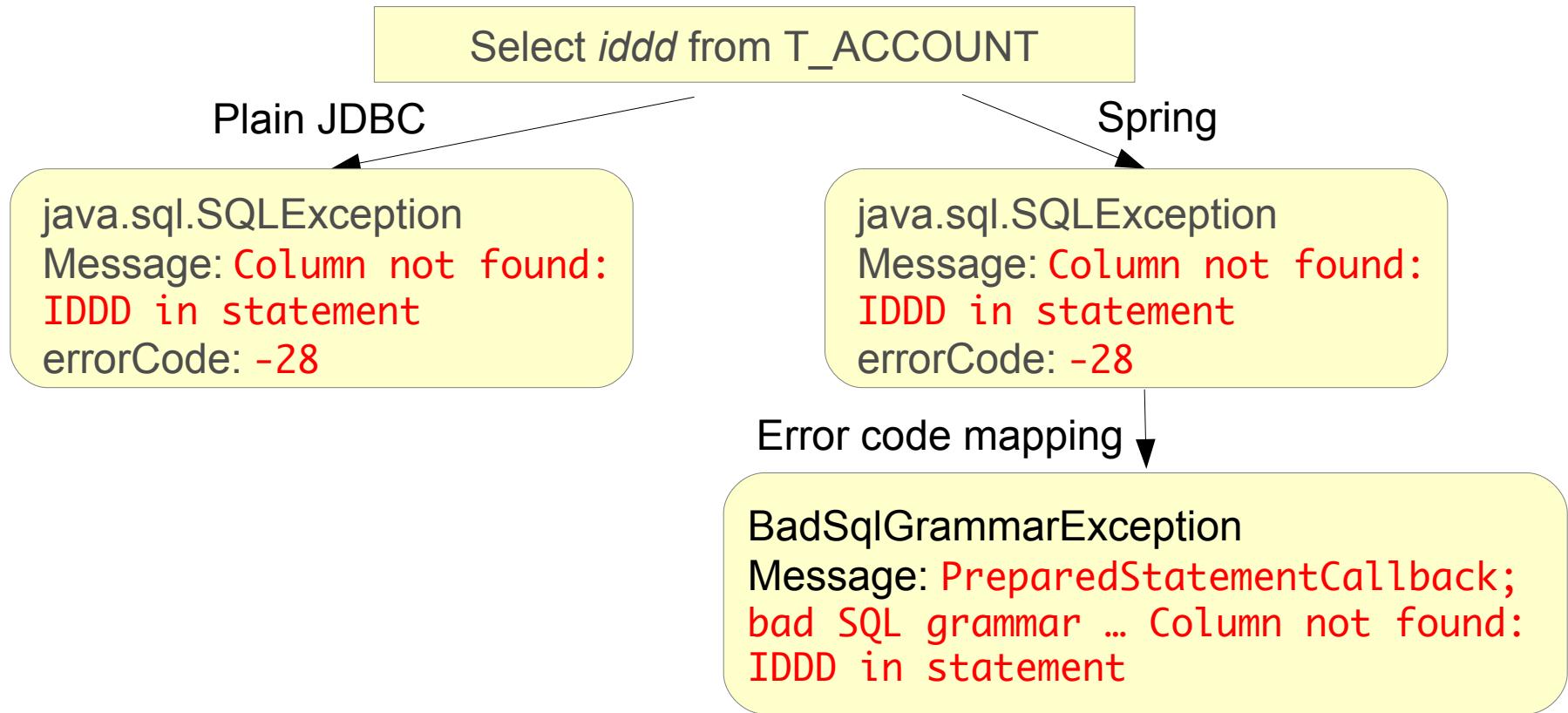
- Checked Exceptions
  - Force developers to handle errors
    - But if you can't handle it, must declare it
  - **Bad:** intermediate methods must declare exception(s) from *all* methods below
    - A form of tight-coupling
- Unchecked Exceptions
  - Can be thrown up the call hierarchy to the best place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



# Data Access Exceptions

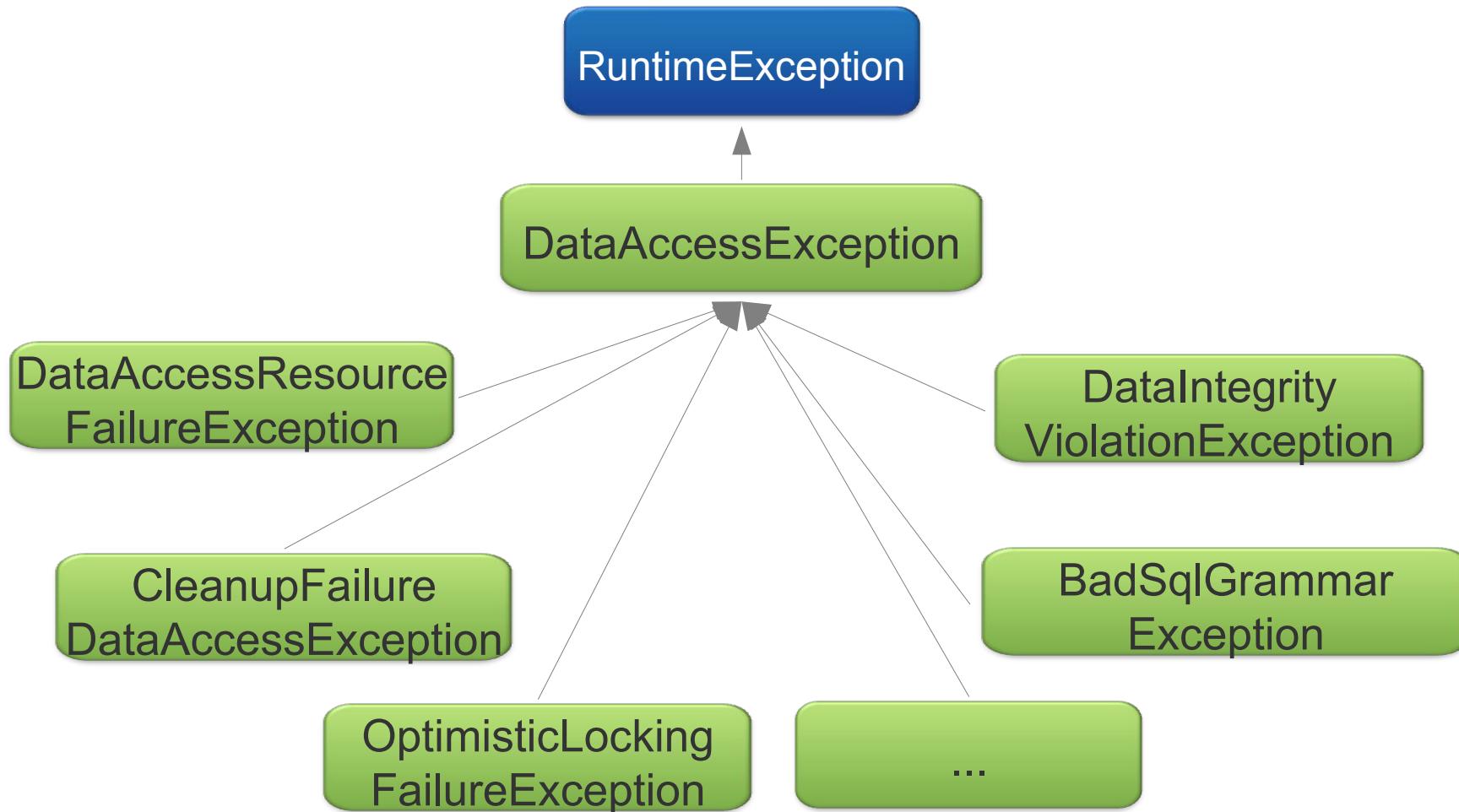
- SQLException
  - Too general – one exception for every database error
  - Calling class 'knows' you are using JDBC
  - Tight coupling
- Would like a hierarchy of exceptions
  - Consistent among all Data Access technologies
  - Needs to be unchecked
  - Not just one exception for everything
- Spring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

# Example: *BadSqlGrammarException*



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

# Spring Data Access Exceptions



# Topics in this Session

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

# JDBC Namespace

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

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

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

In memory database  
(created at startup)

# JDBC Namespace

- Allows populating other DataSources, too

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

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

Initializes an external database

# Topics in this Session

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

# About Caching

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

# Caching Support

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



See: Spring Framework Reference – Cache Abstraction  
<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#cache>

# Caching with @Cacheable

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

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

# Caching via Annotations

```
public class BookService {
```

Use 'books' cache

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

Only cache if condition true

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

use object  
property

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

custom key  
generator

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

clear cache *before* method invoked

# Enabling Caching Proxy

- Caching must be enabled ...

```
@Configuration  
@EnableCaching  
public class MyConfig {  
    @Bean  
    public BookService bookService() { ... }  
}
```

OR

```
<cache:annotation-driven />  
  
<bean id="bookService" class="example.BookService" />
```

# Pure XML Cache Setup

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

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

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

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

XML Cache Setup – no @Cachable

# Setup Cache Manager

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

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

Or use Java Config,  
but less verbose in XML

Concurrent Map Cache

# Third-Party Cache Managers

- Two are provided already

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

EHcache

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

 EHCache

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

Pivotal Gemfire  
Cache

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

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

 GEMFIRE®

# Spring Gemfire Project



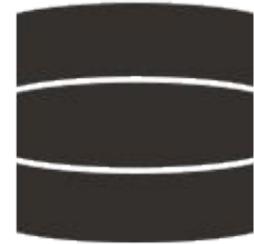
- GemFire configuration in Spring config files
  - Also enables configuration injection for environments
- Features
  - Exception translation
  - GemfireTemplate
  - Transaction management (GemfireTransactionManager)
  - Injection of transient dependencies during deserialization
  - *Gemfire Cache Manager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

**GEMFIRE®**

# Topics in this Session

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

# Not Only SQL!



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

**SPRING DATA**

# So Many Databases ...

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



# Summary

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



# Introduction to Spring JDBC

## Using JdbcTemplate

Simplifying JDBC-based data-access with Spring

# 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



See: **Spring Framework Reference – Data access with JDBC**  
<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jdbc>

# 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()) {
            personList.add(new Person(rs.getString("first_name"), ...));
        }
    } catch (SQLException e) { /* ??? */ }
    finally {
        try {
            conn.close();
        } catch (SQLException e) { /* ??? */ }
    }
    return personList;
}
```

# Redundant, Error Prone Code

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

The bold matters - the rest is boilerplate

# Poor Exception Handling

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

What can  
you do?

# Topics in this session

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

# Spring's JdbcTemplate

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

# JdbcTemplate in a Nutshell

```
int count = jdbcTemplate.queryForObject(  
    "SELECT COUNT(*) FROM CUSTOMER", Integer.class);
```

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

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

# Creating a JdbcTemplate

- Requires a DataSource

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

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

# When to use JdbcTemplate

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

# Implementing a JDBC-based Repository

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

No try / catch needed  
(unchecked exception)



# Querying with JdbcTemplate

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

# Query for Simple Java Types

- Query with no bind variables: *queryForObject*

```
public Date getOldest() {  
    String sql = "select min(dob) from PERSON";  
    return jdbcTemplate.queryForObject(sql, Date.class);  
}  
  
public long getPersonCount() {  
    String sql = "select count(*) from PERSON";  
    return jdbcTemplate.queryForObject(sql, Long.class);  
}
```



*queryForInt, queryForLong deprecated since Spring 3.2, just as easy to queryForObject instead (API improved in Spring 3)*

# Query With Bind Variables

- Can query using bind variables: ?
  - Note the use of a variable argument list

```
private JdbcTemplate jdbcTemplate;  
  
public int getCountOfNationalsOver(Nationality nationality, int age) {  
    String sql = "select count(*) from PERSON " +  
        "where age > ? and nationality = ?";  
  
    return jdbcTemplate.queryForObject  
        (sql, Integer.class, age, nationality.toString());  
}
```

Bind to first ?

Bind to second ?

# Generic Queries

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

# Querying for Generic Maps (1)

- Query for a single row

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

- returns:

Map { ID=1, FIRST\_NAME="John", LAST\_NAME="Doe" }

A Map of [Column Name | Field Value] pairs

# Querying for Generic Maps (2)

- Query for multiple rows

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

- returns:

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

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

# Domain Object Queries

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

# RowMapper

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

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

# Querying for Domain Objects (1)

- Query for single row with JdbcTemplate

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

No need to cast

Maps rows to Person objects

Parameterizes return type

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

# Querying for Domain Objects (2)

- Query for multiple rows

No need to cast

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

Same row mapper can be used

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

# Querying for Domain Objects (3)

- Simplify using Java 8 Lambda Expressions
  - No need for Mapper class
  - Use inline code instead

```
public List<Person> getAllPersons() {  
    return jdbcTemplate.query(  
        "select first_name, last_name from PERSON",  
        (rs, rowNum) -> {  
            return new Person(rs.getString("first_name"),  
                rs.getString("last_name"));  
        });  
}
```

Replace RowMapper  
by *lambda*

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

# RowCallbackHandler

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

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

# Using a RowCallbackHandler (1)

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

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

# Using a RowCallbackHandler (2)

- Or using a Lambda – if *no state* needed

```
public class JdbcOrderRepository {  
    public void generateReport(final Writer out) {  
        // select all orders of year 2009 for a full report  
        jdbcTemplate.query("select * from order where year=?",  
            (rs) -> { out.write( rs.getString("customer") ... ); },  
            2009);  
    }  
}  
  
public interface RowCallbackHandler {  
    void processRow(ResultSet rs) throws SQLException;  
}
```

# ResultSetExtractor

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

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

# Using a ResultSetExtractor (1)

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

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

# Using a ResultSetExtractor (2)

Or using a *lambda*

```
public class JdbcOrderRepository {  
    public Order findByConfirmationNumber(String number) {  
        // execute an outer join between order and item tables  
        return jdbcTemplate.query(  
            "select...from order o, item i...conf_id = ?",
            (rs) -> {
                Order order = null;
                while (rs.next()) {
                    if (order == null)
                        order = new Order(rs.getLong("ID"), rs.getString("NAME"), ...);

                    order.addItem(mapItem(rs));
                }
                return order;
            },
            number);
    }
}
```

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

# Summary of Callback Interfaces

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

# Inserts and Updates (1)

- Inserting a new row
  - Returns number of rows modified

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

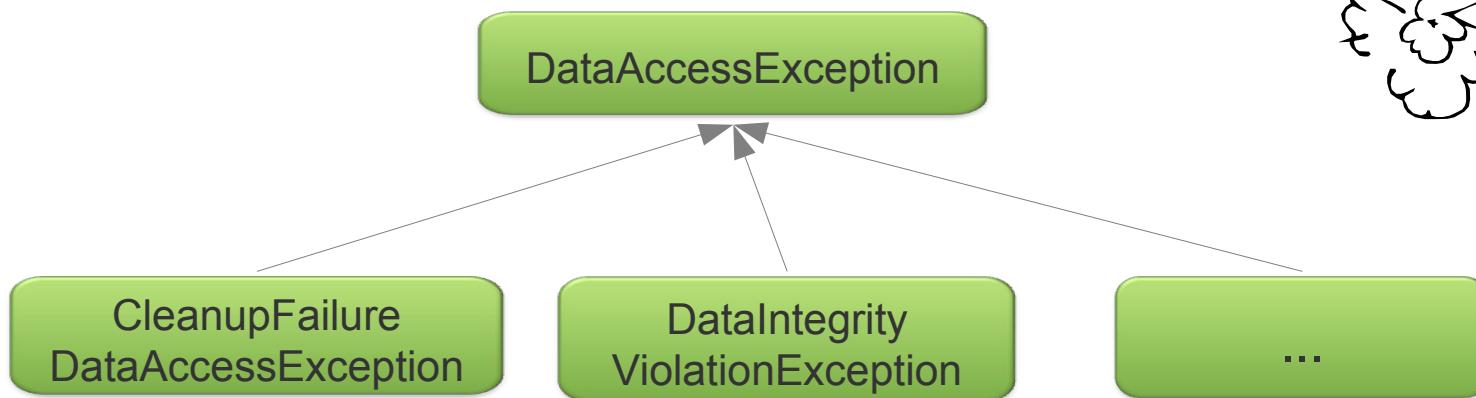
# Inserts and Updates (2)

- Updating an existing row

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

# Exception Handling

- The JdbcTemplate transforms SQLExceptions into DataAccessExceptions



*DataAccessException* hierarchy was discussed in module “Introduction to Data Access”. You can refer to it for more information on this topic.

# Lab

Reimplementing repositories using  
Spring's JdbcTemplate

# Transaction Management with Spring

Spring's Consistent Approach

Transactional Proxies and @Transactional

# Topics in this session

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

# What is a Transaction?

- A set of tasks which take place as a single, atomic, consistent, isolated, durable operation.

# Why use Transactions? To Enforce the ACID Principles:

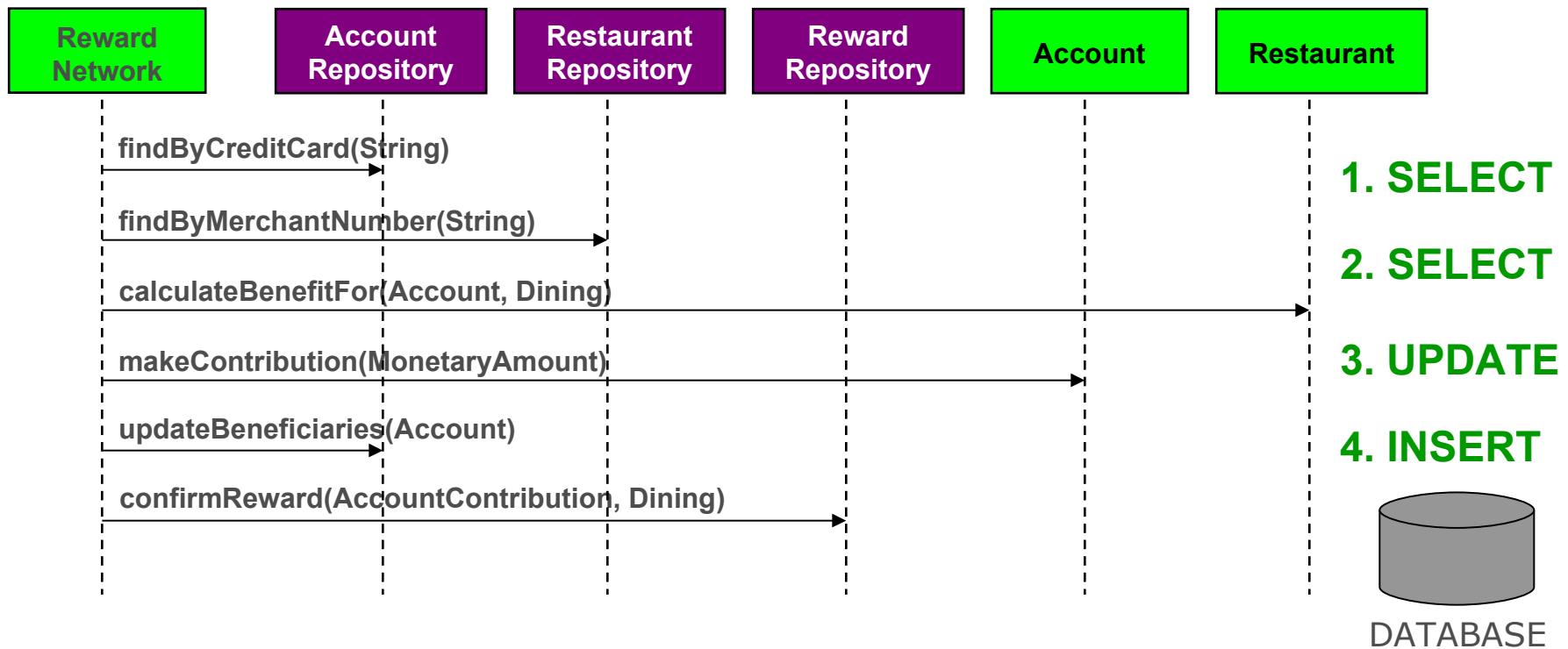
- **A**tomic
  - Each unit of work is an all-or-nothing operation
- **C**onsistent
  - Database integrity constraints are never violated
- **I**solated
  - Isolating transactions from each other
- **D**urable
  - Committed changes are permanent

# Transactions in the RewardNetwork

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

# RewardNetwork Atomicity

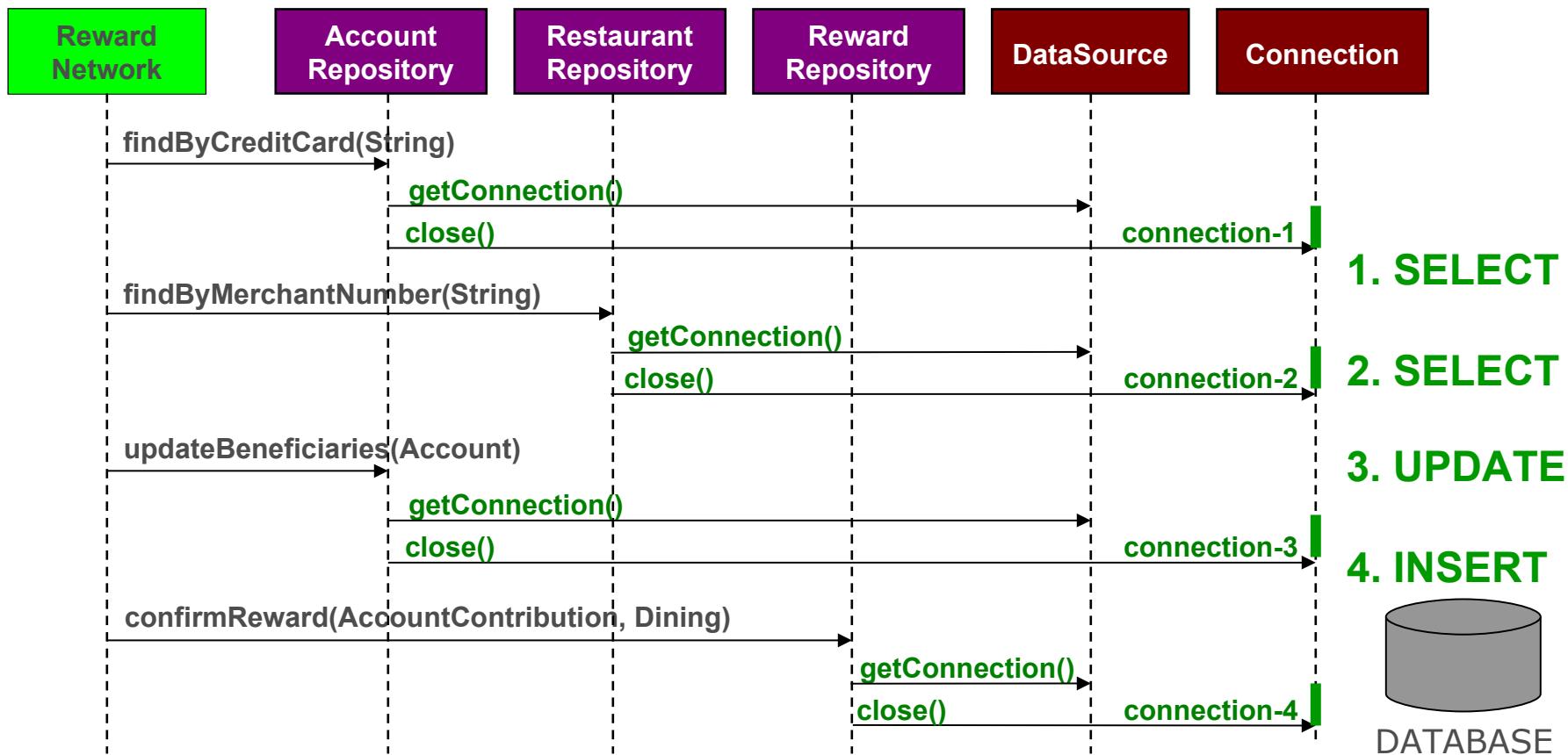
- The rewardAccountFor(Dining) unit-of-work:



# Naïve Approach: Connection per Data Access Operation

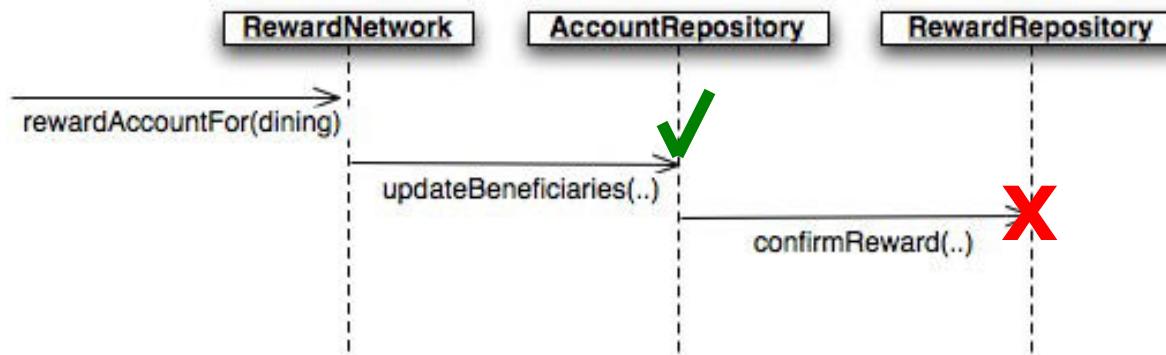
- This unit-of-work contains 4 data access operations
  - Each acquires, uses, and releases a distinct Connection
- The unit-of-work is *non-transactional*

# Running non-Transactionally



# Partial Failures

- Suppose an Account is being rewarded



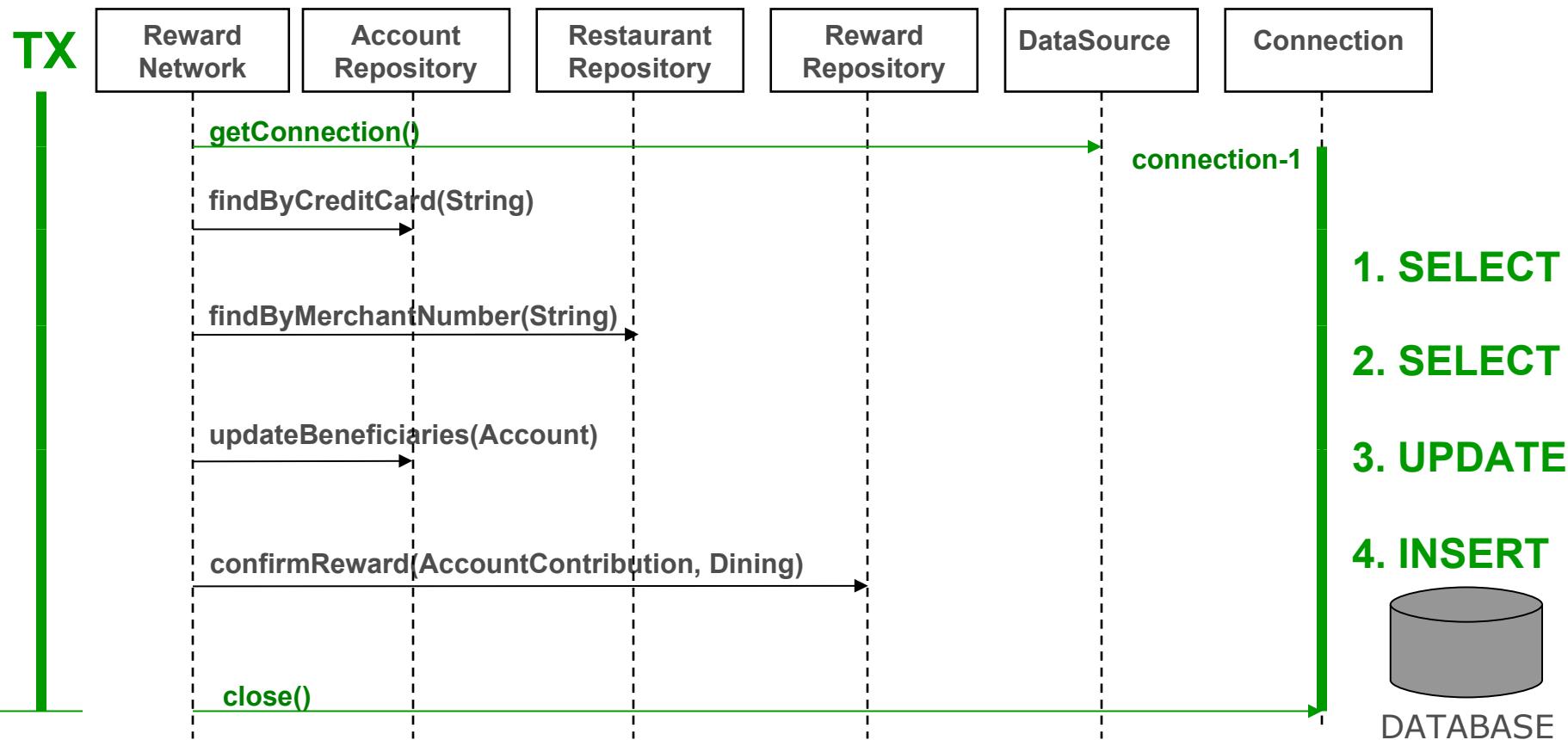
- If the beneficiaries are updated...
- But the reward confirmation fails...
- There will be no record of the reward!

The unit-of-work  
is **not atomic**

# Correct Approach: Connection per Unit-of-Work

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

# Running in a Transaction



# Topics in this session

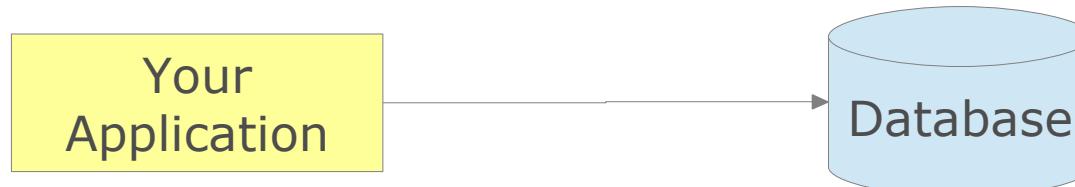
- Why use Transactions?
- **Java Transaction Management**
- Spring Transaction Management
- Annotations or XML
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

# Java Transaction Management

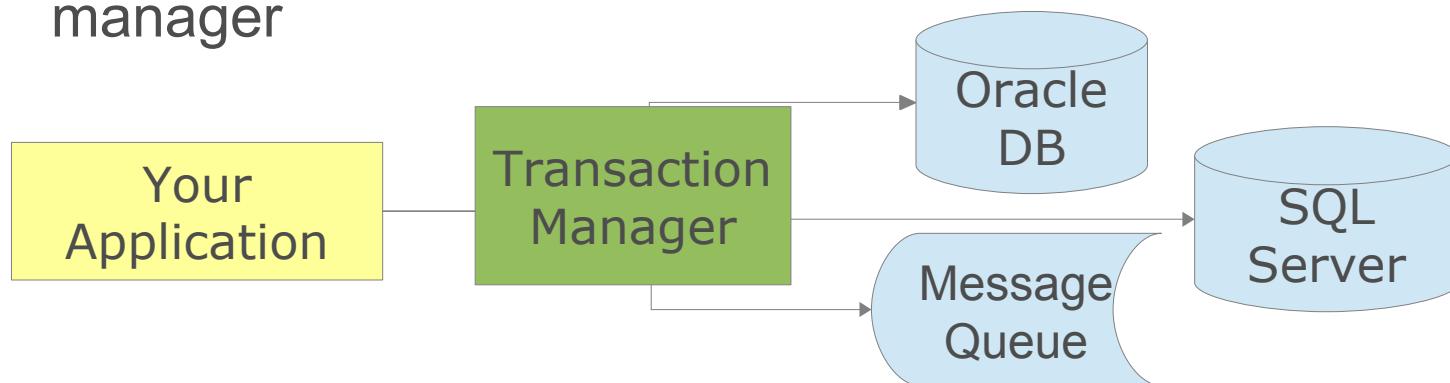
- Java has several APIs which handle transactions differently
  - JDBC, JMS, JTA, Hibernate, JPA, etc.
- Each uses program code to mark the start and end of the transaction
  - Transaction Demarcation
- Different APIs for Global vs Local transactions

# Local and Global Transaction Management

- Local Transactions – Single Resource
  - Transactions managed by underlying resource



- Global (distributed) Transactions – Multiple
  - Transaction managed by separate, dedicated transaction manager



# JDBC Transaction Management Example

```
try {  
    conn = dataSource.getConnection(); ← Specific To JDBC API  
    conn.setAutoCommit(false);  
  
    ...  
  
    conn.commit(); ← Programmatic Transaction Demarcation  
} catch (Exception e) {  
    conn.rollback();  
  
    ...  
}  
}
```

Specific To JDBC API

Programmatic Transaction Demarcation

Checked Exceptions

Code cannot 'join' a transaction already in progress  
Code cannot be used with global transaction

# JMS Transaction Management Example

```
try {  
    session = connection.createSession ( true, false );  
    ...  
    session.commit();  
} catch (Exception e) {  
    session.rollback();  
    ...  
}
```

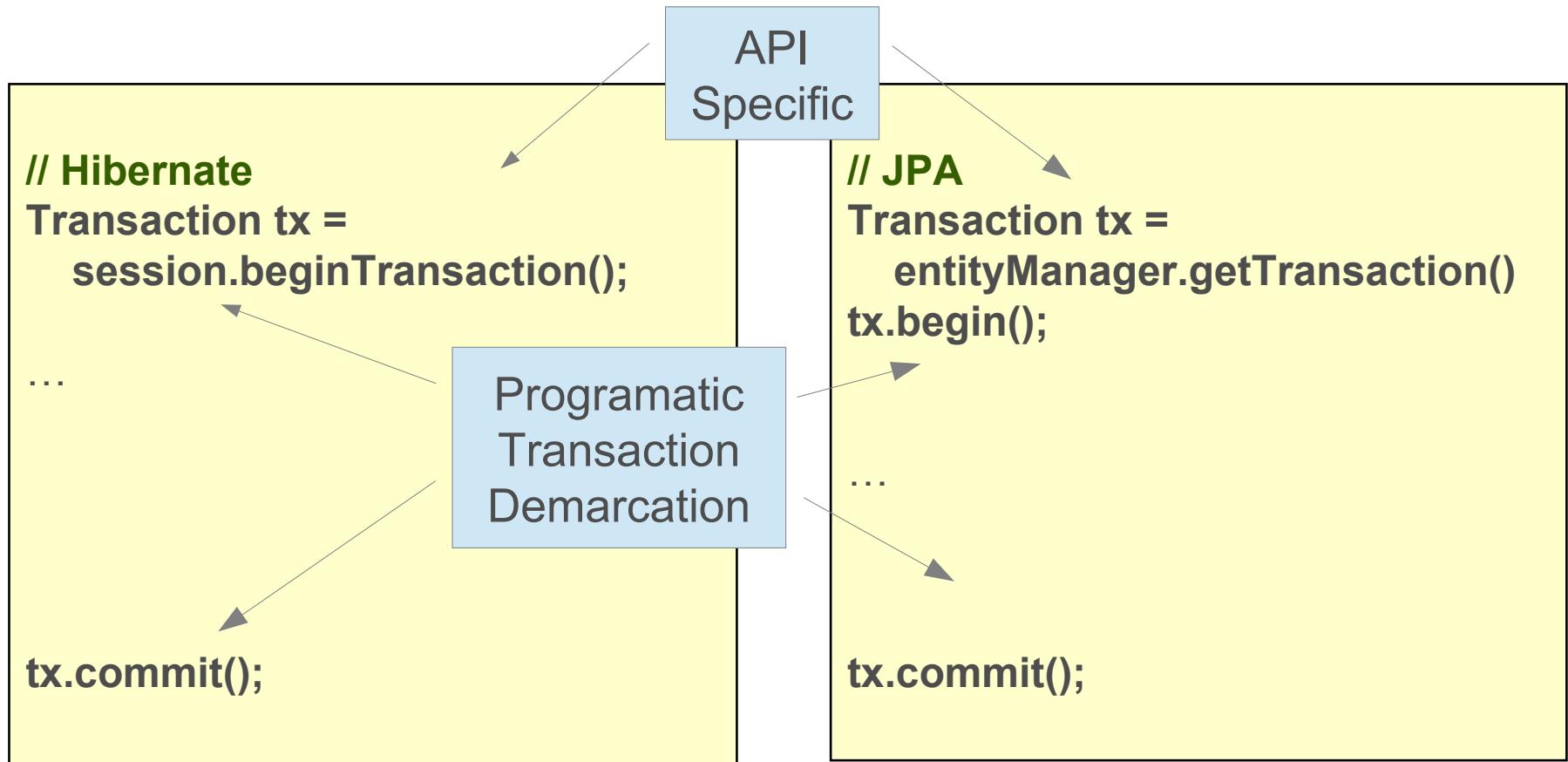
Specific To  
JMS API

Programmatic  
Transaction  
Demarcation

Checked  
Exceptions

Code cannot 'join' a transaction already in progress  
Code cannot be used with global transaction

# JPA / Hibernate Transaction Management Example



# Java Transaction API (JTA) Example

```
try {  
    UserTransaction ut =  
        (UserTransaction) new InitialContext()  
            .lookup("java:comp/UserTransaction");  
    ut.begin(); ←  
    ...  
    ut.commit(); ←  
} catch (Exception e) {  
    ut.rollback(); ←  
    ...  
}
```

Programmatic  
Transaction  
Demarcation

Checked  
Exceptions

Requires a JTA implementation:

- “Full” application server (WebSphere, WebLogic, JBoss, etc.)
- Standalone implementation (Atomikos, JTOM, etc.)

# Problems with Java Transaction Management

- Multiple APIs for different local resources
- Programmatic transaction demarcation
  - Usually located in the repository layer
  - Usually repeated (cross-cutting concern)
  - Service layer more appropriate
    - Multiple data access methods may be called within a transaction
- *Orthogonal* concerns
  - Transaction *demarcation* should be independent of transaction *implementation*

# Topics in this session

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

# Spring Transaction Management

- Spring separates transaction *demarcation* from transaction *implementation*
  - Demarcation expressed declaratively via AOP
    - Programmatic approach also available
  - **PlatformTransactionManager** abstraction hides implementation details.
    - Several implementations available
- Spring uses the same API for global vs. local.
  - Change from local to global is minor.

# Spring Transaction Management

- There are only 2 steps
  - Declare a **PlatformTransactionManager** bean
  - Declare the transactional methods
    - Using Annotations, XML, Programmatic
    - Can mix and match

# PlatformTransactionManager

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



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

# Deploying the Transaction Manager

- Pick the specific implementation

```
@Bean
```

```
public PlatformTransactionManager transactionManager() {  
    return new DataSourceTransactionManager(dataSource);  
}
```

- XML:

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

A dataSource must be defined elsewhere



Bean id “*transactionManager*” is default name. Can change it but must specify alternative name everywhere – easier not to!

# Automatic JTA Implementation Resolution

For JTA, also possible to use custom XML tag:

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

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

# @Transactional configuration using Java Configuration

- In the code:

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

- In the configuration:

```
@Configuration  
@EnableTransactionManagement  
public class TxnConfig {  
    @Bean  
    public PlatformTransactionManager transactionManager(DataSource ds);  
    return new DataSourceTransactionManager(ds);  
}
```

Defines a Bean Post-Processor  
– proxies @Transactional beans

# @Transactional configuration using XML

- In the code:

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

- In the configuration:

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

Defines a Bean Post-Processor  
– proxies @Transactional beans

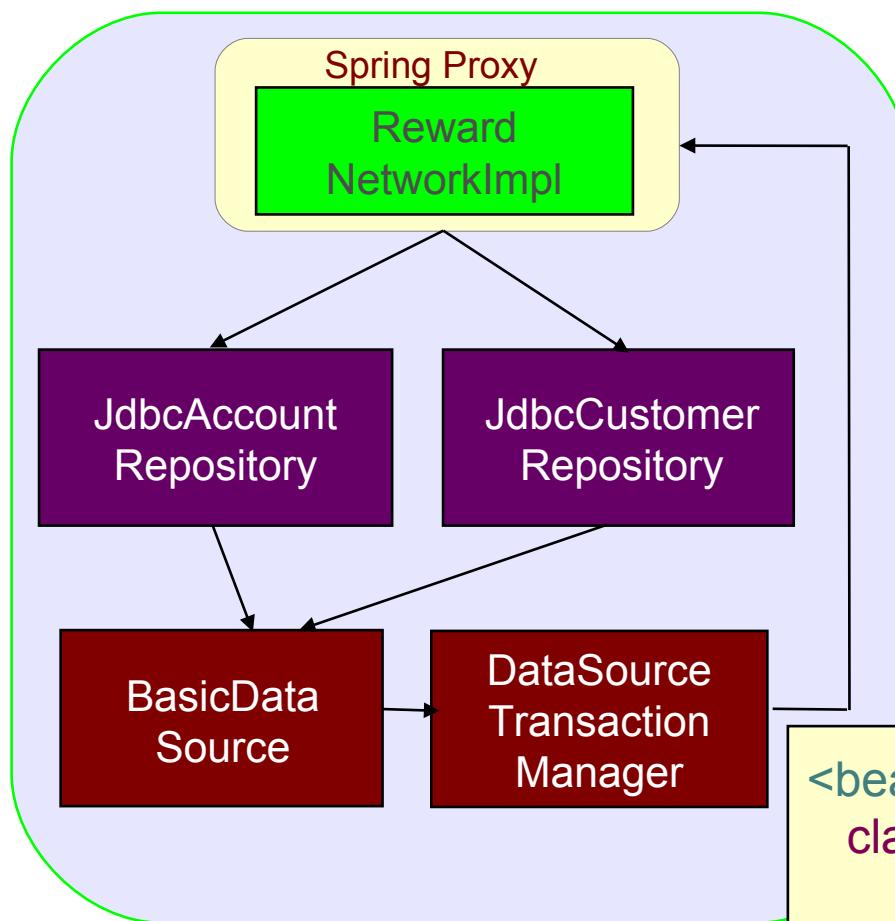
# @Transactional: what happens exactly?

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

Spring Proxy

Reward  
NetworkImpl

# Local JDBC Configuration

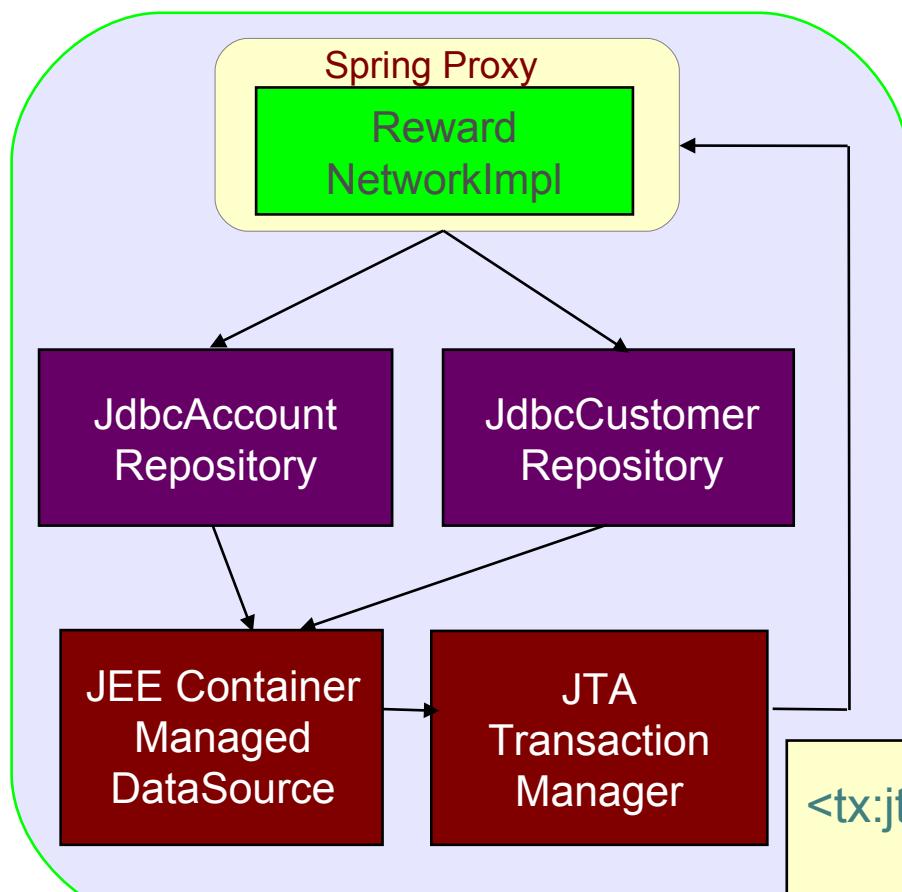


- How?
  - Define local data source
  - DataSource Transaction Manager
- Purpose
  - Integration testing
  - Deploy to Tomcat or other servlet container

