



# Core Spring

Four Day Workshop

Building Enterprise Applications using Spring

Version 5.0.a

Pivotal

# Copyright Notice

Copyright © 2017 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

- Student introductions
- Self introduction
- Course registration (if needed)
- Courseware
- Internet access
- Phones on silent
- Working hours
- Lunch and breaks
- Toilets/Restrooms
- Fire alarms
- Emergency exits
- Any other questions?



# How You will Benefit

- Learn to use Spring for web and other applications
- Gain hands-on experience
  - Generous mixture of presentation and labs
- Access to experienced, certified instructors



# 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
- Spring FactoryBeans

1

# Course Agenda: Day 2

- Inside the Spring Container
  - Bean Lifecycle and related Post Processors
- 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 with JPA
- Working with JPA in a Spring environment
- Rapidly start new projects with Spring Boot
- Using Spring Data Repositories
- Getting started with Spring MVC

3

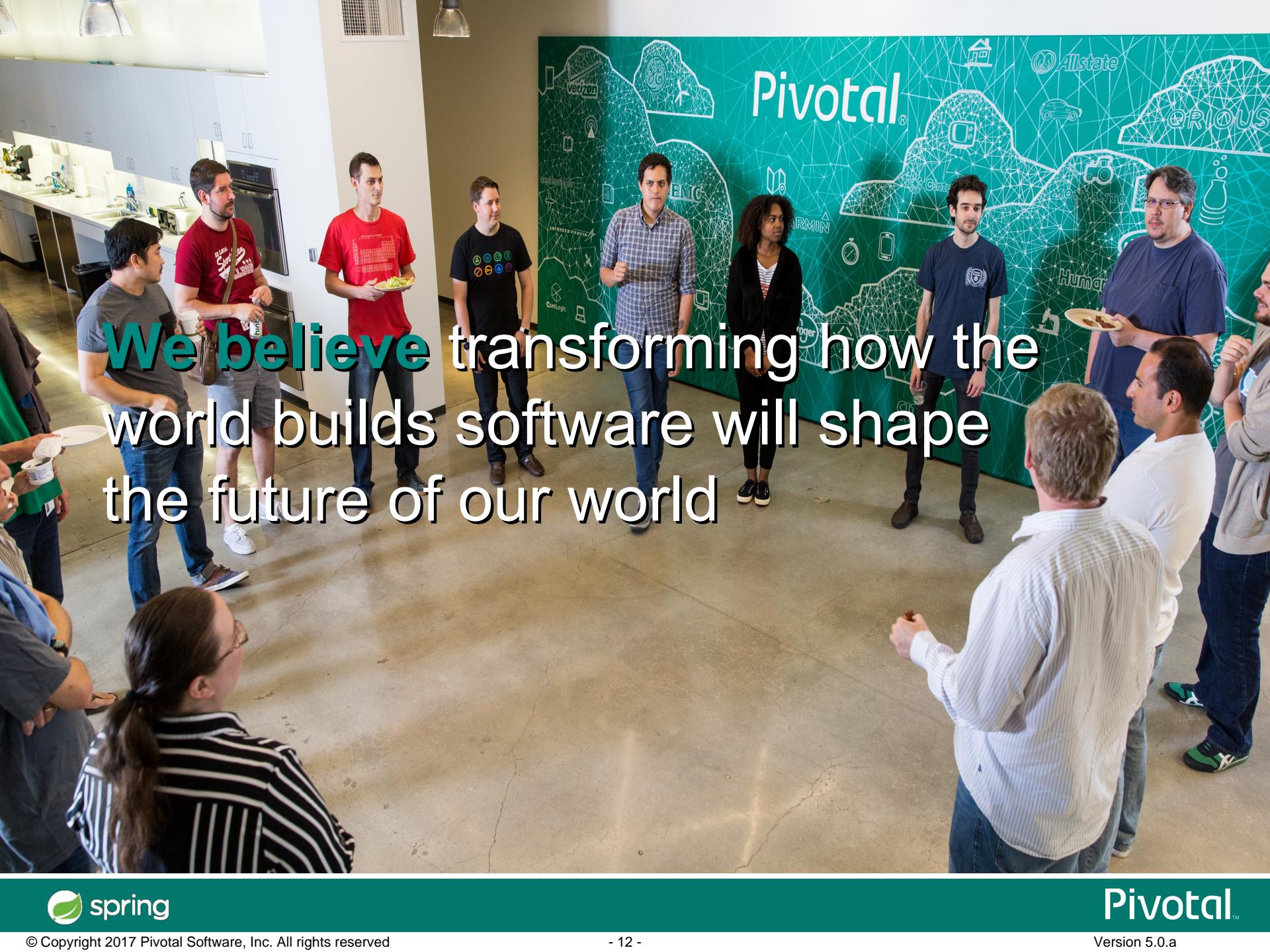
# Course Agenda: Day 4

- Securing web applications with Spring Security
- Implementing REST with Spring MVC
- Optional topics
  - Microservices and Cloud Native Applications using Spring Cloud
  - Reactive Programming with Spring

4

# Covered in this section

- Agenda
- **Spring and Pivotal**



We believe transforming how the world builds software will shape the future of our world

# Spring @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
  - Cloud Foundry PaaS
    - Deploy to private, public, hybrid clouds
    - Microservices, Spring Cloud
  - Real-time analytics, IoT
    - Spot trends as they happen
    - Spring Data, Spring Hadoop, Spring Cloud Data Flow



# Pivotal Platform

## Cloud Foundry

*Cloud Independence  
Microservices  
Continuous Delivery  
Dev Ops*



CLOUD FOUNDRY



## Development

*Frameworks  
Services  
Analytics*



## Big Data Suite

*High Capacity  
Real-time Ingest  
SQL Query  
Scale-out Storage*



HAWQ



GREENPLUM



Pivotal Labs

*Working with clients to build better apps more quickly*

# Spring Projects

Spring Framework

<http://spring.io>



Spring  
Android



Spring Web Flow

Spring  
Session



Spring  
Kafka



Spring Cloud  
Data Flow



Spring  
Boot



Spring  
Mobile



Spring  
Integration



Spring  
AMQP



Spring  
Hateoas



Spring  
Social



Spring  
Cloud



Spring  
Data



Spring  
Batch



Spring (SOAP)  
Web Services



Framework

# 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
  - Explain what Spring is used for
  - Discuss 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>

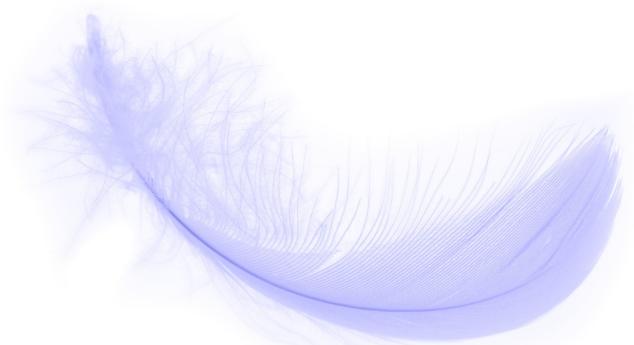


The use of a transitive dependency management system (Maven, Gradle, Ant/Ivy) is recommended for any Java application

# 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
- Low overhead
  - Spring jars are relatively small
    - 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

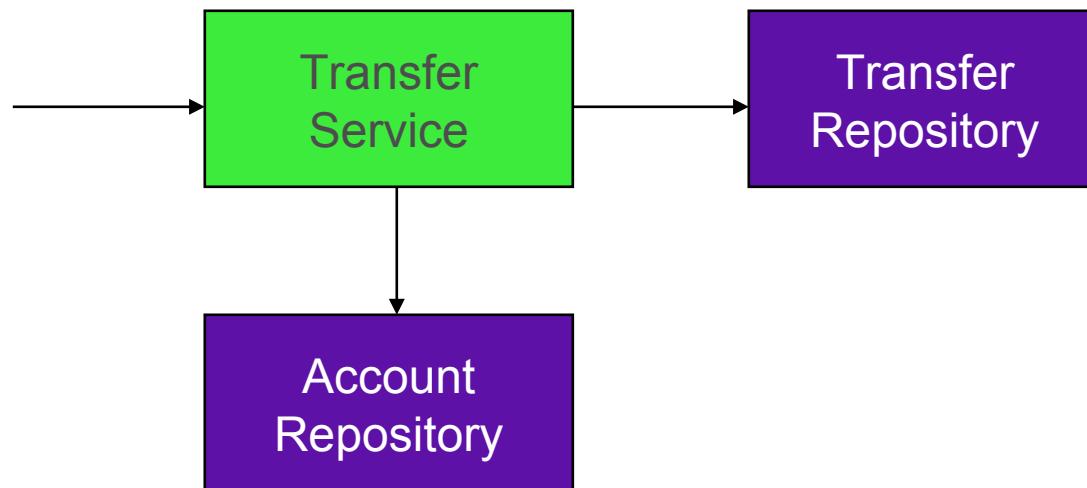


# Topics in this session

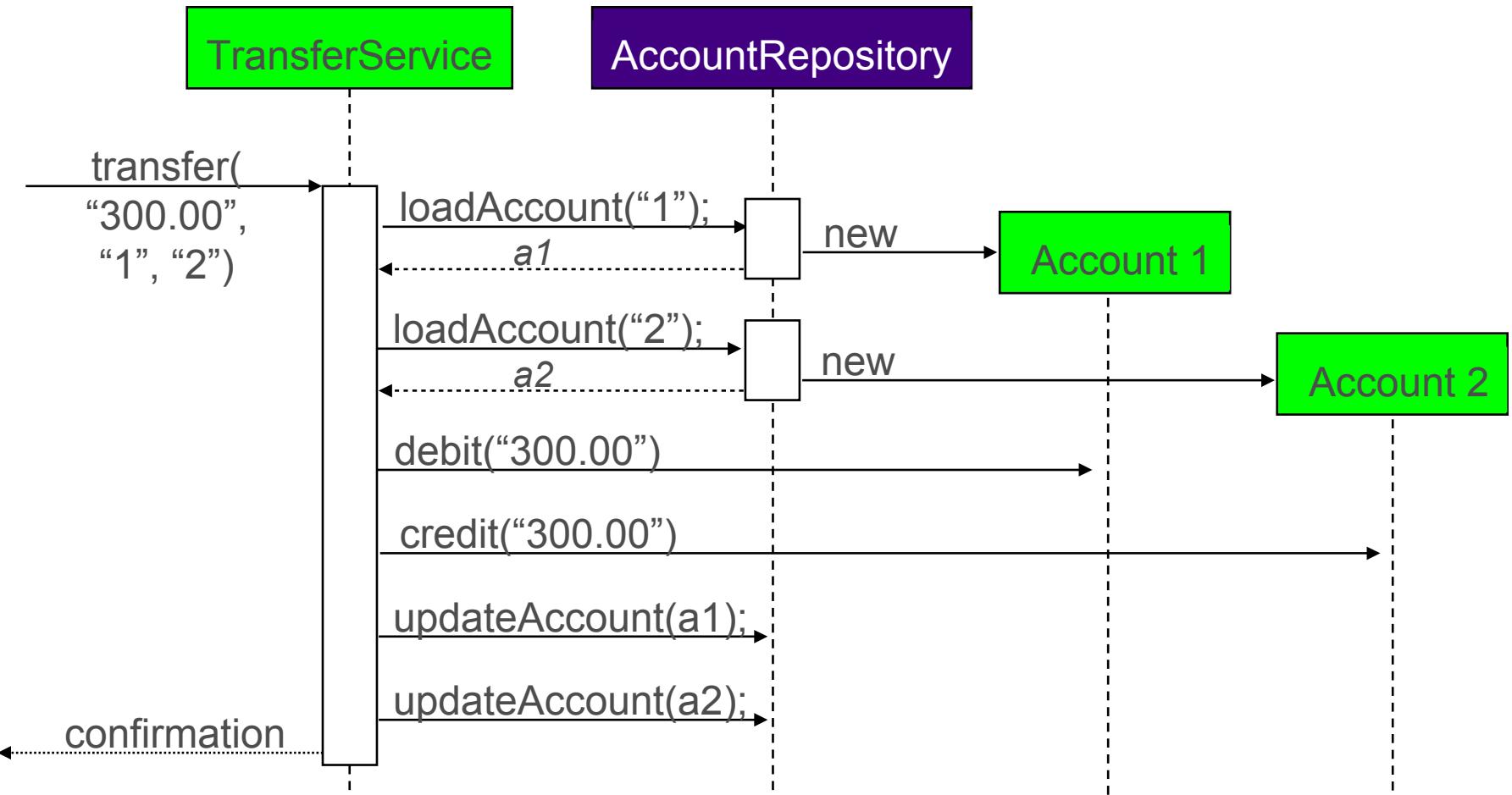
- What is the Spring Framework?
- **Spring is a Container**
- Spring Framework History
- What is Spring Used For?

# Application Configuration

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



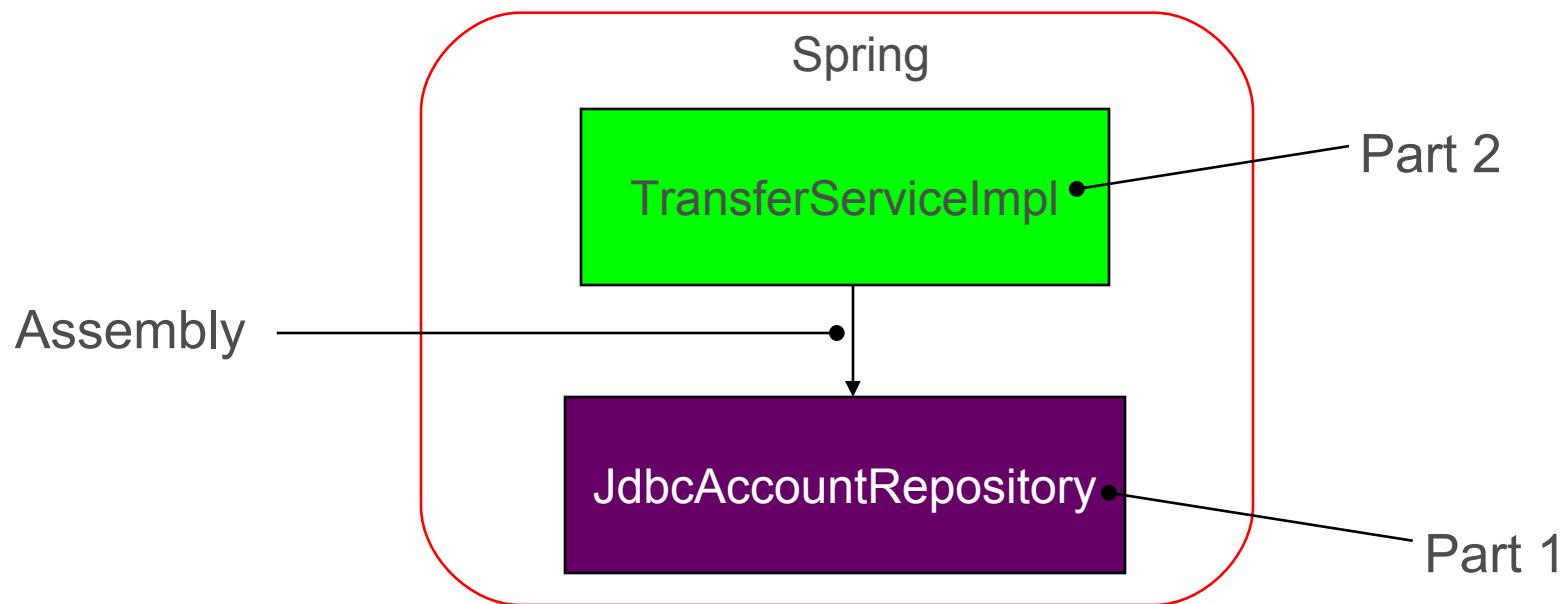
# Example: Money Transfer System



# 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* (POJOs)

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

Implements a service (business) interface

Part 1

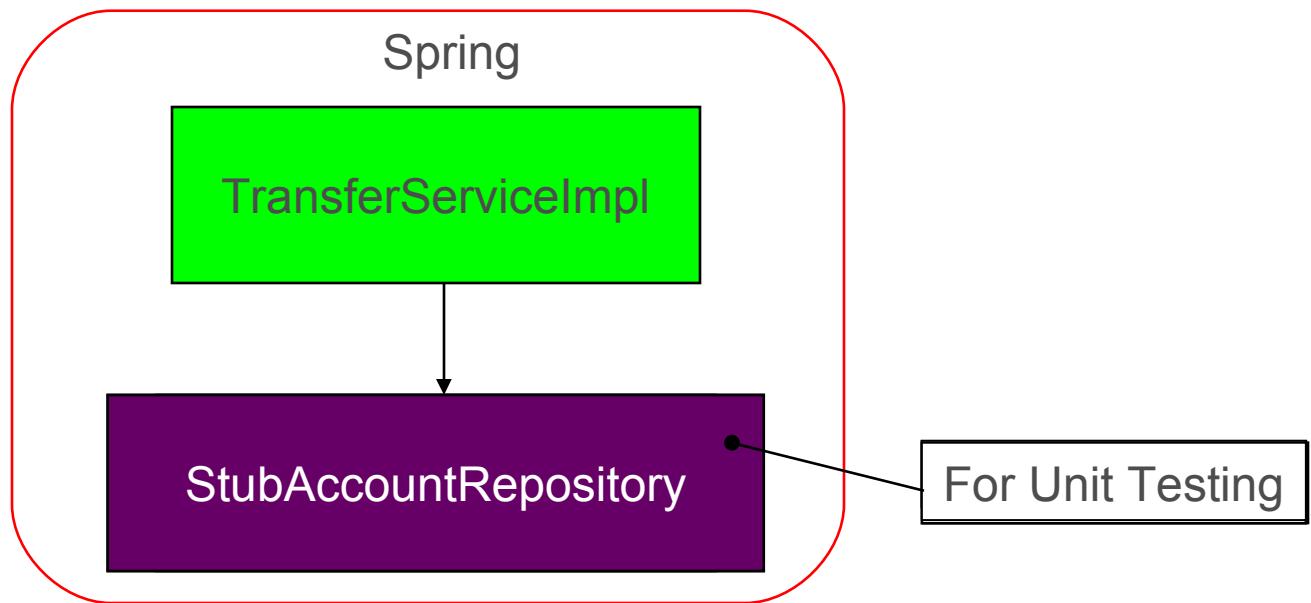
```
public class TransferServiceImpl implements TransferService {  
    private AccountRepository accountRepository;  
  
    public void setAccountRepository(AccountRepository ar) {  
        accountRepository = ar;  
    }  
    ...  
}
```

Part 2

Depends on an *interface*:

- conceals complexity of implementation;
- allows for swapping out implementation

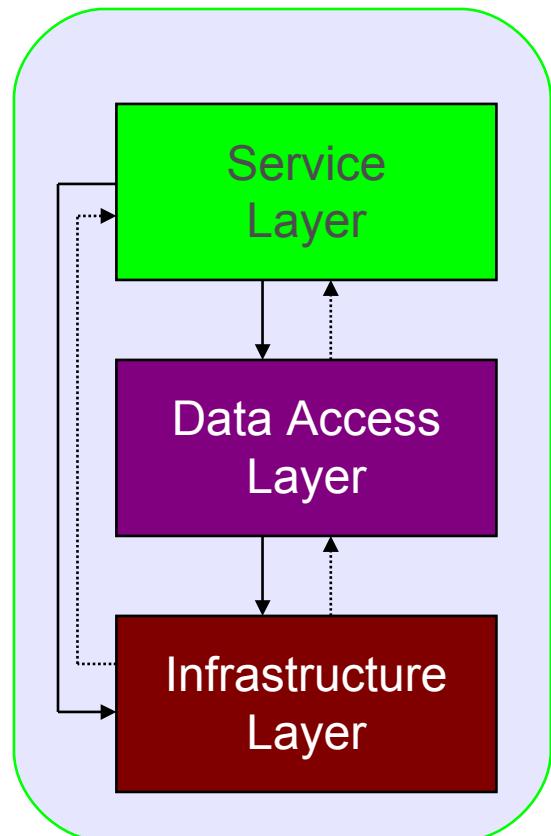
# Swapping Out Part Implementations



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

# Using a Layered Architecture

- Examples in this course assume a three-layered architecture
  - *Service Layer* (or application layer)
    - Exposes high-level application functions
    - Use-cases, business logic defined here
  - *Data access Layer*
    - Defines interface to the application's data repository (such as a Relational or NoSQL database)
  - *Infrastructure Layer*
    - Exposes low-level services to the other layers



*A classic “Separation of Concerns”*

# 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 an EJB alternative



# Spring Framework History

- Spring 2.0 (2006):
  - XML simplification, async JMS, JPA, AspectJ support
- Spring 2.5 (2007, last release 2.5.6)
  - Requires Java 1.4+ and supports JUnit 4
  - Annotation DI, @MVC controllers, XML namespaces
- Spring 3.x (3.2.17 released July 2016)
  - Environment & Profiles, @Cacheable, @EnableXXX ...
  - Requires Java 1.5+ and JUnit 4.7+
  - REST support, JavaConfig, SpEL, more annotations
- Spring 4.x (released Dec 2013)
  - Support for Java 8, @Conditional, Web-sockets
- Spring 5.x (2017)
  - Reactive programming focus

# 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:



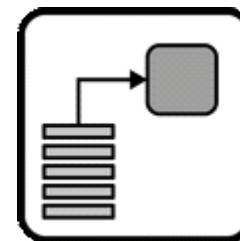
Web Interfaces



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, Cloud, Microservices
  - **Productivity:** Spring Boot, Spring Cloud Data Flow
  - and many more

# More on Spring's Ecosystem

- Visit <http://spring.io/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

# Objectives

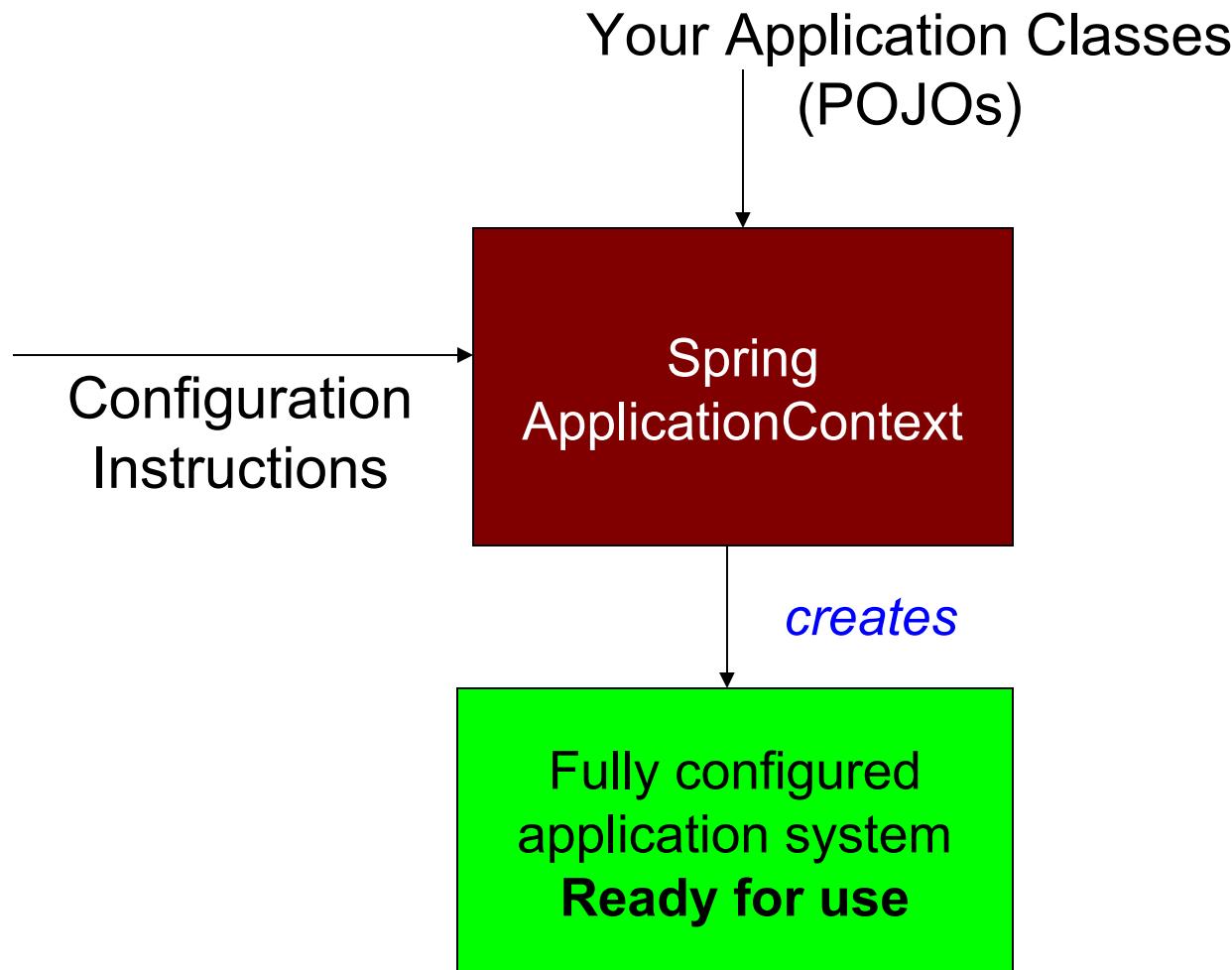
- After completing this lesson, you should be able to:
  - Define Spring Beans using Java code
  - Access Beans in the Application Context
  - Handle multiple Configuration files
  - Handle Dependencies between Beans
  - Explain and define Bean Scopes



# Topics in this session

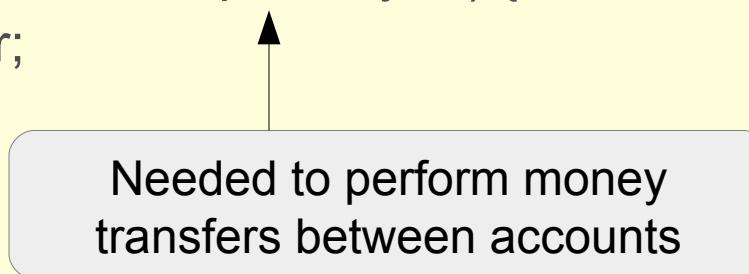
- **Spring quick start**
- Creating an application context
- Multiple Configuration Files
- Bean scope
- Lab

# How Spring Works



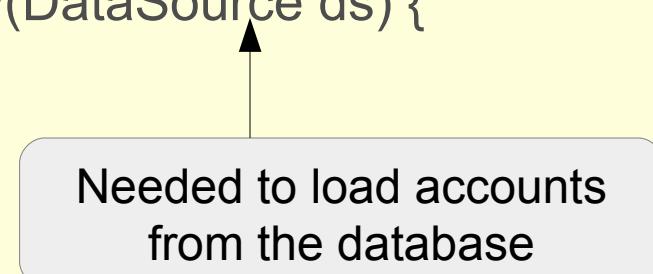
# Your Application Classes

```
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() {  
        BasicDataSource dataSource = new BasicDataSource();  
        dataSource.setDriverClassName("org.postgresql.Driver");  
        dataSource.setUrl("jdbc:postgresql://localhost/transfer" );  
        dataSource.setUsername("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
```

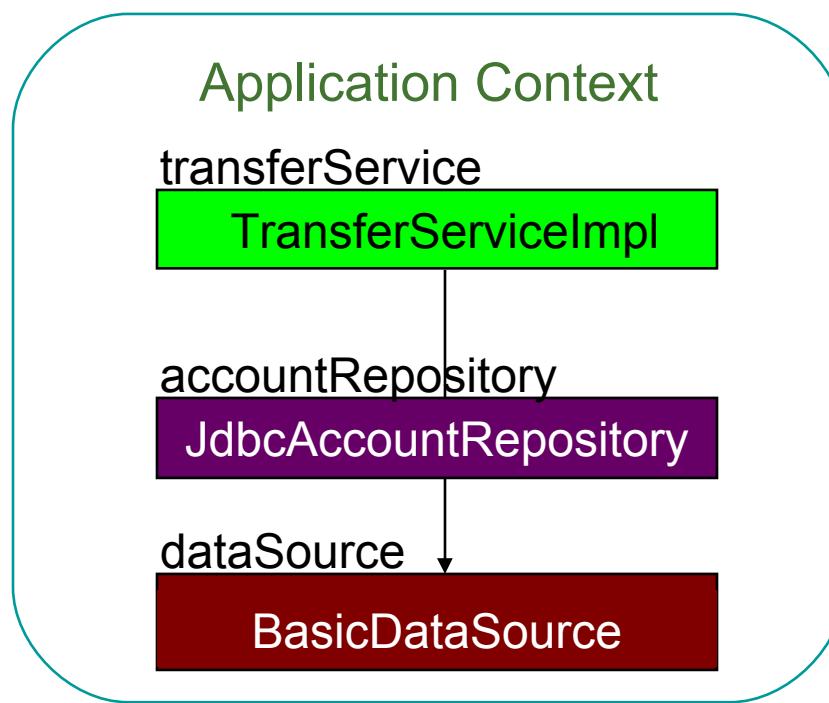
```
TransferService ts2 = context.getBean("transferService", TransferService.class);
```

```
// 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 )
```



# Bean Descriptions

- Allows you to provide helpful information about any bean

```
@Bean  
{@Description("Handles all transfer related use-cases")  
public TransferService transferService() { ... }}
```

```
@Bean  
{@Description("Provides access to data from the Accounts table")  
public AccountRepository accountRepository() { ... }}
```

```
@Bean  
{@Description("Data-source for the underlying RDB we are using")  
public DataSource dataSource() { ... }}
```

# Quick Start Summary

- Spring separates application configuration from application objects
- Spring manages your application objects
  - Creating them in the correct order
  - Ensuring they are fully initialized before use
- Each bean is given a unique id / name
  - Should reflect service or role the bean provides to clients
  - Bean ids 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

# ApplicationContext Example

## Instantiating Within a System (Integration) Test

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

Bootstraps the system to test

Tests the system

Using JUnit 5 – JUnit 3 & 4 or TestNG also supported

# 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 {  
    ...  
}
```

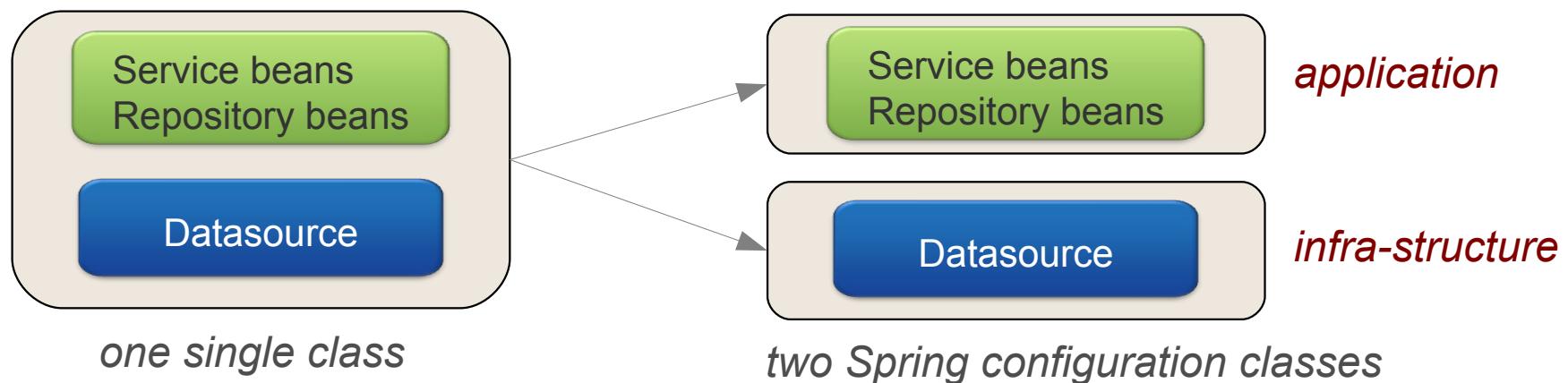


```
@Configuration  
public class InfrastructureConfig {  
    ...  
}
```

```
@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()  
    {  
        BasicDataSource dataSource = new BasicDataSource();  
        dataSource.setDriverClassName("org.postgresql.Driver");  
        dataSource.setUrl("jdbc:postgresql://localhost/transfer" );  
        dataSource.setUsername("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;
```

*application beans*

```
    @Bean public TransferService transferService() {  
        return new TransferServiceImpl( accountRepository() );  
    }
```

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

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

*infrastructure bean*

# Referencing Beans Defined in Another File

- Use `@Autowired` to inject a bean defined elsewhere

```
@Configuration  
public class AppConfig {  
    private final DataSource dataSource;  
  
    @Autowired  
    public AppConfig(DataSource ds) {  
        this.dataSource = ds;  
    }  
  
    @Bean  
    public AccountRepository accountRepository() {  
        return new JdbcAccountRepository( dataSource );  
    }  
}
```

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

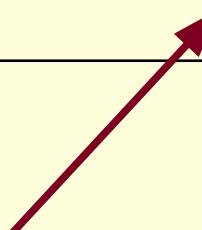
Pre Spring 4.3: Auto-wire a property setter, can't use a constructor

# Referencing Dependencies Via Arguments

- Alternative: Define @Bean method arguments
  - Spring finds bean that matches type & injects the argument

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

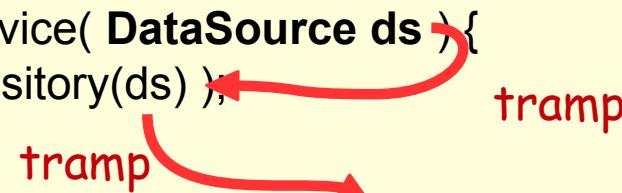
```
@Configuration  
@Import(ApplicationConfig.class)  
public class InfrastructureConfig {  
    @Bean public DataSource dataSource() {  
        DataSource ds = new BasicDataSource();  
        ...  
        return ds;  
    }  
}
```



# ... But Avoid “Tramp Data”

```
@Configuration  
public class ApplicationConfig {  
    @Bean public AccountService accountService( DataSource ds ) {  
        return new accountService( accountRepository(ds) );  
    }  
  
    @Bean public AccountRepository accountRepository( DataSource ds ) {  
        return new accountRepository( dataSource );  
    }  
}
```

Bad: dataSource is a “tramp”!



tramp  
tramp

```
@Configuration  
public class ApplicationConfig {  
    @Bean public AccountService accountService( AccountRepository repo ) {  
        return new accountService( repo );  
    }  
  
    @Bean public AccountRepository accountRepository( DataSource ds ) {  
        return new accountRepository( ds );  
    }  
}
```

Better: Pass *actual* dependency

# Beware Duplicate Beans

Use `@Order` annotation to control which order `@Bean` methods run (since Spring 4.2)

- It is *not* illegal to define the same bean more than once
  - You get the last bean Spring sees defined

```
@Configuration  
public class Config1 {  
    @Bean  
    public String example() {  
        return new String("example1");  
    }  
}
```

```
@Import({ Config1.class, Config2.class })  
public class TestApp {  
    public static void main(String[] args) {  
        ApplicationContext context = SpringApplication.run(TestApp.class);  
        System.out.println("Id=" + context.getBean("example"));  
    }  
}
```

```
@Configuration  
public class Config2 {  
    @Bean  
    public String example() {  
        return new String("example2");  
    }  
}
```

Console output is `Id=example2`

# Topics in this session

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

# Bean Scope: default

service1 == service2

- Default scope is *singleton*

```
@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");  
assert service1 == service2; // True – same object
```

service1 != service2

# Bean Scope: prototype

- Scope "prototype"
  - New instance created every time bean is referenced

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

@Scope(scopeName="prototype")

```
AccountService service1 = (AccountService) context.getBean("accountService");  
AccountService service2 = (AccountService) context.getBean("accountService");  
assert service1 != service2; // True – different objects
```

TWO instances

# Common Spring Scopes

- The most commonly used scopes are:

**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 only**

**request**

A new instance is created once per request – **web environment only**

# Other Scopes

- Spring has other more specialized scopes
  - Web Socket scope
  - Refresh Scope
  - Thread Scope (defined but not registered by default)
- Custom scopes (rarely)
  - You define a factory for creating bean instances
  - Register to define a custom scope name
- All are outside the scope of this course

# 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

# Objectives

- After completing this lesson, you should be able to:
  - Use external Properties to control Configuration
  - Demonstrate the purpose of Profiles
  - Use the Spring Expression Language (SpEL)
  - Explain how Spring Java Configuration implements *Singleton Beans*



# Topics in this session

- **External Properties**
- Profiles
- Spring Expression Language
- Singleton “Magic”

# Setting property values

- Consider this bean definition from the last chapter:

```
@Bean  
public DataSource dataSource() {  
    BasicDataSource 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 DbConfig {  
    private static final String DB_DRIVER = "db.driver";  
    private static final String DB_URL = "db.url";  
    private static final String DB_USER = "db.user";  
    private static final String DB_PWD = "db.password";
```

Property names

```
@Autowired Environment env;
```

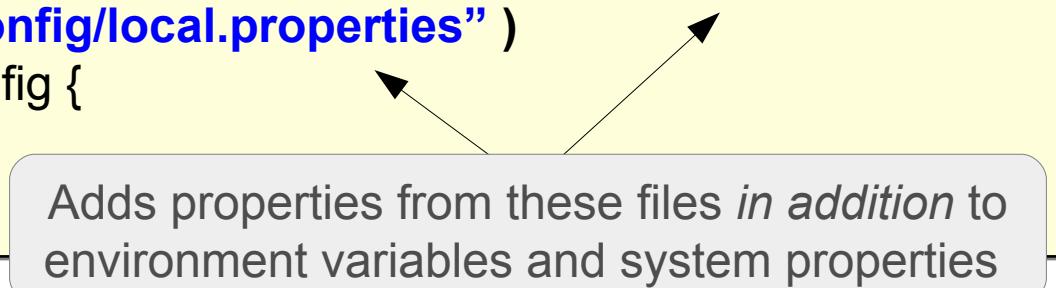
```
@Bean public DataSource dataSource() {  
    BasicDataSource ds = new BasicDataSource();  
    ds.setDriverClassName( env.getProperty( DB_DRIVER ) );  
    ds.setUrl( env.getProperty( DB_URL ) );  
    ds.setUser( env.getProperty( DB_USER ) );  
    ds.setPassword( env.getProperty( DB_PWD ) );  
    return ds;  
}
```

Fetch property values from environment

# Property Sources

- Environment obtains values from “property sources”
  - *Environment Variables* and *Java System Properties* always populated automatically
  - **@PropertySource** contributes *additional* properties
  - Available resource prefixes: **classpath:** **file:** **http:**

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



Adds properties from these files *in addition* to environment variables and system properties

# Accessing Properties using @Value

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

Convenient alternative to explicitly using Environment

BUT: How are these \${...} variables resolved? Next slide ...

# Evaluating \${ . . } Variables

Spring Boot does this for you automatically

- \${...} variables are evaluated by a dedicated Spring bean
  - The `PropertySourcesPlaceholderConfigurer`
  - **Note:** make this a `static` bean
    - Ensures \${..} placeholder expressions are evaluated *before* any beans are created that might use them

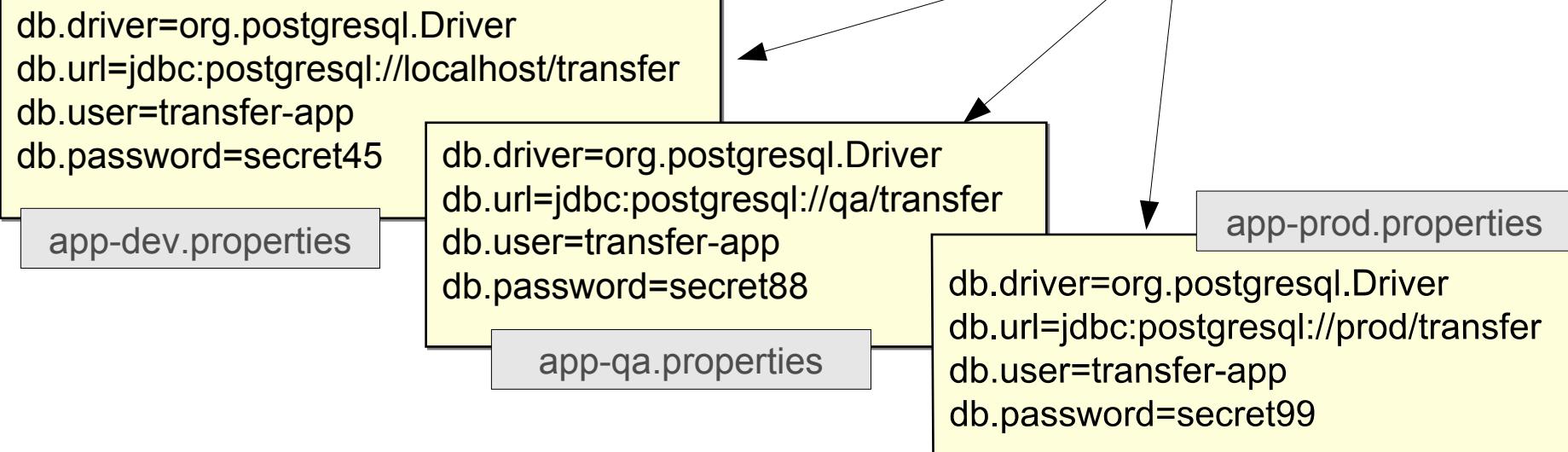
```
@Bean  
public static PropertySourcesPlaceholderConfigurer pspc() {  
    return new PropertySourcesPlaceholderConfigurer();  
}
```

\${..} placeholders are *not resolved unless* this bean declared

# `${...}` Placeholders

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

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

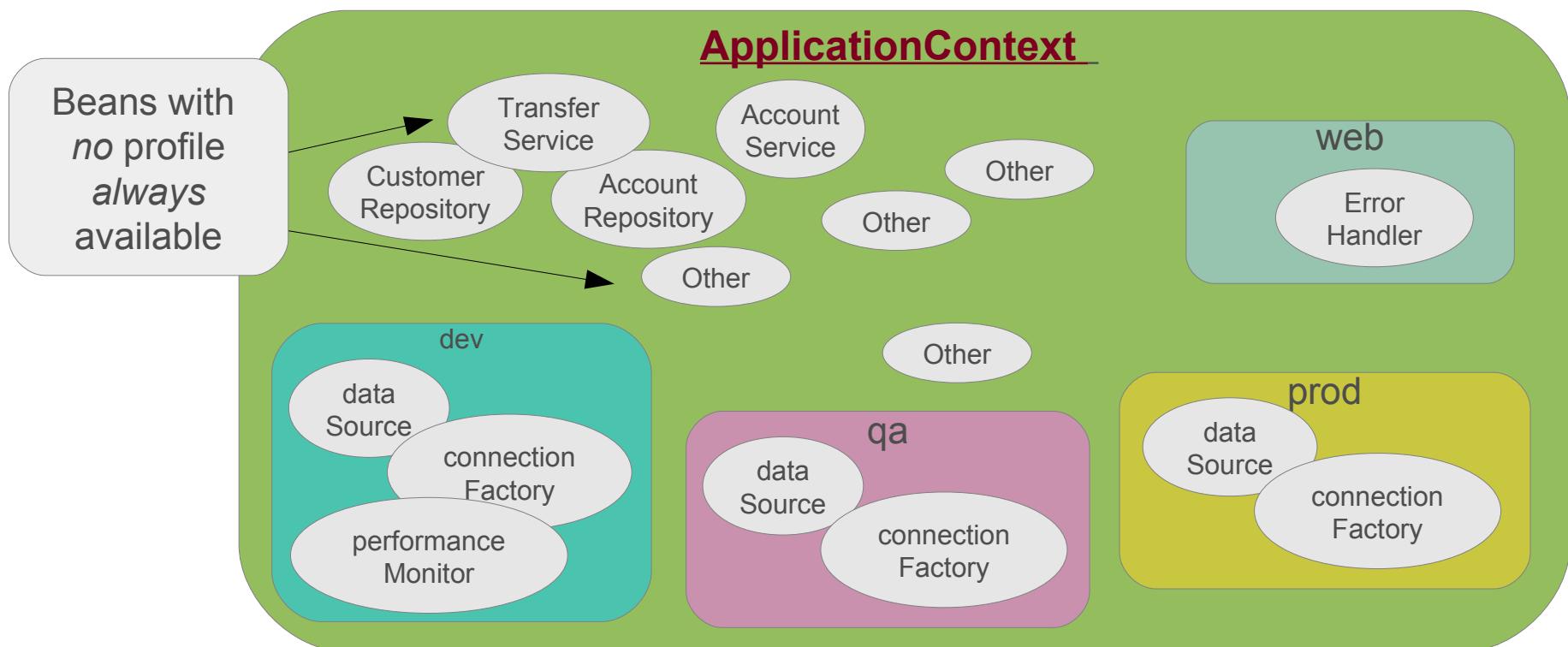


# Topics in this session

- External Properties
- **Profiles**
- Spring Expression Language
- Singleton “Magic”

# 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();  
    }  
    ...  
}
```

H2, Derby are also supported

# Defining Profiles - 2

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

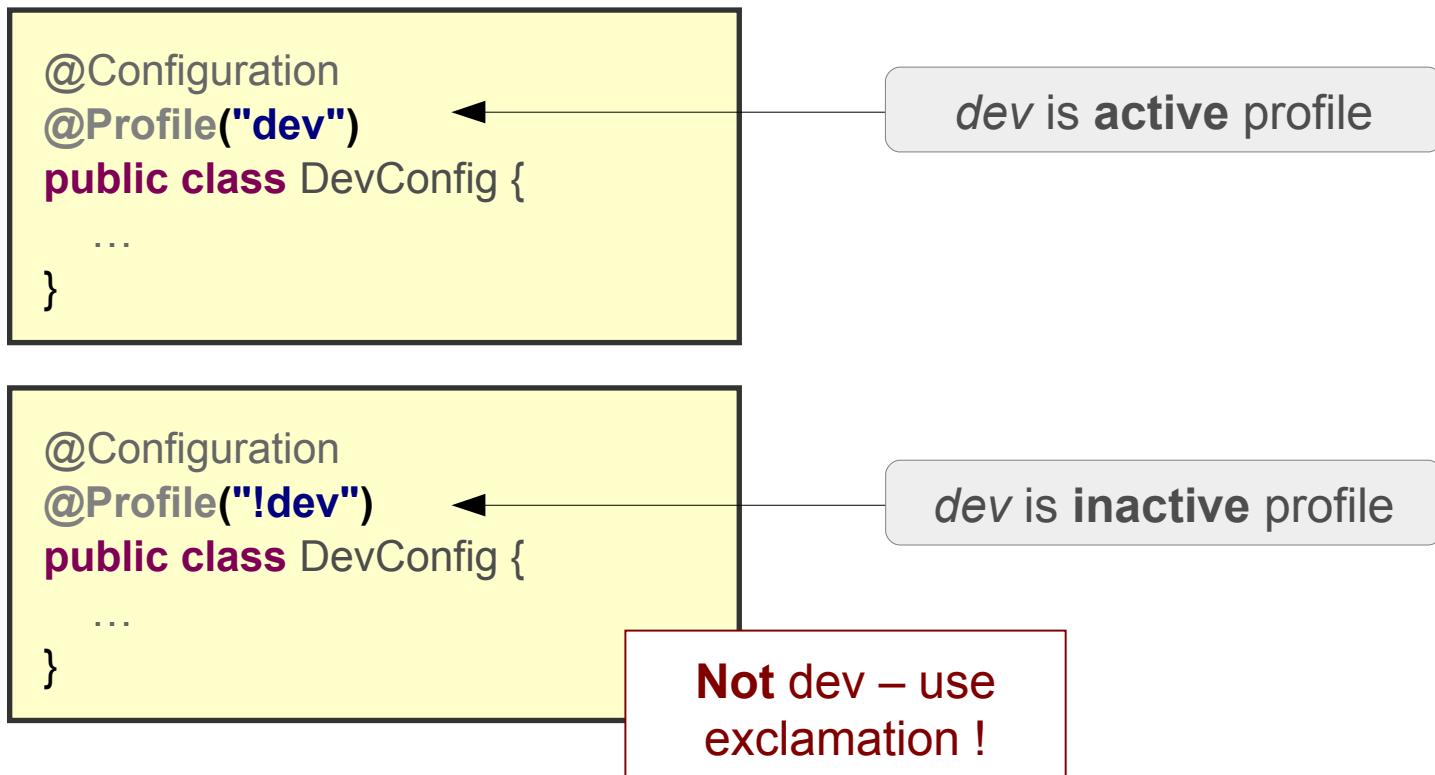
```
@Configuration  
public class DataSourceConfig {  
  
    @Bean(name="dataSource")  
    @Profile("dev")  
    public DataSource dataSourceForDev() {  
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();  
        return builder.setName("testdb") ...  
    }  
  
    @Bean(name="dataSource")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

# Defining Profiles - 3

- Beans when a profile is *not* active



# Ways to Activate Profiles

- Profiles must be activated at run-time

- System property via command-line

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

- System property programmatically

```
System.setProperty("spring.profiles.active", "dev,jpa");  
SpringApplication.run(AppConfig.class);
```

- Integration Test *only*: `@ActiveProfiles` (later section)

# 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")  
class DevConfig { ... }
```

```
@Configuration  
@Profile("prod")  
@PropertySource ("prod.properties")  
class ProdConfig { ... }
```

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

dev.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
- **Spring Expression Language**
- Singleton “Magic”

# Spring Expression Language

- SpEL for short
  - Inspired by the Expression Language used in Spring WebFlow
  - Based on Unified Expression Language used by JSP and JSF
- Pluggable/extendable by other Spring-based frameworks



This is just a brief introduction, for full details see  
<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html>

# SpEL examples – Using @Value

```
@Configuration  
class TaxConfig  
{  
    @Value("#{ systemProperties['user.region'] }") String region;  
  
    @Bean  
    public TaxCalculator taxCalculator1() {  
        return new TaxCalculator( region );  
    }  
  
    @Bean  
    public TaxCalculator taxCalculator2  
        (@Value("#{ systemProperties['user.region'] }") String region, ... ) {  
        return new TaxCalculator( region );  
    }  
    ...  
}
```

**Option 1:** Set an attribute then use it

**Option 2:** Pass as a bean method argument

# SpEL – Accessing Spring Beans

```
class StrategyBean {  
    private KeyGenerator gen = new KeyGenerator.getInstance("Blowfish");  
    public KeyGenerator getKeyGenerator() { return gen; }  
}
```

```
@Configuration  
class StrategyConfig  
{  
    @Bean public StrategyBean strategyBean() {  
        return new StrategyBean();  
    }  
}
```

```
@Configuration  
class AnotherConfig  
{  
    @Value("#{strategyBean.keyGenerator}") KeyGenerator kgen;  
    ...  
}
```

# Accessing Properties

- Can access properties via the *environment*
  - These are equivalent

```
@Value("${daily.limit}")  
int maxTransfersPerDay;
```

```
@Value("#{environment['daily.limit']}")  
int maxTransfersPerDay;
```

- Properties are Strings
  - May need to cast in expressions

```
@Value("#{new Integer(environment['daily.limit']) * 2}")  
@Value("#{new java.net.URL(environment['home.page']).host}")
```

# Fallback Values

- Providing a fall-back value
  - If `daily.limit` undefined, use colon :

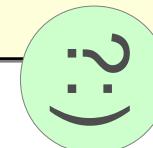
```
@Autowired  
public TransferServiceImpl(@Value("${daily.limit : 100000}") int max) {  
    this.maxTransfersPerDay = max;  
}
```

- For SpEL, use the “Elvis” operator ?:

```
@Autowired  
public setLimit(@Value("#{environment['daily.limit'] ?: 100000}") int max) {  
    this.maxTransfersPerDay = max;  
}
```

Equivalent operators

`x ?: y` is short for `x != null ? x : y`



*Elvis lives!*

# SpEL

- EL Attributes can be:
  - Spring beans (like *strategyBean*)
  - Implicit references
    - Spring's *environment*, *systemProperties*, *systemEnvironment* available by default
    - Others depending on context
- SpEL allows to create custom functions and references
  - Widely used in Spring projects
    - Spring Security, Spring WebFlow
    - Spring Batch, Spring Integration
  - Each may add *their own* implicit references

# Topics in this session

- External Properties
- Profiles
- Spring Expression Language
- **Singleton “Magic”**

# 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() );  
}
```



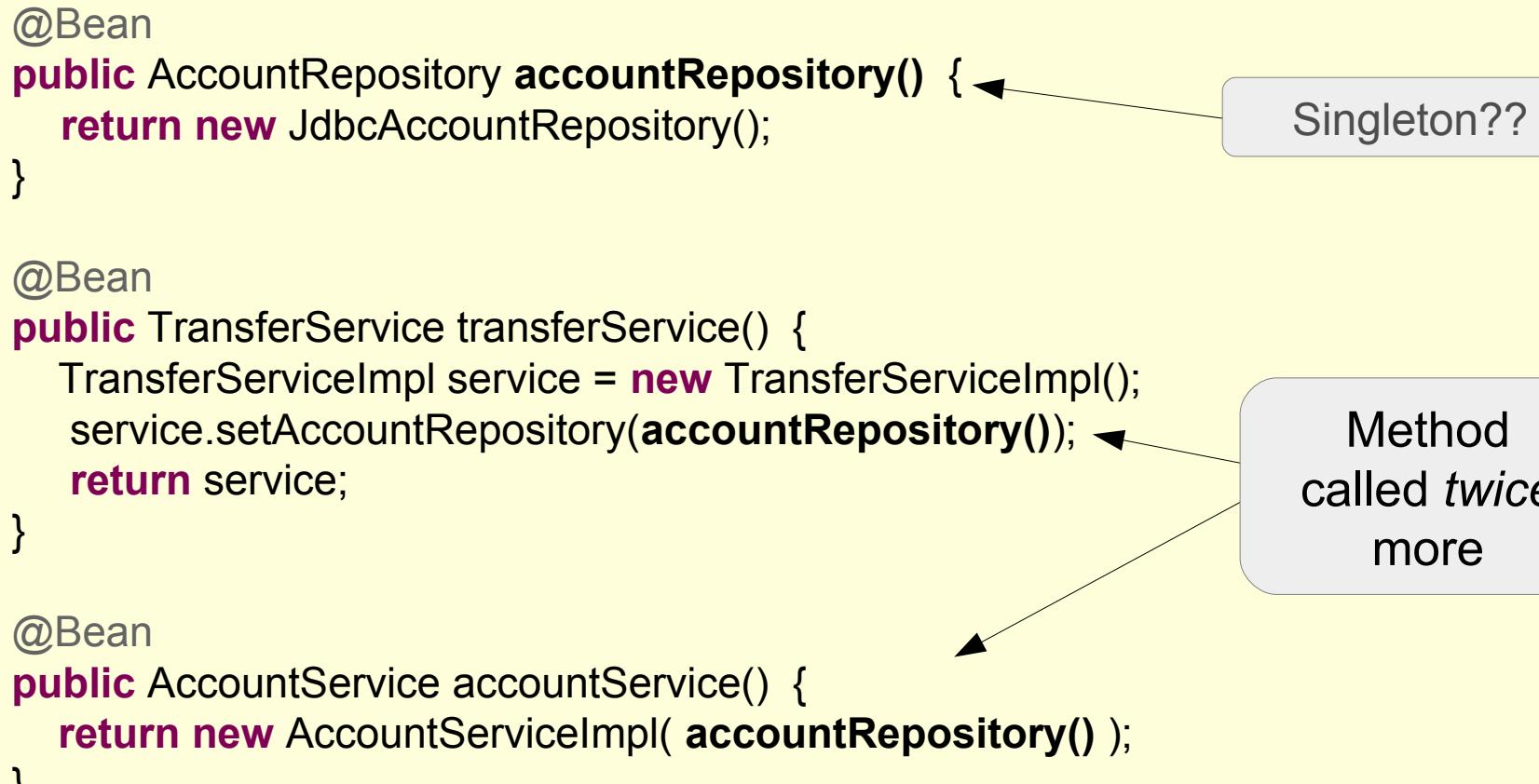
1. Method call?

2. New instance?

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

# Working with Singletons

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



**HOW IS IT POSSIBLE?**

# Singletons Require “Magic”

- At startup time, a *subclass* is created
  - Subclass performs *scope-control*
    - Only calls *super* on *first* invocation of singleton bean method
    - Singleton instance is cached by the *ApplicationContext*

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

↑ *inherits from*

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

# Inheritance-based Subclasses

- Child class is the entry point

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



Java Configuration uses *cglib* for inheritance-based subclasses

# Summary

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

# Annotation-Driven Configuration

Annotations for Dependency Injection and Interception

Component scanning and auto-injection

# Objectives

- After completing this lesson, you should be able to:
  - Explain and use Annotation-based Configuration
  - Discuss Best Practices for Configuration choices
  - Use `@PostConstruct` and `@PreDestroy`
  - Explain and use Spring's “Stereotype” Annotations



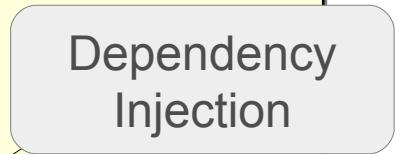
# Topics in this Session

- Fundamentals
  - **Annotation-based Configuration**
  - Best practices: when to use what?
  - `@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

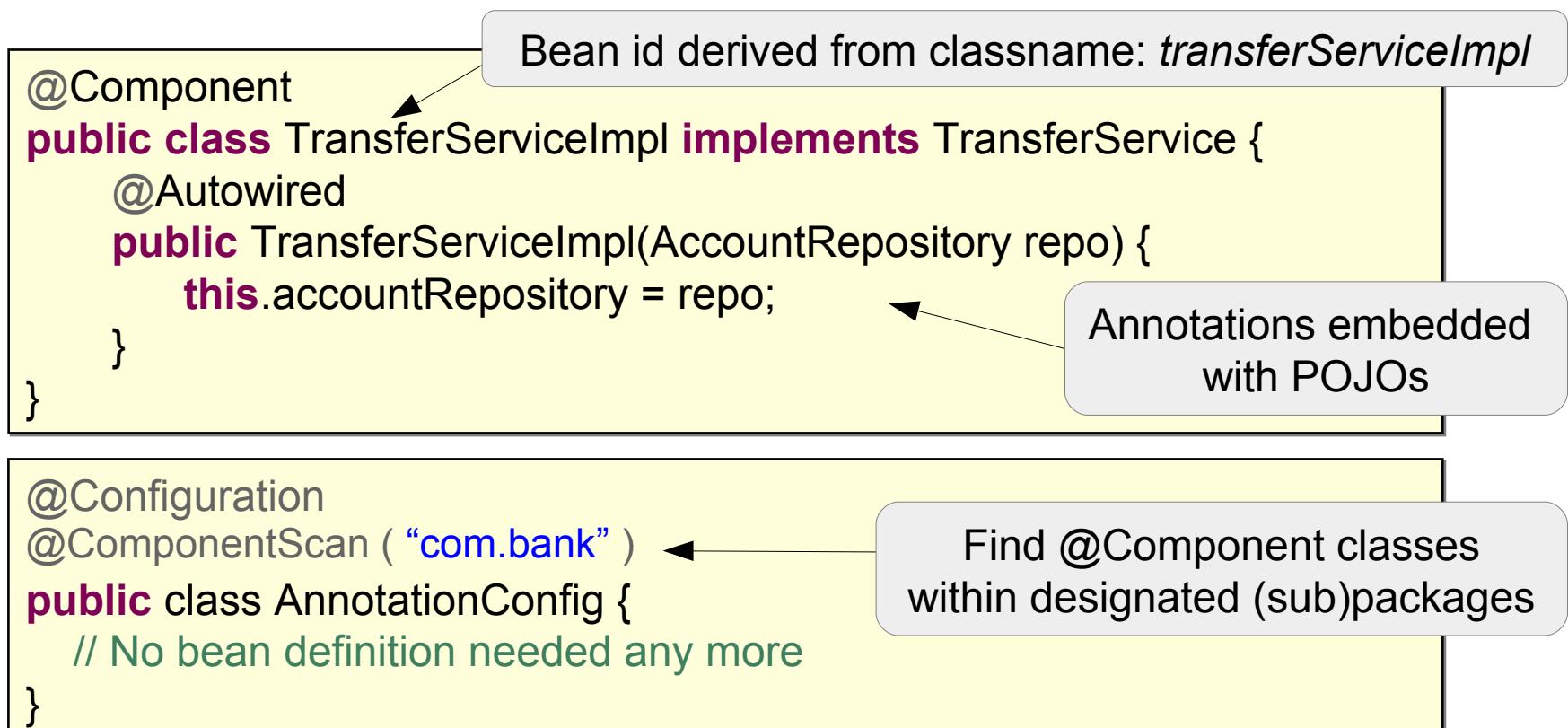
```
@Configuration  
public class TransferModuleConfig {  
  
    @Bean public TransferService transferService() {  
        return new TransferServiceImpl( accountRepository() );  
    }  
  
    @Bean public AccountRepository accountRepository() {  
        ...  
    }  
}
```



A callout bubble containing the text "Dependency Injection" has an arrow pointing to the line of code where the TransferService bean is instantiated, specifically the argument of the TransferServiceImpl constructor.

# After - *Implicit* Configuration

- Annotation-based configuration *within* bean-class



# 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

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

Even when field is private!!  
– *but* hard to unit test, see URL

# @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

# Java 8 Optional<T>

- Another way to inject optional dependencies
  - `Optional<T>` introduced to reduce null pointer errors

```
@Autowired(required=false)  
AccountService accountService;  
  
public void doSomething() {  
    if (accountService != null) {  
        // do something  
    }  
}
```