```
<bean id="transactionManager"
      class="...DataSourceTransactionManager"> ...
<jdbc:embedded-database id="dataSource"> ...
```

# JDBC Java EE Configuration

No code changes  
Just configuration



- How?
  - Use container-managed datasource (JNDI)
  - JTA Transaction Manager
- Purpose
  - Deploy to JEE container

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

```
<jee:jndi-lookup id="dataSource" ... />
```

# Topics in this session

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

# @Transactional – 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(RewardConfirmantion rc) {  
        // atomic unit-of-work  
    }  
}
```



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

# @Transactional

## – Class and method levels

- Combining class and method levels

```
@Transactional(timeout=60)
```

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

default settings

```
    public RewardConfirmation rewardAccountFor(Dining d) {
```

```
        // atomic unit-of-work
```

```
}
```

```
    @Transactional(timeout=45)
```

```
    public RewardConfirmation updateConfirmation(RewardConfirmation rc) {
```

```
        // atomic unit-of-work
```

```
}
```

overriding attributes at  
the method level

# XML-based Spring Transactions

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

# Declarative Transactions: XML

```
<aop:config>
  <aop:pointcut id="rewardNetworkMethods"
    expression="execution(* rewards.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 *named* pointcut expression

Method-level configuration for transactional advice

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

# Topics in this session

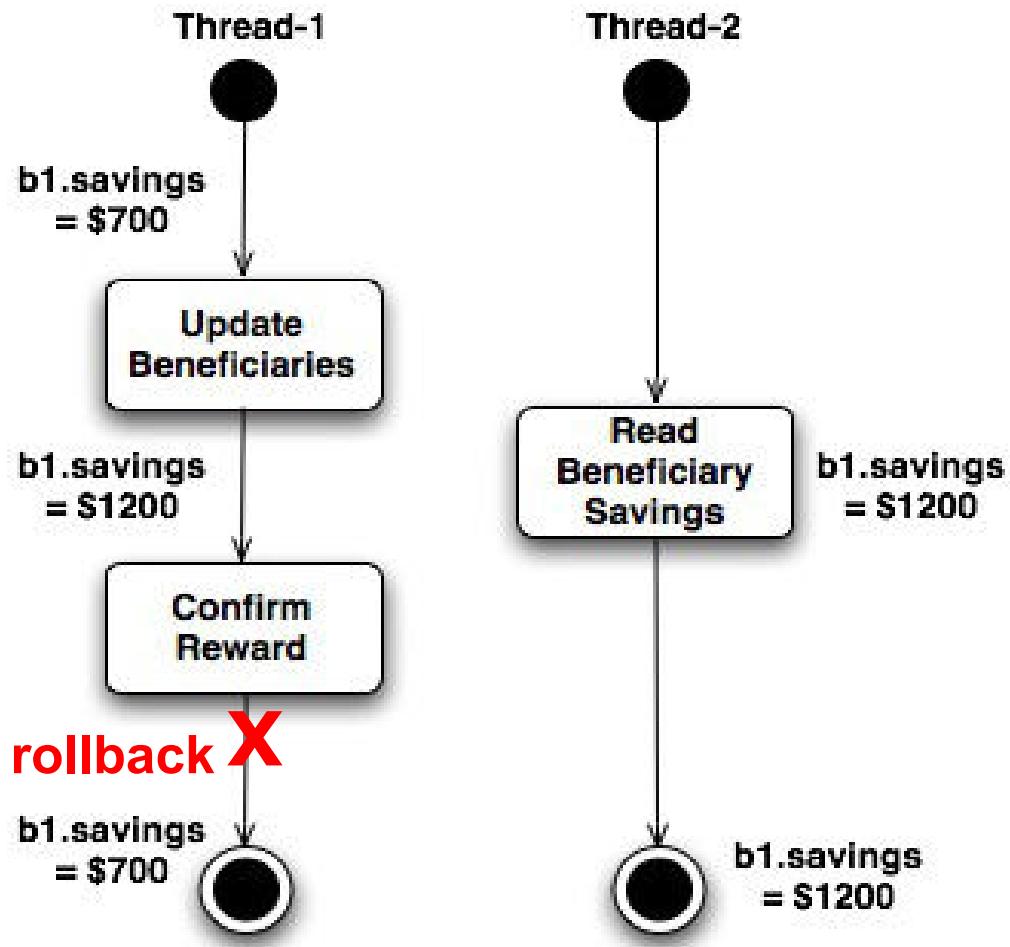
- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Annotations or XML
- **Isolation Levels**
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

# Isolation levels

- 4 isolation levels can be used:
  - READ\_UNCOMMITTED
  - READ\_COMMITTED
  - REPEATABLE\_READ
  - SERIALIZABLE
- Some DBMSs do not support all isolation levels
- Isolation is a complicated subject
  - DBMS all have differences in the way their isolation policies have been implemented
  - We just provide general guidelines

# Dirty Reads

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



# READ\_UNCOMMITTED

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

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

# READ\_COMMITTED

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

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

# Highest isolation levels

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

# Topics in this session

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

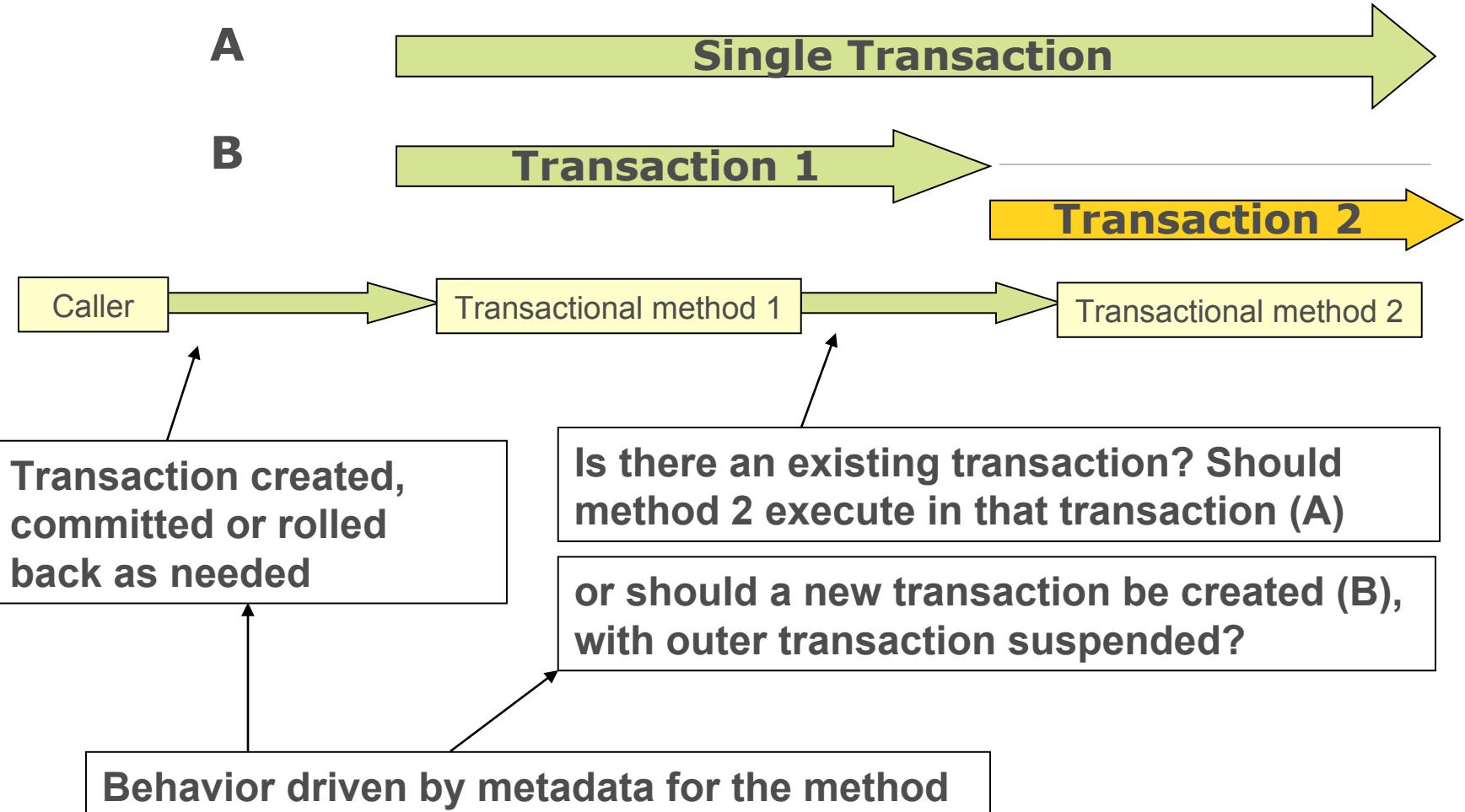
# Understanding Transaction Propagation

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

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

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

# Understanding Transaction Propagation



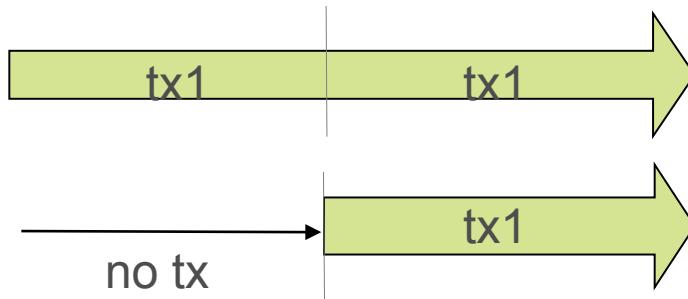
# Transaction Propagation with Spring

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

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

# REQUIRED

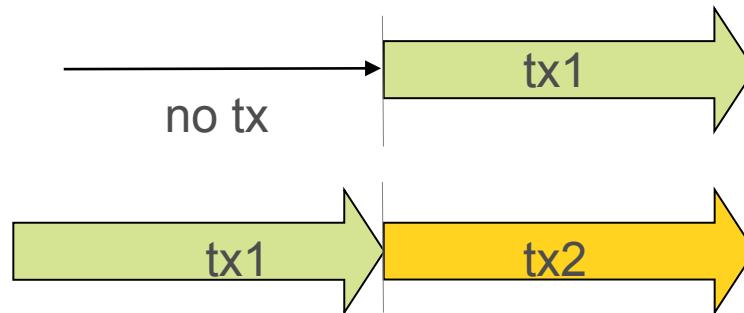
- REQUIRED
  - Default value
  - Execute within a current transaction, create a new one if none exists



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

# REQUIRES\_NEW

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



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

# Topics in this session

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

# Default behavior

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

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



Triggers a rollback

# rollbackFor and noRollbackFor

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

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

# Topics in this session

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

# @Transactional within Integration Test

- Annotate test method (or class) with `@Transactional`
  - Runs test methods in a transaction
  - Transaction will be *rolled back* afterwards
    - No need to clean up your database after testing!

```
@ContextConfiguration(classes=RewardsConfig.class)
```

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
public class RewardNetworkTest {
```

```
    @Test @Transactional
```

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

```
    ...
```

```
}
```

```
}
```

Test now  
transactional

# Advanced use of @Transactional within Integration 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 do a *commit* at the end by default

Overrides default *rollback* settings



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

# @Before vs @BeforeTransaction

```
@ContextConfiguration(locations={"/rewards-config.xml"})
```

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
public class RewardNetworkTest {
```

```
    @BeforeTransaction
```

Run before transaction is started

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

```
    @Before
```

Within the transaction

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

```
    @Test @Transactional
```

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

```
    ...
```

```
}
```



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

# Lab

Managing Transactions Declaratively  
using Spring Annotations

# Topics in this session

- Advanced topics
  - Programmatic transactions
  - Read-only transactions
  - Multiple transaction managers
  - Global transactions
  - Available Propagation Options

# Programmatic Transactions with Spring

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



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

# Programmatic Transactions: example

```
public RewardConfirmation rewardAccountFor(Dining dining) {  
    ...  
    return new TransactionTemplate(txManager).execute( (status) -> {  
        try {  
            ...  
            accountRepository.updateBeneficiaries(account);  
            confirmation = rewardRepository.confirmReward(contribution, dining);  
        }  
        catch (RewardException e) {  
            status.setRollbackOnly(); ←  
            confirmation = new RewardFailure();  
        }  
        return confirmation;  
    }  
};  
  
public interface TransactionCallback<T> {  
    public T doInTransaction(TransactionStatus status)  
        throws Exception;  
}
```

Method *not* @Transactional

Lambda syntax

Method no longer throws exception  
– *manual* rollback

# Read-only transactions (1)

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

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

2 connections

1 single connection

# Read-only transactions (2)

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

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

# Multiple Transaction Managers

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

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

Uses the bean with id  
"myOtherTransactionManager"  
"

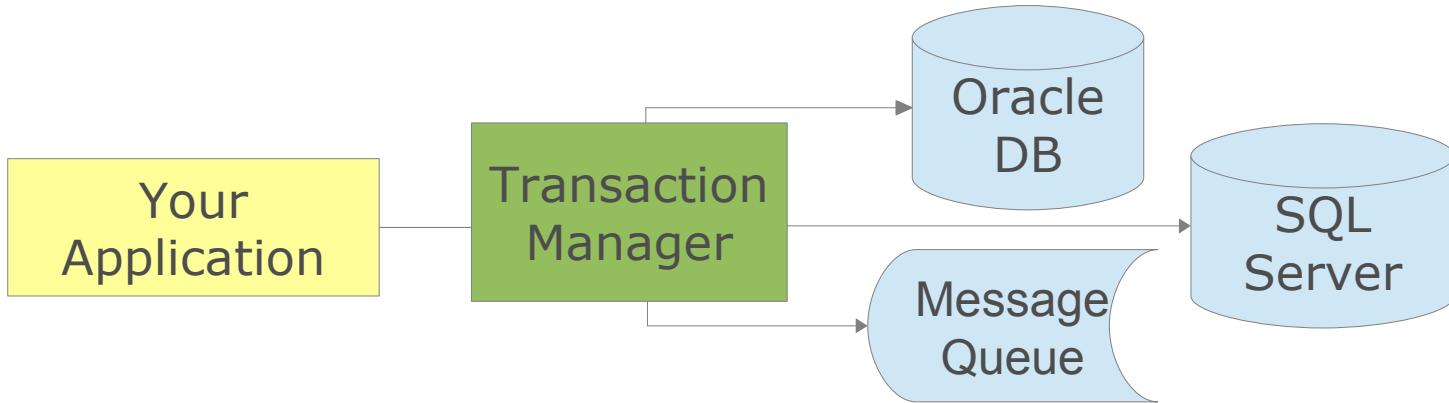
```
@Transactional
public void rewardAccount2() {
    jdbcTemplate.queryForList(...);
    jdbcTemplate.queryForInt(...);
}
```

Uses "transactionManager"  
bean by default

Important: Separate transaction managers = separate transactions!

# Global Transactions

- Also called distributed transactions
- Involve multiple dissimilar resources:



- Global transactions typically require JTA and specific drivers (XA drivers)

# Global transactions → Spring Integration

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

# Propagation Levels and their Behaviors

Propagation Type	If NO current transaction	If there is a current transaction
<b>MANDATORY</b>	throw exception	use current transaction
<b>NEVER</b>	don't create a transaction, run method outside any transaction	throw exception
<b>NOT_SUPPORTED</b>	don't create a transaction, run method outside any transaction	suspend current transaction, run method outside any transaction
<b>SUPPORTS</b>	don't create a transaction, run method outside any transaction	use current transaction
<b>REQUIRED(default)</b>	create a new transaction	use current transaction
<b>REQUIRES_NEW</b>	create a new transaction	suspend current transaction, create a new independent transaction
<b>NESTED</b>	create a new transaction	create a new nested transaction



# JPA with Spring and Spring Data

Object Relational Mapping with  
Spring & Java Persistence API

Using JPA with Spring, Spring Data Repositories

# Topics in this session

- Introduction to JPA
  - General Concepts
  - Mapping
  - Querying
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- Optional Topics

# Introduction to JPA

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

# About JPA

- Java Persistence API
  - Released May 2006
  - Version 2 since Dec 2009
    - Removes many of the limitations of JPA 1
    - Less need for ORM specific annotations and extensions
  - JPA 2.1 May 2013
    - Convertors, bulk updates, stored procedures, listeners
    - DDL generation, partial entity loading, extra query syntax
- Key Concepts
  - Entity Manager
  - Entity Manager Factory
  - Persistence Context

# JPA General Concepts (1)

- **EntityManager**

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

- **EntityManagerFactory**

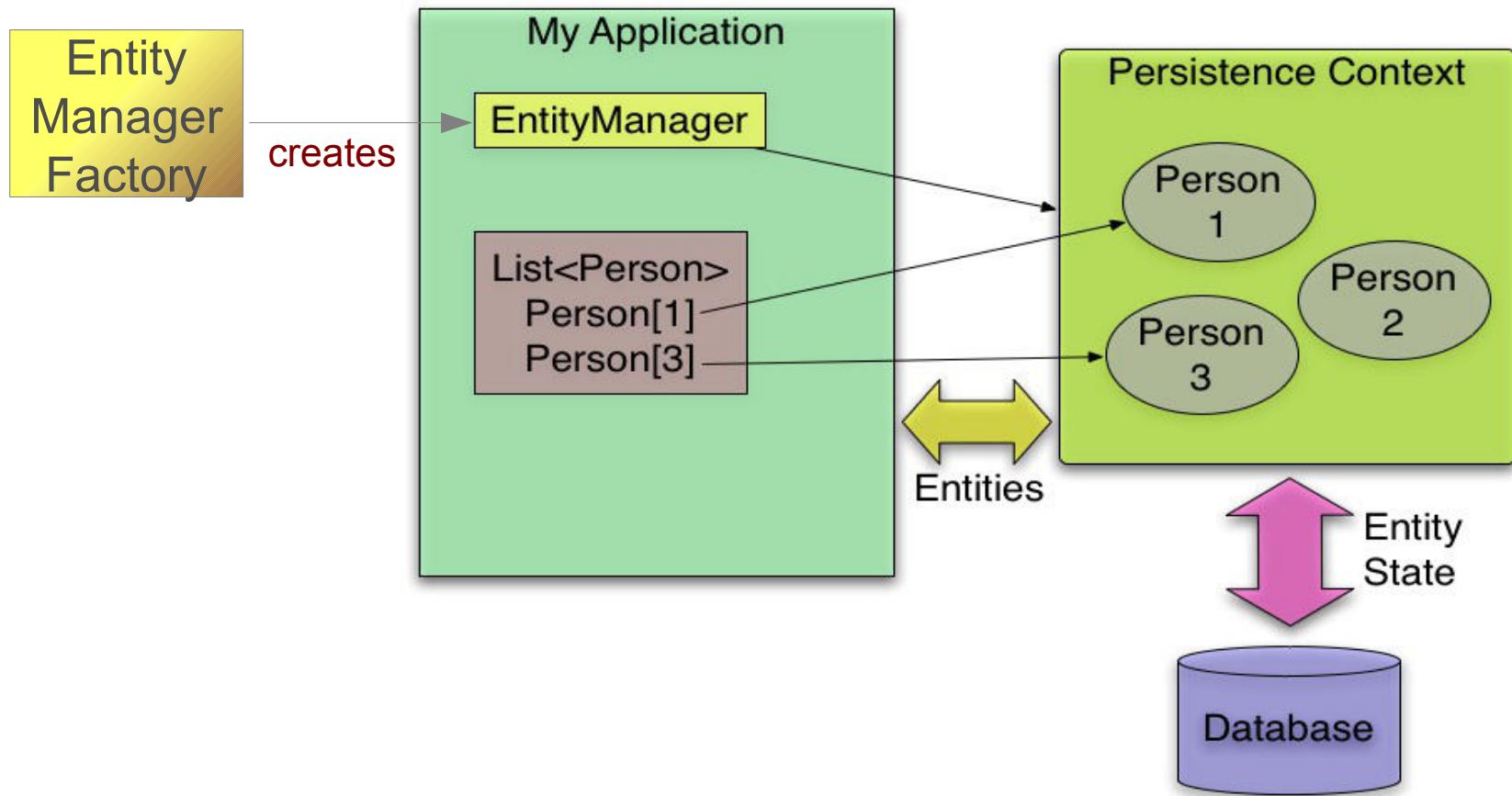
- thread-safe, shareable object that represents a single data source / persistence unit
- Provides access to new application-managed EntityManagers

# JPA General Concepts (2)

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



# Persistence Context and EntityManager



# The EntityManager API

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

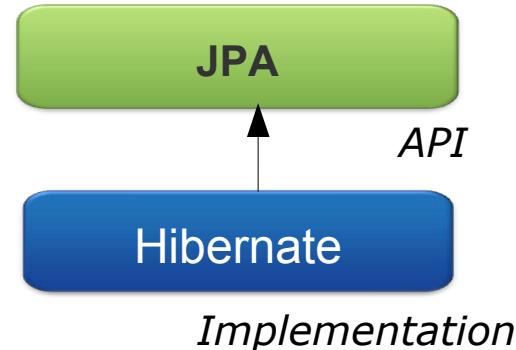
Plus other methods ...

# JPA Providers

- Several major implementations of JPA spec
  - Hibernate EntityManager
    - Used inside Jboss
  - EclipseLink (RI)
    - Used inside Glassfish
  - Apache OpenJPA
    - Used by Oracle WebLogic and IBM Websphere
  - Data Nucleus
    - Used by Google App Engine
- **Can all be used without application server as well**
  - Independent part of EJB 3 spec

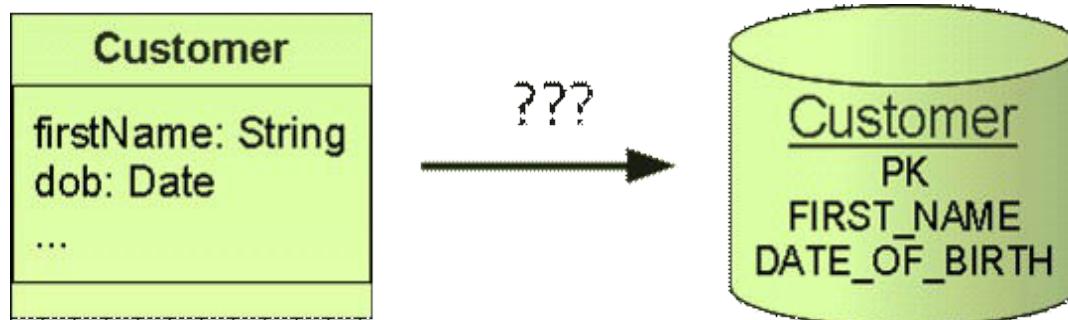
# Hibernate JPA

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



# JPA Mapping

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



# What can you Annotate?



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

# Mapping using fields (Data-Members)

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

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

Mark as an *entity*  
Optionally override  
table name

Mark *id-field*  
(primary key)

Optionally override  
column names

Not stored in database

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

# Mapping using accessors (Properties)

Must place @Id on the  
*getter* method

Other annotations now also  
placed on *getter* methods

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

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

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

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

# Relationships

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

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

```
@Entity  
@Table(name= "T_ADDRESS")  
public class Address {  
    @Id private Long id;  
    private String street;  
    private String suburb;  
    private String city;  
    private String postcode;  
    private String country;  
}
```

Foreign key in  
Address table

# Embeddables

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

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

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

Maps to ZIP  
column in  
`T_CUSTOMER`

# JPA Querying

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

# JPA Querying: By Primary Key

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

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

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

No cast required – JPA uses  
generics

# JPA Querying: JPQL

- SELECT clause required
- can't use \*

- Query for objects based on properties or associations ...

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

```
TypedQuery<Customer> query = entityManager.createQuery(  
    "select c from Customer c where c.address.city = :city", Customer.class);  
query.setParameter("city", "Chicago");  
List<Customer> customers = query.getResultList();
```

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

```
List<Customer> customers2 = entityManager.  
    createQuery("select c from Customer c ...", Customer.class).  
    setParameter("city", "Chicago").getResultList();
```

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

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

Specify Class to  
Populate / return

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

# Topics in this session

- Introduction to JPA
  - General Concepts
  - Mapping
  - Querying
- **Configuring JPA in Spring**
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- Optional Topics

# Quick Start – Spring JPA Configuration

## Steps to using JPA with Spring

1. Define an EntityManagerFactory bean.
2. Define a DataSource bean
3. Define a Transaction Manager bean
4. Define Mapping Metadata (already covered)
5. Define DAOs



Note: There are many configuration options for EntityManagerFactory, persistence.xml, and DataSource. See the optional section for details.

# Define the EntityManagerFactory

```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory(){  
  
    HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();  
    adapter.setShowSql(true);  
    adapter.setGenerateDdl(true);  
    adapter.setDatabase(Database.HSQL);  
  
    Properties props = new Properties();  
    props.setProperty("hibernate.format_sql", "true");  
  
    LocalContainerEntityManagerFactoryBean emfb =  
        new LocalContainerEntityManagerFactoryBean();  
    emfb.setDataSource(dataSource);  
    emfb.setPackagesToScan("rewards.internal");  
    emfb.setJpaProperties(props);  
    emfb.setJpaVendorAdapter(adapter);  
  
    return emfb;  
}
```

*NOTE: no persistence.xml  
needed when using  
packagesToScan property*

# Configuration – XML Equivalent

```
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="packagesToScan" value="rewards.internal"/>

  <property name="jpaVendorAdapter">
    <bean class="org.sfwk.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="showSql" value="true"/>
      <property name="generateDdl" value="true"/>
      <property name="database" value="HSQL"/>
    </bean>
  </property>

  <property name="jpaProperties">
    <props> <prop key="hibernate.format_sql">true</prop> </props>
  </property>
</bean>
```

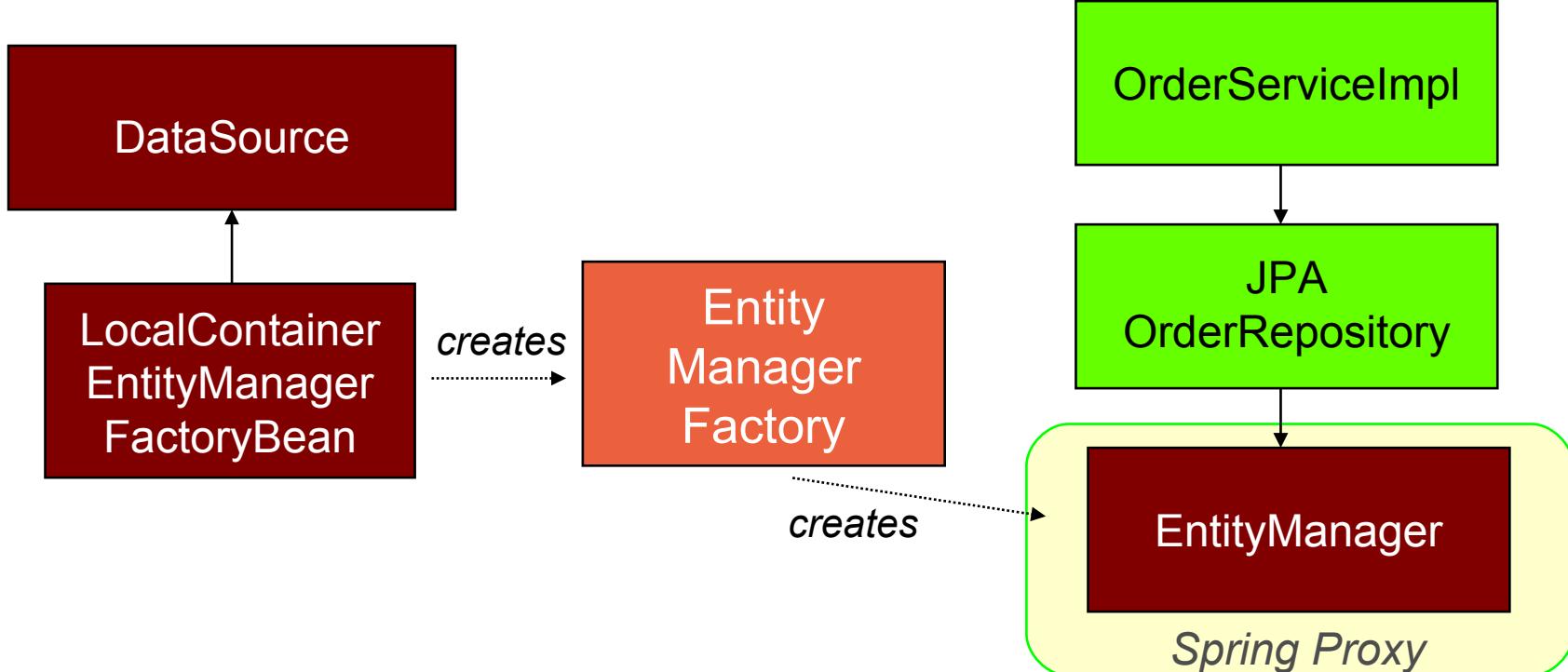
# Define DataSource & Transaction Manager

```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    LocalContainerEntityManagerFactoryBean emfb =  
        new LocalContainerEntityManagerFactoryBean();  
    emfb.setDataSource(dataSource());  
    ...  
    return emfb; }  
  
@Bean  
public PlatformTransactionManager  
    transactionManager(EntityManagerFactory emf) {  
    return new JpaTransactionManager(emf); }  
  
@Bean  
public DataSource dataSource() { // Lookup via JNDI or create locally. }
```

Method returns a *FactoryBean*...

...Spring calls `getObject()` on the FactoryBean to obtain the *EntityManagerFactory*:

# EntityManagerFactoryBean Configuration



Proxy automatically finds entity-manager for current transaction

# Topics in this session

- Introduction to JPA
  - General Concepts
  - Mapping
  - Querying
- Configuring JPA in Spring
- **Implementing JPA DAOs**
- Spring Data – JPA
- Lab
- Optional Topics

# Implementing JPA DAOs

- JPA provides configuration options so Spring can manage transactions and the EntityManager
- There are no Spring dependencies in your DAO implementations
- *Optional:* Use AOP for transparent exception translation
  - Rethrows JPA *PersistenceExceptions* as Spring's *DataAccessExceptions*
  - See Advanced Topics at end of section

# Spring-Managed Transactions & EntityManager (1)

- To transparently participate in Spring-driven transactions
  - Use a Spring FactoryBean for building the EntityManagerFactory
  - Inject an EntityManager reference with `@PersistenceContext`
- Define a transaction manager
  - JpaTransactionManager
  - JtaTransactionManager

# Spring-Managed Transactions & EntityManager (2)

- The code – no Spring dependencies

```
public class JpaOrderRepository implements OrderRepository {  
    private EntityManager entityManager;  
  
    @PersistenceContext  
    public void setEntityManager (EntityManager entityManager) {  
        this.entityManager = entityManager;  
    }  
  
    public Order findById(long orderId) {  
        return entityManager.find(Order.class, orderId);  
    }  
}
```

Automatic injection of EM Proxy

Proxy resolves to EM when used

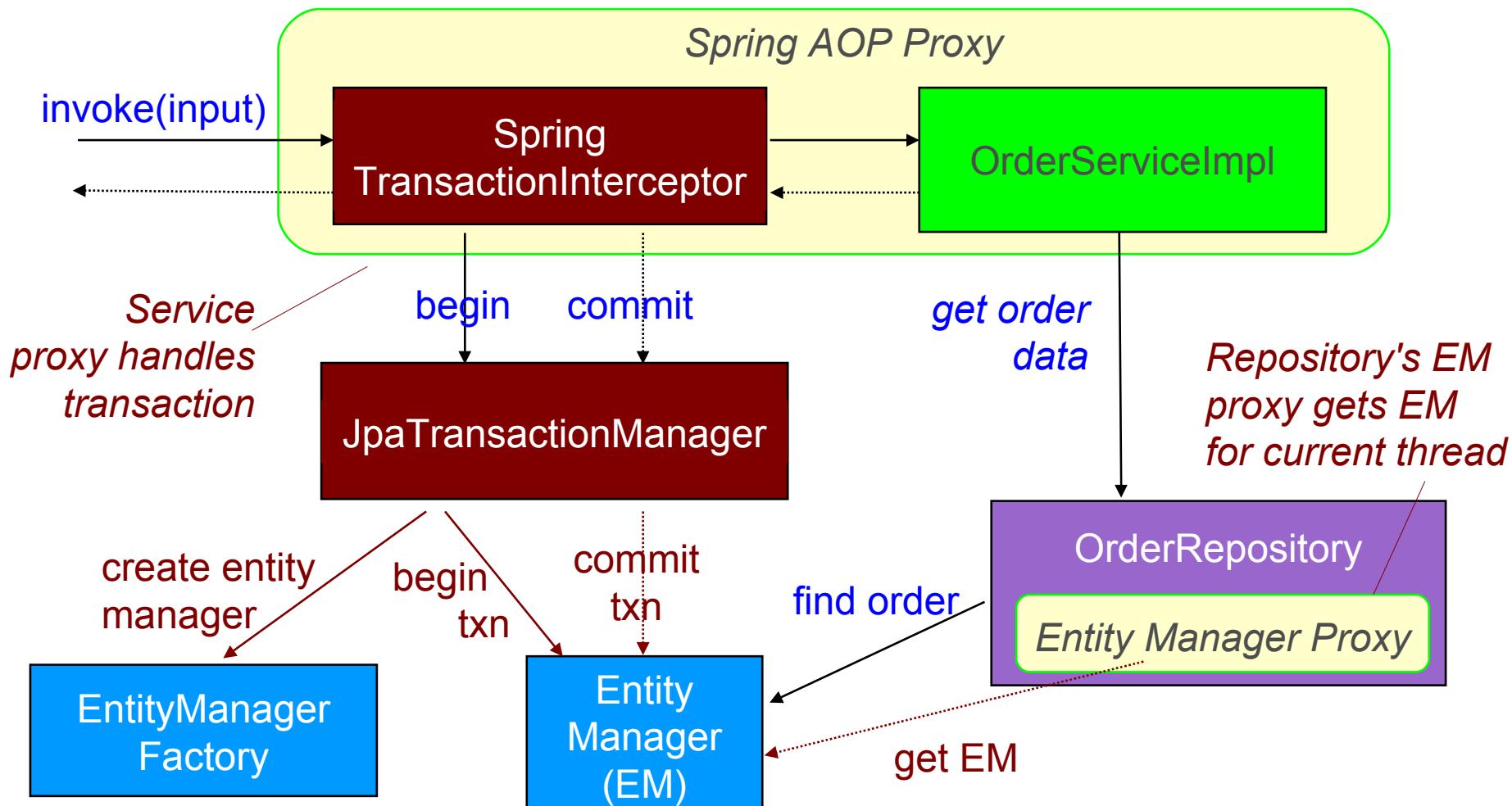
# Spring-managed Transactions and EntityManager (3)

- The Configuration

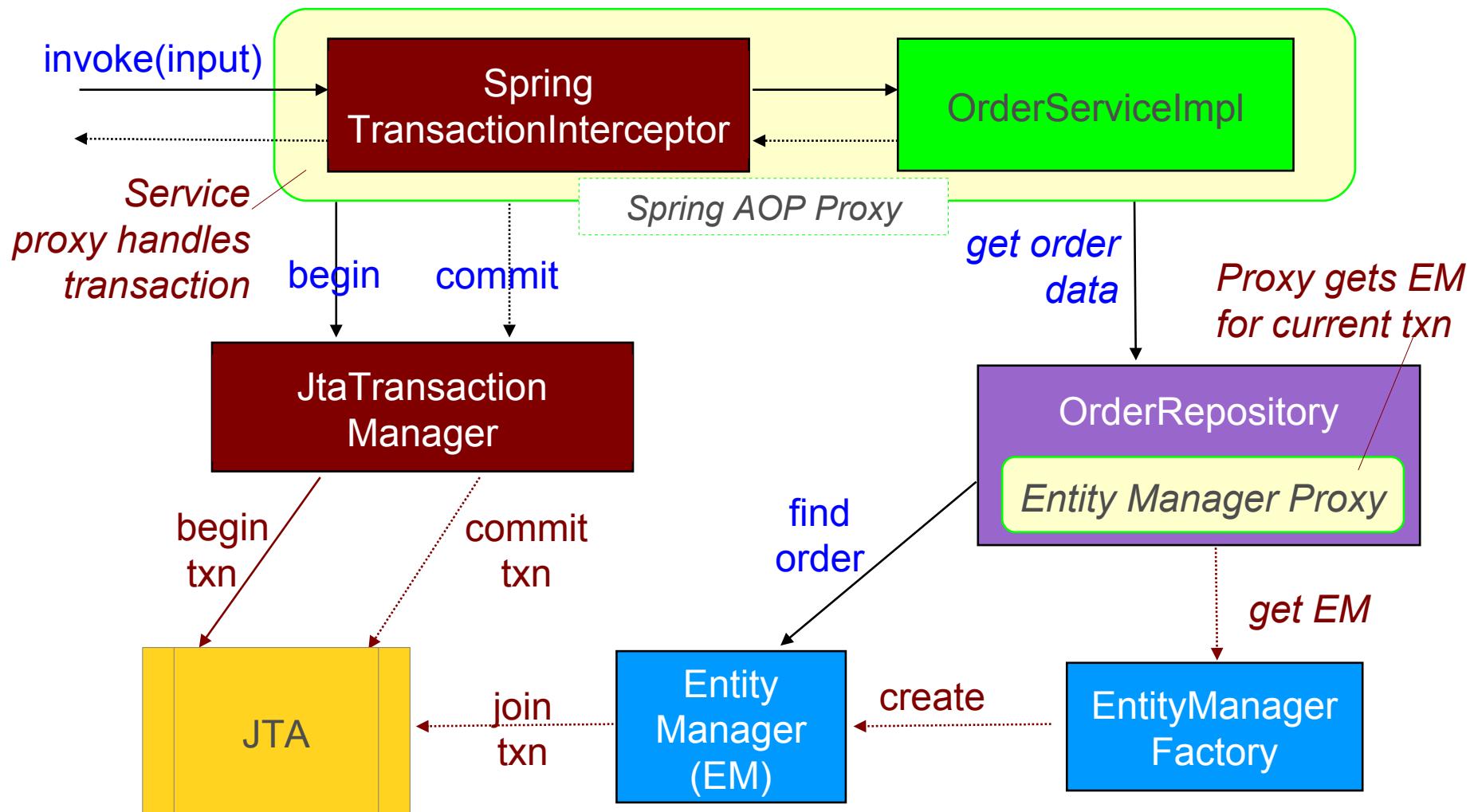
```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    ...  
}  
  
@Bean  
public OrderRepository jpaOrderRepository() {  
    return new JpaOrderRepository(); }  
  
@Bean  
public PlatformTransactionManager  
    transactionManager(EntityManagerFactory emf) throws Exception {  
    return new JpaTransactionManager(emf); }
```

Automatic injection of entity-manager proxy

# How it Works (JPA)



# How it Works (JTA)



# Topics in this session

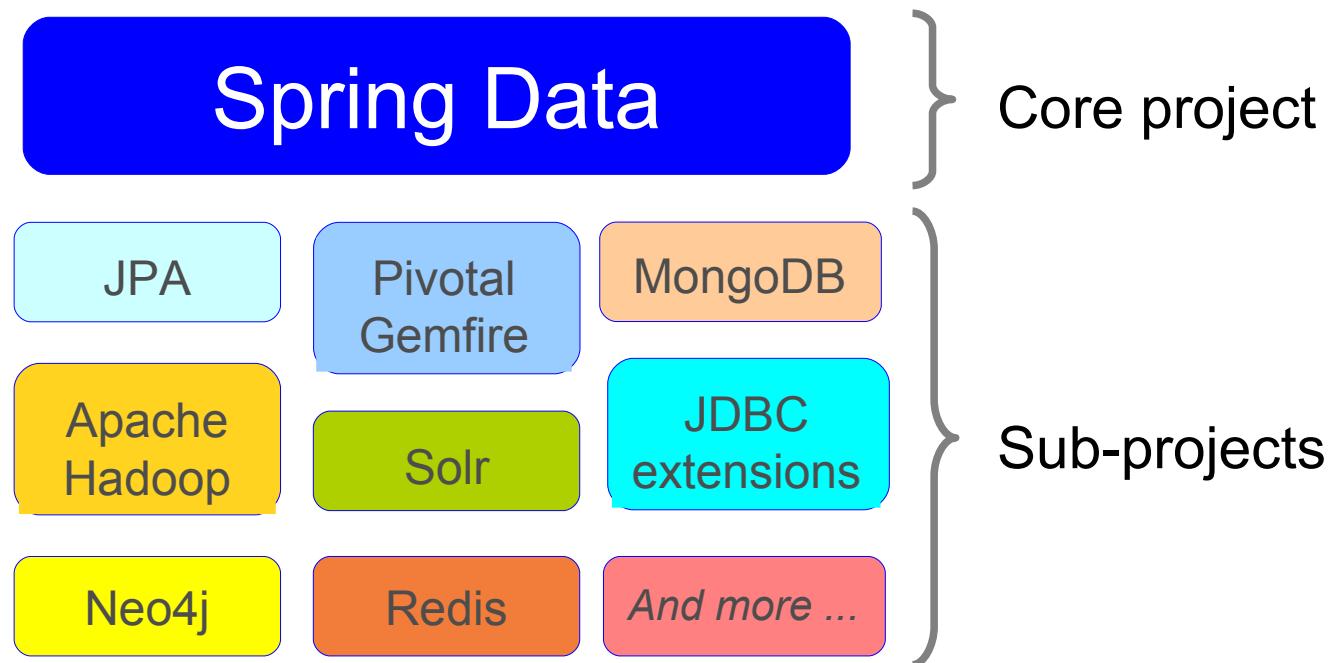
- Introduction to JPA
  - General Concepts
  - Mapping
  - Querying
- Configuring JPA in Spring
- Implementing JPA DAOs
- **Spring Data – JPA**
- Lab
- Optional Topics



# What is Spring Data?

SPRING DATA

- Reduces boiler plate code for data access
  - Works in many environments





# Instant Repositories

- How?
  - Annotate domain class
    - define keys & enable persistence
  - Define your repository as an *interface*
- Spring will implement it at run-time
  - Scans for interfaces extending Spring's `Repository<T, K>`
  - CRUD methods auto-generated
  - Paging, custom queries and sorting supported
  - Variations exist for most Spring Data sub-projects



# Domain Objects: Using JPA

- Annotate JPA Domain object as normal
  - Nothing to see here!

```
@Entity  
@Table(...)  
public class Customer {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id; ←  
    private Date firstOrderDate;  
    private String email;  
  
    // Other data-members and getters and setters omitted  
}
```

Domain Class

Key is a *Long*



# Domain Objects: Other Data Stores

- Spring Data provides similar annotations to JPA
  - `@Document`, `@Region`, `@NodeEntity` ...
- Templates (like `JdbcTemplate`) for basic CRUD access
  - `MongoTemplate`, `GemfireTemplate`, `RedisTemplate` ...

*MongoDB – map to a JSON document*

```
@Document  
public class Account {  
...}
```

`@NodeEntity`  
**public class** Account {

```
@GraphId  
Long id;  
...
```

*Neo4J – map to a graph*

*Gemfire – map to a region*

```
@Region  
public class Account {  
...}
```

# Extend Predefined Repository Interfaces

```
public interface Repository<Customer, Long> { }
```

Marker interface – add any methods from CrudRepository or add custom finders

```
public interface CrudRepository<T, ID  
    extends Serializable> extends Repository<T, ID> {
```

```
    public <S extends T> save(S entity);  
    public <S extends T> Iterable<S> save(Iterable<S> entities);
```

```
    public T findOne(ID id);  
    public Iterable<T> findAll();
```

You get all these methods automatically

```
    public void delete(ID id);  
    public void delete(T entity);  
    public void deleteAll();
```

PagingAndSortingRepository<T, K>  
- adds Iterable<T> findAll(Sort)  
- adds Page<T> findAll(Pageable)



# Generating Repositories

- Spring scans for Repository interfaces
  - Implements them and creates as a Spring bean
- XML

```
<jpa:repositories base-package="com.acme.**.repository" />
<mongo:repositories base-package="com.acme.**.repository" />
<gfe:repositories base-package="com.acme.**.repository" />
```

- Java Configuration

```
@Configuration
@EnableJpaRespositories(basePackages=com.acme.**.repository")
@EnableMongoRepositories(...)
public class MyConfig { ... }
```

# Defining a JPA Repository

- Auto-generated finders obey naming convention
  - findBy<DataMember><Op>
  - <Op> can be Gt, Lt, Ne, Between, Like ... etc

```
public interface CustomerRepository  
    extends CrudRepository<Customer, Long> {  
  
    public Customer findByEmail(String someEmail); // No <Op> for Equals  
    public Customer findByFirstOrderDateGt(Date someDate);  
    public Customer findByFirstOrderDateBetween(Date d1, Date d2);  
  
    @Query("SELECT c FROM Customer c WHERE c.email NOT LIKE '%@%'")  
    public List<Customer> findInvalidEmails();  
}
```

**id**

Custom query uses query-language of underlying product (here JPQL)

# Convention over Configuration

Extend **Repository** and build your own interface using conventions.

- Note: Repository is an *interface* (*not a class!*)

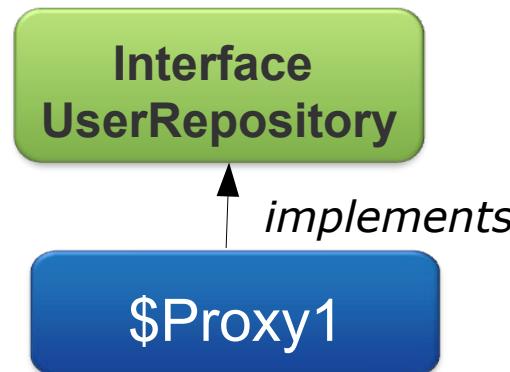
```
import org.springframework.data.repository.Repository;  
import org.springframework.data.jpa.repository.Query;  
  
public interface CustomerRepository extends Repository<Customer, Long> {  
  
    <S extends Customer> save(S entity); // Definition as per CrudRepository  
  
    Customer findById(long i); // Query determined from method name  
  
    Customer findByEmailIgnoreCase(String email); // Case insensitive search  
  
    @Query("select u from Customer u where u.emailAddress = ?1")  
    Customer findByEmail(String email); // ?1 replaced by method param  
}
```

# Internal behavior

- Before startup



- After startup



```
<jpa:repositories base-package="com.acme.repository"/>
```



You can conveniently use Spring to inject a dependency of type UserRepository. Implementation will be generated at startup time.

# Adding Custom Behavior (1)

- Not all use cases satisfied by automated methods
  - Enrich with custom repositories
- Step 1: Create normal interface and implementation
  - Instantiate as a normal Spring bean:

```
public interface AccountRepositoryCustom {  
    Account findOverdrawnAccounts();  
}
```



```
@Repository // This is a normal Spring bean, id = "accountRepositoryImpl"  
public class AccountRepositoryImpl implements AccountRepositoryCustom {  
    Account findOverdrawnAccounts() {  
        // Your custom implementation  
    }  
}
```

# Adding Custom Behavior (2)

- Step 2: Combine with automatic repository:

```
public interface AccountRepository  
    extends CrudRepository<Account, Long>, AccountRepositoryCustom {  
}
```

- Spring Data looks for implementation beans
  - ID = repository interface + “Impl” (configurable)
  - In this example: “*AccountRepositoryImpl*”
- Result: *AccountRepository* bean contains automatic and custom methods!

# Lab

Reimplementing Repositories using  
Spring and JPA

# Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional Topics**
  - JPA Typed Queries / Native Queries
  - EntityManagerFactoryBean alternatives / persistence.xml
  - Exception Translation

# JPA Querying: Typed Queries

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

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

Meta-data class  
created by JPA  
(note underscore)

# JPA Querying: SQL

- Use a *native* query to execute raw SQL

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

No *named* parameter support

```
// ... or if expecting a single result
String customerNumber = (String) query.getSingleResult();
```

Indexed from 1  
– like JDBC

```
// Query for multiple columns
Query query = entityManager.createNativeQuery(
    "SELECT ... FROM T_CUSTOMER c WHERE ...", Customer.class);
List<Customer> customers = query.getResultList();
```

Specify Class to  
Populate / return

# Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional Topics**
  - JPA Typed Queries / Native Queries
  - **EntityManagerFactoryBean alternatives / persistence.xml**
  - Exception Translation

# Setting up an EntityManagerFactory

- Three ways to set up an EntityManagerFactory:
  - LocalEntityManagerFactoryBean
  - LocalContainerEntityManagerFactoryBean
  - Use a JNDI lookup
- **persistence.xml** required for configuration
  - From version 3.1, Spring allows no *persistence.xml* with LocalContainerEntityManagerFactoryBean

# persistence.xml

<?xml?>

- Always stored in META-INF
- Specifies “persistence unit”:
  - optional vendor-dependent information
  - DB Connection properties often specified here.

```
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="rewardNetwork">
        ...
    </persistence-unit>
</persistence>
```

- File is required in JPA, but optional when using Spring with JPA!

# LocalEntityManagerFactoryBean

- Useful for standalone apps, integration tests
- Cannot specify a DataSource
  - Useful when only data access is via JPA
  - Uses standard JPA service location (SPI) mechanism  
`/META-INF/services/javax.persistence.spi.PersistenceProvider`

```
@Bean
public LocalEntityManagerFactoryBean entityManager() {
    LocalEntityManagerFactoryBean em =
        new LocalEntityManagerFactoryBean();
    em.setPersistenceUnitName("rewardNetwork");
    return em;
}
```

# LocalContainer EntityManagerFactoryBean

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

We saw this earlier with  
100% Spring configuration



# Configuration – Spring and Persistence Unit

```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    LocalContainerEntityManagerFactoryBean emfb =  
        new LocalContainerEntityManagerFactoryBean();  
    emfb.setDataSource(dataSource);  
    emfb.setPersistenceUnitName("rewardNetwork");  
    return emfb;  
}  
  
<persistence-unit name="rewardNetwork">  
    <provider>org.hibernate.ejb.HibernatePersistence</provider>  
    <properties>  
        <property name="hibernate.dialect"  
            value="org.hibernate.dialect.HSQLDialect"/>  
        <property name="hibernate.hbm2ddl.auto" value="create"/>  
        <property name="hibernate.show_sql" value="true" />  
        <property name="hibernate.format_sql" value="true" />  
    </properties>  
</persistence-unit>
```

*If using JTA – declare <jta-data-source> in the persistence-unit*

# JNDI Lookups

- A jee:jndi-lookup can be used to retrieve *EntityManagerFactory* from application server
- Useful when deploying to JEE Application Servers (WebSphere, WebLogic, etc.)

```
@Bean  
public EntityManagerFactory entityManagerFactory() throws Exception {  
    Context ctx = new InitialContext();  
    return (DataSource) ctx.lookup("persistence/rewardNetwork");  
}
```

OR

```
<jee:jndi-lookup id="entityManagerFactory"  
    jndi-name="persistence/rewardNetwork"/>
```

# Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional Topics**
  - JPA Typed Queries / Native Queries
  - EntityManagerFactoryBean alternatives / persistence.xml
  - **Exception Translation**

# Transparent Exception Translation (1)

- Used as-is, the DAO implementations described earlier will throw unchecked PersistenceExceptions
  - Not desirable to let these propagate up to the service layer or other users of the DAOs
  - Introduces dependency on the specific persistence solution that should not exist
- AOP allows translation to Spring's rich, vendor-neutral DataAccessException hierarchy
  - Hides access technology used

# Transparent Exception Translation (2)

- Spring provides this capability out of the box
  - Annotate with `@Repository`
  - Define a Spring-provided BeanPostProcessor

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

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

# Transparent Exception Translation (3)

- Or use XML configuration:

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

No annotations

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

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

# Summary

- Use 100% JPA to define entities and access data
  - Repositories have no Spring dependency
  - Spring Data Repositories need no code!
- Use Spring to configure JPA entity-manager factory
  - Smart proxy works with Spring-driven transactions
  - Optional translation to DataAccessExceptions

*Spring-JPA – 3 day in-depth JPA course with  
emphasis on Hibernate as JPA provider*

# Overview of Spring Web

## Developing Modern Web Applications

Servlet Configuration, Product Overview

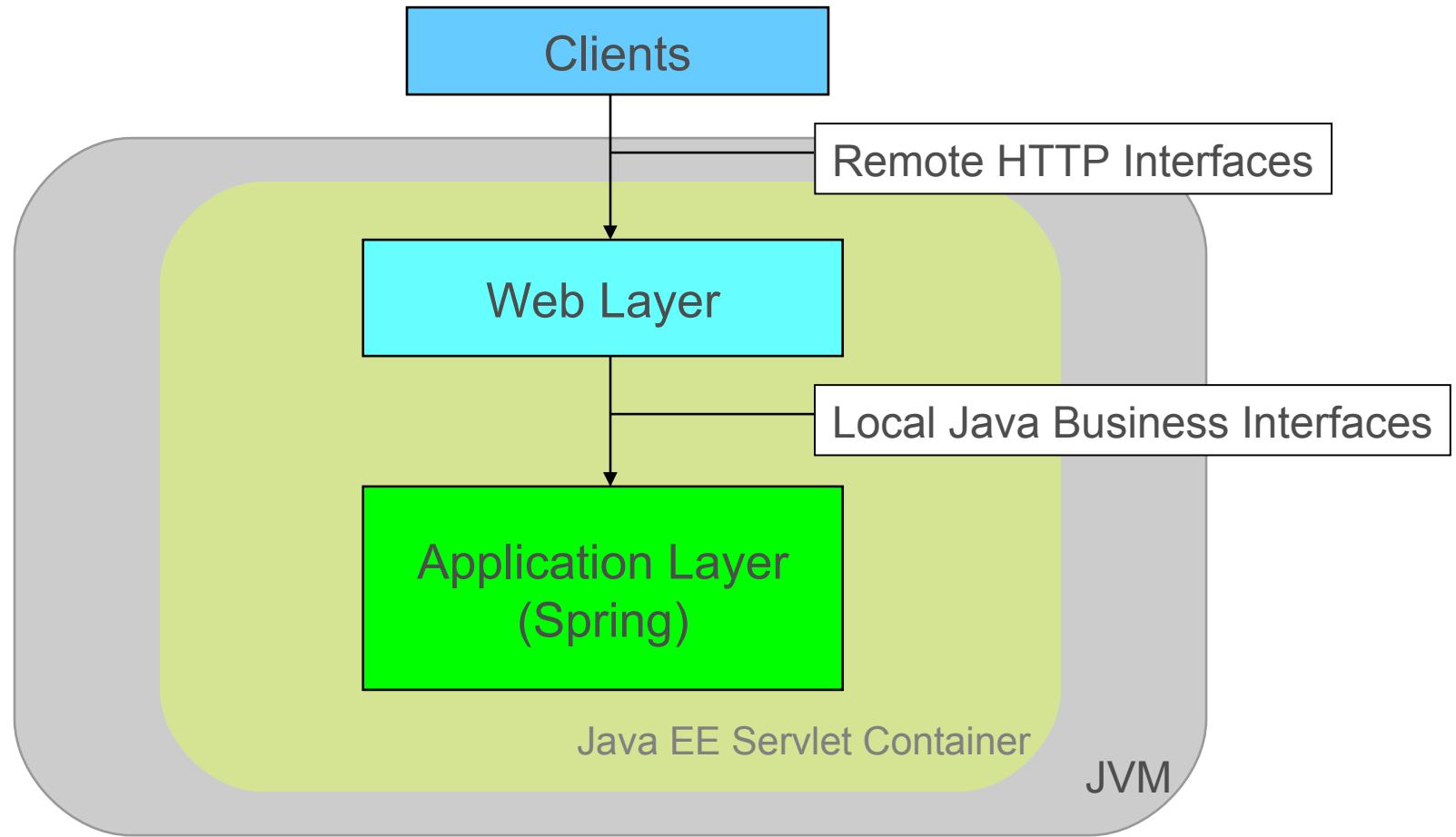
# Topics in this Session

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

# Web layer integration

- Spring provides support in the Web layer
  - Spring MVC, Spring WebFlow...
- However, you are free to use Spring with any Java web framework
  - Integration might be provided by Spring or by the other framework itself

# Effective Web Application Architecture



# Topics in this Session

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

# Spring Application Context Lifecycle in Web Applications

- Spring can be initialized within a webapp
  - start up business services, repositories, etc.
- Uses a standard servlet listener
  - initialization occurs before any servlets execute
  - application ready for user requests
  - `ApplicationContext.close()` is called when the application is stopped

# Configuration via WebApplicationInitializer

```
public class MyWebAppInitializer  
    extends AbstractContextLoaderInitializer {  
  
    @Override  
    protected WebApplicationContext createRootApplicationContext() {  
  
        // Create the 'root' Spring application context  
        AnnotationConfigWebApplicationContext rootContext =  
            new AnnotationConfigWebApplicationContext();  
  
        rootContext.getEnvironment().setActiveProfiles("jpa"); // optional  
        rootContext.register(RootConfig.class);  
        return rootContext;  
    }  
    ...
```

Implements  
*WebApplicationInitializer*  
Automatically detected  
by servlet container.

Available in Servlet 3.0+ Environments, no more web.xml!

# Configuration in web.xml

- Only option prior to servlet 3.0
  - Just add a Spring-provided servlet listener

```
<context-param>                                              web.xml
    <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*

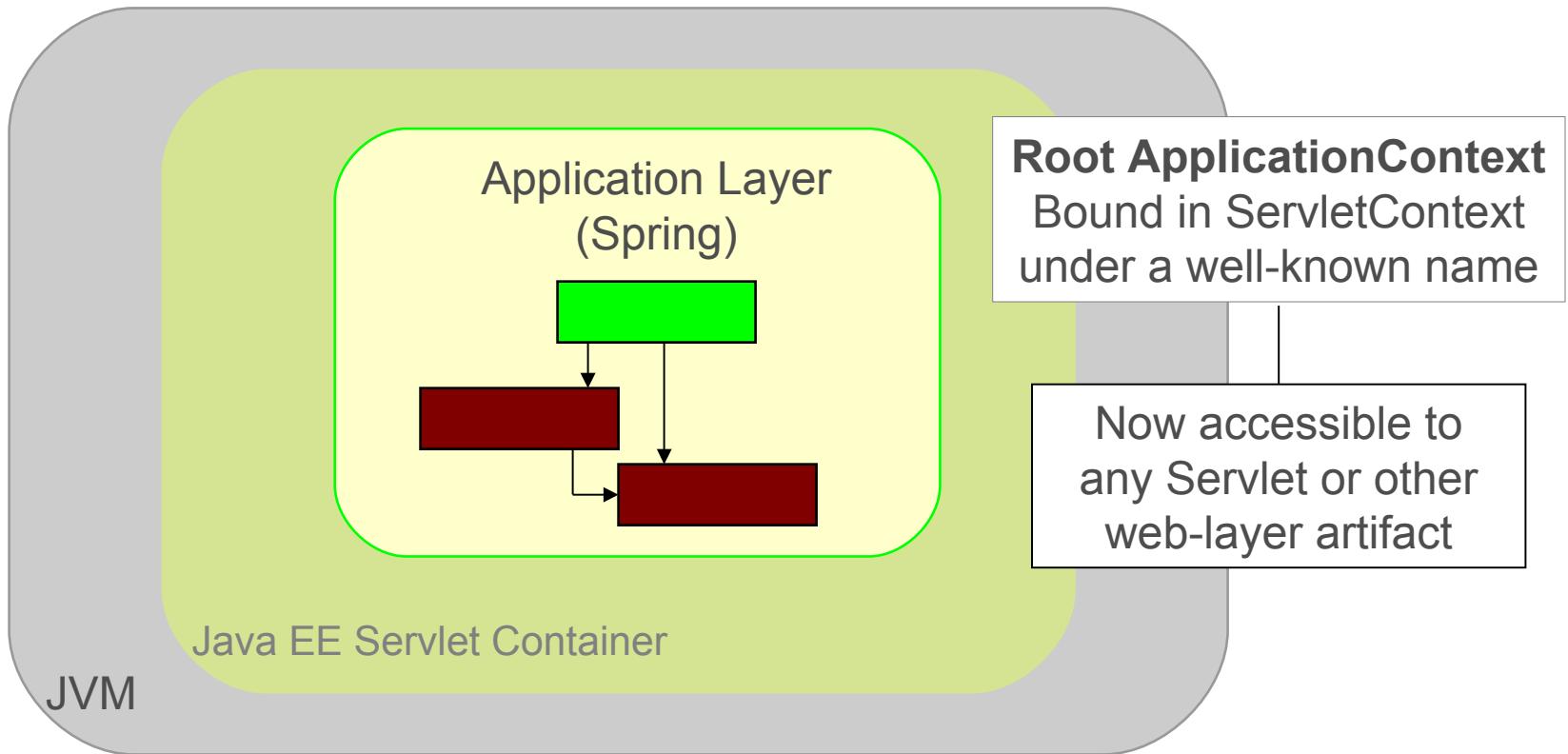
# Configuration Options

- Default resource location is document-root
  - Can use *classpath*: designator
  - Defaults to WEB-INF/applicationContext.xml
- Can optionally select profile to use

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:/rewards/internal/application-config.xml
        /WEB-INF/merchant-reporting-webapp-config.xml
    </param-value>
</context-param>
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>web</param-value>
</context-param>
```

web.xml

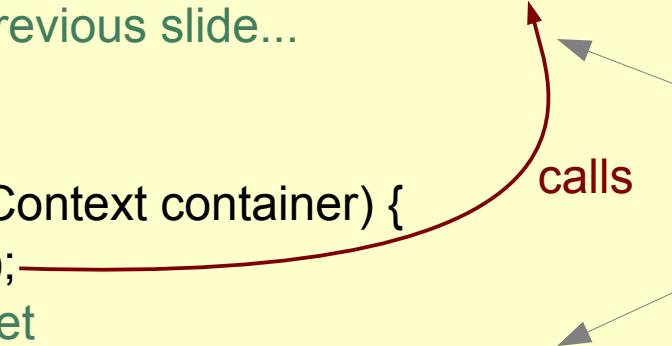
# Servlet Container After Starting Up



# Instantiating Servlets

- Override `onStartup()` method to define servlets
  - **Warning:** Can't inject the beans, not available yet

```
public class MyWebAppInitializer extends AbstractContextLoaderInitializer {  
  
    protected WebApplicationContext createRootApplicationContext() {  
        // ...Same configuration as previous slide...  
    }  
  
    public void onStartup(ServletContext container) {  
        super.onStartup(container);  
        // Register and map a servlet  
        ServletRegistration.Dynamic svlt =  
            container.addServlet("myServlet", new TopSpendersReportGenerator());  
        svlt.setLoadOnStartup(1);  
        svlt.addMapping("/");  
    }  
}
```



No beans are loaded yet at this point in the lifecycle...

# Dependency Injection of Servlets

- Suitable for *web.xml* or *AbstractContextLoaderInitializer*
  - Neither provides access to Spring Beans
- Use *WebApplicationContextUtils*
  - provides Spring *ApplicationContext* via *ServletContext*

```
public class TopSpendersReportGenerator extends HttpServlet {  
    private ClientService clientService;  
  
    public void init() {  
        ApplicationContext context = WebApplicationContextUtils.  
            getRequiredWebApplicationContext(getServletContext());  
        clientService = (ClientService) context.getBean("clientService");  
    }  
    ...  
}
```

# Dependency injection

- Example using Spring MVC

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

Dependency is automatically injected by type



No need for *WebApplicationContextUtils* anymore

# Topics in this Session

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

# Spring Web

- Spring MVC
  - Web framework bundled with Spring
- Spring WebFlow
  - Plugs into Spring MVC
  - Implements navigation flows
- Spring Mobile
  - Routing between mobile / non-mobile versions of site
- Spring Social
  - Easy integration with Facebook, Twitter, etc.

# Spring Web MVC

- Spring's web framework
  - Uses Spring for its own configuration
  - Controllers are Spring beans
  - testable artifacts
- Annotation-based model since Spring 2.5
- Builds on the Java Servlet API
- The core platform for developing web applications with Spring
  - All higher-level modules such as WebFlow build on it

# Spring Web Flow

- Plugs into Spring Web MVC as a Controller technology for implementing stateful "flows"
  - Checks that users follow the right navigation path
  - Manages back button and multiple windows issues
  - Provides scopes beyond request and session
    - such as the *flow* and *flash* scope
  - Addresses the double-submit problem elegantly

# Example Flow Definition

## Online Check-in

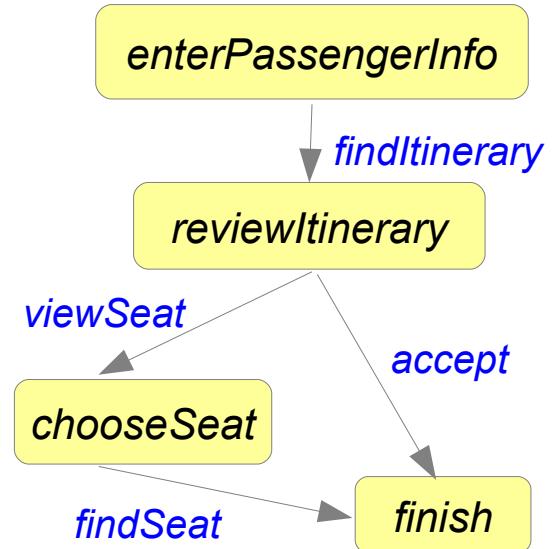
- Flows are declared in XML

```
<flow ...>
    <view-state id="enterPassengerInfo">
        <transition on="findItinerary" to="reviewItinerary" />
    </view-state>

    <view-state id="reviewItinerary">
        <transition on="viewSeat" to="chooseSeat" />
        <transition on="accept" to="finish" />
    </view-state>

    <view-state id="chooseSeat">
        <transition on="findSeat" to="finish" />
    </view-state>

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



# More about WebFlow

- Online sample application is available here:  
<http://richweb.springframework.org/swf-booking-faces/spring/intro>
- Sample applications can be downloaded here:  
<http://www.springsource.org/webflow-samples>



The screenshot shows a Mozilla Firefox browser window displaying the "Spring Faces: Hotel Booking Sample Application". The title bar reads "Spring Faces: Hotel Booking Sample Application - Mozilla Firefox". The menu bar includes File, Edit, View, History, Delicious, Bookmarks, Yahoo!, Tools, and Help. The main content area has a header "Hotel Results" and a "Change Search" link. Below is a table listing five hotels:

Name	Address	City, State	Zip	Action
Westin Diplomat	3555 S. Ocean Drive	Hollywood, FL, USA	33019	<a href="#">View Hotel</a>
Jameson Inn	890 Palm Bay Rd NE	Palm Bay, FL, USA	32905	<a href="#">View Hotel</a>
Chilworth Manor	The Cottage, Southampton Business Park	Southampton, Hants, UK	SO16 7JF	<a href="#">View Hotel</a>
Marriott Courtyard	Tower Place, Buckhead	Atlanta, GA, USA	30305	<a href="#">View Hotel</a>
Doubletree	Tower Place, Buckhead	Atlanta, GA, USA	30305	<a href="#">View Hotel</a>

A large image of a modern hotel building is displayed on the left side of the page. At the bottom left, there is a banner with the text "THE SPRING EXPERIENCE". At the bottom right, there is a link "More Results".

# Topics in this Session

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

# Spring – Struts 1 integration

- Integration provided by the Spring framework
  - Inherit from ActionSupport instead of Action

```
public class UserAction extends ActionSupport {  
  
    public ActionForward execute(ActionMapping mapping,  
                                ActionForm form,...) throws Exception {  
        WebApplicationContext ctx = getWebApplicationContext();  
        UserManager mgr = (UserManager) ctx.getBean("userManager");  
        return mapping.findForward("success");  
    }  
}
```

Provided by the  
Spring framework



This is one of the 2 ways to integrate Spring and Struts 1 together.  
More information in the [reference documentation](#)

# Spring – JSF integration

- Two options
  - Spring-centric integration
    - Provided by Spring Faces
  - JSF-centric integration
    - Spring plugs in as a JSF managed bean provider

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>com.springsource.web.ClientController</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

*JSF-centric integration*

# Integration provided by other frameworks

- Struts 2
  - provides a Spring plugin  
<http://struts.apache.org/2.0.8/docs/spring-plugin.html>
- Wicket
  - Comes with an integration to Spring  
<http://cwiki.apache.org/WICKET/spring.html>
- Tapestry 5
  - Comes with an integration to Spring  
<http://tapestry.apache.org/tapestry5/tapestry-spring/>

# Summary

- Spring can be used with any web framework
  - Spring provides the ContextLoaderListener that can be declared in web.xml
- Spring MVC is a lightweight web framework where controllers are Spring beans
  - More about Spring MVC in the next module
- WebFlow plugs into Spring MVC as a Controller technology for implementing stateful "flows"

# Spring Web MVC Essentials

## Getting Started With Spring MVC

### Implementing a Simple Controller

# What is Spring MVC?

- Web framework based on the Model/View/Controller pattern
  - Alternative to Struts 1, Struts 2 (WebWork), Tapestry, Wicket, JSF, etc.
- Based on Spring principles
  - POJO programming
  - Testable components
  - Uses Spring for configuration
- Supports a wide range of view technologies
  - JSP, XSLT, PDF, Excel, Velocity, Freemarker, Thymeleaf, etc.

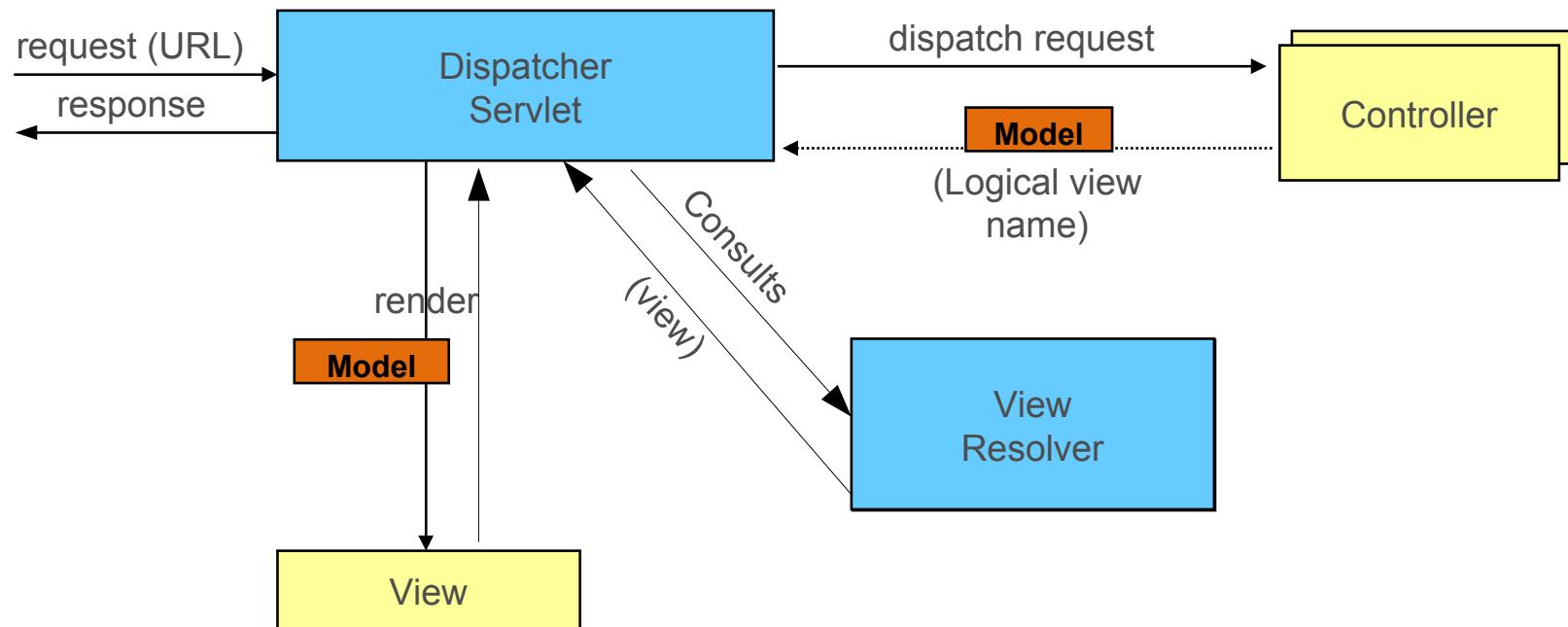
# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - Controllers
  - Views
- Quick Start

# Web Request Handling Overview

- Web request handling is rather simple
  - Based on an incoming URL...
  - ...we need to call a method...
  - ...after which the return value (if any)...
  - ...needs to be rendered using a view

# Request Processing Lifecycle



# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - Controllers
  - 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 or WebApplicationInitializer
- 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

# Dispatcher Servlet Configuration Example

```
public class WebInitializer  
    extends AbstractAnnotationConfigDispatcherServletInitializer {  
  
    // Tell Spring what to use for the Root context:  
    @Override protected Class<?>[] getRootConfigClasses() {  
        return new Class<?>[]{ RootConfig.class };  
    }  
  
    // Tell Spring what to use for the DispatcherServlet context:  
    @Override protected Class<?>[] getServletConfigClasses() {  
        return new Class<?>[]{ MvcConfig.class };  
    }  
  
    // DispatcherServlet mapping:  
    @Override protected String[] getServletMappings() {  
        return new String[]{"/*main/*"};  
    }  
}
```

Servlet 3.0+

“Root” configuration

MVC configuration

Beans defined in  
dispatcherContext  
have access to  
beans defined in  
rootContext.

# Dispatcher Servlet Configuration Example

```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/web-config.xml</param-value>
    </init-param>
</servlet>

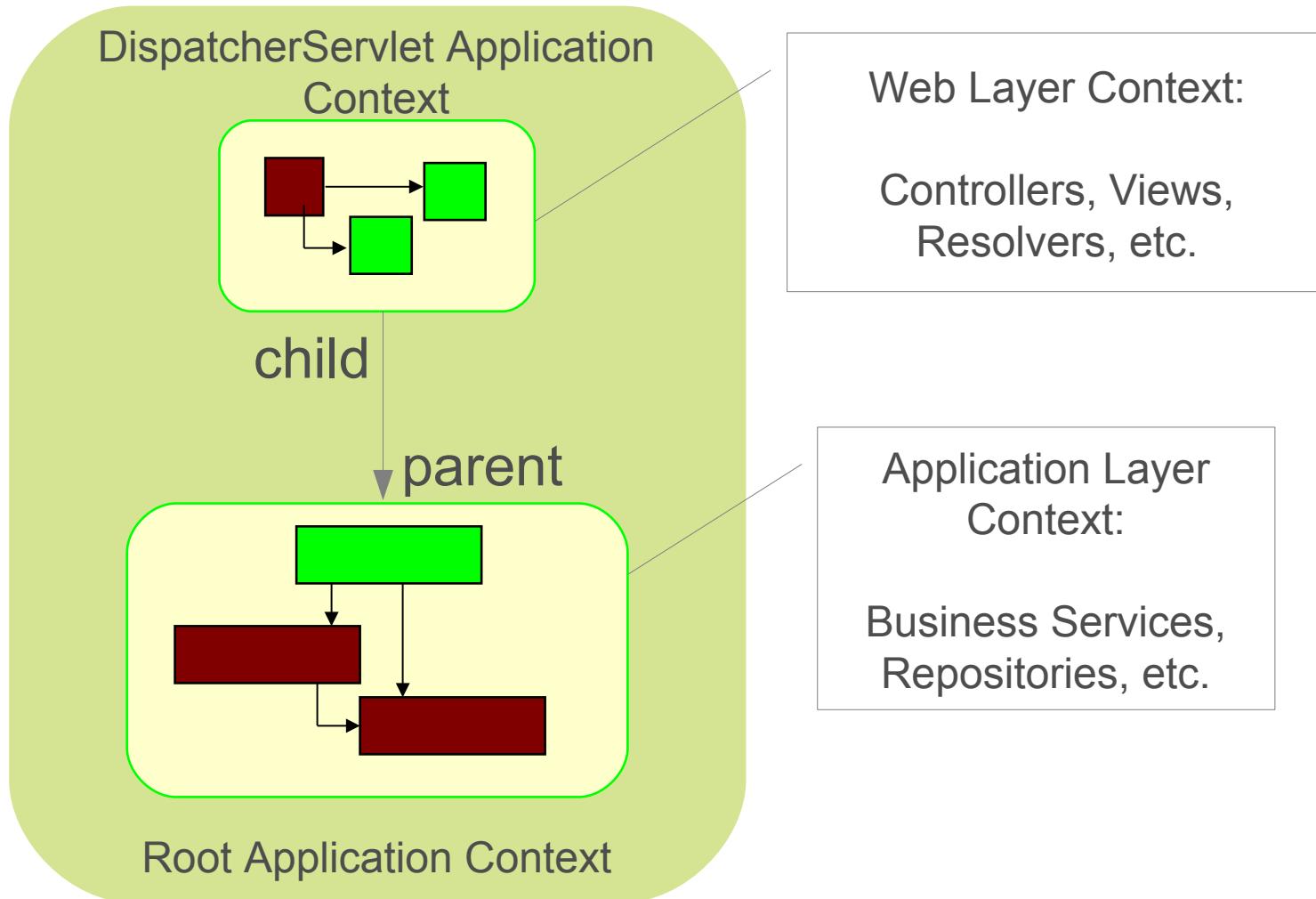
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/main/*</url-pattern>
</servlet-mapping>
```

web.xml

Pre-Servlet 3.0

Beans defined in web layer have access  
to beans defined in RootApplicationContext.

# Servlet Container After Starting Up



# Topics in this Session

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

# Controller Implementation

- Annotate controllers with @Controller
- @RequestMapping tells Spring what method to execute when processing a particular request

```
@Controller  
public class AccountController {
```

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

```
}
```

*Example of calling URL:*

[http://localhost:8080 / mvc-1 / rewardsadmin / listAccounts](http://localhost:8080/mvc-1/rewardsadmin/listAccounts)

application server

webapp

servlet mapping

request mapping

# URL-Based Mapping Rules

- Mapping rules typically URL-based, optionally using wild cards:
  - /login
  - /editAccount
  - /listAccounts.htm
  - /reward/\*/\*\*

# Controller Method Parameters

- Extremely flexible!
- You pick the parameters you need, Spring provides them
  - HttpServletRequest, HttpSession, Principal ...
  - Model for sending data to the view.
  - See [Spring Reference, Handler Methods](#)

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

View name

Model holds data for view

# Extracting Request Parameters

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

```
@Controller  
public class AccountController {  
  
    @RequestMapping("/showAccount")  
    public String show(@RequestParam("entityId") long id,  
                      Model model) {  
  
        ...  
    }  
}
```

*Example of calling URL:*

`http://localhost:8080/mvc-1/rewardsadmin/showAccount.htm?entityId=123`

# URI Templates

- Values can be extracted from request URLs
  - Based on *URI Templates*
  - not Spring-specific concept, used in many frameworks
  - Use {...} placeholders and @PathVariable
- Allows clean URLs without request parameters

```
@Controller  
public class AccountController {  
  
    @RequestMapping("/accounts/{accountId}")  
    public String show(@PathVariable("accountId") long id,  
                      Model model) {  
  
        ...  
    }  
    ...  
}
```

*Example of calling URL:*

<http://localhost:8080/mvc-1/rewardsadmin/accounts/123>

# Method Signature Examples

```
@RequestMapping("/accounts")
public String show(HttpServletRequest request,
                    Model model)
```

```
@RequestMapping("/orders/{id}/items/{itemId}")
public String show(@PathVariable("id") Long id,
                   @PathVariable int itemId,
                   Model model, Locale locale,
                   @RequestHeader("user-agent") String agent )
```

```
@RequestMapping("/orders")
public String show(@RequestParam Long id,
                   @RequestParam("itemId") int itemId,
                   Principal user, Map<String, Object> model,
                   Session session )
```

# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - Controllers
  - Views
- Quick Start

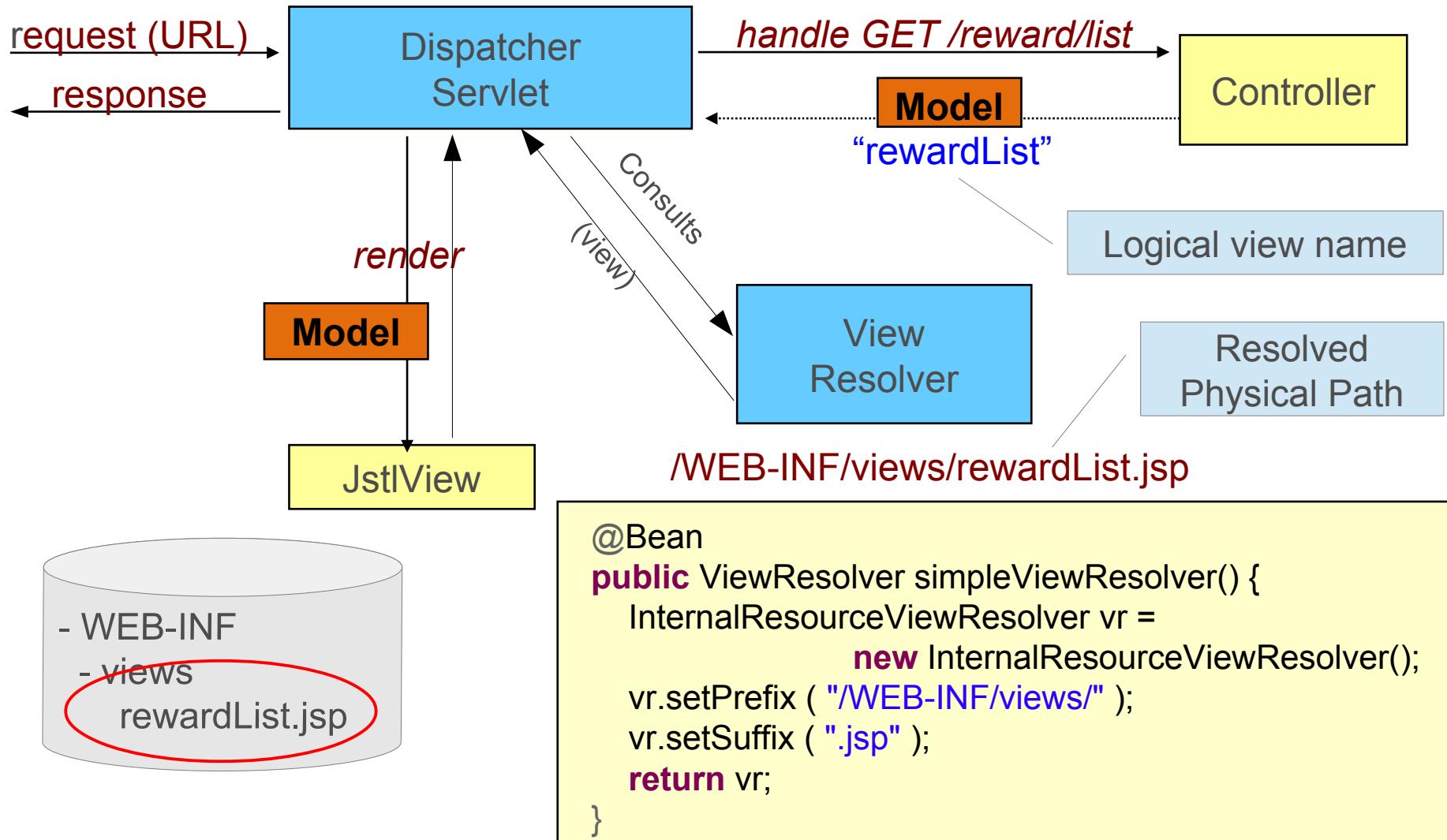
# Views

- A **View** renders web output.
  - Many built-in views available for JSPs, XSLT, templating approaches (Velocity, FreeMarker), etc.
  - View support classes for creating PDFs, Excel spreadsheets, etc.
- Controllers typically return a 'logical view name' String.
- **ViewResolvers** select View based on view name.