```
@Autowired  
Optional<AccountService> accountService;  
  
public void doSomething() {  
    accountService.ifPresent( s -> {  
        // s is the AccountService instance,  
        // use s to do something  
    });  
}
```

Note the use of the lamda

# Constructor vs Setter Dependency Injection

- Spring doesn't care – can use either
  - But which is best?

Constructors	Setters
Mandatory dependencies	Optional / changeable dependencies
Immutable dependencies	Circular dependencies
Concise (pass several params at once)	Inherited automatically
	If constructor needs too many params

- Follow the same rules as standard Java
  - Be consistent across your project team
  - Many classes use both

# Autowiring and Disambiguation – 1

- 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 – 2

- 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)

# Autowiring and Disambiguation – 3

- Autowired resolution rules
  - Look for unique bean of required *type*
  - Use @Qualifier if supplied
  - Try to find a matching bean by *name*
- Example
  - We have multiple *Queue* beans
  - Spring finds bean with id matching what is being set: “**ack**”

```
@Autowired  
public myBean(Queue ack) {  
    ...  
}
```

```
@Autowired  
public void setQueue(Queue ack) {  
    ...  
}
```

```
@Autowired  
private Queue ack;
```

Looks for Queue bean with id = “**ack**”

# 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*

# Using @Value to set Attributes

- Constructor-injection

Can use \$ variables or SpEL

```
@Autowired  
public TransferServiceImpl(@Value("${daily.limit}") int max) {  
    this.maxTransfersPerDay = max;  
}
```

- Method-injection

```
@Autowired  
public void setDailyLimit(@Value("${daily.limit}") int max) {  
    this.maxTransfersPerDay = max;  
}
```

- Field-injection

```
@Value("#{environment['daily.limit']}")  
int maxTransfersPerDay;
```

Not private so we can initialize in a unit-test

# Delayed Initialization

- Beans normally created on startup when application context created
- Lazy beans created first time used
  - When dependency injected
  - By `ApplicationContext.getBean` methods
- Useful if bean's dependencies *not* available at startup

```
@Lazy @Component  
public class MailService {  
    public MailService(@Value("smtp:...") String url) {  
        // connect to mail-server  
    }  
    ...  
}
```

Mail-server may not be running  
when this process starts up

# 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*

```
@Configuration
public class TransferConfiguration
    @Bean(name="transferService")
    @Scope("prototype")
    @Profile("dev")
    @Lazy(false)
    public TransferService tsvc() {
        return
            new TransferServiceImpl(
                accountRepository());
    }
}
```

*Java Configuration*

# Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - **Best practices: when to use what?**
  - @PostConstruct and @PreDestroy
  - Stereotypes and meta annotations
- Lab
- Advanced features
  - @Resource
  - Standard annotations (JSR 330)

# About 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" } )
```

# 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 your own 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)

# Mixing Java Config and Annotations

- You can mix and match in many ways
- Common approach:
  - Use annotations whenever possible
    - Your classes
  - But still use Java Configuration for
    - Third-party beans that aren't annotated
    - Legacy code that can't be changed

# Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - **@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 all injection

```
    @PreDestroy
```

```
    void flushCache() { }
```

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*

# About @PostConstruct & @PreDestroy

- Beans are created in the usual ways:
  - Returned from @Bean methods
  - Found and created by the component-scanner
- Spring then invokes these methods *automatically*
  - During bean-creation process
- These are not Spring annotations
  - Defined by JSR-250, part of Java since Java 6
  - In `javax.annotation` package
  - Supported by Spring, *and* by JEE

# @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(); //... }  
}
```

1      2



# @PreDestroy

- Called when a *ConfigurableApplicationContext* is *closed*
  - Useful for releasing resources & 'cleaning up'
  - Not** called for prototype beans

```
ConfigurableApplicationContext context = ...;  
// Triggers call of all @PreDestroy annotated methods  
context.close();
```

Causes Spring to invoke this method

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

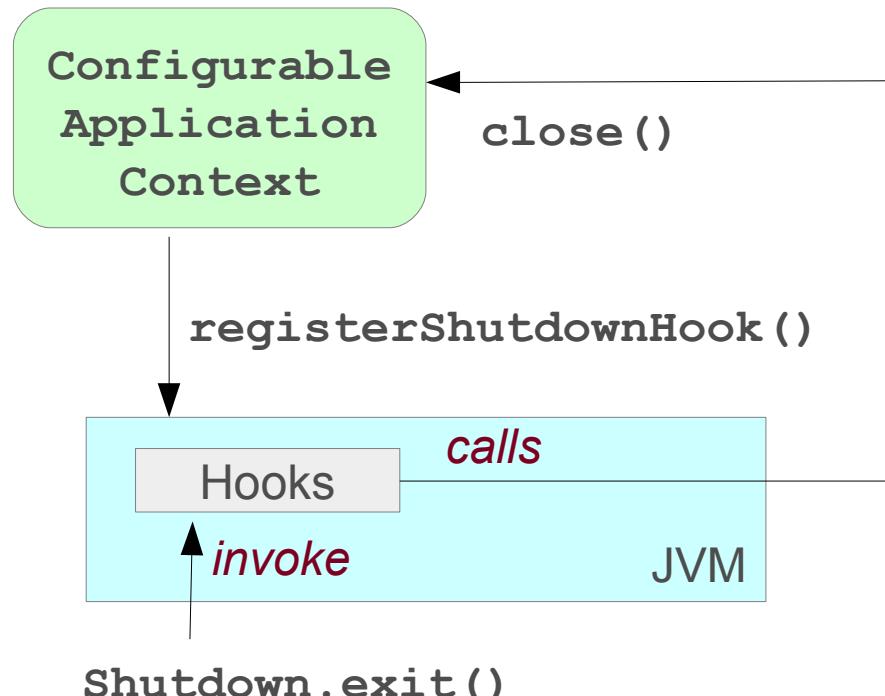
# Register a Shutdown hook

- `ConfigurableApplicationContext.registerShutdownHook()`
  - “Hook” automatically called when JVM shuts down
    - Calls `ConfigurableApplicationContext.close()`
- `SpringApplication.run()`
  - Does this automatically
  - Returns a `ConfigurableApplicationContext`

```
ConfigurableApplicationContext context = SpringApplication.run(...);  
// Registers the shutdownHook for you
```

# Use a JVM Shutdown Hook

- Shutdown hooks
  - Automatically run when JVM shuts down
- `SpringApplication.run`
  - Does this automatically
  - Returns a `ConfigurableApplicationContext`



```
ConfigurableApplicationContext context = SpringApplication.run(...);  
// Registered the shutdownHook for you
```

# Lifecycle Methods via @Bean

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

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

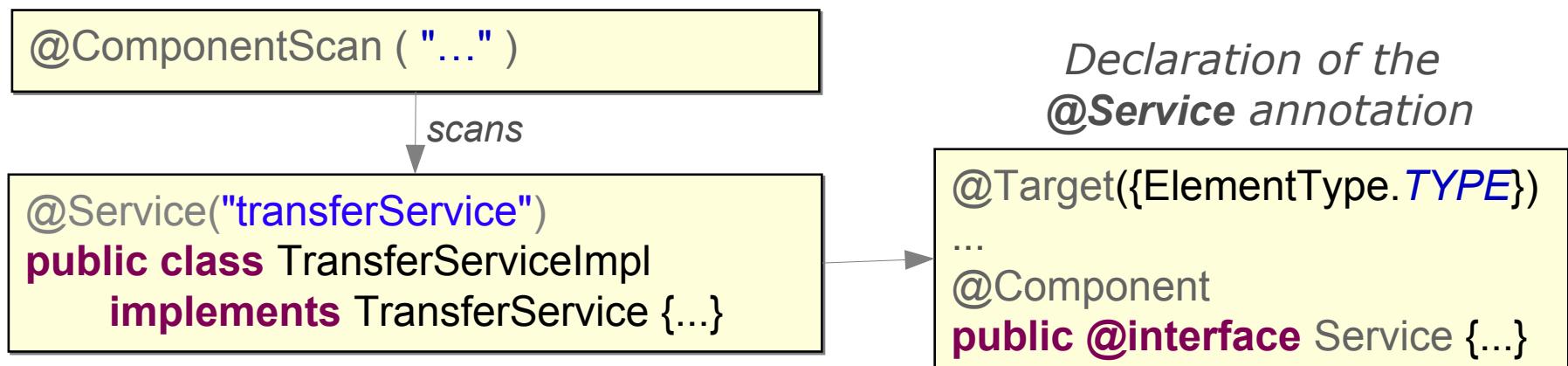
- Common Usage:
  - **@PostConstruct/@PreDestroy** for your own classes
  - **@Bean** properties for classes you didn't write and can't annotate

# Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - `@PostConstruct` and `@PreDestroy`
  - **Stereotypes and meta annotations**
- Lab
- Advanced features
  - `@Resource`
  - Standard annotations (JSR 330)

# Stereotype Annotations

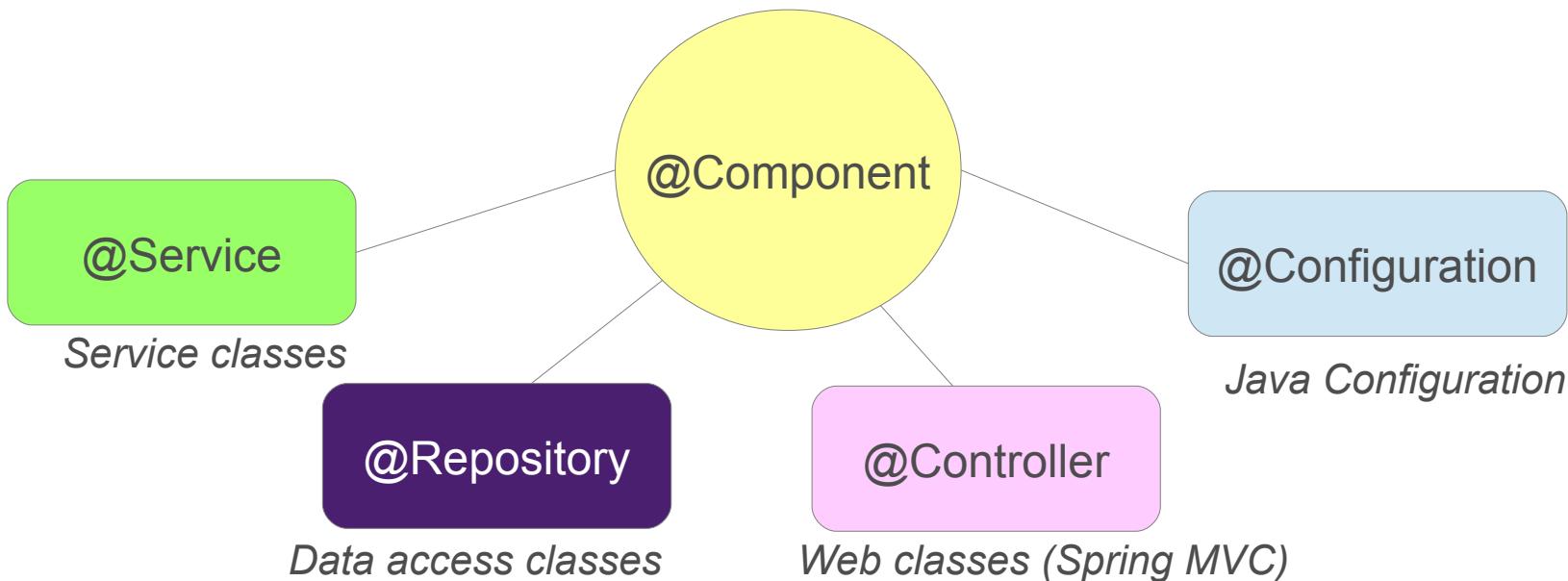
- Component scanning also checks for annotations that are themselves annotated with @Component
  - So-called *stereotype annotations*



@Service annotation is part of the Spring framework

# Predefined Stereotype Annotations

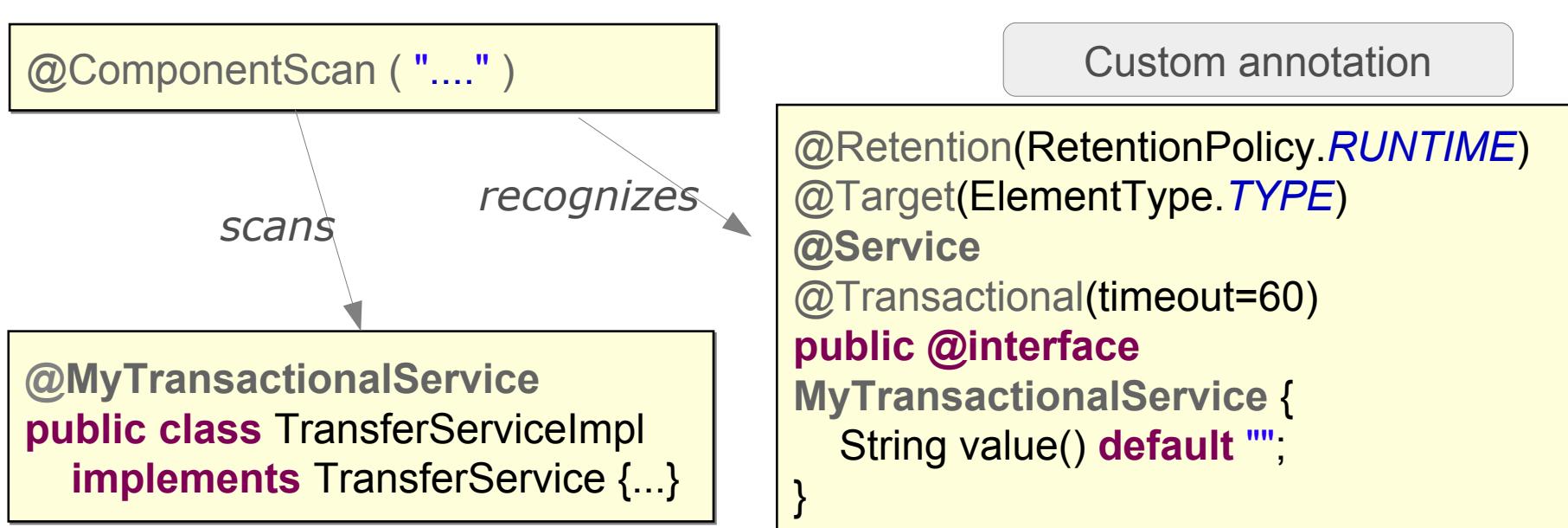
- Spring framework stereotype annotations



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



# Summary

- Spring beans can be defined:
  - Explicitly using `@Bean` methods
  - Implicitly using `@Component` and component-scanning
- Most applications use both
  - Implicit for your classes
  - Explicit for the rest
- Can perform initialization and clean-up
  - Use `@PostConstruct` and `@PreDestroy`
- Use Spring's stereotypes and/or define your own meta annotations

# Lab

Using Spring Annotations  
To Configure and Test an application

**Coming Up:** Other Annotations for Dependency Injection

# Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - `@PostConstruct` and `@PreDestroy`
  - Stereotypes and meta annotations
- Lab
- Advanced features
  - `@Resource`
  - Standard annotations (JSR 330)

**Note:** Advanced topics *not* required for Certification

# 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*

**@Autowired:** type *then* name

**@Resource:** name *then* type

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

# Topics in this Session

- Fundamentals
  - Annotation-based Configuration
  - Best practices: when to use what?
  - `@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

# Factories in Spring

## Introducing Factory Beans

Spring's *FactoryBean* interface

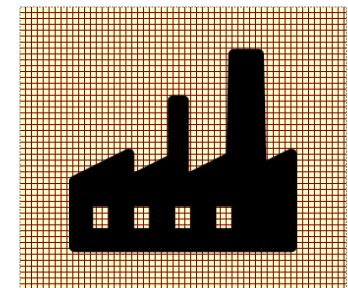
# Objectives

- After completing this lesson, you should be able to:
  - Describe the Factory Pattern
  - Explain purpose of Factory Beans and be able to use them



# Topics in this session

- **Defining Bean using the Factory Pattern**
- Spring's FactoryBean Interface



# Factory Pattern

- Object creation more complicated than a simple use of operator `new`
  - Different implementations
    - Depending on platform, configuration, user, ...
- Wrap all creation code in a dedicated method or class
  - A “factory” for creating objects

```
public class AccountServiceFactory {  
    public AccountService getInstance() {  
        // Conditional logic – for example: selecting the right  
        // implementation or sub-class of AccountService to create  
        return accountService;  
    }  
}
```

# @Bean Methods are Factories

- Spring's @Bean methods are factory methods
  - They create Spring Beans

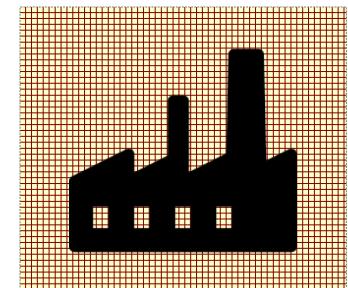
```
@Configuration
public class AccountServiceFactory {
    @Bean
    public AccountService accountService() {
        // Conditional logic – for example: selecting the right
        // implementation or sub-class of AccountService to create
        return accountService;
    }
}
```

# Spring *is* a Factory

- Spring creates Spring Beans
  - No matter how you define the beans
- Bean definition options
  - **Java Configuration:** `@Configuration` classes
  - **Annotation-Based:** `@Component`, `@Autowired` and component-scanning
  - **XML Configuration:** `<bean>` elements
    - Common in existing applications
    - Not covered in this course
    - See optional sections at end of notes

# Topics in this session

- Defining Bean using the Factory Pattern
- **Spring's FactoryBean Interface**



# Complex Bean Instantiation

**Note:** XML is not in the certification exam

- No Conditional configuration in Spring XML
  - <bean> definitions are declarative only
  - Unlike @Bean methods – can use *any* Java you need
- Instead Spring XML relied on the *Factory Pattern*
  - Use a factory to create the bean(s) we want
    - Implement Spring's **FactoryBean** interface
  - Put *any* complex Java code needed in factory's internal logic
  - *Spring FactoryBeans used in Java Configuration also*

# The Spring FactoryBean interface



- Originally invented as a fall-back for complex configuration in XML
  - Used long before `@Bean` methods introduced

```
interface FactoryBean<T> {  
    // The factory method  
    public T getObject() throws Exception;  
  
    // Is this a singleton instance or not?  
    public boolean isSingleton() { return true; }  
  
    // What type of object is this – easier than introspecting T  
    public Class<?> getObjectType() { return AccountService.class; }  
}
```

**Note:** Some Java Configuration also uses factory beans

# FactoryBean Example



```
public class AccountServiceFactoryBean
    implements FactoryBean <AccountService>
{
    public AccountService getObject() throws Exception {
        // Conditional logic – for example: selecting the right
        // implementation or sub-class of AccountService to create
        return accountService;
    }

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

# FactoryBeans with Java Configuration

- Convention: Spring calls `getObject()` automatically

```
@Configuration  
public class ServiceConfig {  
  
    @Bean  
    public AccountServiceFactoryBean accountService() {  
        return new AccountServiceFactoryBean();  
    }  
  
    @Bean  
    public CustomerService customerService(AccountService accountService) {  
        return new CustomerService(accountService);  
    }  
}
```

*getObject() called by Spring *internally**

*creates*

*Do not call getObject() yourself  
Spring often does additional setup *internally* first*

# Factory Beans in Spring

- FactoryBeans are widely used within Spring
  - EmbeddedDatabaseFactoryBean
  - JndiObjectFactoryBean
    - One option for looking up JNDI objects
  - Creating Remoting proxies
  - Creating Caching proxies\*\*
  - For configuring data access technologies\*\*
    - JPA, Hibernate or MyBatis

# Summary

- Factory Beans
  - Important configuration device
  - Understand how `getObject()` works

- XML configuration has existed in Spring since first release
  - Many existing applications use it
  - Two optional sections at end of course
    - *XML Configuration Basics*
    - *XML Best Practices*



# Advanced Spring Concepts

## Understanding the Spring Container

Bean Lifecycle, Bean Pre- and Post-Processors,  
Advanced Bean declarations

# Objectives

- After completing this lesson, you should be able to:
  - Explain the 3 Phases of the Spring Bean Lifecycle
  - Use a **BeanFactoryPostProcessor** and a **BeanPostProcessor**
  - Explain how Spring Proxies add behavior at runtime
  - Describe issues when injecting beans by Type

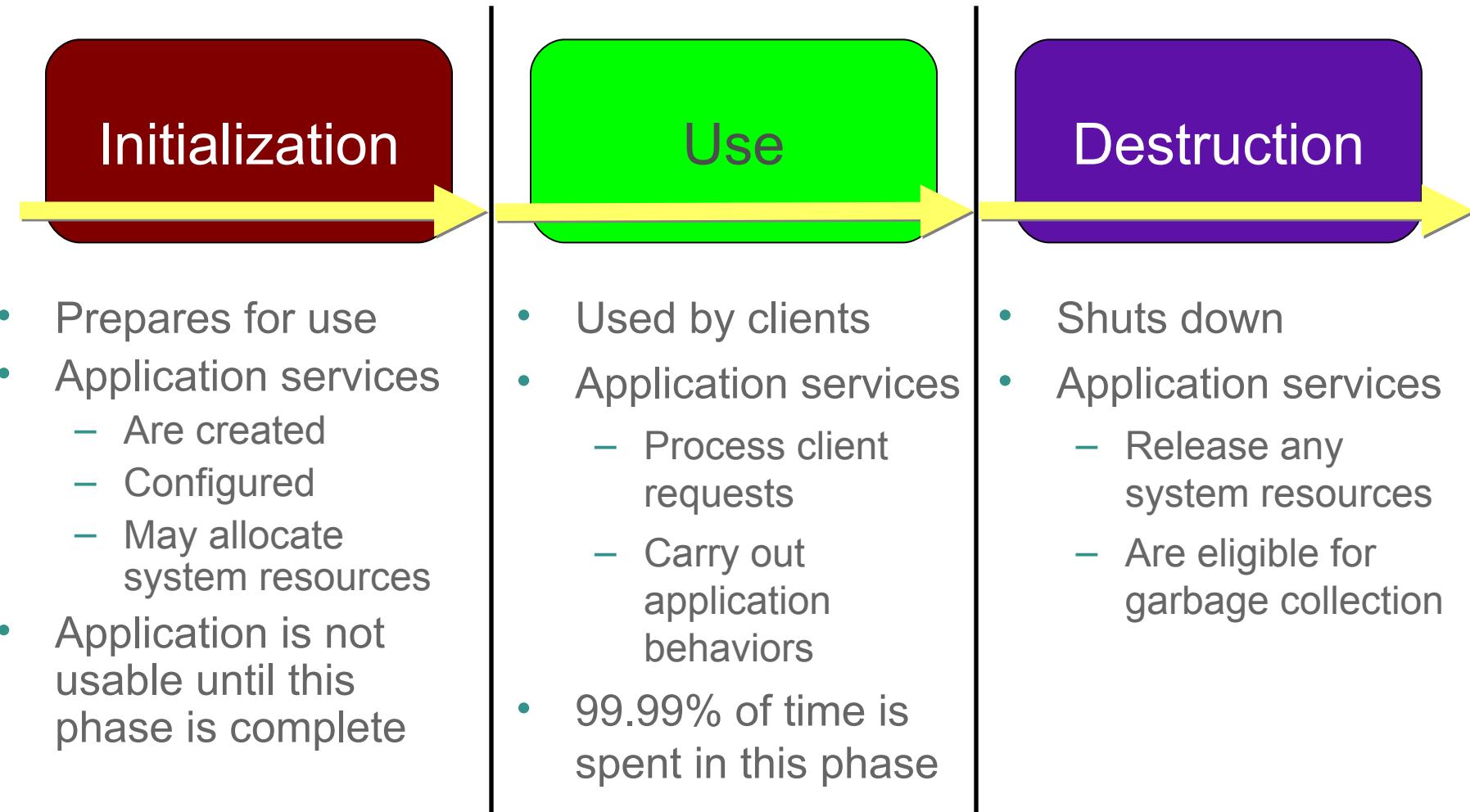


# Topics in this session

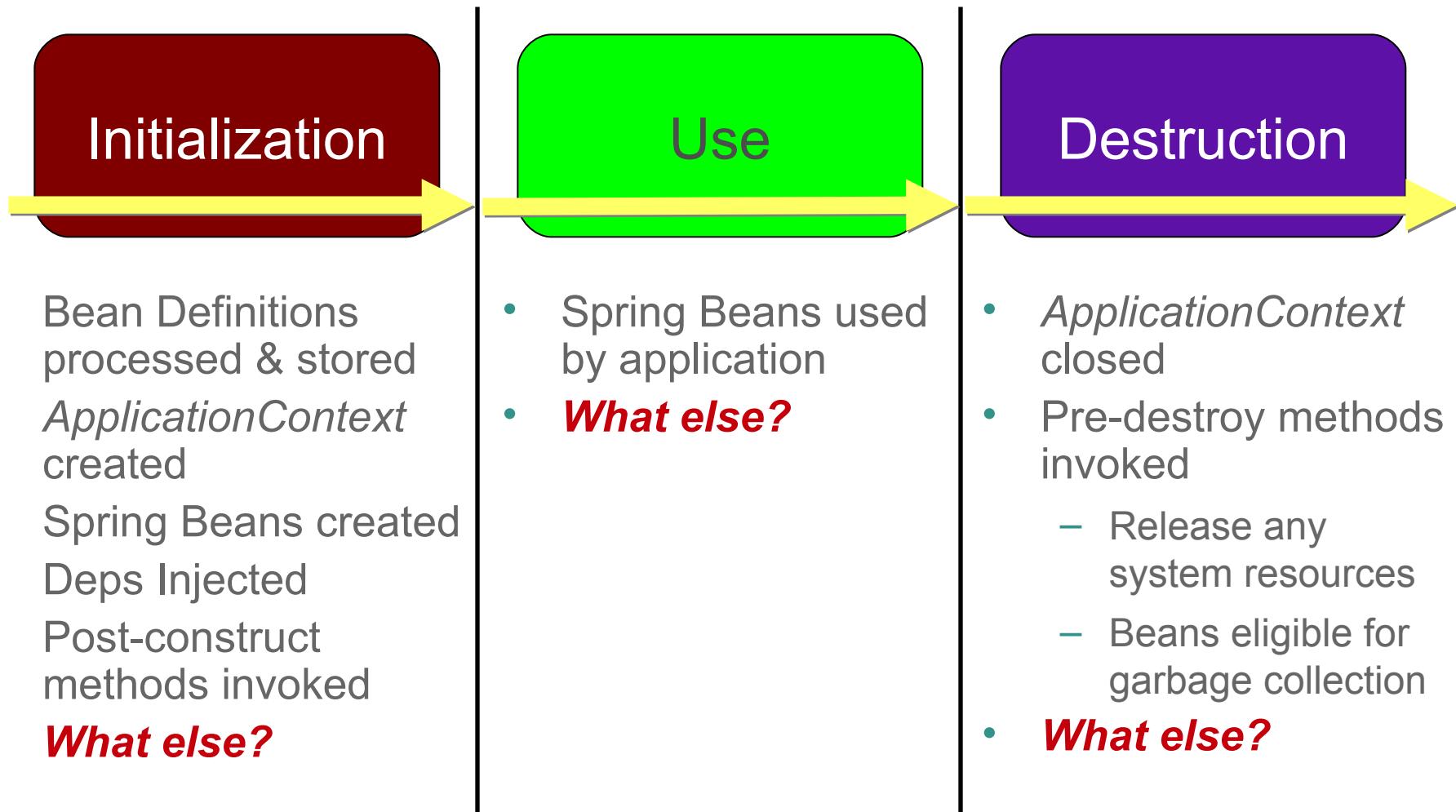
- **The Spring Bean Lifecycle**
- Bean Definition Post Processing
- Post Processing Beans
- Bean Proxies
- Interfaces vs Implementations

The content of this chapter is a *much simplified* view of Spring's inner workings

# Phases of the Application Lifecycle

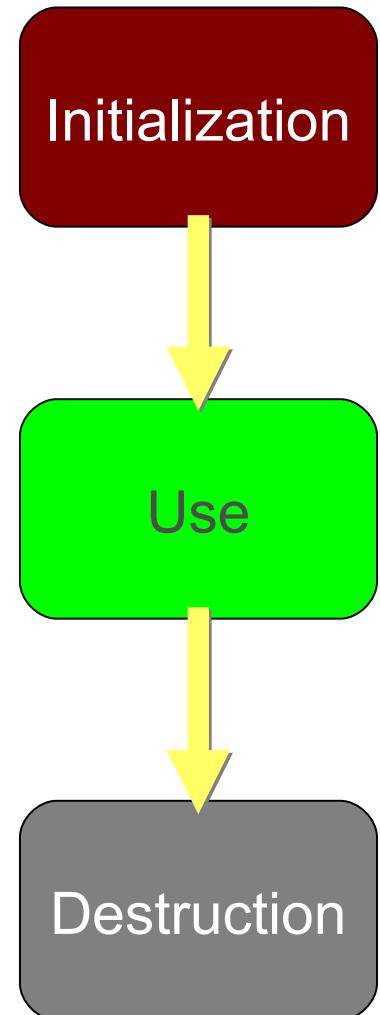


# Spring in Your Application Lifecycle



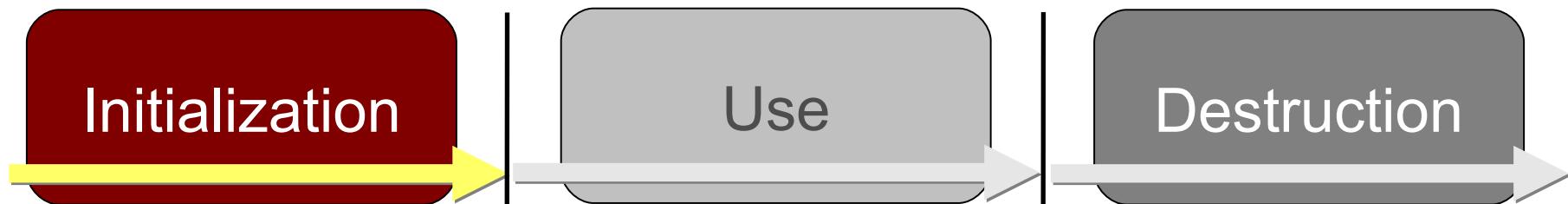
# But Wait ... There's More!

- Initialization
  - Two additional entry-points
    - Post Process Bean Definitions
    - Post Process the Beans themselves
- Use
  - Not all Spring Beans as you requested
    - They may have been “*Proxied*”
- Destruction: Nothing more to add



# Topics in this session

- The Spring Bean Lifecycle
- **Bean Definition Post Processing**
- Post Processing Beans
- Bean Proxies
- Interfaces vs Implementations



# Lifecycle of a Spring Application Context

## (1) *The Initialization Phase*

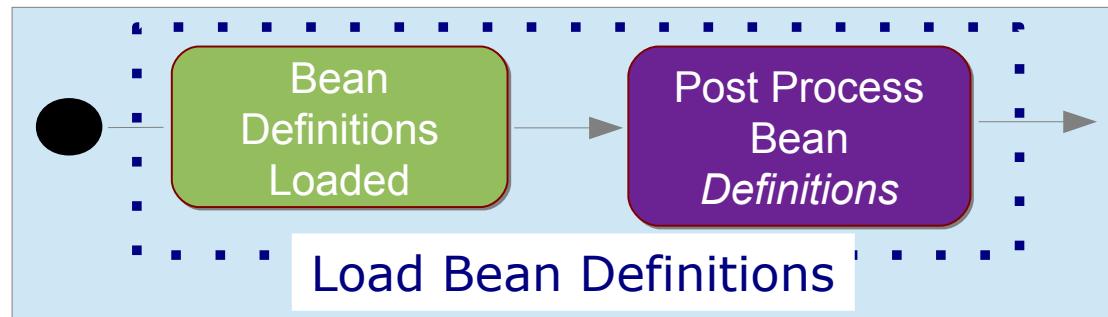
- When a context is created the initialization phase completes

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

- But what exactly happens in this phase?
  - Two separate steps
    - Step 1:** Load Bean Definitions
    - Step 2:** Perform Bean Creation

# Step 1. Load Bean Definitions

- The `@Configuration` classes are processed
  - And/or `@Components` are scanned for
  - And/or XML files are parsed
- Bean definitions added to a *BeanFactory*
  - Each indexed under its id and type
- 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

ApplicationContext

BeanFactory

transferService  
accountRepository  
dataSource

postProcess(BeanFactory)

BeanFactoryPostProcessors

# BeanFactoryPostProcessor

## *Internal Extension Point*

- Applies transformations to bean *definitions*
  - *Before* objects are actually created
- Several useful implementations provided in Spring
  - Reading properties, registering a custom scope ...
- You can write your own (not common)
  - Implement **BeanFactoryPostProcessor** interface

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

# BeanFactoryPostProcessor

## *Most Common Example*

- Recall @Value and \${ . . } variables
  - Need a PropertySourcesPlaceholderConfigurer to evaluate them
  - *This is a BeanFactoryPostProcessor*

```
@Configuration  
@PropertySource ( "classpath:/config/app.properties" )  
public class ApplicationConfig {  
  
    @Value("${max.retries}")  
    int maxRetries;  
  
    ...  
}
```

# BeanFactoryPostProcessor

Declare as *static* beans

Spring Boot  
sets this up  
*automatically*

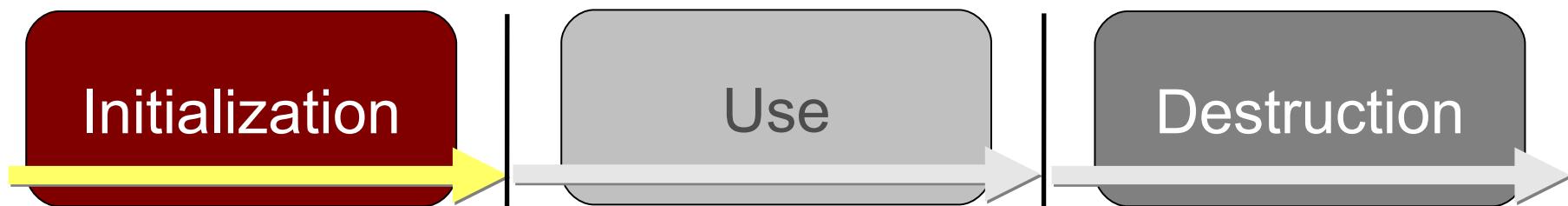
- To ensure these beans are created *before any other beans*, they must be defined as *static* methods
- Example:

— **PropertySourcesPlaceholderConfigurer**

```
@Bean
public static PropertySourcesPlaceholderConfigurer
    propertySourcesPlaceholderConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
}
```

# Topics in this session

- The Spring Bean Lifecycle
- Bean Definition Post Processing
- **Post Processing Beans**
- Bean Proxies
- Interfaces vs Implementations

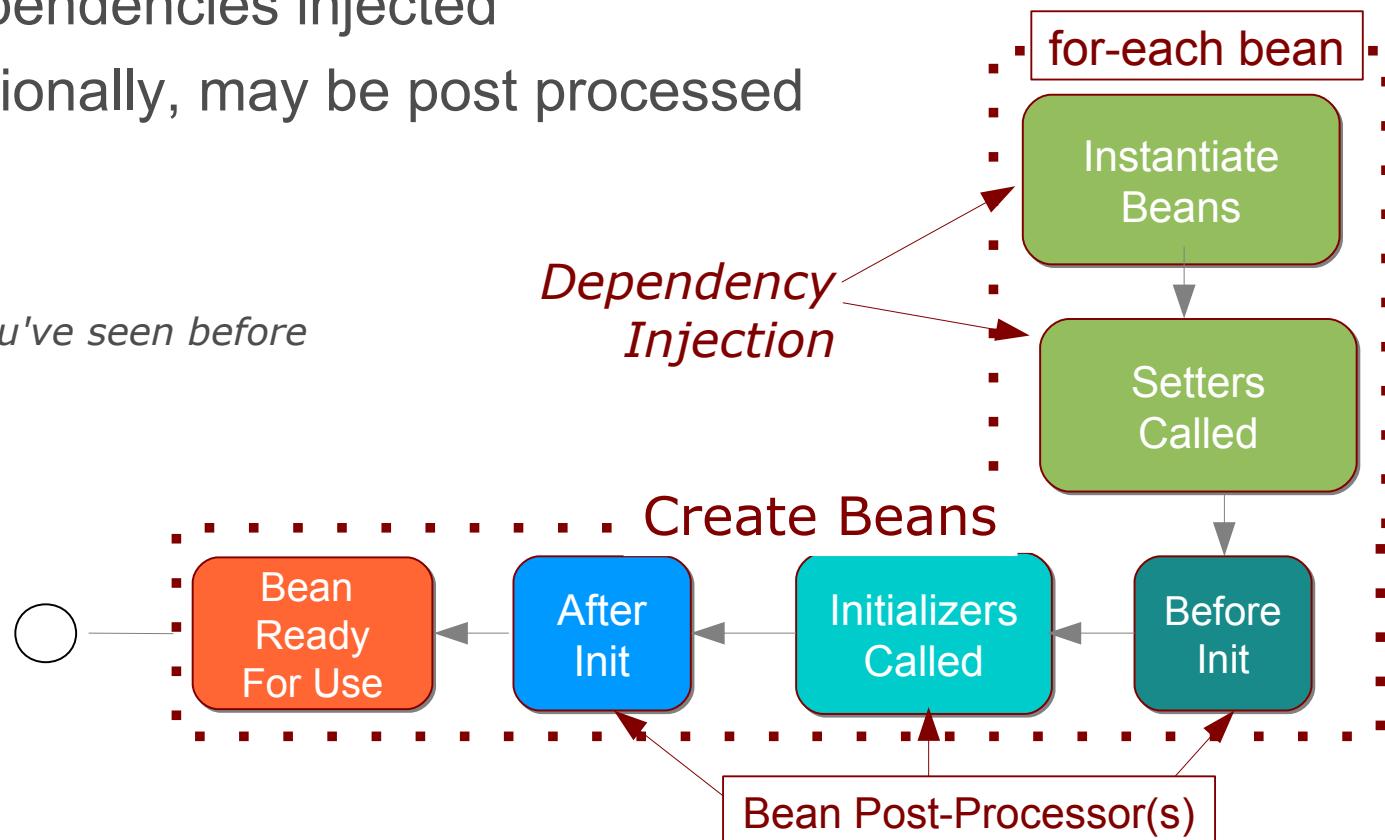


# Step 2: Perform Bean Creation

- Each Bean created in turn
  - Dependencies injected
  - Optionally, may be post processed

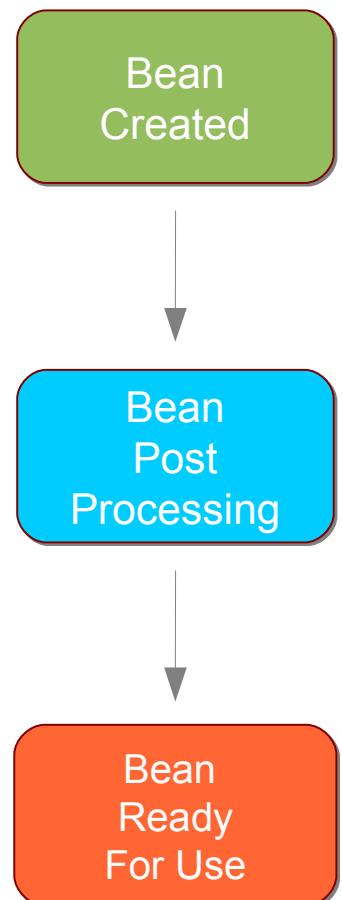


*What you've seen before*



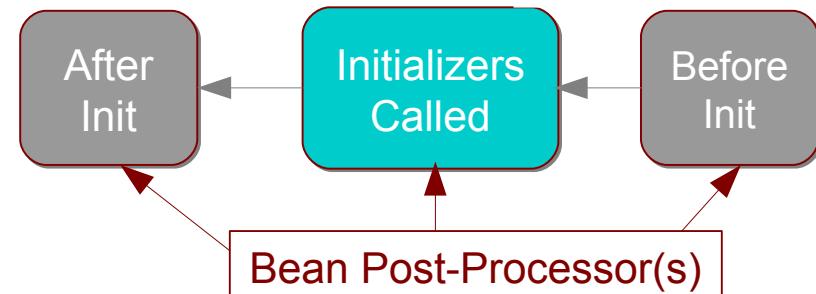
# Bean Creation Sequence of Events

- Each bean *eagerly* instantiated by default
  - Created in right order, with dependencies injected
    - Unless marked as lazy
- Next each bean goes through a *post-processing* phase
  - BeanPostProcessors
- Now bean is fully initialized & ready to use
  - Tracked by id until the context is destroyed
    - Except prototype beans



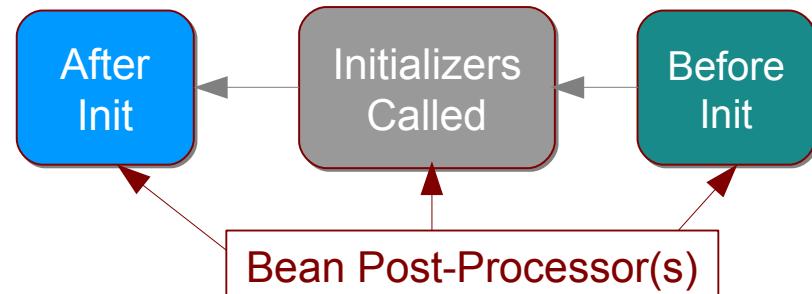
# The Initializer Extension Point

- Special case of a bean post-processor
  - Causes initialization methods to be called
    - Such as `@PostConstruct`
- Internally Spring uses several initializer BPPs
  - Example: `CommonAnnotationBeanPostProcessor` enables `@PostConstruct`, `@Resource` ...



# BeanPostProcessor Extension Point

- Important extension point in Spring
  - Can modify bean *instances* in any way
  - *Powerful* enabling feature
  - Can modify a bean before and/or after Initialization
    - If initializing before, be careful, not fully setup yet



# BeanPostProcessor Interface

Course will  
show several  
BPPs

- Bean Post Processors implement a known interface
  - Spring provides several implementations
  - You can write your own (not common)
    - Typically implement the after initialization method

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

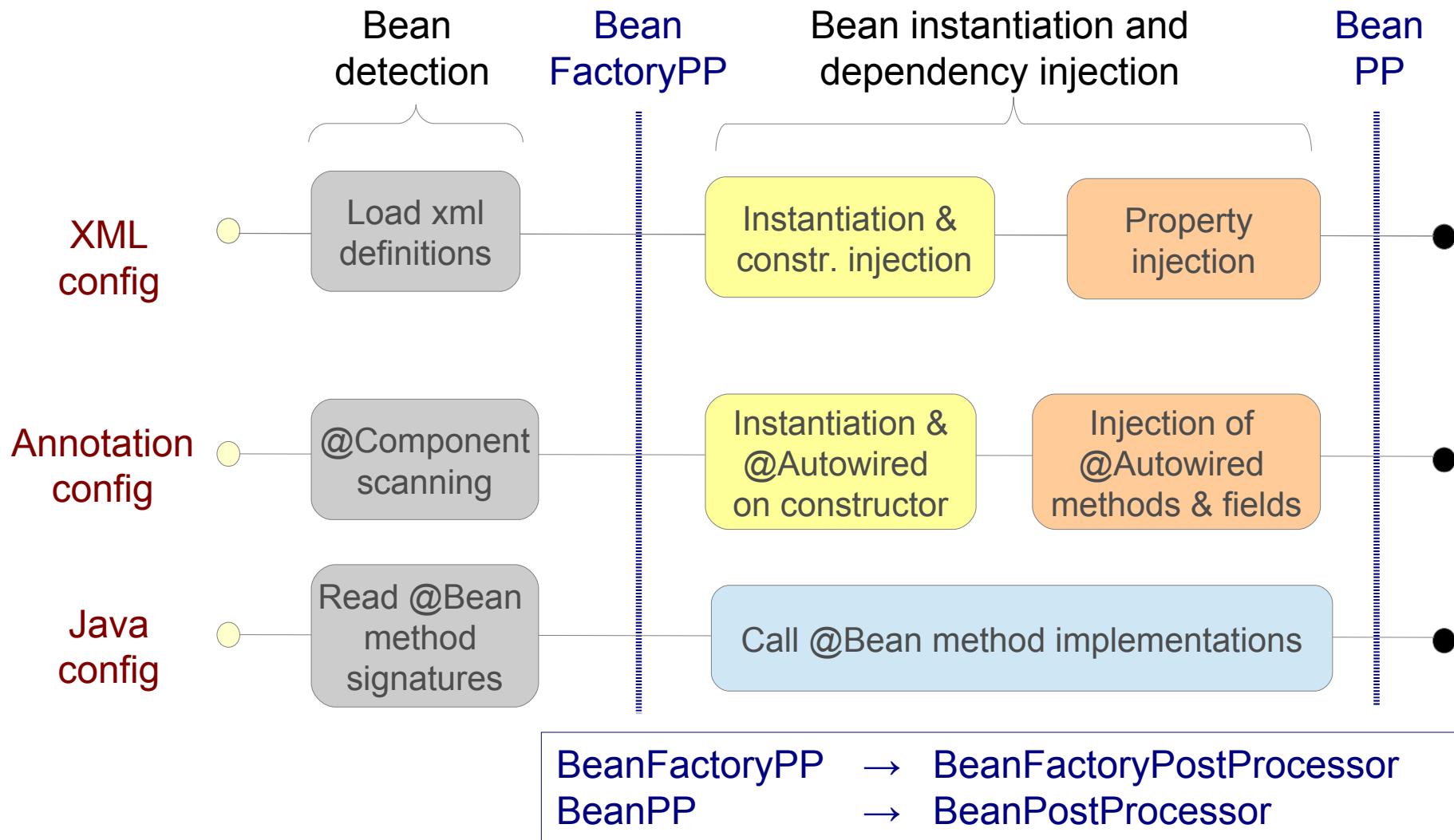
The diagram illustrates the flow of processing. On the left, the code for the `BeanPostProcessor` interface is shown. Two arrows point from the interface methods to two boxes at the bottom: a red box labeled "Post-processed bean" and a white box labeled "Original bean".

# Example: LoggingBeanPostProcessor

- An alternative to using DEBUG level logging

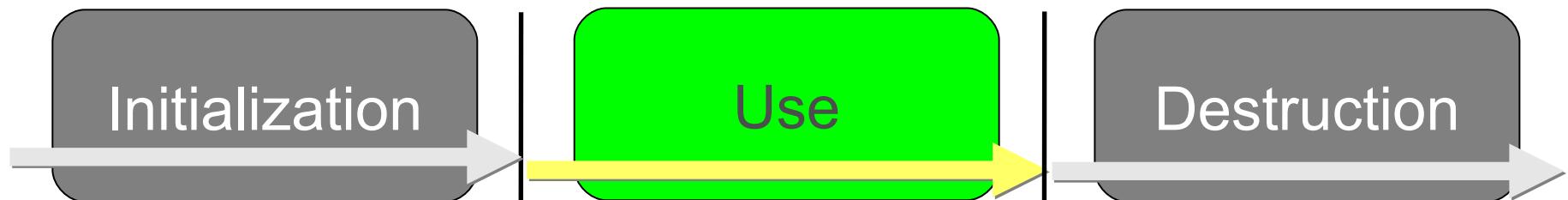
```
public class LoggingBeanPostProcessor extends BeanPostProcessor {  
    Logger logger = Logger.getLogger(LoggingBeanPostProcessor.class);  
  
    public Object postProcessBeforeInitialization(Object bean, String beanName) {  
        return bean; // Remember to return your bean or you'll lose it!  
    }  
  
    public Object postProcessAfterInitialization(Object bean, String beanName) {  
        logger.info(bean + ": " + bean.getClass());  
        return bean; // Remember to return your bean or you'll lose it!  
    }  
}
```

# Configuration Lifecycle



# Topics in this session

- The Spring Bean Lifecycle
- Bean Definition Post Processing
- **Post Processing Beans**
- Bean Proxies
- Interfaces vs Implementations



# Lifecycle of a Spring Application Context

## (2) The Use Phase

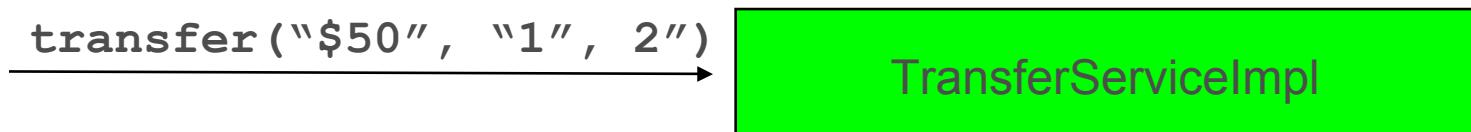
- When you invoke a bean obtained from the context the application is used

```
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");
```

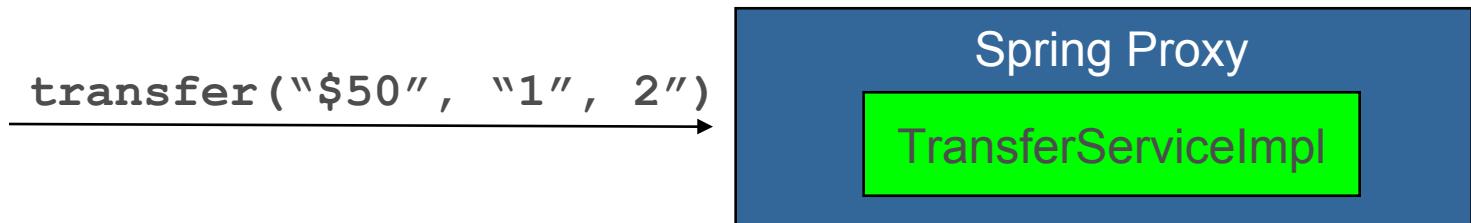
- But exactly what happens in this phase?

# Inside The Bean Request (Use) Lifecycle

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



- Your bean is wrapped in a *proxy*

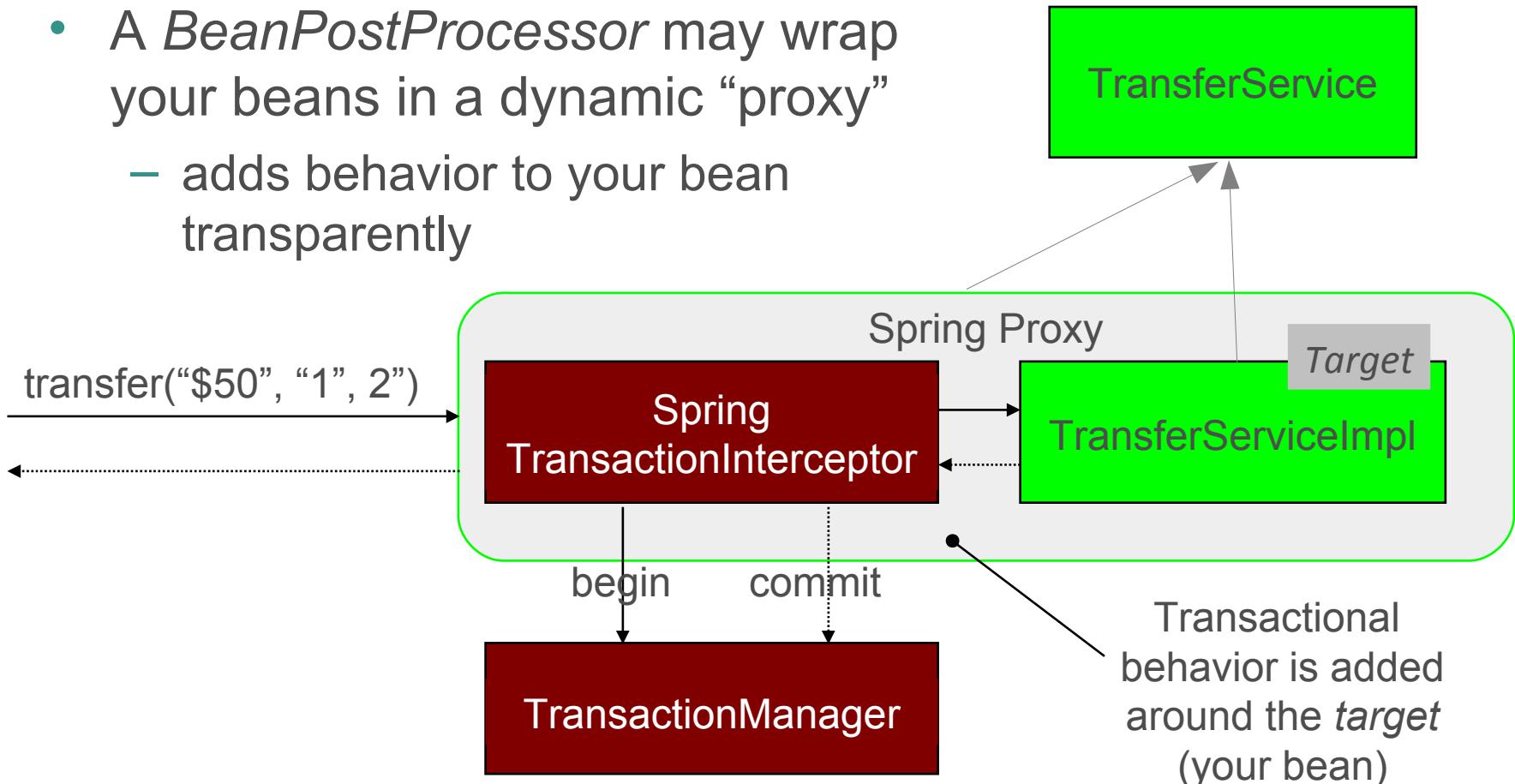


- Proxy created in initialization phase by BeanPostProcessor

```
public Object postProcess...Initialization(Object bean, String beanName);
```

# 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)
- All interfaces proxied

## CGLib Proxy

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

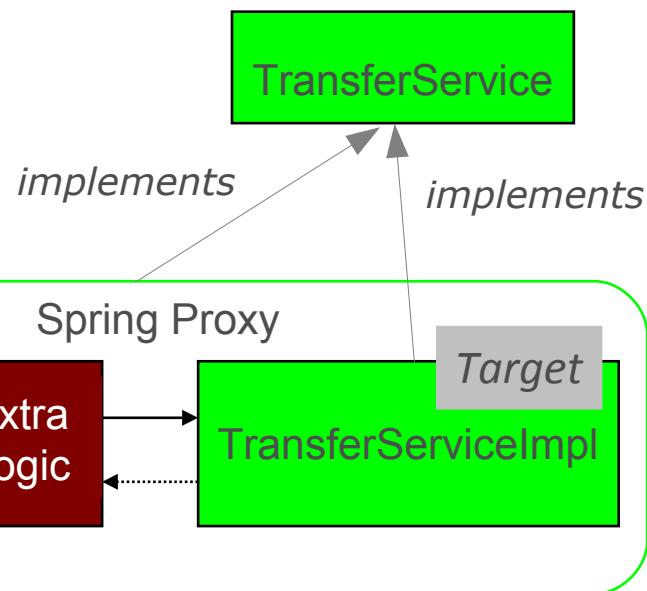


*Recommendation:* Code to interfaces / Use JDK proxies (default)

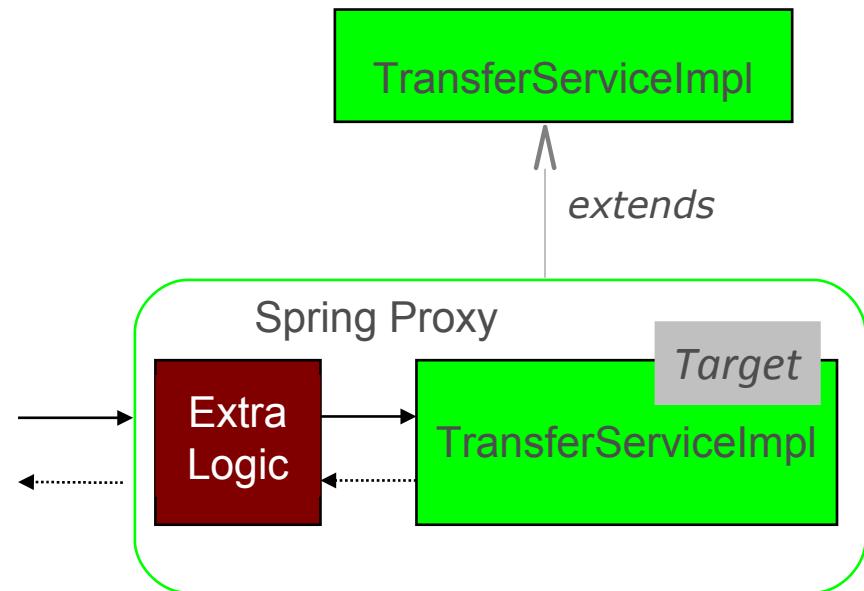
**See Spring Reference - 10.5.3 JDK- and CGLIB-based proxies**

# JDK vs CGLib Proxies

- JDK Proxy
  - Interface based



- CGLib Proxy
  - subclass based



# Topics in this session

- The Spring Bean Lifecycle
- Bean Definition Post Processing
- Post Processing Beans
- Bean Proxies
- **Interfaces vs Implementations**

# Why Won't This Work?

```
@Configuration  
class Config  
{
```

```
class BankTransferService  
    implements TransferService, BankService {  
    ...  
}
```

```
@Bean  
public TransferService transferService(AccountRepository repo) {  
    return new BankTransferService( repo );  
}
```

```
@Bean  
public BankingClient bankingService(BankService svc) {  
    return new BankingClient( svc );  
}
```

...

??

No **@Bean** method exists returning a **BankService**

# Solution 1: Return Actual Type

```
@Configuration  
class Config  
{
```

```
class BankTransferServiceImpl  
    implements TransferService, BankService {  
    ...  
}
```

```
@Bean  
public BankTransferServiceImpl transferService(AccountRepository repo) {  
    return new BankTransferServiceImpl( repo );  
}
```

```
@Bean  
public BankingClient bankingService(BankService svc) {  
    return new BankingClient( svc );  
}
```

...