# View Resolvers

- The DispatcherServlet delegates to a **ViewResolver** to obtain **View** implementation based on view name.
- The default ViewResolver treats the view name as a Web Application-relative file path
  - i.e. a JSP: `/WEB-INF/reward/list.jsp`
- Override this default by registering a ViewResolver bean with the DispatcherServlet
  - We will use **InternalResourceViewResolver**
  - Several other options available.

# Internal Resource View Resolver Example



# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
  - DispatcherServlet
  - Controllers
  - Views
- Quick Start

# Quick Start

Steps to developing a Spring MVC application

1. Deploy a Dispatcher Servlet (one-time only)
2. Implement a controller
3. Register the Controller with the DispatcherServlet
4. Implement the View(s)
5. Register a ViewResolver (optional, one-time only)
6. Deploy and test

Repeat steps 2-6 to develop new functionality

# 1a. Deploy DispatcherServlet

```
public class WebInitializer  
    extends AbstractAnnotationConfigDispatcherServletInitializer {  
  
    // Root context:  
    @Override protected Class<?>[] getRootConfigClasses() {  
        return new Class<?>[]{ RootConfig.class };  
    }  
  
    // DispatcherServlet context:  
    @Override protected Class<?>[] getServletConfigClasses() {  
        return new Class<?>[]{ MvcConfig.class };  
    }  
  
    // DispatcherServlet mapping:  
    @Override protected String[] getServletMappings() {  
        return new String[]{"/rewardsadmin/*"};  
    }  
}
```

Contains Spring MVC configuration

## 1b. Deploy DispatcherServlet

- Can handle URLs like ...

```
http://localhost:8080/mvc-1/rewardsadmin/reward/list
```

```
http://localhost:8080/mvc-1/rewardsadmin/reward/new
```

```
http://localhost:8080/mvc-1/rewardsadmin/reward/show?id=1
```

- We will implement *show*

# Initial Spring MVC Configuration

```
@Configuration  
public class MvcConfig {  
  
    // No beans required for basic Spring MVC usage.  
}
```

DispatcherServlet automatically defines several beans.  
Provide overrides to default values as desired (view resolvers).

## 2. Implement the Controller

```
@Controller  
public class RewardController {  
    private RewardLookupService lookupService;  
  
    @Autowired  
    public RewardController(RewardLookupService svc) {  
        this.lookupService = svc;  
    }  
  
    @RequestMapping("/reward/show")  
    public String show(@RequestParam("id") long id,  
                      Model model) {  
        Reward reward = lookupService.lookupReward(id);  
        model.addAttribute("reward", reward);  
        return "rewardView";  
    }  
}
```

Depends on application service

Automatically filled in by Spring

Selects the “rewardView” to render the reward

### 3. Register the Controller

```
@Configuration  
@ComponentScan("accounts.web")  
public class MvcConfig {
```

```
}
```

- Component-scanning very effective for MVC controllers!
- **Be specific** when indicating base package, avoid loading non-web layer beans
- Feel free to use <bean /> or @Configuration approaches as desired

## 4. Implement the View

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

References result model object by name

*/WEB-INF/views/rewardView.jsp*

Note: no references to Spring object / tags required in JSP.

# 5. Register ViewResolver

```
@Configuration  
@ComponentScan("accounts.web")  
public class MvcConfig {  
  
    @Bean  
    public ViewResolver simpleViewResolver() {  
        InternalResourceViewResolver vr =  
            new InternalResourceViewResolver();  
        vr.setPrefix ( "/WEB-INF/views/" );  
        vr.setSuffix ( ".jsp" );  
        return vr;  
    }  
}
```

Controller returns rewardList  
ViewResolver converts to /WEB-INF/views/rewardList.jsp

# 6. 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

# Lab

Adding a Web Interface

# MVC Additions from Spring 3.0

- @MVC and legacy Controllers enabled by default
  - Appropriate Controller Mapping and Adapters registered out-of-the-box
- New features *not* enabled by default
  - Stateless converter framework for binding & formatting
  - Support for JSR-303 declarative validation for forms
  - HttpMessageConverters (for RESTful web services)
- *How do you use these features?*

# @EnableWebMvc

- Registers Controller Mapping/Adapter for @MVC only
  - You lose legacy default mappings and adapters!
  - Enables custom conversion service and validators
  - Beyond scope of this course

```
@Configuration  
@EnableWebMvc  
public class RewardConfig {  
  
    @Bean  
    public rewardController(RewardLookupService service) {  
        return new RewardController(service);  
    }  
    ...  
}
```

# WebMvcConfigurerAdapter

- Optionally extend WebMvcConfigurerAdapter
  - Override methods to define/customize web-beans

```
@Configuration  
@EnableWebMvc  
public class RewardConfig extends WebMvcConfigurerAdapter {  
  
    @Bean public rewardController(RewardLookupService service) { ... }  
  
    @Override  
    public void addFormatters(FormatterRegistry registry) {  
        // Register your own type converters and formatters...  
    }  
    ...  
}
```



*Example: add custom formatters*

# MVC Namespace

- XML Equivalent to `@EnableWebMvc`

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="...">

    <!-- Provides default conversion service, validator and message converters -->
    <mvc:annotation-driven/>
```

*Learn More:  
Spring-Web – 4 day course on Spring Web Modules*

# Older Versions of Spring MVC

- Spring MVC is highly backwards compatible
  - Most default settings have remained unchanged since spring 2.5 (versions 3.0, 3.1, 3.2, 4.0, 4.1!)
- However, old default settings are no longer recommended
  - Newer styles of controllers, adapters, message convertors, validators ...
- Use `<mvc:annotation-config/>` or `@EnableWebMvc` to enable the more modern set of defaults

# Spring Boot

A new way to create Spring Applications

# Topics in this session

- **What is Spring Boot?**
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - Containerless Applications
  - Packaging
- Spring Boot inside of a Servlet Container
- Ease of Use Features
- Customizing Spring Boot

# What is Spring Boot?

- Spring Applications typically require a lot of setup
- Consider working with JPA. You need:
  - Datasource, TransactionManager, EntityManagerFactory, etc.
- Consider a web MVC app. You need:
  - WebApplicationInitializer / web.xml, ContextLoaderListener, DispatcherServlet, etc.
- An MVC app using JPA would need all of this
- Much of this is predictable

# What is Spring Boot?

- An opinionated runtime for Spring Projects
- Supports different project types, like Web and Batch
- Handles most low-level, predictable setup for you
- It is not:
  - A code generator
  - An IDE plugin



See: **Spring Boot Reference**

<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle>

# Opinionated Runtime?

- Spring Boot uses sensible defaults, mostly based on the classpath contents.
- E.g.:
  - Sets up a JPA Entity Manager Factory if a JPA implementation is on the classpath.
  - Creates a default Spring MVC setup, if Spring MVC is on the classpath.
- Everything can be overridden easily
  - But most of the time not needed

# Spring Boot Demo

# Topics in this session

- What is Spring Boot?
- **Spring Boot Explained**
  - Dependency Management
  - Auto Configuration
  - Containerless Applications
  - Packaging
- Spring Boot inside of a Servlet Container
- Ease of Use Features
- Customizing Spring Boot

# Spring Boot Dependencies

- You've been using Spring Boot already in this class!
  - Spring Boot defines parent POM and “starter” POMs

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>...</version>
</parent>
...
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
</dependencies>
```

- See next...

# Spring Boot Parent POM

- Parent POM defines key versions of dependencies and Maven plugin
  - Ultimately optional – you can have your own parent POM

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>...</version>
</parent>
```

- Note: Spring Boot works with Maven, Gradle, Ant/Ivy
  - But our content here will show Maven.

# Spring Starter POMs

- Allow an easy way to bring in multiple coordinated dependencies
  - “Transitive” Dependencies

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
</dependencies>
```

Version not needed!  
Defined by parent.

Resolves ~ 16 JARs!

- spring-boot-\* .jar
- spring-core-\* .jar
- spring-context-\* .jar
- spring-aop-\* .jar
- spring-beans-\* .jar
- aopalliance-\* .jar
- etc.

# Available Starter POMs

- Many Starter POMs available
- Web Starter (includes the basic starter):

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- Test Starter (adds Spring Test Framework and JUnit)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```



See: **Spring Boot Reference, Starter POMs**

<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter-poms>

# Topics in this session

- What is Spring Boot?
- Spring Boot Explained
  - Dependency Management
  - **Auto Configuration**
  - Containerless Applications
  - Packaging
- Spring Boot inside of a Servlet Container
- Ease of Use Features
- Customizing Spring Boot

# Spring Boot @EnableAutoConfiguration

- `@EnableAutoConfiguration` annotation on a Spring Java configuration class.
- Causes Spring Boot to automatically create beans it thinks are needed
  - Usually based on classpath contents
  - But you can easily override

```
@Configuration  
@EnableAutoConfiguration  
public class AppConfig {  
// ...  
}
```

# Shortcut @SpringBootApplication

- Very common to use `@EnableAutoConfiguration`, `@Configuration`, and `@ComponentScan` together.
- Boot 1.2 combines these with `@SpringBootApplication`

```
@SpringBootApplication  
public class AppConfig {  
    // ...  
}
```

# Topics in this session

- What is Spring Boot?
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - **Containerless Applications**
  - Packaging
- Spring Boot inside of a Servlet Container
- Ease of Use Features
- Customizing Spring Boot

# Spring Boot as a runtime

- Spring Boot can startup an embedded web server
  - You can run a web application from a JAR file!
- Tomcat included in Web Starter
- Jetty instead of Tomcat:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

# Why Run a Web Application Outside of a Container?

- No separation of container config and app config
  - They depend on each other anyway (like JNDI DS names, security config)
- Apps mostly target to a specific container
  - Why not include that already?
- Easier debugging and profiling
- Easier hot code replacement
- No special IDE support needed
- Familiar model for non-Java developers

# Topics in this session

- What is Spring Boot?
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - Containerless Applications
  - **Packaging**
- Spring Boot inside of a Servlet Container
- Ease of Use Features
- Customizing Spring Boot

# Packaging

- Spring Boot creates an “Ueber-Jar”
- Contains your classes as well as all 3<sup>rd</sup> party libs in one JAR (a JarJar).
- Can be executed with “java -jar yourapp.jar”
- Gradle and Maven plugins available
- Original JAR is still created

# Maven Packaging

- Add Boot Maven plugin to pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

# Packaging Result

- “mvn package” execution produces:

```
22M  yourapp-0.0.1-SNAPSHOT.jar  
5,1K  yourapp-0.0.1-SNAPSHOT.jar.original
```

- .jar.original contains only your code (traditional JAR file)
- .jar contains your code and all libs

# Topics in this session

- What is Spring Boot?
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - Containerless Applications
  - Packaging
- **Spring Boot inside of a Servlet Container**
- Ease of Use Features
- Customizing Spring Boot

# Spring Boot in a Servlet Container

- Spring Boot can run in any Servlet 3.x container
- Only small changes required
  - Change artifact type to WAR (instead of JAR)
  - Extend SpringBootServletInitializer
  - Override configure method
- Still no web.xml required
- Produces hybrid WAR file
  - Can still be executed with embedded Tomcat using “java -jar yourapp.war”
- Traditional WAR file (without embedded Tomcat) is produced as well

# Spring Boot in a Servlet Container

```
@ComponentScan  
@EnableAutoConfiguration  
public class Application extends SpringBootServletInitializer {  
  
    protected SpringApplicationBuilder configure(  
        SpringApplicationBuilder application) {  
        return application.sources(Application.class);  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
}
```

# WAR packaging result

- “mvn package” execution produces:

```
22M  yourapp-0.0.1-SNAPSHOT.war  
20M  yourapp-0.0.1-SNAPSHOT.war.original
```

- .war.original is a traditional WAR file
- .war is a hybrid WAR file, additionally containing the embedded Tomcat
  - Can be executed with “java -jar yourapp-0.0.1-SNAPSHOT.war”

# It's Your Choice

- There is no force to go containerless
  - Embedded container is just one feature of Spring Boot
- Traditional WAR also benefits a lot from Spring Boot:
  - Automatic Spring MVC setup, including DispatcherServlet
  - Sensible defaults based on the classpath content
  - Embedded container can be used during development

# Topics in this session

- What is Spring Boot?
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - Containerless Applications
  - Packaging
- Spring Boot inside of a Servlet Container
- **Ease of Use Features**
- Customizing Spring Boot

# Externalized Properties: application.properties

- Developers commonly externalize properties to files
  - Easily consumable via Spring PropertySource
- ...But developers name / locate their files different ways.
- Spring Boot will automatically look for **application.properties** in the classpath root.
- Or use **application.yml** if you prefer YAML

```
database.host: localhost  
database.user: admin  
database.poolsize: 2-20
```

application.properties

```
database  
host: localhost  
user: admin  
poolsize: 2-20
```

application.yml

# Relaxed property binding

- No need for an exact match between desired properties and names
- Intuitive mapping between system properties and environment variables
  - test.property equivalent to TEST\_PROPERTY
- Access via SpEL

Or without Spring Boot:

```
@Value("${TEST_PROPERTY}")
```

```
@Configuration  
class AppConfig {  
  
    @Value("${test.property}")  
    String testProperty;  
  
    @Bean SomeObject myBean() { ... }  
}
```

# @ConfigurationProperties

- Easily set multiple properties on a bean with one annotation
- Enable with `@EnableConfigurationProperties` in configuration class

```
@Component  
@ConfigurationProperties(prefix="my.car")  
public class Car {  
    private String color;  
    private String engine;  
    // getters / setters
```

```
my.car.color=red  
my.car.engine=5.0
```

application.properties

# @ConfigurationProperties with @Bean

- Set properties on @Bean method automatically
- Replace this:

```
@Bean
public DataSource dataSource(
    @Value("${db.credentials.driver}") dbDriver,
    @Value("${db.credentials.url}") dbUrl,
    @Value("${db.credentials.user}") dbUser,
    @Value("${db.credentials.password}") dbPassword) {
    DataSource ds = new BasicDataSource();
    ds.setDriverClassName( dbDriver );
    ds.setUrl( dbUrl );
    ds.setUser( dbUser );
    ds.setPassword( dbPassword ) );
    return ds;
}
```

# @ConfigurationProperties with @Bean

- Set properties on @Bean method automatically
- Replace this:
- With this:

```
@Bean  
@ConfigurationProperties(prefix="db.credentials")  
public DataSource dataSource() {  
    return new BasicDataSource();  
}
```

# Even Easier...

- Use either **spring-boot-starter-jdbc** or **spring-boot-starter-data-jpa** and include a JDBC driver on classpath
- Declare properties

```
spring.datasource.url=jdbc:mysql://localhost/test  
spring.datasource.username=dbuser  
spring.datasource.password=dbpass  
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

- That's It!
  - Spring Boot will create a DataSource with properties set.
  - Will even use a connection pool if one is found on the classpath!

# Controlling Logging Level

- Boot can control the logging level
- Just set the logging level in application.properties
- Works with most logging frameworks
  - Java Util Logging, Logback, Log4J, Log4J2

```
logging.level.org.springframework: DEBUG  
logging.level.com.acme.your.code: INFO
```

# Web Application Convenience

- Boot automatically configures Spring MVC DispatcherServlet and `@EnableWebMvc` defaults
  - When `spring-webmvc*.jar` on classpath
- Static resources served from classpath
  - `/static`, `/public`, `/resources` or `/META-INF/resources`
- Templates served from `/templates`
  - When Velocity, Freemarker, Thymeleaf, or Groovy on classpath
- Provides default `/error` mapping
  - Easily overridden

# Topics in this session

- What is Spring Boot?
- Spring Boot Explained
  - Dependency Management
  - Auto Configuration
  - Containerless Applications
  - Packaging
- Spring Boot inside of a Servlet Container
- Ease of Use Features
- **Customizing Spring Boot**

# Customizing Spring Boot

- Many ways to customize Spring Boot
  - application.properties
  - Replacing generated beans
  - Selectively disabling auto configuration

# Application properties

- Spring Boot looks for **application.properties** in these locations (in this order):
  - /config subdir of the working directory
  - The working directory
  - config package in the classpath
  - classpath root
- Creates a PropertySource based on these files.
- Check AutoConfigure classes for available properties
  - e.g. spring.datasource.name in  
EmbeddedDataSourceConfiguration.class



See: [Spring Boot Reference, Appendix A. Common Application Properties](#)  
<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties>

# Replacing generated beans (1)

- Self declared beans typically disable automatically created ones.
  - E.g. Own DataSource stops creating Spring Boot DataSource
  - Bean name often not important
  - XML, Component Scan and Java Config supported

```
@Bean  
public DataSource dataSource() {  
    return new EmbeddedDatabaseBuilder()  
        .setName("RewardsDb").build();  
}
```

# Replacing generated beans (2)

- Check AutoConfigure classes for details:

```
@Configuration
public class DataSourceAutoConfiguration
    implements EnvironmentAware {
    ...
    @Conditional(...)
    @ConditionalOnMissingBean(DataSource.class)
    @Import(...)
    protected static class EmbeddedConfiguration { ... }
    ...
}
```

# Selectively disabling auto configuration

- Check /autoconfig to see active configuration classes
- “exclude” completely disables them:

```
@EnableAutoConfiguration(exclude=EmbeddedDatabaseConfigurer.class)
public class Application extends SpringBootServletInitializer {
    ...
}
```

# Additional Integrations

- Spring Data (see lab)
- Spring Security
- MongoDB
- Testing
- Cloud Foundry
- ...

# Summary

- Spring Boot speeds up Spring application development
- You always have full control and insight
- Nothing is generated
- No special runtime requirements
- No servlet container needed (if you want)
  - E.g. ideal for microservices
- Stay tuned for much more features soon

# Spring Boot Lab

# Spring Security

## Web Application Security

Addressing Common Security Requirements

# Topics in this Session

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

# Security Concepts

- Principal
  - User, device or system that performs an action
- Authentication
  - Establishing that a principal's credentials are valid
- Authorization
  - Deciding if a principal is allowed to perform an action
- Secured item
  - Resource that is being secured

# Authentication

- There are many authentication mechanisms
  - e.g. basic, digest, form, X.509
- There are many storage options for credential and authority information
  - e.g. Database, LDAP, in-memory (development)

# Authorization

- Authorization depends on authentication
  - Before deciding if a user can perform an action, user identity must be established
- The decision process is often based on roles
  - ADMIN can cancel orders
  - MEMBER can place orders
  - GUEST can browse the catalog

# Topics in this Session

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



See: **Spring Security Reference**

<http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/>

# Motivations - I

- Spring Security is portable across containers
  - Secured archive (WAR, EAR) can be deployed as-is
  - Also runs in standalone environments
  - Uses Spring for configuration
- Separation of Concerns
  - Business logic is decoupled from security concerns
  - Authentication and Authorization are decoupled
    - Changes to the authentication process have *no impact* on authorization

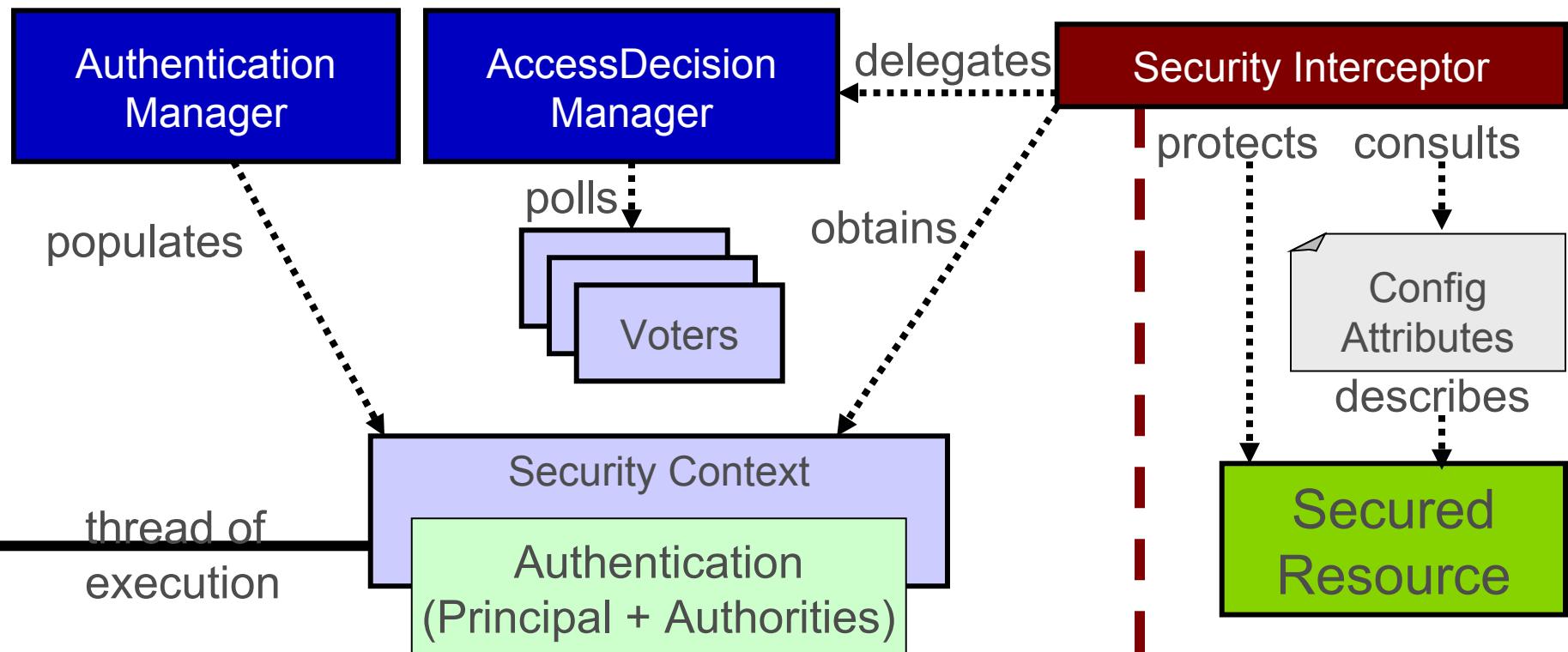
# Motivations: II

- Flexibility
  - Supports all common authentication mechanisms
    - Basic, Form, X.509, Cookies, Single-Sign-On, etc.
  - Configurable storage options for user details (credentials and authorities)
    - RDBMS, LDAP, custom DAOs, properties file, etc.
- Extensible
  - All the following can be customized
    - How a principal is defined
    - How authorization decisions are made
    - Where security constraints are stored

# Consistency of Approach

- The goal of authentication is *always the same* regardless of the mechanism
  - Establish a security context with the authenticated principal's information
  - Out-of-the-box this works for web applications
- The process of authorization is *always the same* regardless of resource type
  - Consult the attributes of the secured resource
  - Obtain principal information from security context
  - Grant or deny access

# Spring Security – the Big Picture



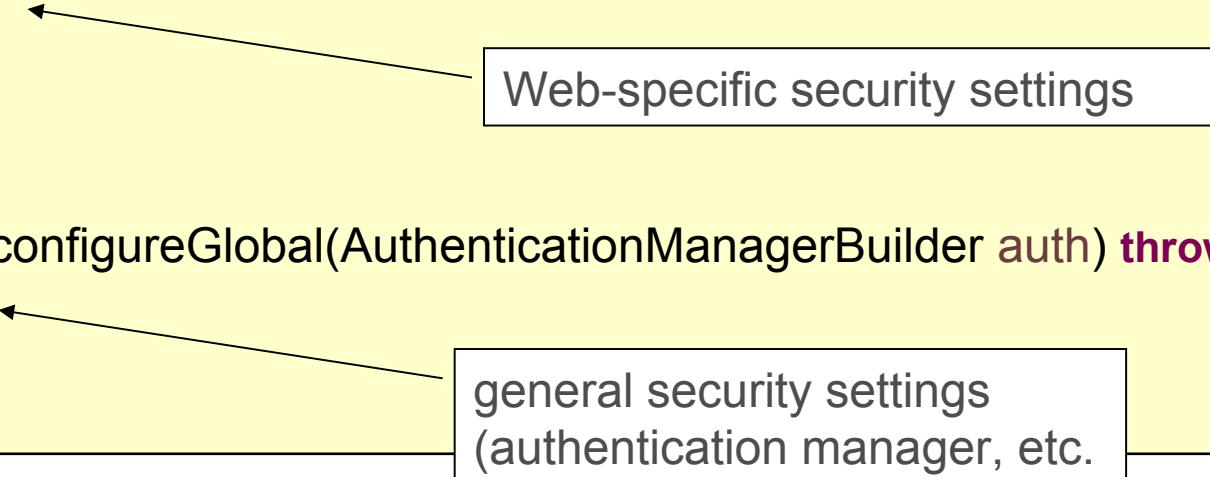
# Topics in this Session

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

# Configuration in the Application Context

- Java Configuration (XML also available)
- Extend WebSecurityConfigurerAdapter for easiest use

```
@Configuration  
@EnableWebMvcSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        //  
    }  
  
    @Autowired  
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
        //  
    }  
}
```



Web-specific security settings

general security settings  
(authentication manager, etc.)

# Configuration in web.xml

- Define the single proxy filter
  - springSecurityFilterChain is a mandatory name
  - Refers to an existing Spring bean with same name

```
<filter>                                              web.xml
  <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>
```

# authorizeRequests()

- Adds specific authorization requirements to URLs
- Evaluated in the order listed
  - first match is used, put specific matches first

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
        .antMatchers("/css/**", "/images/**", "/javascript/**").permitAll()  
        .antMatchers("/accounts/edit*").hasRole("ADMIN")  
        .antMatchers("/accounts/account*").hasAnyRole("USER", "ADMIN")  
        .antMatchers("/accounts/**").authenticated()  
        .antMatchers("/customers/checkout*").authenticatedFully()  
        .antMatchers("/customers/**").anonymous()  
}
```

# Specifying login and logout

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
            .antMatchers("/aaa*").hasRole("ADMIN")  
            .and()                                // method chaining!  
  
        .formLogin()                            // setup form-based authentication  
            .loginPage("/login.jsp")             // URL to use when login is needed  
            .permitAll()                      // any user can access  
            .and()                            // method chaining!  
  
        .logout()                             // configure logout  
            .permitAll();                     // any user can access  
}
```

# An Example Login Page

URL that indicates an authentication request.  
Default: POST against URL used to display the page.

```
<c:url var='loginUrl' value='/login.jsp' />
<form:form action="${loginUrl}" method="POST">
    <input type="text" name="username"/>
    <br/>
    <input type="password" name="password"/>
    <br/>
    <input type="submit" name="submit" value="LOGIN"/>
</form:form>
```

The expected keys  
for generation of  
an authentication  
request token

*login-example.jsp*

# Topics in this Session

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

# Configure Authentication

- DAO Authentication provider is default
  - Expects a *UserDetailsService* implementation to provide credentials and authorities
    - Built-in: In-memory (properties), JDBC (database), LDAP
    - Custom
- Or define your own Authentication provider
  - *Example:* to get pre-authenticated user details when using single sign-on
    - CAS, TAM, SiteMinder ...
  - See online examples

# Authentication Provider

- Use a `UserDetailsManagerConfigurer`
  - Three built in options:
    - LDAP
    - JDBC
    - In-memory (for quick testing)
  - Or use your own `UserDetailsService` implementation

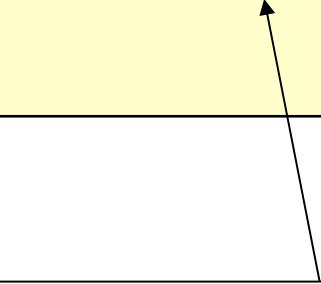
```
@Autowired  
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
    auth  
        .inMemoryAuthentication()  
            .withUser("hugie").password("hugie").roles("GENERAL").and()  
            .withUser("dewey").password("dewey").roles("ADMIN").and()  
            .withUser("louie").password("louie").roles("SUPPORT");  
}
```

The diagram illustrates the configuration of an in-memory authentication manager. It shows the code snippet within a yellow box. Annotations explain the components: 'Not web-specific' points to the `@Autowired` and `throws Exception` parts; 'Adds a UserDetailsManagerConfigurer' points to the `inMemoryAuthentication()` call; and three arrows point from the labels 'login', 'password', and 'Supported roles' to their respective parameters in the configuration code.

# Sourcing Users from a Database

- Configuration:

```
@Autowired DataSource dataSource;  
  
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
  
    auth.jdbcAuthentication().dataSource(dataSource);  
}  
}
```



Can customize queries using methods:  
`usersByUsernameQuery()`  
`authoritiesByUsernameQuery()`  
`groupAuthoritiesByUsername()`

# Sourcing Users from a Database

Queries RDBMS for users and their authorities

- Provides default queries
  - `SELECT username, password, enabled FROM users WHERE username = ?`
  - `SELECT username, authority FROM authorities WHERE username = ?`
- Groups also supported
  - `groups`, `group_members`, `groupAuthorities` tables
  - See online documentation for details
- Advantage
  - Can modify user info while system is running

# Password Encoding

- Can encode passwords using a hash
  - sha, md5, bcrypt, ...

```
auth.jdbcAuthentication()  
    .dataSource(dataSource)  
    .passwordEncoder(new StandardPasswordEncoder());
```

SHA-256 encoding

- Secure passwords using a well-known string
  - Known as a 'salt', makes brute force attacks harder

```
auth.jdbcAuthentication()  
    .dataSource(dataSource)  
    .passwordEncoder(new StandardPasswordEncoder("sodium-chloride"));
```

encoding with salt

# Other Authentication Options

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

Authorization is *not* affected by changes to Authentication!

# @Profile with Security Configuration

- Use @Profile to separate development vs. production

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {  
    protected void configure(HttpSecurity http) throws Exception {  
        http.authorizeRequests().antMatchers("/resources/**").permitAll();  
    }  
}
```

```
@Configuration  
@EnableWebMvcSecurity  
@Profile("development")  
public class SecurityDevConfig extends SecurityBaseConfig {  
    @Autowired  
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
        auth  
            .inMemoryAuthentication()  
                .withUser("hughie").password("hughie").roles("GENERAL");  
    }  
}
```

# Topics in this Session

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

# Tag library declaration

- The Spring Security tag library is declared as follows

available since Spring Security 2.0

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

Facelet tags for JSF are also available

- You need to define and install them manually
- See “*Using the Spring Security Facelets Tag Library*” in the Spring Webflow documentation
- Principle is available in SpEL: #{principle.username}

# Spring Security's Tag Library

- Display properties of the Authentication object

You are logged in as:

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

jsp

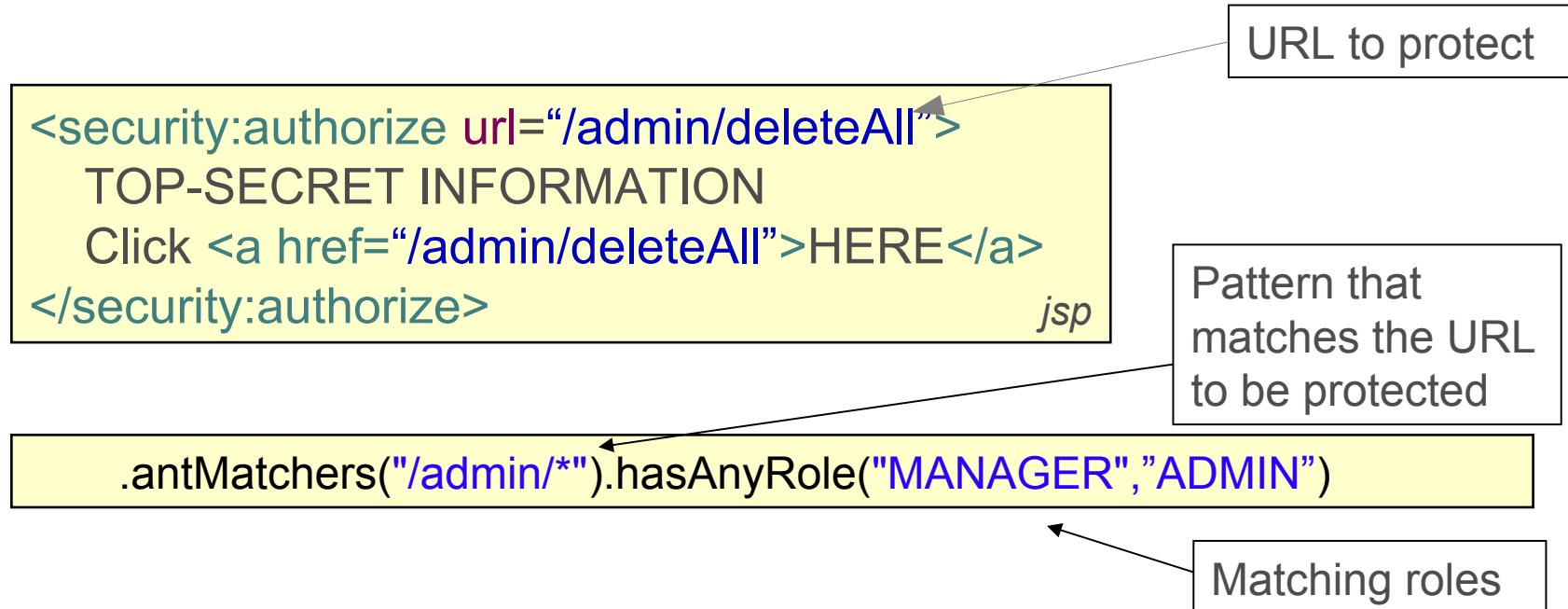
- Hide sections of output based on role

```
<security:authorize access="hasRole('ROLE_MANAGER')">  
TOP-SECRET INFORMATION  
Click <a href="/admin/deleteAll">HERE</a> to delete all records.  
</security:authorize>
```

jsp

# Authorization in JSP based on intercept-url

- Role declaration can be centralized in Spring config files



# Topics in this Session

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

# Method Security

- Spring Security uses AOP for security at the method level
  - annotations based on Spring annotations or JSR-250 annotations
  - XML configuration with the Spring Security namespace
- Typically secure your services
  - Do not access repositories directly, bypasses security (and transactions)

# Method Security - JSR-250

- JSR-250 annotations should be enabled

```
@EnableGlobalMethodSecurity(jsr250Enabled=true)
```

```
import javax.annotation.security.RolesAllowed;  
  
public class ItemManager {  
    @RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})  
    public Item findItem(long itemNumber) {  
        ...  
    }  
}
```



Only supports *role-based security* – hence the name

# Method Security - @Secured

- Secured annotation should be enabled

```
@EnableGlobalMethodSecurity(securedEnabled=true)
```

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

```
public class ItemManager {  
    @Secured("IS_AUTHENTICATED_FULLY")  
    public Item findItem(long itemNumber) {
```

```
    ...  
}
```

```
        @Secured("ROLE_MEMBER")  
        @Secured({"ROLE_MEMBER", "ROLE_USER"})
```



*Spring 2.0 syntax, so **not** limited to roles. SpEL **not** supported.*

# Method Security with SpEL

- Use Pre/Post annotations for SpEL

```
@EnableGlobalMethodSecurity(prePostEnabled=true)
```

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

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

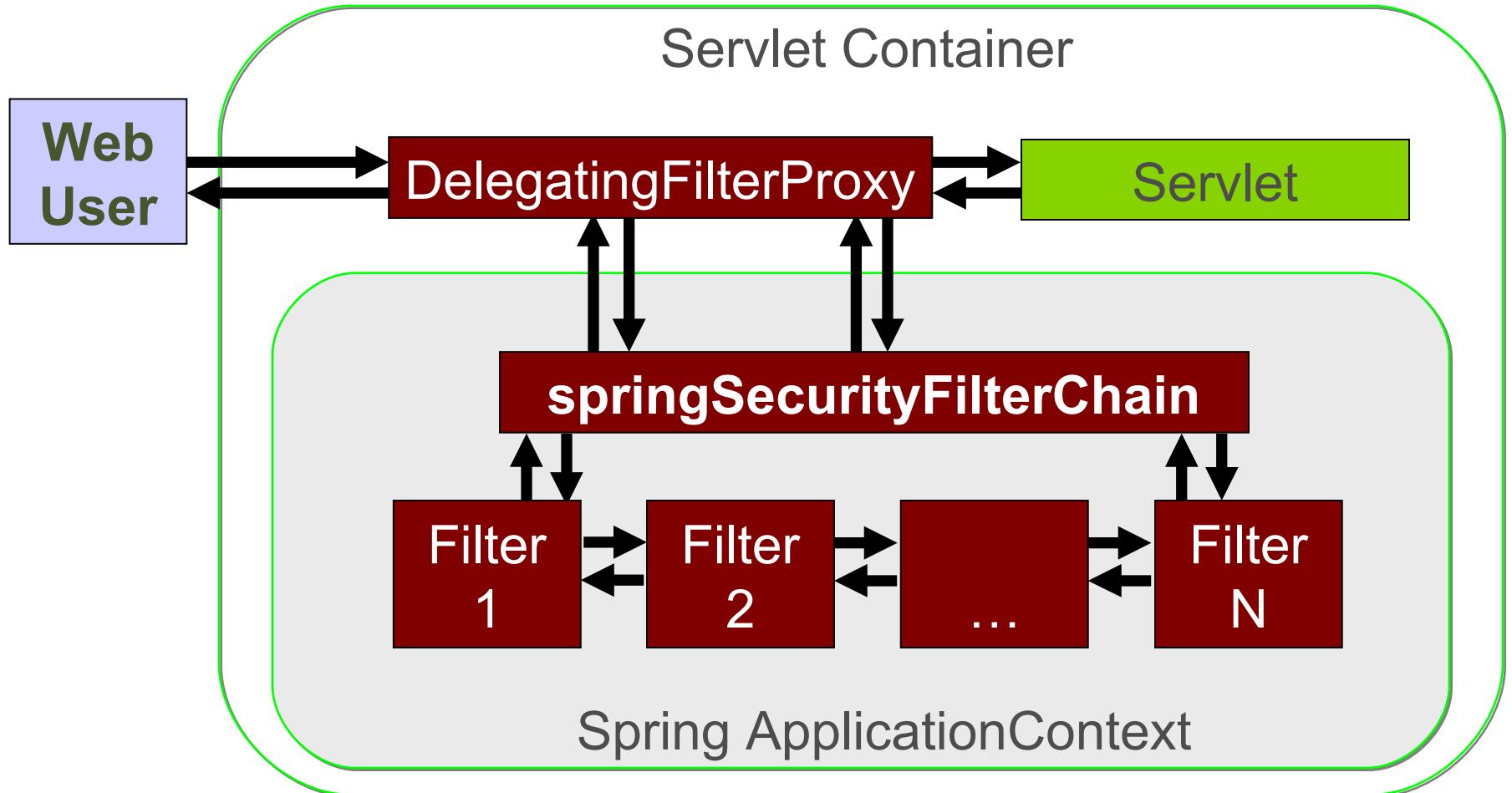
# Topics in this Session

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

# Spring Security in a Web Environment

- `springSecurityFilterChain` is declared in `web.xml`
- This single proxy filter delegates to a chain of Spring-managed filters
  - Drive authentication
  - Enforce authorization
  - Manage logout
  - Maintain `SecurityContext` in `HttpSession`
  - *and more*