Can determine `BankTransferService` implements both `TransferService` and `BankService`

## Solution 2: Return Composite Interface

```
@Configuration  
class Config  
{  
  
    @Bean  
    public BankTransferService transferService(AccountRepository repo) {  
        return new BankTransferServiceImpl( repo );  
    }  
  
    @Bean  
    public BankingClient bankingService(BankService svc) {  
        return new BankingClient( svc );  
    }  
    ...  
  
    interface BankTransferService  
        implements TransferService, BankService {  
    }  
  
    class BankTransferServiceImpl  
        implements BankTransferService { ... }
```

Can determine `BankTransferService` extends both `TransferService` and `BankService`



# Which is Best?

- Aim to be “*sufficiently expressive*”
  - Return interfaces except
    - Where *multiple* interfaces exist
    - They are needed for dependency injection
  - Writing to interfaces is *good practice*
- **Warning:** If you return actual types
  - Still dependency inject *interfaces*
  - Injecting actual types is brittle
    - Dependency injection *fails* if the implementation changes or is proxied



# Topics Covered

- Spring Bean Lifecycle
  - Three phases: *initialize, use, destroy*
  - **BeanFactoryPostProcessor**
    - Processes bean *definitions* (**no** beans yet)
    - Allocate using *static @Bean* method
  - **BeanPostProcessor**
    - Processes Beans
    - Performs initialization, creates proxies, ...
- Advanced **@Bean** Definitions
  - When to return actual types instead of interfaces

# Testing Spring Applications

Unit Testing without Spring  
Integration Testing with Spring

Testing in General, Spring and JUnit,  
Profiles, Database Testing

# Objectives

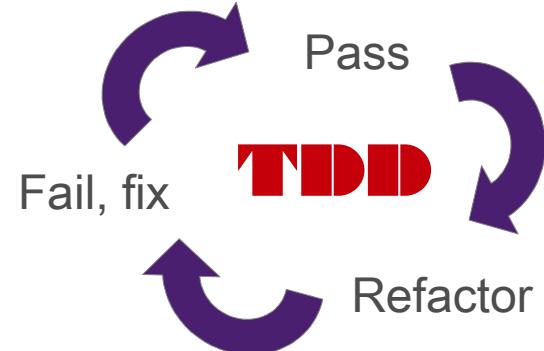
- After completing this lesson, you should be able to:
  - Explain the concepts of Test Driven Development
  - Write tests using JUnit 5
  - Write Integration Tests using Spring
  - Configure Tests using Spring Profiles
  - Extend Spring Tests to work with Databases



# Topics in this Session

- **Test Driven Development**
- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

# 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?
  - Ultimately 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 We 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
  - Testing tool is better at this than you are
- Properly done TDD is faster than development without tests



# TDD, Refactoring 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



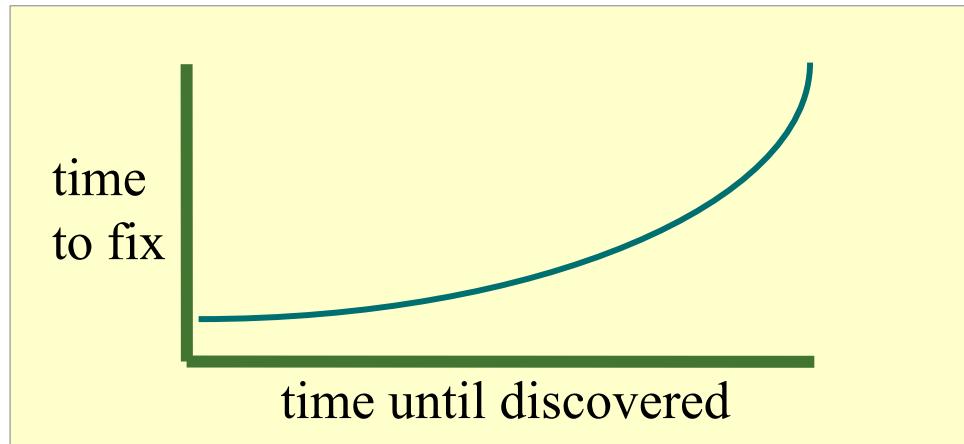
# TDD and Focus

- 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 (CI)

- 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
- **JUnit 5**
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

## What is JUnit 5?

- JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
  - JUnit Platform
    - A foundation for launching testing frameworks on the JVM
  - JUnit Jupiter
    - Defines a new programming model and extension model for writing tests and extensions in JUnit 5
  - JUnit Vintage
    - A TestEngine for running JUnit 3 & 4 tests on the platform
- In JUnit 4, a monolithic junit.jar handled everything
- JUnit 5 requires Java 8 (or higher) at runtime
  - Leverages Lambdas

# JUnit 5: New Programming Models

- Replaces JUnit 4 annotations
    - `@Before` → `@BeforeEach`
    - `@BeforeClass` → `@BeforeAll`
    - `@After` → `@AfterEach`
    - `@AfterClass` → `@AfterAll`
    - `@Ignore` → `@Disabled`
  - Introduces new annotations
    - `@DisplayName`
    - `@Nested`
    - `@ParameterizedTest`
    - ...
- 
- JUnit 5

# Writing Test – JUnit 5 Style

```
import static org.junit.jupiter.api.Assertions.fail;
```

New package

```
import org.junit.jupiter.api.AfterAll;  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeAll;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Disabled;  
import org.junit.jupiter.api.Test;
```

```
class StandardTests {
```

```
    @BeforeAll  
    static void initAll() {  
    }
```

Replaces  
`@BeforeClass`

```
    @BeforeEach  
    void init() {  
    }
```

Replaces  
`@Before`

```
@Test  
void succeedingTest() {  
}
```

```
@Test  
void failingTest() {  
    fail("a failing test");  
}
```

Replaces  
`@Ignore`

```
@Test  
@Disabled("for demo purposes")  
void skippedTest() {  
    // not executed  
}
```

# JUnit 5: New Programming Models

- Introduces new assertions
  - `assertThrows`  
`(<exception.class>, <lambda-expression>)`
  - `assertTimeout`  
`(<duration>, <lambda-expression>)`
  - `assertAll`  
`(<description>, <multiple-assertions>)`
- Improves assumptions
  - Uses lambda expression
    - The code to be tested

# assertThrows(..) and assertTimeout(..)

```
@Test
void exceptionTesting() {
    Throwable exception = assertThrows(IllegalArgumentException.class,
        () -> { /* Perform task that throws illegal argument exception */ })
    );
    assertEquals("some error message", exception.getMessage());
}
```

No need to specify expected attribute

```
@Test
void timeoutNotExceeded() {
    // The following assertion succeeds.
    assertTimeout(ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}
```

No need to specify timeout attribute

# Assertions

## *Using Lambdas and assertAll(...)*

```
@Test
void standardAssertions() {
    assertEquals(2, 2);
    assertEquals(4, 4, "Optional assertion message is now last parameter.");
    assertTrue(2 == 2, () -> "Assertion messages can be lazily evaluated -- "
        + "to avoid constructing complex messages unnecessarily.");
}

@Test
void groupedAssertions() {
    // In a grouped assertion all assertions are executed, and any
    // failures will be reported together.
    assertAll("person",
        () -> assertEquals("John", person.getFirstName()),
        () -> assertEquals("Doe", person.getLastName())
    );
}
```

# Assumptions

```
@Test void testOnlyOnCiServer() {  
    assumeTrue("CI".equals(System.getenv("ENV")));  
    // remainder of test  
}
```

```
@Test void testOnlyOnDeveloperWorkstation() {  
    assumeTrue("DEV".equals(System.getenv("ENV")),  
        () -> "Aborting test: not on developer workstation");  
    // remainder of test  
}
```

```
@Test void testInAllEnvironments() {  
    assumingThat("CI".equals(System.getenv("ENV"))),  
        () -> {  
            // perform these assertions only on the CI server  
            assertEquals(2, 2);  
        };  
  
    // perform these assertions in all environments  
    assertEquals("a string", "a string");  
}
```

# Topics in this Session

- Test Driven Development
- JUnit 5
- **Integration Testing with Spring**
- Testing with Profiles
- Testing with Databases
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

# Unit Testing

Unit Testing  
*Without Spring*

- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
  - Uses simplified alternatives for dependencies
    - Stubs and/or Mocks
    - See *Appendix for more details*

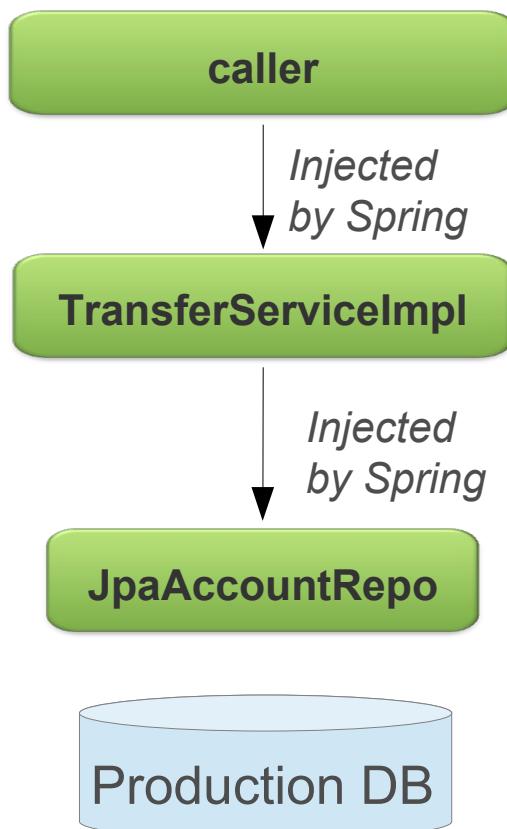
# Integration Testing

## Integration Testing *With Spring*

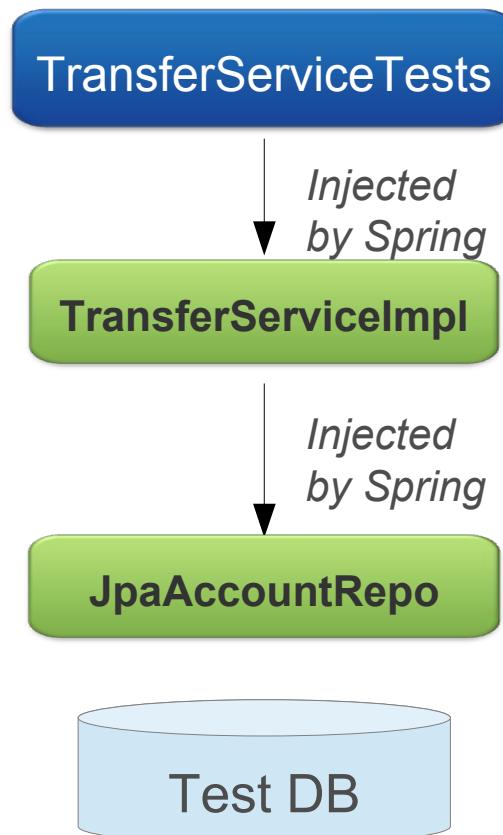
- Integration (System) Testing
  - Tests the interaction of multiple units working together
    - All should work individually (unit tests showed this)
  - Tests application classes in context of their surrounding infrastructure
    - Out-of-container testing, no need to run up full JEE system
    - Infrastructure may be “scaled down”
      - Use Apache DBCP connection pool instead of container-provider pool obtained through JNDI
      - Use ActiveMQ to save expensive commercial messaging server licenses

# Integration test example

- Production mode



- Integration test



# Spring Support for JUnit 5

- Support all core Spring `TestContext` Framework features
  - Constructor and method injection via
    - `@Autowired`, `@Qualifier`, `@Value`
  - `@ExtendWith(SpringExtension.class)`
  - `@SpringJUnitConfig`
- Packaged as a separate module
  - `spring-test.jar`



See: [Spring Framework Reference – Integration Testing](#)

<https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#integration-testing>

# @ExtendWith in JUnit 5

- JUnit 5 has an extensible architecture which can be taken advantage of via the `@ExtendWith` annotation
  - No longer limited to using just one extension on Test Class
    - As was the case with JUnit 4
- Example extension points
  - `ParameterResolver`
    - Dynamically resolve parameters at runtime

# @ExtendWith(SpringExtension.class)

- Integrates the Spring `TestContext` Framework into JUnit 5's Jupiter programming model
  - Replaces Junit 4's `@RunWith`
    - `@RunWith(SpringRunner.class)`
- Serves as a `ParameterResolver` implementation
  - Resolves requested parameter dependency against Spring's `ApplicationContext`
  - Dependencies can be specified as arguments of test method

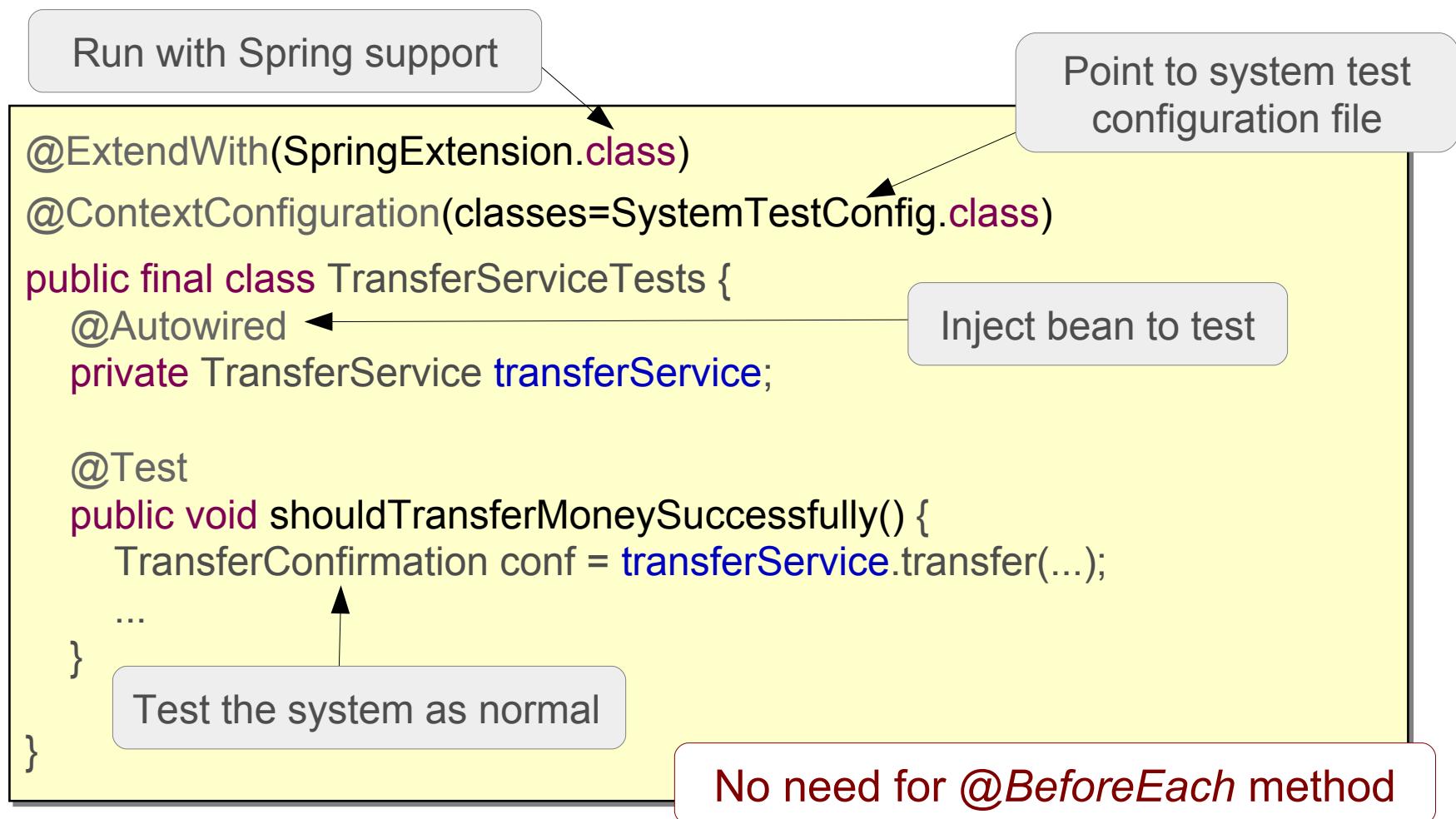


See Appendix (end of this section) for Spring's JUnit 4 support.

# @SpringJUnitConfig

- `@SpringJUnitConfig` is a “composed” annotation that combines
  - `@ExtendWith(SpringExtension.class)` from JUnit Jupiter
  - `@ContextConfiguration` from the Spring TestContext Framework

# Using Spring's Test Support



# Alternative Approaches

```
@SpringJUnitConfig(classes=SystemTestConfig.class)
public final class TransferServiceTests {
    // @Autowired
    // private TransferService transferService;

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

Use composite annotation

no longer required –  
dependency injected as  
test method *argument*

# Using JUnit 4 to run JUnit 5 Test code

- **@RunWith (JUnitPlatform.class)**
  - Provides backwards compatibility
  - Allows test to be run with IDEs and build systems that support JUnit 4
    - but do not yet support the JUnit 5 Platform directly

# Running JUnit 5 test with JUnit 4

```
@RunWith(JUnitPlatform.class)
```

```
@ExtendWith(SpringExtension.class)
```

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

```
public final class TransferServiceTests {
```

```
    @Test
```

```
    public void shouldTransferMoneySuccessfully
```

```
        (@Autowired TransferService transferService) {
```

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

```
            ...
```

```
        }
```

```
}
```

Run JUnit 5 tests  
using JUnit 4 runner

# Including Configuration as an inner class

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration
public class JdbcAccountRepoTest {

    private JdbcAccountRepo repo = ...;

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

    @Configuration
    @Import(SystemTestConfig.class)
    static class TestConfiguration {
        @Bean public DataSource dataSource() { ... }
    }
}
```

Don't specify any config classes

Looks for configuration *embedded* in test class

Override a bean with a test alternative

# Multiple Test Methods

```
@SpringJUnitConfig(classes=SystemTestConfig.class)
public class TransferServiceTests {
    @Autowired
    private TransferService transferService;

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

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

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

Using same TransferService instance



Most Spring Beans are *stateless/immutable* singletons, never modified during any test. No need for a new context for each test.

# Dirties Context

- Forces context to be closed at end of test method
  - Allows testing of `@PreDestroy` behavior
- Next test gets a *new* Application Context
  - Cached context destroyed, new context cached instead

```
@Test  
{@DirtiesContext  
public void testTransferLimitExceeded() {  
    transferService.setMaxTransfers(0);  
    ... // Do a transfer, expect a failure  
}}
```

Context closed and destroyed  
at end of test

# Test Property Sources

- Custom properties *just* for testing
  - Specify one or more properties
    - Will override any existing properties of same name
  - Specify location of one or more properties files to load
    - Defaults to looking for `[classname].properties`

```
@SpringJUnitConfig(classes=SystemTestConfig.class)
@TestPropertySource(properties = { "username=foo", "password=bar" })
                           locations = "classpath:/transfer-test.properties")
public class TransferServiceTests {
    ...
}
```

# Benefits of Testing with Spring

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

# Topics in this Session

- Test Driven Development
- JUnit 5
- Integration Testing with Spring
- **Testing with Profiles**
- Testing with Databases
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

# 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**

```
@SpringJUnitConfig(classes=DevConfig.class)
```

```
@ActiveProfiles( { "jdbc", "dev" } )
```

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

# Profiles Activation with JavaConfig

- `@ActiveProfiles` inside the test class
- `@Profile` inside an `@Configuration` class

```
@SpringJUnitConfig(classes=DevConfig.class)
@ActiveProfiles("jdbc")
public class TransferServiceTests
{...}
```

```
@Configuration
@Profile("jdbc")
public class DevConfig {

    @Bean
    {...}
}
```



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

# Profiles Activation with Annotations

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

```
@SpringJUnitConfig(classes=DevConfig.class)
@ActiveProfiles("jdbc")
public class TransferServiceTests
{...}
```

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



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

# Topics in this Session

- Test Driven Development
- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- **Testing with Databases**
- Lab
- Appendix on Unit Testing (Stubs & Mocks)

# 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.

# @Sql Examples

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

```
public class MainTests {
```

```
    @Test  
    @Sql  
    public void success() { ... }
```

Run these scripts before each `@Test` method

Run script named (by default) `MainTests.success.sql` in same package

```
    @Test  
    @Sql ( "/testfiles/error.sql" )  
    @Sql ( scripts="/testfiles/cleanup.sql",  
           executionPhase=Sql.ExecutionPhase.AFTER_TEST_METHOD )  
    public void transferError() { ... }
```

Run before `@Test` method...

... run after `@Test` method

# @Sql Options

- When does the SQL run?
  - *executionPhase*: before (default) or after the test method
  - *config*: SqlConfig has many options to control SQL scripts
    - What to do if script fails? **FAIL\_ON\_ERROR**, **CONTINUE\_ON\_ERROR**, **IGNORE\_FAILED\_DROPS**, **DEFAULT\***
    - SQL syntax control: comments, statement separator

```
@Sql( scripts = "/test-user-data.sql",
       executionPhase = ExecutionPhase.AFTER_TEST_METHOD,
       config = @SqlConfig(errorMode = ErrorMode.FAIL_ON_ERROR,
                            commentPrefix = "//", separator = "@@") )
```

**\*DEFAULT** = whatever @Sql defines at class level, otherwise **FAIL\_ON\_ERROR**

# 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
  - Built-in support for testing with Databases

# Lab

## Testing Spring Applications

**Coming Up:** Appendix on Unit Testing using Stubs or Mocks

# Topics in this Session

- Test Driven Development
- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Appendix
  - JUnit4 Support
  - Unit Testing (Stubs & Mocks)

# 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

<https://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/htmlsingle/#integration-testing>

# Using Spring's Test Support

Run with Spring support

Point to system test configuration file

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

public final class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void shouldTransferMoneySuccessfully() {
        TransferConfirmation conf = transferService.transfer(...);

        ...
    }
}
```

Inject bean to test

Test the system as normal

No need for @Before method

# Including Configuration as an inner class

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

```
@ContextConfiguration
```

```
public class JdbcAccountRepoTest {
```

Don't specify any config classes

```
private JdbcAccountRepo repo = ...;
```

```
@Test
```

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

```
| @Configuration
```

```
| @Import(SystemTestConfig.class)
```

```
| static class TestConfiguration {
```

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

```
| }
```

```
}
```

Looks for configuration *embedded* in test class

Override a bean with a test alternative

# Multiple test methods

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

public 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

# Spring 4.3 Simplification

- Can use **SpringRunner** as an alternative to the **SpringJUnit4ClassRunner**
  - Simply a sub-class with a nicer name

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)
public class TransferServiceTests {

    ...
}
```

# Topics in this Session

- Test Driven Development
- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- **Appendix**
  - JUnit4 Support
  - **Unit Testing (Stubs & Mocks)**

# Unit Testing vs. Integration Testing

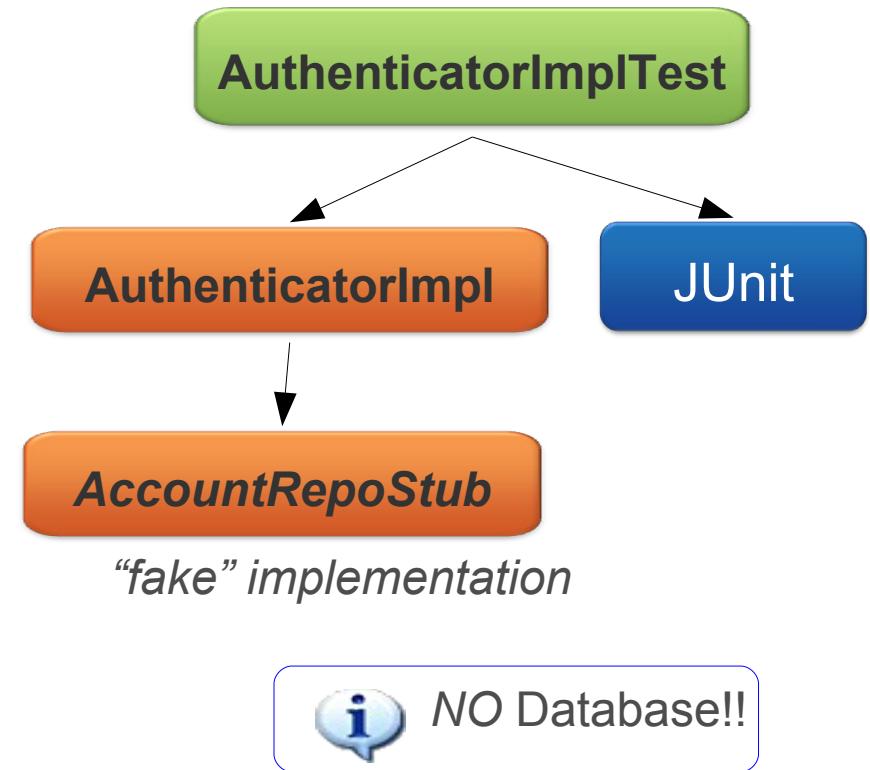
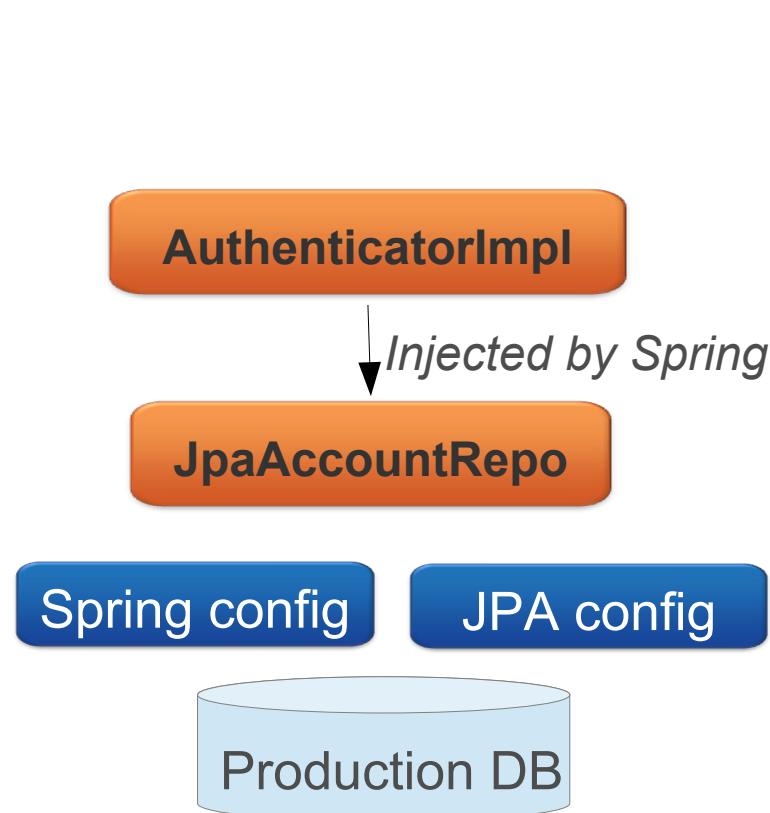
- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
- Integration Testing
  - Tests the interaction of multiple units working together
  - Integrates infrastructure
- Discussed Integration Testing earlier
  - Let's discuss Unit Testing here
  - Remember: *Unit Testing does not use Spring*

# Unit Testing

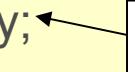
- Remove links with dependencies
  - The test shouldn't fail because of external dependencies
  - Spring is also considered as a dependency
  -
- 2 ways to create a “testing-purpose” implementation of your dependencies:
  - Stubs Create a simple test implementation
  - Mocks Dependency class generated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

# Unit Testing example

- Production mode
- Unit test with Stubs



# Example Unit to be Tested

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

External dependency

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

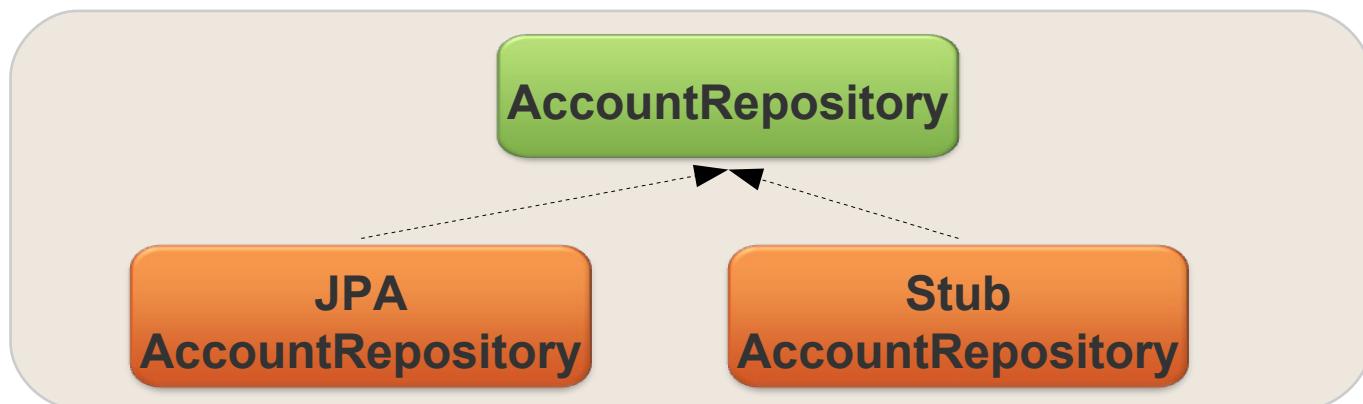
Note: Validation failure paths ignored for simplicity

# Implementing a Stub

- Class created manually
  - Implements Business interface

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

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

Recording

What behavior to  
expect?

Recording      Playback

“playback”  
mode

Mock now fully available

Verification

No planned method call  
has been omitted

# Same Example using Mockito

```
import static org.mockito.Mockito.*;  
  
public class AuthenticatorImplTests {  
    private AccountRepository accountRepository  
        = mock( AccountRepository.class );           // Create a mock object  
    private AuthenticatorImpl authenticator  
        = new AuthenticatorImpl(accountRepository); // Inject the mock object  
  
    @Test public void validUserWithCorrectPassword() {  
        when(accountRepository.getAccount("lisa")).           // Train the mock  
           thenReturn(new Account("lisa", "secret"));  
  
        assertTrue( authenticator.authenticate("lisa", "secret") ); // Run test  
        verify(accountRepository);                            // Verify getAccount() was  
    }                                                       // invoked on the mock  
}
```

No replay() step with Mockito

# 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>

# Developing Aspects with Spring AOP

Aspect Oriented Programming For  
Declarative Enterprise Services

Using and Implementing Spring Proxies

# Objectives

- After completing this lesson, you should be able to:
  - Explain the concepts behind AOP and the problem that it solves
  - Implement and deploy an Advice using Spring
  - Use AOP Pointcut Expressions
  - Explain the 5 different Types of Advice
    - And when to use them



# Topics in this session

- **What Problem Does AOP Solve?**
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics



# 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



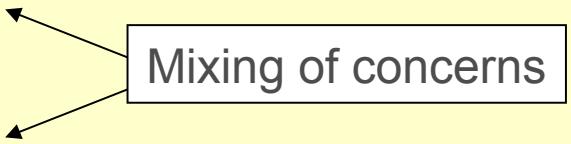
A sign this requirement is a cross-cutting concern

# 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

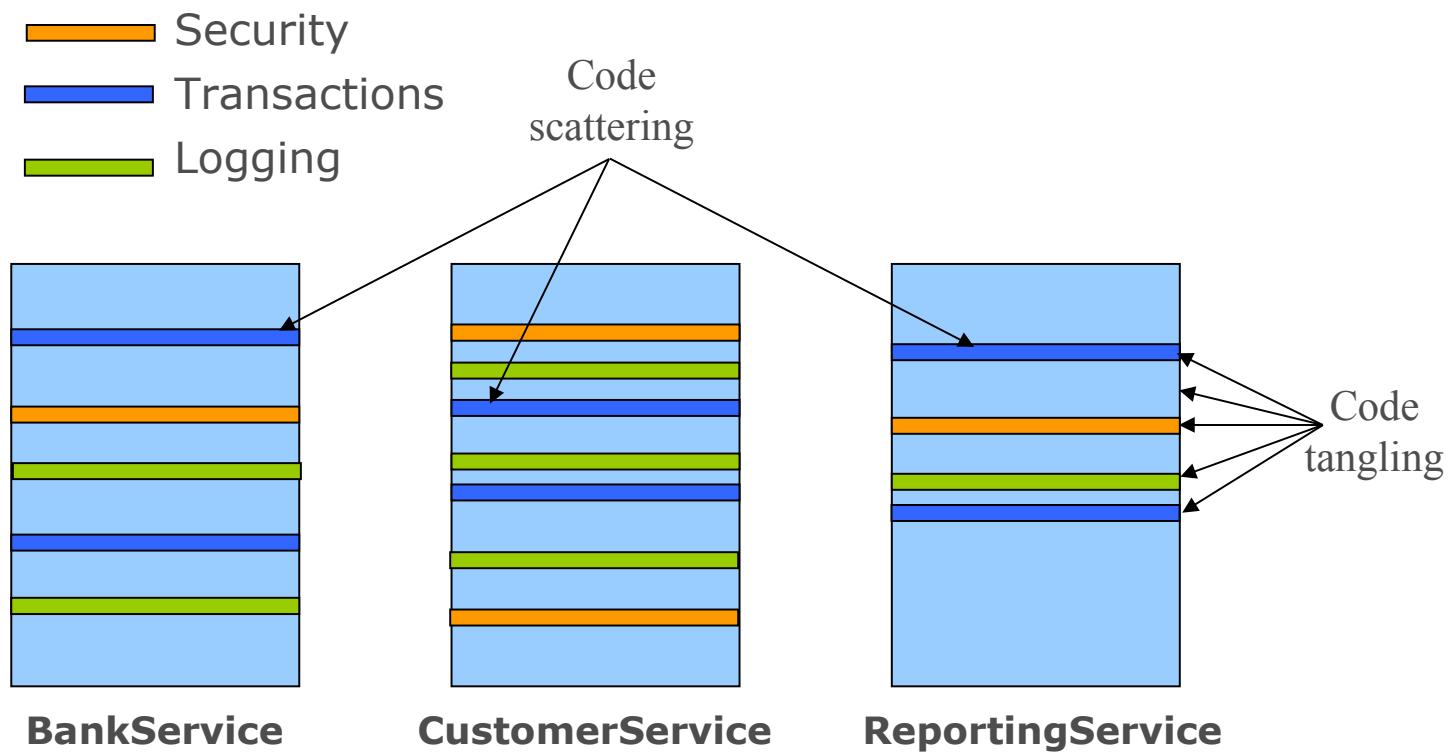
## Symptom #2: Scattering

```
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



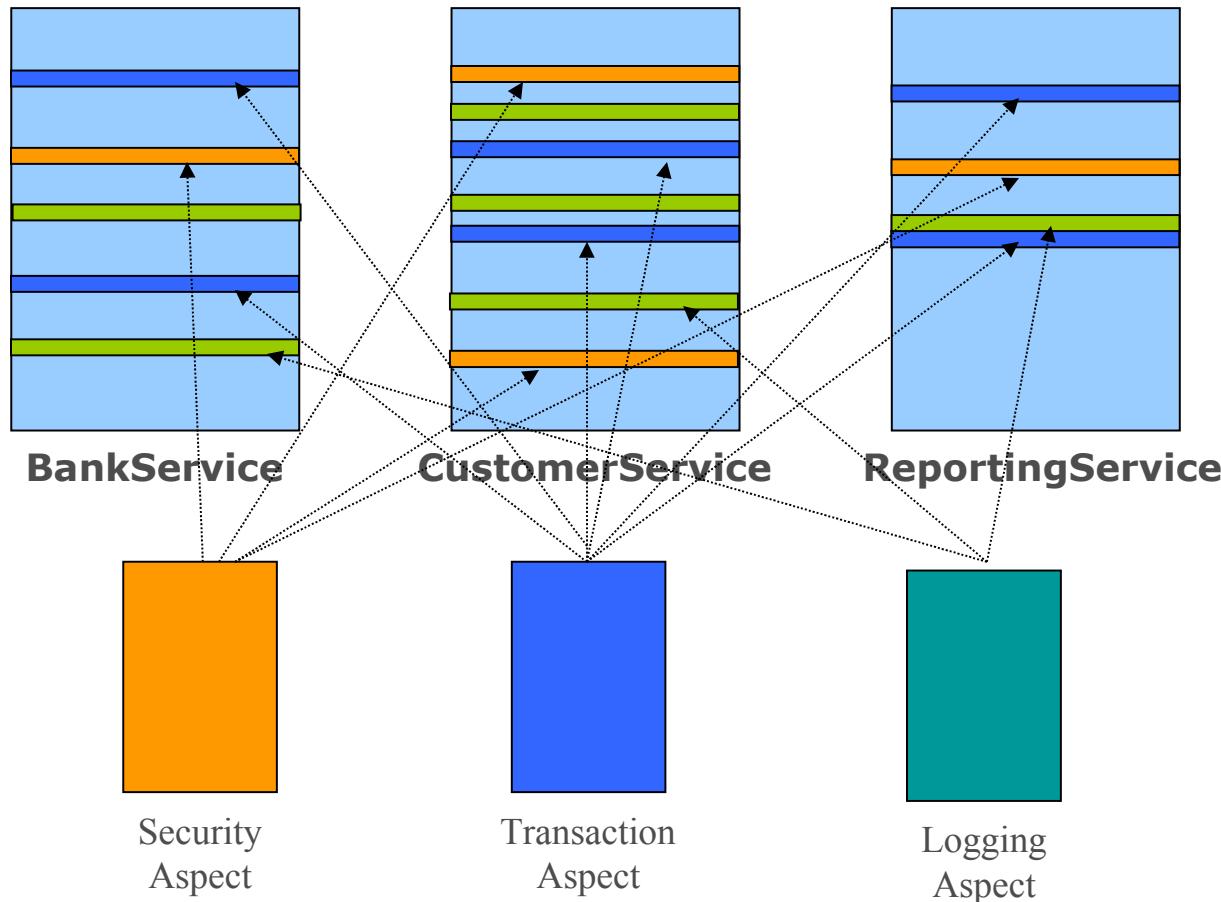
# Aspect Oriented Programming (AOP)

- 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)
  - 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](#)  
<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop>

# Topics in this session

- What Problem Does AOP Solve?
- **Core AOP Concepts**
- Quick Start
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics

# 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

# Core AOP Concepts: Proxy



- Proxy
  - Someone who stands in place of someone else
    - Such as at an auction or an official meeting
  - Web-proxy
    - Allows access through company firewall
  -
- In AOP
  - The “woven” class that stands in place of your original
    - With extra behavior (Aspect) added (woven) into it

# Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- **Quick Start**
- Defining Pointcuts
- Implementing Advice
- Lab
- Advanced Topics

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

```
public class SimpleCache implements Cache {  
    private int cacheSize;  
    private String name;  
  
    public SimpleCache(String beanName) { name = beanName; }  
  
    public void setCacheSize(int size) { cacheSize = size; }  
  
    ...  
  
    public String toString() { return name; }      // For convenience later  
}
```

# 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

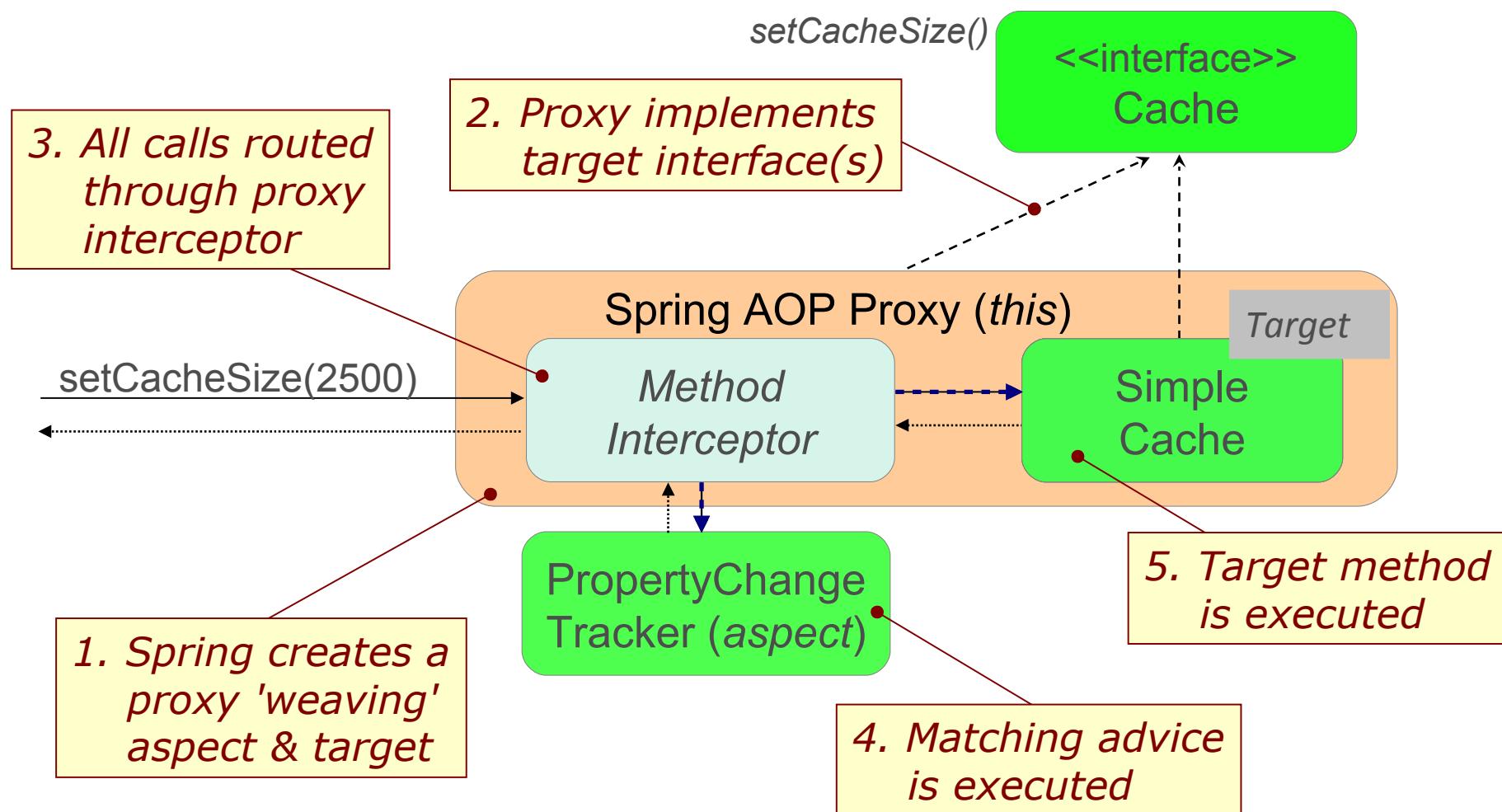
# Test the Application

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

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

INFO: Property about to change...

# How Aspects are Applied



# Which Setter is Proxied?

```
public class DatabaseCache 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);
}
```

YES – on **Cache** interface

NO – *not* on **Cache** interface

# Tracking Property Changes – With Context

- Context provided by the *JoinPoint* parameter

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

@Before("execution(void set*(*))")
public void trackChange(JoinPoint point) {
    String methodName = point.getSignature().getName();
    Object newValue = point.getArgs()[0];
    logger.info(methodName + " 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
- Lab
- Advanced Topics

# 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



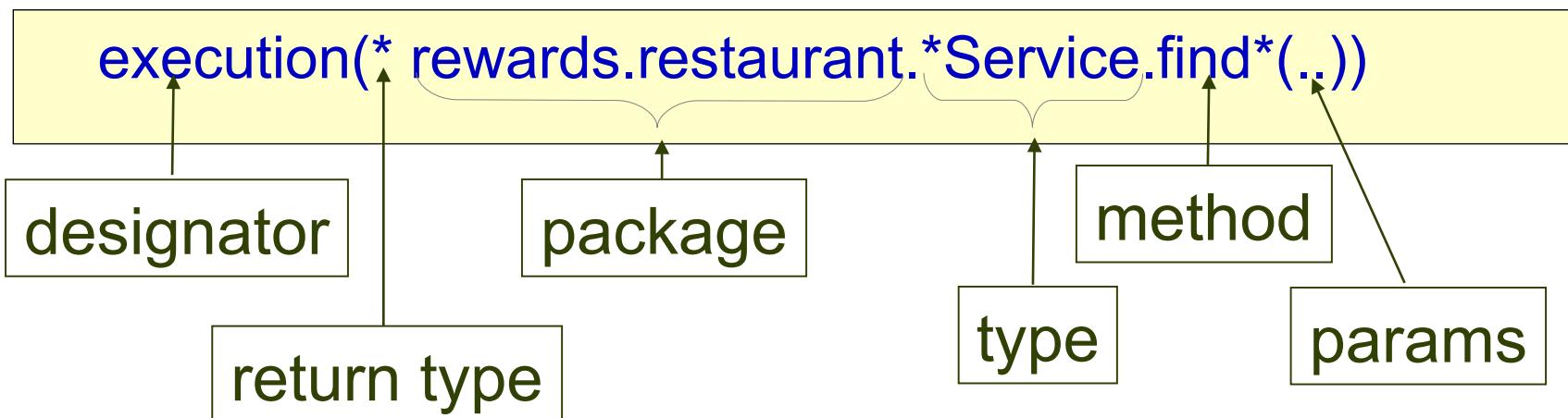
See: Spring Framework Reference – Declaring a Pointcut

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-api-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



## Wildcards:

- \* – matches once (return type, package, class/method name, argument)
- .. – matches zero or more (arguments or packages)

# Execution Expression Examples

## Any Class or Package

`execution(void send*(rewards.Dining))`

- Any method starting with send that takes a single Dining parameter and has a void return type
- Note use of *fully-qualified classname*

`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

## Implementations vs Interfaces

- Restrict by *class*

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

- Any void method in the *MessageServiceImpl* class
    - Including any sub-class
  - But will be ignored if a different implementation is used

- Restrict by *interface*

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

- Any void *send* method taking one argument, in any object implementing *MessageService*
  - More flexible choice – works if implementation changes

# Execution Expression Examples

## Using Annotations

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

- Any void method whose name starts with “send” that is annotated with the `@RolesAllowed` annotation

```
public interface Mailer {  
    @RolesAllowed("USER")  
    public void sendMessage(String text);  
}
```

- Ideal for your own classes
  - Matches if annotation is present
  - Not if it isn't

# 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

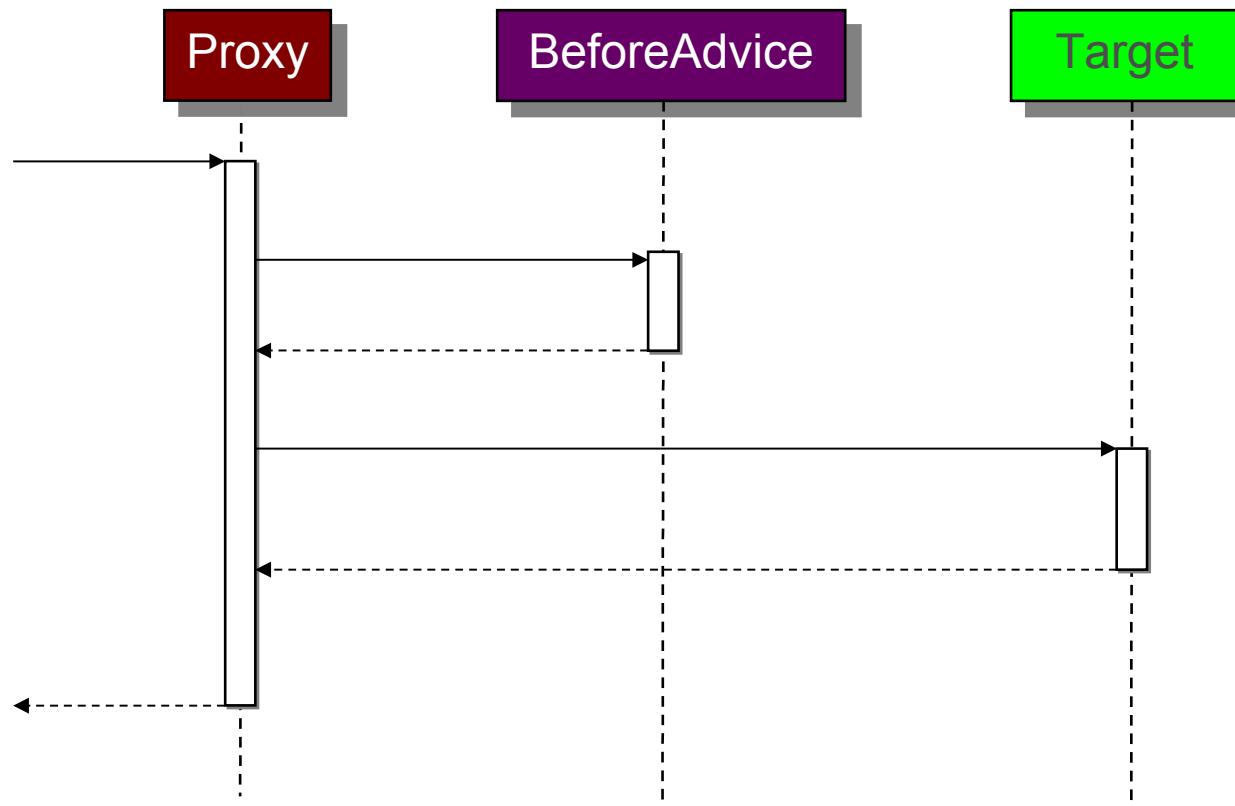
`execution(* *..restaurant.*.*(..))`

- Any sub-package called restaurant

# Topics in this session

- What Problem Does AOP Solve?
- Core AOP Concepts
- Quick Start
- Defining Pointcuts
- **Implementing Advice**
- Lab
- Advanced Topics

# Advice Types: Before



# Before Advice Example

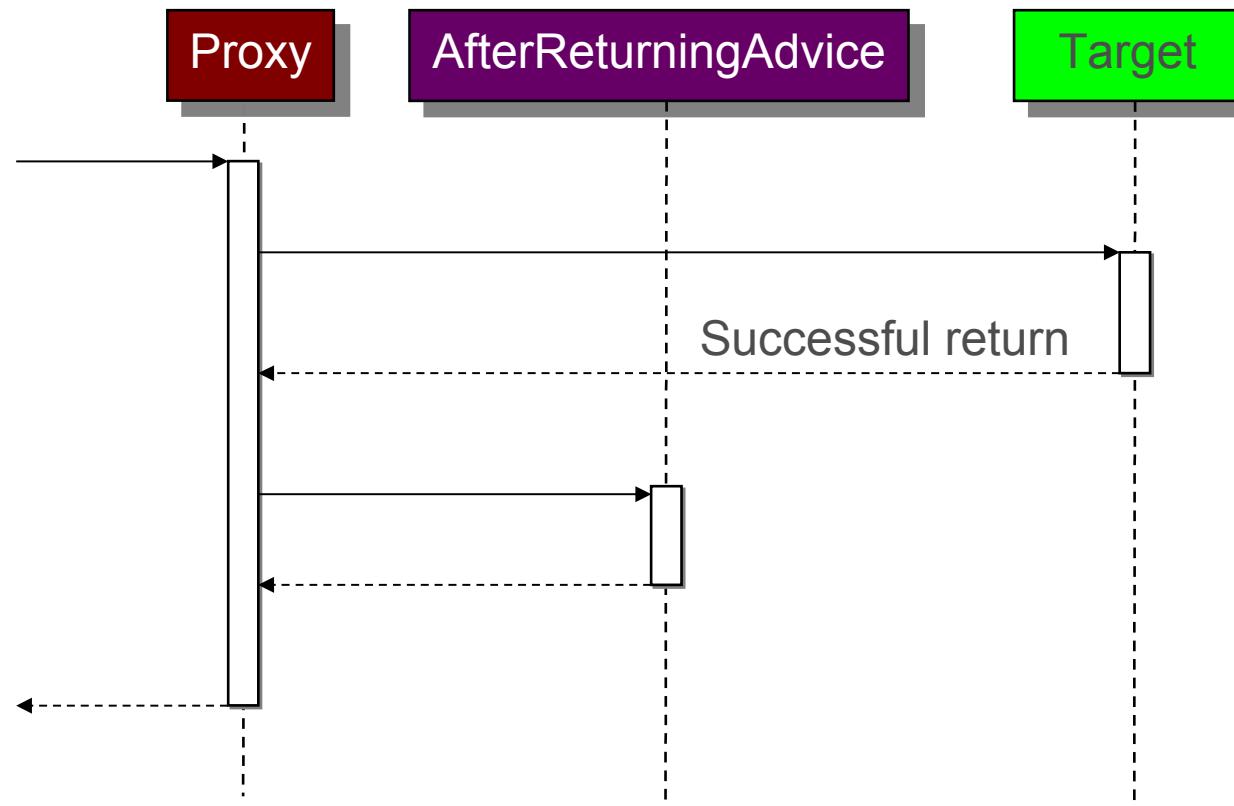
- Use `@Before` annotation

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...");  
    }  
}
```

- **Note:** if the advice throws an exception, target will not be called – this is a valid use of a *Before Advice*

# Advice Types: After Returning



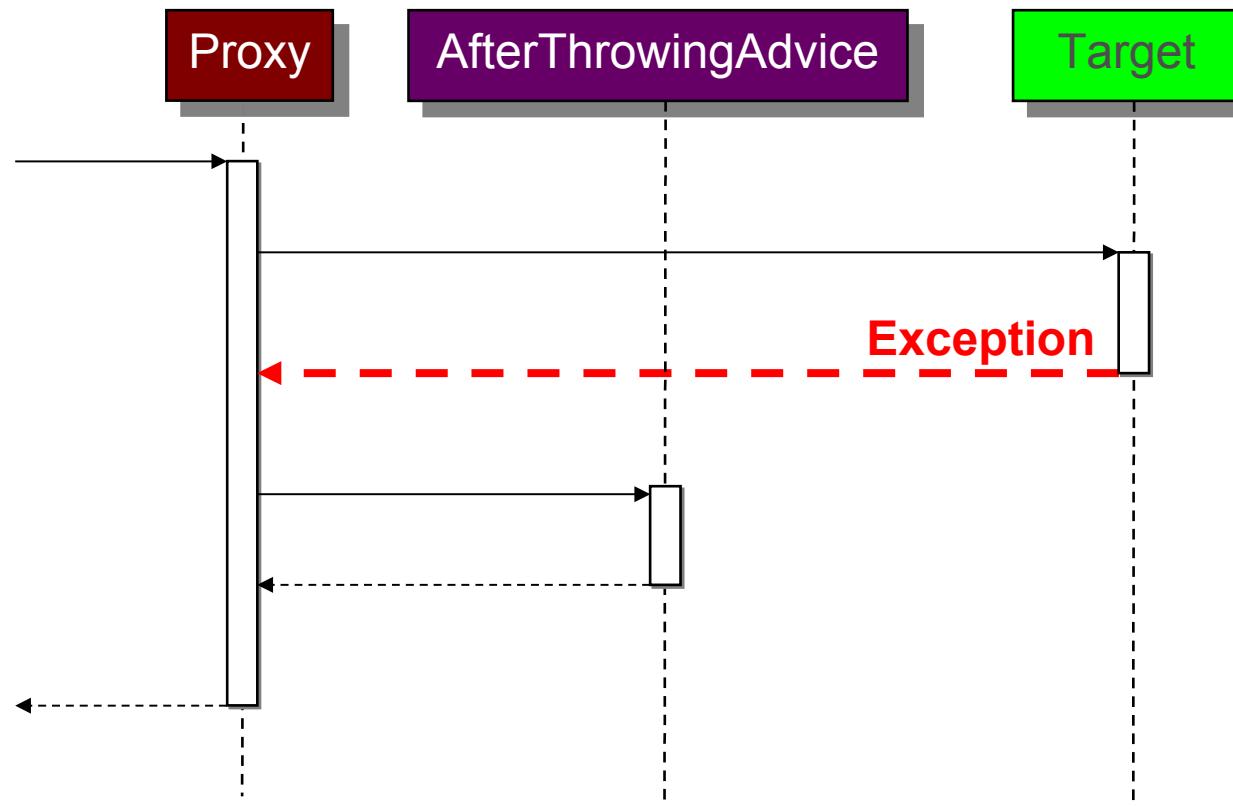
# After Returning Advice - Example

- 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
  - Only invokes advice if the right exception type is thrown

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

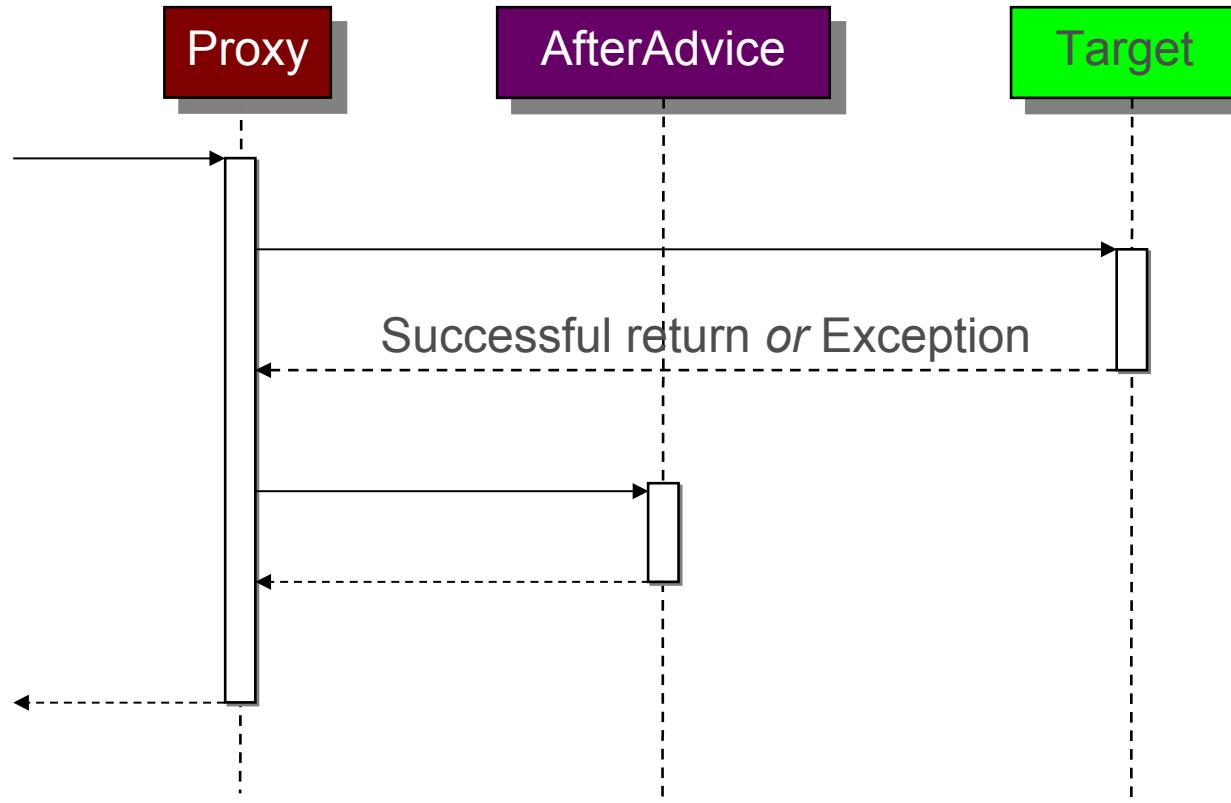
- The @AfterThrowing advice will not stop the exception from propagating
  - However it can throw a different type of exception

```
@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

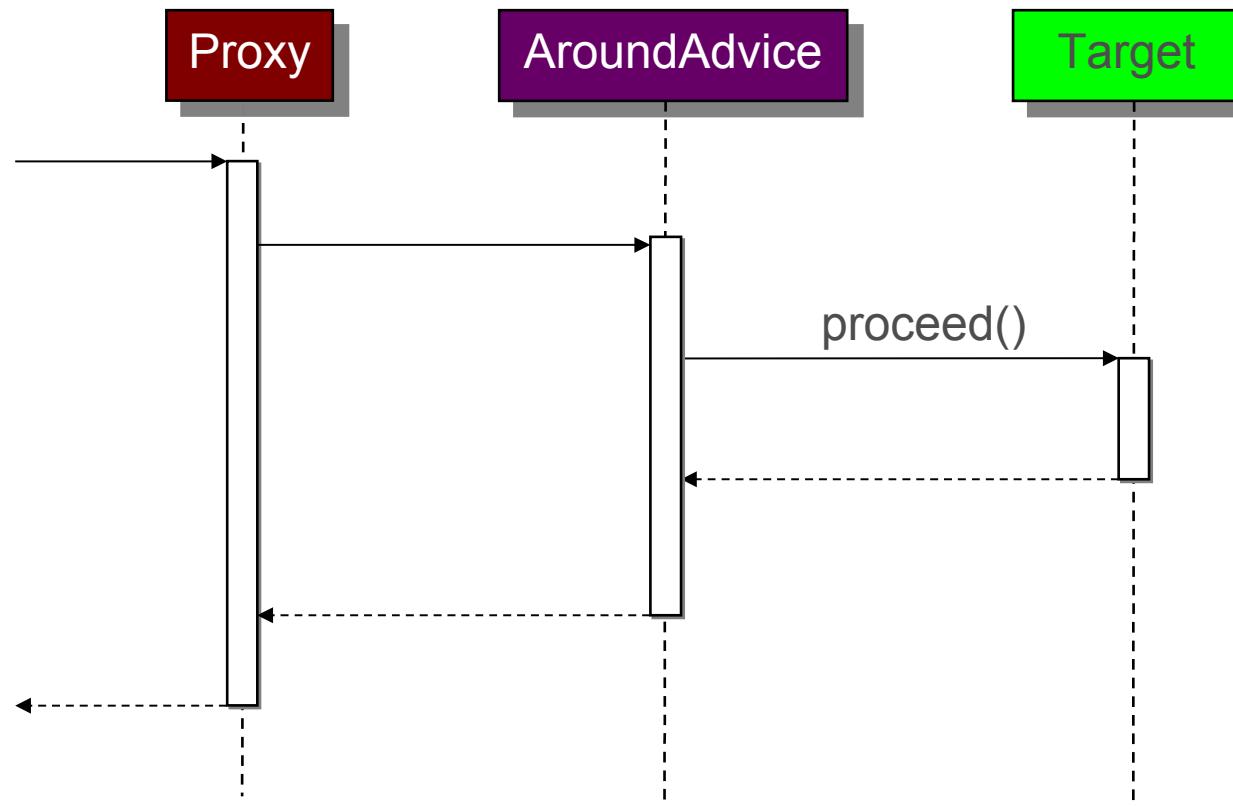
- Use `@After` annotation
  - Called regardless of whether an exception has been thrown by the target or not

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 and a `ProceedingJoinPoint`
  - Inherits from `JoinPoint` and adds the `proceed()` method

```
@Around("execution(@example.Cacheable * rewards.service..*.*(..))")
public Object cache(ProceedingJoinPoint point) throws Throwable {
    Object value = cacheStore.get(CacheUtils.toKey(point));

    if (value != null)
        return value;

    value = point.proceed();
    cacheStore.put(CacheUtils.toKey(point), value);
    return value;
}
```

Value exists? If so just return it

Proceed only if not already cached

Cache values returned by *cacheable* services

# 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()

# Summary



- Aspect Oriented Programming (AOP) *modularizes cross-cutting concerns*
- An aspect is a module (Java class) containing the cross-cutting behavior
  - Annotated with `@Aspect`
  - Behavior is implemented as an “advice” method
  - Pointcuts select *joinpoints* (methods) where advice applies
  - Five advice types
    - Before, AfterThrowing, AfterReturning, After and Around

# Lab

## Developing Aspects using Spring AOP

**Note:** The lab is working when

- 1) your unit test is green *and*
- 2) you get console logging output

**Coming Up:** Named pointcuts, context selection, annotations in pointcuts

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

```
@Aspect  
public class ServiceMethodInvocationMonitor {  
    private Logger logger = Logger.getLogger(getClass());  
  
    @Before("com.acme.Pointcuts.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 target object
  - Method arguments
  - Annotations associated with the method, target, or arguments
  - The currently executing object (proxy)
- 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 also 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

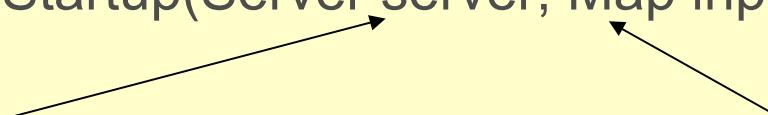
- Best practice: use context selection
  - Method attributes are bound automatically
  - Types must match or advice skipped