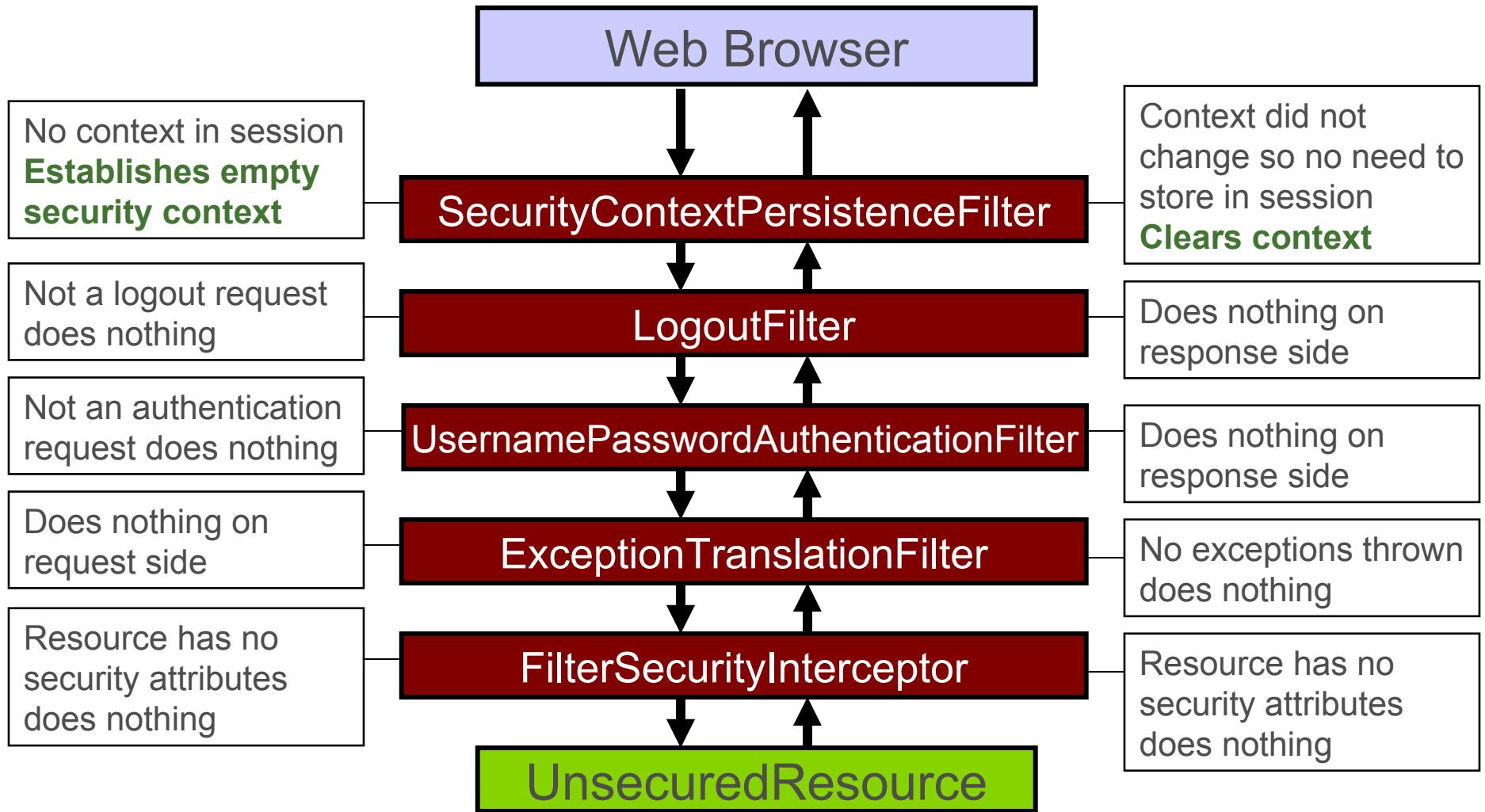
# Web Security Filter Configuration



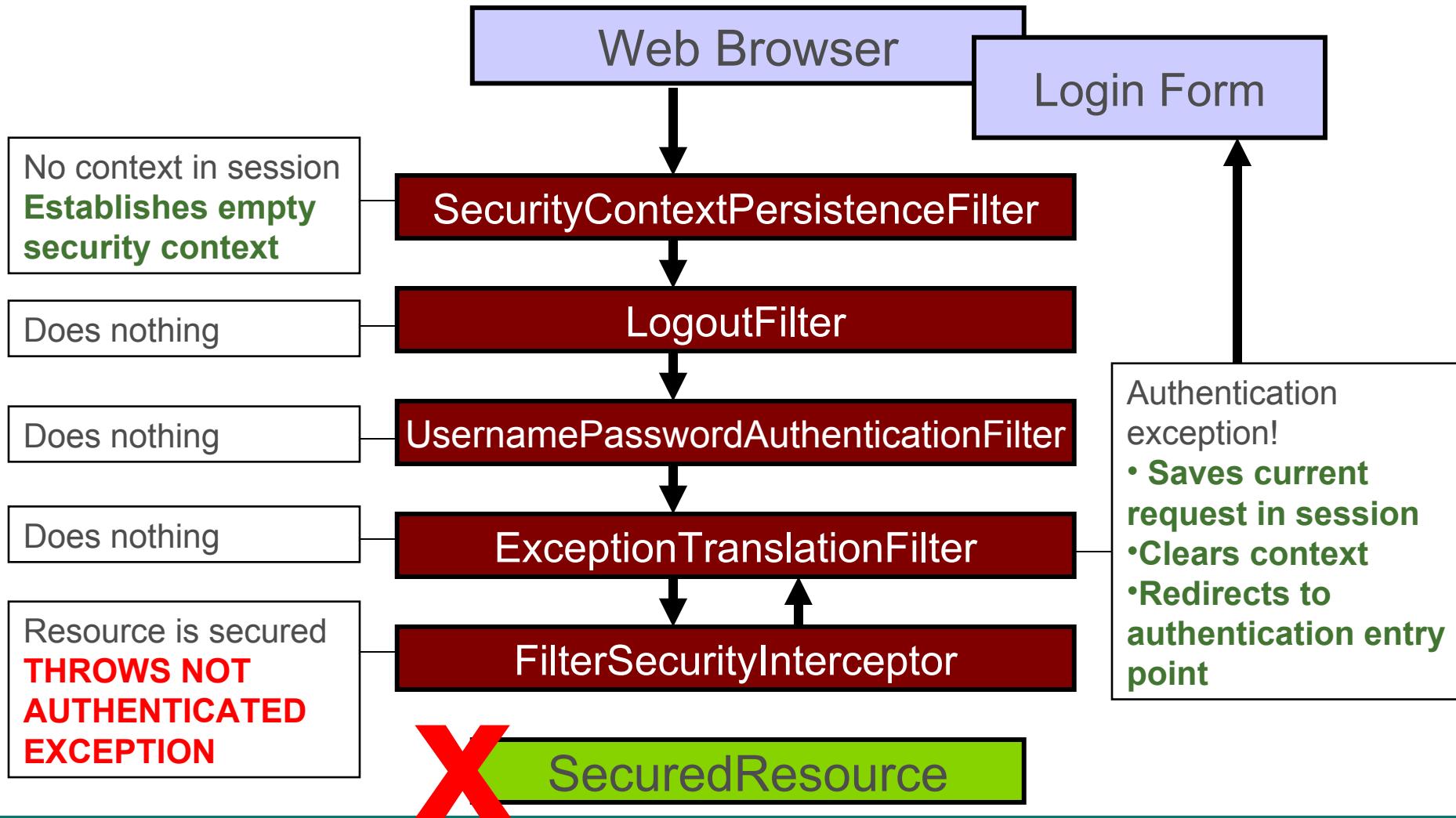
# The Filter chain

- With ACEGI Security 1.x
  - Filters were manually configured as individual <bean> elements
  - Led to verbose and error-prone XML
- Spring Security 2.x and 3.x
  - Filters are initialized with correct values by default
  - Manual configuration is not required **unless you want to customize Spring Security's behavior**
  - It is still important to understand how they work underneath

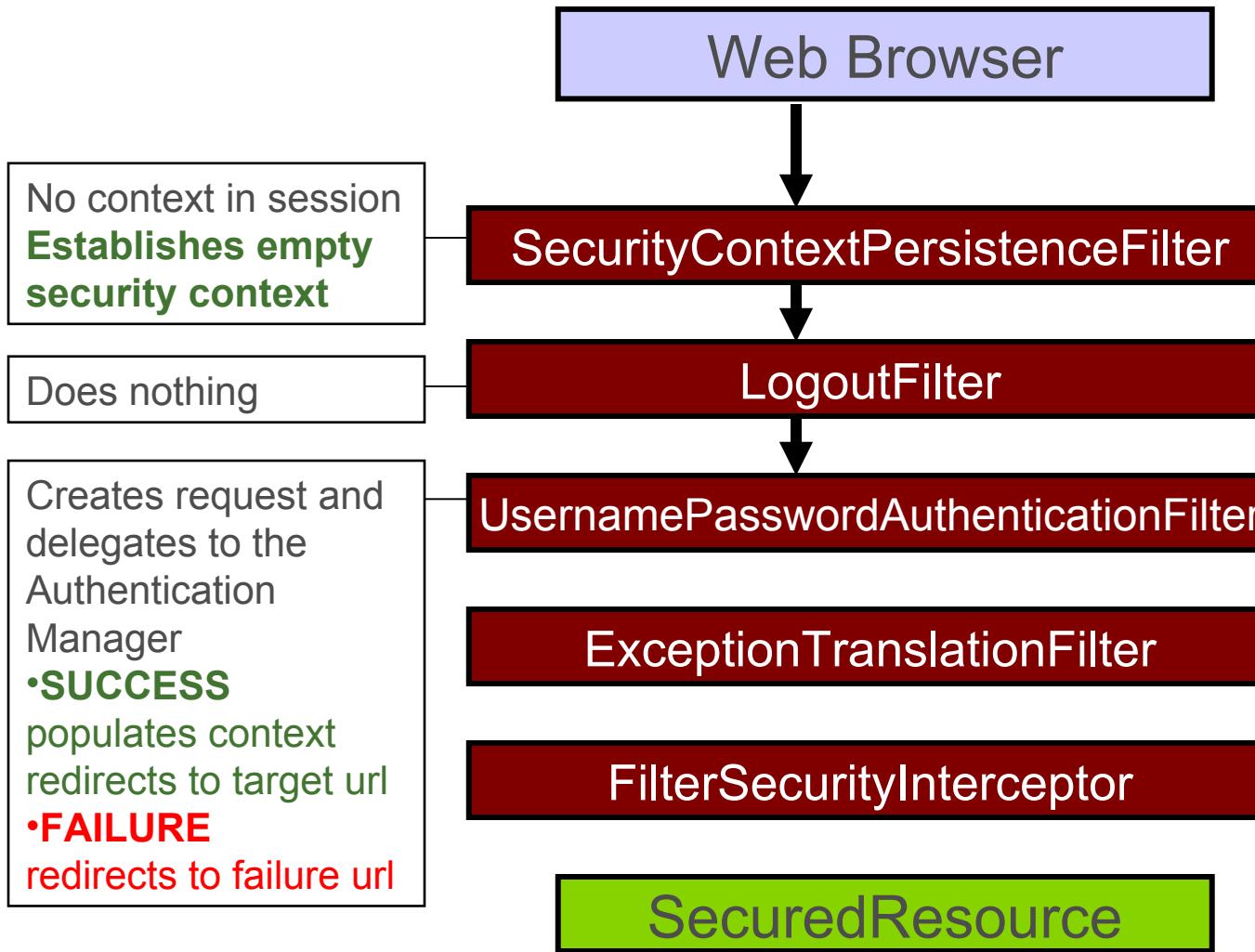
# Access Unsecured Resource Prior to Login



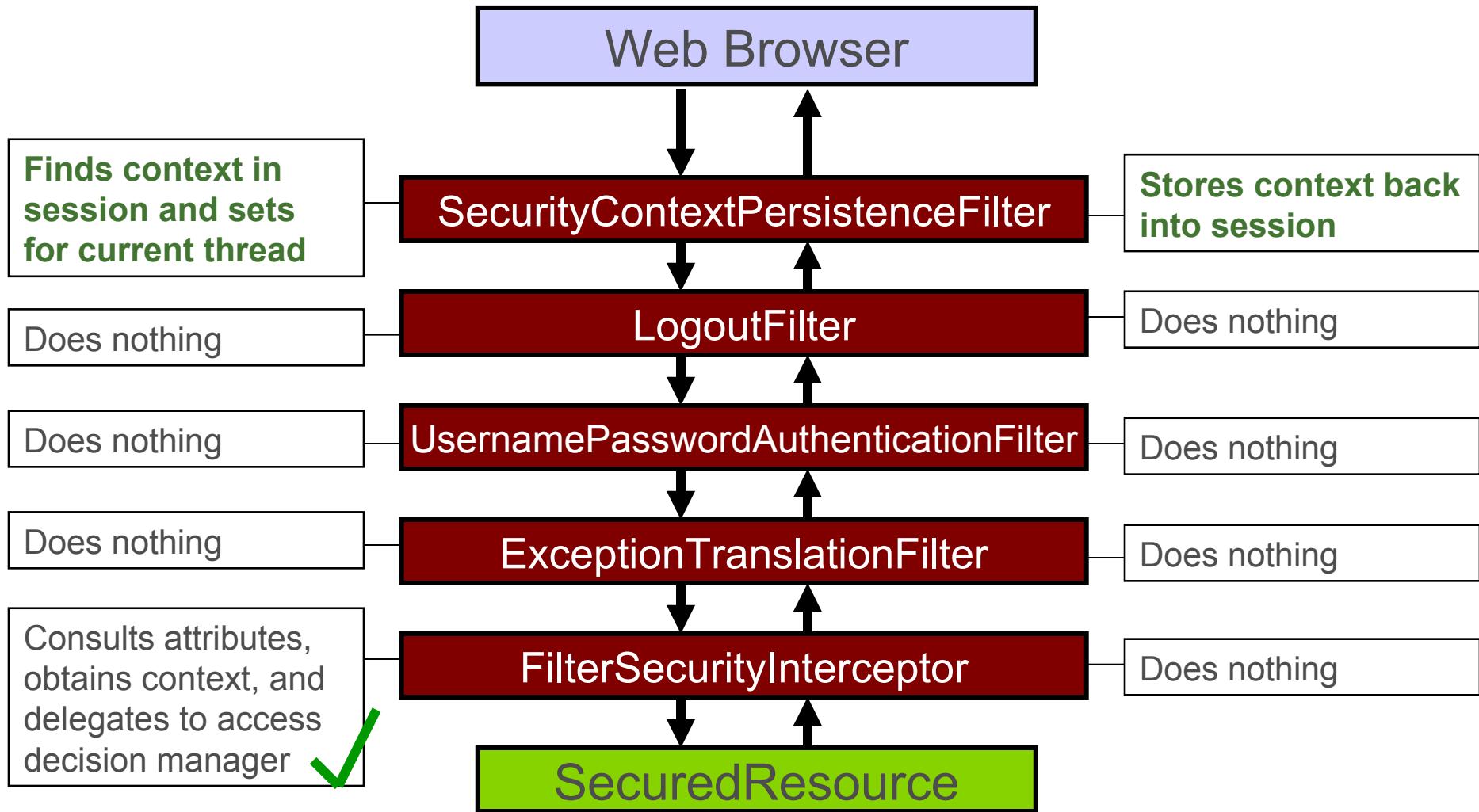
# Access Secured Resource Prior to Login



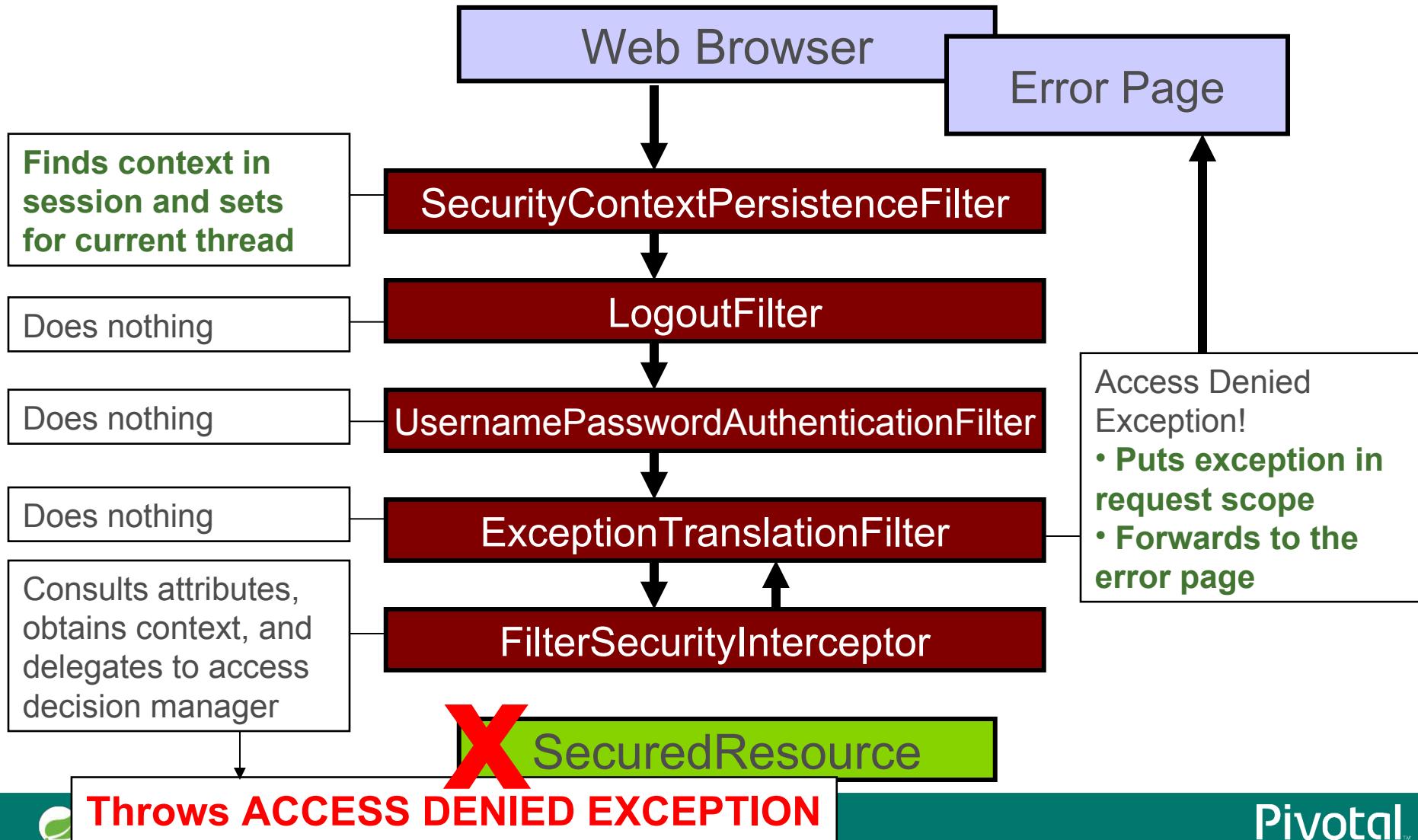
# Submit Login Request



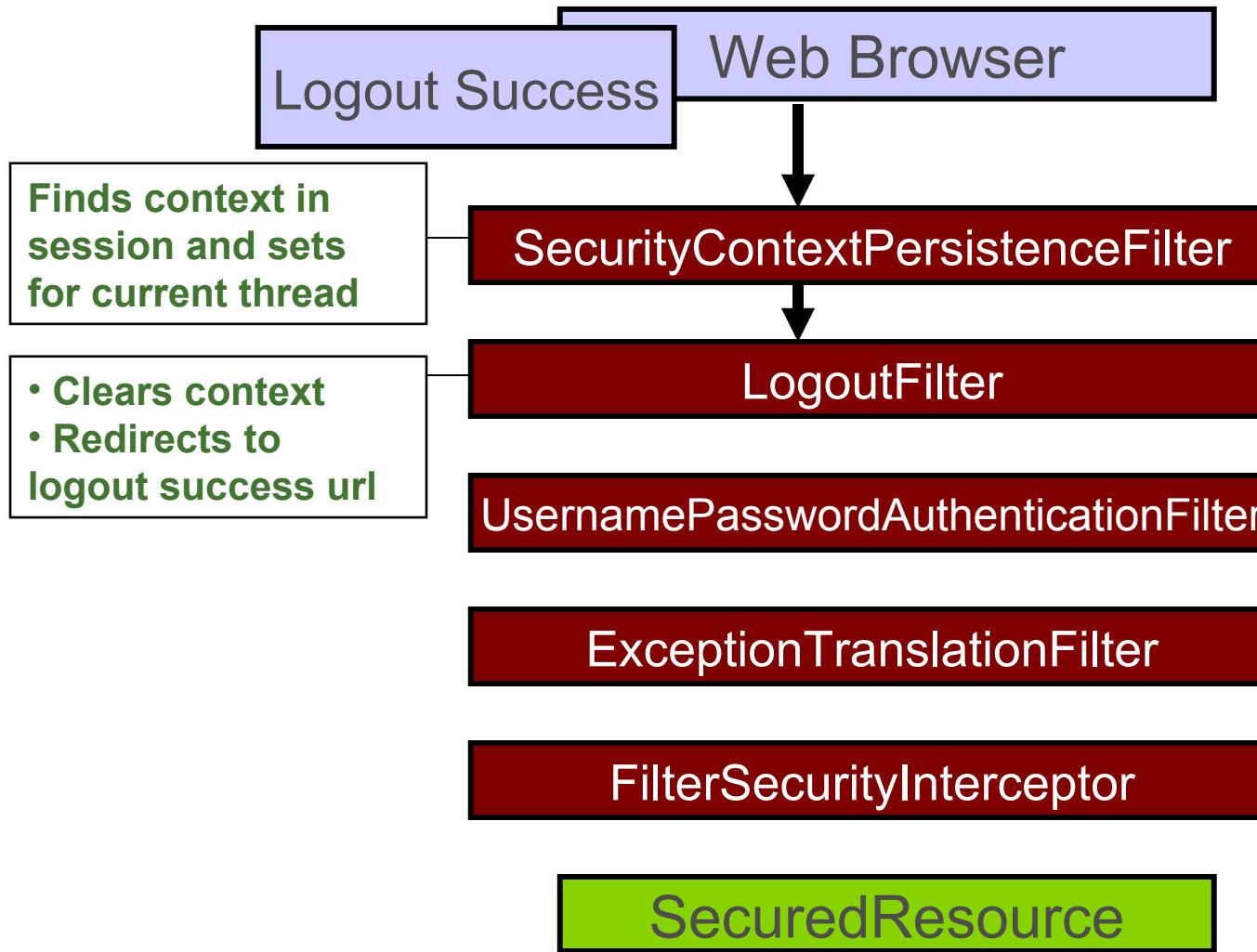
# Access Resource With Required Role



# Access Resource Without Required Role



# Submit Logout Request



# The Filter Chain: Summary

#	Filter Name	Main Purpose
1	SecurityContext IntegrationFilter	Establishes SecurityContext and maintains between HTTP requests <i>formerly: HttpSessionContextIntegrationFilter</i>
2	LogoutFilter	Clears SecurityContextHolder when logout requested
3	UsernamePassword AuthenticationFilter	Puts Authentication into the SecurityContext on login request <i>formerly: AuthenticationProcessingFilter</i>
4	Exception TranslationFilter	Converts SpringSecurity exceptions into HTTP response or redirect
5	FilterSecurity Interceptor	Authorizes web requests based on config attributes and authorities

# Custom Filter Chain

- Filter can be **added** to the chain
  - Before or after existing filter

```
http.addFilterAfter ( myFilter, UsernamePasswordAuthenticationFilter.class );
```

```
...
```

```
@Bean  
public Filter myFilter() {  
    return new MySpecialFilter();  
}
```

- Filter on the stack may be **replaced** by a custom filter
  - Replacement must extend the filter being replaced.

```
http.addFilter ( myFilter );
```

```
...
```

```
@Bean  
public Filter myFilter() {  
    return new MySpecialFilter();  
}
```

```
public class MySpecialFilter  
    extends UsernamePasswordAuthenticationFilter {}
```

# Lab

Applying Spring Security to a Web Application

# Practical REST Web Services

Using Spring MVC to build RESTful Web Services

Extending Spring MVC to handle REST

# Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - Using MessageConverters
  - Content Negotiation
  - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

# REST Introduction

- Web apps not just usable by browser clients
  - Programmatic clients can also connect via HTTP
  - Such as: mobile applications, AJAX enabled web-pages
- REST is an *architectural style* that describes best practices to expose web services over HTTP
  - REpresentational State Transfer, term by Roy Fielding
  - HTTP as *application* protocol, not just transport
  - Emphasizes scalability
  - *Not* a framework or specification

# REST Principles (1)

- Expose *resources* through URIs
  - Model nouns, not verbs
  - <http://mybank.com/banking/accounts/123456789>
- Resources support limited set of operations
  - GET, PUT, POST, DELETE in case of HTTP
  - All have well-defined semantics
- Example: update an order
  - PUT to </orders/123>
  - don't POST to </order/edit?id=123>



# REST Principles (2)

- Clients can request particular representation
  - Resources can support multiple representations
  - HTML, XML, JSON, ...
- Representations can link to other resources
  - Allows for extensions and discovery, like with web sites
- Hypermedia As The Engine of Application State
  - *HATEOAS*: Probably the world's worst acronym!
  - RESTful responses contain the links you need – just like HTML pages do

# REST Principles (3)

- Stateless architecture
  - No HttpSession usage
  - GETs can be cached on URL
  - Requires clients to keep track of state
  - Part of what makes it scalable
  - Looser coupling between client and server
- HTTP headers and status codes communicate result to clients
  - All well-defined in HTTP Specification

# Why REST?

- Benefits of REST
  - Every platform/language supports HTTP
    - Unlike for example SOAP + WS-\* specs
  - Easy to support many different clients
    - Scripts, Browsers, Applications
  - Scalability
  - Support for redirect, caching, different representations, resource identification, ...
  - Support for XML, but also other formats
    - JSON and Atom are popular choices



# Topics in this Session

- REST introduction
- **REST and Java**
- Spring MVC support for RESTful applications
- RESTful clients with the RestTemplate
- Conclusion

# REST and Java: JAX-RS

- JAX-RS is a Java EE 6 standard for building RESTful applications
  - Focuses on programmatic clients, not browsers
- Various implementations
  - Jersey (RI), RESTEasy, Restlet, CXF
  - All implementations provide Spring support
- Good option for full REST support using a standard
- No support for building clients in standard
  - Although some implementations do offer it

# REST and Java: Spring-MVC

- Spring-MVC provides REST support as well
  - Since version 3.0
  - Using familiar and consistent programming model
  - Spring MVC does not implement JAX-RS
- Single web-application for everything
  - Traditional web-site: HTML, browsers
  - Programmatic client support (RESTful web applications, HTTP-based web services)
- RestTemplate for building programmatic clients in Java

# Topics in this Session

- REST introduction
- REST and Java
- **Spring MVC support for RESTful applications**
  - Request/Response Processing
  - Using MessageConverters
  - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

# Spring-MVC and REST

- Some features have been covered already
  - URI templates for 'RESTful' URLs
- Will now extend Spring MVC to support REST
  - Map requests based on HTTP method
  - More annotations
  - Message Converters
- Support for RESTful web applications targeting browser-based clients is also offered
  - See *HttpMethodFilter* in online documentation

# Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
  - **Request/Response Processing**
  - Using MessageConverters
  - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

# Request Mapping Based on HTTP Method

- Can map HTTP requests based on method
  - Allows same URL to be mapped to multiple methods
  - Often used for form-based controllers (GET & POST)
  - Essential to support RESTful resource URLs
    - incl. PUT and DELETE

```
@RequestMapping(value="/orders", method=RequestMethod.GET)
public void listOrders(Model model) {
```

```
    // find all Orders and add them to the model
```

```
}
```

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
public void createOrder(HttpServletRequest request, Model model) {
```

```
    // process the order data from the request
```

```
}
```

# HTTP Status Code Support

- Web apps just use a handful of status codes
  - Success: 200 OK
  - Redirect: 302/303 for Redirects
  - Client Error: 404 Not Found
  - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many *additional* codes to communicate with their clients
- Add `@ResponseStatus` to controller method
  - don't have to set status on `HttpServletResponse` manually

# Common Response Codes

200

- ◆ After a successful GET returning content

201

- ◆ New resource was created on POST or PUT
- ◆ Location header contains its URI

204

- ◆ The response is empty – such as after successful PUT or DELETE

404

- ◆ Requested resource was not found

405

- ◆ HTTP method is not supported by resource

409

- ◆ Conflict while making changes – such as POSTing unique data that already exists

415

- ◆ Request body type not supported

# @ResponseStatus

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public void createOrder(HttpServletRequest request,
                        HttpServletResponse response) {
    Order order = createOrder(request);
    // determine full URI for newly created Order based on request
    response.addHeader("Location",
        getLocationForChildResource(request, order.getId()));
}
```

- **IMPORTANT:** When using `@ResponseStatus`
  - `void` methods *no longer* imply a default view name!
  - There will be no View at all
  - This example gives a response with an *empty* body

# Determining Location Header

- Location header value must be full URL
  - Determine based on request URL
    - Controller shouldn't know host name or servlet path
- URL of created child resource usually a sub-path
  - POST to <http://www.myshop.com/store/orders> gives <http://www.myshop.com/store/orders/123>
  - Can use Spring's UriTemplate for encoding where needed

```
private String getLocationForChildResource(HttpServletRequest request,  
                                         Object childIdentifier) {  
    StringBuffer url = request.getRequestURL();  
    UriTemplate template = new UriTemplate(url.append("/{childId}").toString());  
    return template.expand(childIdentifier).toASCIIString();  
}
```

# @ResponseStatus & Exceptions

- Can also annotate exception classes with this
  - Given status code used when exception is thrown from controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
public class OrderNotFoundException extends RuntimeException {
    ...
}
```

```
@RequestMapping(value="/orders/{id}", method=GET)
public String showOrder(@PathVariable("id") long id, Model model) {
    Order order = orderRepository.findOrderById(id);
    if (order == null) throw new OrderNotFoundException(id);
    model.addAttribute(order);
    return "orderDetail";
}
```

# @ExceptionHandler

- For existing exceptions you cannot annotate, use @ExceptionHandler method on controller
  - Method signature similar to request handling method
  - Also supports @ResponseStatus

```
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler({DataIntegrityViolationException.class})
public void conflict() {
    // could add the exception, response, etc. as method params
}
```

# Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - **Using MessageConverters**
  - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

# HttpMessageConverter

- Converts between HTTP request/response and object
- Various implementations registered by default when using `@EnableWebMvc` or `<mvc:annotation-driven/>`
  - XML (using JAXP Source or JAXB2 mapped object\*)
  - Feed data\*, i.e. Atom/RSS
  - Form-based data
  - JSON\*
  - Byte[], String, BufferedImage
- Define HandlerAdapter explicitly to register other HttpMessageConverters

\* Requires 3<sup>rd</sup> party libraries on classpath



You **must** use `@EnableWebMvc`/`<mvc:annotation-driven>` or register custom converters to have any HttpMessageConverters defined at all!

# @RequestBody

- Annotate method parameter with **@RequestBody**
  - Enables converters for *request* data
  - Right converter chosen automatically
    - Based on content type of request
    - Order could be mapped from XML with JAXB2 or from JSON with Jackson, for example

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

# @ResponseBody

- Use converters for *response* data by annotating method with **@ResponseBody**
  - Converter handles rendering to response
  - No ViewResolver and View involved anymore!

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

# Automatic Content Negotiation

- `HttpMessageConverter` selected automatically
  - For `@Request/ResponseBody` annotated parameters
  - Each converter has list of supported media types
- Allows *multiple* representations for a *single* controller method
  - Without affecting controller implementation
  - Alternative to using Views
- Flexible media-selection
  - Based on `Accept` header in HTTP request, or URL suffix, or URL format parameter

**See:** <http://spring.io/blog/2013/05/11/content-negotiation-using-spring-mvc>

# Content Negotiation Sample

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseBody @ResponseStatus(HttpStatus.OK) // 200
public Order getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}
```

GET /store/orders/123  
Host: www.myshop.com  
**Accept: application/xml, ...**  
...

HTTP/1.1 200 OK  
Date: ...  
Content-Length: 1456  
**Content-Type: application/xml**  
<order id="123">  
...  
</order>

GET /store/orders/123  
Host: www.myshop.com  
**Accept: application/json, ...**  
...

HTTP/1.1 200 OK  
Date: ...  
Content-Length: 756  
**Content-Type: application/json**  
{  
 "order": {"id": 123, "items": [ ... ], ... }  
}

# Using Views and Annotations - 1

- REST methods cannot return HTML, PDF, ...
  - No message converter
  - Views better for presentation-rich representations
- Need two methods on controller *for same URL*
  - One uses a converter, the other a view
  - Identify using produces attribute
- Similarly use consumes to differentiate methods
  - Such as RESTful POST from an HTML form submission

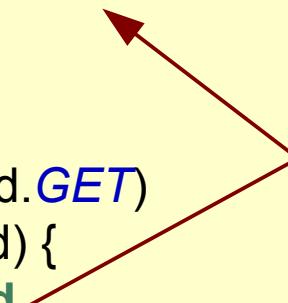
```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,  
    produces = {"application/json"})
```

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.POST,  
    consumes = {"application/json"})
```

# Using Views and Annotations - 2

- Identify RESTful method with produces
- Call RESTful method from View method
  - Implement all logic *once* in *RESTful* method

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,  
    produces = {"application/json", "application/xml"})  
@ResponseStatus(HttpStatus.OK) // 200  
public @ResponseBody Order getOrder(@PathVariable("id") long id) {  
    return orderRepository.findOrderById(id);  
}  
  
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)  
public String getOrder(Model model, @PathVariable("id") long id) {  
    model.addAttribute(getOrder(id)); // Call RESTful method  
    return "orderDetails";  
}
```



# Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - Using MessageConverters
  - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

# Retrieving a Representation: GET

```
GET /store/orders/123  
Host: www.myshop.com  
Accept: application/xml, ...  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 1456  
Content-Type: application/xml  
<order id="123">  
...  
</order>
```

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)  
@ResponseStatus(HttpStatus.OK) // 200: this is the default  
public @ResponseBody Order getOrder(@PathVariable("id") long id) {  
    return orderRepository.findOrderById(id);  
}
```

# Creating a new Resource: POST - 1

POST /store/orders/123/items

Host: [www.myshop.com](http://www.myshop.com)

Content-Type: application/xml

<item>

...

</item>

HTTP/1.1 201 Created

Date: ...

Content-Length: 0

Location: http://myshop.com/store/order/123/items/abc

```
@RequestMapping(value="/orders/{id}/items",
method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public void createItem(@PathVariable("id") long id,
                      @RequestBody Item newItem,
                      HttpServletRequest request,
                      HttpServletResponse response) {
    orderRepository.findOrderById(id).addItem(newItem); // adds id to Item
    response.addHeader("Location",
                      getLocationForChildResource(request, newItem.getId()));
}
```

*getLocationForChildResource was shown earlier*

# Creating a new Resource: POST - 2

- Simplify POST
  - Use @Value for request properties
  - Use HttpEntity instead of HttpServletResponse

```
@RequestMapping(value="/orders/{id}/items", method=RequestMethod.POST)
@ResponseBody(HttpStatus.CREATED) // 201
public HttpEntity<?> createItem(@PathVariable("id") long id,
    @RequestBody Item newItem,
    @Value("#{request.requestURL}") StringBuffer originalUrl) {
    orderRepository.findOrderById(id).addItem(newItem); // adds id to Item

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setLocation
        (getLocationForChildResource(originalUrl, newItem.getId()));
    return new HttpEntity<String>(responseHeaders);
}
```

Controller is a POJO again – *no* HttpServletResponse\* parameters

# Updating existing Resource: PUT

PUT /store/orders/123/items/abc

Host: [www.myshop.com](http://www.myshop.com)

Content-Type: application/xml

<item>

...

</item>

HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0

```
@RequestMapping(value="/orders/{orderId}/items/{itemId}",
    method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateItem(
    @PathVariable("orderId") long orderId,
    @PathVariable("itemId") String itemId
    @RequestBody Item item) {
    orderRepository.findOrderById(orderId).updateItem(itemId, item);
}
```

# Deleting a Resource: DELETE

```
DELETE /store/orders/123/items/abc  
Host: www.myshop.com  
...
```

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

```
@RequestMapping(value="/orders/{orderId}/items/{itemId}",  
                 method=RequestMethod.DELETE)  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
public void deleteItem(  
    @PathVariable("orderId") long orderId,  
    @PathVariable("itemId") String itemId {  
        orderRepository.findOrderById(orderId).deleteItem(itemId);  
    }
```

# @RestController Simplification

```
@Controller  
public class OrderController {  
    @RequestMapping(value="/orders/{id}", method=RequestMethod.GET)  
    public @ResponseBody Order getOrder(@PathVariable("id") long id) {  
        return orderRepository.findOrderById(id);  
    }  
    ...  
}
```

```
@RestController  
public class OrderController {  
    @RequestMapping(value="/orders/{id}", method=RequestMethod.GET)  
    public Order getOrder(@PathVariable("id") long id) {  
        return orderRepository.findOrderById(id);  
    }  
    ...  
}
```

No need for @ResponseBody on GET methods

# Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
- **RESTful clients with the RestTemplate**
- Conclusion

# RestTemplate

- Provides access to RESTful services
  - Supports URI templates, HttpMessageConverters and custom execute() with callbacks
  - Map or String... for vars, java.net.URI or String for URL

HTTP Method	RestTemplate Method
DELETE	delete(String url, String... urlVariables)
GET	getForObject(String url, Class<T> responseType, String... urlVariables)
HEAD	headForHeaders(String url, String... urlVariables)
OPTIONS	optionsForAllow(String url, String... urlVariables)
POST	postForLocation(String url, Object request, String... urlVariables)
	postForObject(String url, Object request, Class<T> responseType, String... urlVariables)
PUT	put(String url, Object request, String... urlVariables)

# Defining a RestTemplate

- Just call constructor in your code

```
RestTemplate template = new RestTemplate();
```

- Has default HttpMessageConverters
  - Same as on the server, depending on classpath
- Or use external configuration
  - To use Apache Commons HTTP Client, for example

```
<bean id="restTemplate" class="org.sfw.web.client.RestTemplate">
  <property name="requestFactory">
    <bean class="org.sfw.http.client.CommonsClientHttpRequestFactory"/>
  </property>
</bean>
```

# RestTemplate Usage Examples

```
RestTemplate template = new RestTemplate();
String uri = "http://example.com/store/orders/{id}/items";
{id} = 1
// GET all order items for an existing order with ID 1:
OrderItem[] items = template.getForObject(uri, OrderItem[].class, "1");

// POST to create a new item
OrderItem item = // create item object
URI itemLocation = template.postForLocation(uri, item, "1");

// PUT to update the item
item.setAmount(2);
template.put(itemLocation, item);

// DELETE to remove that item again
template.delete(itemLocation);
```



*Also supports `HttpEntity`, which makes adding headers to the HTTP request very easy as well*

# Topics in this Session

- REST introduction
- REST and Java
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - Using MessageConverters
  - Putting it all together
- RESTful clients with the RestTemplate
- Conclusion

# Conclusion

- REST is an architectural style that can be applied to HTTP-based applications
  - Useful for supporting diverse clients and building highly scalable systems
  - Java provides JAX-RS as standard specification
- Spring-MVC adds REST support using a familiar programming model
  - Extended by `@Request-/@ResponseBody`
- Use `RestTemplate` for accessing RESTful apps

# Lab

Restful applications with Spring MVC

# Spring JMS

## Simplifying Messaging Applications

Introducing `JmsTemplate` and Spring's Listener Container

# Topics in this Session

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

# Java Message Service (JMS)

- The JMS API provides an abstraction for accessing Message Oriented Middleware
  - Avoid vendor lock-in
  - Increase portability
- JMS does *not* enable different MOM vendors to communicate
  - Need a bridge (expensive)
  - Or use AMQP (standard msg protocol, like SMTP)
    - See RabbitMQ

# JMS Core Components

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

# JMS Message Types

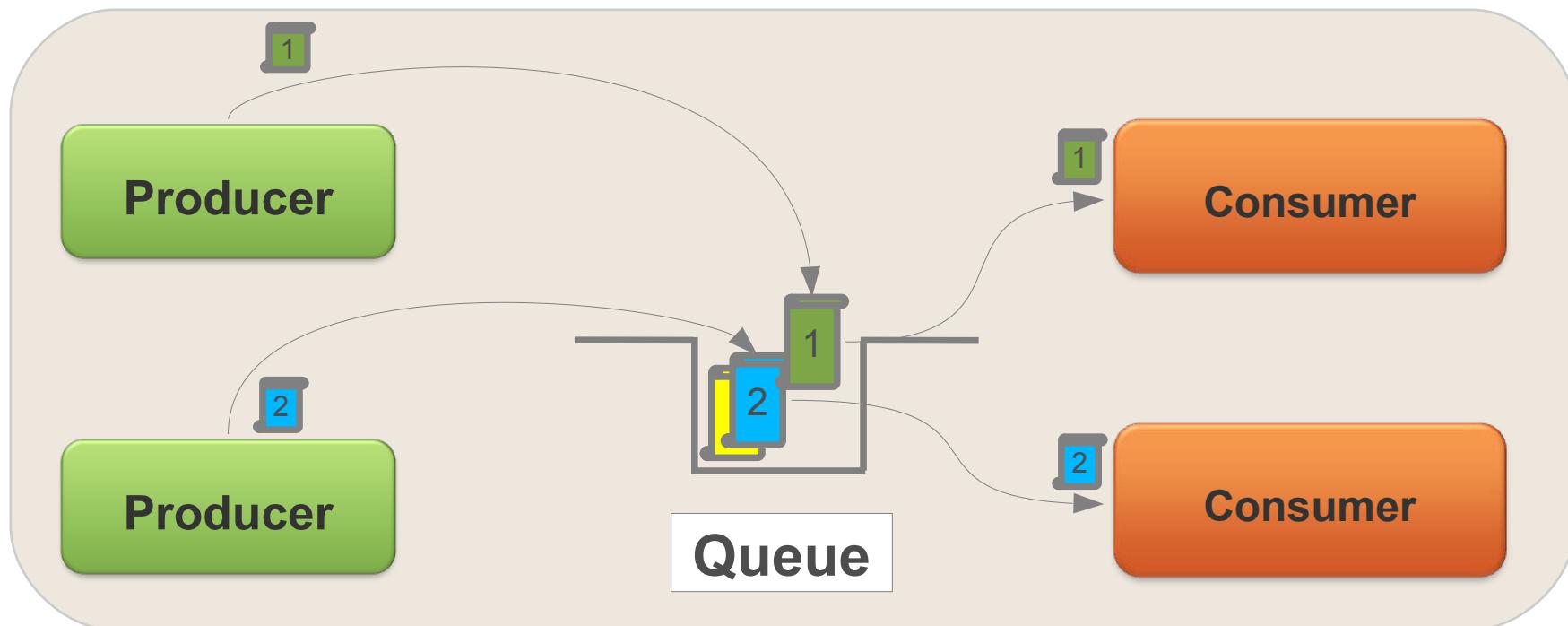
- Implementations of the Message interface
  - TextMessage
  - ObjectMessage
  - MapMessage
  - BytesMessage
  - StreamMessage

# JMS Destination Types

- Implementations of the Destination interface
  - Queue
    - Point-to-point messaging
  - Topic
    - Publish/subscribe messaging
- Both support *multiple* producers and consumers
  - Messages are different
  - Let's take a closer look ...

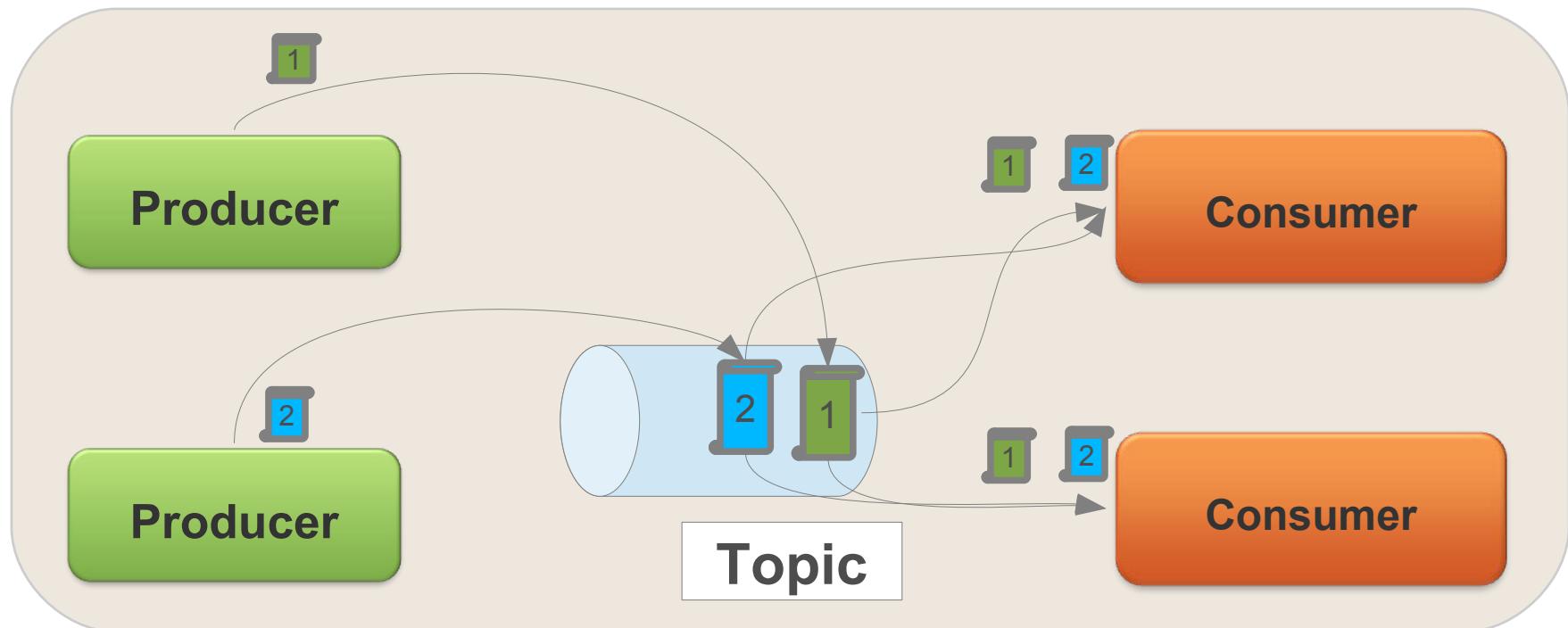
# JMS Queues: Point-to-point

1. Message sent to queue
2. Message queued
3. Message consumed by *single* consumer



# JMS Topics: Publish-subscribe

1. Message sent to topic
2. Message optionally stored
3. Message distributed to *all* subscribers



# The JMS Connection

- A JMS Connection is obtained from a factory

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

- Typical enterprise application:
  - ConnectionFactory is a managed resource bound to JNDI