```
@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) {}
```

# Advanced Topics Summary

- Topics covered were:
  - Named Pointcuts
  - Context-Selecting Pointcuts
  - Working with Annotations

# Introduction to Data Management with Spring

## Implementing Data Access and Caching

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

# Objectives

- After completing this lesson, you should be able to:
  - Explain the Role of Spring in Enterprise Data Access
  - Use Spring's `DataAccessExceptionHierarchy`
  - Configure Test Databases
  - Implement Caching
  - Discuss NoSQL databases

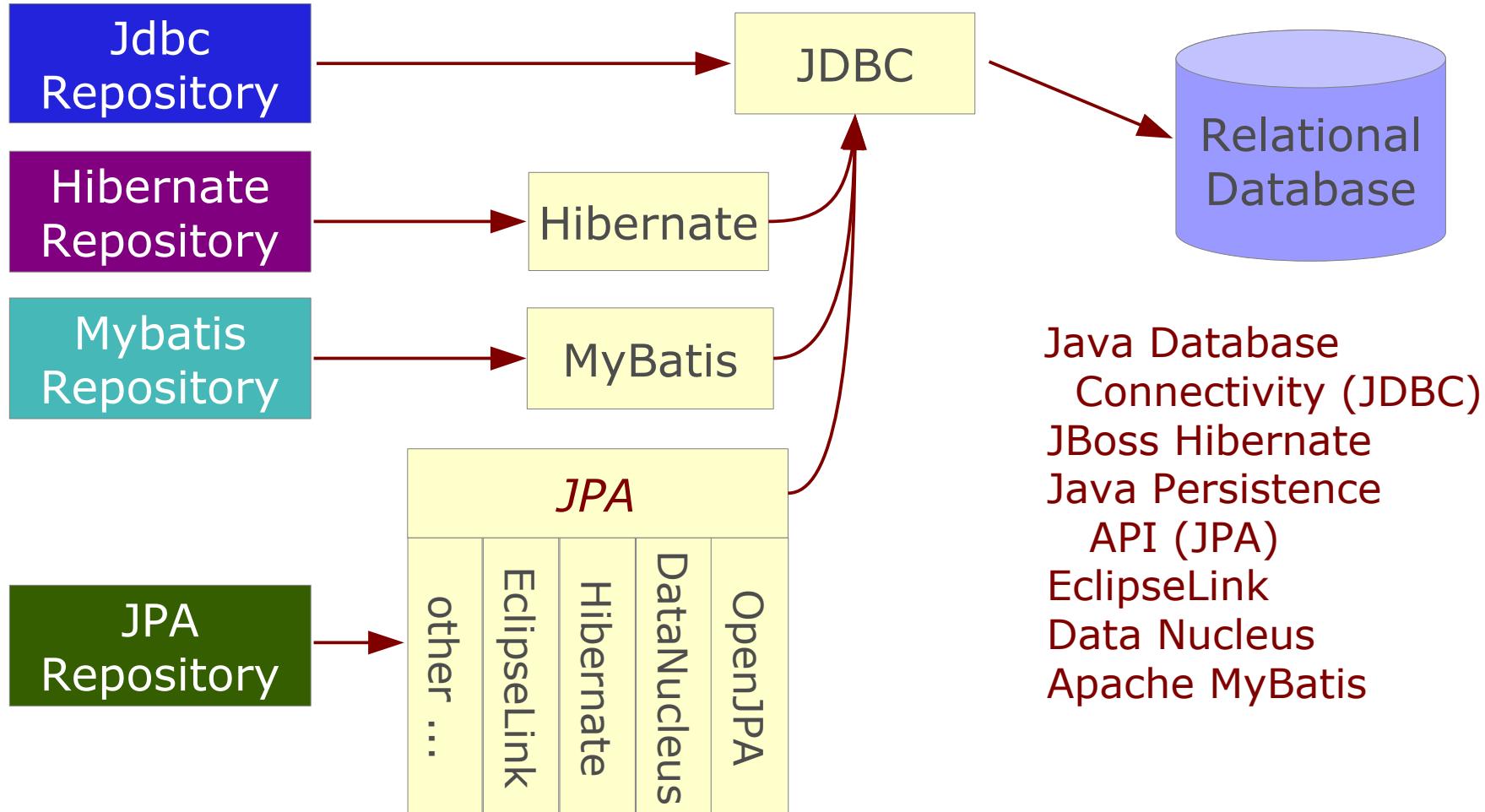


# Topics in this Session

- **The Role of Spring in Enterprise Data Access**
- Spring's **DataAccessExceptionHierarchy**
- Implementing Caching
- NoSQL databases

# Spring Resource Management Works Everywhere

*Works consistently with leading data access technologies*

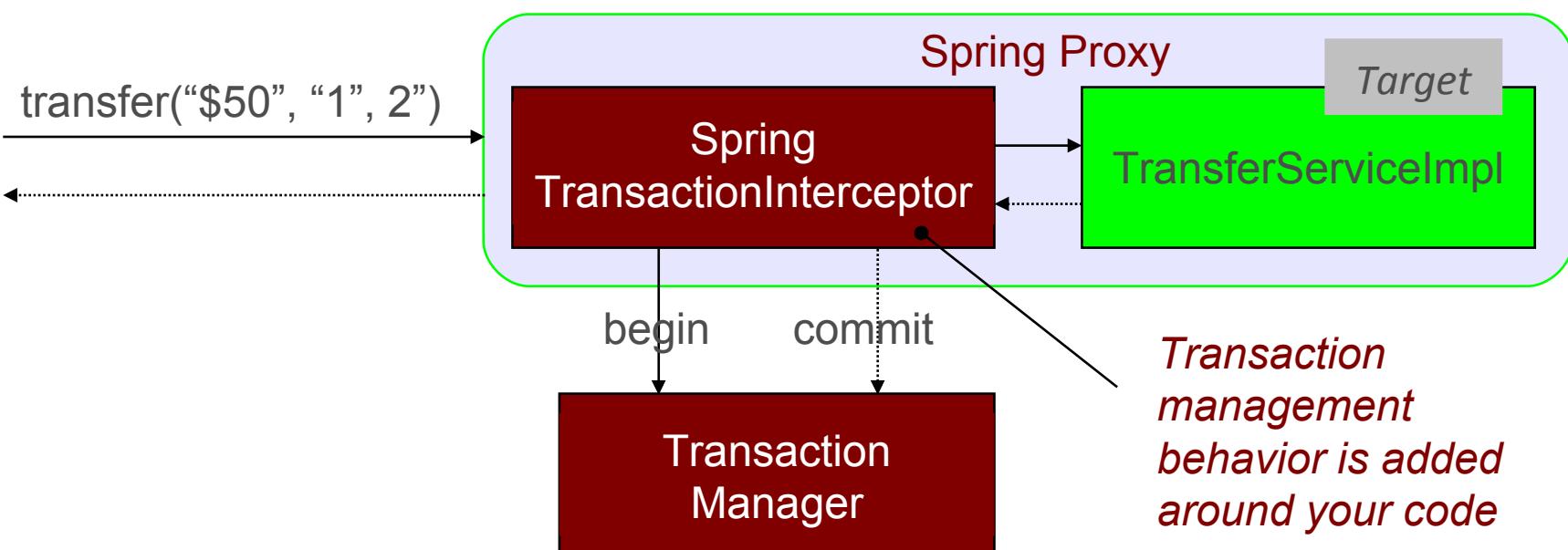


# The Resource Management Problem

- Steps Required
  - Access a data source and establish a connection
  - Begin a transaction
  - Do the work – execute business logic
  - Commit or rollback the transaction
  - Close the connection
- Spring Advantages
  - No code to implement (classic cross-cutting concern)
  - No connection or session leakage
  - Throws own exceptions, independent of underlying API

# Declarative Transaction Management

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



# 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



# Topics in this Session

- The Role of Spring in Enterprise Data Access
- **Spring's DataAccessExceptionHierarchy**
- Implementing Caching
- NoSQL databases

# Exception Handling

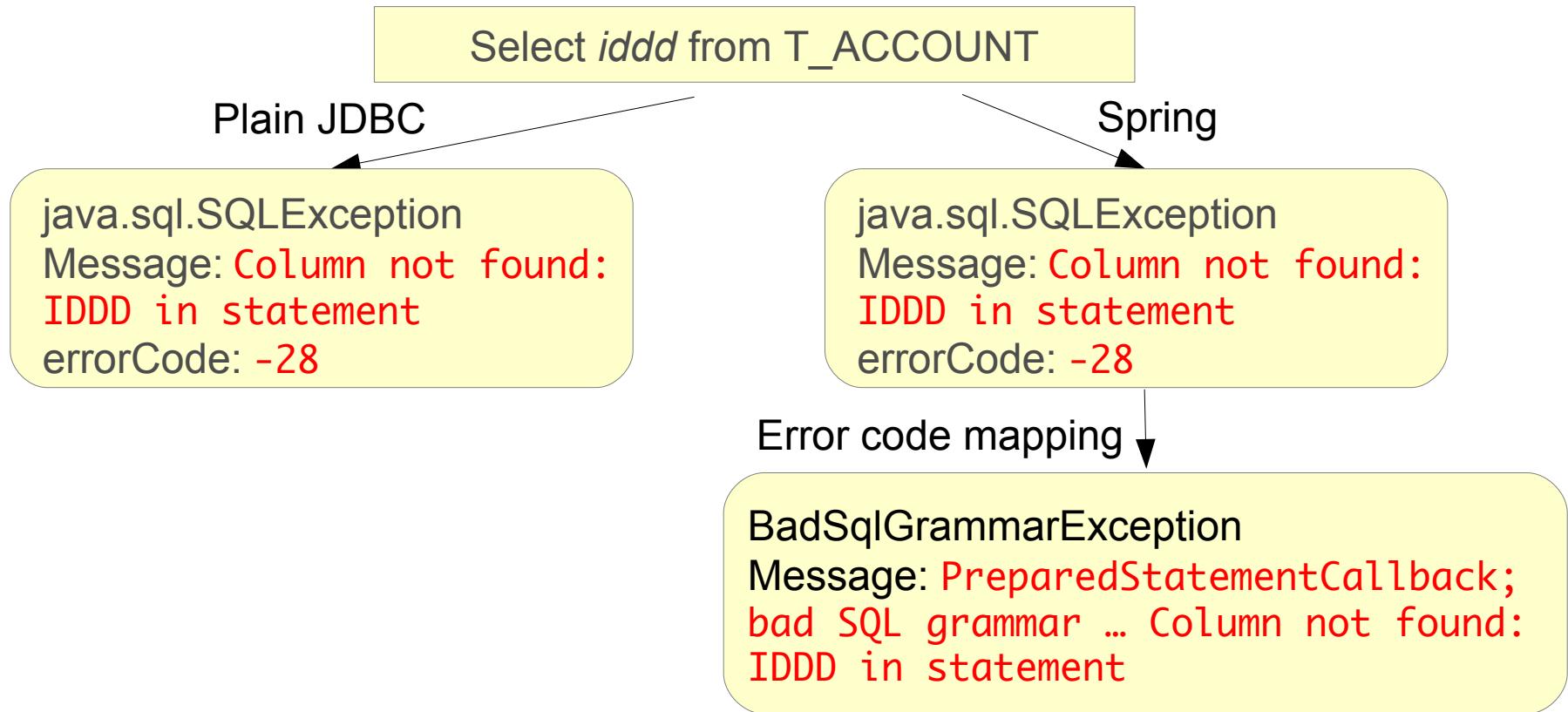
- Checked Exceptions
  - Force developers to handle errors
    - But if you can't handle it, must declare it
  - **Bad:** intermediate methods must declare exception(s) from *all* methods below
    - A form of tight-coupling
- Unchecked Exceptions
  - Can be thrown up the call hierarchy to the best place 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

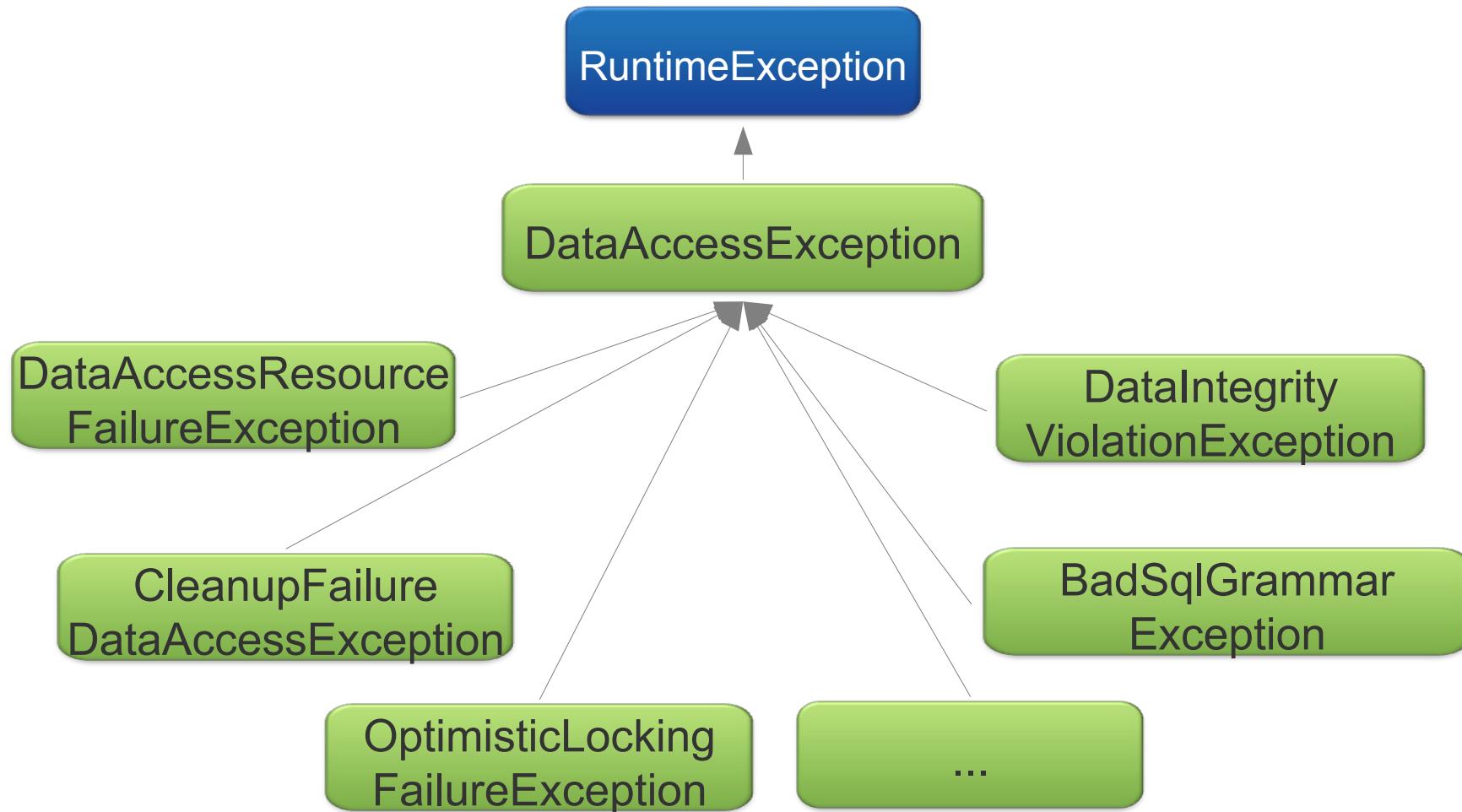
- SQLException
  - Too general – one exception for every database error
  - Calling class 'knows' you are using JDBC
  - Tight coupling
- Spring provides **DataAccessException** hierarchy
  - Hides whether you are using JPA, Hibernate, JDBC ...
  - Actually a hierarchy of sub-exceptions
    - Not just one exception for everything
  - Consistent across all supported Data Access technologies
  - Unchecked

# Example: *BadSqlGrammarException*



<https://github.com/spring-projects/spring-framework/blob/master/spring-jdbc/src/main/resources/org/springframework/jdbc/support/sql-error-codes.xml>

# Spring Data Access Exceptions



# Topics in this Session

- The Role of Spring in Enterprise Data Access
- Spring's `DataAccessExceptionHierarchy`
- **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

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



See: **Spring Framework Reference – Cache Abstraction**  
<https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#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="topBooks", key="#refId.toUpperCase()")  
public Book findBook(String refId) { ... }
```

# Caching via Annotations

```
public class BookService {
```

Use 'topBooks' cache

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

Only cache if condition true

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

use object  
property

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

```
@CacheEvict(value="topBooks", beforeInvocation=true)  
public void loadBooks() { ... }
```

custom key  
generator

clear cache *before* method invoked

# Enabling Caching Proxy

- Caching must be enabled ...

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

# Setup Cache Manager

- Must specify a cache-manager
  - Some provided, or write your own
  - See `org.springframework.cache` package
- SimpleCacheManager
  - For each cache name, it creates a **ConcurrentHashMap**

```
@Bean  
public CacheManager cacheManager() {  
    SimpleCacheManager cacheManager =  
        new SimpleCacheManager("topAuthors", "topBooks");  
    return cacheManager;  
}
```

Keep `cacheManager`  
bean name

Concurrent Map Cache

# Third-Party Cache Implementations

- Simple Cache is OK for testing
  - But has no cache control options (overflow, eviction)
- Third-party alternatives
  - Terracotta's EhCache
  - Pivotal's Gemfire
  - Google's Caffeine
  - Infinispan
  - Hazelcast
  - Redis
  - Couchbase

# Third-Party Cache Manager – EHCache

- Must specify EhCache's XML configuration file

```
@Autowired  
ApplicationContext context;
```

```
@Value("${ehcache.xml.location}")  
String location;
```

```
@Bean  
public CacheManager cacheManager() {  
    Resource cacheConfig = context.getResource(location);  
    net.sf.ehcache.CacheManager cache =  
        EhCacheManagerUtils.buildCacheManager(cacheConfig);  
    return new EhCacheCacheManager(cache);  
}
```



Location of `ehcache.xml`

Supports `file:` and  
`classpath:` prefixes

# Third-Party Cache Managers – Gemfire

- Gemfire: A distributed, shared nothing data-grid
  - Can be used to setup a distributed cache
  - Caches (regions) replicated across multiple nodes
    - Consistent updates occur on all copies in parallel
    - No loss of data if a storage node fails
    - Automatic recovery and rebalancing



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

Pivotal Gemfire  
Cache

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

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

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

# Spring Gemfire Project



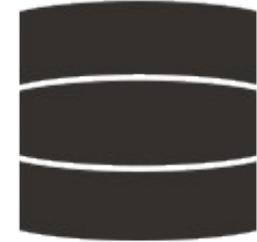
- GemFire configuration in Spring config files
  - Also enables configuration injection for environments
- Features
  - Exception translation
  - GemfireTemplate
  - Transaction management (*GemfireTransactionManager*)
  - Injection of transient dependencies during deserialization
  - *Gemfire Cache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

**GEMFIRE®**

# Topics in this Session

- The Role of Spring in Enterprise Data Access
- Spring's `DataAccessExceptionHierarchy`
- Implementing Caching
- `NoSQL databases`

# Not Only Relational

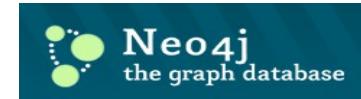


- NoSQL
  - Relational databases only store some data
    - LDAP, data-warehouses, files
    - Most documents and spreadsheets aren't in *any* database
  - Other database products exist
    - Have strengths where RDB are weak
      - Non-tabular data
        - Hierarchical data: parts inventory, org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

**SPRING DATA**

# So Many Choices ...

- 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

# Objectives

- After completing this lesson, you should be able to:
  - Explain the problems with traditional JDBC
  - Use and configure Spring's `JdbcTemplate`
  - Execute queries using callbacks to handle result sets
  - Handle Exceptions



# 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**  
<https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#jdbc>

# Redundant, Error Prone Code

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

# Redundant, Error Prone Code

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

The bold matters - the  
rest is *boilerplate*

# Redundant, Error Prone Code

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

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

# 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

# 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 exceptions)

# 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 *ad hoc* reporting, testing use cases
  - The data fetched does not need mapping to a Java object



*ad hoc* – created or done for a particular purpose as necessary

- sometimes called “window-on-data” queries

# 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 to define its return-type

```
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 a *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, int year) {  
        // select ALL orders of given year for a full report  
        jdbcTemplate.query("select * from order where year=?",  
            new OrderReportWriter(out), year);  
    }  
}
```

Method returns “void”

```
class OrderReportWriter implements RowCallbackHandler {  
    public void processRow(ResultSet rs) throws SQLException {  
        // parse current row from ResultSet and stream to output  
    }  
    // May also be a stateful object: you could accumulate data and add  
    // convenience methods like getTotalOrderValue(), getCount() ...  
}
```

# Using a RowCallbackHandler (2)

- Might use a Lambda – if no state needed
  - Cast tells Java which callback class the lambda replaces

Or using a *lambda*

```
public class JdbcOrderRepository {  
    public void generateReport(final PrintWriter out) {  
        // select all orders of year 2009 for a full report  
        jdbcTemplate.query("select * from order where year=?",  
            (RowCallbackHandler)(rs) ->  
                out.write(rs.getString("customer"), ...), 2016);  
    }  
}
```

Cast needed

```
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;  
}
```



You may need this for the lab!

# 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 = ?",
            (ResultSetExtractor<Order>)(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, especially large queries
- 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)

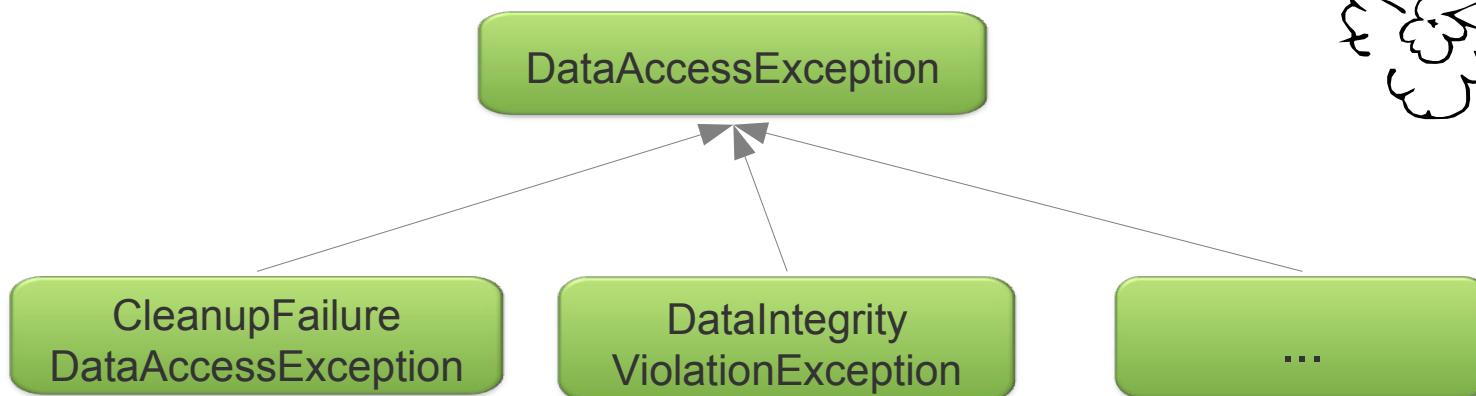
- Updating an existing row

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

Any non-SELECT SQL is run using update()

# Exception Handling

- The JdbcTemplate transforms SQLExceptions into DataAccessExceptions



*DataAccessException* hierarchy was discussed in module “Introduction to Data Management” (refer to it for more information on this topic).

# Summary

- JDBC is useful
  - But using JDBC API directly is tedious and error-prone
- *JdbcTemplate* simplifies data access and enforces consistency
  - DRY principle hides most of the JDBC
  - Many options for reading data
- *SQLExceptions* typically cannot be handled where thrown
  - Should not be *checked* Exceptions
  - Spring provides *DataAccessException* instead

# Lab

Reimplementing repositories using  
Spring's JdbcTemplate

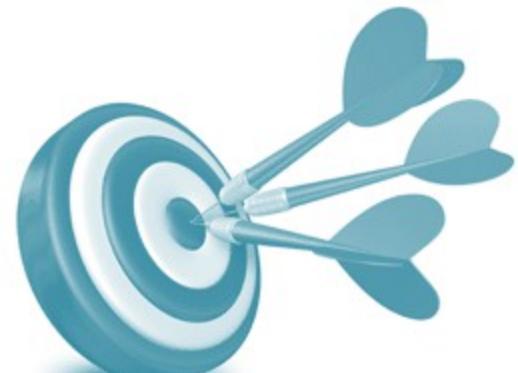
# Transaction Management with Spring

Spring's Consistent Approach

Transactional Proxies and @Transactional

# Objectives

- After completing this lesson, you should be able to:
  - Explain why Transactions are used
    - And how Java supports them in different ways
  - Describe and use Spring Transaction Management
  - Configure Isolation Levels and Transaction Propagation
  - Setup Rollback rules
  - Use Transactions in Tests



# Topics in this session

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

# What is a Transaction?

- A set of tasks which take place as a single, indivisible action
  - An *Atomic, Consistent, Isolated, Durable* operation
  - Acronym: **ACID**



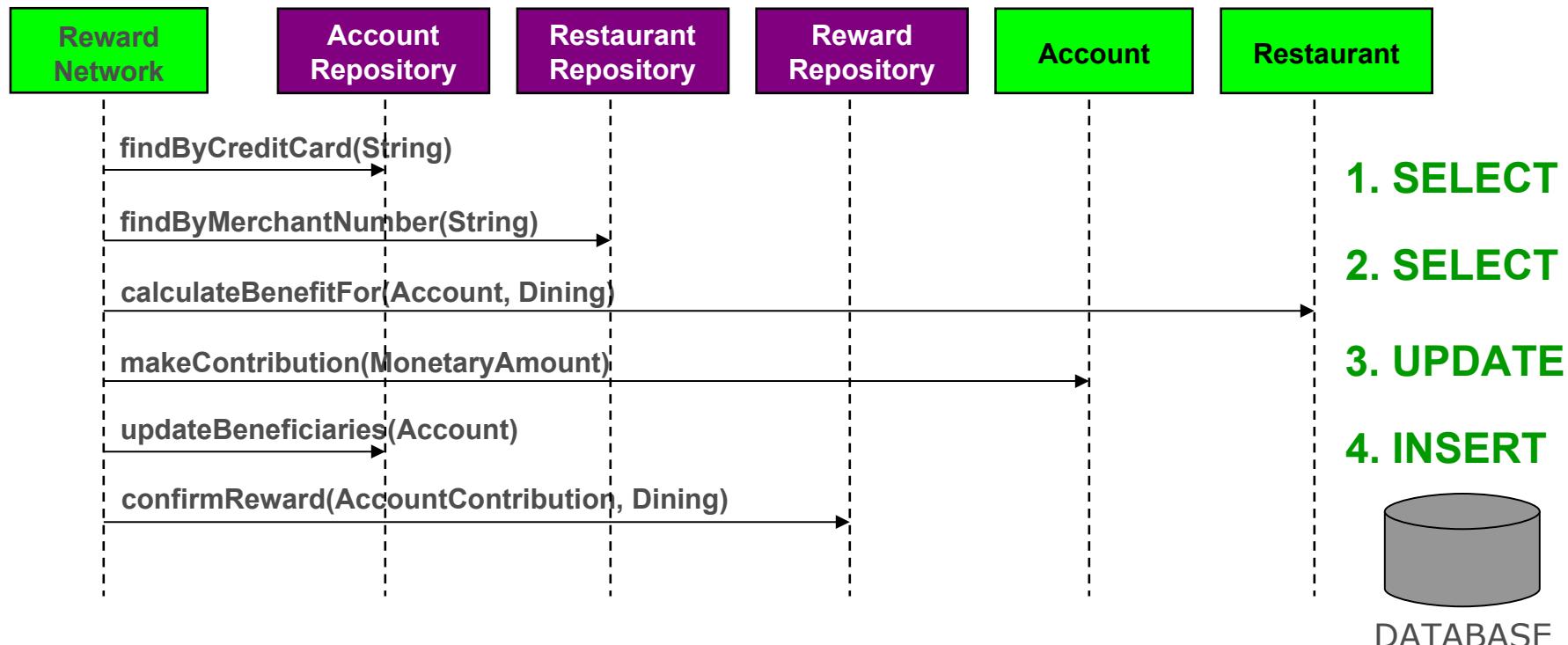
# Why use Transactions?

## 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

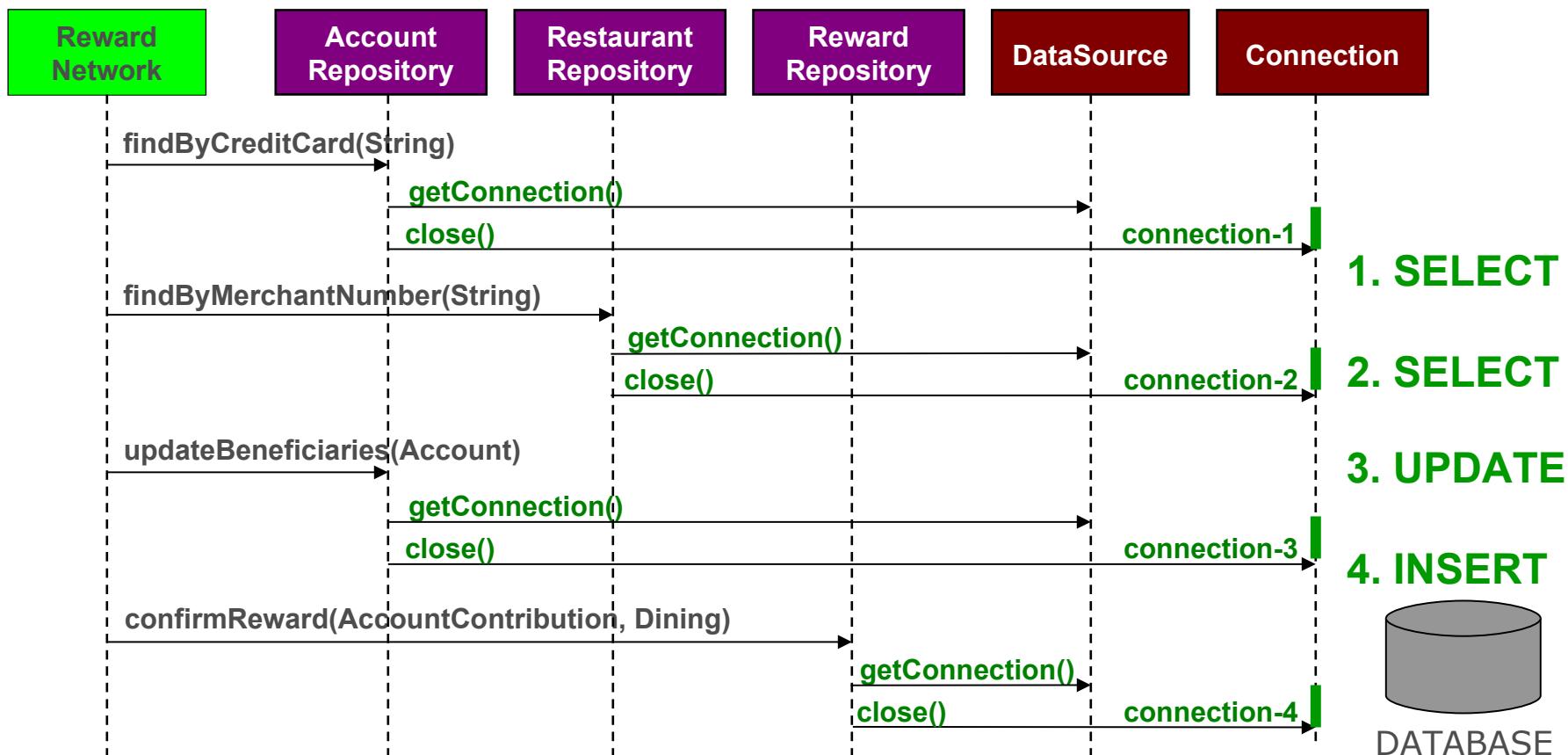


# Naïve Approach

## Connection per Data Access Operation

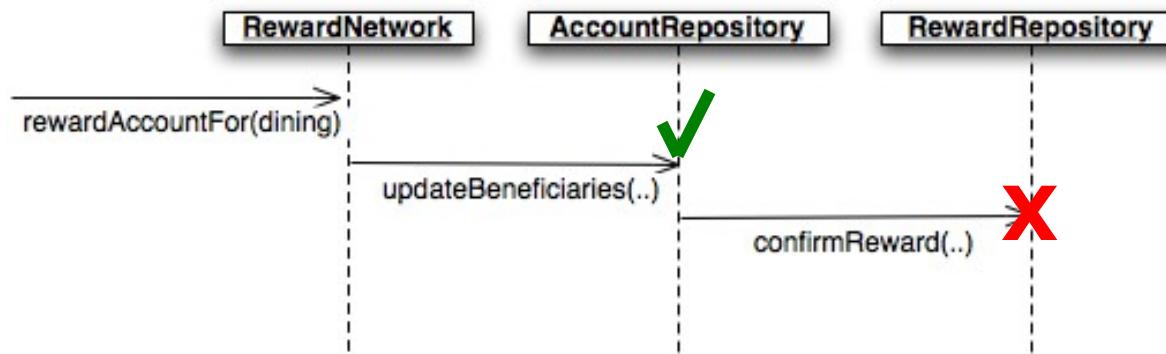
- This unit-of-work contains 4 data access operations
  - Each acquires, uses, and releases a distinct Connection
- The unit-of-work is *non-transactional*

# 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!

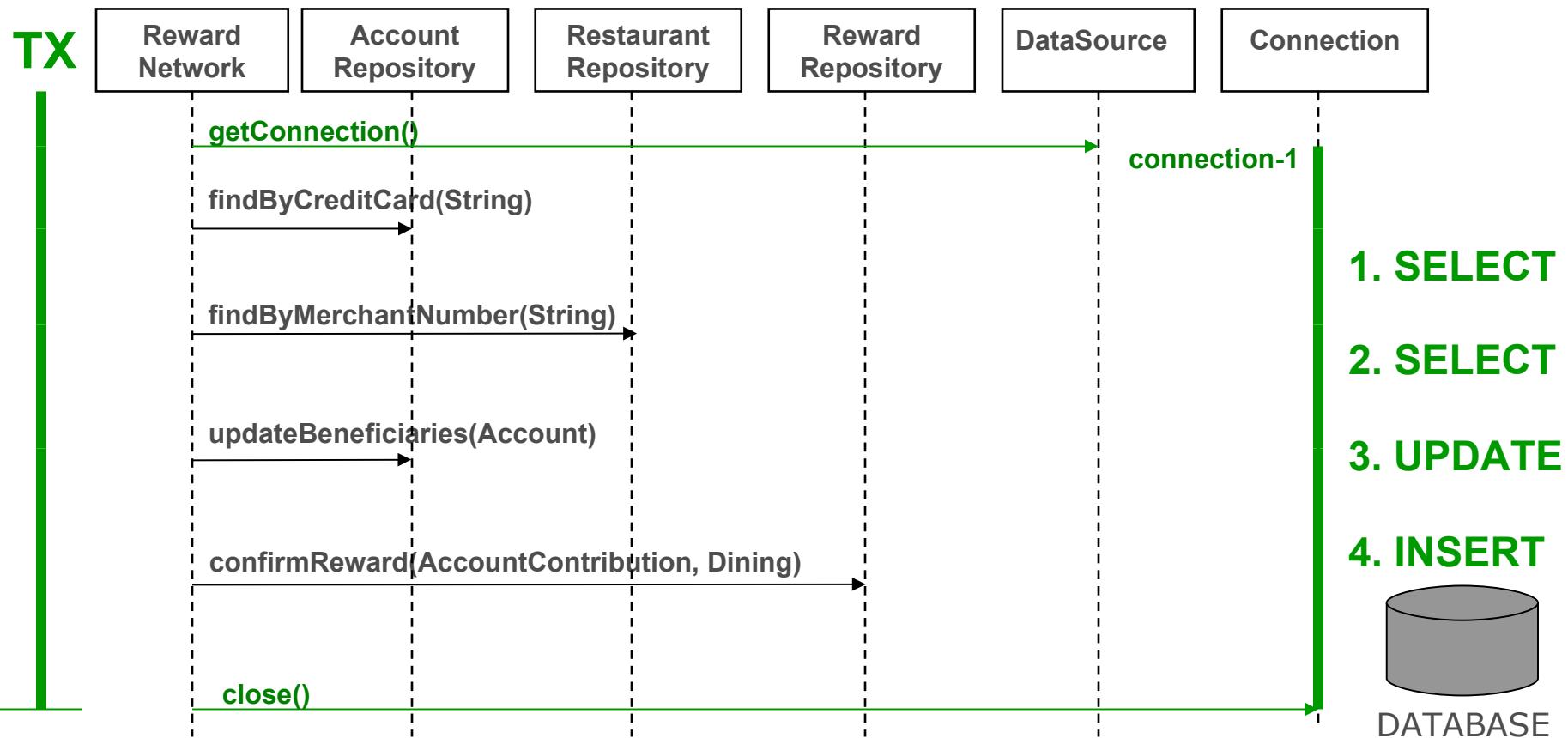
The unit-of-work  
is **not atomic**

# 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
- Isolation Levels
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

# Java Transaction Management

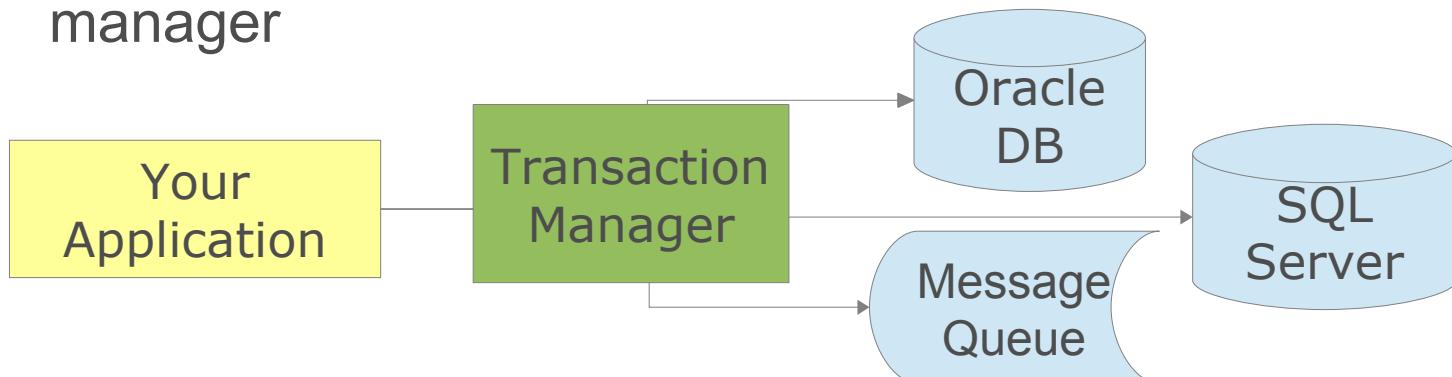
- Java has several APIs which handle transactions differently
  - JDBC, JMS, JTA, Hibernate, JPA, etc.
- Each uses program code to mark the start and end of the transaction
  - Transaction Demarcation
- Different APIs for Global vs Local transactions

# 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();  
    conn.setAutoCommit(false);  
  
    ...  
  
    conn.commit();  
} catch (Exception e) {  
  
    conn.rollback();  
    ...  
}
```

JDBC specific API

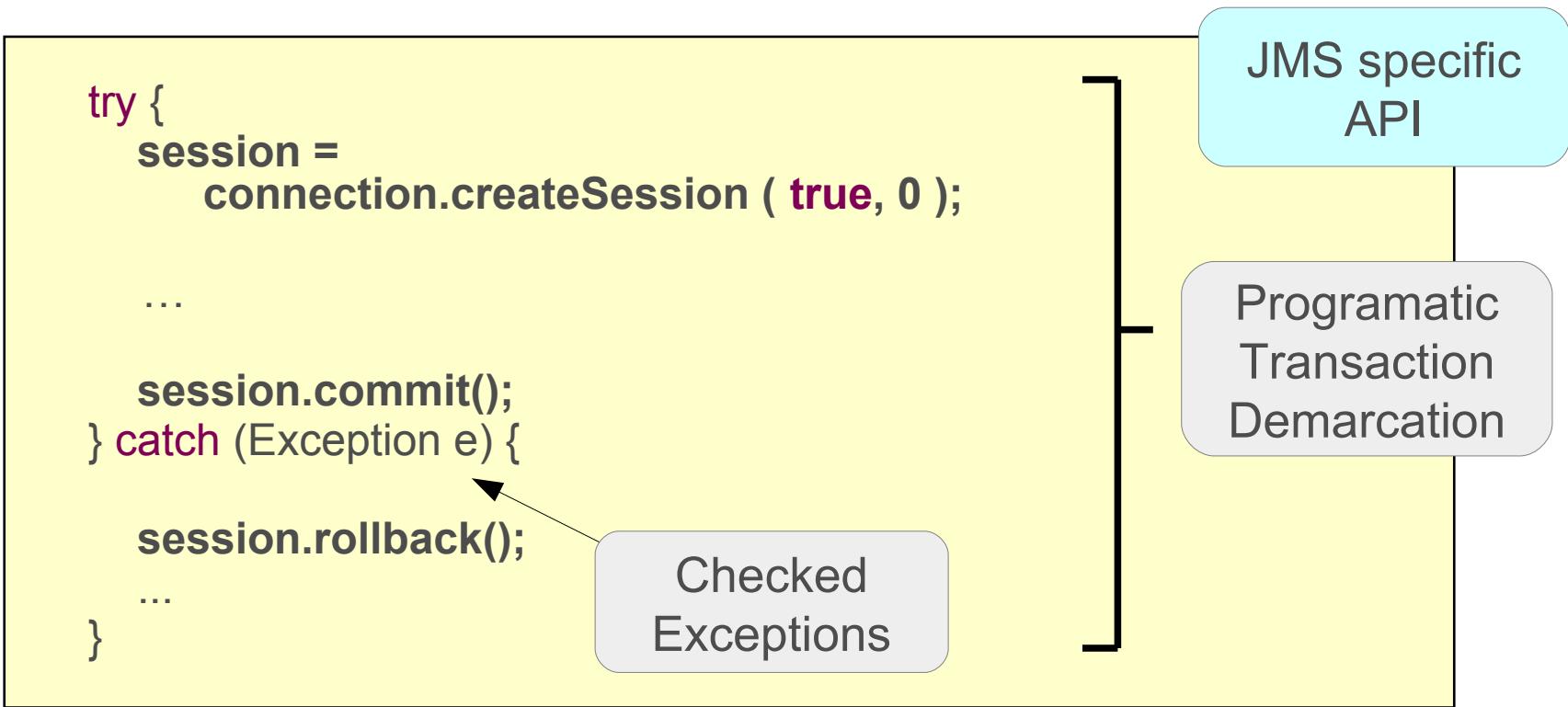
Programmatic Transaction Demarcation

Checked Exceptions

*Local transactions only:*

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

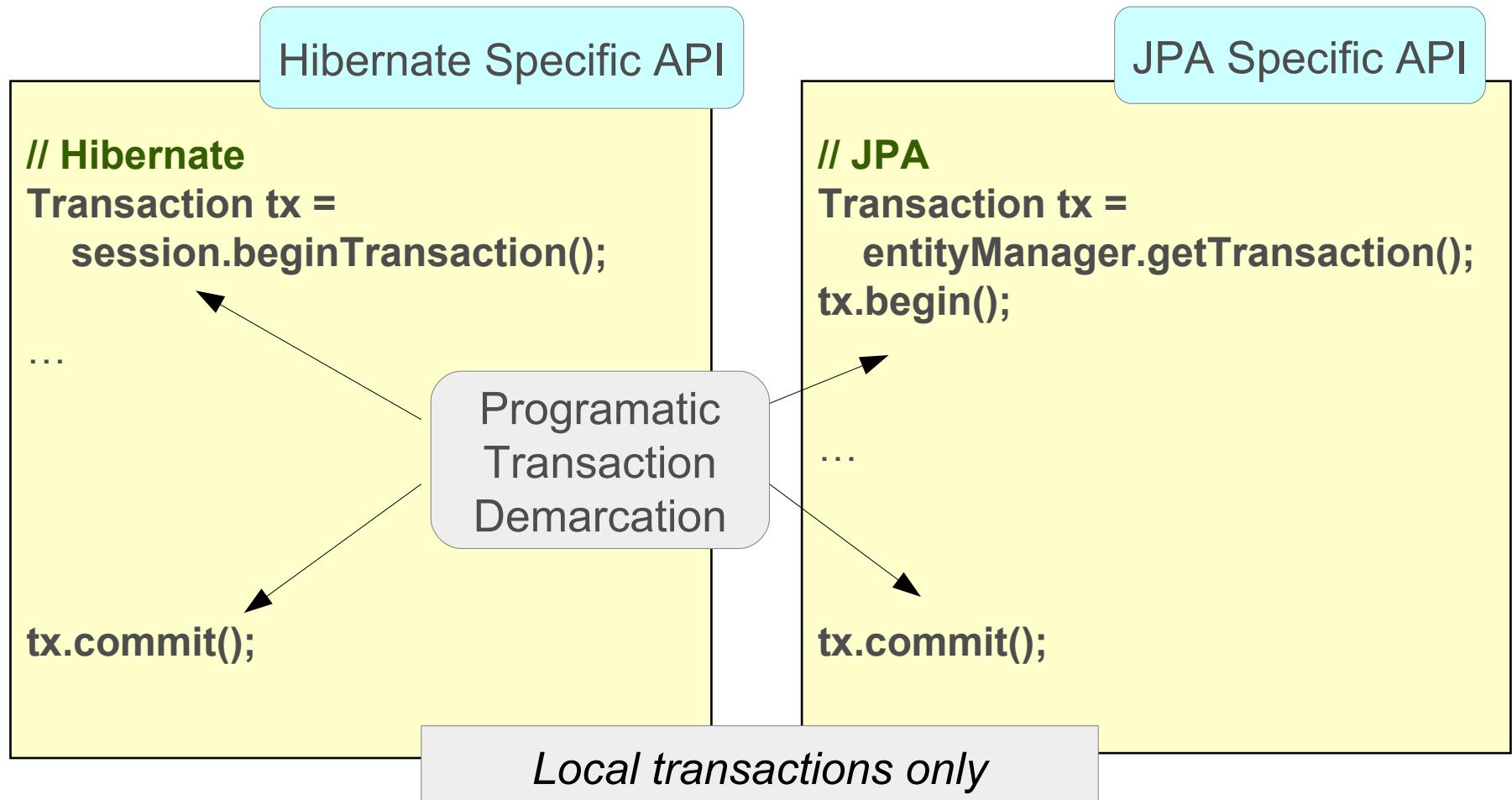
# JMS Transaction Management Example



*Local transactions only:*

- *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();  
    ...  
}
```

JTA Specific API

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
  - Typically performed in the repository layer (wrong place)
  - Usually repeated (cross-cutting concern)
- Service layer more appropriate
  - Multiple data access methods often called within a single transaction
  - But: don't want data-access code in service-layer
- Orthogonal concerns
  - Transaction demarcation should be independent of transaction implementation

# Topics in this session

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

# Spring Transaction Management – 1

- 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
    - Just change the transaction manager

# Spring Transaction Management – 2

- 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

- Create the required implementation
  - Just like any other Spring bean
    - Configure as appropriate
  - Here is the manager for a DataSource

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

A DataSource  
bean must be  
defined elsewhere



Bean id “*transactionManager*” is recommended name. Spring is preconfigured to expect a bean with that name.

# Accessing a JTA Transaction Manager

- Use a JNDI lookup for container-managed DataSource
  - Or use container-specific subclasses:
    - `WebLogicJtaTransactionManager`
    - `WebSphereUowTransactionManager`

```
@Bean  
public PlatformTransactionManager transactionManager() {  
    return new JtaTransactionManager();  
}  
  
@Bean  
public DataSource dataSource(@Value("${db.jndi}") String jndiName) {  
    JndiDataSourceLookup lookup = new JndiDataSourceLookup();  
    return lookup.getDataSource(jndiName);  
}
```

# @Transactional Configuration

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

In your code

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

In your Spring configuration

Defines a Bean Post-Processor  
– proxies @Transactional beans

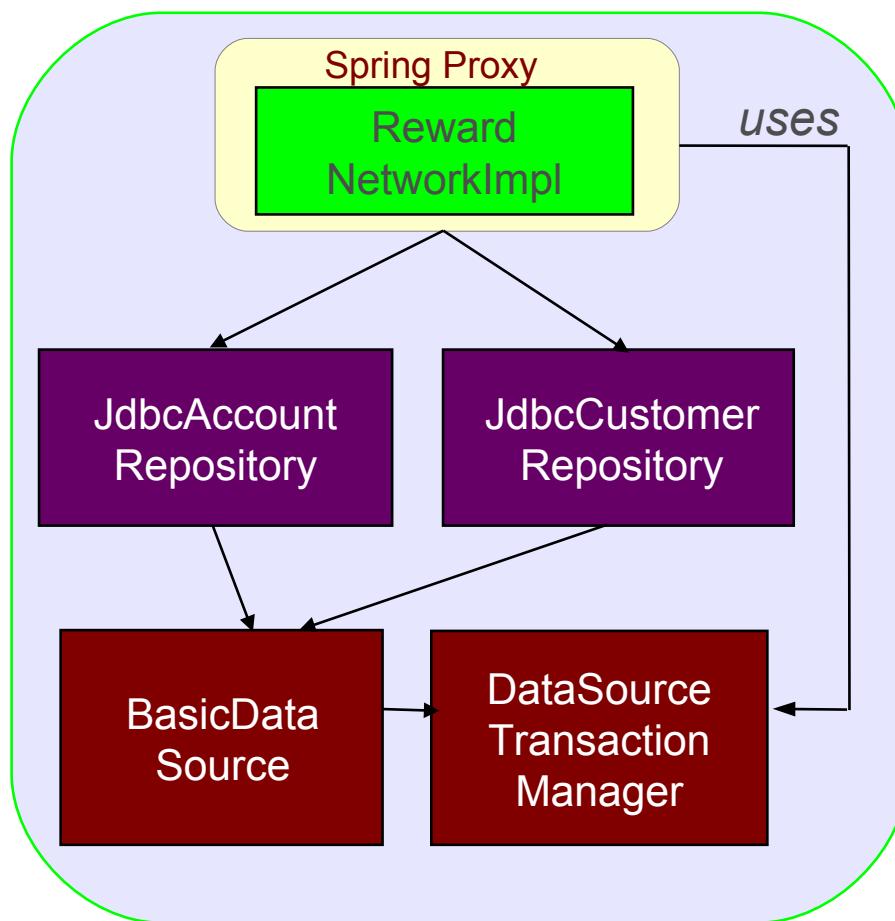
# @Transactional: What Happens Exactly?

- 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

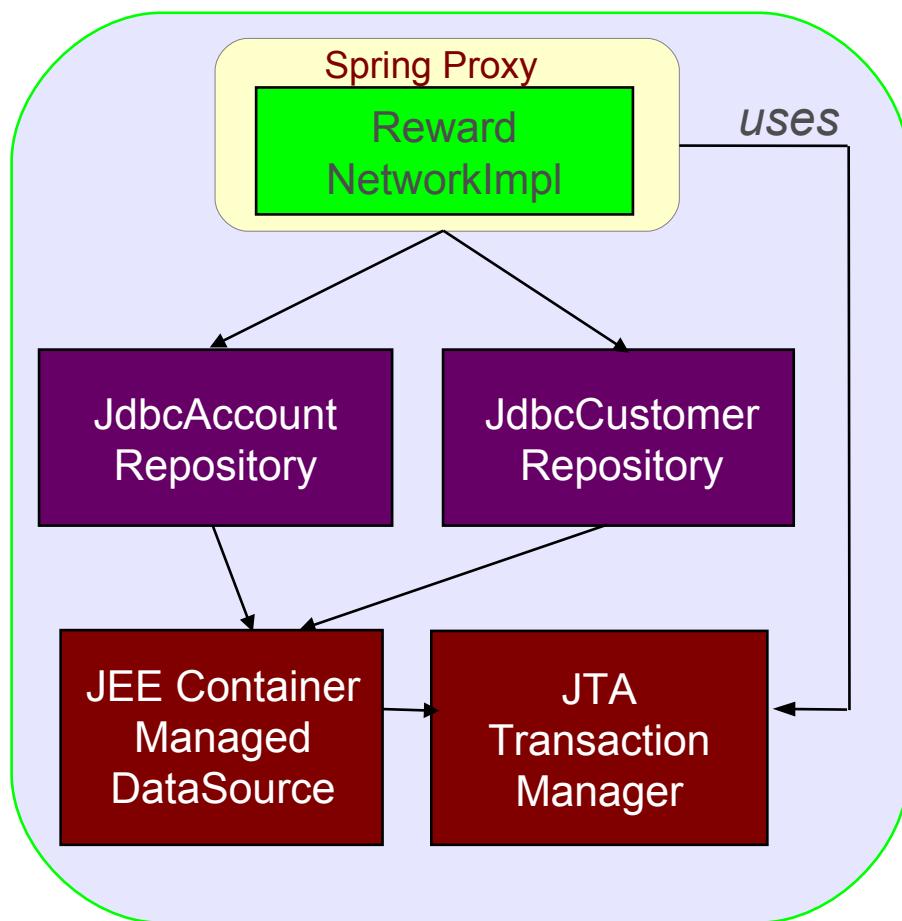
# Local JDBC Configuration



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

# JDBC Java EE Configuration

No code changes  
*Just configuration*



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

# @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  
    }  
}
```



Alternatively *@Transactional* can be declared on the interface instead  
– since Spring Framework 5.0

# @Transactional

## – Class and method levels

- Combining class and method levels

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

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

override attributes at method level

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

# Topics in this session

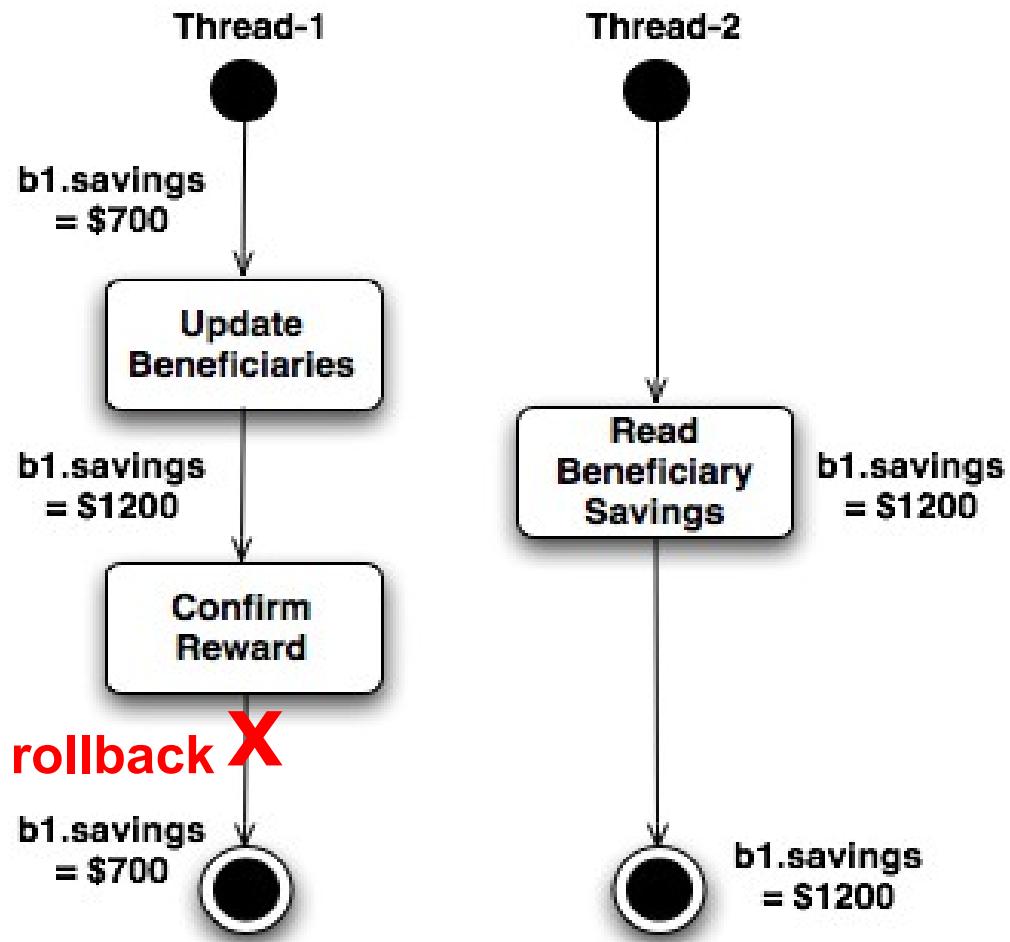
- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- **Isolation Levels**
- Transaction Propagation
- Rollback rules
- Testing
- Advanced topics

# Isolation levels

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

# Dirty Reads

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



# READ\_UNCOMMITTED

- Lowest isolation level – allows *dirty reads*
- Current transaction can see the results of another uncommitted unit-of-work
- Typically used for large, intrusive read-only transactions
- And/or where the data is constantly changing

```
public class RewardNetworkImpl implements RewardNetwork {  
    @Transactional (isolation=Isolation.READ_UNCOMMITTED)  
    public BigDecimal totalRewards(String merchantNumber, int year)  
        // Calculate total rewards for a restaurant for a whole year  
    }  
}
```

# 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
- Isolation Levels
- **Transaction Propagation**
- Rollback rules
- Testing
- Advanced topics

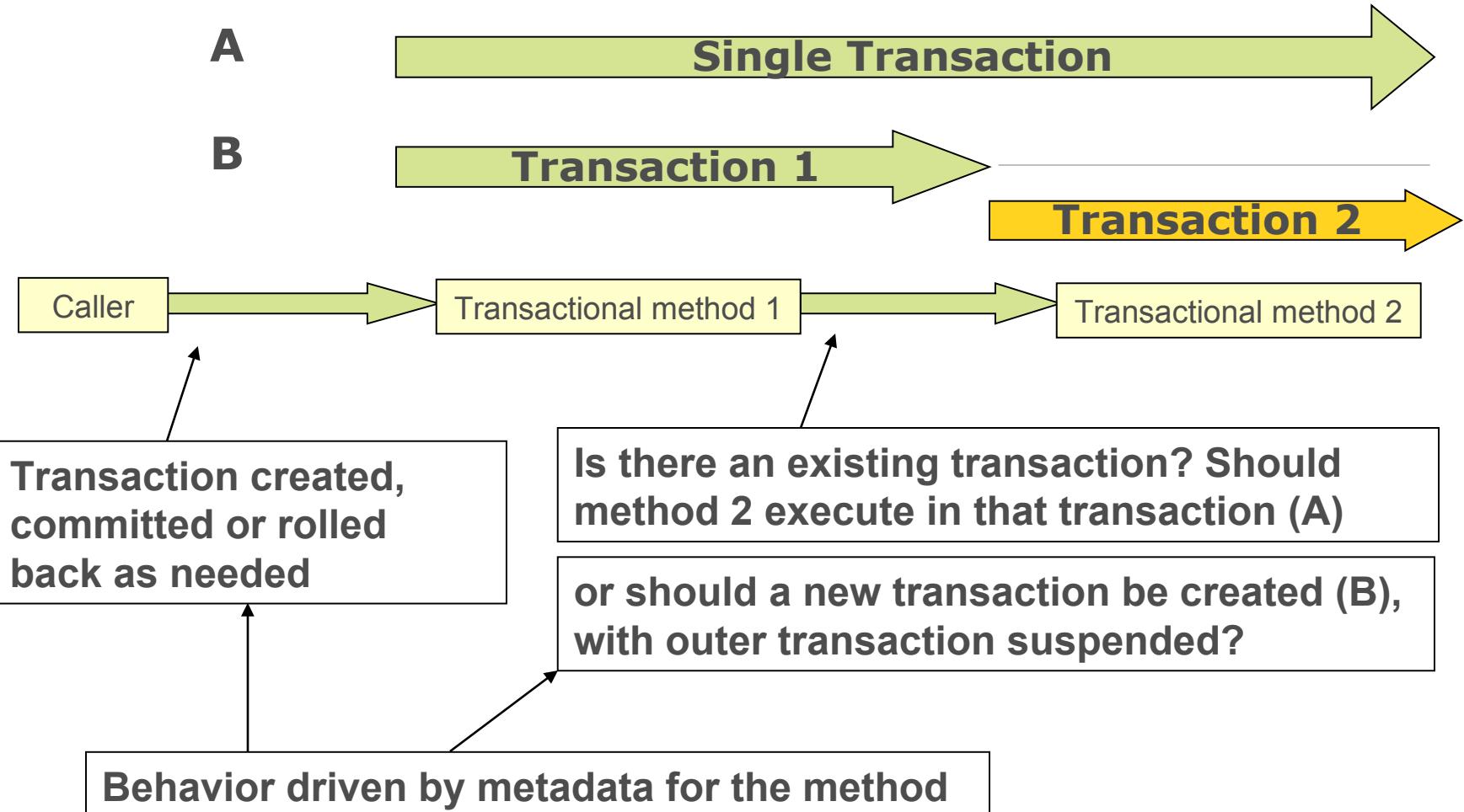
# Understanding Transaction Propagation

- 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



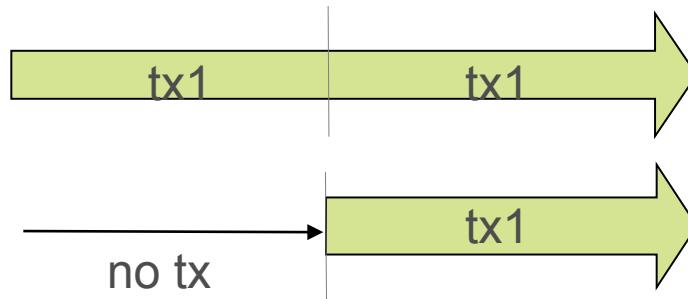
# Transaction Propagation with Spring

- 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

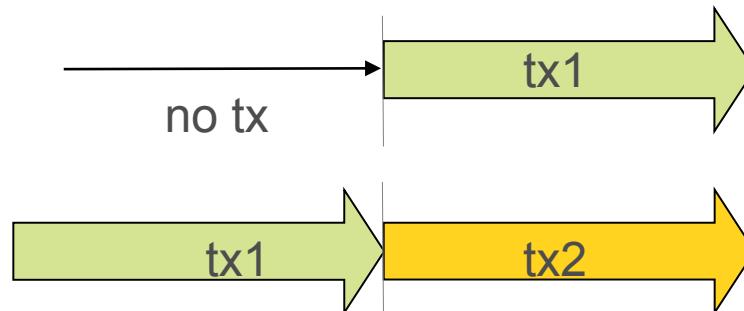
- REQUIRED
  - Default value
  - Execute within a current transaction, create a new one if none exists



```
@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
- 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/or *noRollbackFor* attributes

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

# Topics in this session

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- 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() {
```

```
    ...
```

```
}
```

```
}
```

This test is now  
transactional

# Controlling Transactional Tests

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

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

```
@Transactional
```

```
public class RewardNetworkTest {
```

Make *all* tests  
transactional

```
@Test
```

```
@Commit
```

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

```
    ... // Whatever happens here will be committed
```

```
}
```

```
}
```

Commit transaction  
at end of test

# Lab

Managing Transactions Declaratively  
using Spring Annotations

**Coming Up:** Programmatic transactions, read-only and multiple  
transactions, Global transactions, Propagation options

# Topics in this session

- Advanced topics
  - (1) Programmatic transactions
  - (2) Read-only transactions
  - (3) More on Transactional Tests
  - (4) Multiple and Global Transactions
  - (5) Propagation Options

# 1. 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) {
```

Method *not*  
@Transactional

```
    ...  
    return new TransactionTemplate(txManager).execute( (status) -> {  
        try {
```

Lambda syntax

```
            ...  
            accountRepository.updateBeneficiaries(account);  
            confirmation = rewardRepository.confirmReward(contribution, dining);  
        }
```

Method no longer throws  
exception, using status to  
perform *manual* rollback

```
        catch (RewardException e) {  
            status.setRollbackOnly(); ←  
            confirmation = new RewardFailure();  
        }
```

```
    }  
    return confirmation;
```

```
};  
};
```

```
public interface TransactionCallback<T> {  
    public T doInTransaction(TransactionStatus status)  
        throws Exception;  
}
```

## 2. Read-only Transactions – Faster

- 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(...);  
}
```

Two connections

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

One single connection

# Read-only Transactions – Isolation

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

```
@Transactional(readOnly=true, isolation=Isolation.REPEATABLE_READ)
public void myAccounts(long userId) {
    List accounts = jdbcTemplate.queryForList
        ("SELECT * FROM Accounts WHERE user = ?", userId);
    process(accounts);
    int nAccounts = jdbcTemplate.queryForInt
        ("SELECT count(*) FROM Accounts WHERE user = ?", userId);
    assert accounts.size() == nAccounts;
}
```

### 3. Transactional Tests @Before vs @BeforeTransaction

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

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

```
public class RewardNetworkTest {
```

```
    @BeforeTransaction
```

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

Run *before*  
transaction is started

```
    @Before
```

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

Run *within* the  
transaction

```
    @Test @Transactional
```

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

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

# @Sql and Transaction Control

- Transaction control options
  - **ISOLATED**: Uses own txn, a PTM *must* exist
  - **INFERRRED**: If PTM exists, txn started using default propagation (same txn as test method)  
otherwise a DataSource *must* exist (used with *no* txn)
  - **DEFAULT**: Whatever @Sql defines at class level, *INFERRRED* otherwise

```
@Sql( scripts = "/test-user-data.sql",
       config = @SqlConfig
       ( transactionMode = TransactionMode.ISOLATED,
         transactionManager = "myTxnMgr",
         dataSource= "myDataSource" )
```

Optionally specify  
bean ids

*PTM = PlatformTransactionManager, txn = transaction*



# 4. Multiple Transaction Managers

- Configuration – mark *one* as primary

Java Config

```
@Bean  
public PlatformTransactionManager myOtherTransactionManager() {  
    return new DataSourceTransactionManager(dataSource1());  
}  
  
@Bean  
@Primary  
public PlatformTransactionManager transactionManager() {  
    return new DataSourceTransactionManager(dataSource2());  
}
```

XML

```
<bean id="transactionManager" primary="true" ... > ... </bean>
```

# @Transactional with Multiple 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(...);
}
```

Defaults to use the bean  
annotated as the *primary*

**Important:** Separate transaction managers = separate transactions!

# Transaction Manager Naming

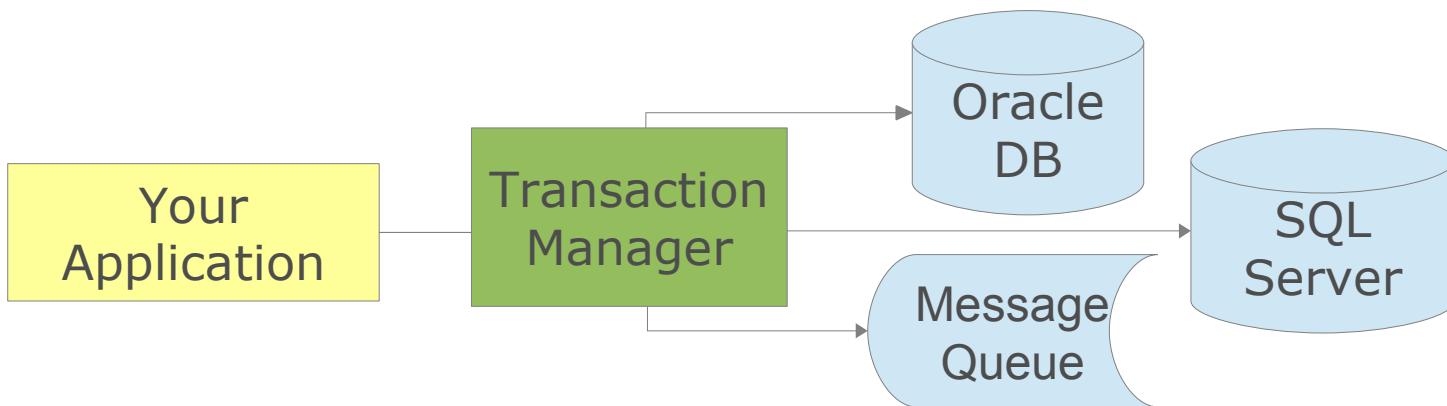
- **@EnableTransactionManagement**
  - Expects a bean called `txManager`
  - Or looks for `PlatformTransactionManager` by type
- **Spring Boot**
  - Creates a bean called `transactionManager` by default
- **@Transactional**
  - Looks for *primary* transaction manager if exists
  - Or looks for singleton `PlatformTransactionManager`
  - Or bean called `transactionManager` by default



*Recall:* bean id “`transactionManager`” is recommended name and `@EnableTransactionManagement` will find it by type.