```
Properties env = new Properties();
// provide JNDI environment properties
Context ctx = new InitialContext(env);
ConnectionFactory connectionFactory =
    (ConnectionFactory) ctx.lookup("connFactory");
```

# The JMS Session

- A Session is created from the Connection
  - Represents a unit-of-work
  - Provides transactional capability

```
Session session = conn.createSession(  
    boolean transacted, int acknowledgeMode);
```

```
// use session  
if (everythingOkay) {  
    session.commit();  
} else {  
    session.rollback();  
}
```

# Creating Messages

- The Session is responsible for the creation of various JMS Message types

```
session.createTextMessage("Some Message Content");
```

```
session.createObjectMessage(someSerializableObject);
```

```
MapMessage message = session.createMapMessage();
message.setInt("someKey", 123);
```

```
BytesMessage message = session.createBytesMessage();
message.writeBytes(someByteArray);
```

# Producers and Consumers

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

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

# Topics in this Session

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

# JMS Providers

- Most providers of Message Oriented Middleware (MoM) support JMS
  - WebSphere MQ, Tibco EMS, Oracle EMS, JBoss AP, SwiftMQ, etc.
  - Some are Open Source, some commercial
  - Some are implemented in Java themselves
- The lab for this module uses Apache ActiveMQ

# Apache ActiveMQ

- Open source message broker written in Java
- Supports JMS and many other APIs
  - Including non-Java clients!
- Can be used stand-alone in production environment
  - 'activemq' script in download starts with default config
- Can also be used embedded in an application
  - Configured through ActiveMQ or Spring configuration
  - What we use in the labs

# Apache ActiveMQ Features

Support for:

- Many cross language clients & transport protocols
  - Incl. excellent Spring integration
- Flexible & powerful deployment configuration
  - Clustering incl. load-balancing & failover, ...
- Advanced messaging features
  - Message groups, virtual & composite destinations, wildcards, etc.
- Enterprise Integration Patterns when combined with Spring Integration or Apache Camel
  - from the book by Gregor Hohpe & Bobby Woolf

# Topics in this Session

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

# Configuring JMS Resources with Spring

- Spring enables decoupling of your application code from the underlying infrastructure
  - Container provides the resources
  - Application is simply coded against the API
- Provides deployment flexibility
  - use a standalone JMS provider
  - use an application server to manage JMS resources



See: **Spring Framework Reference – Using Spring JMS**  
<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jms>

# Configuring a ConnectionFactory

- ConnectionFactory may be standalone

```
@Bean  
public ConnectionFactory connectionFactory() {  
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();  
    cf.setBrokerURL("tcp://localhost:60606");  
    return cf;  
}
```

- Or retrieved from JNDI

```
@Bean  
public ConnectionFactory connectionFactory() throws Exception {  
    Context ctx = new InitialContext();  
    return (ConnectionFactory) ctx.lookup("jms/ConnectionFactory");  
}
```

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

# Configuring Destinations

- Destinations may be standalone

```
@Bean  
public Destination orderQueue() {  
    return new ActiveMQQueue( "order.queue" );  
}
```

- Or retrieved from JNDI

```
@Bean  
public Destination connectionFactory() throws Exception {  
    Context ctx = new InitialContext();  
    return (Destination) ctx.lookup("jms/OrderQueue");  
}
```

```
<jee:jndi-lookup id="orderQueue" jndi-name="jms/OrderQueue"/>
```

# Topics in this Session

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

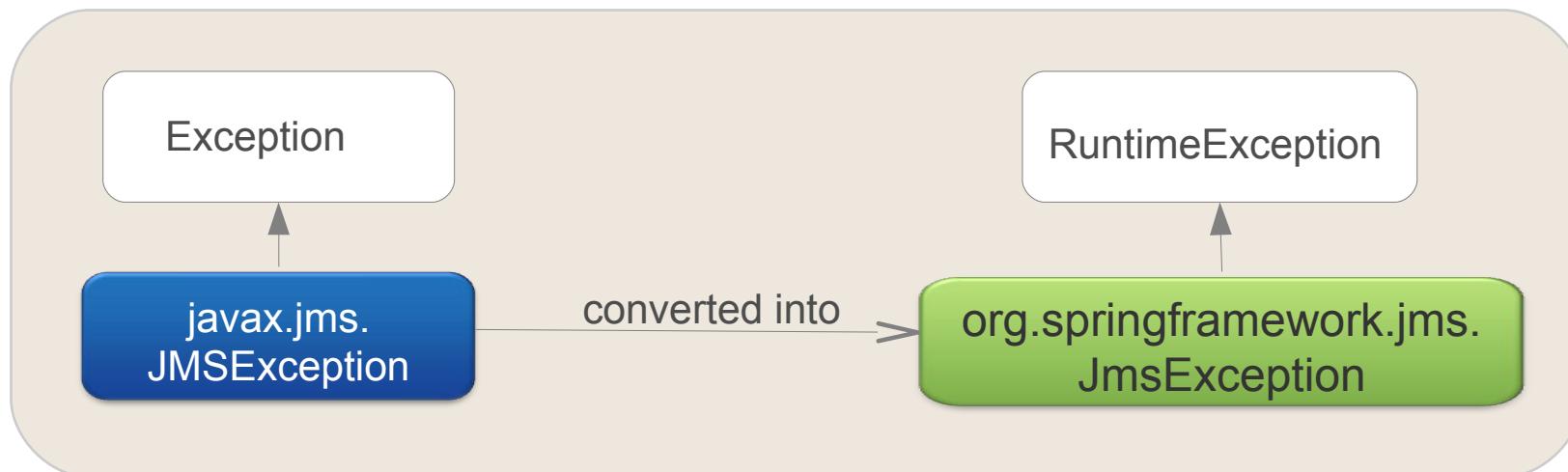
# Spring's JmsTemplate

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

**NOTE:** The *AmqpTemplate* (used with RabbitMQ) has an almost identical API to the *JmsTemplate* – they offer similar abstractions over very different products

# Exception Handling

- Exceptions in JMS are checked by default
- JmsTemplate converts checked exceptions to runtime equivalents

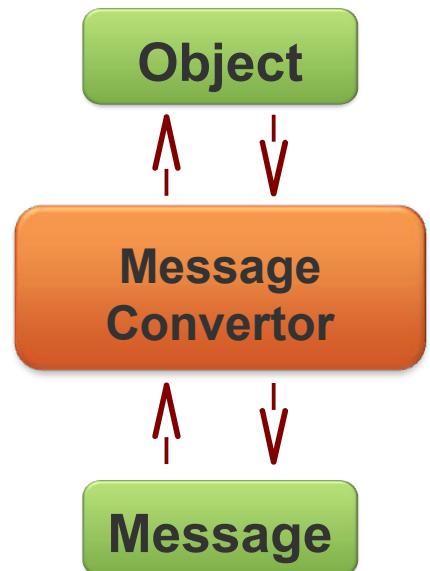


# JmsTemplate Strategies

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

# MessageConverter

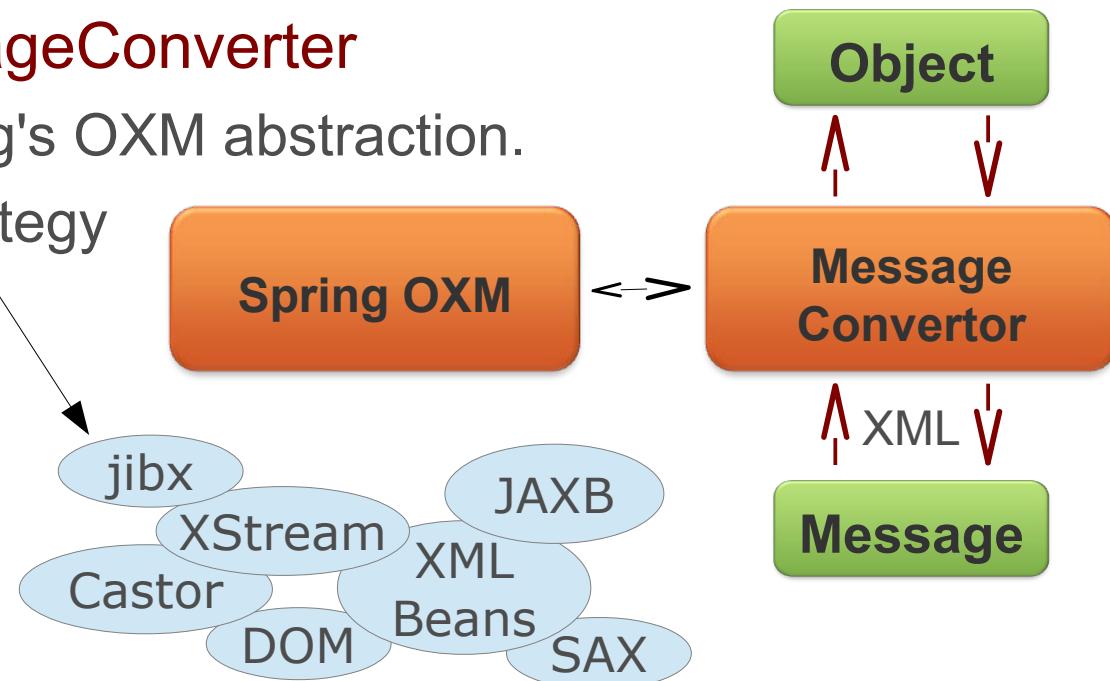
- The JmsTemplate uses a **MessageConverter** to convert between objects and messages
  - You only send and receive objects
- The default **SimpleMessageConverter** handles basic types
  - String to TextMessage
  - Map to MapMessage
  - byte[] to BytesMessage
  - Serializable to ObjectMessage



**NOTE:** It is possible to implement custom converters by implementing the *MessageConverter* interface

# XML MessageConverter

- XML is a common message payload
  - ...but there is no “`XmlMessage`” in JMS
  - Use `TextMessage` instead.
- **MarshallingMessageConverter**
  - Plugs into Spring's OXM abstraction.
  - You choose strategy



# MarshallingMessageConverter Example

```
@Bean public JmsTemplate jmsTemplate () {  
    JmsTemplate template = new JmsTemplate( connectionFactory );  
    template.setMessageConverter ( msgConverter() );  
    return template;  
}  
  
@Bean public MessageConverter msgConverter() {  
    MessageConverter converter = new MarshallingMessageConverter();  
    converter.setMarshaller ( marshaller() );  
    return converter;  
}  
  
@Bean public Marshaller marshaller() {  
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();  
    marshaller.setContextPath ( "example.app.schema" );  
    return marshaller;  
}
```

JAXB2 Illustrated here,  
other strategies  
available.

# DestinationResolver

- Convenient to use destination names at runtime
- DynamicDestinationResolver used by default
  - Resolves topic and queue names
  - Not their Spring bean names
- JndiDestinationResolver also available



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

publish-subscribe?  
*true* q Topic  
*false* q Queue

# JmsTemplate configuration

- *Must* provide reference to ConnectionFactory
  - via either constructor or setter injection
- Optionally provide other facilities
  - setMessageConverter
  - setDestinationResolver
  - setDefaultDestination or setDefaultDestinationName

```
@Bean  
public JmsTemplate jmsTemplate () {  
    JmsTemplate template = new JmsTemplate( connectionFactory );  
    template.setMessageConverter ( ... );  
    template.setDestinationResolver ( ... );  
    return template;  
}
```

# Topics in this Session

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

# Sending Messages

- The template provides options
  - Simple methods to send a JMS message
  - One line methods that leverage the template's MessageConverter
  - Callback-accepting methods that reveal more of the JMS API
- Use the simplest option for the task at hand

# Sending POJO

- A message can be sent in one single line

```
public class JmsOrderManager implements OrderManager {  
    @Autowired JmsTemplate jmsTemplate;  
    @Autowired Destination orderQueue;  
  
    public void placeOrder(Order order) {  
        String stringMessage = "New order " + order.getNumber();  
        jmsTemplate.convertAndSend("message.queue", stringMessage );  
        // use destination resolver and message converter  
  
        jmsTemplate.convertAndSend(orderQueue, order); // use message converter  
  
        jmsTemplate.convertAndSend(order); // use converter and default destination  
    }  
}
```

No @Qualifier so Destination is wired by *name*

# Sending JMS Messages

- Useful when you need to access JMS API
  - eg. set expiration, redelivery mode, reply-to ...

```
public void sendMessage(final String msg) {
```

Lambda syntax

```
this.jmsTemplate.send( session) -> {  
    TextMessage message = session.createTextMessage(msg);  
    message.setJMSExpiration(2000); // 2 seconds  
    return message;  
});  
}
```

```
public interface MessageCreator {  
    public Message createMessage(Session session)  
        throws JMSException;  
}
```

# Topics in this Session

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

# Synchronous Message Reception

- JmsTemplate can also receive messages
  - but methods are blocking (synchronous)
  - optional timeout: **setReceiveTimeout()**

```
public void receiveMessages() {  
  
    // use message converter and destination resolver  
    String s = (String) jmsTemplate.receiveAndConvert("message.queue");  
    // use message converter  
    Order order1 = (Order) jmsTemplate.receiveAndConvert(orderQueue);  
  
    // handle JMS native message from default destination  
    ObjectMessage orderMessage = (ObjectMessage) jmsTemplate.receive();  
    Order order2 = (Order) orderMessage.getObject();  
}
```

# Synchronous Message Exchange

- JmsTemplate also implements a request/reply pattern
  - Sending a message and blocking until a reply has been received (also uses receiveTimeout)
  - Manage a temporary reply queue automatically by default

```
public void processMessage(String msg) {  
  
    Message reply = jmsTemplate.sendAndReceive("message.queue",  
        (session) -> {  
            return session.createTextMessage(msg);  
        });  
    // handle reply  
}
```

# Spring's MessageListener Containers

- Spring provides containers for asynchronous JMS reception
  - *SimpleMessageListenerContainer*
    - Uses plain JMS client API
    - Creates a fixed number of Sessions
  - *DefaultMessageListenerContainer*
    - Adds transactional capability
- Many configuration options available for each container type

# Quick Start

## Steps for Asynchronous Message Handling

- 1) Define POJO / Bean to process Message
- 2) Define JmsListenerContainerFactory / Enable Annotations

# Step 1 – Define POJO / Bean to Process Message

- Define a POJO to process message
  - Note: No references to JMS

```
public class OrderServiceImpl {  
    @JmsListener(destination="queue.order")  
    @SendTo("queue.confirmation")  
    public OrderConfirmation order(Order o) { ... }  
}
```

- Define as a Spring bean using XML, JavaConfig, or annotations as preferred
- **@JmsListener** enables a JMS message consumer for the method
- **@SendTo** defines response destination (optional)

# Step 2 – Define JmsListenerContainerFactory to use

- JmsListenerContainerFactory: separates JMS API from your POJO:

```
@Configuration @EnableJms  
public class MyConfiguration {
```

```
    @Bean
```

```
    public DefaultJmsListenerContainerFactory  
        jmsListenerContainerFactory () {
```

```
        DefaultJmsListenerContainerFactory cf =  
            new DefaultJmsListenerContainerFactory();  
        cf.setConnectionFactory(connectionFactory());
```

```
        ...
```

```
        return cf;
```

Enable annotations

Default container name

Set ConnectionFactory

Many settings available:  
TransactionManager, TaskExecutor, ContainerType ...

# @JmsListener features

- By default, the container with name **jmsListenerContainerFactory** is used

```
public class OrderServiceImpl {  
    @JmsListener(containerFactory="myFactory",  
        destination="orderConfirmation")  
    public void process(OrderConfirmation o) { ... }  
}
```

- Can also set a custom concurrency or a selector

```
public class OrderServiceImpl {  
    @JmsListener(selector="type = 'Order'",  
        concurrency="2-10", destination = "order")  
    public OrderConfirmation order(Order o) { ... }  
}
```

# Step 2: JMS XML Namespace support

- Equivalent Capabilities
  - The **containerId** attribute exposes the configuration of the container with that name
  - Same configuration options available
    - task execution strategy, concurrency, container type, transaction manager and more

```
<jms:annotation-driven/>

<jms:listener-container
    containerId="jmsMessageContainerFactory"
    connection-factory="myConnectionFactory"/>

<bean id="orderService" class="org.acme.OrderService"/>
```

# 100% XML Equivalent

- Use jms:listener-container with embedded jms:listeners
  - Allows multiple listeners per container
  - Same configuration options available

```
<jms:listener-container connection-factory="myConnectionFactory">
    <jms:listener destination="order.queue"
        ref="orderService"
        method="order"
        response-destination="confirmation.queue" />
    <jms:listener destination="confirmation.queue"
        ref="orderService"
        method="confirm" />
</jms:listener-container>

<bean id="orderService" class="org.acme.OrderService"/>
```

# Messaging: Pros and Cons

- Advantages
  - Resilience, guaranteed delivery
  - Asynchronous support
  - Application freed from messaging concerns
  - Interoperable – languages, environments
- Disadvantages
  - Requires additional third-party software
    - Can be expensive to install and maintain
  - More complex to use – *but not with JmsTemplate!*

*Spring Enterprise* – 4 day course on application integration

# Lab

Sending and Receiving Messages in  
a Spring Application

# Topics in this Session

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

# Advanced Option: CachingConnectionFactory

- JmsTemplate aggressively closes and reopens resources like Sessions and Connections
  - Normally these are cached by connection factory
  - Without caching can cause lots of overhead
    - Resulting in poor performance
- Use our CachingConnectionFactory to add caching within the application if needed

```
<bean id="connectionFactory"
      class="org.springframework.jms.connection.CachingConnectionFactory">
    <property name="targetConnectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://embedded?broker.persistent=false"/>
      </bean>
    </property>
  </bean>
```

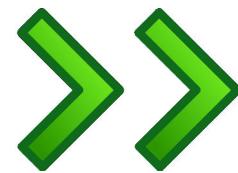
# Finishing Up

Course Completed

What's Next?

# What's Next

- Congratulations, we've finished the course
- What to do next?
  - Certification
  - Other courses
  - Resources
  - Evaluation
- Check-out optional sections on Remoting and SOAP web-services



# Certification



- Computer-based exam
  - 50 multiple-choice questions
  - 90 minutes
  - Passing score: 76% (38 questions answered successfully)
- Preparation
  - Review all the slides
  - Redo the labs
  - Study Guide:  
[http://www.pivotal.io/files/pdfs/pivotal\\_studyguide\\_corespring.docx](http://www.pivotal.io/files/pdfs/pivotal_studyguide_corespring.docx)

# Certification: Questions

## Sample question

- Statements
  - a. An application context holds Spring beans
  - b. An application context manages bean scope
  - c. Spring provides many types of application context
- Pick the correct response:
  - 1. Only a. is correct
  - 2. Both a. and c. are correct
  - 3. All are correct
  - 4. None are correct

# Certification: Logistics

- Where?
  - At any Pearson VUE Test Center
  - Most large or medium-sized cities
    - See <http://www.pearsonvue.com/vtclocator>
- How?
  - At the end of the class, you will receive a certification voucher by email
  - Make an appointment
  - Give them the voucher when you take the test
- For any further inquiry, you can write to
  - [education@pivotal.io](mailto:education@pivotal.io)

# Other courses



- Many courses available
  - Web Applications with Spring
  - Enterprise Spring
  - JPA with Spring
  - What's New in Spring
  - Groovy and Grails
  - Cloud Foundry
  - Hadoop, Gemfire, Rabbit MQ ...
- See <http://www.pivotal.io/training>

# Spring Web

- Four-day workshop
- Making the most of Spring in the web layer
  - Spring MVC
  - Spring Web Flow
  - REST using MVC and AJAX
  - Security of Web applications
  - Mock MVC testing framework
- Spring Web Application Developer certification

# Spring Enterprise

- Building loosely coupled, event-driven architectures
  - Separate processing, communications & integration
  - Formerly Enterprise Integration & Web Services
- Four day course covering
  - Concurrency
  - Advanced transaction management
  - SOAP Web Services with Spring WS
  - REST Web Services with Spring MVC
  - Spring Batch
  - Spring Integration

# JPA with Spring

- Three day course covering
  - Using JPA with Spring and Spring Transactions
  - Implement inheritance and relationships with JPA
  - Discover how JPA manages objects
  - Go more in depth on locking with JPA
  - Advanced features such as interceptors, caching and batch updates

# Developing Applications with Cloud Foundry

- Three day course covering
  - Application deployment to Cloud Foundry
  - Cloud Foundry Concepts
  - Deployment using cf tool or an IDE
  - Accessing and defining Services
  - Using and customizing Buildpacks
  - Design considerations: “12 Factor”
  - JVM Application specifics, using Spring Cloud



# Pivotal Support Offerings

- Global organization provides 24x7 support
  - How to Register: <http://tinyurl.com/piv-support>
- Premium and Developer support offerings:
  - <http://www.pivotal.io/support/offering>
  - <http://www.pivotal.io/support/oss>
  - Both Pivotal App Suite *and* Open Source products
- Support Portal: <https://support.pivotal.io>
  - Community forums, Knowledge Base, Product documents



# Pivotal Consulting

- Custom consulting engagement?
  - Contact us to arrange it
    - <http://www.pivotal.io/contact/spring-support>
    - Even if you don't have a support contract!
- Pivotal Labs
  - Agile development experts
  - Assist with design, development and product management
    - <http://www.pivotal.io/agile>
    - <http://pivotallabs.com>



# Resources

- The Spring reference documentation
  - <http://spring.io/docs>
  - Already 800+ pages!
- The official technical blog
  - <http://spring.io/blog>
- Stack Overflow – Active Spring Forums
  - <http://stackoverflow.com>

# Resources (2)

- You can register issues on our Jira repository
  - <https://jira.spring.io>
- The source code is available here
  - <https://github.com/spring-projects/spring-framework>

# Thank You!

We hope you enjoyed the course

Please fill out the evaluation form

<http://tinyurl.com/mylearneval>



# Object Relational Mapping

## Using OR Mapping in the Enterprise

### Fundamental Concepts and Concerns

# Topics in this session

- **The Object/Relational mismatch**
- ORM in context
- Benefits of O/R Mapping

# The Object/Relational Mismatch (1)

- A domain object model is designed to serve the needs of the application
  - Organize data into abstract concepts that prove useful to solving the domain problem
  - Encapsulate behavior specific to the application
  - Under the control of the application developer

# The Object/Relational Mismatch (2)

- Relational models relate business data and are typically driven by other factors:
  - Performance
  - Space
- Furthermore, a relational database schema often:
  - Predates the application
  - Is shared with other applications
  - Is managed by a separate DBA group

# Object/Relational Mapping

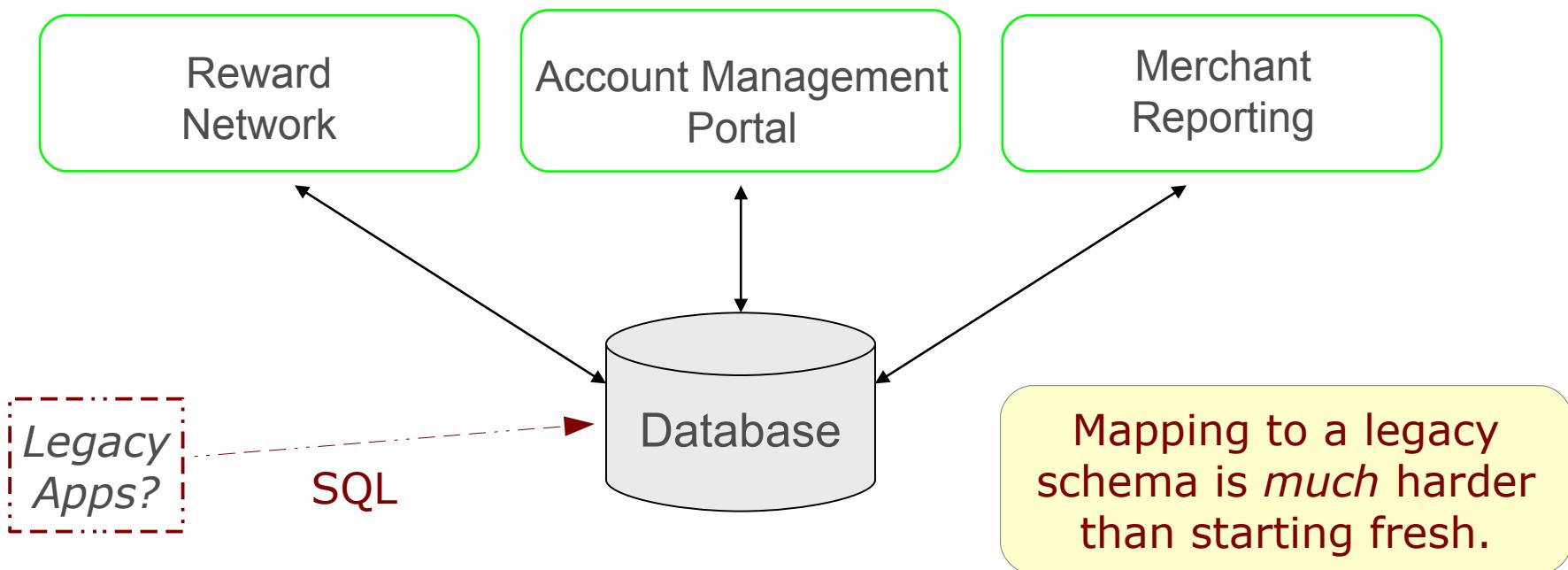
- Object/Relational Mapping (ORM) engines exist to mitigate the mismatch
- Spring supports all of the major ones:
  - Hibernate
  - EclipseLink
  - Other JPA (Java Persistence API) implementations, such as OpenJPA
- This session will focus on Hibernate

# Topics in this session

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

# ORM in context

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



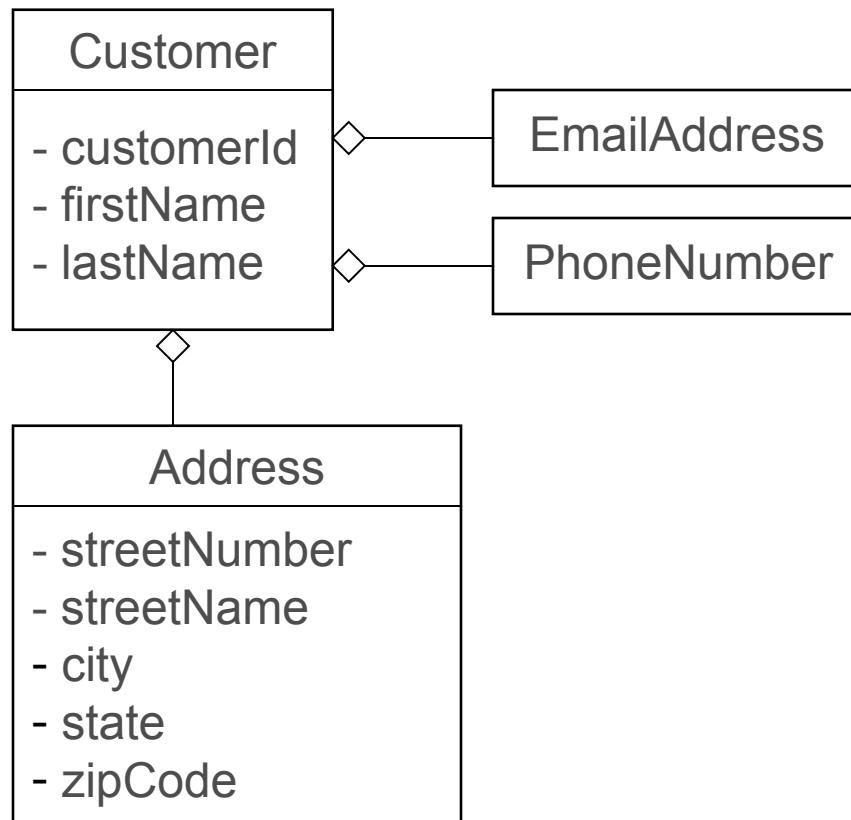
# O/R Mismatch: Granularity (1)

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

# O/R Mismatch: Granularity (2)

*just one example...*

**Domain Model in Java**



**Table in Database**

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

# O/R Mismatch: Identity (1)

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

# O/R Mismatch: Identity (2)

- When working with persistent Objects, the identity problem leads to difficult challenges
  - Two different Java objects may correspond to the same relational row
  - But Java says they are *not* equal
- Some of the challenges:
  - Implement equals() to accommodate this scenario
  - Determine when to update and when to insert
  - Avoid duplication when adding to a Collection

# O/R Mismatch: Inheritance and Associations (1)

- In an object-oriented language:
  - *IS-A* relations are modeled with inheritance
  - *HAS-A* relations are modeled with composition
- In a database schema, relations are limited to what can be expressed by *foreign keys*

# O/R Mismatch: Inheritance and Associations (2)

- Bi-directional associations are common in a domain model (e.g. Parent-Child)
  - This can be modeled naturally in each Object
- In a database:
  - One side (parent) provides a primary-key
  - Other side (child) provides a foreign-key reference
- For many-to-many associations, the database schema requires a *join table*

# Topics in this session

- The Object/Relational Mismatch
- ORM in Context
- **Benefits of O/R Mapping**

# Benefits of ORM

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

# Object Query Language

- When working with domain objects, it is more natural to query based on objects.
  - Query with SQL:

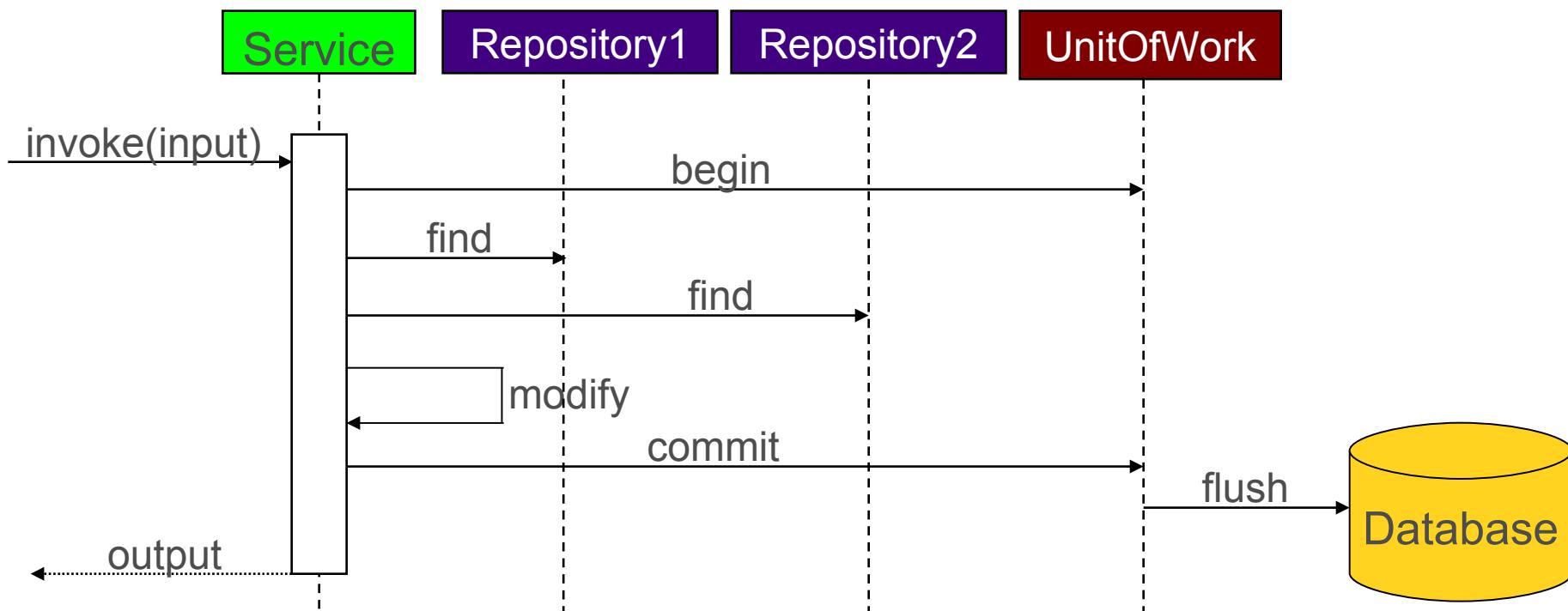
```
SELECT c.first_name, c.last_name, a.city, ...
    FROM customer c, customer_address ca, address a
 WHERE ca.customer_id = c.id
   AND ca.address_id = a.id
   AND a.zip_code = 12345
```

- Query with object properties and associations:

```
SELECT c FROM Customer c WHERE c.address.zipCode = 12345
```

# Automatic Change Detection

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



# Persistence by Reachability

- When a persistent object is being managed, other associated objects may become managed transparently:

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

```
LineItem item = new LineItem(..);  
order.addLineItem(item);  
// item is now a managed object – reachable from order
```

# (Un)Persistence by Reachability

## = Make Transient

- The same concept applies for deletion:

```
Order order = orderRepository.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  
}
```

Item becomes transient

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

Order and all its  
items now transient

# Caching

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

# Summary

- Managing persistent objects is hard
  - Especially if caching is involved
  - Especially on a shared, legacy schema with existing applications
- The ORM overcomes *some* of these problems
  - Automatic change detection, queries, caching
  - Ideal if your application *owns* its database
  - It is *not* a magic-bullet
    - JDBC may still be better for some tables/queries
    - True distributed cache coherency is *very* hard
    - *Design* for it and *test* performance

# ORM With Spring and Hibernate

Configuring Hibernate to work with Spring

Equivalent to JPA section but using Hibernate  
Session API

# Topics in this session

- **Introduction to Hibernate**
  - Mapping
  - Querying
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- Exception Mapping

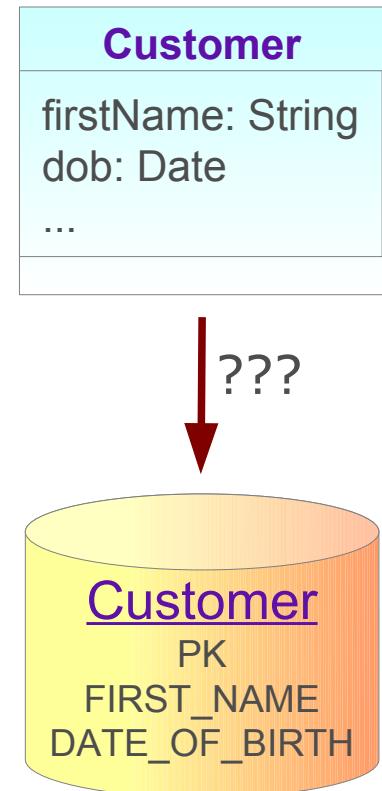
# Introduction to Hibernate

- A **SessionFactory** is a thread-safe, shareable object that represents a *single* data source
  - Provides access to a transactional Session
  - Ideal candidate for a singleton Spring bean
- Hibernate's **Session** is a stateful object representing a unit-of-work
  - Often corresponds at a higher-level to a JDBC Connection
  - Manages persistent objects within the unit-of-work
  - Acts as a transaction-scoped cache (1LC)



# Hibernate Mapping

- Hibernate requires metadata
  - for mapping classes to database tables
  - and their properties to columns
- Metadata can be annotations or XML
  - This session will present JPA annotations
  - XML shown in the appendix



# Annotations support in Hibernate

- Hibernate supports all JPA 2 annotations
  - Since Hibernate 3.5
  - javax.persistence.\*
  - Suitable for most needs
- Hibernate Specific Extensions
  - In *addition* to JPA annotations
    - For behavior not supported by JPA
    - Performance enhancements specific to Hibernate
  - Less important since JPA 2



# What can you Annotate?

- Classes
  - Applies to the entire class (such as table properties)
- Data-members
  - Typically mapped to a column
  - By default, *all* data-members treated as persistent
    - Mappings will be defaulted
    - Unless annotated with `@Transient` (non-persistent)
  - All data-members accessed directly
    - No accessors (getters/setters) needed
    - Any accessors will not be used
    - This is called “field” access

# Mapping simple data-members: Field Access

```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    @Id  
    @Column (name="cust_id")  
    private Long id;
```

Mark as an *entity*  
Optionally override  
*table name*

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

Mark *id-field* (primary  
key)

```
    @Transient  
    private User currentUser;
```

Optionally override  
*column names*

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

Not stored in database

```
}
```

Setters and *not* used by  
Hibernate

Only *@Entity* and *@Id* are mandatory

...

# Mapping simple Properties

## - Traditional Approach

**Must** place `@Id` on the *getter* method

Other annotations now also placed on *getter* methods



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

```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    private Long id;  
    private String firstName;  
  
    @Id  
    @Column (name="cust_id")  
    public String getId()  
    { return this.id; }  
  
    @Column (name="first_name")  
    public String getFirstName()  
    { return this.firstName; }  
  
    public void setFirstName(String name)  
    { this.firstName = name; }  
}
```

# Mapping collections with annotations

```
@Entity  
@Table(name= "T_CUSTOMER")  
public class Customer {  
    @Id  
    @Column (name="cust_id")  
    private Long id;
```

```
    @OneToMany(cascade=CascadeType.ALL)  
    @JoinColumn (name="cid")  
    private Set<Address> addresses;  
    ...
```

Propagate all operations  
to the underlying objects

Foreign key in  
Address table

JoinTable also supported, more  
commonly used in  
@ManyToMany associations

# Accessing Persistent Data

- Hibernate's key class is the **Session**
  - Provides methods to manipulate entities
    - persist, delete, get ...
  - Create queries using Hibernate Query Language
  - Manages transactions
  - JPA defines a similar class: *EntityManager*
- Sessions more popular than EntityManagers
  - Already widely used before JPA
  - Common to write hybrid applications
    - JPA annotations, Hibernate API and HQL
    - But remember: it is not a JPA application



# Hibernate Querying

- Hibernate provides several options for accessing data
  - Retrieve an object by primary key
  - Query for objects with the Hibernate Query Language (HQL)
  - Query for objects using Criteria Queries
  - Execute standard SQL

# Hibernate Querying: by Primary Key

- To retrieve an object by its database identifier simply call `get(..)` on the Session