# Global Transactions

- Also called *distributed* transactions
- Involve multiple dissimilar resources:



- Global transactions typically require JTA and specific drivers (XA drivers)
  - Two-phase commit protocol

# 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>

Learn More: **Enterprise Spring**  
– 4 day course, including *global transactions*

# 5. Propagation Levels and their Behaviors

Propagation Type	If NO current transaction (txn) exists	If there IS a current transaction (txn)
<b>MANDATORY</b>	Throw exception	Use current txn
<b>NEVER</b>	Don't create a txn, run method without a txn	Throw exception
<b>NOT_SUPPORTED</b>	Don't create a txn, run method without a txn	Suspend current txn, run method without a txn
<b>SUPPORTS</b>	Don't create a txn, run method without a txn	Use current txn
<b>REQUIRED</b> (default)	Create a new txn	Use current txn
<b>REQUIRES_NEW</b>	Create a new txn	Suspend current txn, create a new independent txn
<b>NESTED</b>	Create a new txn	Create a new nested txn

# JPA with Spring

Object Relational Mapping with  
Spring & Java Persistence API

Using JPA with Spring, Writing Repositories

# Objectives

- After completing this lesson, you should be able to:
  - Explain the basic concepts of JPA
  - Configure JPA in Spring
  - Implement a JPA DAOs
  - Explain how JPA Integration is implemented by Spring



# Topics in this session

- **Introduction to JPA**
  - **General Concepts, Mapping, Querying**
- Configuring JPA in Spring
- Implementing JPA DAOs
- Lab
- Optional and Advanced Topics

# Introduction to JPA

- The Java Persistence API is designed for operating on domain objects
  - Defined as POJO entities
  - No special interface required
- A common API for object-relational mapping
  - Derived from experience with existing products
    - JBoss Hibernate
    - Oracle TopLink (now EclipseLink)

# About JPA

- Java Persistence API
  - Current version: 2.1 May 2013
  - 2.2 Under discussion
- Configuration
  - Persistence Unit
- Key Features
  - Entity Manager
  - Entity Manager Factory
  - Persistence Context

# JPA Configuration

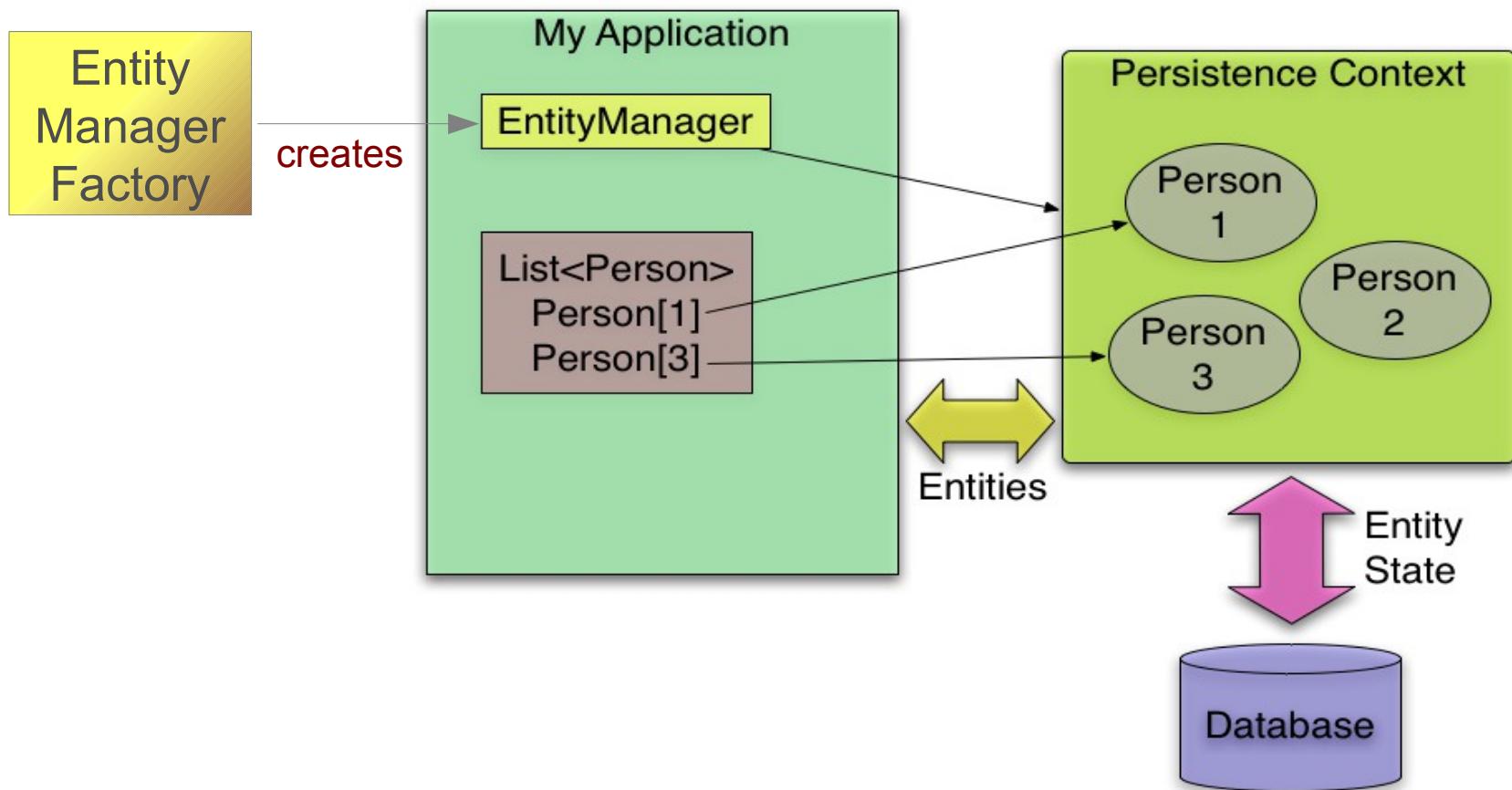
- **Persistence Unit**
  - Describes a group of persistent classes (entities)
  - Defines provider(s)
  - Defines transactional types (local vs JTA)
  - Multiple Units per application are allowed
  - Defined by the file: **persistence.xml**



# JPA General Concepts

- **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

# 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

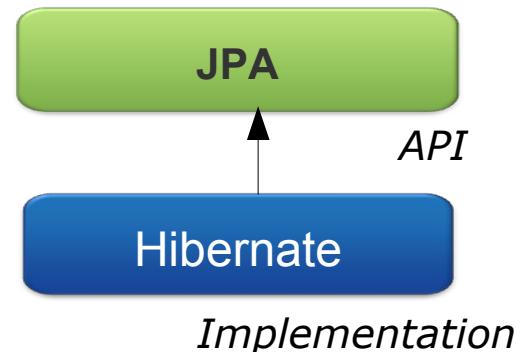
Plus many other methods ...

# 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

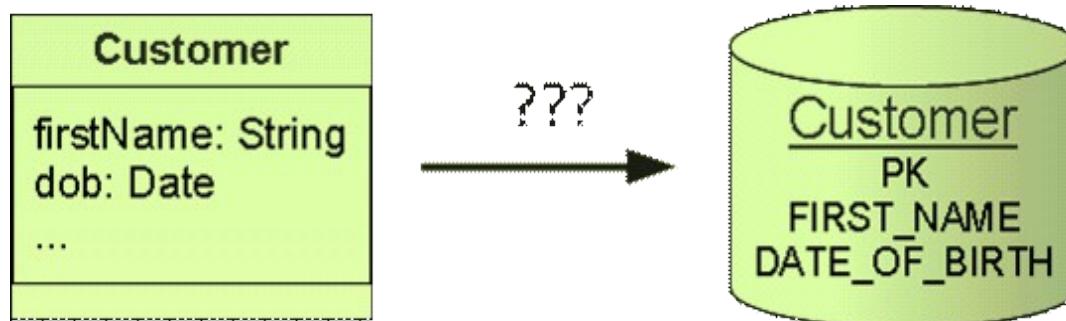
# Hibernate JPA

- 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
  - 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
- **Configuring JPA in Spring**
- Implementing JPA DAOs
- Lab
- Optional and Advanced 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 Spring's  
*packagesToScan* property

# Add Transaction Manager & DataSource

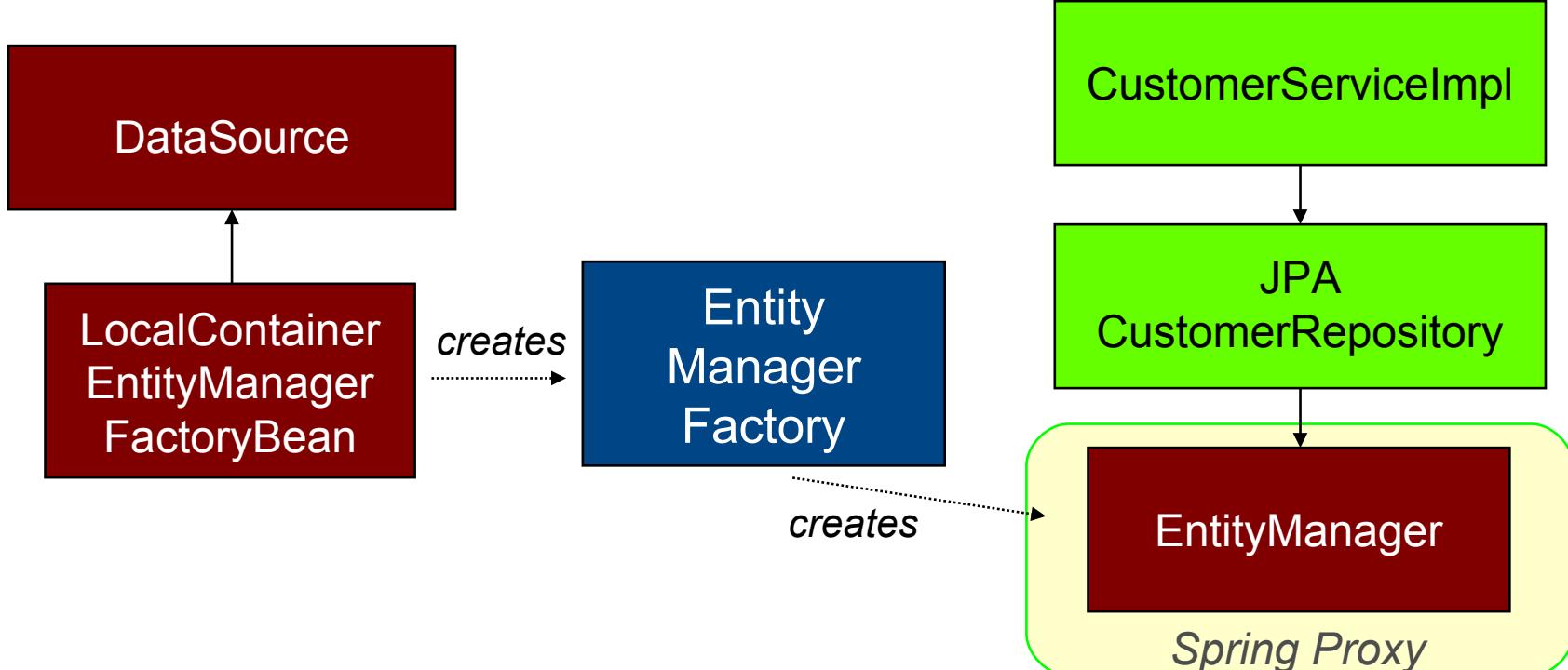
```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    // See previous slide  
    ...  
    return entityManagerFactoryBean; }  
  
@Bean  
public PlatformTransactionManager  
transactionManager(EntityManagerFactory emf) {  
    return new JpaTransactionManager(emf); }  
  
@Bean  
public DataSource dataSource() { /* Lookup via JNDI or create locally */ }
```

Method returns a *FactoryBean*...

... Spring calls `getObject()` on the FactoryBean to obtain the *EntityManagerFactory*:

Or ... use **JtaTransactionManager**

# EntityManagerFactoryBean Configuration



Proxy automatically finds entity-  
manager for current transaction

# Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- **Implementing JPA DAOs**
- Lab
- Optional and Advanced Topics

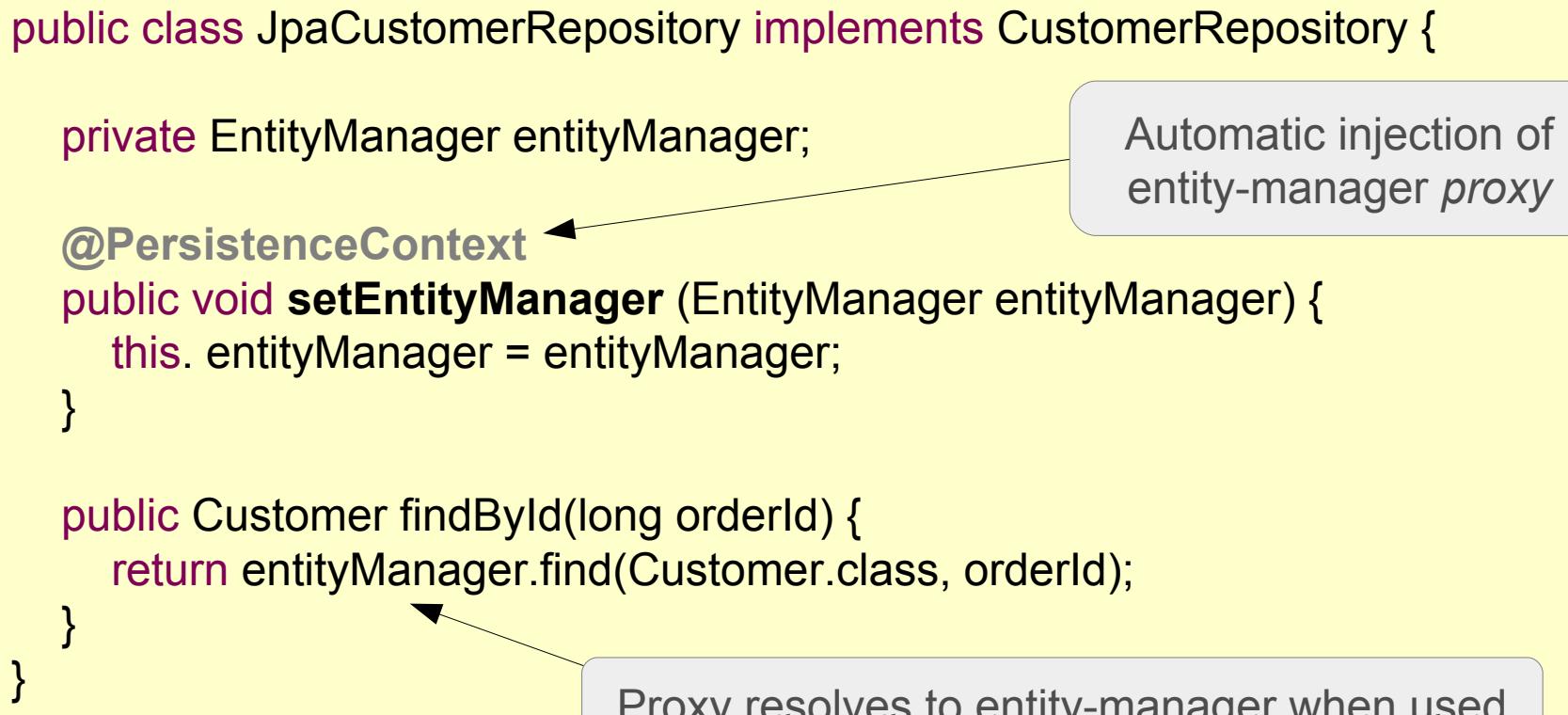
# Implementing JPA DAOs with Spring (1)

- Spring defines *where* transactions occur
  - Delegates to a JPA EntityManager to implement them
  - Supports local or global (JTA) transactions
- Inject *EntityManager* “proxy” via `@PersistenceContext`
  - JPA's equivalent to `@Autowired`
  - At runtime the proxy resolves to current *EntityManager* for current transaction in current thread
- There are *no* Spring dependencies in your Repository (DAO) implementations

# Implementing JPA DAOs with Spring (2)

- The code – no Spring dependencies

```
public class JpaCustomerRepository implements CustomerRepository {  
  
    private EntityManager entityManager;  
  
    @PersistenceContext  
    public void setEntityManager (EntityManager entityManager) {  
        this.entityManager = entityManager;  
    }  
  
    public Customer findById(long orderId) {  
        return entityManager.find(Customer.class, orderId);  
    }  
}
```



Automatic injection of entity-manager proxy

Proxy resolves to entity-manager when used

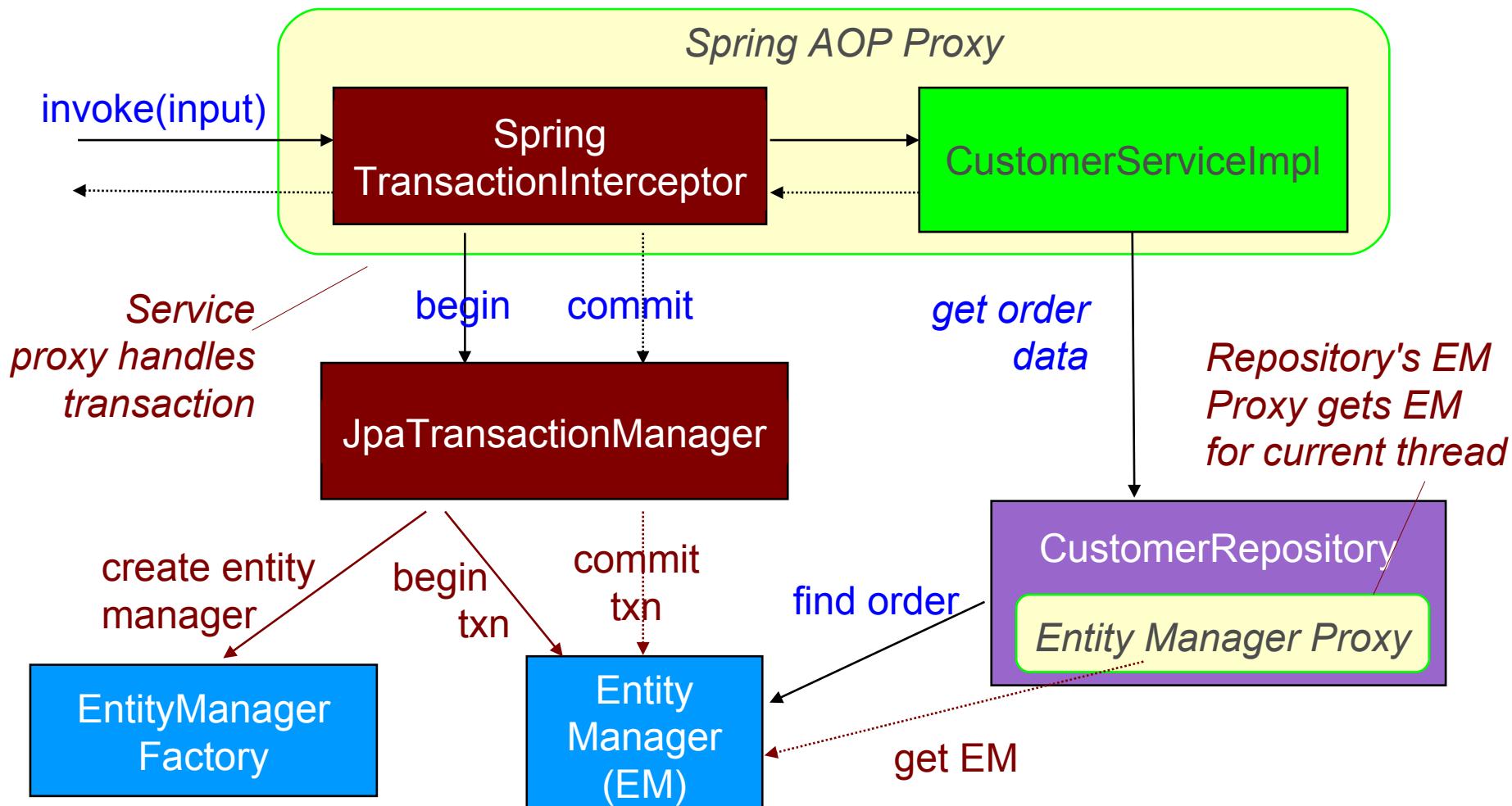
# Implementing JPA DAOs with Spring (3)

- The configuration
  - No need to explicitly call `setEntityManager()`

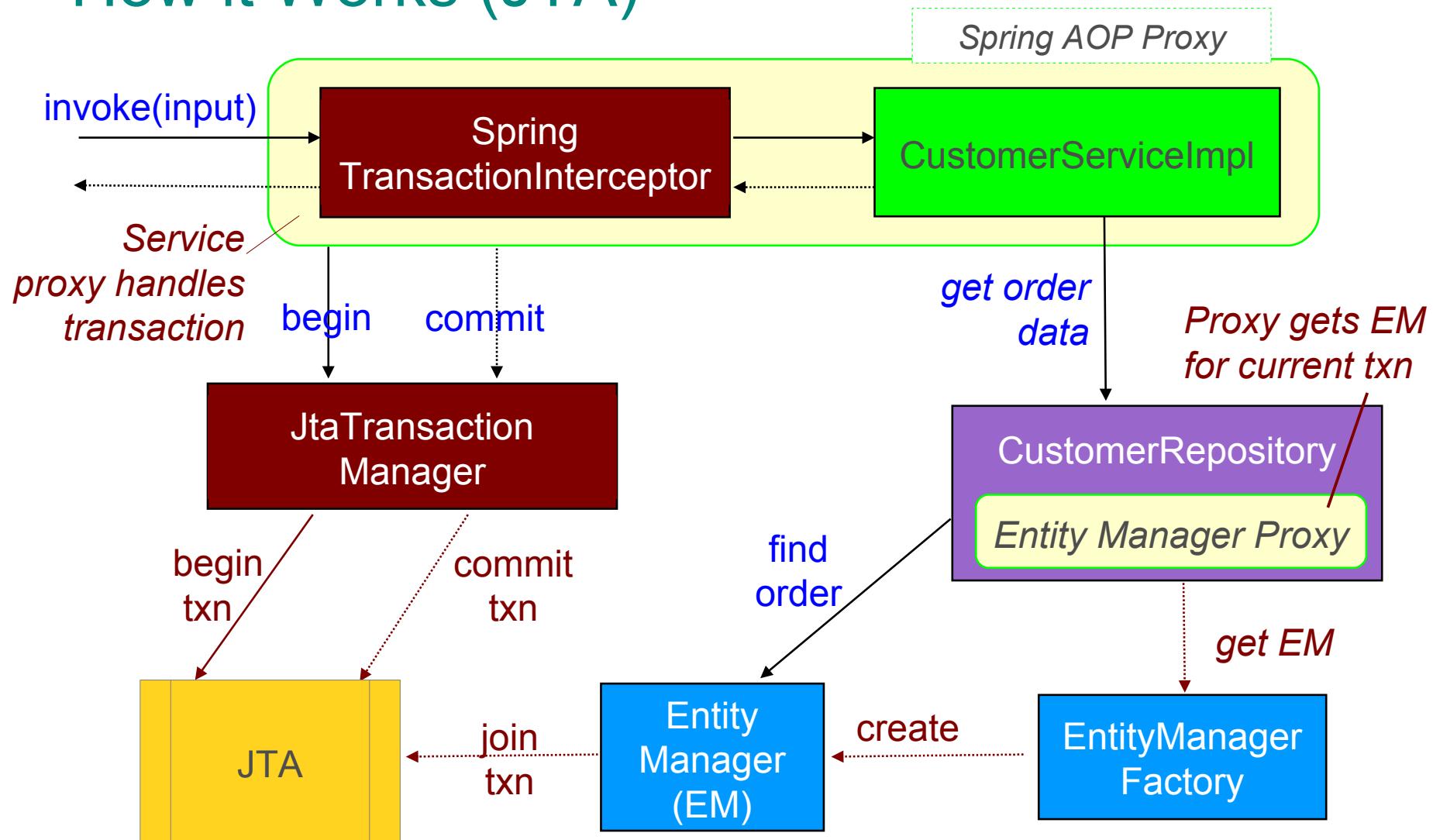
```
@Bean  
public CustomerRepository jpaCustomerRepository() {  
    return new JpaCustomerRepository(); ←  
}
```

Automatic injection of entity-manager proxy

# How it Works (JPA)



# How it Works (JTA)



# 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 (see advanced section)

# Lab

## Reimplementing Repositories using Spring and JPA

**Coming Up:** Optional topics on JPA queries, connection factories  
and `DataAccessExceptions`

# Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Spring Data – JPA
- Lab
- **Optional and Advanced Topics**
  - Exception Translation
  - JPA Typed Queries / Native Queries
  - EntityManagerFactoryBean alternatives / persistence.xml

# Transparent Exception Translation (1)

- Used as-is, the DAO implementations described earlier will throw unchecked JPA 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`
  - Is also an `@Component`, so will be found by component-scanner

```
@Repository  
public class JpaCustomerRepository implements CustomerRepository {  
    ...  
}
```

# Transparent Exception Translation (3)

- Make Spring apply AOP to the annotated repository
  - Define a Spring-provided BeanPostProcessor

```
@Bean  
public PersistenceExceptionTranslationPostProcessor petpp() {  
    return new PersistenceExceptionTranslationPostProcessor();  
}
```

- OR implement **PersistenceExceptionTranslator**
  - Spring creates a **PersistenceException-TranslationPostProcessor** bean automatically
  - **LocalContainerEntityManagerFactoryBean** does this

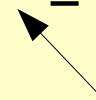
# Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Lab
- **Optional and Advanced Topics**
  - Exception Translation
  - **JPA Typed Queries / Native Queries**
  - EntityManagerFactoryBean alternatives / persistence.xml

# 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();
```

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

```
String customerNumber = (String) query.getSingleResult();
```

```
// Query for multiple columns
```

```
Query query = entityManager.createNativeQuery(  
    "SELECT ... FROM T_CUSTOMER c WHERE ...", Customer.class);  
List<Customer> customers = query.getResultList();
```

No *named* parameter support

Indexed from 1  
– like JDBC

Specify class to  
populate/return

# Topics in this session

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Lab
- **Optional and Advanced Topics**
  - Exception Translation
  - JPA Typed Queries / Native Queries
  - **EntityManagerFactoryBean alternatives**
    - Using `persistence.xml`

# 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>
```

- File required by JPA, optional when using Spring with JPA!

# 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 using  
100% Spring configuration  
In both XML and Java



# Configuration – Spring and Persistence Unit

@Bean

```
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    LocalContainerEntityManagerFactoryBean emfb =  
        new LocalContainerEntityManagerFactoryBean();  
    emfb.setDataSource(dataSource);  
    emfb.setPersistenceUnitName("rewardNetwork");  
    return emfb;  
}
```

Minimal Spring Configuration

Do JPA configuration  
in persistence.xml

```
<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 persistence.xml

# JNDI Lookups

- JNDI lookup used to retrieve an *EntityManagerFactory* from an application server
  - Use when deploying to JEE Application Servers
    - WebSphere, WebLogic, JBoss, Glassfish ...

```
@Bean  
public EntityManagerFactory entityManagerFactory() throws Exception {  
    Context ctx = new InitialContext();  
    return (EntityManagerFactory) ctx.lookup("persistence/rewardNetwork");  
}
```

# Topics Covered

- Introduction to JPA
- Configuring JPA in Spring
- Implementing JPA DAOs
- Optional and Advanced Topics

# Spring Boot - Basics

Getting started with Spring Boot

Starter POMs, Auto-Configuration

# Objectives

- After completing this lesson, you should be able to:
  - Explain what Spring Boot is and why it is Opinionated
  - Describe how Spring Boot is driven by dependencies, properties and auto-configuration



# Topics in this session

- **What is Spring Boot?**
- Spring Boot Explained



# What is Spring Boot?

- Spring Applications typically require a lot of setup
  - Consider working with JPA. We needed:
    - DataSource, TransactionManager, EntityManagerFactory ...
  - Soon we will look at using Spring MVC
    - Needs significant configuration
  - An MVC app using JPA would need all of this
- BUT: Much of this is predictable
  - Spring Boot can do most of this setup for you

# 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>

# Topics in this Session

- **What is Spring Boot?**
  - Definition and Hello World example
- Spring Boot Explained

# Opinionated Runtime?

- Spring Boot uses sensible defaults, “*opinions*”, mostly based on the classpath contents.
- For example
  - Sets up a JPA Entity Manager Factory
    - If a JPA implementation is on the classpath.
  - Creates a `JdbcTemplate`
    - If `spring-jdbc.jar` is on the classpath
- Everything can be overridden easily
  - But most of the time that is not necessary

# Hello World example

- Just three files to get a running Spring application

pom.xml

*Setup Spring Boot (and any other) dependencies*

application.properties

*Database setup*

Application class

*Application launcher*



Maven is just one option. You can also use Gradle or Ant/Ivy

# Hello World (1a) – Maven descriptor

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.M5</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
  </dependency>
</dependencies>
```

Parent POM

Defines dependencies for:  
Spring JDBC, JDBC Connection  
Pool, Spring Boot itself

Embedded SQL  
Database

*pom.xml*

# Hello World (1b) – Maven descriptor

- Will also use the Spring Boot plugin

*pom.xml  
(continued)*

```
<!-- Continued from previous slide -->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Makes “fat” executable jars

# Hello World (2) – Database Properties

- Spring Boot has many, many configuration properties
  - We will see more throughout the course
  - For now ... configure the database

*application.properties*

```
spring.datasource.schema=/testdb/schema.sql  
spring.datasource.data=/testdb/data.sql
```

Automatically configures HSQLDB as an *embedded* in-memory database – and populates by running specified scripts

# Hello World (3) – Application Class

```
@SpringBootApplication
public class Application {
    public static final String query = "SELECT count(*) FROM Account";

    @Autowired private JdbcTemplate jdbcTemplate

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);

        System.out.println("Hello, there are " +
            jdbcTemplate.query(query) + " accounts");
    }
}
```

This annotation turns on Spring Boot

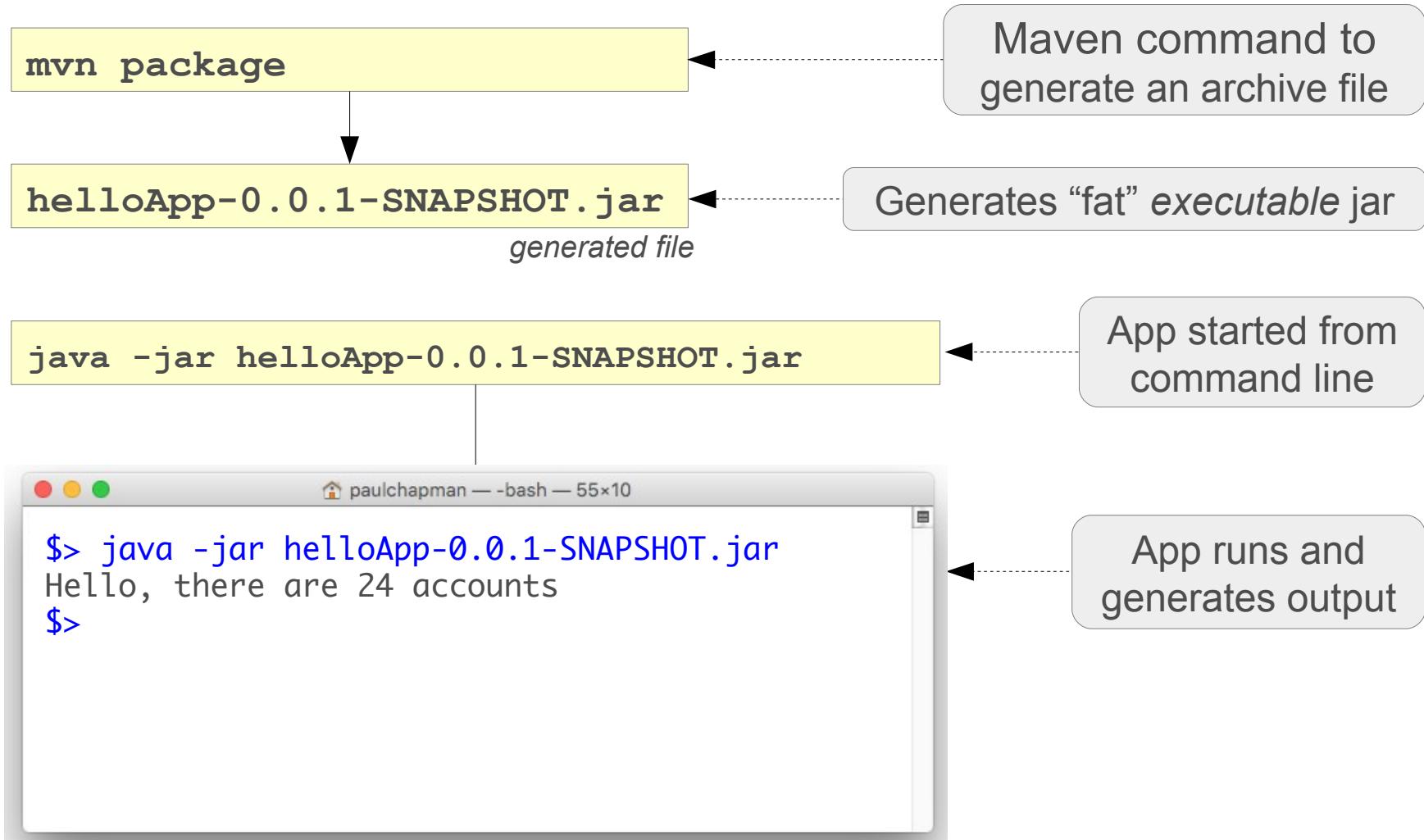
Automatically  
created by Boot and  
injected by Spring

*Application.java*



Main method will be used to run the packaged application from the command line – *old style!*

# Putting it all together



# Topics in this Session

- What is Spring Boot?
- **Spring Boot Explained**
  - Dependency Management
  - Auto Configuration
  - Packaging
  - Testing



# Spring Boot Needs Dependencies

- Auto-configuration works by analyzing the classpath
  - If you forget a dependency, Spring Boot can't configure it
  - A dependency management tool is recommended
  - Spring Boot parent and starters make it much easier
- Spring Boot works with Maven, Gradle, Ant/Ivy
  - Our content here will show Maven



# Spring Boot Parent POM

- Parent POM
  - Defines key versions of dependencies and Maven plugins
    - Uses a `dependencyManagement` section internally

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
</parent>
```

Defines properties for dependencies, for example:  
 `${spring.version} = 5.0.0.RELEASE`

# Spring Boot “Starter” Dependencies

- Easy way to bring in multiple coordinated dependencies
  - Including “*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!

<i>spring-boot-*jar</i>	<i>spring-core-*jar</i>
<i>spring-context-*jar</i>	<i>spring-aop-*jar</i>
<i>spring-beans-*jar</i>	<i>aopalliance-*jar</i>
...	

# Test “Starter” Dependencies

- Common test libraries

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
  </dependency>
</dependencies>
```

**Resolves**  
*spring-test-\*jar*  
*junit-\*jar*  
*mockito-\*jar*  
...

# Available Starter POMs

- Not essential but *strongly* recommended
- Coordinated dependencies for common Java enterprise frameworks
  - Pick the starters you need in your project
- To name a few:
  - **spring-boot-starter-jdbc**
  - **spring-boot-starter-data-jpa**
  - **spring-boot-starter-web**
  - **spring-boot-starter-batch**



See: **Spring Boot Reference, Starter POMs**

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter>

# Creating a New Spring Boot Project

SPRING INITIALIZR

- Use **Spring Initializr** at <https://start.spring.io>
  - *Online Form*: specify project meta-data, packaging, maven/gradle and your starters (dependencies)
  - Create and download project zip
  - Unzip and import into your IDE
- STS: *New Spring Starter Project Wizard*
  - Creates new project directly into STS
    - Uses Spring Initializr to do so



No need to do this for the lab, but you will find it useful when starting your own applications. See *appendix in lab-instructions*.

# Spring Initializr Web Page

SPRING INITIALIZR bootstrap your application now

Generate a  with Spring Boot

## Project Metadata

Artifact coordinates

### Group

com.example

### Artifact

demo

## Dependencies

Add Spring Boot Starters and dependencies to your application

### Search for dependencies

Web, Security, JPA, Actuator, Devtools...

### Selected Dependencies

Web X

Rest Repositories X

JPA X

Specify  
dependencies

Generate Project \*

Don't know what to look for? Want more options? [Switch to the full version.](#)

Switch to full version: more options, explicit check-list of dependencies

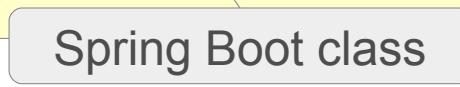
# Topics in this Session

- What is Spring Boot?
- **Spring Boot Explained**
  - Dependency Management
  - **Auto Configuration**
  - Packaging
  - Testing

# Spring Boot @EnableAutoConfiguration

- *@EnableAutoConfiguration* annotation on a Spring Java configuration class
  - Causes Spring Boot to automatically create beans it thinks you need
  - Usually based on classpath contents, can easily override

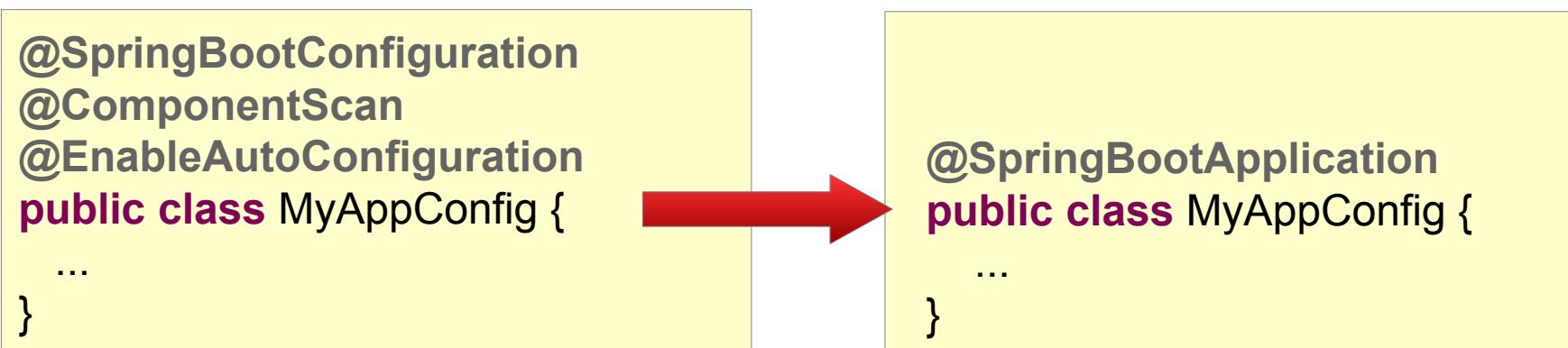
```
@EnableAutoConfiguration
public class AppConfig {
    public static void main(String[] args) {
        SpringApplication.run(MyAppConfig.class, args);
    }
}
```



Spring Boot class

# Shortcut @SpringBootApplication

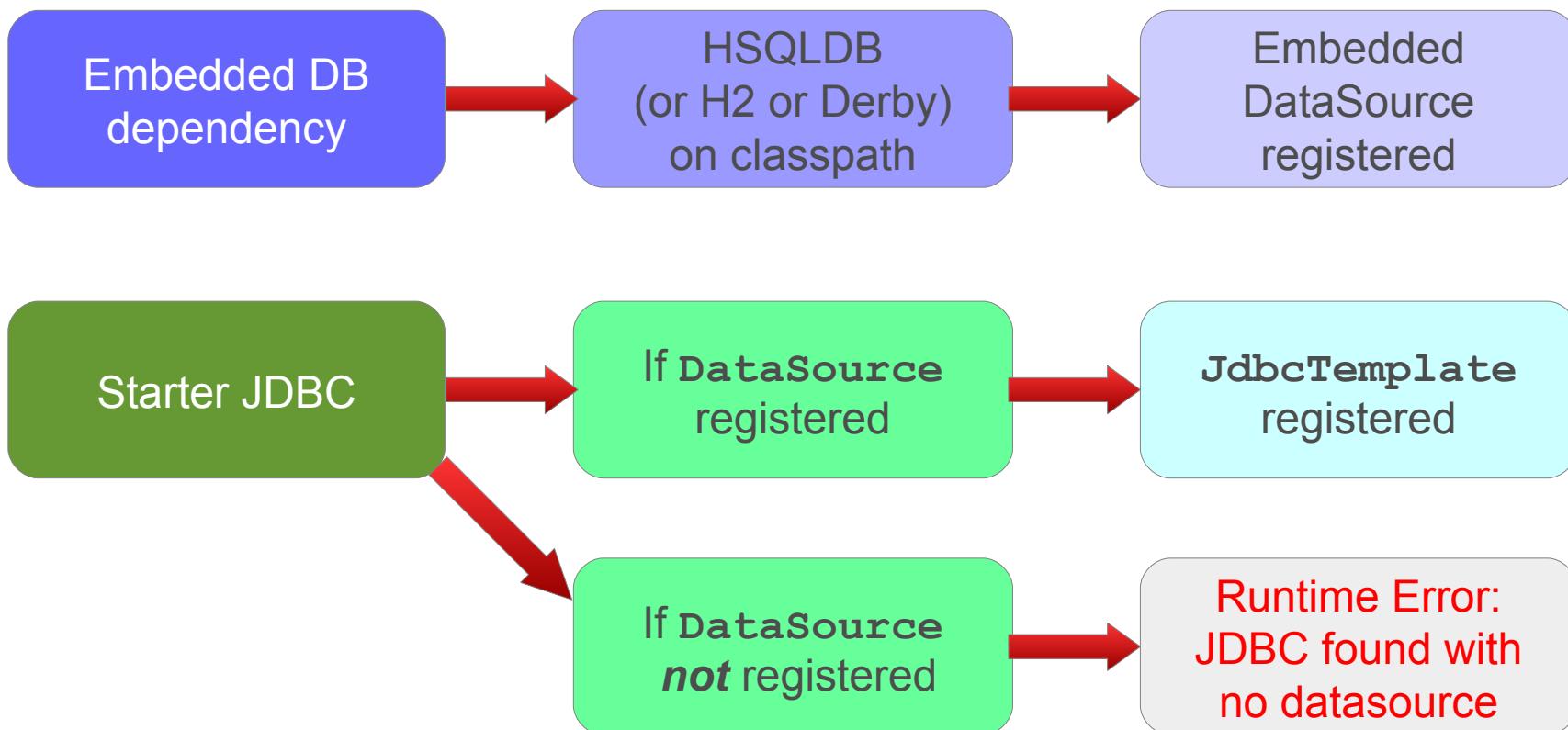
- Very common to use `@EnableAutoConfiguration`, `@Configuration`, and `@ComponentScan` together
  - `@ComponentScan`, with no arguments, scans the current package *and* its sub-packages



`@SpringBootConfiguration` simply extends `@Configuration`

# Auto-configuration: Examples

What you include



# Configuration Properties

## Using *application.properties*

- Developers commonly externalize properties to files
  - Easily consumable via Spring Property-Source
  - But developers name / locate their files different ways
- Spring Boot looks for **application.properties**
  - Many properties exist to control auto-configuration
  - Can put *any* properties you need in here
    - Boot will automatically find and load them
  - Available to **Environment** and **@Value** in usual way



See Appendix A of Spring Boot documentation:

<http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

# *Example: External Database*

- Configuring an *external* database
  - Such as MySQL
  - Make sure project defines JDBC driver dependency

```
spring.datasource.url=jdbc:mysql://localhost/test  
spring.datasource.username=dbuser  
spring.datasource.password=dbpass  
spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
  
spring.datasource.schema=/testdb/schema.sql  
spring.datasource.data=/testdb/data.sql
```

# *Example: Controlling Logging Level*

- Boot can control the logging level
  - Just set it 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
```

*application.properties*



Try to stick to SLF4J in the application.  
The *advanced* section covers how to change the logging framework

# Topics in this Session

- What is Spring Boot?
- **Spring Boot Explained**
  - Dependency Management
  - Auto Configuration
  - **Packaging**
  - Testing

# Packaging

- Spring Boot creates your archive
  - JAR (or WAR)
- Gradle and Maven plugins available
  - Generates an *executable* JAR

```
java -jar yourapp.jar
```



# Maven Packaging

- *Recall:* Spring Boot Maven plugin in 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 (in `target`)

```
22M  yourapp-0.0.1-SNAPSHOT.jar
5K   yourapp-0.0.1-SNAPSHOT.jar.original
```

- `.jar.original` contains only your code (a traditional JAR file)
- `.jar` contains your code *and* all libs – executable
  - *Notice that it is much bigger*

# Topics in this Session

- What is Spring Boot?
- **Spring Boot Explained**
  - Dependency Management
  - Auto Configuration
  - Packaging
  - Testing

# Testing: `@SpringBootTest`

Loads the specified configuration invoking Spring Boot's defaults

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes=TransferConfig.class)
public class TransferServiceTests {
    @Autowired
    private TransferService transferService;

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

```
@Configuration
@EnableAutoConfiguration
@ComponentScan("transfers")
public class TransferApplication {
    // Bean methods
}
```

# Testing: `@SpringBootConfiguration`

- Spring Boot can find `@Configuration` class for itself
  - Provided it is in a package *above* the test
  - Only one `@SpringBootConfiguration` allowed in a hierarchy

```
@ExtendWith(SpringExtension.class)
@SpringBootTest // classes not needed
public class TransferServiceTests {
    // Same tests as previous slide
}
```

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan("transfers")
public class TransferApplication {
    // Bean methods
}
```

≡ `@SpringBootApplication(scanBasePackages="transfers")`



# Summary

- Spring Boot significantly simplifies Spring setup
  - Will setup much of your application for you
  - Uses in-built defaults (opinions) to do the obvious setup
  - Many properties available to customize what it does
  - Use `@SpringBootTest` to enable Spring Boot in tests



See optional Advanced Spring Boot section in Student Handout PDF for more on Spring Boot configuration and usage.

# JPA with Spring Boot & Spring Data

Simplifying JPA setup and implementation

Spring Boot for JPA, Spring Data Repositories

# Objectives

- After completing this lesson, you should be able to:
  - Implement a Spring JPA application using Spring Boot
  - Create Spring Data Repositories for JPA



# Topics in this session

- **Spring JPA using Spring Boot**
- Spring Data – JPA
- Lab
- Advanced Topics

# Spring JPA “Starter” Dependencies

- Everything you need to develop a Spring JPA application

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jpa</artifactId>
  </dependency>
</dependencies>
```

## Resolves

*spring-boot-starter.jar  
spring-boot-starter-jdbc.jar  
spring-boot-starter-aop.jar  
spring-data-jpa.jar  
hibernate-core  
javax.transaction-api*

...

# Spring Boot and JPA

- If `spring-jpa` on classpath, Spring Boot automatically
  - Creates a `DataSource`
  - Creates an `EntityManagerFactoryBean`
  - Sets up a `JpaTransactionManager`
- Can customize
  - `EntityManagerFactoryBean`
  - Transaction manager – use JTA instead

# Recall: EntityManagerFactory Setup

```
@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;  
}
```

Boot now implements this for us  
– so how do we customize it?