```
Long custId = new Long(123);
Customer customer = (Customer) session.get(Customer.class, custId);
```

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

# Hibernate Querying: HQL

- To query for objects based on properties or associations use HQL
  - Pre Java 5 API, not type aware

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

No generics

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

Must cast

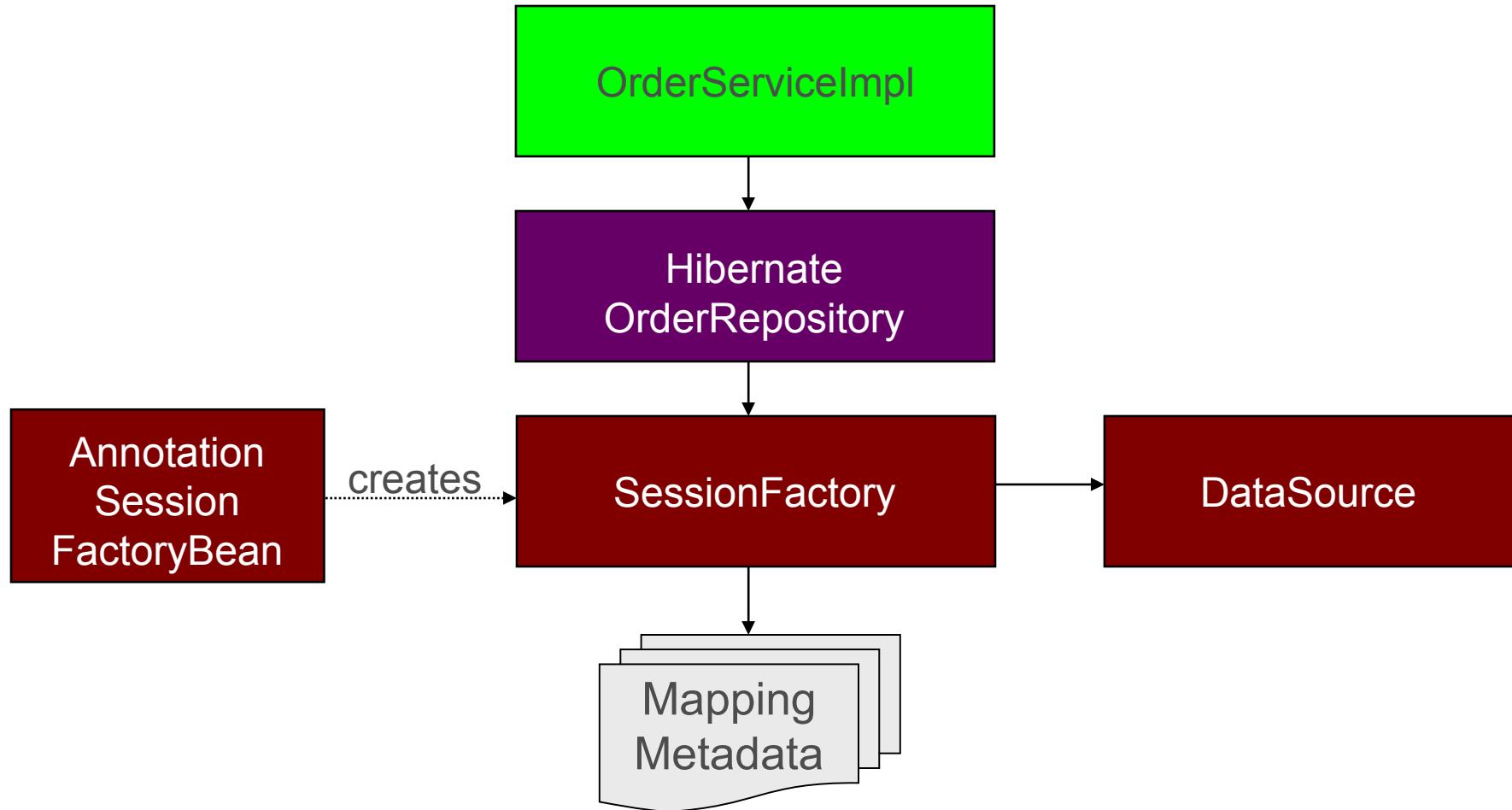
# Topics in this session

- Introduction to Hibernate
- **Configuring a Hibernate SessionFactory**
- Implementing Native Hibernate DAOs
- Exception Mapping

# Configuring a SessionFactory (1)

- Hibernate implementations of data access code require access to the SessionFactory
- The SessionFactory requires
  - DataSource (local or container-managed)
  - Mapping metadata
- Spring provides a FactoryBean for configuring a shareable SessionFactory
  - Supports Hibernate V4 from Spring 3
  - There are different versions in hibernate2, hibernate3, hibernate4 packages

# Configuring a SessionFactory (2)



# SessionFactory and Annotated Classes

```
<bean id="orderRepository" class="example.HibernateOrderRepository">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Spring 3.1

```
<bean id="sessionFactory" class="org.springframework.orm.
    hibernate4.annotation.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>

    <property name="annotatedClasses">
        <list>
            <value>example.Customer</value>
            <value>example.Address</value>
        </list>
    </property>
</bean>
```

Entities listed one by one  
- wildcards *not* supported

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

Prior to Spring 3.1 use the AnnotationSessionFactoryBean

# SessionFactory and Scanned Packages

Spring 3.1

- Or use `packagesToScan` attribute
  - Wildcards are supported
  - Also scans all sub-packages

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate4.  
annotation.LocalSessionFactoryBean">  
    <property name="dataSource" ref="dataSource"/>  
    <property name="packagesToScan">  
        <list>  
            <value>com/springsource/*/entity</value>  
        </list>  
    </property>  
</bean>  
  
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/orders"/>
```

# Topics in this session

- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- **Implementing Native Hibernate DAOs**
- Exception Mapping

# Implementing Native Hibernate DAOs (1)

- Since Hibernate 3.1+
  - Hibernate provides hooks so Spring can manage transactions and Sessions in a transparent fashion
  - Use AOP for transparent exception translation to Spring's **DataAccessException** hierarchy
- No dependency on Spring in your DAO implementations
  - No longer require **HibernateTemplate** (obsolete)

# Spring-Managed Transactions and Sessions (1)

- Transparently participate in Spring-driven transactions
  - use one of Spring's Factory Beans to build the **SessionFactory**
- Provide it to the data access code
  - dependency injection
- Define a transaction manager
  - **HibernateTransactionManager**
  - **JtaTransactionManager**

# Spring-managed Transactions and Sessions (2)

- The code – with no Spring dependencies

```
public class HibernateOrderRepository implements OrderRepository {  
    private SessionFactory sessionFactory;  
  
    public HibernateOrderRepository(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
    ...  
    public Order findOrderId(long orderId) {  
        // use the Spring-managed Session  
        Session session = this.sessionFactory.getCurrentSession();  
        return (Order) session.get(Order.class, orderId);  
    }  
}
```

The diagram illustrates two annotations in the code:

- dependency injection**: An annotation pointing to the constructor `HibernateOrderRepository(SessionFactory sessionFactory)`.
- use existing session**: An annotation pointing to the line `Session session = this.sessionFactory.getCurrentSession();`.

# Spring-managed Transactions and Sessions (3)

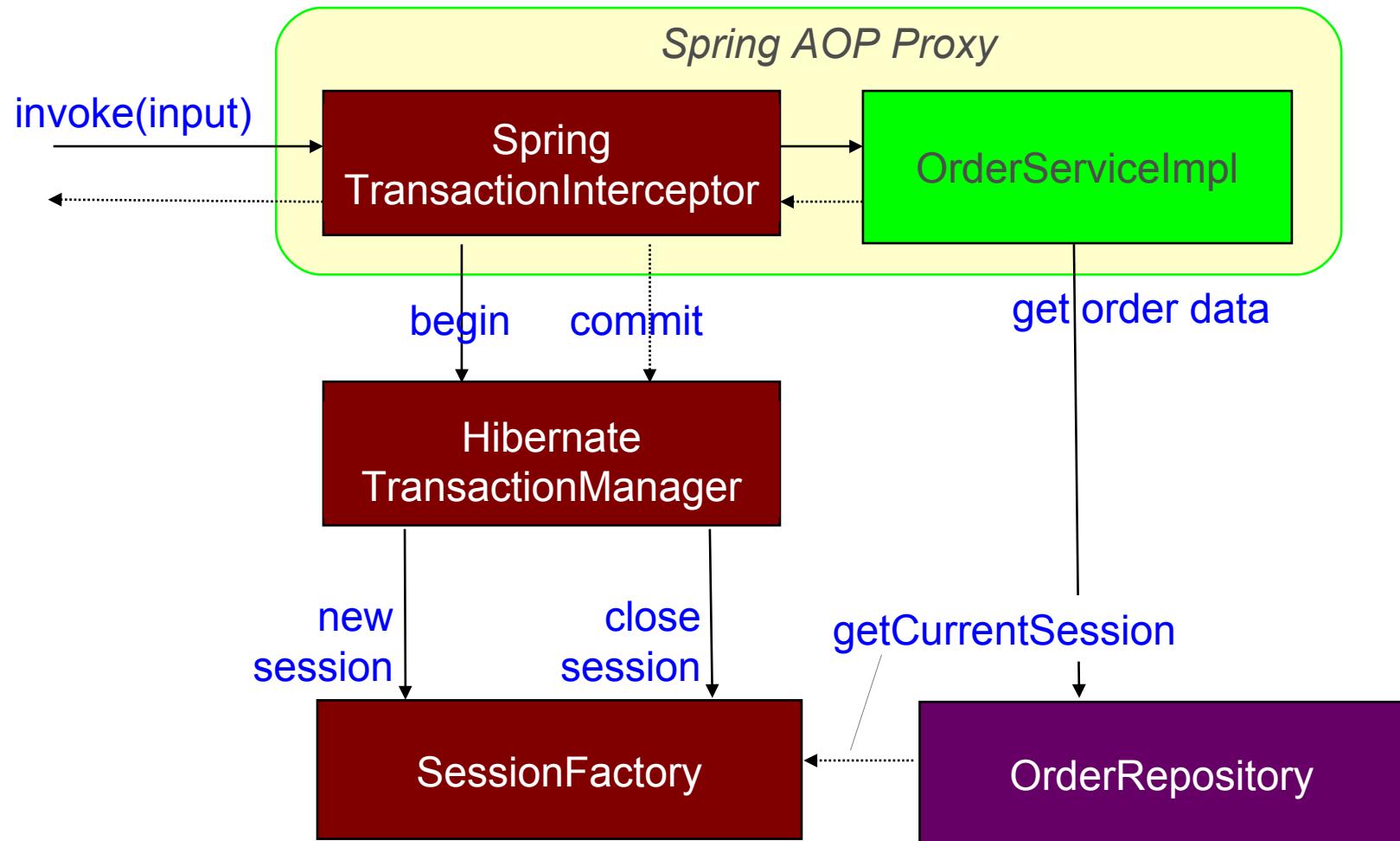
- The configuration

```
<beans>
  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.
      annotation.AnnotationSessionFactoryBean">
    ...
  </bean>
  <bean id="orderRepository" class="HibernateOrderRepository">
    <constructor-arg ref="sessionFactory"/>
  </bean>
  <bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>
</beans>
```

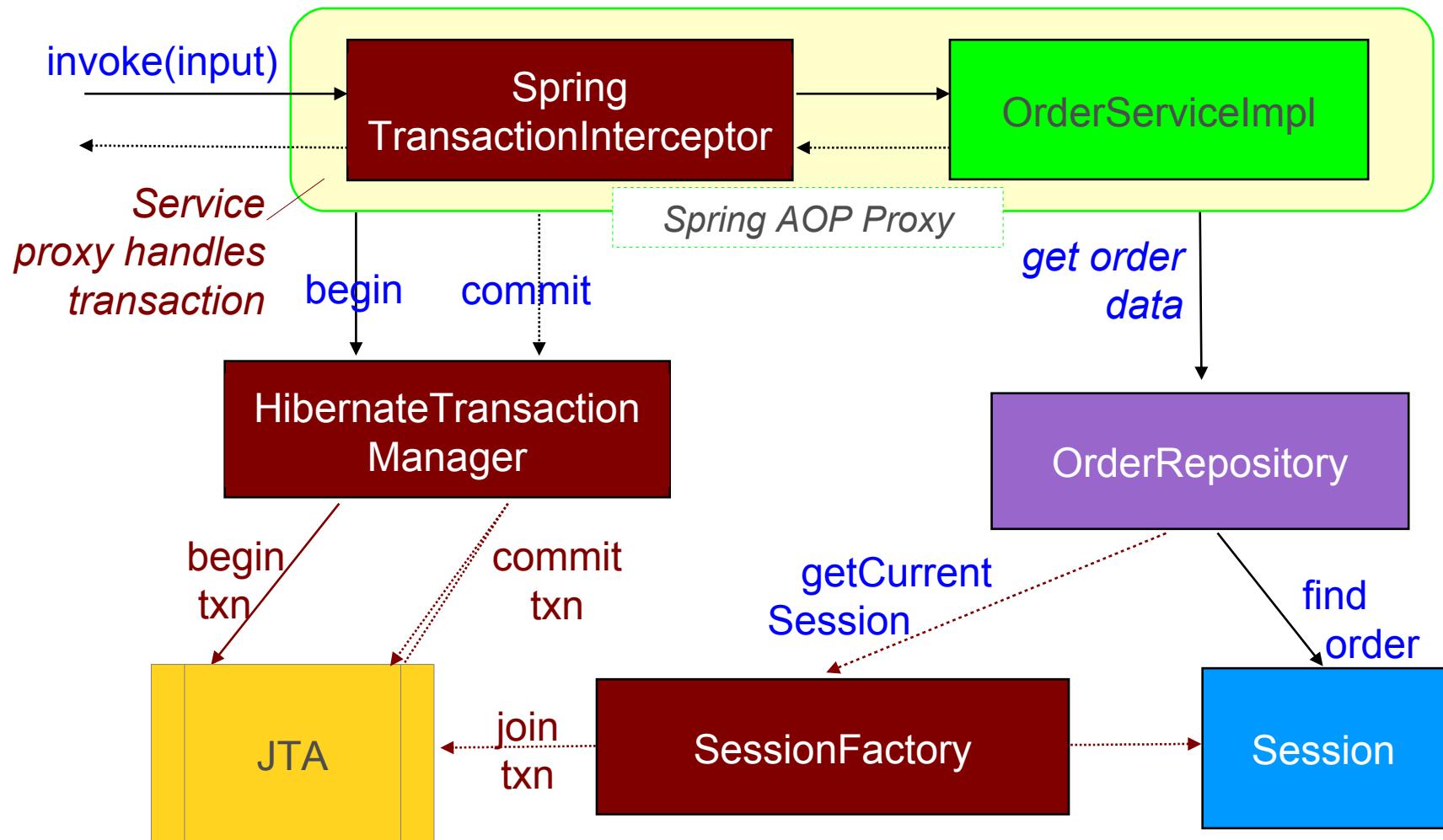


or use **JtaTransactionManager** if needed

# How it Works (Hibernate)



# How it Works (JTA)



# Spring-managed Transactions and Sessions (4)

- Best practice: use read-only transactions when you *don't* write anything to the database
- Prevents Hibernate from flushing its session
  - possibly dramatic performance improvement
- Also marks JDBC Connection as read-only
  - provides additional safeguards with some databases
  - for example: Oracle only accepts SELECT statements

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

# Topics in this session

- Introduction to Hibernate
- Configuring a Hibernate SessionFactory
- Implementing Native Hibernate DAOs
- **Exception Mapping**

# Transparent Exception Translation

- Used as-is, the previous DAO implementation can throw an access specific **HibernateException**
  - Propagate up to the caller
    - service layer or other users of the DAOs
  - Introduces dependency on specific persistence solution that should not exist
    - Service layer “knows” you are using Hibernate
- Can use AOP to convert them to Spring’s rich **DataAccessException** hierarchy
  - Hides access technology used

# Exception Translation

## – Using `@Repository`

Spring provides this capability out of the box

- Annotate with `@Repository`
- Define a Spring-provided BeanPostProcessor

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

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

# Exception Translation - XML

- Can't always use annotations
  - For example with a third-party repository
  - Use XML to configure instead

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

No annotations

```
<bean id="persistenceExceptionInterceptor"  
      class="org.springframework.dao.support.  
          PersistenceExceptionTranslationInterceptor"/>  
  
<aop:config>  
    <aop:advisor pointcut="execution(* *..OrderRepository+.*(..))"  
                 advice-ref="persistenceExceptionInterceptor" />  
</aop:config>
```

# Summary

- Use Hibernate and/or JPA to define entities
  - Repositories have no Spring dependency
- Use Spring to configure Hibernate session factory
  - Works with Spring-driven transactions
  - Optional translation to DataAccessExceptions

*Spring-Hibernate – 3 day in-depth Hibernate course*

# Lab

Using Hibernate with Spring

# Introduction to Spring Remoting

## Simplifying Distributed Applications

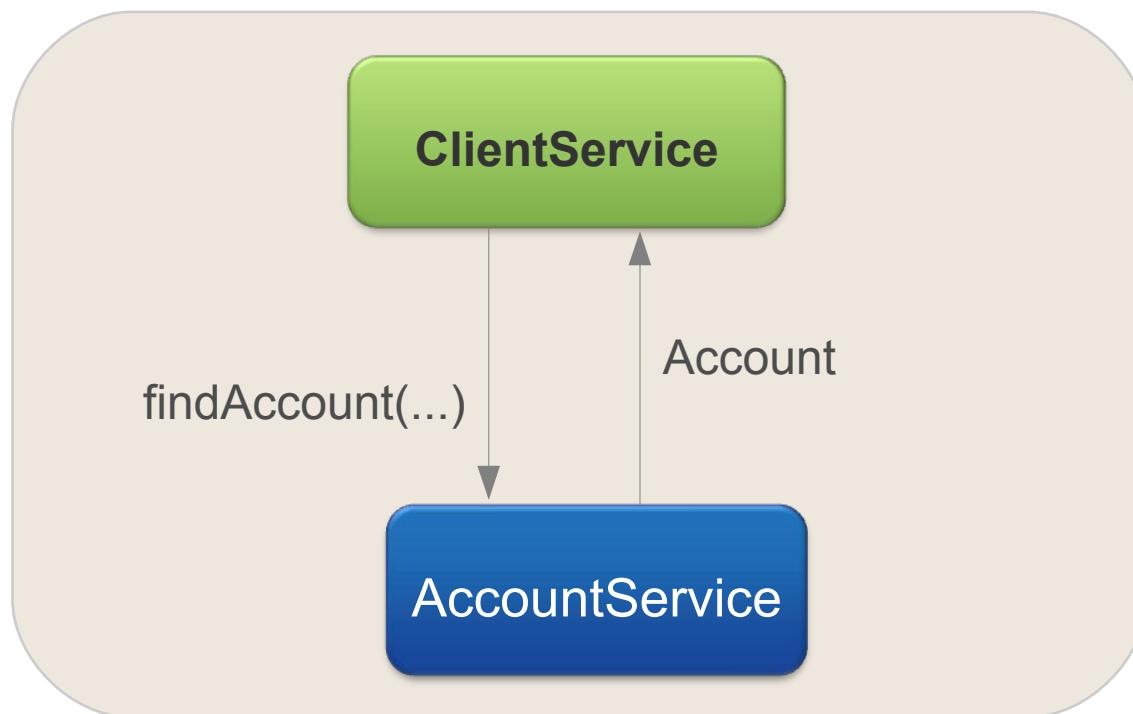
Painless RMI and HTTP Tunnelling

# Topics in this Session

- **Introduction to Remoting**
- Spring Remoting Overview
- Spring Remoting and RMI
- `HttpInvoker`

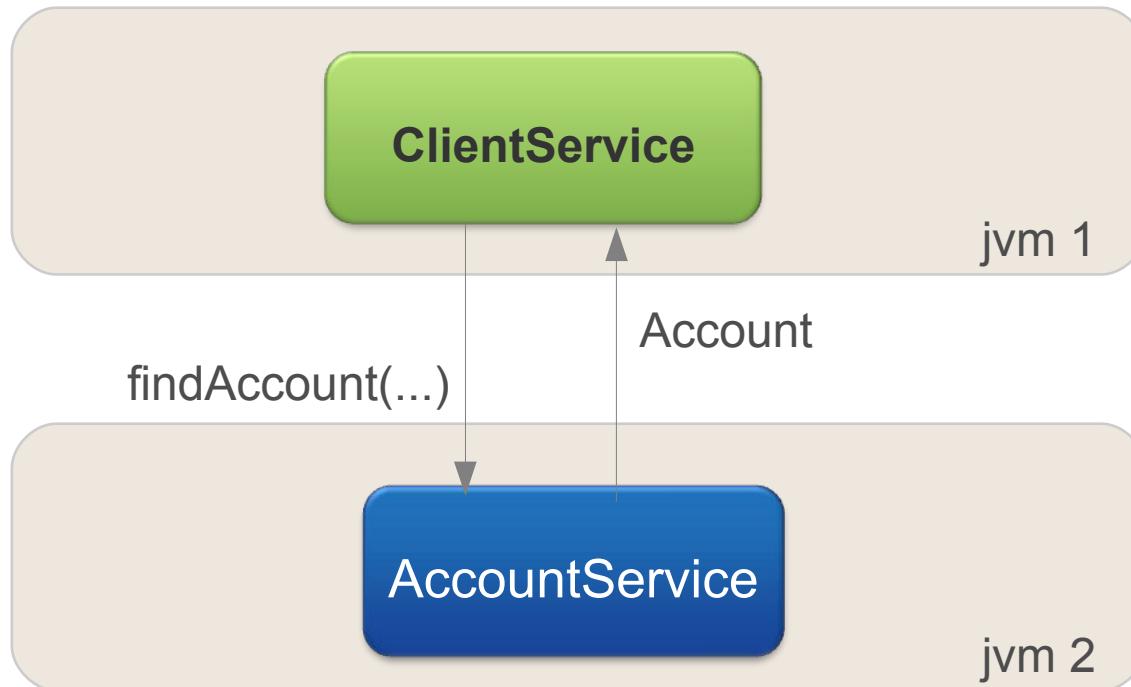
# Local access

- So far, you have seen how to access objects locally



# Remote access

- What if those 2 objects run in some separate JVMs?



In this module, only Java-to-Java communication is addressed  
(as opposed to remote access using Web Services or JMS)

# The RMI Protocol

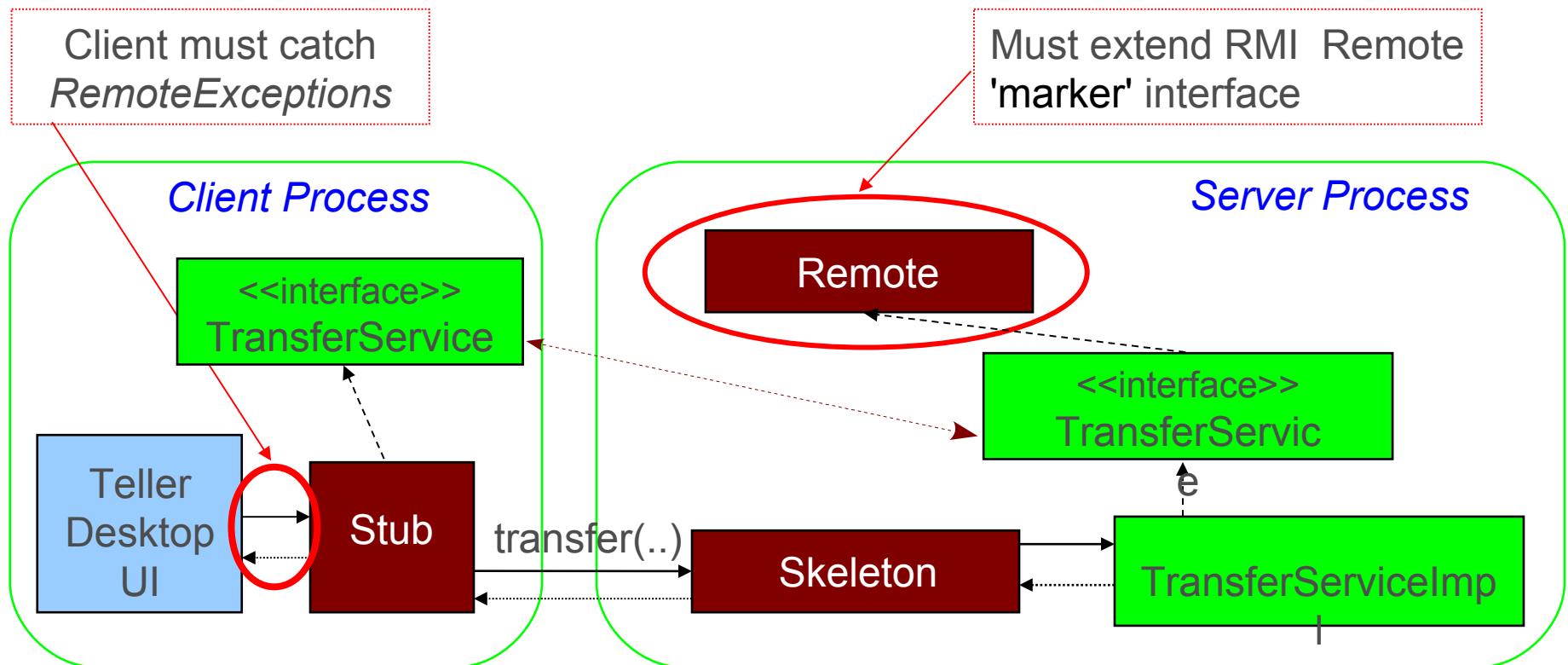
- Standard Java remoting protocol
  - Remote Method Invocation
  - Java's version of RPC
- Server-side exposes a *skeleton*
- Client-side invokes methods on a *stub* (proxy)
- Java serialization is used for marshalling

# Working with plain RMI

- RMI violates separation of concerns
  - Couples business logic to remoting infrastructure
- For example, with RMI:
  - Service interface extends **Remote**
  - Client must catch **RemoteExceptions**
- Technical Java code needed for binding and retrieving objects on the RMI server

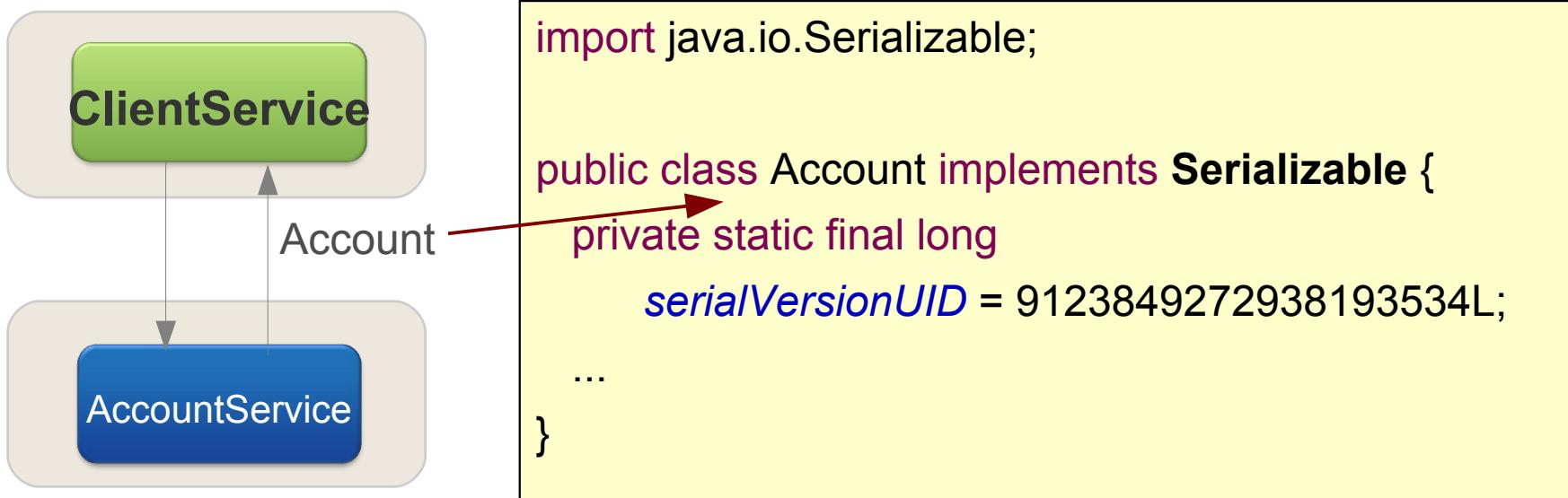
# Traditional RMI

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



# RMI and Serialization

- RMI relies on Object Serialization
  - Objects transferred using RMI should implement 'marker' interface *Serializable*
  - Marker interface, no method to be implemented



# Topics in this Session

- Introduction to Remoting
- **Spring Remoting Overview**
- Spring Remoting and RMI
- HttpInvoker

# Goals of Spring Remoting

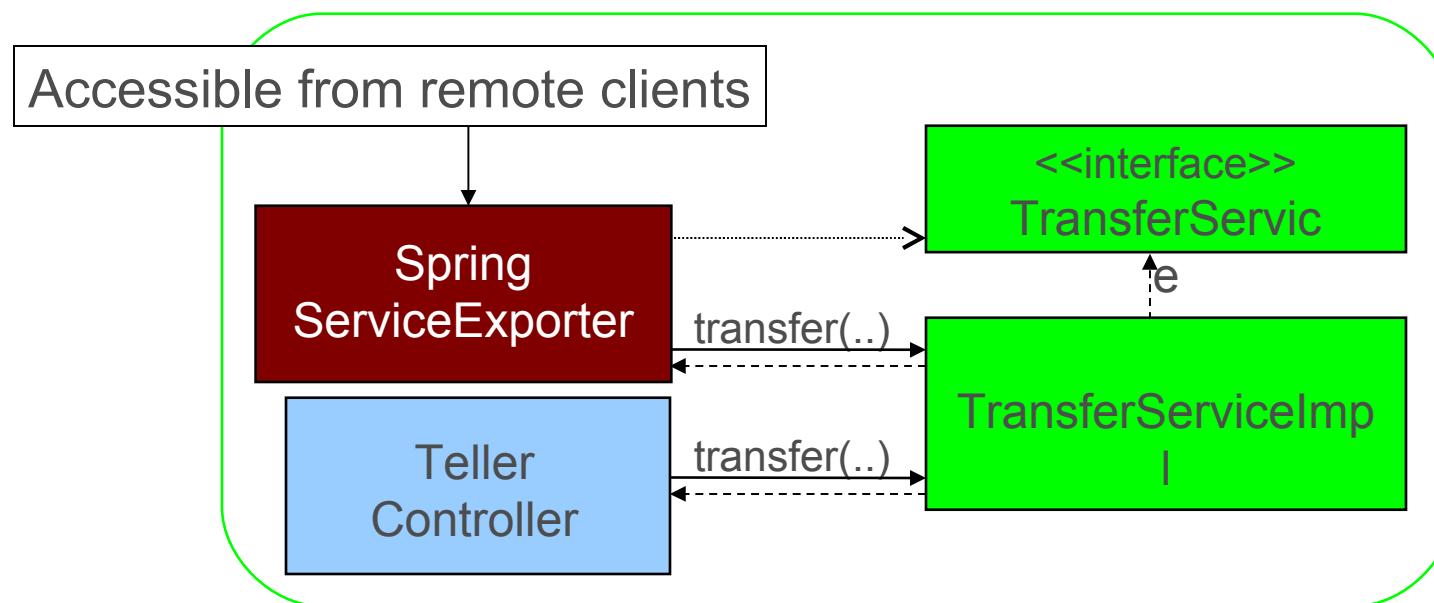
- Hide “plumbing” code
- Configure and expose services declaratively
- Support multiple protocols in a consistent way

# Hide the Plumbing

- Spring provides **exporters** to handle server-side requirements
  - Binding to registry or exposing an endpoint
  - Conforming to a programming model if necessary
- Spring provides FactoryBeans that generate **proxies** to handle client-side requirements
  - Communicate with the server-side endpoint
  - Convert remote exceptions to a runtime hierarchy

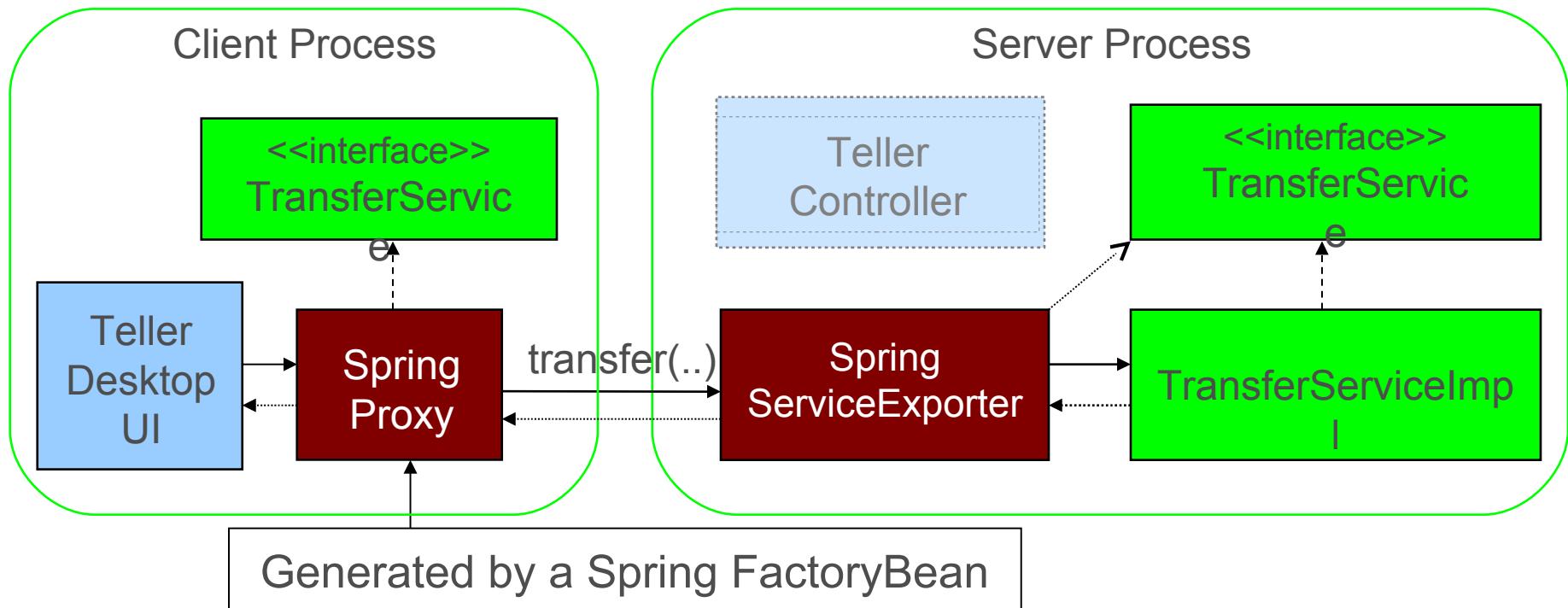
# Service Exporters

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



# Client Proxies

- Dynamic proxies generated by Spring communicate with the service exporter



# A Declarative Approach

- Uses a configuration-based approach
  - No code to write
- On the server side
  - Expose existing services with NO code changes
- On the client side
  - Invoke remote methods from existing code
  - Take advantage of polymorphism by using dependency injection
  - Migrate between remote vs. local deployments

# Consistency across Protocols

- Spring's exporters and proxy FactoryBeans bring the same approach to *multiple* protocols
  - Provides flexibility
  - Promotes ease of adoption
- On the server side
  - Expose a single service over multiple protocols
- On the client side
  - Switch easily between protocols

# Topics in this Session

- Introduction to Remoting
- Spring Remoting Overview
- **Spring Remoting and RMI**
- HttpInvoker

# Spring's RMI Service Exporter

- Transparently expose an existing POJO service to the RMI registry
  - No need to write the binding code
- Avoid traditional RMI requirements
  - Service interface does not extend **Remote**
  - Service class is a POJO



Transferred objects still need to implement the  
*java.io.Serializable* interface

# Configuring the RMI Service Exporter

Server

- Start with an existing POJO service

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

- Define a bean to export it

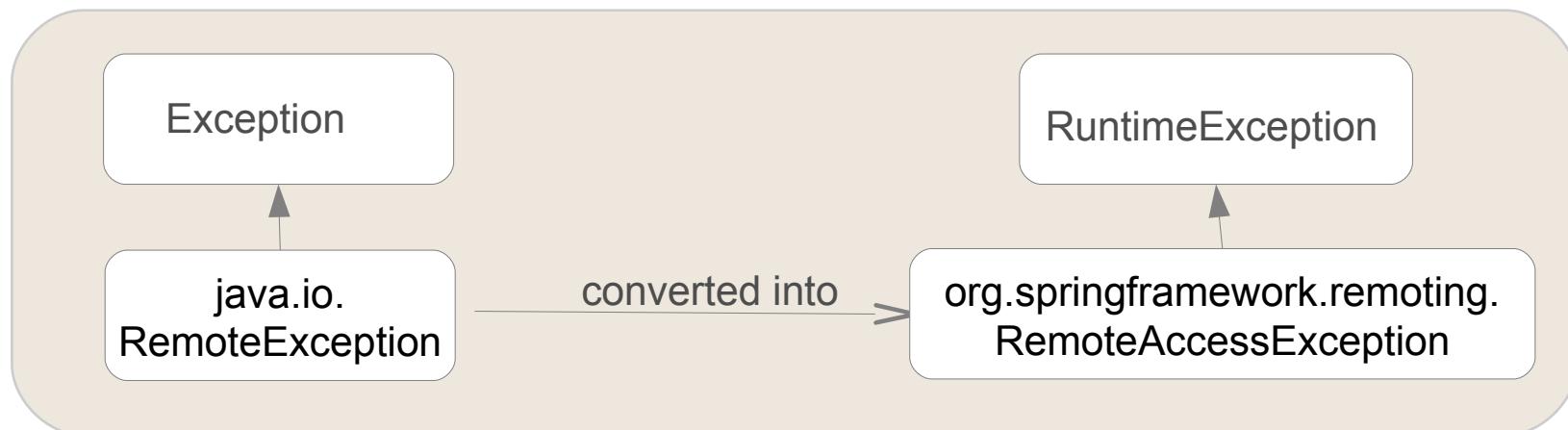
```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName" value="transferService"/>
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="service" ref="transferService"/>
    <property name="registryPort" value="1096"/>
</bean>
```

*"registryPort" defaults to "1099"*

Binds to rmiRegistry  
as "transferService"

# Spring's RMI Proxy Generator

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



# Configuring the RMI Proxy

Client

- Define a factory bean to generate the proxy

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

- Inject it into the client

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

TellerDesktopUI only depends on the TransferService interface

# Topics in this Session

- Introduction to Remoting
- Spring Remoting Overview
- Spring Remoting and RMI
- **HttpInvoker**

# Spring's HttpInvoker

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



HttpInvoker is using serialization → transferred objects still need to implement the `java.io.Serializable` interface

# Configuring the HttpInvoker Service Exporter (1)

Server

- Start with an existing POJO service

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

- Define a bean to export it

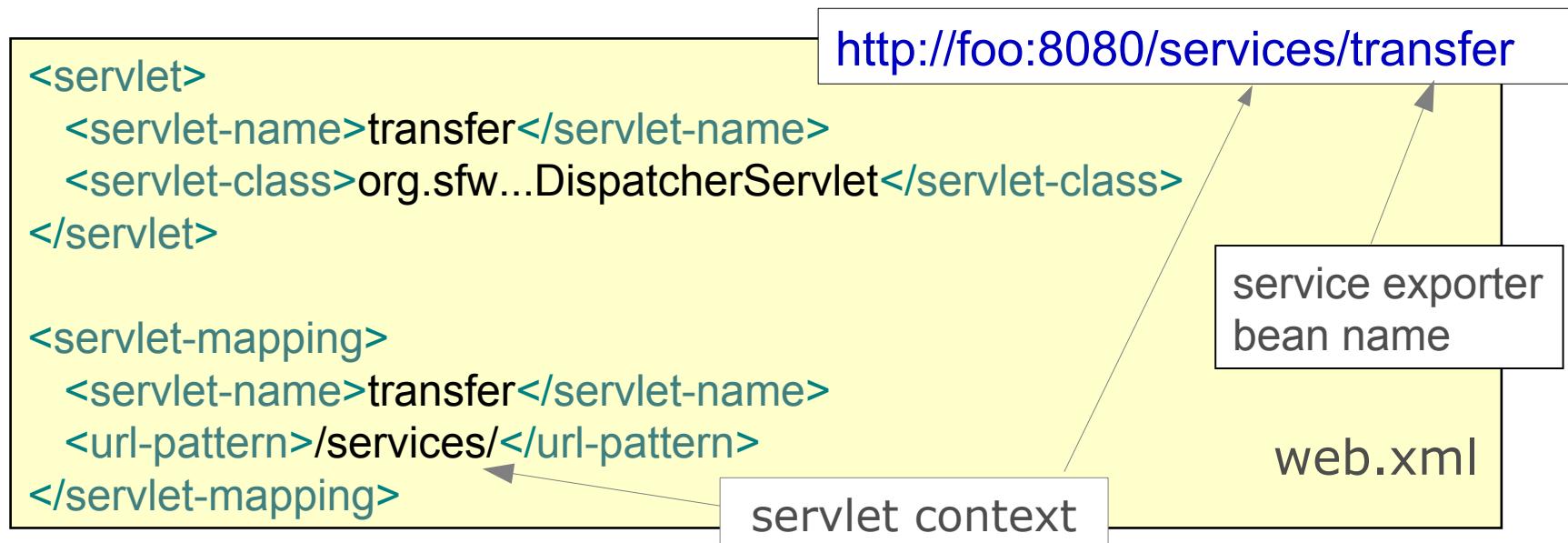
```
<bean id="/transfer" <-- endpoint for HTTP request handling
      class="org.springframework.remoting.httpinvoker.
            HttpInvokerServiceExporter">
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="service" ref="transferService"/>
</bean>
```



**Pre Spring 3.1:** Spring did not allow characters such as '/' in an *id* attribute.  
If special characters are needed, the *name* attribute could be used instead.

# Configuring the HttpInvoker Service Exporter (2a)

- Expose the HTTP service via DispatcherServlet
  - Uses **BeanNameUrlHandlerMapping** to map requests to service
  - Can expose *multiple* exporters



# Configuring the HttpInvoker Service Exporter (2b)

- Alternatively define HttpServletRequestHandlerServlet
  - Doesn't require Spring-MVC
  - Service exporter bean is defined in root context
  - No handler mapping – *one servlet per HttpInvoker exporter*
  - Servlet name *must* match bean name

The diagram illustrates the configuration of an `HttpServletRequestHandlerServlet` in a `web.xml` file. It shows two main sections: a `<servlet>` section and a `<servlet-mapping>` section.

**<servlet> Configuration:**

- `<servlet-name>transfer</servlet-name>` (highlighted in blue)
- `<servlet-class>org.sfw...HttpServletRequestHandlerServlet</servlet-class>` (highlighted in blue)

A callout box labeled "http://foo:8080/services/transfer" points to the `<servlet-name>` element.

**<servlet-mapping> Configuration:**

- `<servlet-name>transfer</servlet-name>` (highlighted in blue)
- `<url-pattern>/services/transfer</url-pattern>` (highlighted in blue)

A callout box labeled "service exporter bean name" points to the `<servlet-name>` element in the `<servlet-mapping>` section.

Below the `<servlet-mapping>` section, the word "web.xml" is written.

# Configuring the HttpInvoker Proxy

Client

- Define a factory bean to generate the proxy

```
<bean id="transferService"
      class="org.springframework.remoting.httpinvoker.
      HttpInvokerProxyFactoryBean">
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="serviceUrl" value="http://foo:8080/services/transfer"/>
</bean>
```

HTTP POST requests will be sent to this URL

- Inject it into the client

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

# Remoting: Pros and Cons

- Advantages
  - (Too) Simple to setup and use
  - Abstracts away all messaging concerns
- Disadvantages
  - Client-server tightly coupled by shared interface
    - Hard to maintain, especially with lots of clients
  - Can't control underlying messaging (hidden)
  - Difficult to scale
    - Make your requests *high-level*
  - Not interoperable, Java only

*Spring Enterprise – 4 day course on application integration*

# Lab

Implementing Distributed Applications  
with Spring Remoting

# Performance and Operations

Management and Monitoring of Spring Java Applications

Exporting Spring Beans to JMX

# Topics in this Session

- Introduction
- JMX
- Introducing Spring JMX
- Automatically exporting existing MBeans
- Spring Insight

# Overall Goals

- Gather information about application during runtime
- Dynamically reconfigure app to align to external occasions
- Trigger operations inside the application
- Even adapt to business changes in smaller scope

# Topics in this Session

- Introduction
- JMX
- Introducing Spring JMX
- Automatically exporting existing MBeans
- Spring Insight

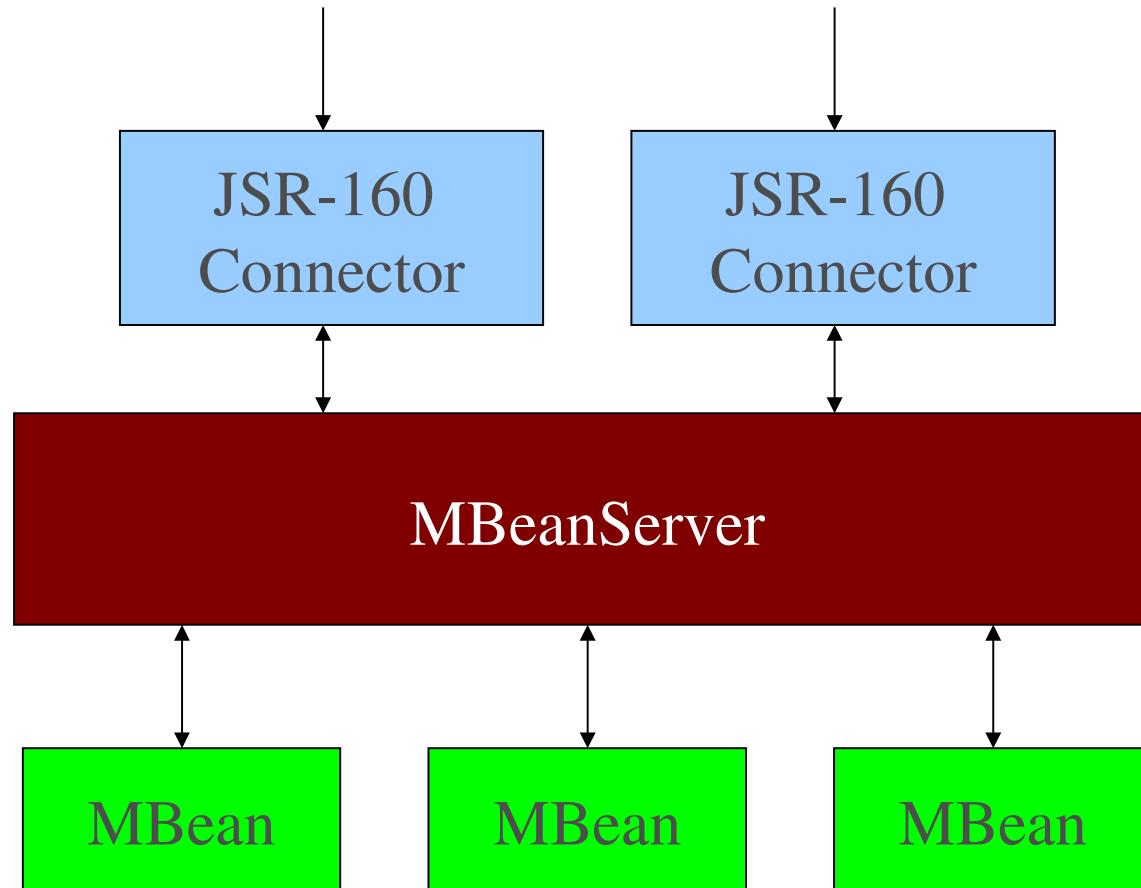
# What is JMX?

- The Java Management Extensions specification aims to create a standard API for adding management and monitoring to Java applications
- Management
  - Changing configuration properties at runtime
- Monitoring
  - Reporting cache hit/miss ratios at runtime

# How JMX Works

- To add this management and monitoring capability, JMX instruments application components
- JMX introduces the concept of the MBean
  - An object with management metadata

# JMX Architecture



# JMX Architecture

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

# JMX Architecture

- An MBean is an object with additional management metadata
  - Attributes (→ properties)
  - Operations (→ methods)
- The management metadata can be defined statically with a Java interface or defined dynamically at runtime
  - Simple MBean or Dynamic MBean respectively

# Plain JMX – Example Bean

```
public interface JmxCounterMBean {  
  
    int getCount(); // becomes Attribute named 'Count'  
  
    void increment(); // becomes Operation named 'increment'  
}
```

```
public class JmxCounter implements JmxCounterMBean {  
    ...  
    public int getCount() {...}  
  
    public void increment() {...}  
}
```

# Plain JMX – Exposing an MBean

```
MBeanServer server = ManagementFactory.getPlatformMBeanServer();

JmxCounter bean = new JmxCounter(...);

try {
    ObjectName name = new ObjectName("ourapp:name=counter");
    server.registerMBean(bean, name);
} catch (Exception e) {
    e.printStackTrace();
}
```

# Topics in this Session

- Introduction
- JMX
- **Introducing Spring JMX**
- Automatically exporting existing MBeans
- Spring Insight

# Goals of Spring JMX

- Using the raw JMX API is difficult and complex
- The goal of Spring's JMX support is to simplify the use of JMX while hiding the complexity of the API

# Goals of Spring JMX

- Configuring JMX infrastructure
  - Declaratively using context namespace or FactoryBeans
- Exposing Spring beans as MBeans
  - Annotation based metadata
  - Declaratively using Spring bean definitions
- Consuming JMX managed beans
  - Transparently using a proxy-based mechanism

# Spring JMX Steps

1. Configuring MBean Server
2. Configure Exporter
3. Control Attribute / Operation Exposure.

# Step 1: Creating an MBeanServer

- Use context namespace to locate or create an MBeanServer

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

XML

- Or declare it explicitly

or JavaConfig

```
@Bean  
public MBeanServerFactoryBean mbeanServer () {  
    MBeanServerFactoryBean server = new MBeanServerFactoryBean();  
    server.setLocateExistingServerIfPossible( true );  
    ...  
    return server;  
}
```

# Step 2: Exporting a Bean as an MBean

- Start with one or more existing POJO bean(s)

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

- Use the MBeanExporter to export it
  - By default: *all public* properties exposed as attributes, *all public* methods exposed as operations.