# Customize EntityManagerFactoryBean

## Entity Locations

- Where to find entities?
  - By default, Boot looks in same package as class annotated with `@EnableAutoConfiguration`
    - And all its sub-packages
  - Override using `@EntityScan`

```
@SpringBootApplication  
@EntityScan("/rewards/internal")  
public class Application {  
    //...  
}
```

```
setPackagesToScan("rewards.internal");
```

# Customize EntityManagerFactoryBean Configuration Properties

- Specifying vendor-provider properties

*application.properties*

```
# Leave blank - Spring Boot will work it out
# Set to default - Hibernate will work it out
spring.jpa.database=default

# Create tables automatically? Default is:
#   Embedded database: create-drop
#   Any other database: none (do nothing)
# Options: validate | update | create | create-drop
spring.jpa.hibernate.ddl-auto=update

# Show SQL being run (nicely formatted)
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format-sql=true

# Any hibernate property 'xxx'
spring.jpa.properties.hibernate.xxx=???
```

# Spring Boot and JTA

- To use JTA instead
  - Set property: `spring.jta.enabled=true`
- Spring Boot supports 3 standalone JTA implementations
  - Atomikos, Bitronix and Narayana
    - Many specific properties to configure each one
  - Also works with a JEE container provided JTA



For more information, refer to

<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-jta.html>

# JPA Configuration without Spring Boot

```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    ...  
    return entityManagerFactory;  
}  
  
@Bean  
public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {  
    return new JpaTransactionManager(emf);  
}  
  
@Bean  
public DataSource dataSource() { /* Lookup via JNDI or create locally */ }
```

# Replaced By ..

- One annotation

*Application.java*

```
@SpringBootApplication  
@EntityScan("/rewards/internal")  
public class Application {  
    //...  
}
```

- Some properties

*application.properties*

```
# Show SQL being run (nicely formatted)  
spring.jpa.show-sql=true  
spring.jpa.properties.hibernate.format-sql=true
```

- And lots of defaults

# Topics in this Session

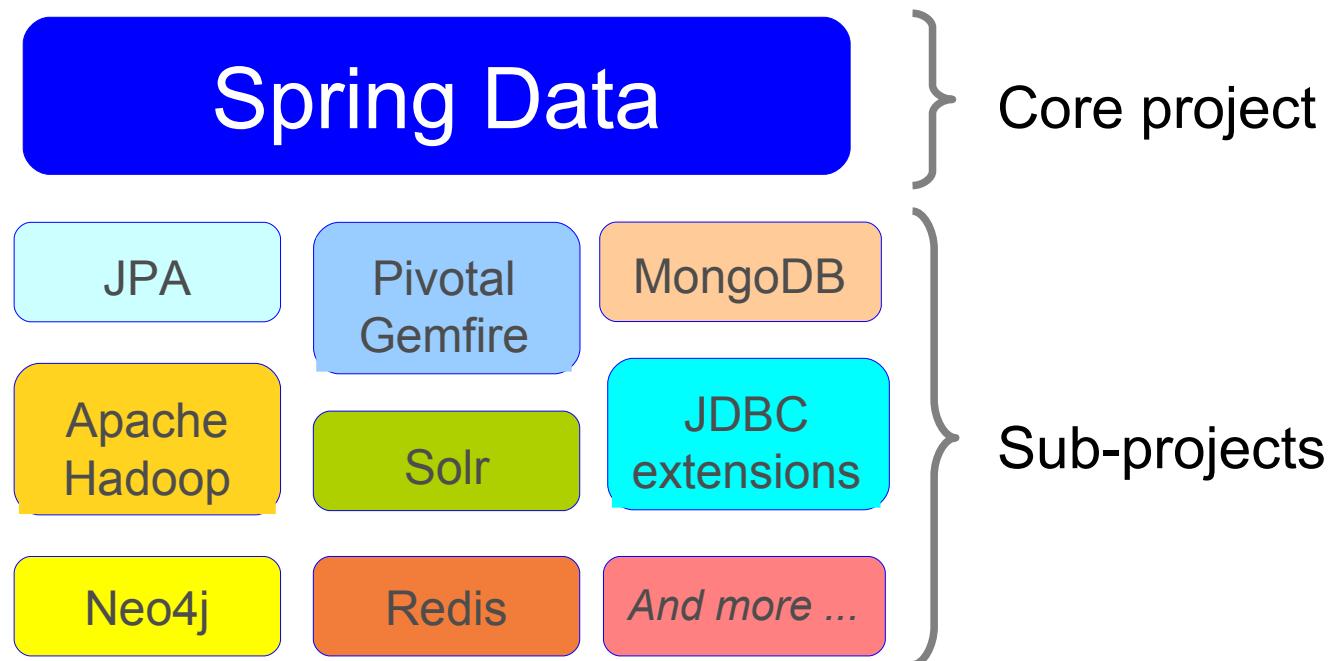
- Spring JPA using Spring Boot
- **Spring Data – JPA**
- Lab
- Advanced Topics



# What is Spring Data?

SPRING DATA

- Reduces boiler plate code for data access
  - Works in many environments





# Instant Repositories

- How?
  - Step 1: Annotate domain class
    - define keys & enable persistence
  - Step 2: 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



# Step 1: Annotate Domain Class

Here we are 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 orderDate;  
    private String email;  
  
    // Other data-members and getters and setters omitted  
}
```

Domain Class

Note: 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 {  
...}
```

# Step 2: Define a Repository Interface

Must extend `Repository<T, ID>`

```
public interface Repository<T, ID> { }
```

Marker interface – add any methods from `CrudRepository` and/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 them as Spring bean's

```
@Configuration  
@EnableJpaRepositories(basePackages="com.acme.*.repository")  
@EnableMongoRepositories(...)  
public class MyConfig { ... }
```

# Defining a JPA Repository

- Auto-generated finders obey naming convention
  - `find(First)By<DataMember><Op>`
  - `<Op>` can be `GreaterThan`, `NotEquals`, `Between`, `Like ...`

```
public interface CustomerRepository
    extends CrudRepository<Customer, Long> {

    public Customer findFirstByEmail(String someEmail);      // No <Op> for Equals
    public List<Customer> findByOrderDateLessThan(Date someDate);
    public List<Customer> findByOrderDateBetween(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 findOne(long i); // Definition as per CrudRepository  
  
    Customer findFirstByEmailIgnoreCase(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 – Another Spring Proxy

- Before startup

Interface  
CustomerRepository

- After startup

Interface  
CustomerRepository

*implements*

\$Proxy1

```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.repository")
public class CustomerConfig { ... }
```

# Accessing the Repository

- Use Spring to inject *CustomerRepository* dependency

```
@Configuration  
@EnableJpaRepositories(basePackages="com.acme.repository")  
public class CustomerConfig {  
  
    @Autowired  
    CustomerRepository customerRepository;  
  
    @Bean  
    public CustomerService customerService() {  
        return new CustomerService( customerRepository );  
    }  
}
```

# Summary

- Spring Boot significantly simplifies Spring setup
  - Will setup most of JPA for you
- Similarly, Spring Data simplifies Repositories
  - Just define an interface - you need no code!

# Lab

Reimplementing JPA application  
using Spring Boot and Spring Data

**Coming Up:** Optional topic on custom Spring Data repositories

# Topics in this session

- Spring JPA using Spring Boot
- Spring Data – JPA
- Lab
- **Optional and Advanced Topics**
  - Customized Spring Data Repositories

# JPA Specific Interface

- Adds EntityManager specific options

```
public interface JpaRepository<T, ID extends Serializable>
    extends PagingAndSortingRepository<T, ID> {

    <S extends T> S saveAndFlush(S entity);
    void flush();

    // Implemented as a single DELETE
    void deleteInBatch(Iterable<T> entities);
    void deleteAllInBatch();

    // Returns a lazy-loading proxy, using JPA's EntityManager.getReference()
    // – equivalent to Hibernate's Session.load()
    T getOne(ID id);
}
```

# Adding Custom Behavior (1)

- Not all use cases satisfied by automated methods
  - Enrich with custom repositories: *mix-ins*
- **Step 1:** Create normal interface and implementation

```
public class CustomerRepositoryImpl implements CustomerRepositoryCustom {  
    Customer findDeadbeatCustomers() {  
        // Your custom implementation to find unreliable  
        // and bad-debt customers  
    }  
}  
  
public interface CustomerRepositoryCustom {  
    Customer findDeadbeatCustomers();  
}
```



# Adding Custom Behavior (2)

- **Step 2:** Combine with an automatic repository:

```
public interface CustomerRepository  
    extends CrudRepository<Account, Long>, CustomerRepositoryCustom {  
}
```

- Spring Data looks for implementation beans
  - ID = repository interface + “Impl” (configurable)
  - In this example: “*CustomerRepositoryImpl*”
- **Result:** *CustomerRepository* bean contains automatic and custom methods!

# Using Optional

- Some methods can return null or Optional

```
public interface CustomerRepository extends Repository<Customer, Long> {  
    // CRUD method using object type – returns null if not found  
    Customer findOne(Long id);  
    // Query method using object type – also returns null if not found  
    Customer findFirstByEmail(String someEmail);  
}
```

OR

```
public interface CustomerRepository extends Repository<Customer, Long> {  
    // CRUD method using Optional  
    Optional<Customer> findOne(Long id);  
    // Query method using Optional  
    Optional<Customer> findFirstByEmail(String someEmail);  
}
```

# Topics Covered

- Spring JPA using Spring Boot
- Spring Data – JPA
- Lab
- Optional and Advanced Topics

# Spring Web MVC Essentials

## Getting Started With Spring MVC

### Implementing a Simple Controller

# Objectives

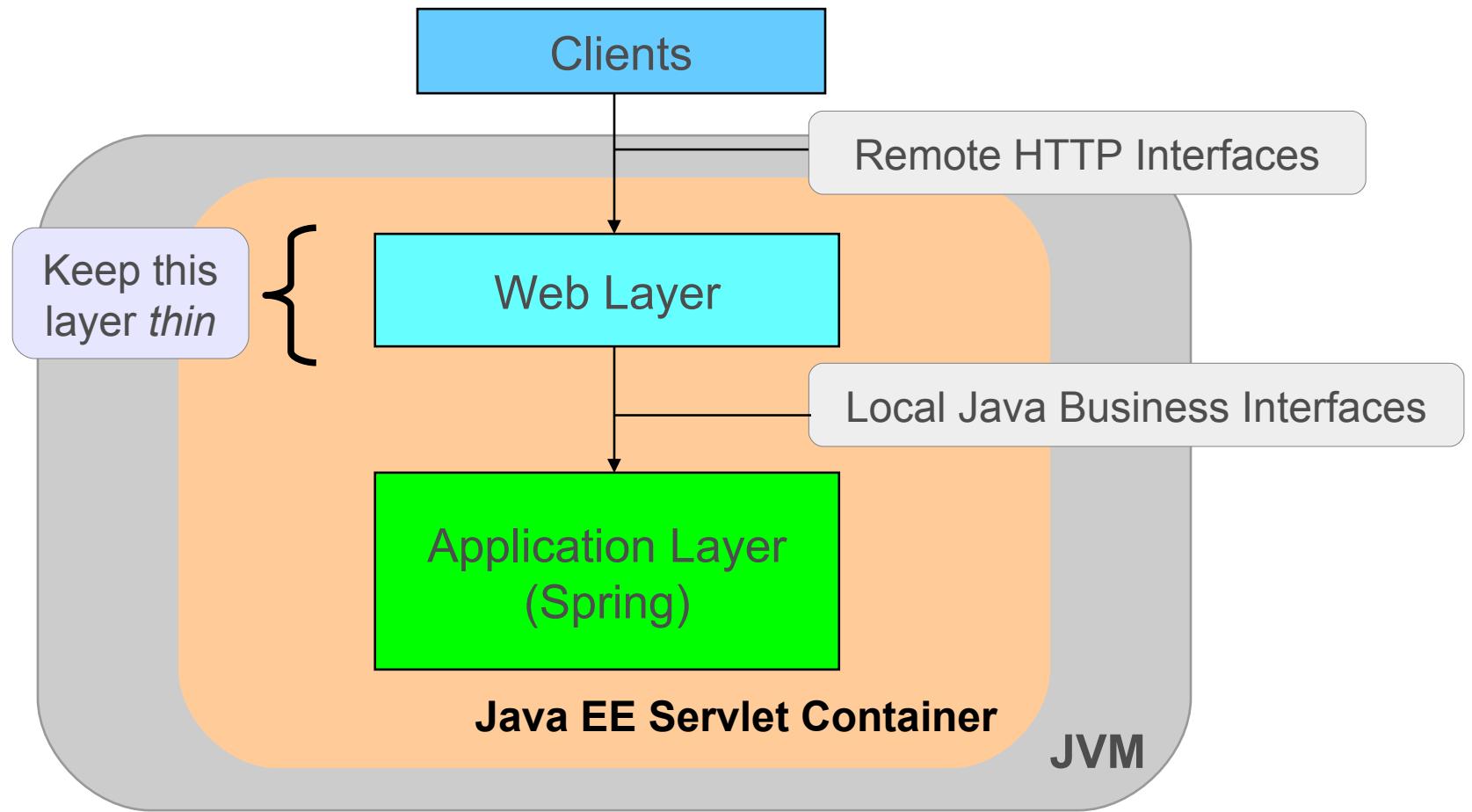
- After completing this lesson, you should be able to:
  - Explain Spring MVC & Request Processing Lifecycle
  - Describe the purpose of the Dispatcher Servlet
  - Implement a Controller
  - Setup a View and a View Resolver
  - Use Spring Boot in a Web Application
  - Explain when to use `@EnableWebMvc`
  - Deploy a Boot-driven WAR



# Web Layer Integration

- Spring provides support in the Web layer
  - Spring MVC, Spring WebFlow, REST, ...
- However, you are free to use Spring with any Java web framework
  - Integration might be provided by Spring or by the other framework itself
  - Spring also integrates with many of the common REST frameworks

# Effective Web Application Architecture



# What is Spring MVC?



- Web framework based on *Model/View/Controller* pattern
  - Alternative to JSF, Struts 1 or 2, Tapestry, Wicket ...
- Based on Spring principles
  - POJO programming
  - Testable components
  - Uses Spring for configuration
- Supports a wide range of server-side rendering
  - JSP, XSLT, PDF, Excel, Velocity, FreeMarker, Thymeleaf, Groovy Markup, Mustache ...
- Basis for Spring REST

# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
- Spring Boot for Web Applications
- Quick Start
- Lab
- `@EnableWebMvc`
- Spring Boot WARs



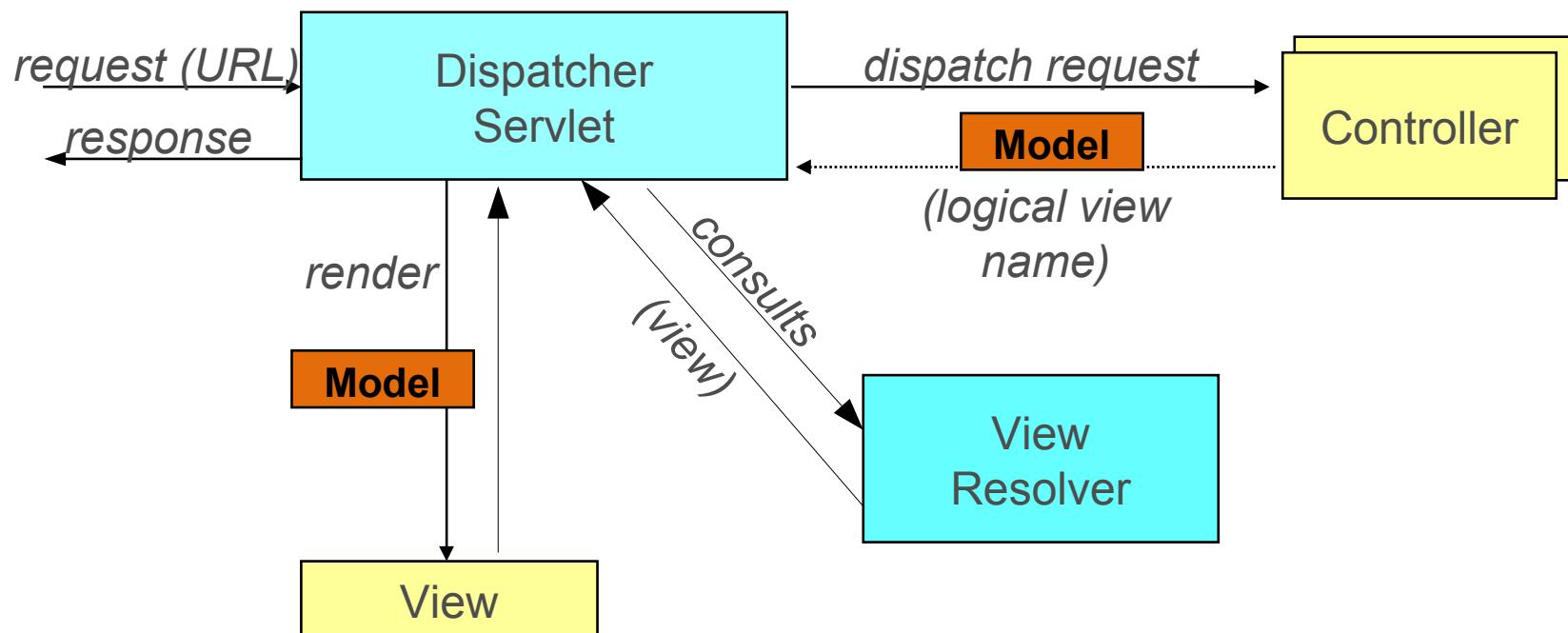
Last two sections may be required for Certification

# 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

*The basis for generating both **HTML and REST** responses*

# Request Processing Lifecycle



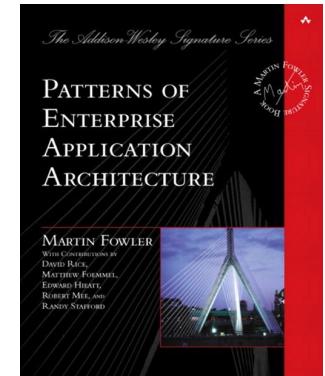
# Topics in this Session

- Request Processing Lifecycle
- **Key Artifacts**
  - **DispatcherServlet**
  - Controllers
  - Views
- Spring Boot for Web Applications
- Quick Start
- Lab
- **@EnableWebMvc**
- Spring Boot WARs

# 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



# Background: Servlet Configuration



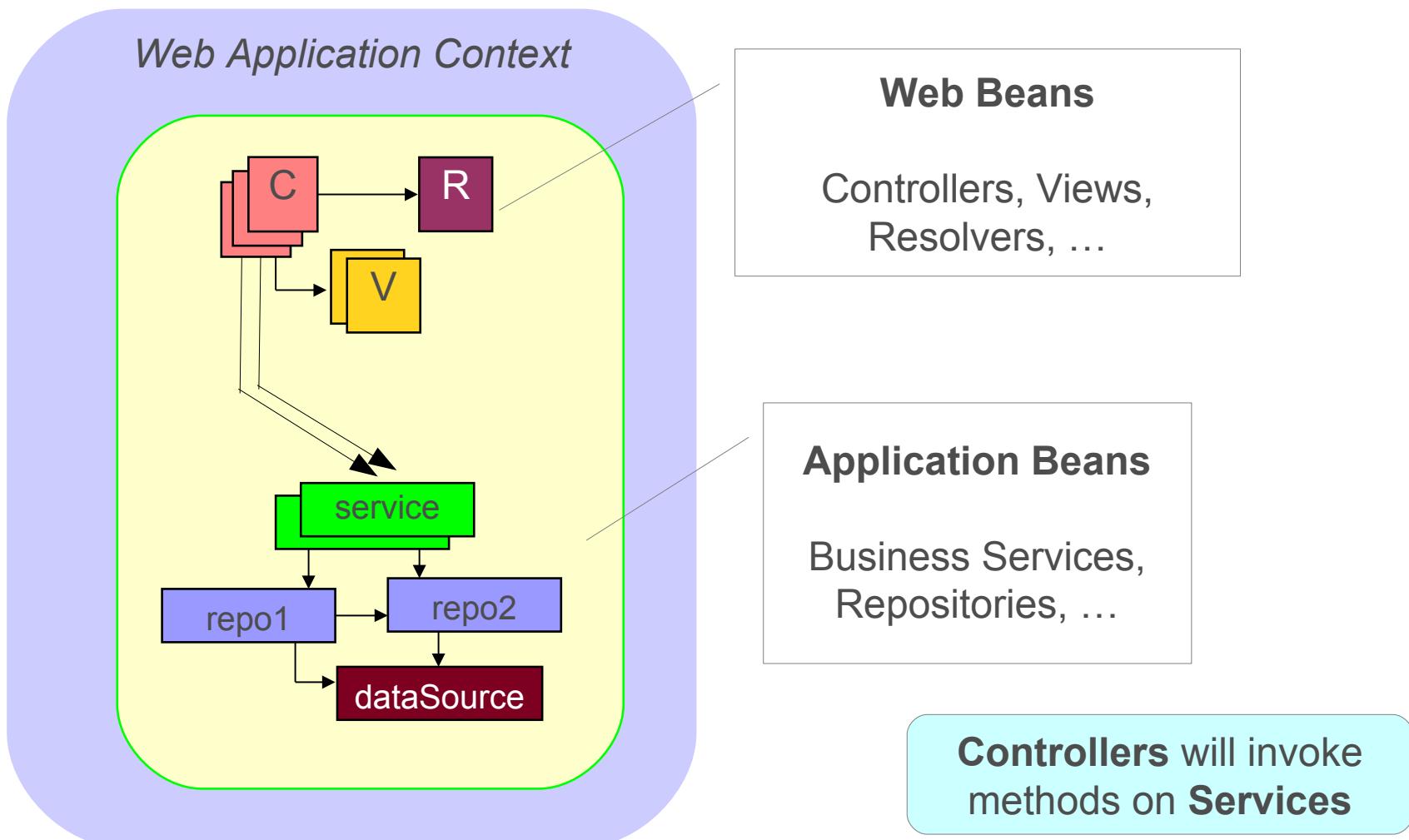
- Servlets can be defined statically
  - Using web.xml (Servlet 2 spec)
  - Using a class that implements ServletContainer-Initializer (Servlet 3+)
- Servlets can be defined dynamically
  - By registering them with the ServletContext as JEE Server starts up (Servlet 3+)
  - *Spring Boot does this*



# DispatcherServlet Configuration

- Automatically created and configured by Spring Boot
  - Given full access to a **WebApplicationContext**
    - Web specific, supports Session, Request scopes
    - Other servlet related artifacts
  - Contains both web-layer beans and application beans
- Web-layer beans also use Spring for their configuration
  - Programming to interfaces + dependency injection
  - Easy to swap parts in and out

# Servlet Container After Starting Up



# Topics in this Session

- Request Processing Lifecycle
- **Key Artifacts**
  - DispatcherServlet
  - **Controllers**
  - Views
- Spring Boot for Web Applications
- Quick Start
- Lab
- **@EnableWebMvc**
- Spring Boot WARs

# Controller Implementation

We are *not* deploying a WAR, so no WAR or servlet context in URLs

- 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 / listAccounts*

application server

request mapping

# URL-Based Mapping Rules

- Mapping rules typically URL-based, optionally using wild cards:
  - /accounts
  - /accounts/edit
  - /editAccount
  - /listAccounts.htm
  - /accounts/\*

Suffixes *ignored*  
by default

# 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) {  
        ... // Do something  
    }  
}
```

*Example of calling URL:*

<http://localhost:8080/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) {  
        ... // Do something  
    }  
}
```

*Example of calling URL: <http://localhost:8080/accounts/123>*

# Method Signature Examples

Example URLs

```
@RequestMapping("/accounts")
public String show(HttpServletRequest request, Model model)
```

*http://localhost:8080/accounts*

```
@RequestMapping("/orders/{id}/items/{itemId}")
public String show(@PathVariable("id") Long id,
                   @PathVariable int itemId,
                   Model model, Locale locale,
                   @RequestHeader("user-agent") String agent )
```

*http://.../orders/1234/items/2*

```
@RequestMapping("/orders")
public String show(@RequestParam Long id,
                   @RequestParam("itemId") int itemId,
                   Principal user, Map<String, Object> model,
                   HttpSession session )
```

*http://.../orders?id=1234&itemId=2*

View name



# Topics in this Session

- Request Processing Lifecycle
- **Key Artifacts**
  - DispatcherServlet
  - Controllers
  - **Views**
- Spring Boot for Web Applications
- Quick Start
- Lab
- **@EnableWebMvc**
- Spring Boot WARs

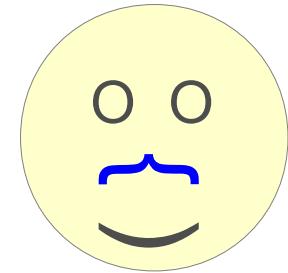
# Views

- 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, ...
- 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
  - By default, a JSP: `/WEB-INF/reward/list.jsp`
- Override this default by registering a ViewResolver bean with the DispatcherServlet
  - Maps “logical” view names to actual views

# View Resolvers

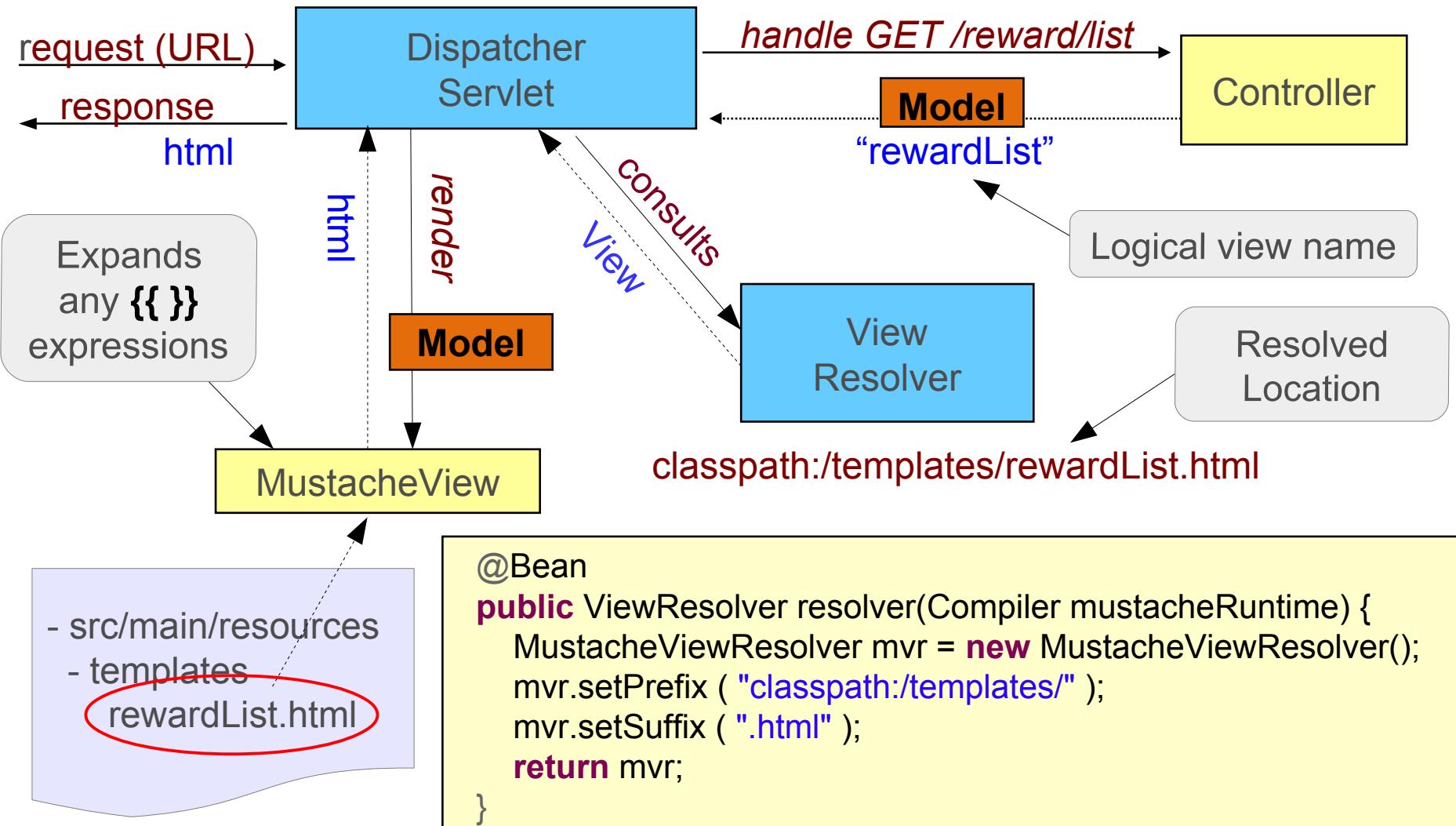


- One for each supported server-side view technology
  - **InternalResourceViewResolver** for JSP/JSTL
  - **ThymeleafViewResolver**, **FreemarkerViewResolver**,  
**GroovyMarkupViewResolver**, **MustacheViewResolver** ...
- For simplicity we will use *Mustache*
  - Embeds dynamic content inside double-braces `{ {var} }`



You won't have to write any Mustache templates in the Lab, we've written them for you.

# View Resolver Example



# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
- **Spring Boot for Web Applications**
- Quick Start
- Lab
- `@EnableWebMvc`
- Spring Boot WARs

# Spring Boot for Web Applications

- Use **spring-boot-starter-web**
  - Ensures Spring Web and Spring MVC is on classpath
  - Sets up a **DispatcherServlet**
  - Sets up internal configuration to support **@Controllers**
  - Sets up default resource locations
  - And much, much more



# Configuring Mustache

- Mustache also needs its compiler & template-loader
  - They process and expand the {{ }} expressions
- Use **spring-boot-starter-mustache**
  - Declare properties and Boot does the rest

**application.properties**

```
spring.mustache.prefix=classpath:/templates/  
spring.mustache.suffix=.html
```

```
@Bean  
public MustacheViewResolver viewResolver() {  
    ...  
    return viewResolver;  
}
```

# Spring Boot as a Runtime

- Spring Boot starts up an embedded web server
  - You can run a web application from a JAR file!
  - Tomcat included by Web Starter

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```



Spring Boot's default behavior. Simpler for running and testing.  
May be easier when deploying *Cloud Native* applications

# Alternative Containers: Jetty, Undertow

- Example: 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>
```

Excludes Tomcat

Adds Jetty

Jetty automatically detected and used!

# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
- Spring Boot for Web Applications
- **Quick Start**
- Lab
- `@EnableWebMvc`
- Spring Boot WARs

# *Revision: What We Have Covered*

- Spring Web applications have *many* features
  - The Dispatcher Servlet
  - Setup Using Spring Boot
  - Writing a Controller
  - Using Views and a View Resolver
- *But you don't need to worry about most of this to write a simple Spring MVC application ...*

# Quick Start

- Only a few files to get a running Spring Web application

pom.xml

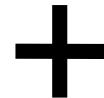
Gradle or Ant/Ivy also supported

*Setup Spring Boot (and any other) dependencies*

RewardController class

*Basic Spring MVC controller*

application.properties



RewardView.html

*View setup*

Application class

*Application launcher*

# 1a. Maven Descriptor

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.M5</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mustache</artifactId>
  </dependency>
</dependencies>
```

Parent POM

Spring MVC, Embedded Tomcat, Jackson ...

Mustache support

pom.xml

# 1b. Maven Descriptor (continued)

- Will also use the Spring Boot plugin

*pom.xml  
(continued)*

```
<!-- Continued from previous slide -->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Makes “fat” executable jars



Maven is just one option. You can also use Gradle or Ant/Ivy

# 1c. Spring Boot sets up Spring MVC

- Spring web jars are on classpath
  - Spring Boot sets up Spring MVC
  - Deploys a Dispatcher Servlet
- We are *not* deploying a WAR file
  - There is no WAR or servlet context in URLs
  - Will implement URLs like ...

`http://localhost:8080/accounts`

`http://localhost:8080/rewards`

`http://localhost:8080/rewards/1`



- We will implement the last example

## 2. Implement the Controller

*RewardController.java*

```
@Controller  
public class RewardController {  
    private RewardLookupService lookupService;  
  
    @Autowired  
    public RewardController(RewardLookupService svc) {  
        this.lookupService = svc;  
    }  
  
    @RequestMapping("/rewards/{id}")  
    public String show(@PathVariable("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 "rewardView" to render the reward

# 3. Setup the ViewResolver

- Will use a *MustacheViewResolver*
  - Just need to configure it

The diagram shows a code snippet in a yellow box representing the `application.properties` file. It contains two properties: `spring.mustache.prefix=classpath:/templates/` and `spring.mvc.view.suffix=.html`. A callout bubble points to the trailing slash in the prefix property with the text "Note trailing /".

```
application.properties
spring.mustache.prefix=classpath:/templates/
spring.mvc.view.suffix=.html
```

Note trailing /

- If Controller returns `rewardView`
  - ViewResolver converts to `classpath:/templates/rewardView.html`
  - Spring Boot V2 defaults are: `classpath:/templates` and `.mustache`

## 4. Implement the View

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

Block references  
model object (reward)

Reference reward  
properties by name

*/src/main/resources/templates/rewardView.html*

**Note:** No references to Spring in Mustache template

# 5. Application Class

- `@SpringBootApplication` annotation enables Spring Boot
  - Runs Tomcat *embedded*

```
@SpringBootApplication
public class Application {

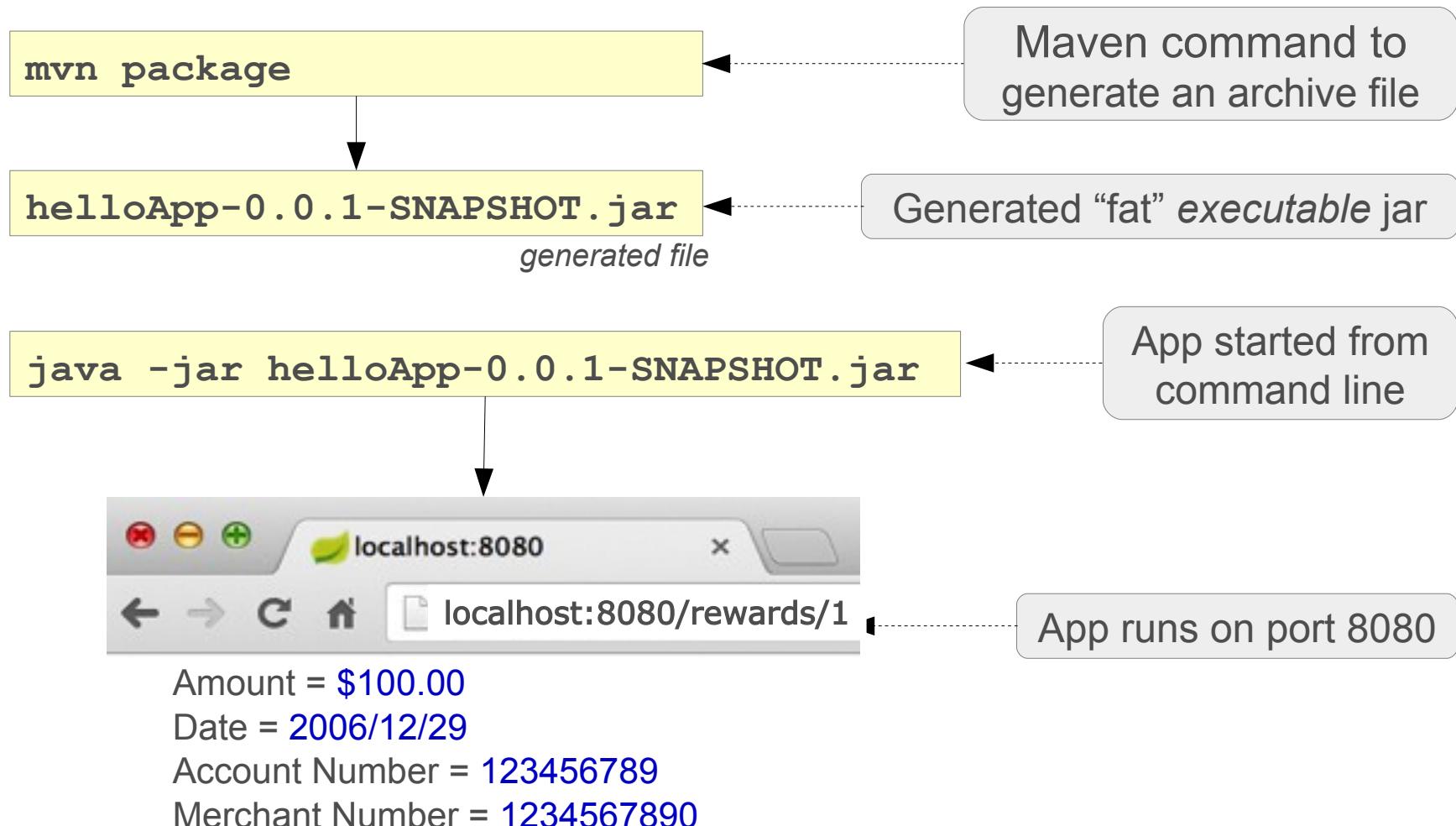
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

*application.java*



Component scanner runs automatically, will find `RewardsController`

# 6. Deploy and Test



# Summary

- Spring MVC is Spring's web framework
  - `@Controller` classes handle HTTP requests
  - URL information available
    - `@RequestParam`, `@PathVariable`
  - Data returned via the *Model*
  - Output (HTML) generated by Views
- Multiple view technologies supported
  - `ViewResolvers` define where Views can be found

**Learn More:** *Spring-Web 4 day course*  
Using Spring for REST and Web applications

# Lab

## Adding a Web Interface

Coming Up: `@EnableWebMvc` vs Spring Boot, Using WARs

# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
- Spring Boot for Web Applications
- Quick Start
- Lab
- **@EnableWebMvc**
- Spring Boot for WARs

Next two sections  
*may be*  
required for  
**Certification**

# A Note on `@EnableWebMvc`

- *Without* Spring Boot
  - Use `@EnableWebMvc` to get many of the same defaults
    - `@Controllers` and Views work “out-of-the-box”
    - REST Support, JSR-303 Validation
    - Stateless converters for form-handling, ...
- With Spring Boot
  - Boot assumes *you* want to control setup
  - Gives you *just* the configuration of `@EnableWebMvc`
    - No extras (next slide)

**Bottom Line:** Most Spring Boot web applications *do not* specify `@EnableWebMvc`

# Spring Boot Convenience



- Boot automatically configures
  - A `DispatcherServlet`
  - Spring MVC using same defaults as `@EnableWebMvc`
- *Plus many useful extra features:*
  - Static resources served from classpath
    - `/static`, `/public`, `/resources` or `/META-INF/resources`
  - Templates served from `/templates`
    - If Groovy, Freemarker, Thymeleaf, Mustache ... on classpath
  - Provides default `/error` mapping
    - Can be overridden
  - Default `MessageSource` for I18N

# Topics in this Session

- Request Processing Lifecycle
- Key Artifacts
- Spring Boot for Web Applications
- Quick Start
- Lab
- `@EnableWebMvc`
- **Spring Boot for WARs**

# JAR vs WAR?

- Why Run a Web Application with Embedded Container?
  - No separation of container config and app config
    - Depend on each other anyway (JNDI DS names, security config ...)
  - Projects usually know which container will be used
    - Why not just include it?
  - No special IDE support needed
    - Easier debugging and profiling, hot code replacement
  - Familiar model for non-Java developers
  - Recommended for Cloud Native applications
    - 12-Factor applications #7 (see <http://12factor.net>)



# Spring Boot in a Servlet Container

- Spring Boot can also run in any Servlet 3.x container
  - Such as: Tomcat 7+, Jetty 8+
- Only small changes required
  - Change artifact type to WAR (instead of JAR)
  - Extend `SpringBootServletInitializer`
  - Override configure method
- Still no `web.xml` required

# Spring Boot in a Servlet Container

Sub-classes Spring's *WebApplicationInitializer*  
– called by the web container (Servlet 3.0)

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

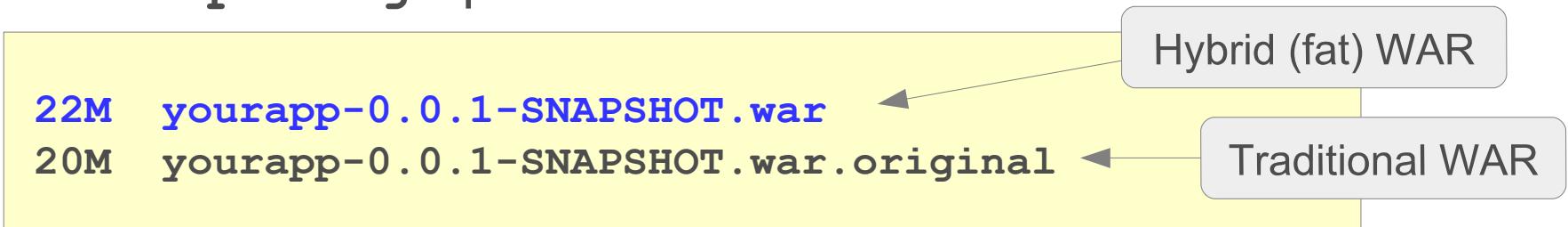
}
```



The above requires no *web.xml* file

# Spring Boot WAR file

- `mvn package` produces *two* WAR files



- Hybrid WAR executable with embedded Tomcat
  - using “`java -jar yourapp.war`”
- Traditional WAR file is produced as well
  - without embedded Tomcat
  - just drop it in your application server web app directory

# Servlet Container and Containerless

- Can execute: `java -jar yourapp-0.0.1-SNAPSHOT.war`

```
@SpringBootApplication
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 support

Provides main method too

**Warning:** Mark `tomcat-starter-web` as “provided” (ignored by 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, especially for the **DispatcherServlet**
    - Sensible defaults based on the classpath content
  - Embedded container can be used during development
  - Your production environment already setup for a WAR
    - Ops want to retain configuration control

# Topics Covered

- Request Processing Lifecycle
- Key Artifacts
  - Dispatcher Servlet, Controllers, Views
- Spring Boot for Web Applications
- Quick Start
- @EnableWebMvc
- Spring Boot for WARs



For more on Spring MVC without Spring Boot, see optional section at back of *Student Handout PDF*.

*Optional Section*

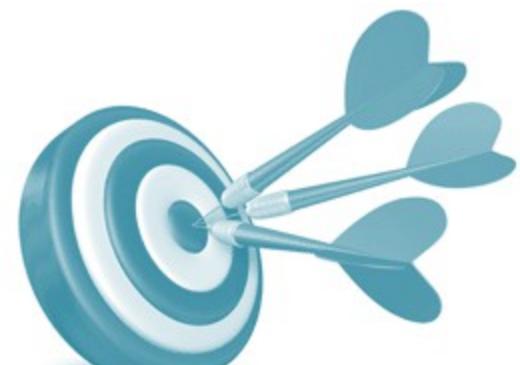
# Spring Boot – Going Further

Going beyond default settings

Customization, Logging, YAML Properties, Testing

# Objectives

- After completing this lesson, you should be able to:
  - Explain how Auto-Configuration drives Spring Boot
  - Use various Techniques to Customize Spring Boot
  - Implement Configuration Properties
  - Fine-tune Logging
  - Use YAML for Configuration
  - Test Spring Boot applications



# Topics in this Session

- **Understanding Auto-Configuration**
- Customizing Spring Boot
- More on Properties
- Fine-tuning Logging
- Using YAML for Configuration
- More on Testing

# How Does Spring Boot Work?

- Extensive use of *pre-written* **@Configuration** classes
- Conditional on
  - The contents of the classpath
  - Properties you have set
  - Beans already defined
- **@Profile** is an example of conditional configuration
  - Spring Boot takes this idea to the next level

# @Conditional Annotations

- Allow conditional bean creation
  - Only create if other beans exist (or don't exist)

```
@Bean  
@ConditionalOnBean(name={"dataSource"})  
public JdbcTemplate jdbcTemplate(DataSource dataSource) {  
    return new JdbcTemplate(dataSource);  
}
```

- Or by type: @ConditionalOnBean(type={DataSource.class})
- Many others:
  - @ConditionalOnClass, @ConditionalOnProperty, ...  
@ConditionalOnMissingBean, @ConditionalOnMissingClass



@Profile is a special case of @Conditional

# What are AutoConfiguration Classes

- Pre-written Spring configurations
  - `org.springframework.boot.autoconfigure` package
  - See `spring-boot-autoconfigure` JAR file
    - Best place to check what they exactly do

```
@Configuration  
public class DataSourceAutoConfiguration  
    implements EnvironmentAware {  
  
    ...  
  
    @Conditional(...)  
    @ConditionalOnMissingBean(DataSource.class)  
    @Import(...)  
    protected static class EmbeddedConfiguration { ... }  
  
    ...  
}
```



Spring Boot defines many of these configurations. They activate in response to dependencies on the classpath

# Topics in this Session

- Understanding Auto-Configuration
- **Customizing Spring Boot**
- More on Properties
- Fine-tuning Logging
- Using YAML for Configuration
- More on Testing

# Controlling What Spring Boot does

- There are several options
  - Set some of Spring Boot's properties
  - Define certain beans yourself so Spring Boot won't
  - Explicitly disable some auto-configuration
  - Changing dependencies

# 1. Using Properties

- Spring Boot looks for **application.properties** in these locations (in this order):
  - /config sub-directory of the working directory
  - The working directory
  - config package in the classpath
  - classpath root
- Creates a *PropertySource* based on these files
- Many, many configuration properties available



See: [Spring Boot Reference, Appendix A. Common Application Properties](#)  
<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties>

## 2. Replacing Generated Beans

- Normally beans you declare *explicitly* disable any auto-created ones.
  - Example: Your **DataSource** stops Spring Boot creating a default **DataSource**
  - Bean name often not important
  - Works with Java Config, Component Scanning and/or XML

```
@Bean  
public DataSource dataSource() {  
    return new EmbeddedDatabaseBuilder()  
        .setName("RewardsDb").build();  
}
```

# 3. Selectively Disabling Auto Configuration

- Can disable some AutoConfiguration classes
  - If they don't suit your needs
- Use the `@EnableAutoConfiguration` annotation
  - Specify the auto-configuration classes to exclude

```
@EnableAutoConfiguration(exclude=DataSourceAutoConfiguration.class)
public class ApplicationConfiguration {
    ...
}
```

- Or use exclude property

```
spring.autoconfigure.exclude=
    org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

application.properties

## 4a. Overriding Dependency Versions

- Spring Boot POMs preselect the versions of frameworks
  - Ensures the versions of all frameworks are consistent
  - Avoids “*dependency hell*”
- Should I override the version of a given framework?
  - Ideally no, it makes your life more complicated
- But there are good reasons to override it sometimes
  - A bug in the given version
  - Company policies

## 4b. Overriding Dependency Versions

- Set the appropriate Maven property in your `pom.xml`

```
<properties>
    <spring.version>4.2.0.RELEASE</spring.version>
</properties>
```

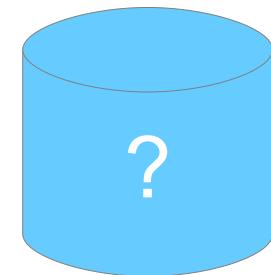
- Check this POM to know all the properties names
  - <https://github.com/spring-projects/spring-boot/blob/master/spring-boot-dependencies/pom.xml>



This only works if you *inherit* from the starter. You need to redefine the artifact if you directly import the dependency

# Configuration Example: DataSource (1)

- A common example of how to control or override Spring Boot's default configuration
- Typical customizations
  - Use the predefined properties
  - Change the underlying data source connection pool implementation
  - Define your own DataSource bean (shown earlier)



# *Example: DataSource Configuration (2)*

- Common properties configurable from properties file

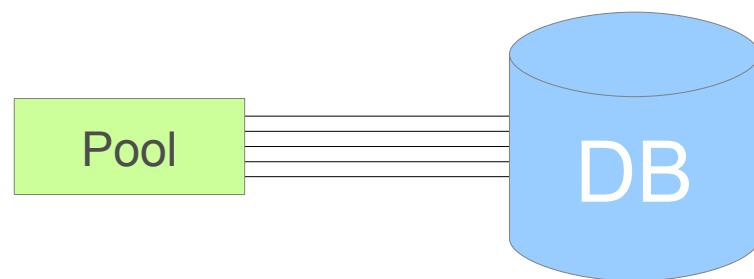
```
spring.datasource.url=          # Connection settings  
spring.datasource.username=  
spring.datasource.password=  
spring.datasource.driver-class-name=  
  
spring.datasource.schema=        # SQL scripts to execute  
spring.datasource.data=  
  
spring.datasource.initial-size=  # Connection pool settings  
spring.datasource.max-active=  
spring.datasource.max-idle=  
spring.datasource.min-idle=
```

# Example: DataSource Configuration (3)

- Spring Boot creates a pooled **DataSource** by default
  - If a known pool dependency is available
    - *spring-boot-starter-jdbc* or *spring-boot-starter-jpa* starters pull in *tomcat-jdbc* connection pool by default
  - Choices: Tomcat, HikariCP, Commons DBCP 1 & 2
    - Simply use relevant dependency

## Default pool:

- Spring Boot 1.x: Tomcat
- Spring Boot 2.x: Hikari



# *Example: Web Container Configuration*

- Many settings accessible from the configuration file

```
server.port=9000
server.address=192.168.1.20
server.session-timeout=1800
server.context-path=/rewards
server.servlet-path=/admin
```

- Also available
  - SSL (keystore, truststore for client authentication)
  - Tomcat specifics (access log, compression, etc)

# Topics in this Session

- Understanding Auto-Configuration
- Customizing Spring Boot
- **More on Properties**
- Fine-tuning Logging
- Using YAML for Configuration
- More on Testing

# Overriding Properties

Applies to Spring  
or Spring Boot

- Order of evaluation of the properties (non-exhaustive)
  - Command line arguments
  - Java system properties
  - OS environment variables
  - Property file(s) – including `application.properties`
- Can access any of them using `@Value` in the usual way
- *Recommendation:*
  - Use Property files to define defaults
  - Override *externally* using one of the other 3 options

# Relaxed Property Binding

- No need for exact match between desired properties and names
- Intuitive mapping between Java-style properties and environment variables
  - `path` equivalent to `PATH`
  - `java.home` equivalent to `JAVA_HOME`
  - Easy overriding of property without changing the name!

```
@Configuration  
class AppConfig {  
  
    @Value("${java.home}")  
    String javaInstallDir;  
  
    ...  
}
```

# The Problem with Property Placeholders

- Using property placeholders is sometimes cumbersome
  - Many properties, prefix has to be repeated

```
@Configuration  
public class RewardsClientConfiguration {  
  
    @Value("${rewards.client.host}") String host;  
    @Value("${rewards.client.port}") int port;  
    @Value("${rewards.client.logdir}") String logdir;  
    @Value("${rewards.client.timeout}") int timeout;  
  
    ...  
}
```

# Use @ConfigurationProperties

- Add `@ConfigurationProperties` to *dedicated* container bean
  - Will hold the externalized properties
  - Avoids repeating the prefix
  - Data-members automatically set from corresponding properties

```
@ConfigurationProperties(prefix="rewards.client")
public class ConnectionSettings {

    private String host;
    private int port;
    private String logdir;
    private int timeout;
    ... // getters/setters
}
```

```
rewards.client.host=192.168.1.42
rewards.client.port=8080
rewards.client.logdir=/logs
rewards.client.timeout=2000
example.properties
```

# Use `@EnableConfigurationProperties`

- `@EnableConfigurationProperties` on configuration class
  - Specify and auto-inject the container bean

```
@Configuration  
@EnableConfigurationProperties(ConnectionSettings.class)  
public class RewardsClientConfiguration {  
    // Spring initialized this automatically  
    @Autowired ConnectionSettings connectionSettings;  
  
    @Bean public RewardClient rewardClient() {  
        return new RewardClient(  
            connectionSettings.getHost(),  
            connectionSettings.getPort(), ...  
        );  
    }  
}
```

# Topics in this Session

- Understanding Auto-Configuration
- Customizing Spring Boot
- More on Properties
- **Fine-tuning Logging**
- Using YAML for Configuration
- More on Testing

# Logging frameworks

- Spring Boot includes by default
  - SLF4J: logging facade
  - Logback: SLF4J implementation
- Best practice: stick to this in your application
  - Use the SLF4J abstraction the application code
- Other logging frameworks are supported
  - Java Util Logging, Log4J, Log4J2

# Using another logging framework

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-websocket</artifactId>
  <exclusions>
    <exclusion>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

Excludes Logback

Includes Log4J

# Logging Output

- Spring Boot logs by default to the console
- Can also log to rotating files
  - Specify file OR path in application.properties

```
# Use only one of the following properties

# absolute or relative file to the current directory
logging.file=rewards.log

# will write to a spring.log file
logging.path=/var/log/rewards
```



Spring Boot can also configure logging by using the appropriate configuration file of the underlying logging framework.

# Topics in this Session

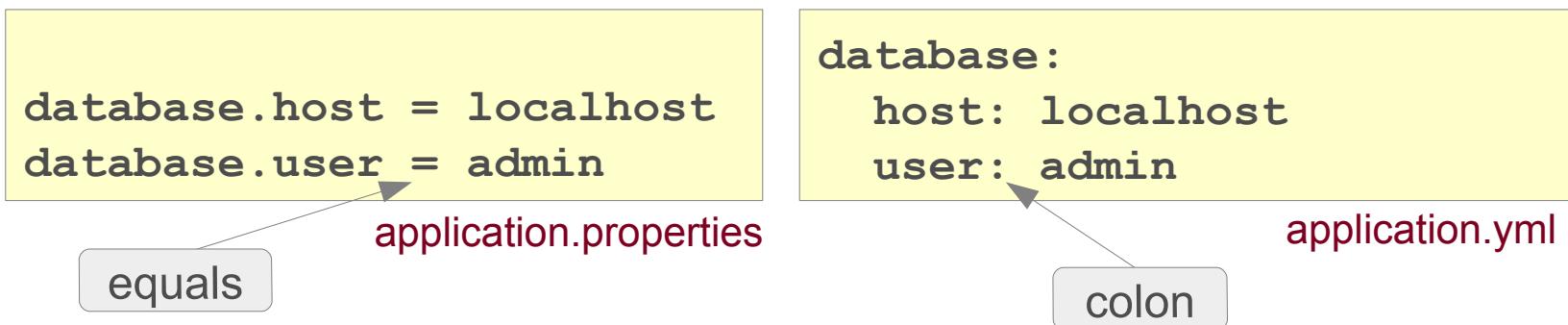
- Understanding Auto-Configuration
- Customizing Spring Boot
- More on Properties
- Fine-tuning Logging
- **Using YAML for Configuration**
- More on Testing

# What is YAML?

- *Yaml Ain't a Markup Language*
  - Recursive acronym
- Created in 2001
- Alternative to .properties files
  - Allows hierarchical configuration
- Java parser for YAML is called SnakeYAML
  - Must be in the classpath
  - Provided by spring-boot-starters

# YAML for Properties

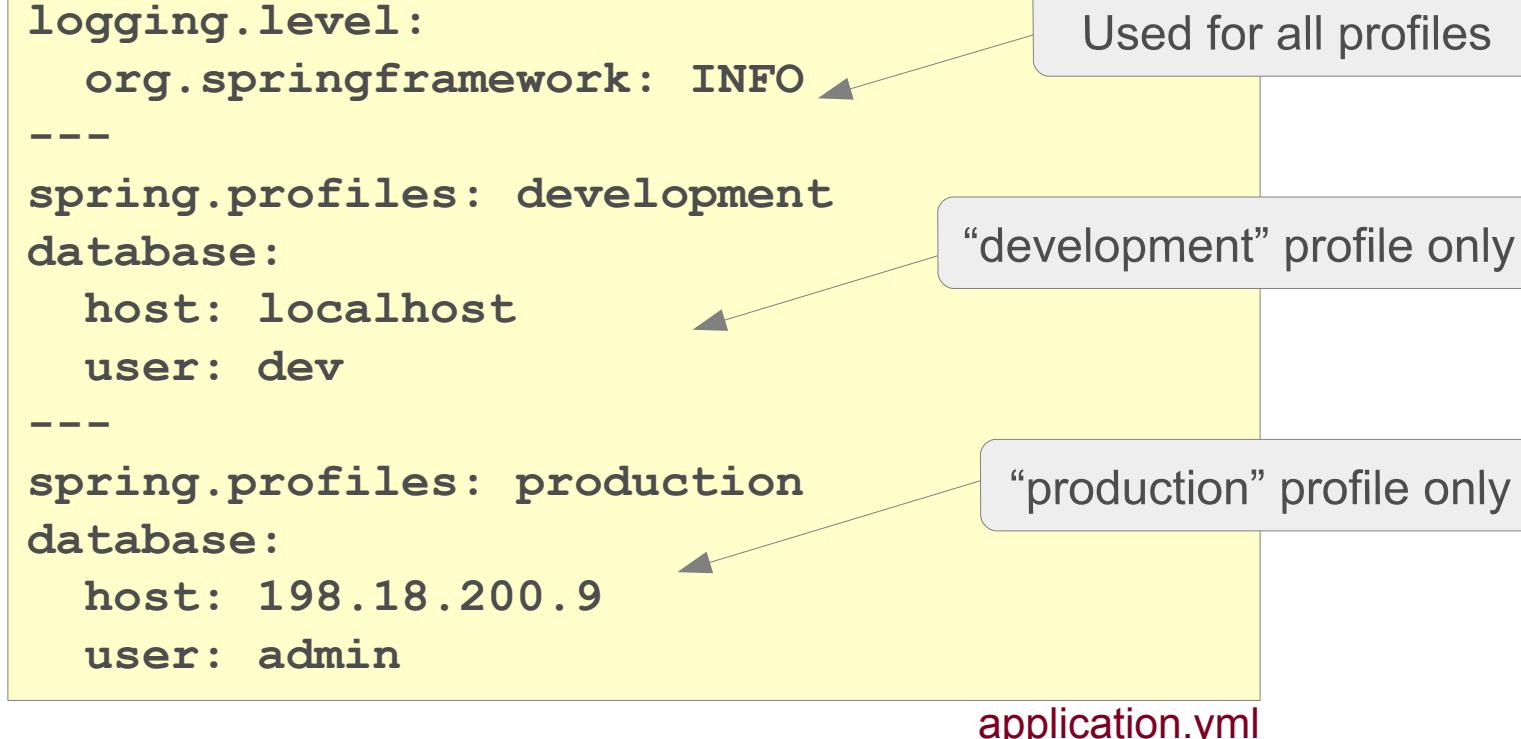
- Spring Boot support YAML for Properties
  - An alternative to properties files



- YAML is convenient for hierarchical configuration data
  - Spring Boot properties are organized in groups
  - Examples: server, database, etc

# Multiple Profiles Inside a Single YAML File

- YAML file can contain configuration for multiple profiles
  - '---' implies a separation between profiles



# Multiple Profiles Inside Multiple Files

```
logging.level:  
    org.springframework: INFO  
---  
spring.profiles: development  
database:  
    host: localhost  
    user: dev  
---  
spring.profiles: production  
database:  
    host: 198.18.200.9  
    user: admin
```

application.yml

```
logging.level:  
    org.springframework: INFO
```

application.yml

```
database:  
    host: localhost  
    user: dev
```

application-development.yml

```
database:  
    host: 198.18.200.9  
    user: admin
```

application-production.yml



Alternatively `application-development.properties` and `application-production.properties` can be used in same way

# Topics in this Session

- Understanding Auto-Configuration
- Customizing Spring Boot
- More on Properties
- Fine-tuning Logging
- Using YAML for Configuration
- **More on Testing**

# Recall: `@SpringBootTest`

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes=TransferApplication.class,
               webEnvironment=WebEnvironment.NONE)
```

```
public class TransferServiceTests {
    @Autowired
    private TransferService transferService;
```

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

```
@SpringBootApplication
public class TransferApplication {
    public static void main(String[] args) {
        SpringApplication.run(TransferApplication.class, args);
    }
}
```

Sets up *same* configuration  
for the tests that the  
application would use

Control the application  
context (next slide)

# Test Setup Options

`@SpringBootTest (webEnvironment=?)`

Value	Description
<b>MOCK</b> (default)	<ul style="list-style-type: none"><li>• Loads a <code>WebApplicationContext</code></li><li>• Setup Mock MVC environment</li><li>• No embedded servlet container</li><li>• Falls back to <b>NONE</b> if servlet APIs not on classpath</li></ul>
<b>RANDOM_PORT</b>	<ul style="list-style-type: none"><li>• <code>WebApplicationContext</code></li><li>• Loads an embedded servlet container</li><li>• Listens on a random port</li></ul>
<b>DEFINED_PORT</b>	<ul style="list-style-type: none"><li>• <code>WebApplicationContext</code></li><li>• As <b>RANDOM_PORT</b> but listens on 8080 or <code>server.port</code> value</li></ul>
<b>NONE</b>	<ul style="list-style-type: none"><li>• <code>AnnotationConfigApplicationContext</code></li><li>• No servlet environment</li></ul>

# Web Application Testing

Alternative  
*without* Spring Boot

- Spring Unit test with `@WebAppConfiguration`
  - Creates a `WebApplicationContext`
  - Can test code that uses web features
    - `ServletContext`, `Session` and `Request` bean scopes
  - Configures the location of resources
    - Defaults to `src/main/webapp`
      - Override using annotation's `value` attribute
    - For classpath resources use `classpath:` prefix

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
public final class TransferServiceTests { ... }
```

JUnit 4

# @SpringJUnitWebConfig

- `@SpringJUnitWebConfig` is a composed annotation that combines
  - JUnit 5's `@ExtendWith(SpringExtension.class)`
  - Spring's `@ContextConfiguration` and `@WebAppConfiguration`

JUnit 5

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

# Summary

- Spring Boot takes care of boilerplate configuration
  - Auto-configuration can be overridden/disabled
  - Frameworks versions can be overridden too
- Spring Boot enhances Spring configuration externalization mechanisms
  - Properties/YAML files
  - Easier to override using env/Java system variables

# Spring Security

## Web Application Security

Addressing Common Web Application Security Requirements



# Objectives

- After completing this lesson, you should be able to:
  - Explain basic Security Concepts
  - Setup Spring Security in a Web Environment
  - Configuring Web Authentication
  - Setup Method Security



# Topics in this Session



- **Security Overview**
- Spring Security in a Web Environment
- Configuring Web Authentication
- Method Security
- Lab
- Advanced Security



See: **Spring Security Reference**

<http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/>

# 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
- **Authority**
  - Permission or credential enabling access (such as a role)
- **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
- Authorization determines if you have the required *Authority*
- The decision process is often based on roles
  - *ADMIN* can cancel orders
  - *MEMBER* can place orders
  - *GUEST* can browse the catalog



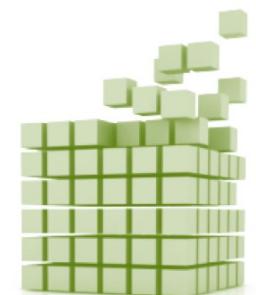
A *Role* is simply a commonly used type of *Authority*

# Motivations - I

- **Portable**
  - Secured archive (WAR, EAR) can be deployed as-is
  - Also works with Spring Boot or standalone
  - 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)
    - Properties file, RDBMS, LDAP, custom DAOs, etc.
- **Extensible**
  - All the following can be customized
    - How a principal is defined
    - How authorization decisions are made
    - Where security constraints are stored

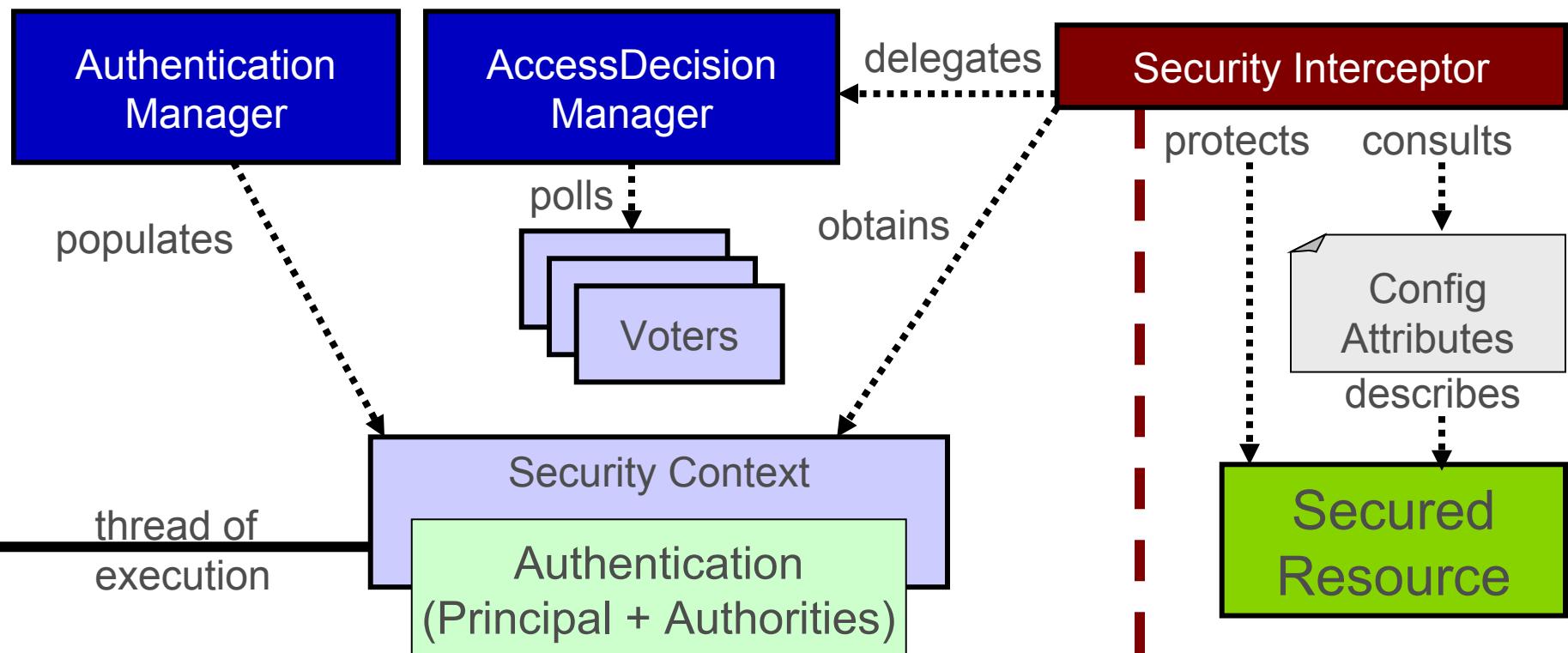


# Consistency of Approach

- *The goal of authentication is always the same*
  - Regardless of the underlying 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 the underlying mechanism
  - Consult the attributes of secured resource
  - Obtain principal information from security context
  - Grant or deny access



# Spring Security – the Big Picture



# Topics in this Session

- Security Overview
- **Spring Security in a Web Environment**
- Configuring Web Authentication
- Method Security
- Lab
- Advanced Security

# Setup and Configuration

## Spring Security in a Web Environment



Three steps

- 1) Setup filter chain
- 2) Configure security (authorization) rules
- 3) Setup Web Authentication



Spring Security is **not** limited to Web security, but that is all we will consider here, and it is configurable “out-of-the-box”

# Spring Security Filter Chain – 1

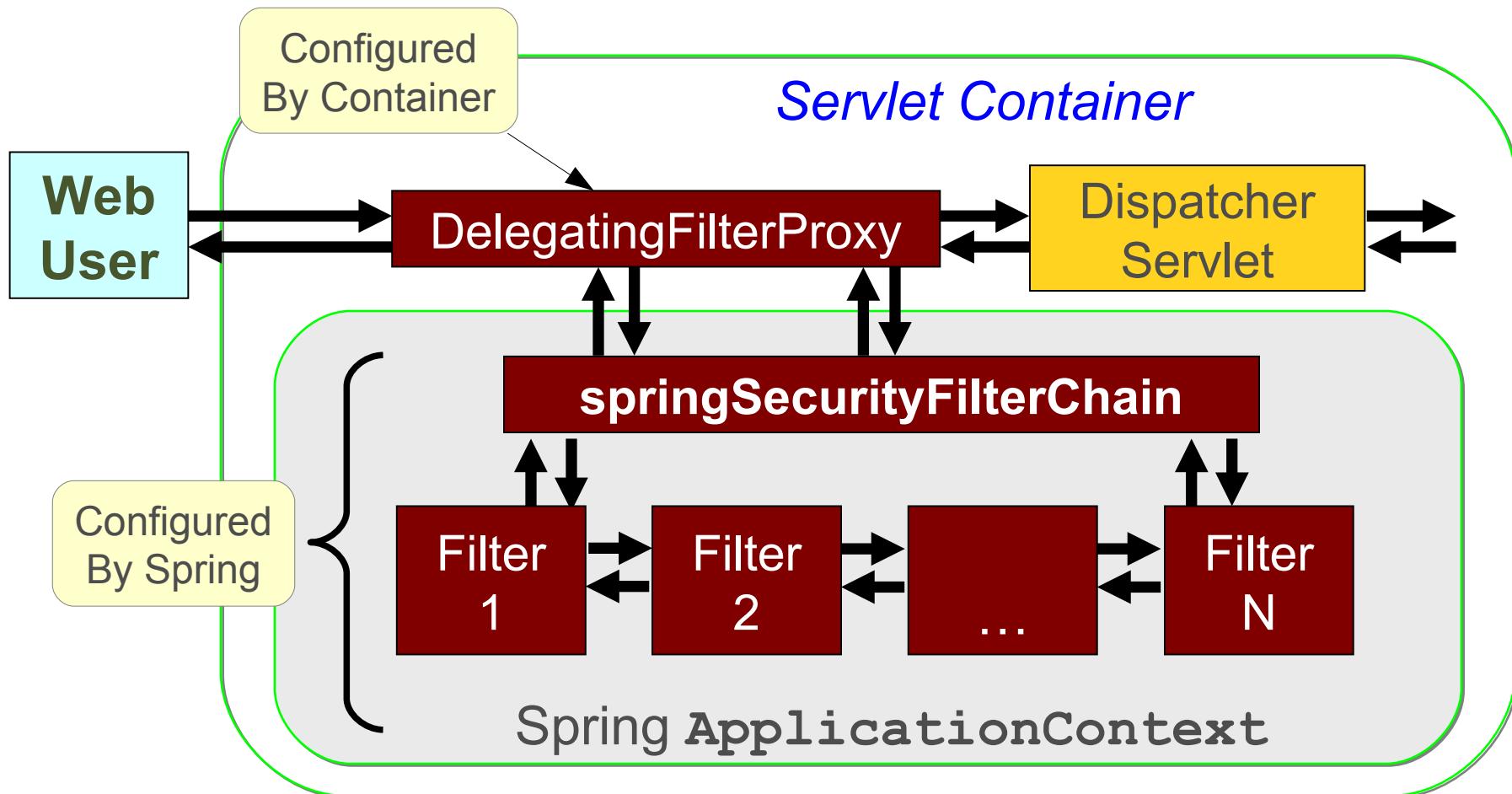


- Implementation is a *chain* of Spring configured *filters*
  - Requires a `DelegatingFilterProxy` which *must* be called `springSecurityFilterChain`
  - Chain consists of many filters (next slide)
- Setup filter chain using *one* of these options
  - Spring Boot does it automatically
  - Subclass `AbstractSecurityWebApplicationInitializer`
  - Declare as a `<filter>` in `web.xml`



For more details (and non-Boot examples) see “Advanced security: working with *filters*” at end of this topic.

# Spring Security Filter Chain – 2



All implement `java.servlet.Filter`

# Configuration in the Application Context

- Extend `WebSecurityConfigurerAdapter`

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
    @Override  
    protected void configure(HttpSecurity http) throws Exception {
```

```
}
```

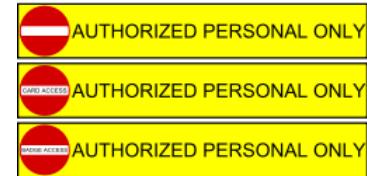
Web-specific security settings

```
    @Autowired  
    public void configureGlobal(AuthenticationManagerBuilder auth)  
        throws Exception {
```

```
}
```

Global security settings  
(authentication manager, ...).