```
@Bean  
public MBeanExporter mbeanExporter () {  
    MBeanExporter exporter = new MBeanExporter();  
    exporter.setAutodetect ( true );  
    ...  
    return exporter;  
}
```

JavaConfig

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

or XML

# Step 1 & 2: JavaConfig Shortcut

- One annotation defines server and exporter:

```
@Configuration  
@EnableMBeanExport  
public class MyConfig {  
    ...  
}
```

Specific server bean  
configurable if desired.

### 3. Control Attribute/Operation Exposure:

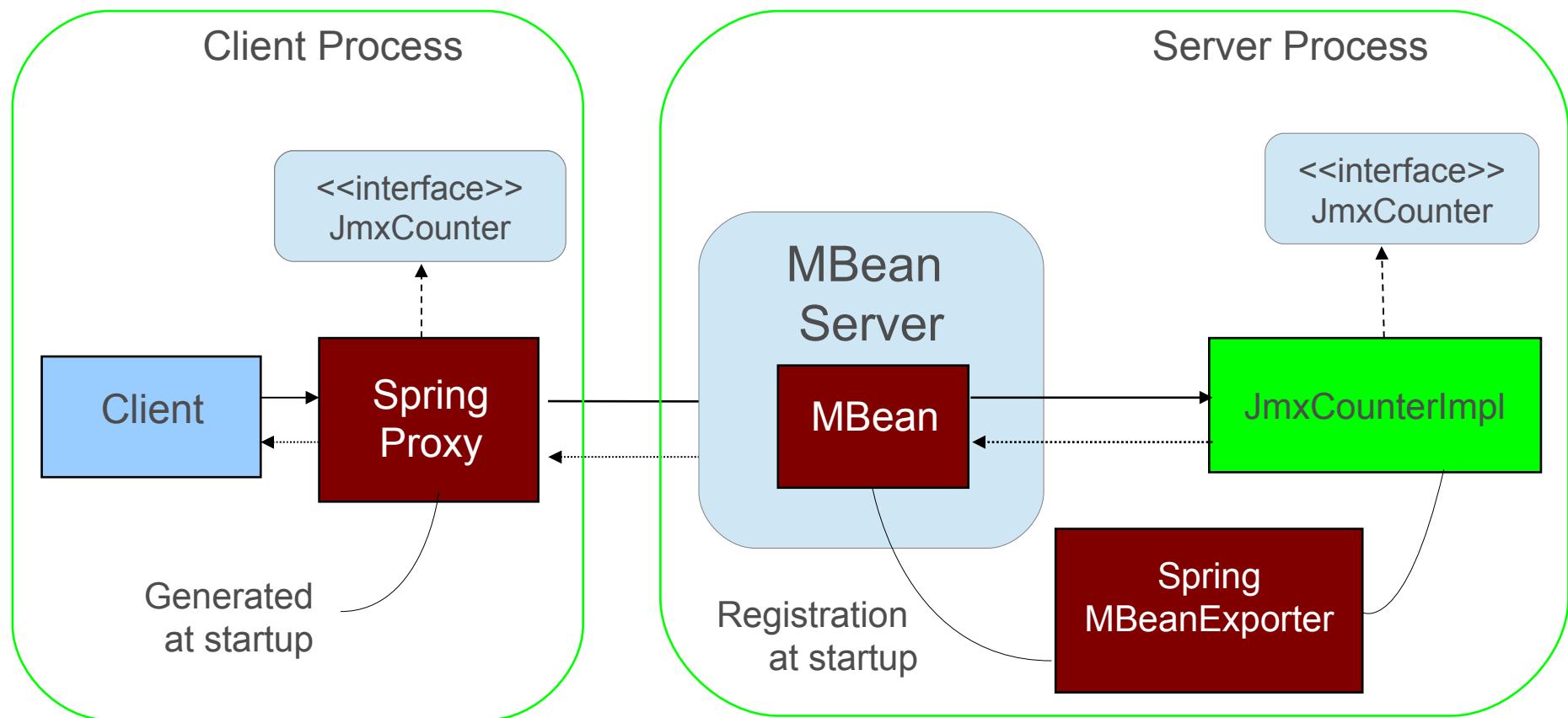
- Combine Annotations with Exporter:
  - Only annotated attributes/operations exposed.

```
@ManagedResource(objectName="statistics:name=counter",
                  description="A simple JMX counter")
public class JmxCounterImpl implements JmxCounter {

    @ManagedAttribute(description="The counter value")
    public int getCount() {...}

    @ManagedOperation(description="Increments the counter value")
    public void increment() {...}
}
```

# Spring in the JMX architecture



# Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- **Automatically exporting existing MBeans**
- Spring Insight

# Automatically Exporting Pre-existing MBeans

- Some beans are MBeans themselves
  - Example: Log4j's LoggerDynamicMBean
  - Spring will auto-detect and export them for you

```
<context:mbean-export/>

<bean class="org.apache.log4j.jmx.LoggerDynamicMBean">
    <constructor-arg>
        <bean class="org.apache.log4j.Logger"
              factory-method="getLogger"/>
        <constructor-arg value="org.springframework.jmx" />
    </bean>
    </constructor-arg>
</bean>
```

# Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- Automatically exporting existing MBeans
- **Spring Insight**

# Spring Insight Overview

- Part of tc Server Developer Edition
  - Monitors web applications deployed to tc Server
  - <http://localhost:8080/insight>
- Focuses on what's relevant
  - esp. performance related parts of the application
- Detects performance issues during development
  - Commercial version for production: *vFabric APM*

# Spring Insight Overview

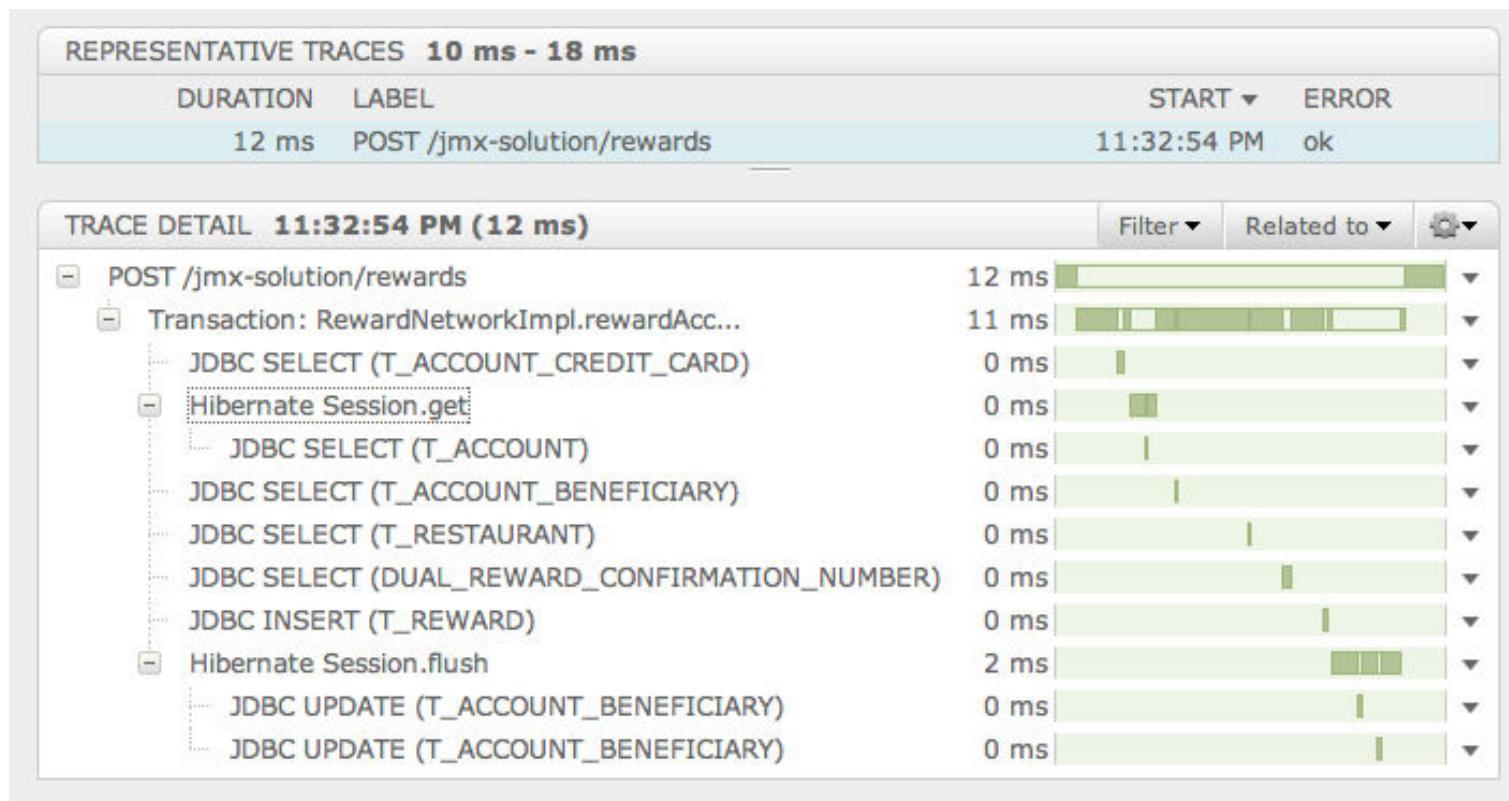
The screenshot shows the Spring Insight interface for monitoring the `/jmx-solution` application. A yellow box labeled "Servlet Selector" points to the tree view under the "APPLICATIONS" section, where the `/jmx-solution` node is expanded to show its sub-components: `Servlet: rewards`, `Servlet: default`, and `/ (Welcome to tc Runtime)`. Another yellow box labeled "Time" points to the "Response Time Trend" chart in the "END POINT" section.

**APPLICATIONS**

- All Applications
  - `/jmx-solution`
    - `Servlet: rewards`
    - `Servlet: default`
  - `/ (Welcome to tc Runtime)`
  - `/manager (Tomcat Manager Appl)`

# Spring Insight Overview

- A request trace from HTTP POST to SQL



# Summary

- Spring JMX
  - Export Spring-managed beans to a JMX MBeanServer
  - Simple value-add now that your beans are managed
- Steps
  - Create MBean server
  - Automatically export annotated and pre-existing Mbeans
    - Use `@EnableMBeanExport` or `<context:mbean-server>` and `<context:mbean-export>`
  - Use Spring annotations to declare JMX metadata
- Spring Insight (tc Server Developer Edition)
  - Deep view into your web-application in STS

# Optional Lab

Monitoring and Managing a Java Application

# Spring Web Services

Implementing Loosely-Coupled, Contract-First Web Services

Using the Spring WS project to implement SOAP

# 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 *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

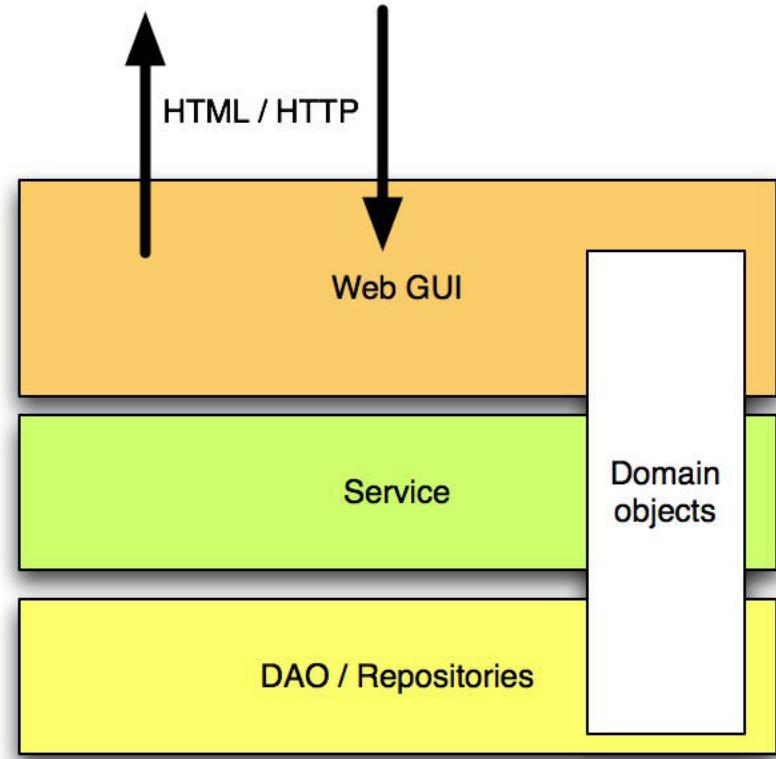
- Remember:
  - web services != SOAP
  - web services != RPC
- Design contract independently from service interface
  - Results in tight coupling and awkward contracts
  - Refrain from using stubs and skeletons
- Consider skipping validation for incoming requests ...  
... use Xpath to *only* extract what you need

## Postel's Law:

“Be conservative in what you do;  
be liberal in what you accept from others.”

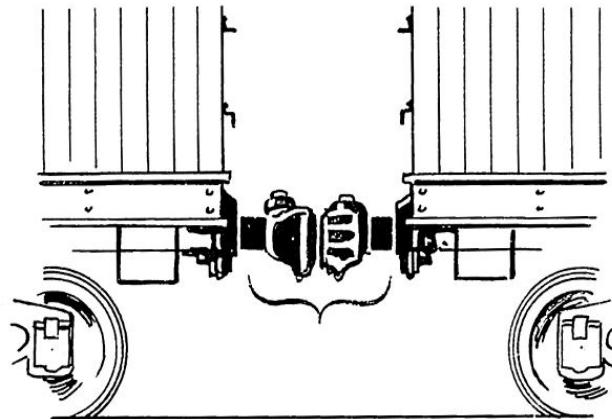
# Web GUI on top of your services

The **Web GUI layer** provides **compatibility** between **HTML-based** world of the user (the browser) and the **OO-based** world of your service



# Web Services enable Loose Coupling

*"Loosely coupled systems are considered useful when either the **source** or the **destination** computer systems are subject to frequent **changes**"*

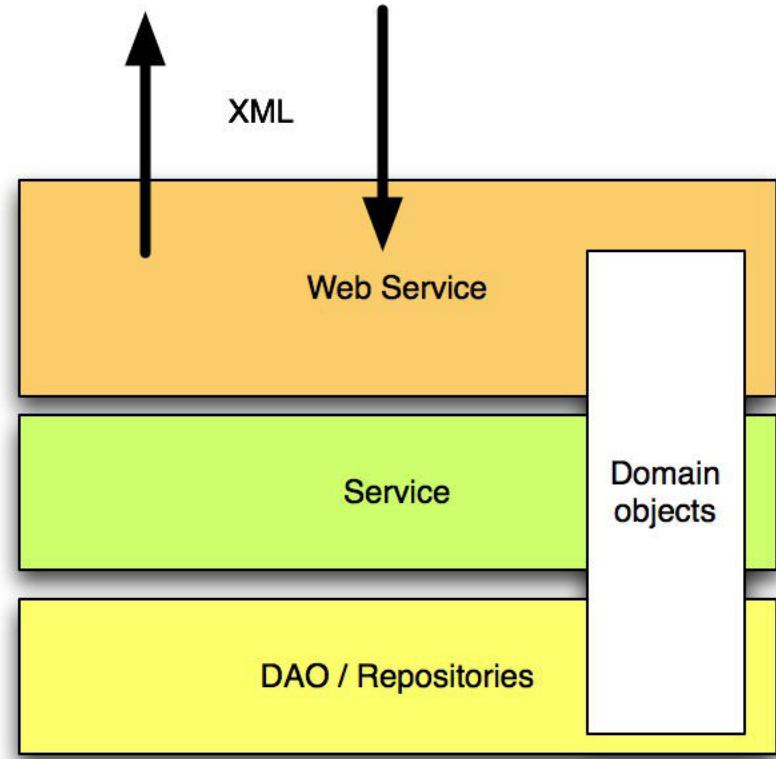


Karl Weikek (attrib)

**Loose coupling increases tolerance...  
changes should not cause incompatibility**

# Web Service on top of your services

**Web Service layer**  
provides **compatibility**  
between **XML-based**  
world of the user and  
the **OO-based** world of  
your service



# Topics in this Session

- Introduction to Web Services
  - Why use or build a web service?
  - Best practices for implementing a web service
- **Spring Web Services**
- Client access

# Define the contract

- Spring-WS uses Contract-first
  - Start with XSD/WSDL
- Widely considered a Best Practice
  - Solves many interoperability issues
- Also considered difficult
  - But isn't

# Contract-first in 3 simple steps

- Create sample messages
- Infer a contract
  - Trang
  - Microsoft XML to Schema
  - XML Spy
- Tweak resulting contract

# Sample Message

Namespace for this message

```
<transferRequest xmlns="http://mybank.com/schemas/tr"
    amount="1205.15">
    <credit>S123</credit>
    <debit>C456</debit>
</transferRequest>
```

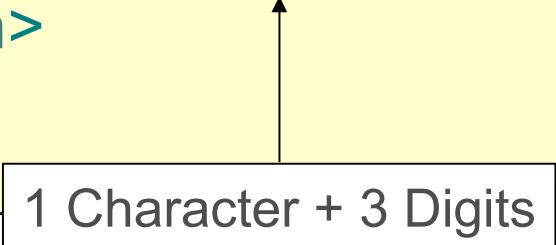
# Define a schema for the web service message

```
<xs:schema  
    xmlns:xs="http://www.w3.org/2001/XMLSchema"  
    xmlns:tr="http://mybank.com/schemas/tr"  
    elementFormDefault="qualified"  
    targetNamespace="http://mybank.com/schemas/tr">  
    <xs:element name="transferRequest">  
        <xs:complexType>  
            <xs:sequence>  
                <xs:element name="credit" type="xs:string"/>  
                <xs:element name="debit" type="xs:string"/>  
            </xs:sequence>  
            <xs:attribute name="amount" type="xs:decimal"/>  
        </xs:complexType>  
    </xs:element>  
</xs:schema>
```

```
<transferRequest  
    amount="1205.15">  
    <credit>S123</credit>  
    <debit>C456</debit>  
</transferRequest>
```

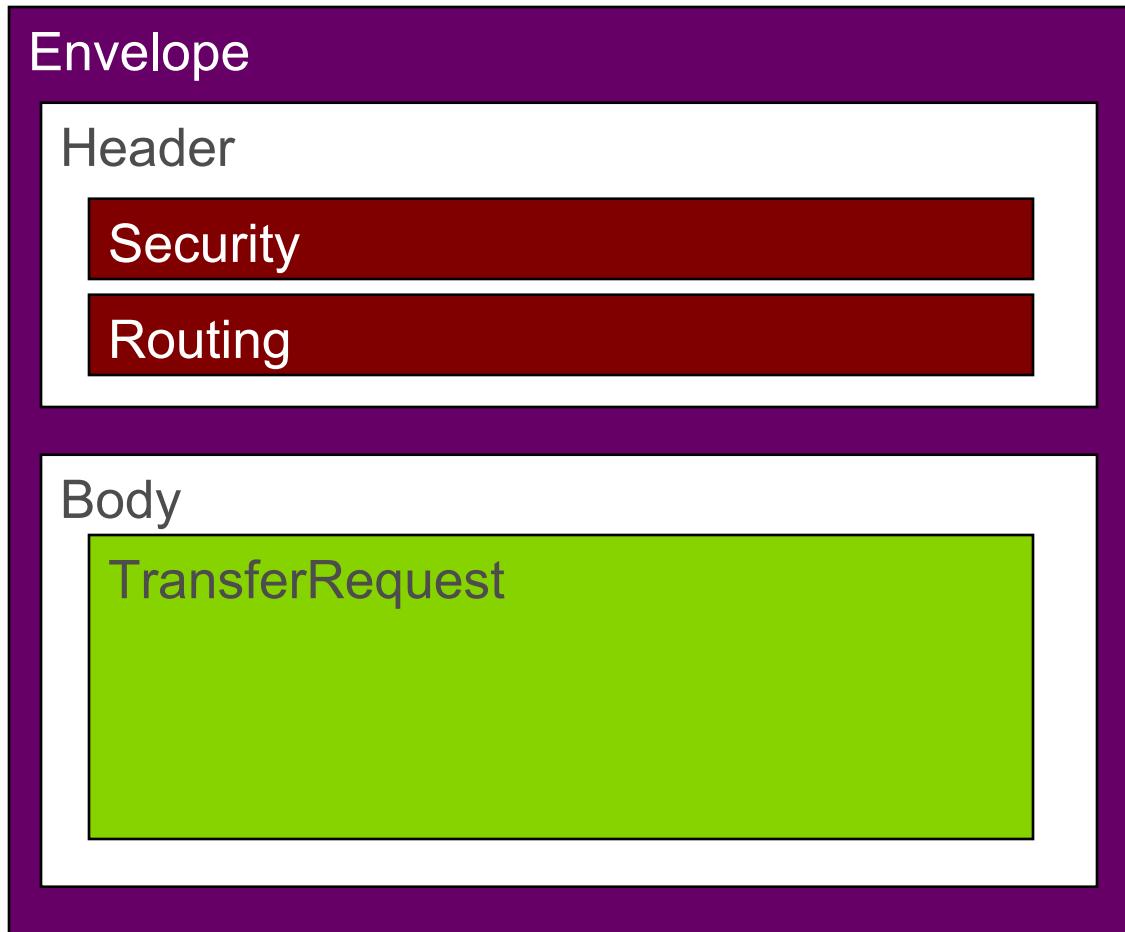
# Type constraints

```
<xs:element name="credit">  
  <xs:simpleType>  
    <xs:restriction base="xs:string">  
      <xs:pattern value="\w\d{3}"/>  
    </xs:restriction>  
  </xs:simpleType>  
</xs:element>
```



The diagram illustrates the type constraint for the 'credit' element. An upward-pointing arrow originates from a rectangular box containing the text "1 Character + 3 Digits". This box is positioned below the XML code. The arrow points directly to the pattern element within the restriction block of the simple type definition.

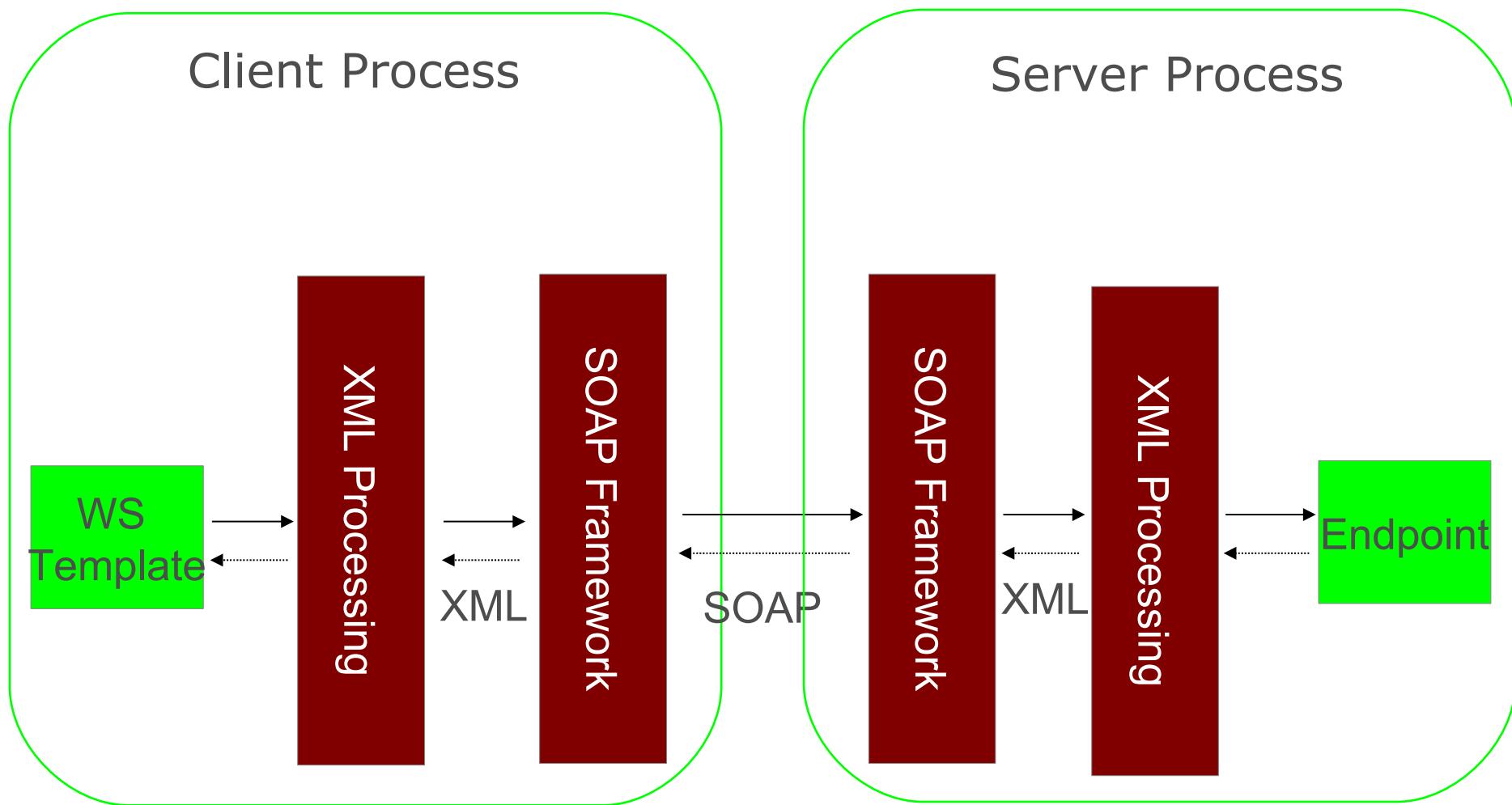
# SOAP Message



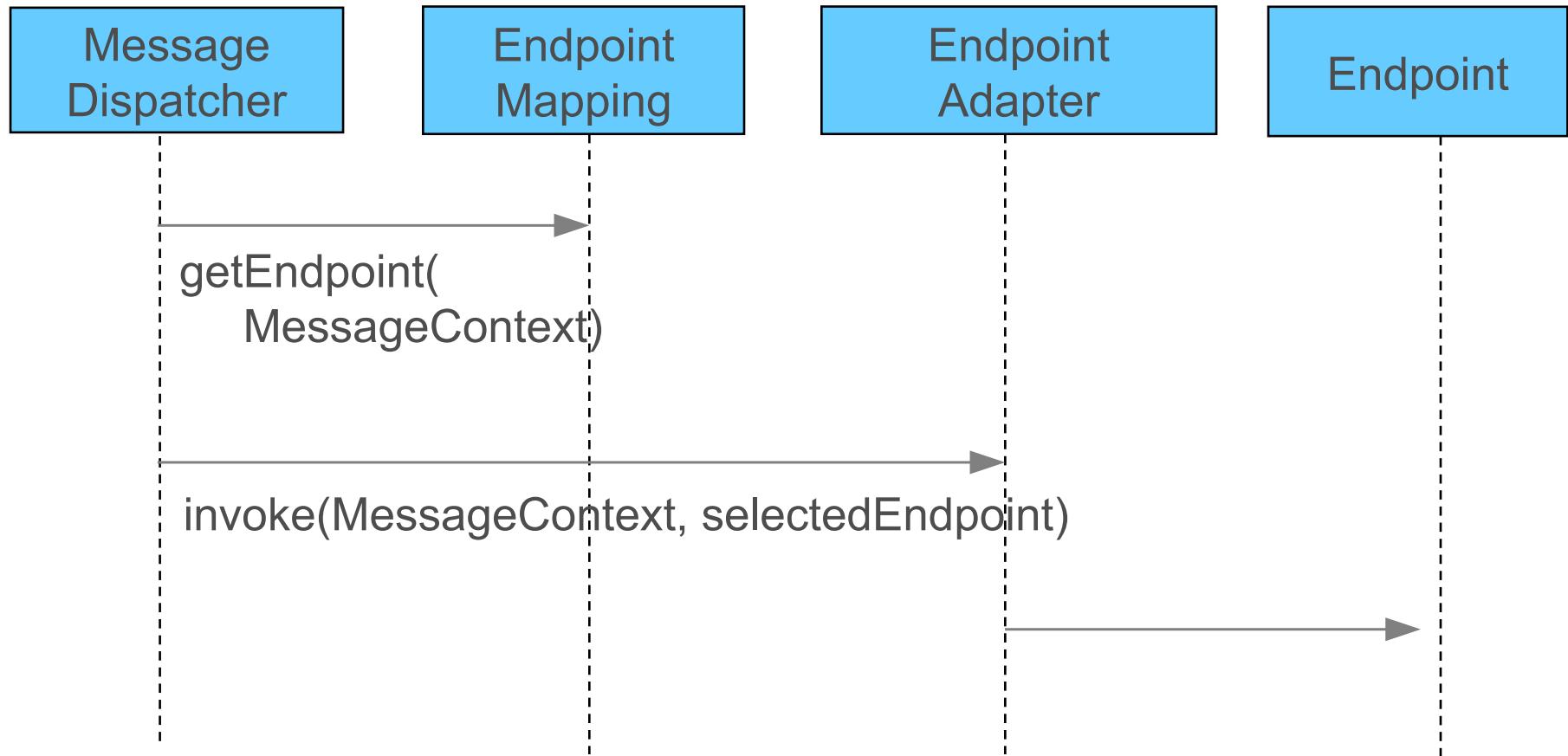
# Simple SOAP 1.1 Message Example

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

# Spring Web Services



# Request Processing



# Bootstrap the Application Tier

- Inside <webapp> within web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/transfer-app-cfg.xml
    </param-value>
</context-param>
```

The application context's configuration file(s)

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

Loads the ApplicationContext into the ServletContext  
before any Servlets are initialized

# Wire up the Front Controller (MessageDispatcher)

- Inside <webapp> within web.xml

```
<servlet>
    <servlet-name>transfer-ws</servlet-name>
    <servlet-class>..ws..MessageDispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/ws-config.xml</param-value>
    </init-param>
</servlet>
```

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

# Map the MessageDispatcherServlet

- Inside <webapp> within web.xml

```
<servlet-mapping>
  <servlet-name>transfer-ws</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

There might also be a web interface (GUI) that is mapped to another path

# Endpoint

- Endpoints handle SOAP messages
- Similar to MVC Controllers
  - Handle input message
  - Call method on business service
  - Create response message
- With Spring-WS you can focus on the Payload
- Switching from SOAP to POX without code change

# XML Handling techniques

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

# JAXB 2

- JAXB 2 is part of Java EE 5 and JDK 6
- Uses annotations for mapping metadata
- Generates classes from a schema and vice-versa
- Supported as part of Spring-OXM

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="rewards.ws.types"/>
```

- No explicit marshaller bean definition necessary to be used in Spring-WS 2.0 endpoints

```
<!-- registers all infrastructure beans needed for annotation-based  
     endpoints, incl. JABX2 (un)marshalling: -->  
<ws:annotation-driven/>
```

# Implement the Endpoint

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

Converted with JAXB2

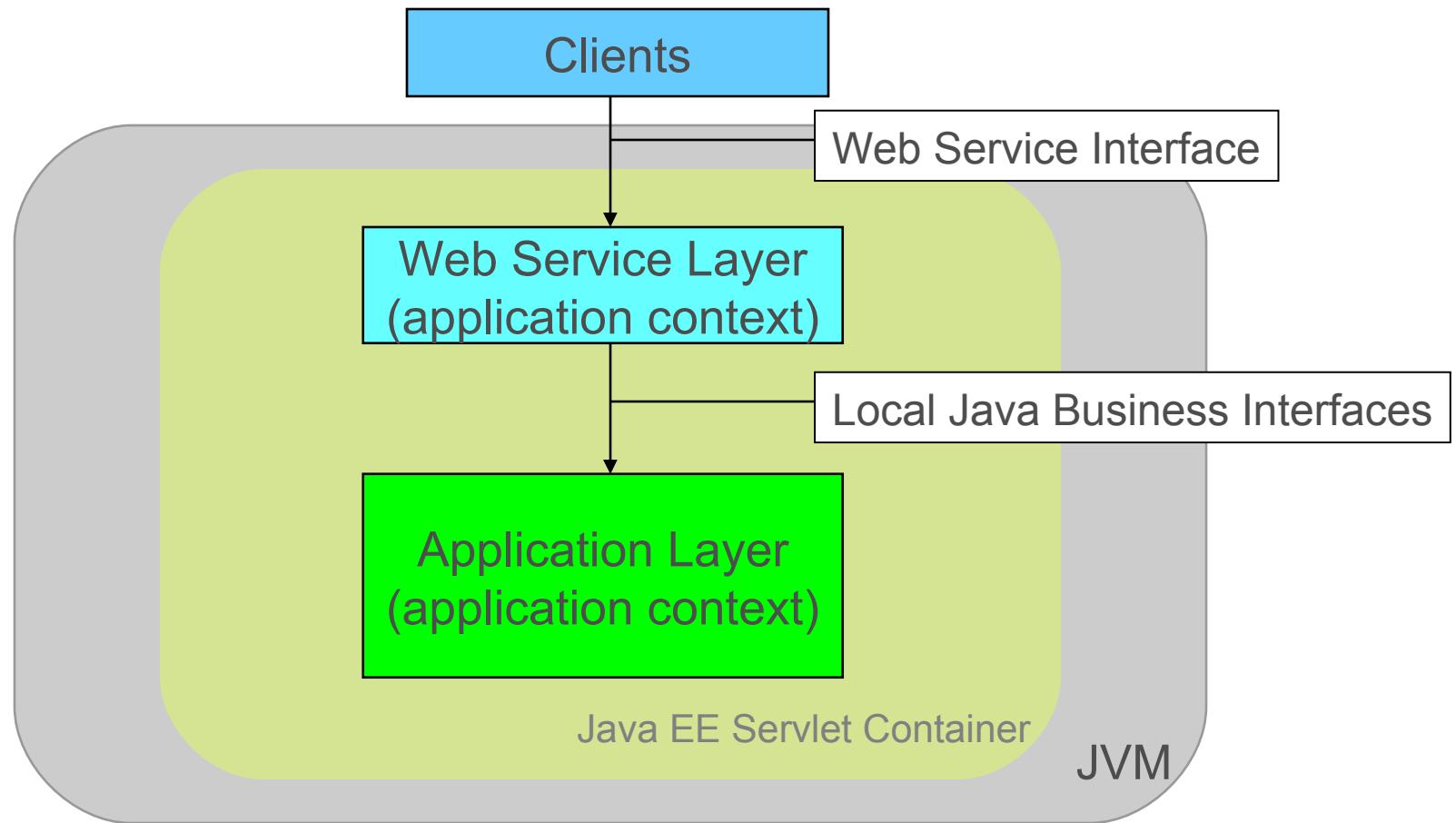
Mapping

# Configure Spring-WS and Endpoint beans

- @Endpoint classes can be component-scanned
  - Saves writing lots of trivial XML bean definitions
  - Just make sure not to scan classes that belong in the root context instead

```
<!-- instead of defining explicit endpoints beans  
     we rely on component scanning: -->  
<context:component-scan base-package="transfers.ws"/>  
  
<!-- registers all infrastructure beans needed for  
     annotation-based endpoints, incl. JAXB2 (un)marshalling: -->  
<ws:annotation-driven/>
```

# Architecture of our Application Exposed using a Web Service



# Further Mappings

- You can also map your Endpoints in XML by
  - Message Payload
  - SOAP Action Header
  - WS-Addressing
  - XPath

# Publishing the WSDL

- Generates WSDL based on given XSD
  - Bean id becomes part of WSDL URL
- Most useful during development
  - Use static WSDL in production
    - Contract shouldn't change unless YOU change it

http://somehost:8080/transferService/transferDefinition.wsdl

```
<ws:dynamic-wsdl id="transferDefinition"
    portTypeName="Transfers"
    locationUri="http://somehost:8080/transferService/"
    <ws:xsd location="/WEB-INF/transfer.xsd"/>
</ws:dynamic-wsdl>
```

# Topics in this Session

- Introduction to Web Services
  - Why use or build a web service?
  - Best practices for implementing a web service
- Spring Web Services
- Client access

# Spring Web Services on the Client

- **WebServiceTemplate**
  - Simplifies web service access
  - Works directly with the XML payload
    - Extracts *body* of a SOAP message
    - Also works with POX (Plain Old XML)
  - Can use marshallers/unmarshallers
  - Provides convenience methods for sending and receiving web service messages
  - Provides callbacks for more sophisticated usage

# Marshalling with WebServiceTemplate

```
<bean id="webServiceTemplate"
      class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="defaultUri" value="http://mybank.com/transfer"/>
    <property name="marshaller" ref="marshaller"/>
    <property name="unmarshaller" ref="marshaller"/>
</bean>

<bean id="marshaller" class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation" value="classpath:castor-mapping.xml"/>
</bean>
```

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

# Lab

Exposing SOAP endpoints using  
Spring Web Services