# Authorizing URLs



- Define specific authorization restrictions for URLs
- Support “Ant-style” pattern matching
  - `"/admin/*"` only matches `"/admin/xxx"`
  - `"/admin/**"` matches any path under `/admin`
    - Such as `"/admin/database/access-control"`

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/admin/**").hasRole("ADMIN")  
        ...  
}
```

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

User must have  
**ADMIN** role



# URL Matching

- **antMatchers ()** vs **mvcMatchers**

```
http.authorizeRequests()  
  
    // Only matches /admin  
    .antMatchers("/admin").hasRole("ADMIN")  
  
    // Matches /admin, /admin/, /admin.html, /admin.xxx  
    .mvcMatchers("/admin").hasRole("ADMIN")
```

- **mvcMatchers**
  - Newer API
  - Uses same matching rules as `@RequestMapping`
  - Typically *more secure, recommended*



# More on `authorizeRequests()`

- *Chain* multiple restrictions
  - Evaluated in the order listed
    - First match is used, *put specific matches first*

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
            .mvcMatchers("/signup", "/about").permitAll()  
            .mvcMatchers("/accounts/edit*").hasRole("ADMIN")  
            .mvcMatchers("/accounts/**").hasAnyRole("USER", "ADMIN")  
            .anyRequest().authenticated();
```

Must be authenticated for any other request

# Specifying login and logout

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
            .mvcMatchers("/admin/**").hasRole("ADMIN")...  
            .and()                                // method chaining!  
  
        .formLogin()                            // setup form-based authentication  
            .loginPage("/login")                  // URL to use when login is needed  
            .permitAll()                        // any user can access  
            .and()                                // method chaining!  
  
        .logout()                               // configure logout  
            .logoutSuccessUrl("/home")          // go here after successful 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.

```
<form action="/login" method="POST">
  <input type="text" name="username"/> ←
  <br/>
  <input type="password" name="password"/> ←
  <br/>
  <input type="submit" name="submit" value="LOGIN"/>
</form>
```

The expected keys  
for generation of an  
authentication  
request token

*login.html*



# By-passing Security

- Some URLs do not need to be secure
  - Such as static resources
  - Using `permitAll()` allows open-access
    - But still processed by Spring Security
- Possible to skip security checks completely

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
    @Override  
    protected void configure(WebSecurity web) throws Exception {  
        web.ignoring().mvcMatchers("/css/**", "/images/**", "/javascript/**");  
    }  
    ...  
}
```

These URLs pass straight through, no checks

# Topics in this Session

- Security Overview
- Spring Security in a Web Environment
- **Configuring Web Authentication**
- Method Security
- Lab
- Advanced Security



# Configure Authentication

- DAO Authentication provider is default
  - Expects a *UserDetailsService* implementation to provide credentials and authorities
    - Options: In-memory (properties), JDBC (database), Custom
- **LdapAuthenticationProvider**
  - Integrate with OpenLDAP or Active Directory
- 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 Manager

- Use a *UserDetailsManagerConfigurer*
  - Some built in options:
    - In-memory (for quick testing), JDBC
  - 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 `AuthenticationManagerBuilder`. It shows the `inMemoryAuthentication()` method being called, which takes three parameters: `login`, `password`, and `Supported roles`. A callout box labeled "Adds a *UserDetailsManagerConfigurer*" points to the `inMemoryAuthentication()` method. Another callout box labeled "Not web-specific" points to the entire code block.

# Sourcing Users from a Database – 1

```
private DataSource dataSource;  
  
@Autowired  
public void setDataSource(DataSource dataSource) throws Exception {  
    this.dataSource = dataSource;  
}  
  
@Autowired  
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
    auth.jdbcAuthentication().dataSource(dataSource);  
}
```

Can customize queries using methods:  
`usersByUsernameQuery()`  
`authoritiesByUsernameQuery()`  
`groupAuthoritiesByUsername()`

# Sourcing Users from a Database – 2

Queries RDBMS for users and their authorities

- Provides default queries
  - `SELECT username, password, enabled FROM users WHERE username = ?`
  - `SELECT username, authority FROM authorities WHERE username = ?`
- Groups also supported
  - `groups`, `group_members`, `group_authorities` tables
  - See online documentation for details
- Advantage
  - Can modify user info while system is running

# Password Encoding – 1

Note: sha and md5  
only suitable for testing  
– too insecure

- Can encode passwords using a hash
  - sha256, bcrypt, (sha, md5, ...)
  - Use with *any* authentication mechanism

SHA-256 by default

```
auth.inMemoryAuthentication()  
.passwordEncoder(new StandardPasswordEncoder());
```

- Add a “salt” string to make encryption stronger
  - Salt prepended to password before hashing

encoding with  
a 'salt' string

```
auth.jdbcAuthentication()  
.dataSource(dataSource)  
.passwordEncoder(new StandardPasswordEncoder("Spr1nGi$Gre@t"));
```

# Password Encoding – 2

**BCryptPasswordEncoder** is recommended – uses Blowfish

- BCrypt is recommended over SHA-256
  - Secure passwords further by specifying a “strength” (N)
  - Internally the hash is rehashed  $2^N$  times, default is  $2^{10}$

```
auth.inMemoryAuthentication()  
.passwordEncoder(new BCryptPasswordEncoder(12));
```

Encoding using  
'strength' 12

- Store *only* encrypted passwords

```
auth.inMemoryAuthentication()  
.withUser("hugie")  
.password("$2a$10$aMxNkanIJ...Ha.h5NKknElEuylt87PNlicYpl1y.IG0C.")  
.roles("GENERAL")
```

# 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

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {  
    protected void configure(HttpSecurity http) throws Exception {  
        http.authorizeRequests().mvcMatchers("/resources/**").permitAll();  
    }  
}
```

```
@Configuration  
@EnableWebSecurity  
@Profile("development")
```

*Use in-memory provider*

```
public class SecurityDevConfig extends SecurityBaseConfig {  
    @Override  
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
        auth.inMemoryAuthentication()  
            .withUser("hughie").password("hughie").roles("GENERAL");  
    }  
}
```

# @Profile with Security Configuration

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {  
    protected void configure(HttpSecurity http) throws Exception {  
        http.authorizeRequests().mvcMatchers("/resources/**").permitAll();  
    }  
}
```

```
@Configuration  
@EnableWebSecurity  
@Profile("!development")
```

```
public class SecurityProdConfig extends SecurityBaseConfig {  
    @Override  
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
        auth.jdbcAuthentication().dataSource(dataSource);  
    }  
}
```

*Use database provider*

Use this profile when “development” *not* defined

# Topics in this Session

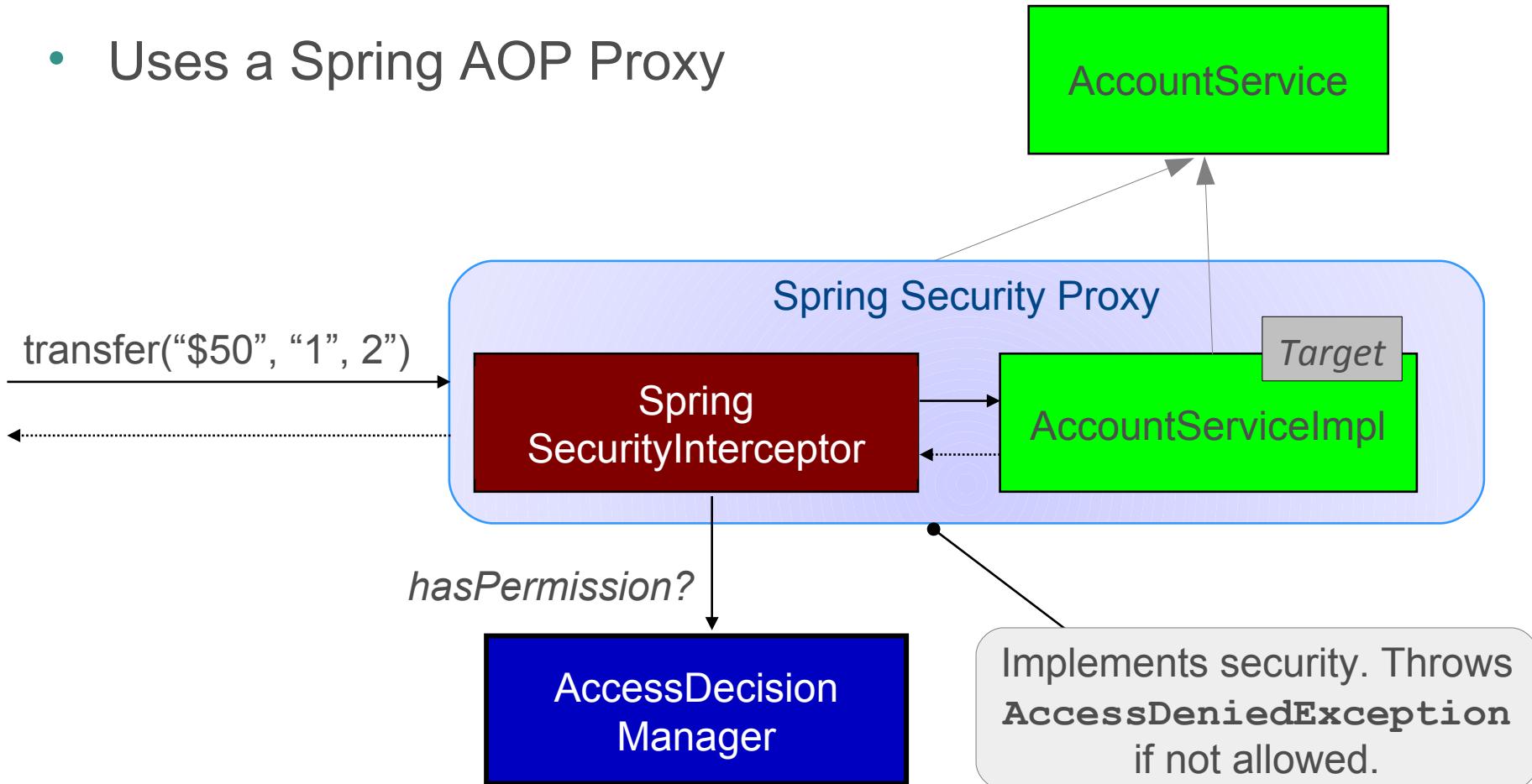
- Security Overview
- Spring Security in a Web Environment
- Configuring Web Authentication
- **Method Security**
- Lab
- Advanced Security

# Method Security

- Spring Security uses AOP for security at the method level
  - annotations based on Spring annotations or JSR-250 annotations
  - Java configuration to activate detection of annotations
- Typically secure your services
  - Do not access repositories directly, bypasses security (and transactions)

# Method Security – How it Works

- Uses a Spring AOP Proxy



# Method Security - JSR-250

- Only supports **role-based** security (hence the name)
  - JSR-250 annotations must be enabled

```
@EnableGlobalMethodSecurity(jsr250Enabled=true)
```

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

```
public class ItemManager {
```

```
    @RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})
```

```
    public Item findItem(long itemNumber) {
```

```
        ...
```

```
}
```

```
}
```

```
    @RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})
```

Can also place  
at class level



Internally role authorities are stored with `ROLE_` prefix. APIs seen previously hide this. Here you *must* use full name

# Method Security with SpEL

- Use Pre/Post annotations for SpEL

```
@EnableGlobalMethodSecurity(prePostEnabled=true)
```

```
import org.springframework.security.annotation.PreAuthorize;  
  
public class ItemManager {  
    @PreAuthorize("hasRole('MEMBER') && " +  
                 "#order.owner.id == principal.user.id")  
    public Item findItem(Order order, long itemNumber) {  
        ...  
    }  
}
```



Full role-names *not* required. **ROLE\_** prepended automatically.



# Summary

- Spring Security
  - Secure URLs using a chain of Servlet filters
  - And/or methods on Spring beans using AOP proxies
- Out-of-the-box setup usually sufficient – you define:
  - URL and/or method restrictions
  - How to login (typically using an HTML form)
  - Supports in-memory, database, LDAP credentials (and more)
  - Password encryption using familiar hashing techniques
  - Support for security tags in JSP views

# Lab

Applying Spring Security to a Web Application

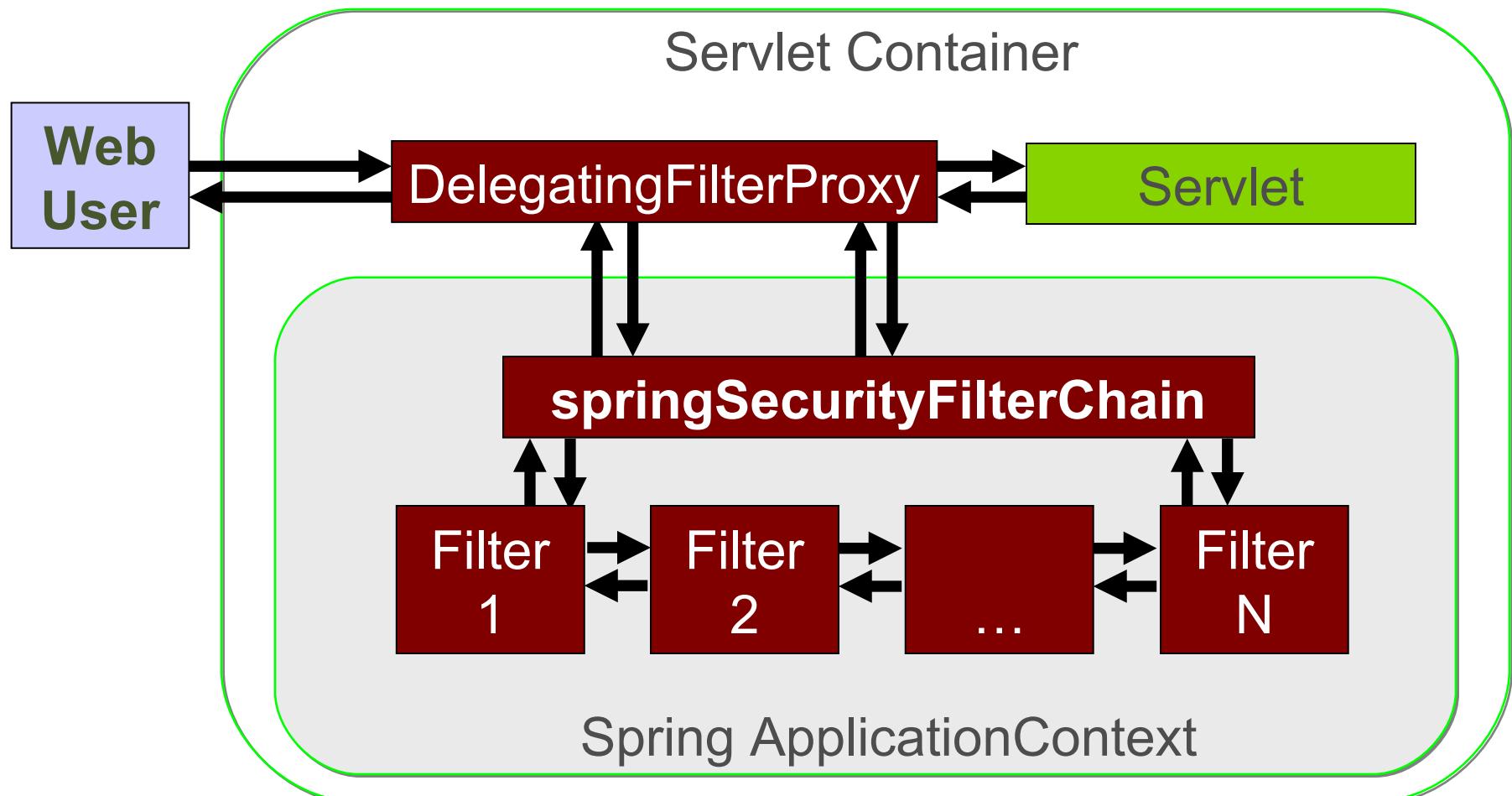
# Topics in this Session

- Security Overview
- Spring Security in a Web Environment
- Configuring Web Authentication
- Method Security
- **Advanced Security**
  - Working with Filters
  - Global Configuration Choices
  - Legacy Applications

# Spring Security in a Web Environment

- *SpringSecurityFilterChain*
  - **Always** first filter in chain
- This single proxy filter delegates to a chain of Spring-managed filters to:
  - Drive authentication
  - Enforce authorization
  - Manage logout
  - Maintain SecurityContext in HttpSession
  - and more

# Web Security Filter Configuration



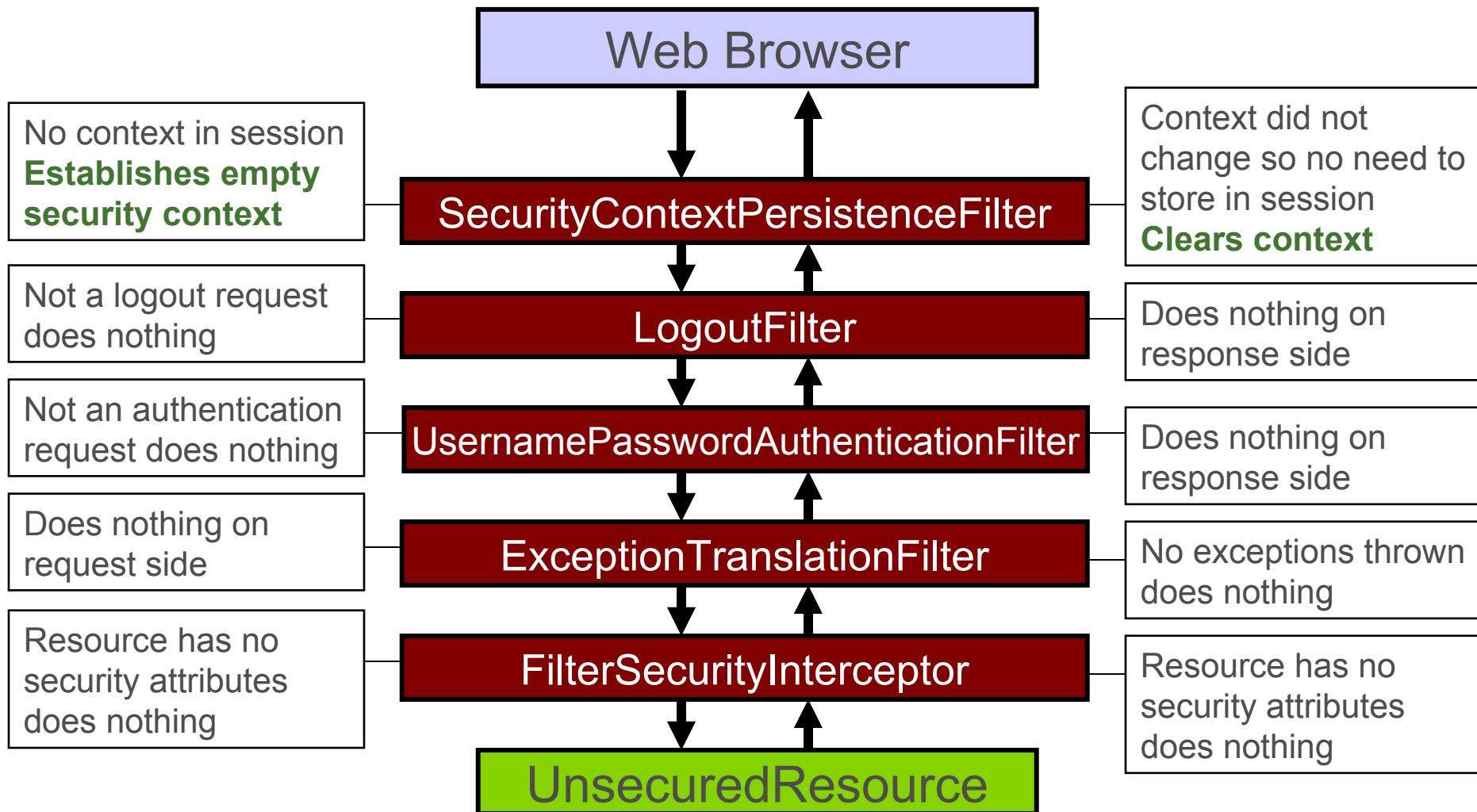
# The Filter Chain

- Spring Security uses a chain of many, many filters
  - Filters 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

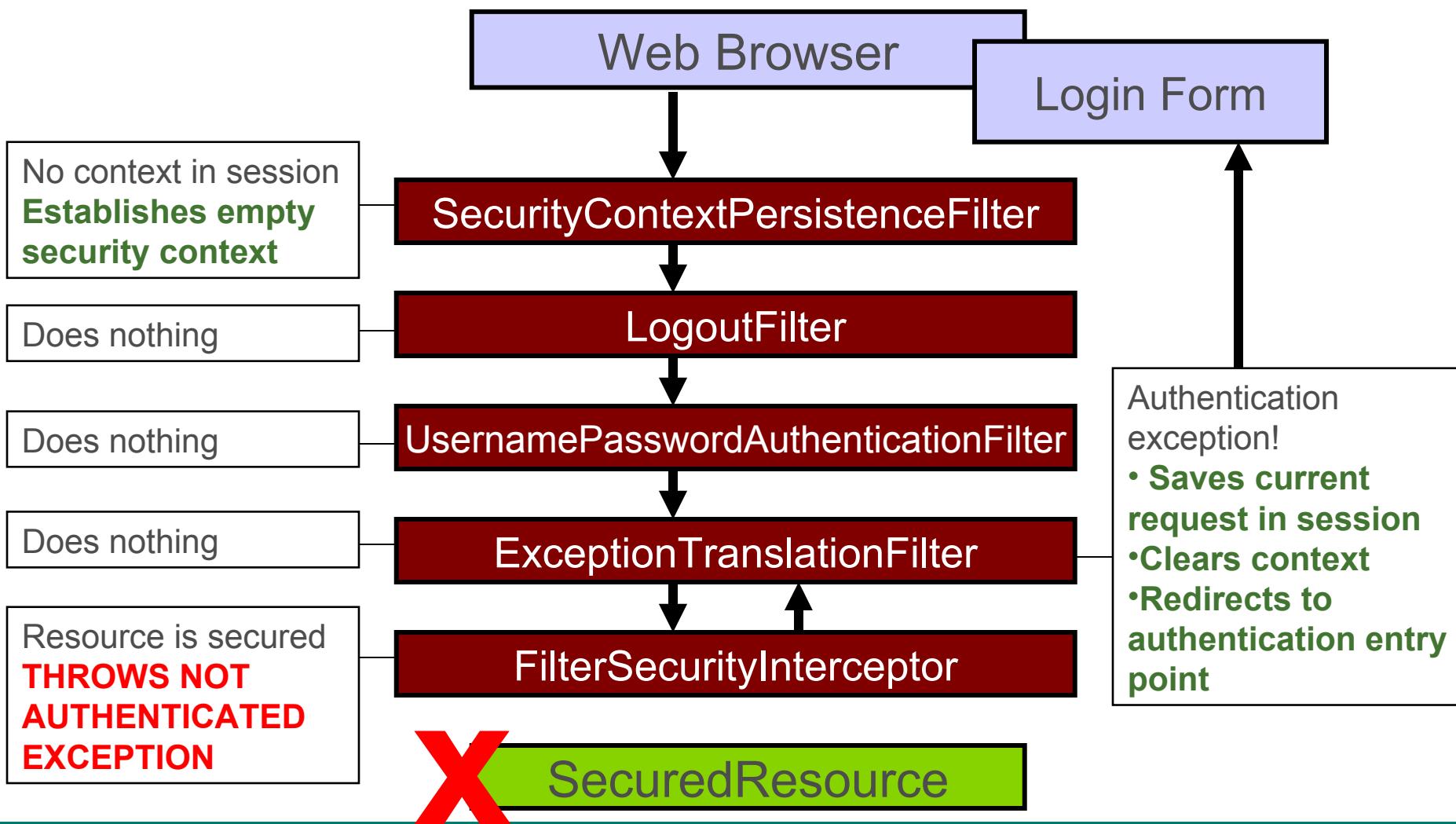


Spring Security originally developed independently of Spring – called *ACEGI Security* and involved far more manual configuration

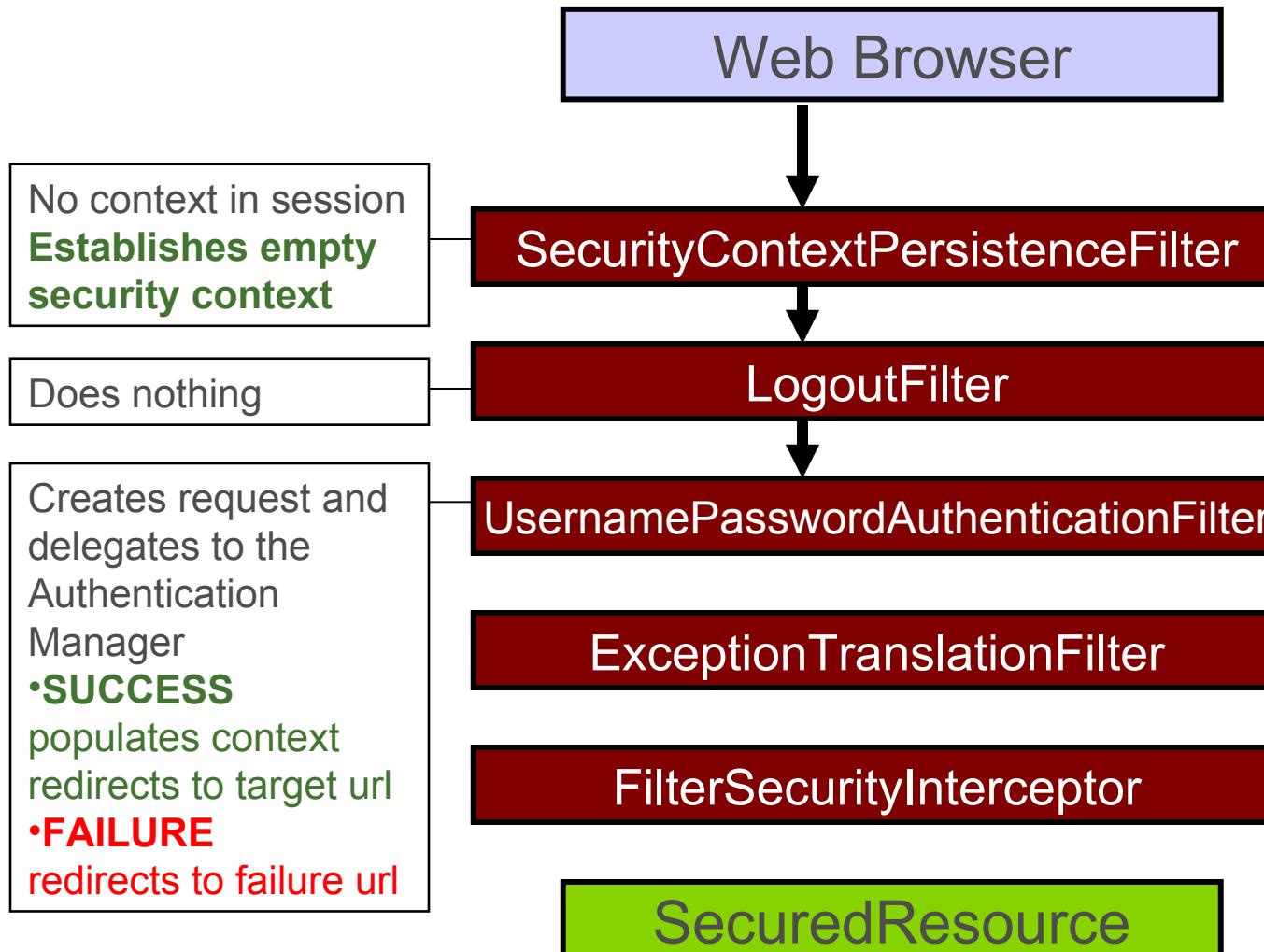
# Access Unsecured Resource Prior to Login



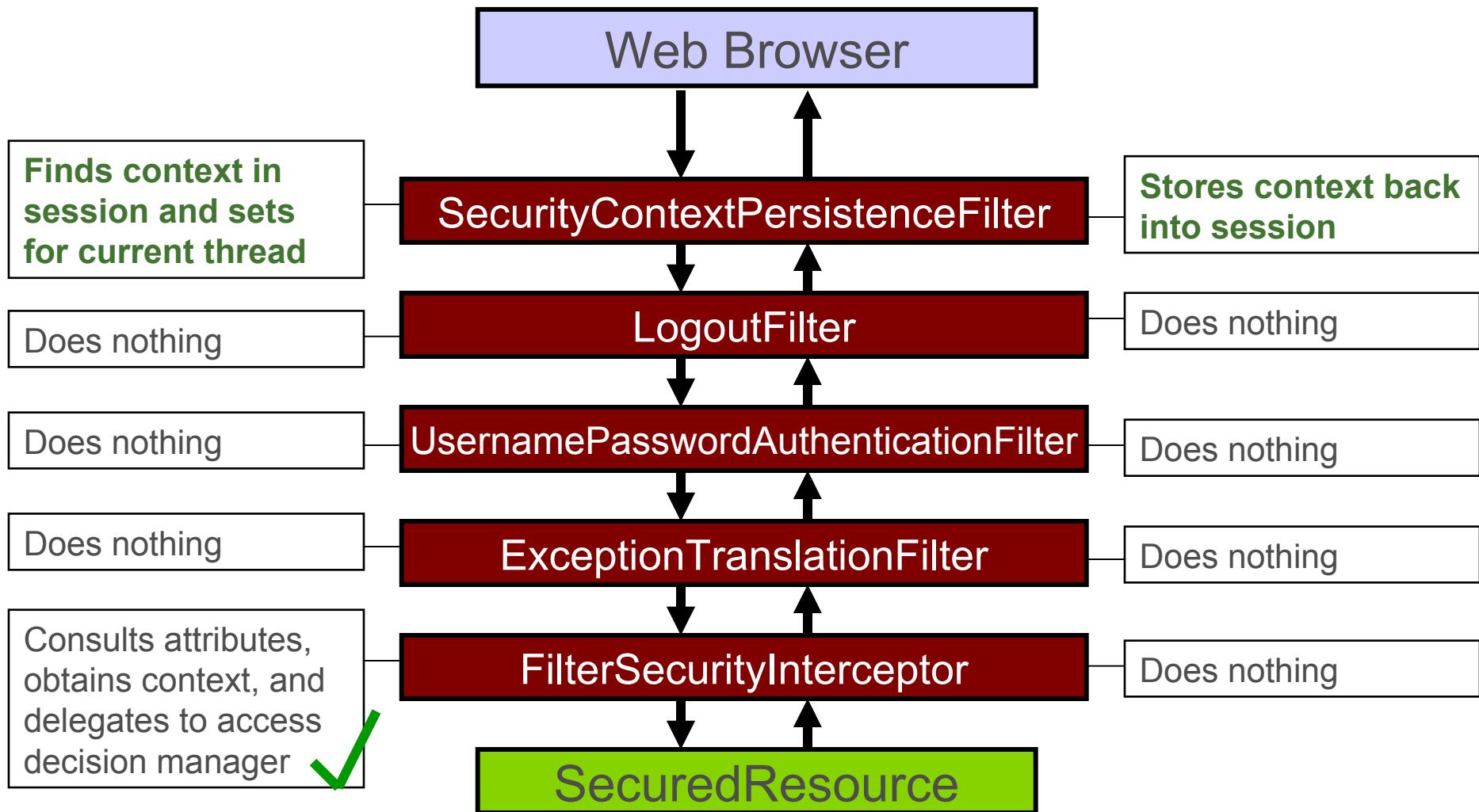
# Access Secured Resource Prior to Login



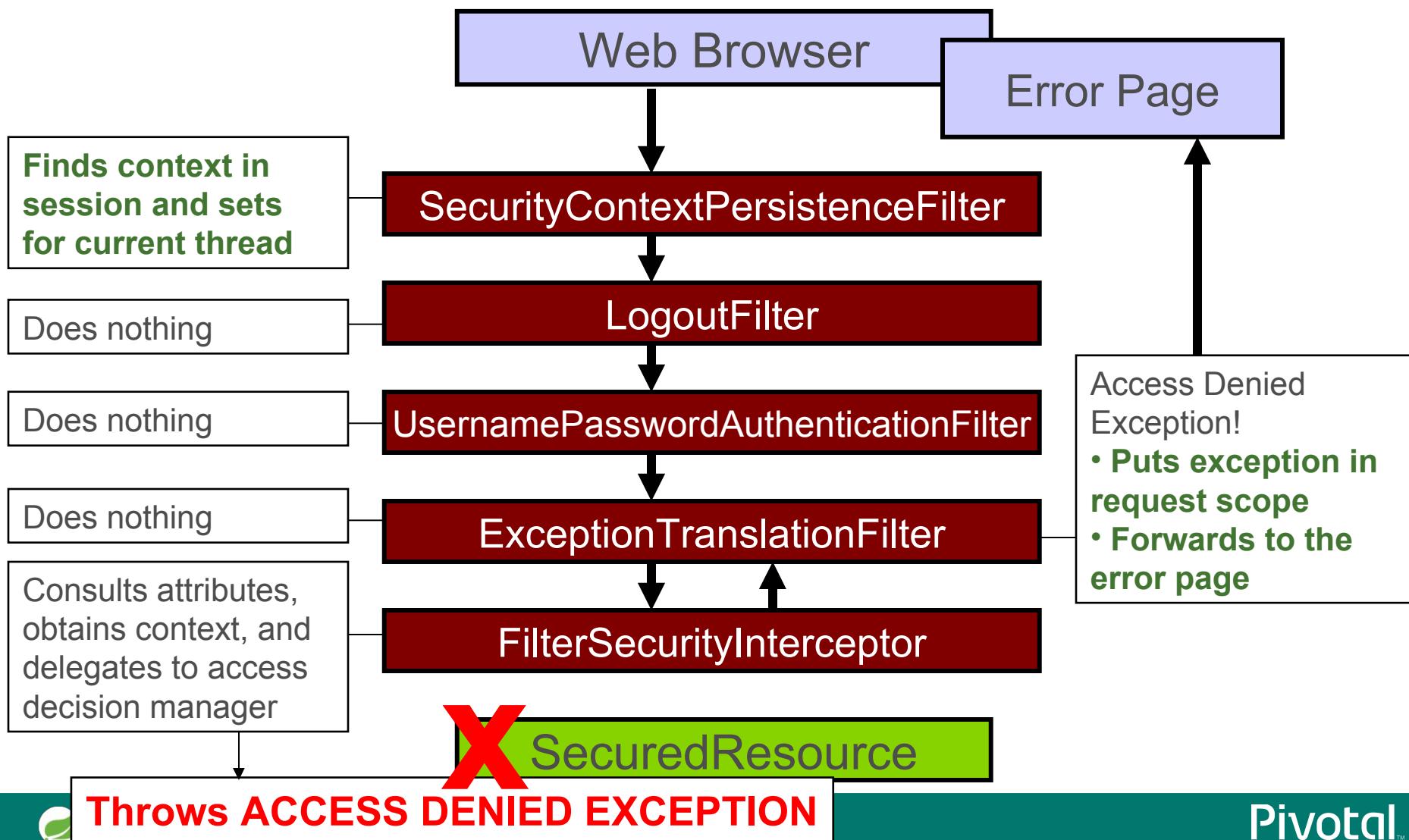
# Submit Login Request



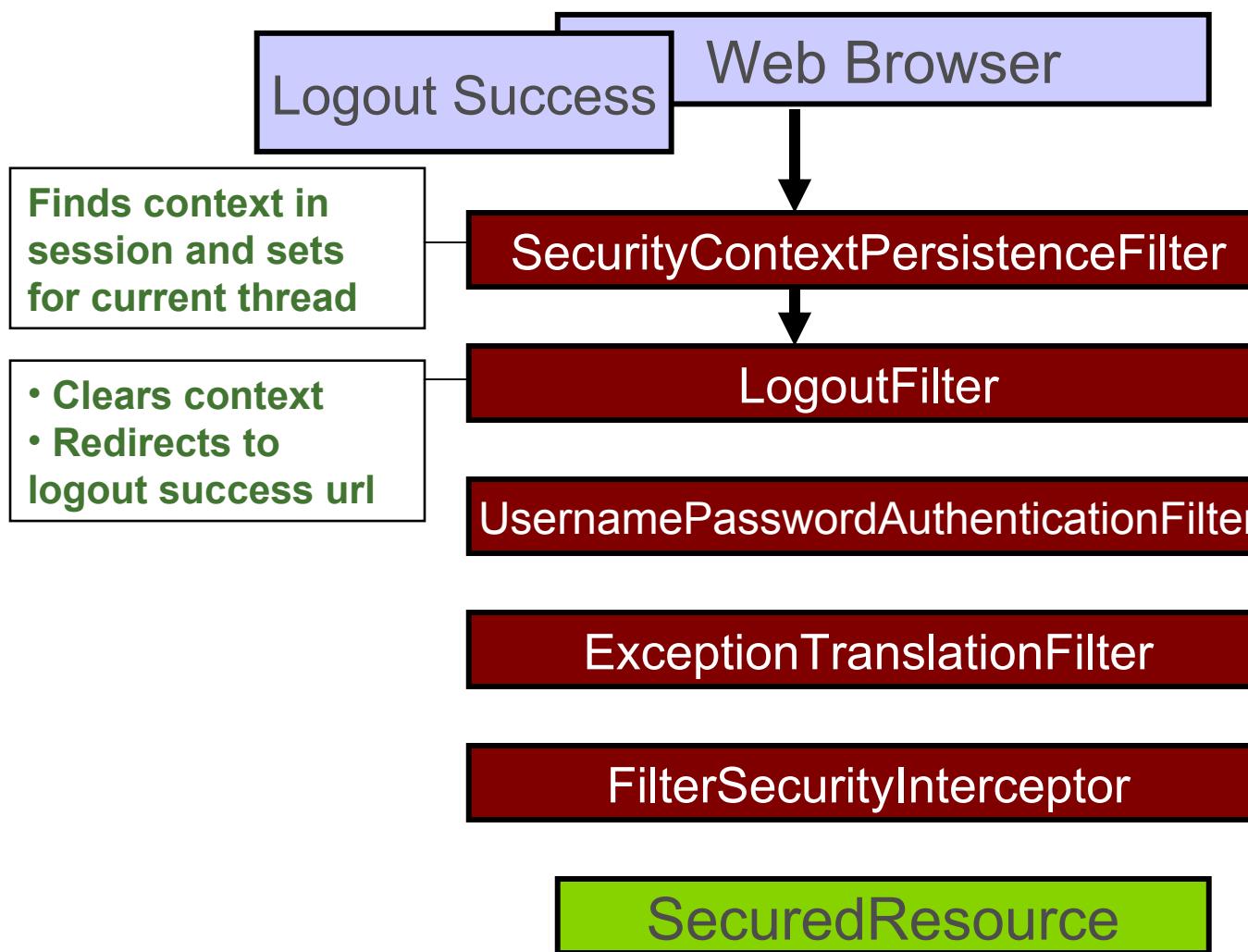
# Access Resource With Required Role



# 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 – Replace Filter

- Filters can be **replaced** in the chain
  - Replace an existing filter with your own
    - Replacement must extend the filter being replaced

```
public class MyCustomLoginFilter  
    extends UsernamePasswordAuthenticationFilter {}
```

```
@Bean  
public Filter loginFilter() {  
    return new MyCustomLoginFilter();  
}
```

```
http.addFilter ( loginFilter() );
```

# Custom Filter Chain – Add Filter

- Filters can be **added** to the chain
  - After any filter

```
public class MyExtraFilter extends Filter { ... }
```

```
@Bean  
public Filter myExtraFilter() {  
    return new MyExtraFilter();  
}
```

```
http.addFilterAfter ( myExtraFilter(),  
    UsernamePasswordAuthenticationFilter.class );
```

# Topics in this Session

- Security Overview
- Spring Security in a Web Environment
- Configuring Web Authentication
- Method Security
- **Advanced Security**
  - Working with Filters
  - **Global Configuration Choices**
  - Legacy Applications

# Global Configuration Choices

1. Add an autowired method to your security configuration
  - As shown in these slides: `configureGlobal(...)`
2. Override `WebSecurityConfigurerAdapter's` `configure(AuthenticationManagerBuilder auth)`
  - Applies *only* to web-configuration, *not* to methods
3. Extend `GlobalAuthenticationConfigurerAdapter`
  - Equivalent to option 1, more control
  - Can setup *multiple* authentication schemes

# Topics in this Session

- Security Overview
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method Security
- **Advanced Security**
  - Working with Filters
  - Global Configuration Choices
  - **Legacy Applications**

# Configuration without Spring Boot

## Servlet 2 using `web.xml`

- Define the single proxy filter
  - `springSecurityFilterChain` is a mandatory name
    - Refers to an existing Spring bean with same name

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

*web.xml*

This name is mandatory

# Configuration without Spring Boot

## *Servlet 3 WebApplicationInitializer*

- Declare your own subclass of **AbstractSecurityWebApplicationInitializer**
  - Sets up the **DelegatingFilterProxy**
  - Automatically called by Spring because it implements **WebApplicationInitializer**

```
import org.springframework.security.web.  
    context.AbstractSecurityWebApplicationInitializer;  
  
public class SecurityWebApplicationInitializer  
    extends AbstractSecurityWebApplicationInitializer {  
}  
}
```

Class is meant to be empty – nothing else is required

# Method Security - @Secured

You may see  
this in older  
applications

- Secured annotation should be enabled

```
@EnableGlobalMethodSecurity(securedEnabled=true)
```

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

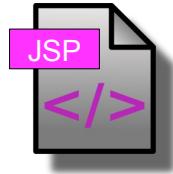
```
public class ItemManager {  
    @Secured("IS_AUTHENTICATED_FULLY")  
    public Item findItem(long itemNumber) {  
        ...  
    }  
}
```

Can also place  
at class level

```
@Secured("ROLE_MEMBER")  
@Secured({"ROLE_MEMBER", "ROLE_USER"})
```



Spring 2.0 syntax, *not limited to roles*. But SpEL *not supported*.



# JSP Security Tag library

- The Spring Security tag library is declared as follows

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

available since Spring Security 2.0

jsp

- Equivalent functionality for other View technologies
  - Velocity, Freemarker, Thymeleaf, JSF ...



# Spring Security's Tag Library

- Display properties of the Authentication object

You are logged in as:

*jsp*

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

- Hide sections of output based on role
  - Implementation on next slide

```
<security:authorize ... >
```

*jsp*

TOP-SECRET INFORMATION

Click [HERE](/admin/deleteAll)

to delete all records.

```
</security:authorize>
```

Content hidden from  
unauthorized users



# JSP Authorization

- Restrict by role explicitly
  - *But puts role information in your web-page*

Required roles

```
<security:authorize access="hasAnyRole('USER','ADMIN')">
```

jsp

- Restrict via URL permissions – preferred
  - *Must specify @EnableWebSecurity (even with Spring Boot)*

```
<security:authorize url="/admin/deleteAll">
```

jsp

```
http.authorizeRequests()  
    .mvcMatchers("/admin/*").hasAnyRole("USER","ADMIN")
```

java



# Practical REST Web Services

Using Spring MVC to build RESTful Web Services

Extending Spring MVC to handle REST

# Objectives

- After completing this lesson, you should be able to:
  - Describe the principles of HTTP REST
  - Use Spring MVC to build RESTful servers
  - Develop RESTful Clients using `RestTemplate`



# Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
- RESTful Clients with the RestTemplate
- Advanced Topics

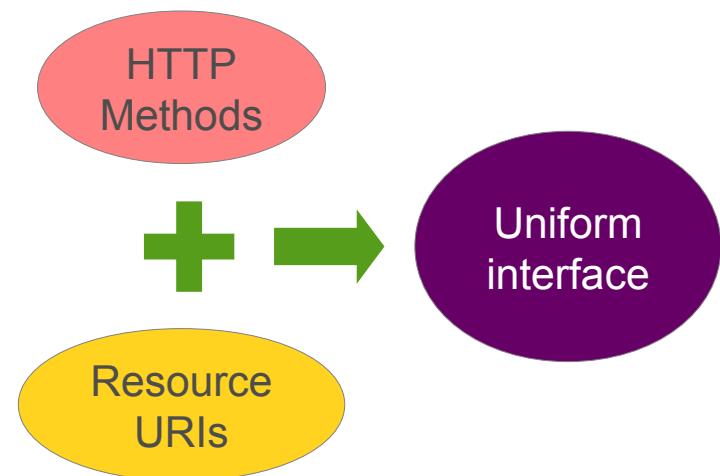
A large, stylized word "REST" in 3D letters. The letters have a vibrant gradient color, transitioning from purple at the top to yellow and orange at the bottom. The perspective is from the bottom left, looking up and to the right.

# REST Introduction

- Open up access to your web-server's functionality
  - Programmatic clients can also connect via HTTP
    - Such as: mobile applications, microservices
  - Browsers: SPA, AJAX
- REST is an *architectural style* that describes best practices to expose web services over HTTP
  - REpresentational S<sub>t</sub>ate T<sub>ransfer</sub>, 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://springbank.io/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



More on HATEOAS in *Advanced Section*

# 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 multiple formats
    - JSON and Atom are popular choices



# REST and Java: JAX-RS



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

# REST and Java: Spring-MVC



- Spring-MVC provides REST support as well
  - 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
- **Spring MVC support for RESTful applications**
- RESTful Clients with the RestTemplate
- Advanced Topics

# Spring-MVC and REST

- Extending Spring MVC to support REST
  - Map requests based on HTTP method
  - Define response status
  - Message Converters
  - Access request and response body data
  - Build valid Location URIs \*

\* For HTTP POST responses

# Topics in this Session

- REST introduction
- **Spring MVC support for RESTful applications**
  - HTTP GET
- RESTful Clients with the RestTemplate
- Advanced Topics

# HTTP GET: Fetch a Resource

- Requirement
  - Respond *only* to GET requests
  - Return requested data in the HTTP Response
  - Determine requested response format

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/json, ...
...
```

```
HTTP/1.1 200 OK
Date: ...
Content-Length: 756
Content-Type: application/json

{
  "order": {
    "id": 123,
    "items": [ ... ], ...
  }
}
```

# Request Mapping Based on HTTP Method

- Map HTTP requests based on method
  - Allows same URL to be mapped to multiple Java methods
- RequestMethod enumerators are
  - GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, TRACE
- Examples:

```
// Get all orders (for current user typically)
@RequestMapping(path="/orders", method=RequestMethod.GET)

// Create a new order
@RequestMapping(path="/orders", method=RequestMethod.POST)
```

# Simpler Mapping Annotations

- Alternative handler mapping shortcuts
  - `@RequestMapping(path="/accounts", method=RequestMethod.GET)`
  - Or `@GetMapping("/accounts")` ;
- Exist for these HTTP methods
  - `@GetMapping`
  - `@PostMapping`
  - `@PutMapping`
  - `@DeleteMapping`
  - `@PatchMapping`

For HEAD, OPTIONS, TRACE use  
RequestMethod enumerators



# Generating Response Data

- The Problem
  - HTTP **GET** needs to return data in response body
    - Typically XML or JSON
  - Prefer to work with Java objects
  - Avoid converting to formats manually

```
HTTP/1.1 200 OK
Date: ...
Content-Length: 756
Content-Type: application/json

{
    "order": {
        "id": 123,
        "items": [ ... ], ...
    }
}
```



# HttpMessageConverter

- Converts HTTP request/response body data
  - XML (using JAXP Source, JAXB2 mapped object, Jackson-Dataformat-XML\*)
  - Jackson JSON\*, GSON\*Feed data\* such as Atom/RSS
  - Google protocol buffers \*
  - Form-based data
  - `Byte[], String, BufferedImage`
- Must enable (or no convertors defined at all!)
  - Automatic with Spring Boot
  - Or use `@EnableWebMvc`
  - Or define explicitly: `WebMvcConfigurer`
    - Allows you to register extra convertors

\* Requires 3rd party libraries on classpath

# @ResponseBody

- Use converters for response data by annotating return data with **@ResponseBody**
- Converter handles rendering a response
  - *No ViewResolver and View involved any more!*

```
@GetMapping(path="/orders/{id}")
public @ResponseBody Order
    getOrder(@PathVariable("id") long id) { ... }
```

If you forget **@ResponseBody**, Spring MVC attempts to find a View (and fails)

# Retrieving a Representation: GET

```
GET /store/orders/123  
Host: shop.spring.io  
Accept: application/json  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 1456  
Content-Type: application/json  
  
{"order": {  
    "id": 123,  
    "items": [ ... ], ...  
}}
```

```
@GetMapping(path="/orders/{id}")  
public @ResponseBody Order getOrder(@PathVariable("id") long id) {  
    return orderService.findOrderById(id);  
}  
  
{@RequestMapping(path="/orders/{id}", method=RequestMethod.GET)}
```

# What Return Format? Accept Header

```
@GetMapping(path="/orders/{id}")
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    return orderService.findOrderById(id);
}
```

```
GET /store/orders/123
Host: shop.spring.io
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: shop.spring.io
Accept: application/json
...
```

```
HTTP/1.1 200 OK
Date: ...
Content-Length: 756
Content-Type: application/json
{
    "order": {"id": 123,
              "items": [ ... ], ... }
}
```

# @RestController Simplification

```
@Controller
public class OrderController {
    @GetMapping(path="/orders/{id}")
    public @ResponseBody Order getOrder(@PathVariable("id") long id) {
        return orderService.findOrderById(id);
    }
    ...
}

@RestController
public class OrderController {
    @GetMapping(path="/orders/{id}")
    public Order getOrder(@PathVariable("id") long id) {
        return orderService.findOrderById(id);
    }
    ...
}
```

No need for `@ResponseBody` on GET methods

# HttpEntity and ResponseEntity

- To build responses explicitly
  - Set headers, control content returned
  - Use **HttpEntity** or **ResponseEntity**

```
// Want to return a String as the response-body
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.TEXT_PLAIN);
HttpEntity<String> entity =
    new HttpEntity<String>("Hello Spring", headers);

// ResponseEntity (since Spring 4.1) supports a "fluent" API
ResponseEntity<String> response =
    ResponseEntity.ok()
        .contentType(MediaType.TEXT_PLAIN)
        .body("Hello Spring");
```

# Setting Response Data

- Can use **HttpEntity** to generate a Response
  - Avoids use of **HttpServletResponse** (easier to test)

```
@GetMapping(path="/orders/{id}")
public HttpEntity<Order> getOrder(long id) {
    Order order = orderService.find(id);

    return ResponseEntity
        .ok()
        .header("Last-Modified", order.lastUpdated())
        .body(order);
}
```

The diagram illustrates the construction of a `ResponseEntity` object from annotations. It shows the flow from the annotated code to the resulting response components:

- No need for `@ResponseBody`**: An annotation on the `getOrder` method.
- HTTP Status 200 OK**: The status code returned.
- Set extra or custom header**: The `.header` method call.
- Response body**: The `.body` method call.

Annotations are highlighted in blue, and their corresponding code segments are highlighted in yellow.

# Topics in this Session

- REST introduction
- **Spring MVC support for RESTful applications**
  - HTTP GET
  - **HTTP PUT**
- RESTful Clients with the RestTemplate
- Advanced Topics

# HTTP PUT: Update a Resource

- Requirement
  - Respond *only* to PUT requests
  - Access data in the HTTP Request
  - Return empty response, status 204

Successful update –  
*nothing* to return

```
PUT /store/orders/123/items/abc
Host: www.mybank.com
Content-Type: application/json

{
  "order": {
    "id": 123, "items": [ ... ], ...
  }
}
```

```
HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
...
```



# HTTP Status Code Support

- Web apps just use a handful of status codes
  - Success: 200 OK
  - Redirect: 30x for Redirects
  - Client Error: 404 Not Found
  - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many additional codes
  - Created Successfully: 201
  - HTTP method not supported: 405
  - Cannot generate requested response body format: 406
  - Request body not supported: 415



For a full list: [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

# @ResponseStatus

- To return a status code *other* than 200
  - Use **HttpStatus** enumerator
- **Note:** @ResponseStatus on **void** methods
  - No longer want to return a view name - *no View at all!*
  - Method returns a response with empty body (*no-content*)

```
@PutMapping("/orders/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrder(...) {
    // Update order
}
```



Can also set error response codes – see Advanced section

# @RequestBody

- Use message converters for incoming request data
  - Correct converter chosen automatically
    - Based on content type of request
  - `updatedOrder` could be mapped from XML (with JAXB2) or from JSON (with Jackson)
    - Annotate Order class (if need be) for JAXB/Jackson to work

```
@PutMapping(path="/orders/{id}")
@ResponseStatus(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);
}
```

# Updating Existing Resource: PUT

```
PUT /store/orders/123/items/abc  
Host: shop.spring.io  
Content-Type: application/json  
  
{  
  "order": {  
    "id": 123, "items": [ ... ], ... }  
}
```

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

```
@PutMapping(path="/orders/{orderId}/items/{itemId}")  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
public void updateItem(@PathVariable("orderId") long orderId,  
                      @PathVariable("itemId") String itemId,  
                      @RequestBody Item item) {  
  
    orderService.findOrderById(orderId)  
        .updateItem(itemId, item);  
}  
@RequestMapping(path="/orders/...", method=RequestMethod.PUT)
```

# Topics in this Session

- REST introduction
- **Spring MVC support for RESTful applications**
  - HTTP GET
  - HTTP PUT
  - HTTP POST
- RESTful Clients with the RestTemplate
- Advanced Topics

# HTTP POST: Create a new Resource

- Requirement
  - Respond *only* to POST requests
  - Access data in the HTTP Request
  - Return “created”, status 201
  - Generate *Location* header for newly created resource

```
POST /store/orders/123/items
Host: shop.spring.io
Content-Type: application/json

{
  "order": {
    "id": 123,
    "items": [ ... ], ...
  }
}
```

```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://shop.spring.io/
          store/orders/123/items/abc
```

# Creating a new Resource: POST (2)

- We can already implement most of this requirement
  - But how do we return the new Item location?

```
@PostMapping(path="/orders/{id}/items")
public ??? createItem
    (@PathVariable("id") long id, @RequestBody Item newItem)
{
    // Add the new item to the order
    orderService.findOrderById(id).addItem(newItem);

    return ???;
}

@RequestMapping(path="/orders/...", method=RequestMethod.POST)
```

# Building URLs



- An HTTP POST typically returns location of newly created resource in the response header
- How to create a URI?
  - **UriComponentsBuilder**
    - Allows explicit creation of URI
    - *But* uses hard-coded URLs
  - **ServletUriComponentsBuilder**
    - Provides access to the URL that invoked the current controller method

# Building URIs: UriComponentsBuilder

- Support for building URIs from URI template strings
  - Escapes illegal characters – such as **%20** for a space

**BUT:** Use of hard-coded URL *not* recommended

```
String templateUrl =  
    "http://store.spring.io/orders/{orderId}/items/{itemId}";
```

```
URI location = UriComponentsBuilder.  
    fromHttpUrl(templateUrl).  
    buildAndExpand("123456", "item A").  
    toUri();
```

```
return ResponseEntity.created(location).build();
```

```
// http://store.spring.io/orders/123456/items/item%20A
```

Convenient way to  
build POST response

# Better: ServletUriComponentsBuilder

- Use in a Controller method
  - Avoids hard-coding URL

Framework puts request URL in current thread – which builder can access

```
// Must be in a Controller method  
// Example: POST /orders/{id}/items
```

```
URI location = ServletUriComponentsBuilder  
    .fromCurrentRequestUri() ←  
    .path("{itemId}")  
    .buildAndExpand("item A")  
    .toUri(); ;  
  
return ResponseEntity.created(location).build();  
  
// http://.../items/item%20A
```



# Creating a new Resource: POST

```
@PostMapping(path="/orders/{id}/items")
public ResponseEntity<Void> createItem
    (@PathVariable("id") long id, @RequestBody Item newItem) {
    // Add the new item to the order
    orderService.findOrderById(id).addItem(newItem);

    // Build the location URI of the new item
    URI location = ServletUriComponentsBuilder
        .fromCurrentRequestUri()
        .path("/{itemId}")
        .buildAndExpand(newItem.getId())
        .toUri();

    // Explicitly create a 201 Created response
    return ResponseEntity.created(location).build();
}
```

@ResponseStatus  
not needed

Assume this  
call also set  
an item-id

@RequestMapping(path="/orders/...", method=RequestMethod.POST)



# Lesson Roadmap

- REST and Java
- **Spring MVC support for RESTful applications**
  - HTTP GET
  - HTTP PUT
  - HTTP POST
  - **HTTP DELETE**
- RESTful Clients with the RestTemplate
- Advanced Topics

# HTTP DELETE: Delete a new Resource

- Requirement
  - Respond *only* to DELETE requests
  - Return empty response, status 204

```
DELETE /store/orders/123/items/abc  
Host: shop.spring.io  
Content-Length: 0  
...
```

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

# Deleting a Resource: DELETE

```
DELETE /store/orders/123/items/abc  
Host: shop.spring.io  
...
```

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0
```

```
@DeleteMapping(path="/orders/{orderId}/items/{itemId}")  
@ResponseStatus(HttpStatus.NO_CONTENT) // 204  
public void deleteItem(@PathVariable("orderId") long orderId,  
                      @PathVariable("itemId") String itemId) {  
    orderService.findOrderById(orderId).deleteItem(itemId);  
}
```

```
@RequestMapping(path="/orders/...", method=RequestMethod.DELETE)
```

# Putting it all Together

- Many new concepts
  - `@ResponseStatus`
  - HTTP Message Converters
  - `@RequestBody`, `@ResponseBody`
  - `@RestController`
  - `HttpEntity`, `ResponseEntity`
  - `ServletUriComponentsBuilder`



# Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
- **RESTful Clients with the RestTemplate**
- Advanced Topics

# RestTemplate

- Provides access to RESTful services
  - Supports all the HTTP methods

HTTP Method	RestTemplate Method
DELETE	delete(String url, Object... urlVariables)
GET	getForObject(String url, Class<T> responseType, Object... urlVariables)
HEAD	headForHeaders(String url, Object... urlVariables)
OPTIONS	optionsForAllow(String url, Object... urlVariables)
POST	postForLocation(String url, Object request, Object... urlVariables) postForObject(String url, Object request, Class<T> responseType, Object... uriVariables)
PUT	put(String url, Object request, Object... urlVariables)

# Defining a RestTemplate

- Just call constructor in your code
  - Sets up default *HttpMessageConverters* internally
    - Same as on the server, depending on classpath

```
RestTemplate template = new RestTemplate();
```

# RestTemplate Usage Examples

```
RestTemplate template = new RestTemplate();
String uri = "http://example.com/store/orders/{id}/items";

// GET all order items for an existing order with ID 1:
OrderItem[] items =
    template.getForObject(uri, OrderItem[].class, "1");

// POST to create a new item
OrderItem item = // create item object
URI itemLocation = template.postForLocation(uri, item, "1");

// PUT to update the item
item.setAmount(2);
template.put(itemLocation, item);

// DELETE to remove that item again
template.delete(itemLocation);
```

{id} = 1

{id} = 1

# Using ResponseEntity and ResponseEntity

- Access response headers and body

```
ResponseEntity<String> response =  
    restTemplate.getForEntity(itemUrl, String.class);  
assert(response.getStatusCode().equals(HttpStatus.OK));  
String content = response.getBody();
```

Body returned as String, but  
object mapping still an option

- Setup your own request

```
HttpEntity<OrderItem> request = new HttpEntity<>(item);  
request.getHeaders().add(HttpHeaders.AUTHORIZATION,  
    "Basic " + getBase64EncodedLogPass());  
ResponseEntity<Void> response = restTemplate  
    .exchange(itemUrl, HttpMethod.POST, request, Void.class);  
assert(response.getStatusCode().equals(HttpStatus.CREATED));
```

# Summary



- REST is an architectural style that can be applied to HTTP-based applications
  - Useful for supporting diverse clients and building highly scalable systems
- Spring-MVC adds REST support using a familiar programming model (but *without* Views)
  - `@ResponseStatus`, `@RequestBody`, `@ResponseBody`
  - `HttpEntity`,  `ResponseEntity`,  `UriComponentsBuilder`
  - HTTP Message Converters
- Clients use *RestTemplate* to access RESTful servers

# Lab

Restful applications with Spring MVC

**Coming Up:** More on Spring MVC REST  
Introduction to Spring HATEOAS



# Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
- RESTful clients with the RestTemplate
- **Advanced Topics**
  - More on Spring REST
  - Spring HATEOAS

# @ResponseStatus & Exceptions

- Can also annotate exception classes with this
  - Given status code used when an unhandled exception is thrown from *any* controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
public class OrderNotFoundException extends RuntimeException {
    ...
}

@GetMapping(value="/orders/{id}")
public String showOrder(@PathVariable("id") long id, Model model) {
    Order order = orderService.findOrderById(id);
    if (order == null) throw new OrderNotFoundException(id);
    model.addAttribute(order);
    return "orderDetail";
}
```

**NOTE:** this is *not* a RESTful method,  
it returns a view-name.

# @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
}
```



Spring MVC offers several ways to handle exceptions, for more details see <http://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc>

# Mixing Views and Annotations - 1

- REST methods do not return HTML, PDF, ...
  - No message converter for these formats
  - Views better for presentation-rich representations
- How to distinguish between representations?
  - Or a RESTful POST from a HTML form submission
- Use *produces* and *consumes* attributes

```
@GetMapping(value="/orders/{id}", produces = {"application/json"})  
  
@PostMapping(value="/orders/{id}", consumes = {"application/json"})
```

# Mixing Views and Annotations – 2

- Need *two* methods on controller for *same URL*
  - One uses a converter, the other a View
  - Identify using *produces* attribute
- Recommendation
  - Mark RESTful method with *produces*
    - To avoid returning XML to normal browser request
  - Call RESTful method from View method
    - Implement all data-access logic *once* in RESTful method

# Mixing Views and Annotations - 3

- Recommendation

RESTful Method

```
@GetMapping(path="/orders/{id}",
    produces = {"application/json", "application/xml"})
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    // Access data here ...
    return orderService.findOrderById(id);
}
```

```
@GetMapping(path="/orders/{id}")
public String getOrder(Model model, @PathVariable("id") long id) {
    // Invoke RESTful method, use result to populate model
    model.addAttribute(getOrder(id));
    return "orderDetails"; // View name
}
```

*View method calls  
RESTful method*

View Method

# HttpMethodFilter

- HTML forms do not support PUT or DELETE
  - Not even in HTML 5
- So use a POST
  - Put PUT or DELETE in a *hidden* form field
- Deploy a special filter to intercept the message
  - Restores the HTTP method you wanted to send
  - Request looks like a PUT or a DELETE to any Controller



See [\*HttpMethodFilter\*](#) in online documentation

# Topics in this Session

- REST introduction
- Spring MVC support for RESTful applications
  - Request/Response Processing
  - Using MessageConverters
  - Putting it all together
- RESTful clients with the RestTemplate
- **Advanced Topics**
  - More on Spring REST
  - **Spring HATEOAS**

# HATEOAS - Concepts

- REST clients need *no* prior knowledge about how to interact with a particular application/server
  - SOAP web-services need a WSDL
  - SOA processes require a fixed interface defined using interface description language (IDL)
- Clients interact entirely through hypermedia
  - Provided dynamically by application servers
- Serves to *decouple* client and server
  - Allows the server to evolve functionality independently
  - Unique compared to other architectures

# HATEOAS Account Example

```
<account>
  <account-number>12345</account-number>
  <balance currency="usd">100.00</balance>
  <link rel="self" href="/accounts/12345" />
  <link rel="deposit" href="/accounts/12345/deposits" />
  <link rel="withdraw" href="/accounts/12345/withdraws" />
  <link rel="transfer" href="/accounts/12345/transfers" />
  <link rel="close" href="/accounts/12345/close" />
</account>
```

**Spring HATEOAS** provides an API for generating these links in MVC Controller responses

```
<account>
  <account-number>12345</account-number>
  <balance currency="usd">-25.00</balance>
  <link rel="self" href="/accounts/12345" />
  <link rel="deposit" href="/accounts/12345/deposits" />
</account>
```

Note: links change as state changes



There is no standard for links yet. This example uses the link style from the *Hypertext Application Language (HAL)*, one possible representation

# Managing Links

- Use **Link** class
  - Holds an href and a rel (relationship)
  - Self implies the current resource
  - Link builder derives URL from Controller mappings

```
// A link can be built with a relationship name
// Use withSelfRel() for a self link
Link link = ControllerLinkBuilder.linkTo(AccountController.class)
    .slash(accountId).slash("transfer").withRel("transfer");

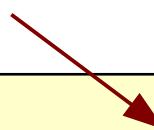
link.getRel();      // => transfer
link.getHref();    // => http://.../account/12345/transfer
```

# Converting to a Resource

- Wrap return value of REST method in a **Resource**
  - Converted by **@ResponseBody** to XML/JSON with links
    - Only HAL supported currently

```
@Controller
@EnableHypermediaSupport(type=HypermediaType.HAL)
public class OrderController {

    @GetMapping(value="/orders/{id}")
    public @ResponseBody Resource<Order>
        getOrder(@PathVariable("id") long id) {
        Links[] = ...; // Some links (see previous slide)
        return new Resource<Order>
            (orderService.findOrderById(id), links);
    }
}
```



# Spring HATEOAS



- Spring Data sub-project for REST
  - For generating links in RESTful responses
  - Supports ATOM (newsfeed XML) and HAL (Hypertext Application Language) links
  - Many other features besides examples shown here
- For more information see
  - <http://projects.spring.io/spring-hateoas/>
  - <http://spring.io/guides/gs/rest-hateoas/>

## *Optional Section*

# Microservices with Spring

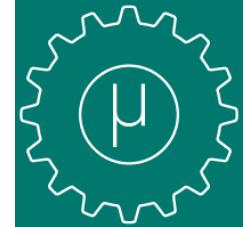
## Building Cloud Native Applications

Introduction to Spring Cloud

# Objectives

- After completing this lesson, you should be able to:
  - Describe a Microservices Architecture and explain the Pros and Cons
  - Explain the Challenges of Managing Microservices
  - Describe Spring's Support for Microservice Applications
  - Build a Simple Microservice System





# Roadmap

- **What is Microservices Architecture?**
- Pros and Cons of Microservices
- Managing Microservices
- Tooling: Spring, Spring Cloud, Netflix
- Building a Simple Microservice System



# Introduction

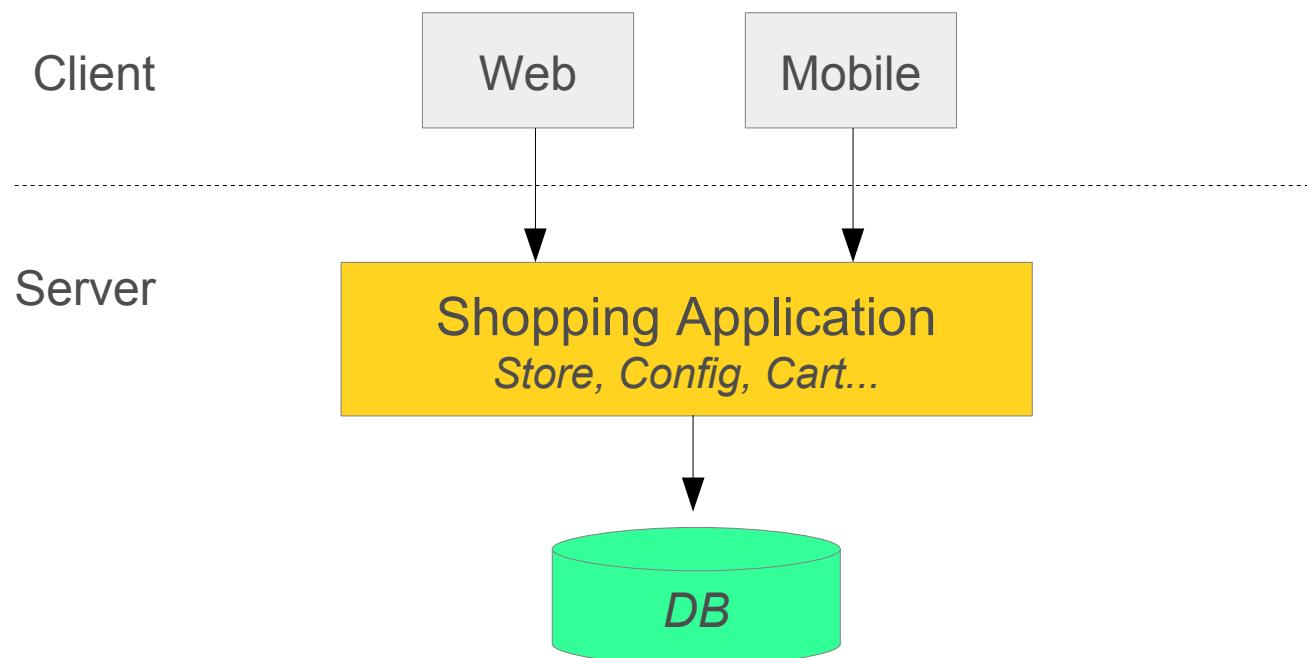
- “Microservices” is not a new word
  - Term coined in 2005 by Dr Peter Rodgers
    - Then called “*micro web services*” and based on SOAP
  - Term started to become popular since 2010
    - Proposed by a group of architects in Venice during 2011
    - Used in 2012 in a presentation from James Lewis
  - Adrian Cockcroft (Netflix) describes this approach
    - "Fine grained Service Oriented Architecture (SOA)"
    - "Loosely coupled SOA with bounded contexts"



See also: <http://martinfowler.com/articles/microservices.html>

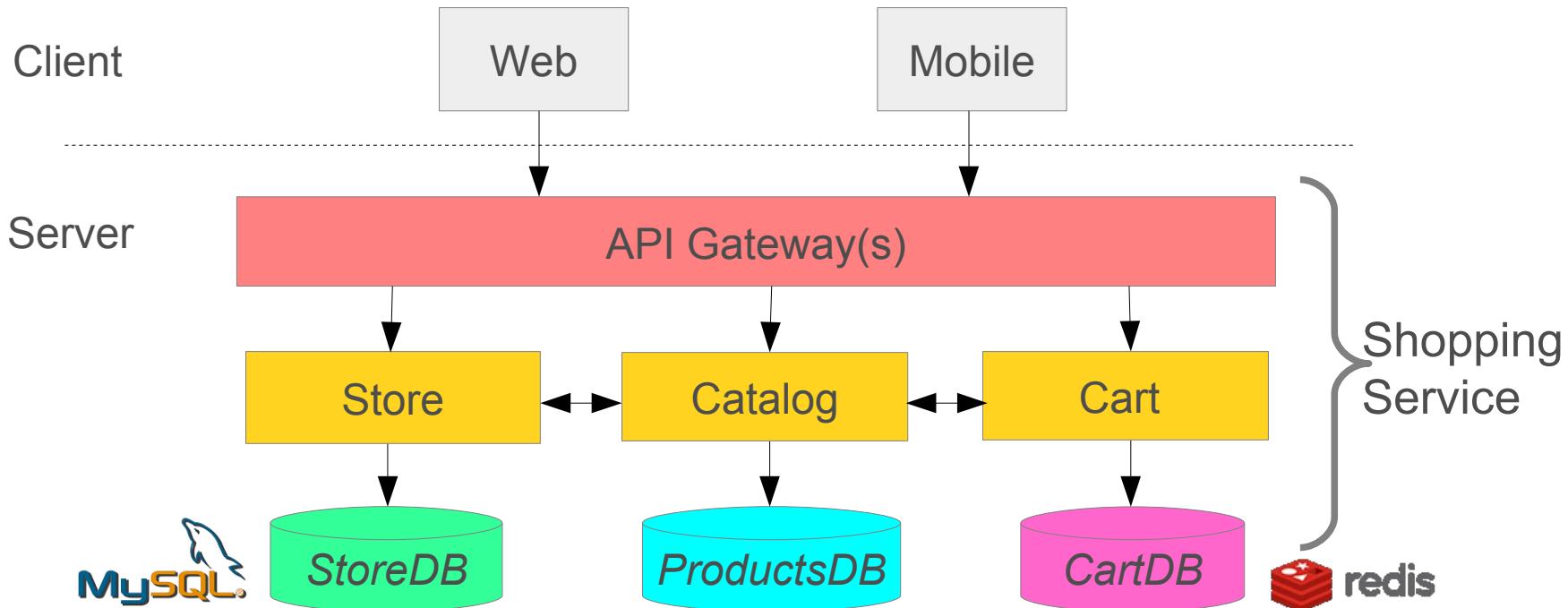
# Without Microservices

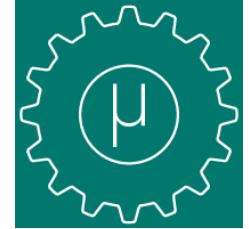
- Using a “monolith” architecture
  - All-in-One application



# With Microservices

- Main application divided into a set of sub-applications
  - Called microservices





# Roadmap

- What is Microservices Architecture?
- **Pros and Cons of Microservices**
- Managing Microservices
- Tooling: Spring, Spring Cloud, Netflix
- Building a Simple Microservice System



# Core Spring Concepts

## Applied to Application Architecture



- Spring enables *separation-of-concerns*
  - *Loose Coupling*: Effect of change is isolated
  - *Tight Cohesion*: Code performs a single well-defined task
- Microservices exhibit the same strengths
  - *Loose Coupling*
    - Applications are built from collaborating services (processes)
    - Can change independently so long as *protocols unchanged*
  - *Tight Cohesion*
    - An application (service) that deals with a *single* view of data
    - Also known as “Bounded Contexts” (*Domain-Driven Design*)



# Microservice Benefits

- Smaller code base is easy to maintain
- Easy to scale
  - Scale individual component
- Technology diversity
  - Mix libraries, frameworks, data storage, languages
- Fault Isolation
  - Component failure should not bring whole system down
- Better support for smaller, parallel teams
- Independent deployment

# Microservice Challenges



- Difficult to achieve strong consistency across services
  - ACID transactions *do not* span multiple processes
  - Eventual consistency, Compensating transactions
- Distributed system
  - Harder to debug/trace
  - Greater need for end-to-end testing
  - Expect, test for and handle the failure of any process
  - More components to maintain: redundancy, HA
- Typically requires “cultural” change (*Dev Ops*)
  - How applications are developed and deployed
  - Dev and Ops working *together*, even on same team!

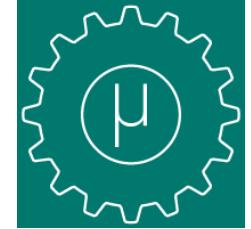


CLOUD FOUNDRY

# Use a Platform to Support This

- Platforms\* like *Pivotal Cloud Foundry* aid deployment
  - Easily run, scale, monitor and recover *multiple* processes
  - Run up a complete dev system for end-to-end testing
- Support for
  - Continuous Deployment
  - Rolling upgrades of new versions of code
    - Also termed: Blue/Green or Canary rollout
    - Quick rollback in case of defects
  - Running multiple versions of same service at same time
    - Makes migration easier for downstream projects

\*Platform as a Service (PaaS)



# Roadmap

- What is Microservices Architecture?
- Pros and Cons of Microservices
- **Developing Microservices**
- Tooling: Spring, Spring Cloud, Netflix
- Building a Simple Microservice System



# Microservice Infrastructure

- Multiple processes working together
- Issues that now arise:

– How do they find each other? Service Discovery

– How do we decide which instance to use? Client-side Load Balancing

– What happens if a microservice is not responding Fault Tolerance

– How do we control access? OAuth, ...

– How do they communicate? Messaging

– To just name a few! REST

We *only* cover  
these today



# Roadmap

- What is Microservices Architecture?
- Pros and Cons of Microservices
- Managing Microservices
- **Tooling: Spring, Spring Cloud, Netflix**
- Building a Simple Microservice System



# Microservices made-easy by Spring

- Setup a new service using Spring Boot
- Expose resources via a **RestController**
- Consume remote services using **RestController**
- Leverage capabilities from *Spring Cloud Project*



# What is Spring Cloud?

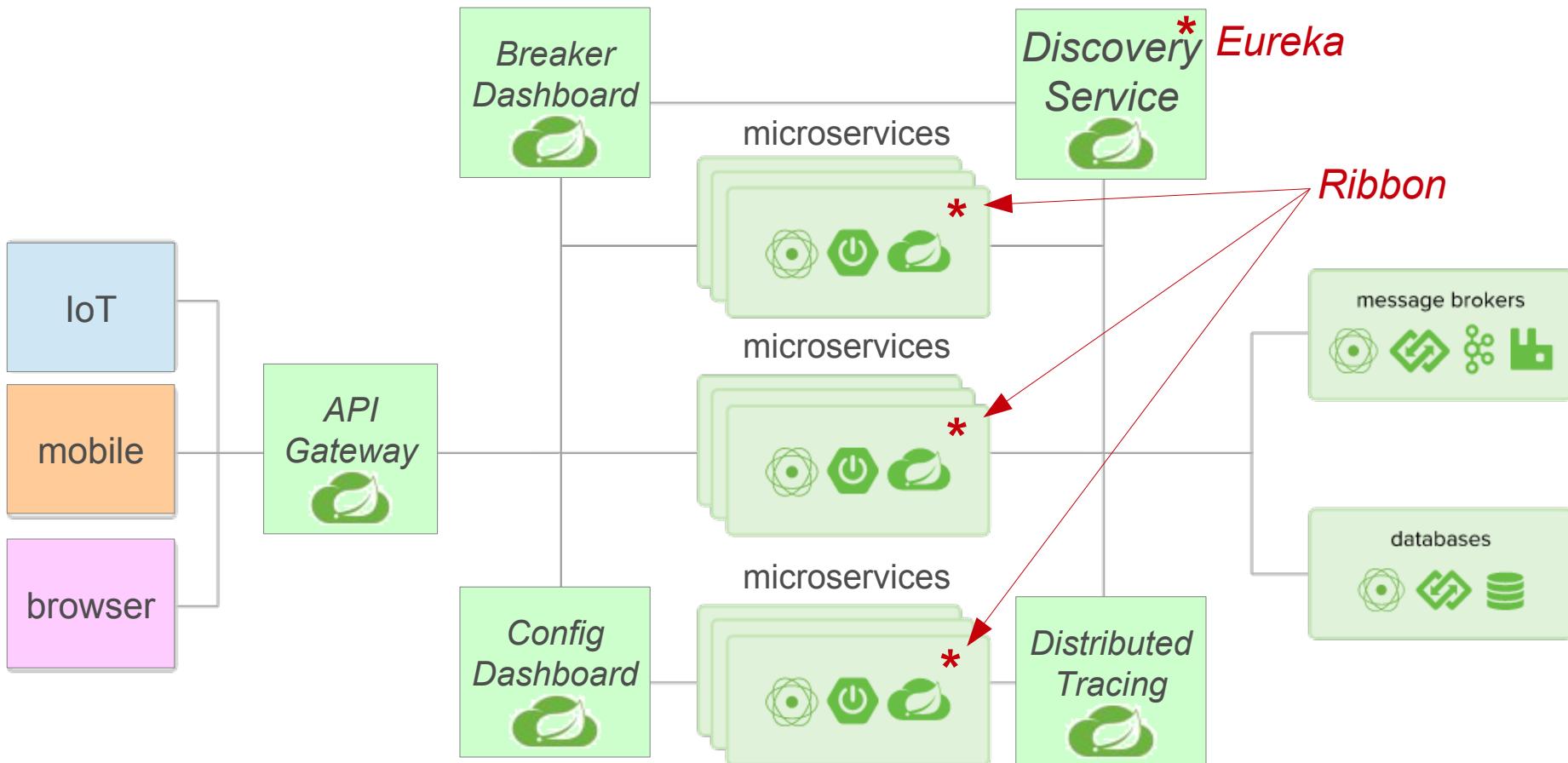


- Building blocks for Cloud and Microservice applications
  - Microservices Infrastructure
    - Provides useful services such as Service Discovery, Configuration Server and Monitoring
    - Several based on other Open Source projects
      - Netflix OSS, HashiCorp's Consul, Apache Zookeeper ...
  - Platform Support
    - Access platform-specific information and services
      - Available for Cloud Foundry, AWS and Heroku
  - Uses Spring Boot style starters
    - Requires Spring Boot to work



# Spring Cloud Ecosystem

\* Today we only have time to look at Eureka & Ribbon



Spring Cloud is at <http://projects.spring.io/spring-cloud/>

# Spring Cloud Usage Examples

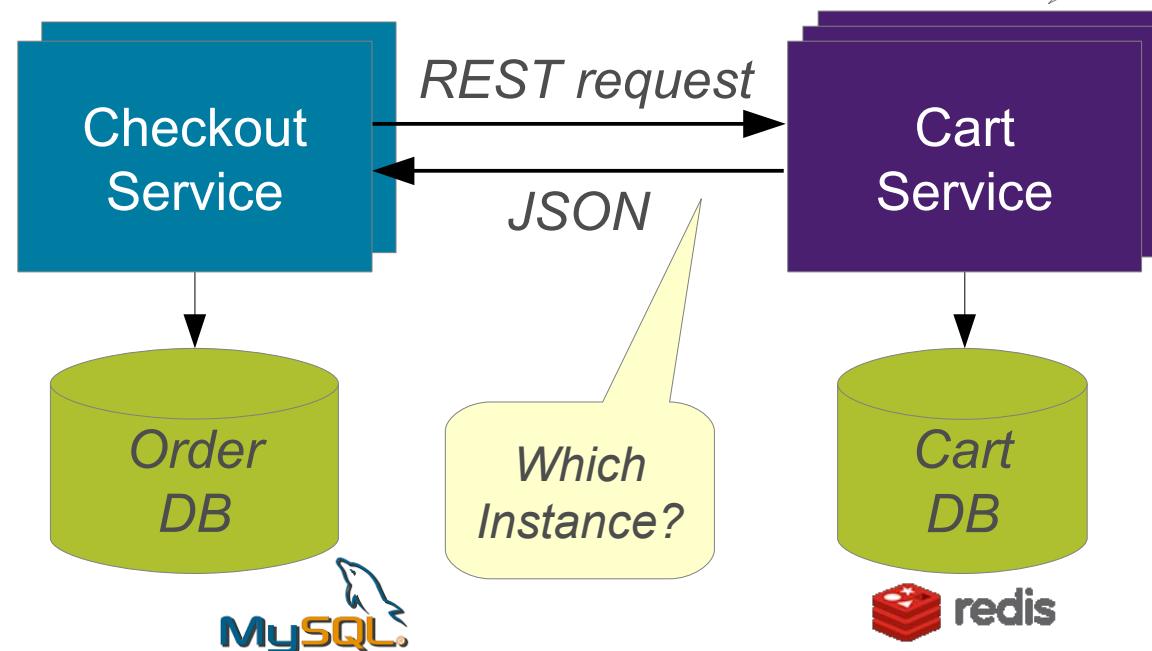


- There are *many* use-cases supported by Spring Cloud
  - Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security, Data Ingestion, Cluster Recovery
- Today we concentrate on *microservices* support
  - **Service Discovery**
    - How do the services find each other?
  - **Client-side Load Balancing**
    - Each service typically deployed as multiple instances
      - For fault-tolerance and load-sharing
    - How do we decide which service *instance* to use?

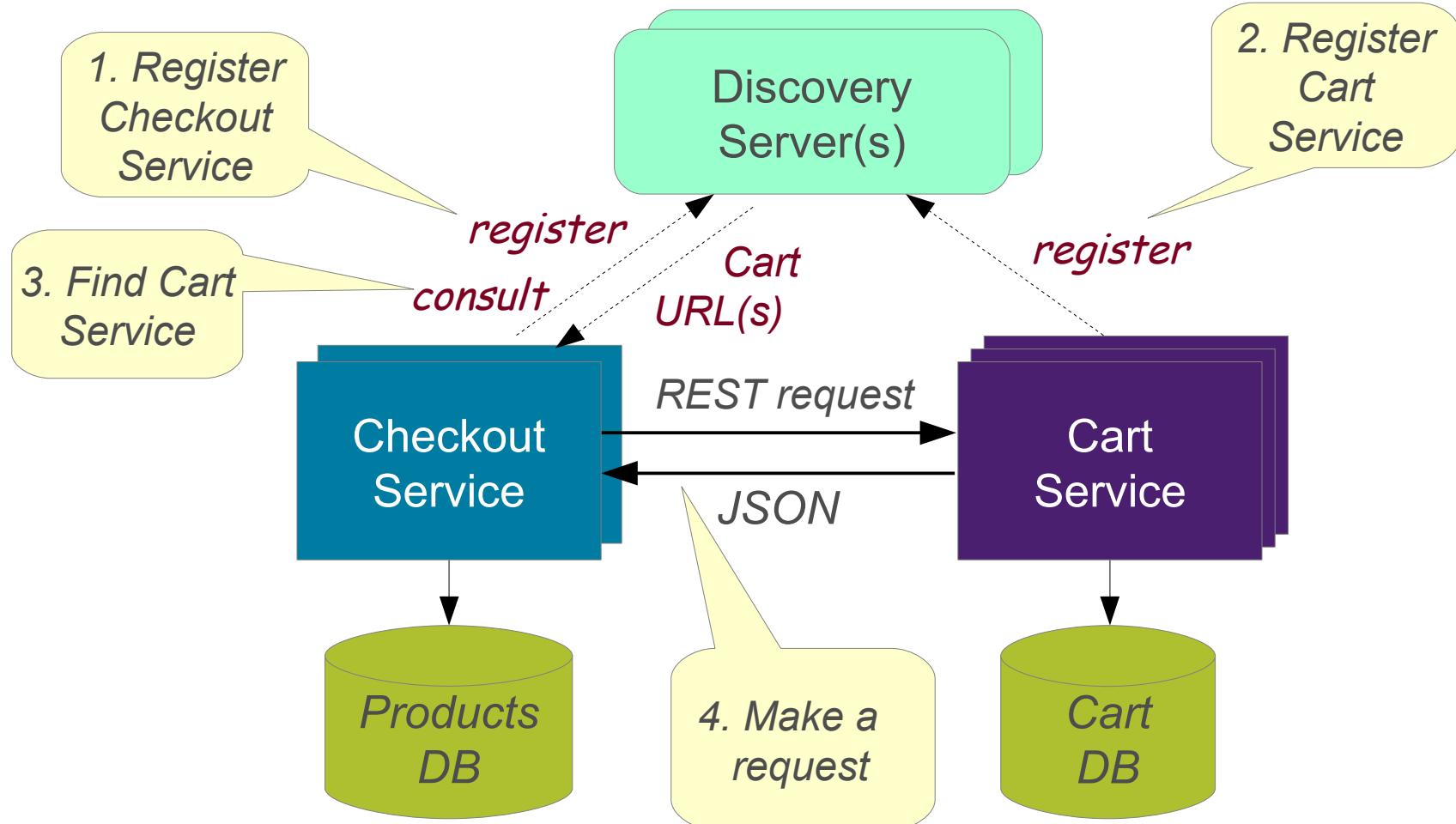
# Why We Need Service Discovery – 1

- Various protocols may be used
  - But how do the two services find each other?
  - What happens if we run multiple instances?

*Multiple instances  
for throughput  
and resilience*



# Why We Need Service Discovery – 2



# Implementing Service Discovery



- Spring Cloud supports several
  - Netflix Eureka (and Ribbon)
  - HashiCorp Consul
  - Apache Zookeeper
- Spring Cloud makes it easy
  - To utilize either of those servers
  - Hiding their internal complexity



See also: <http://spring.io/blog/2015/07/14/microservices-with-spring>

# Implementing Client-Side Load Balancing

- Discovery server may return the location of *multiple* instances
  - *Recall:* multiple instances for resilience *and* load-sharing
  - Client needs to pick one
- We will use Netflix Ribbon
  - Provides several algorithms for client-side load-balancing
- Spring provides a “*smart*” RestTemplate
  - Service-discovery and load-balancing built-in
  - `@LoadBalanced`



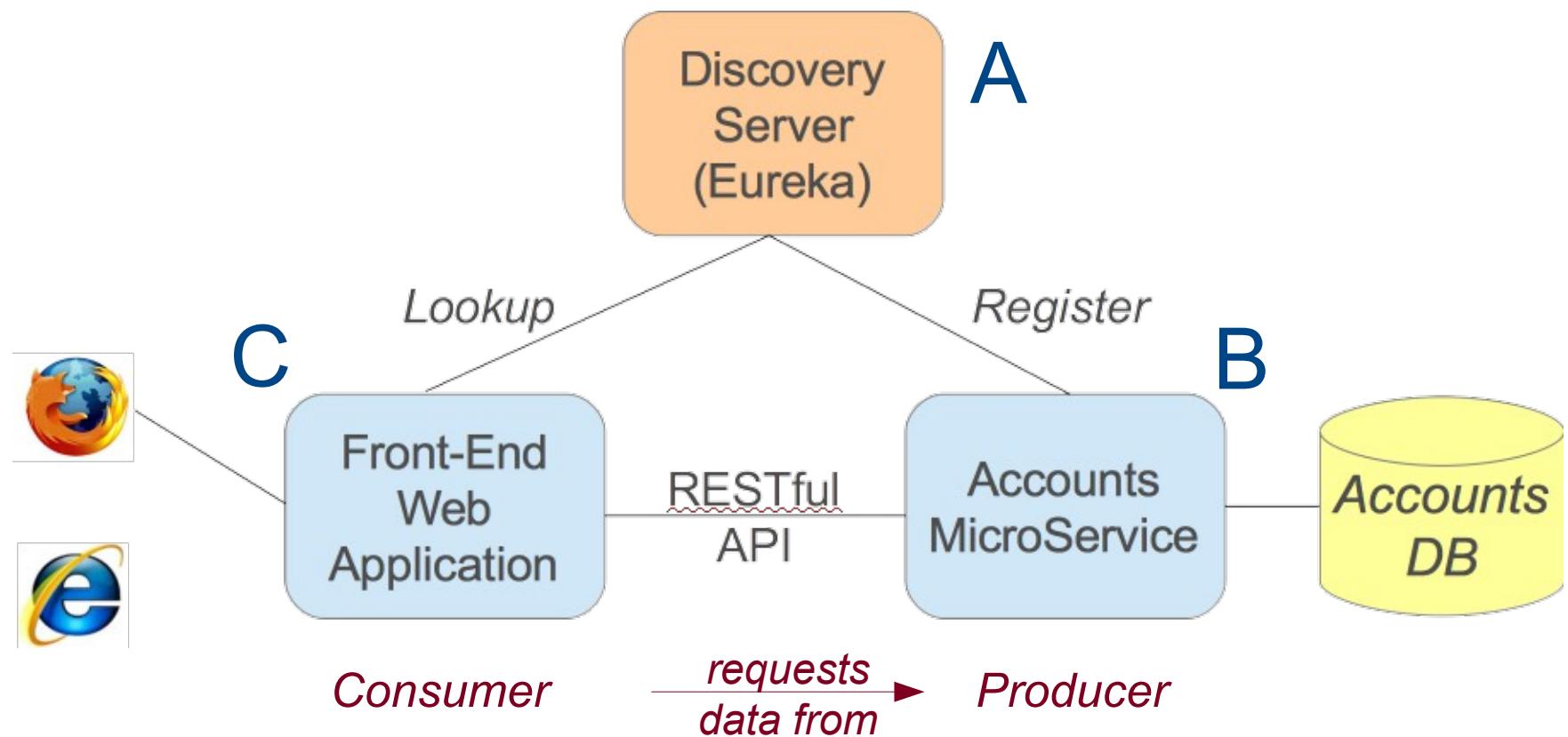


# Roadmap

- What is Microservices Architecture?
- Pros and Cons of Microservices
- Managing Microservices
- Tooling: Spring, Spring Cloud, Netflix
- **Building a Simple Microservice System**



# Our Simple Microservice System



# Building our Simple Microservice System

- A) Run a Discovery Service
  - We will see how to create a Eureka Discovery Service
- B) Run a Microservice (the Producer)
  - Ensure it registers itself with the Discovery Service
  - Registers its *logical* service name with A
- C) How do Microservice *Consumers* find service B?
  - Discovery client using a “smart” RestTemplate
    - Spring performs service lookup for you
    - Uses *logical* service names in URLs

# Maven Dependencies

```
<parent>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-parent</artifactId>
  <version>Dalston.SR4</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  </dependency>
</dependencies>
```

**Parent**

“Release train” = a consolidated set of releases

**Spring Cloud**

**pom.xml**

Access Eureka registry server



Dependencies for A, B & C. Spring Cloud is based on Spring Boot

# (A) Eureka Server using Spring Cloud

- All you need to implement your own registry service!

```
@SpringBootApplication  
@EnableEurekaServer  
public class EurekaApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(EurekaApplication.class, args);  
    }  
}
```

main.java

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>
```

pom.xml



Pivotal™

# (B) Accounts Producer Microservice

## Performs Service Registration

- Microservice declares itself as an available service
  - Using `@EnableDiscoveryClient`
  - Registers using its *application name*

```
@SpringBootApplication
@EnableDiscoveryClient
public class AccountsApplication {
    public static void main(String[] args) {
        SpringApplication.run(AccountsApplication.class, args);
    }
}
spring:
  application:
    name: accounts-microservice
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Service name

Eureka Server URL

# (C) Consumer Service – Step 1

Enable our consumer to find the producer

- Same annotation also allows service *lookup*

```
@SpringBootApplication
@EnableDiscoveryClient
public class FrontEndApplication {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    // Will use this template to access the microservice
    // Spring will enhance this to do service discovery
    @Bean @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Same annotation as (B) –  
allows us to query Discovery  
Server to *find* microservices

# (C) Consumer Service – Step 2

Let our consumer use the producer

```
@Service
public class RemoteAccountManager implements AccountService {

    // Spring injects the “smart” service-aware template
    // defined on previous slide
    // It performs a load-balanced lookup (see next slide)
    @Autowired
    @LoadBalanced
    RestTemplate restTemplate;

    public Account findAccount(String id) {
        // Fetch data
        return restTemplate.getForObject(
            "http://accounts-microservice/accounts/{id}",
            Account.class, id);
    }
}
```

**Service name**

# Load Balanced RestTemplate

- Create using `@LoadBalanced` – an `@Qualifier`
  - Spring enhances it to do service lookup & load-balancing

```
@Bean @LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

- Inject using same qualifier
  - If there are multiple RestTemplates you get the right one
  - Can be used to access multiple microservices

```
@Autowired
@LoadBalanced
RestTemplate restTemplate;
```

# Load Balancing with Ribbon

- Our “smart” RestTemplate automatically integrates *two* Netflix utilities
  - “Eureka” service-discovery
  - “Ribbon” client-side load-balancer
- End result
  - Eureka returns the URL of all available instances
  - Ribbon determines the best available service to use
- Just inject the load-balanced **RestTemplate**
  - Automatic lookup by *logical* service-name

# Spring Cloud Resources

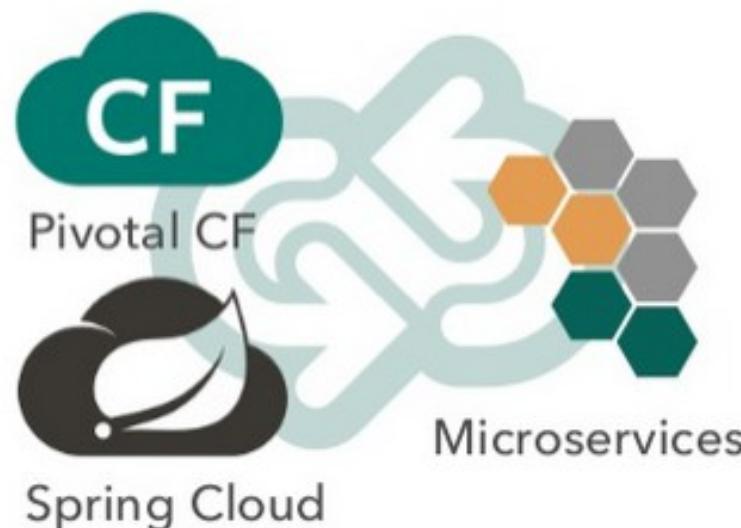


- We have only covered a few Spring Cloud features
  - Project Home page
    - <http://projects.spring.io/spring-cloud>
  - Matt Stine's *free* book on Cloud Native Architectures
    - <https://pivotal.io/platform/migrating-to-cloud-native-application-architectures-ebook>
  - Spring Blog article
    - <https://spring.io/blog/2015/07/14/microservices-with-spring>

Consider taking our ***Spring Cloud Services*** course  
See: <http://pivotal.io/academy>

# Summary

- After completing this lesson, you should have learned:
  - What is a Microservice Architecture?
  - Advantages and Challenges of Microservices
  - A little bit about Spring Cloud projects



# Lab

Building a simple Microservices system

**Coming Up:** Legacy dependency names

# Note: Dependencies have Changed

- Older applications (before *Finchley* build-train)
  - *Warning:* many online examples use old names

```
<!-- Client Dependency -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>

<!-- Server Dependency -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

## *Optional Section*

# Reactive Spring Applications

Introducing Reactive Programming

Spring's reactive support

# Objectives

After completing this lesson, you should be able to:

- Describe the basic concepts of *Reactive Programming*
- Write a “reactive” Spring application



# Roadmap

- **What is Reactive Programming?**
- Reactive Stream Implementations
- Reactive Spring Features

# Why Reactive?

- New applications and environments
  - Distributed (Cloud, PaaS)
  - Multi-process (microservices)
- Challenges
  - Latency inevitable
  - Redundancy and recovery
  - Scale out, not up
- Imperative (traditional) logic becomes *very* complicated



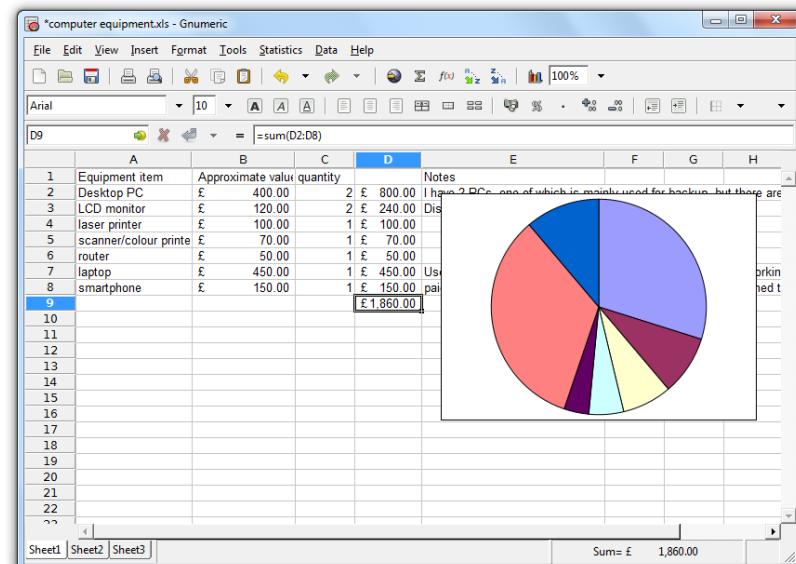
# Reactive Programming

- Non-blocking applications
  - Asynchronous, event-driven
  - Scale using a *minimal* number of threads
  - Flow control (*backpressure*)
  - “*Programming with asynchronous data streams*”\*
- Implications
  - Major shift from *imperative* style logic to a *declarative pipeline* of asynchronous logic
  - Intelligent routing and consumption of events
  - Comparable to **CompletableFuture** in Java 8 and composing follow-up actions via lambda expressions

\* André Staltz – see references at end

# Familiar Example 1: Spreadsheets

- Spreadsheets
  - Formula cells in a spreadsheet automatically “react” to changes in the cells used by the formula
  - Spreadsheet recalculates whenever such cells are modified



# Familiar Example 2: User Interfaces

- Implementing a GUI
  - Must respond to mouse/keyboard events
  - Setup *Listeners*
  - Respond to events by running handlers *asynchronously*
- JavaScript (AJAX, SPA) web-pages
  - Make a REST request to back-end server
  - Define a *call-back* for when data is returned
  - Call-back invoked *asynchronously*

# Asynchronous Components

- Similar to messaging systems
  - Independent components (tasks)
    - Respond to incoming events
    - Pass on results by generating events
  - *But components do not get their own thread*
    - Thread selects and runs components that are ready
    - Then switches to next ready-to-run component
    - Switching components *much* cheaper than thread-switching
- Analogous concepts
  - Actors, Coroutines, C.S.P.

*CSP = Communicating Sequential Processes*

# Back-Pressure

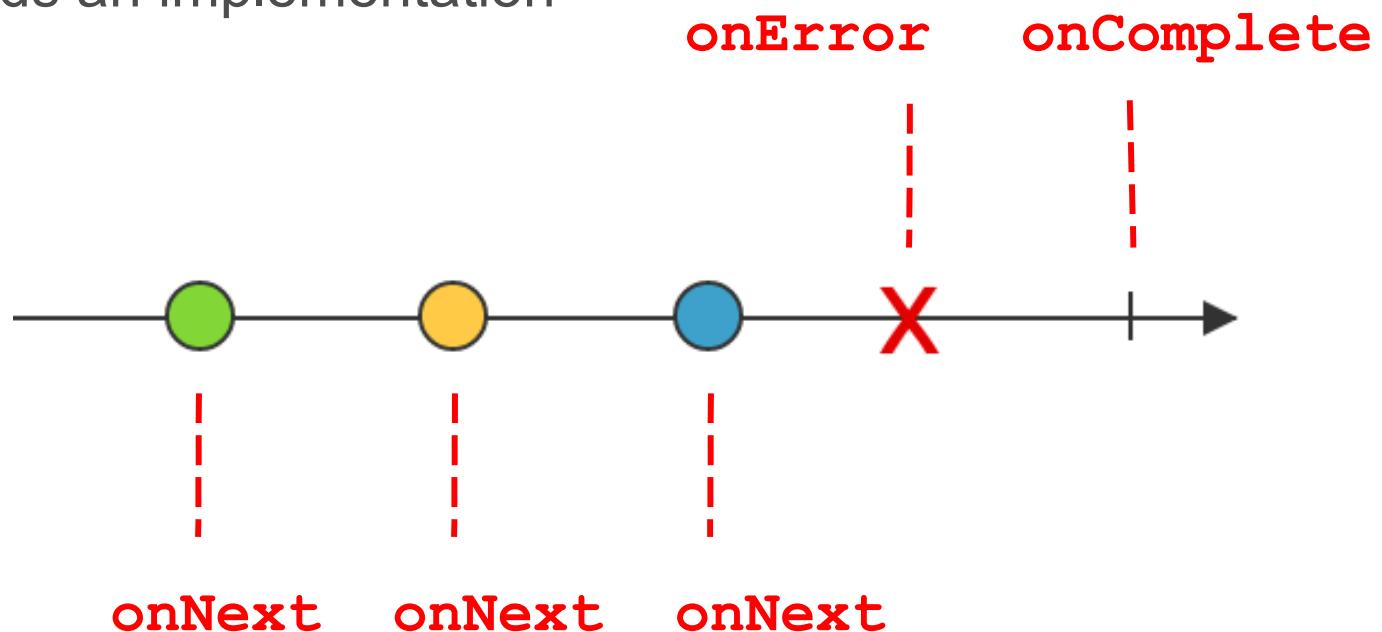
- Controls data flow through the reactive pipeline
  - Ensures producers don't overwhelm consumers
- *Example*
  - Pipeline of reactive components from the database to the HTTP socket
  - If HTTP client is too slow, data repository slows down or stops until capacity frees up

# Roadmap

- What is Reactive Programming?
- **Reactive Stream Implementations**
- Reactive Spring Features

# Handle Stream of Events Asynchronously

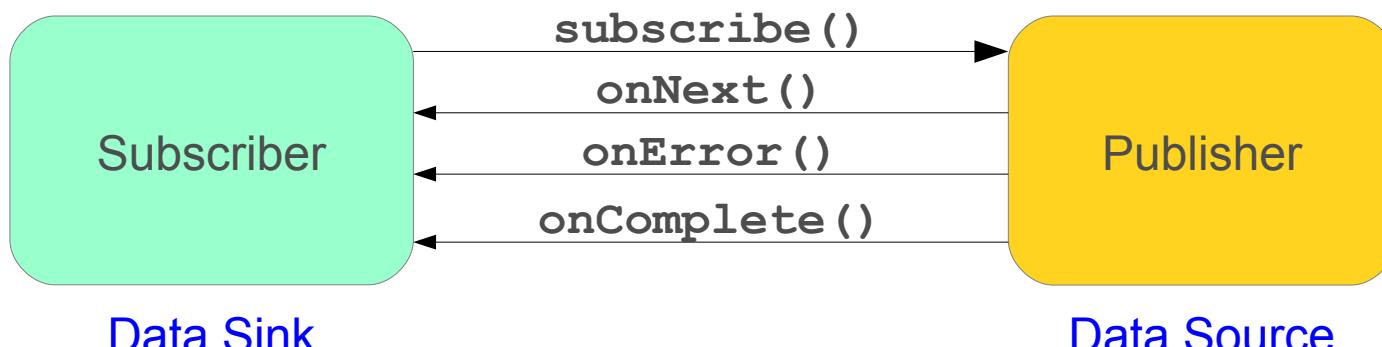
- Reactive Streams
  - Specification of *Interfaces*
  - Needs an implementation





# Reactive Streams

- Require a publisher and a subscriber
  - Like messaging
- Publisher specifies how to collect data
  - Produces a “stream” of 0, 1 or more data items
- Subscriber defines what to do with it
  - *Nothing happens without a subscriber*



# Reactive Stream Implementations

- Flow classes
  - Part of Java 9 JDK
- Project Reactor from Pivotal
  - Supported by Spring 5
    - <http://projectreactor.io>
- RxJava
- Spring supports Reactor and RxJava
  - Don't require Java 9



PROJECT **REACTOR**

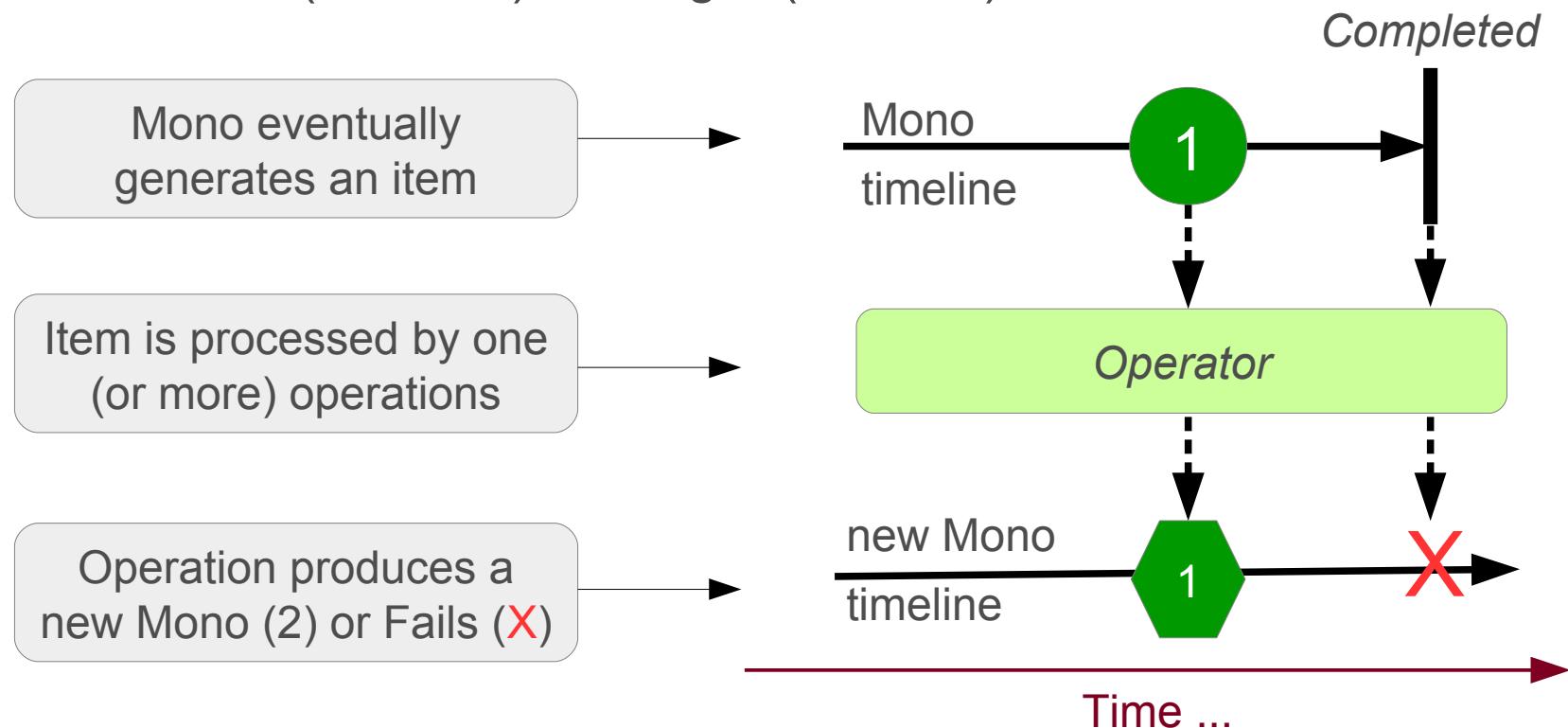


RxJava

# Basic Types: Sequence of [0 .. 1]

Mono and Single  
are Publishers

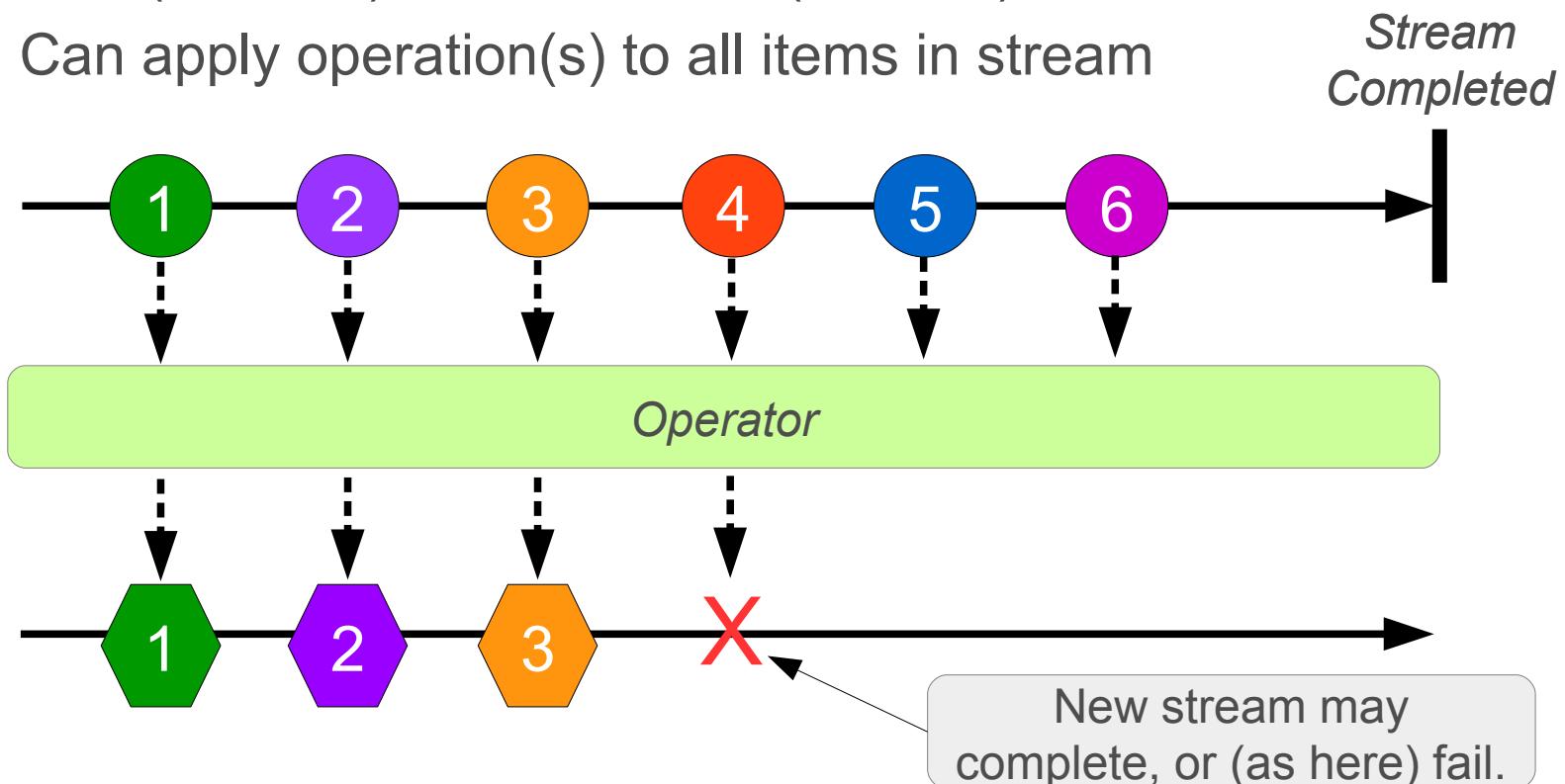
- Used to return a single result (or fail doing so)
  - Mono* (Reactor) or *Single* (RxJava)



Flux, Observable  
are Publishers

# Basic Types: Sequence of [0 .. N]

- Return continuous stream of events with failure handling
  - Flux (Reactor) or Observable (RxJava)
  - Can apply operation(s) to all items in stream

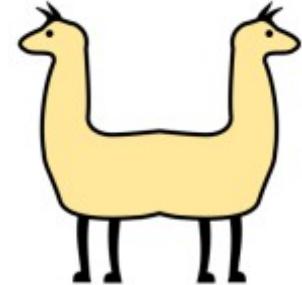


# Subscribing to a Stream

- Two options
  - Implement a `Subscriber<T>`
    - `onSubscribe`, `onNext`, `onError`, `onComplete`
  - Provide a `Consumer`
    - Has a single `accept(T)` method
      - Can pass a lambda

```
Flux.just( "red", "green", "blue" ) // The producer
    .log()                         // Stream operator: log item
    .map(String::toUpperCase())      // Stream operator: convert to upper case
    .subscribe(System::println);     // Subscribe, consumer prints each item
```

# Java Streams are *not* Reactive



*pushmi-pullyu*

- Java Streams
  - Subscriber actively *pulls* in data

```
List<Shop> shops = customer.getVendors();
List<BigDecimal> discountedPrices =
    shops.stream() // Fetches data from list
        .map(Shop::getPrice)
        .map(Discount::applyDiscount)
        .collect(Collectors.toList());
```

- Reactive Streams
  - Data is sent (pushed) to subscriber via *callbacks*
  - By “pushing-back” we get back-pressure

# Roadmap

- What is Reactive Programming?
- Reactive Stream Implementations
- **Reactive Spring Features**

# Reactive Features in Spring

- Spring Data: Reactive Repositories
  - Returns query results as a Stream
- WebFlux
  - Reactive `@Controllers`
- Web Client
  - Reactive alternative to `RestTemplate`

# Reactive Spring Data Repository

- Spring Data is going Reactive too

```
public interface CustomerRepository  
    extends ReactiveCrudRepository<Customer, Long> {  
    Mono<Customer> findBySocialSecurityNumber(String ssn);  
    Flux<Customer> findBySuburb(String suburb);  
}
```

Using  
Reactor API

Reactor defines **Mono** and **Flux**

```
public interface CustomerRepository  
    extends RxJavaCrudRepository<Customer, Long> {  
    Single<Customer> findBySocialSecurityNumber(String ssn);  
    Observable<Customer> findBySuburb(String suburb);  
}
```

Using  
RxJava API

RxJava defines **Single** and **Observable** instead

# WebClient – 1

- Asynchronous alternative to RestTemplate
  - HTTP response automatically handled by *different* thread to HTTP request

```
WebClient client = WebClient.create(ACCOUNT_SERVER_URL);  
Mono<Account> result = client.get()  
    .uri("/accounts/{id}", id) ← HTTP GET for account {id}  
    .accept(MediaType.APPLICATION_JSON)  
    .retrieve()   // Send request  
    .bodyToMono(Account.class); ← Response (a Mono) so  
                                block until available  
  
Account account = result.block(); ← // Wait for account to be returned
```

# WebClient – 2

The Mono is processed  
in a different thread

- Alternative ways of processing the Mono
  - Note you must **subscribe()** every time

```
Mono<Account> result = client.get() ...
```

```
result.subscribe(a -> logger.info("Account: " + a.getName());  
    // Print the account name when its returned
```

```
result.doOnSuccess(a -> {  
    // For each item returned  
    a -> logger.info("Account: " + a.getName());  
}).doOnError(e -> {  
    System.out.println(e.getMessage());  
}).subscribe();
```

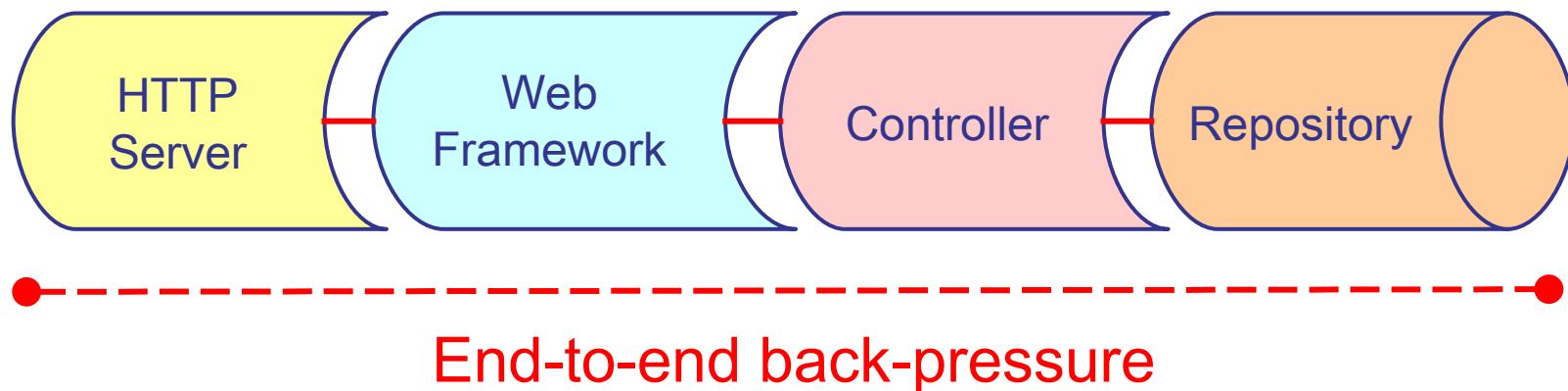
Pass a lambda to  
**subscribe()**

These are just  
callbacks

Must subscribe ()

# Spring WebFlux

- Consider incoming HTTP Requests as a stream
  - Process in usual way
  - Controllers return Reactive Streams
- A *Reactive Web Pipeline*



# Reactive Web Controller

Using  
Reactor API

```
@Controller
public class CustomerController {
    private final CustomerRepository customerRepository; // Reactive repository

    @Autowired
    public CustomerController (CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    @GetMapping("/customers/{id}")
    public Mono<Customer> getCustomer(@PathVariable Long id) {
        return customerRepository.findById(id); // Or return Single<Customer>
    }

    @GetMapping("/customers")
    public Flux<Customer> getCustomer() { // Or return Observable<Customer>
        return customerRepository.findAll();
    }
}
```

# Getting Started

- Support in Spring Boot 2.0 and Spring Initializr

SPRING INITIALIZR bootstrap your application now

Generate a  with Spring Boot

**Project Metadata**

Artifact coordinates

Group

Artifact

**Dependencies**

Add Spring Boot Starters and dependencies to your application

Search for dependencies

**Reactive Web**  
Reactive web development with Tomcat and Spring Reactive (experimental)

**Actuator**  
Production ready features to help you monitor and manage your application

Don't know what to look for? Want more options? [Switch to the full version.](#)

[Generate Project](#)



# References



- Reactive Programming
  - Spring Blog article by Dave Syer
    - Part 1, Part 2, Part 3
  - An Introduction to Reactive by André Staltz
  - Project Reactor documentation
- WebFlux
  - Article by Rossen Stoyanchev (Spring MVC lead)
  - WebFluxing 5 reference documentation



# Summary

- What is Reactive Programming?
- Reactive Spring Features



Consider taking our ***Reactive Spring Programming*** course  
See: <http://pivotal.io/academy>

# Lab

Building a simple Reactive Application

# Spring 5 – Web Stack

**@Controller @RequestMapping**

Router  
Functions

Spring Web MVC

*Spring WebFlux*

NEW

Servlet API

HTTP / Reactive Streams

NEW

Servlet Container

Tomcat, Jetty, Netty, Undertow

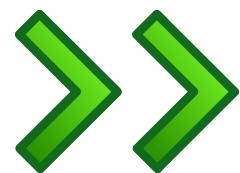
# Finishing Up

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 JMS and JMX



# Certification



- Computer-based exam
  - 50 multiple-choice questions
  - 90 minutes
  - Passing score: 76% (38 questions answered successfully)
- Preparation
  - Download *Spring Professional* certification guide
    - Make sure to pick the guide for *this* version of the course
  - Review all the slides
  - Redo the labs

<https://pivotal.io/academy/certification>



# 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?
  - Online at PSI (Innovative Exams)
    - <https://www.examslocal.com>
- How?
  - When you are ready: buy an exam voucher
  - Register/sign-in and book an exam using the voucher
    - <http://it.psionline.com/exam-faqs/pivotal-faq>
  - Take the test from *any* location
- For more information, email
  - [education@pivotal.io](mailto:education@pivotal.io)



Voucher is valid for 3 months  
– *use it or lose it!*

# Other courses



- Many courses available
  - Spring Web
  - Enterprise Spring
  - Spring Boot Developer
  - Spring Cloud Services (Microservices with Spring)
  - Pivotal Cloud Foundry Developer
  - Pivotal Cloud Foundry Administration
  - Big Data and Analytics, Hadoop, Gemfire, ...
- See <https://pivotal.io/academy>

# Spring Web

- Four-day workshop
- Making the most of Spring in the web layer
  - Spring MVC using Spring Boot
  - REST using MVC and AJAX, CORS
  - Security of Web applications
  - Mock MVC testing framework
  - Spring Web Sockets
- Spring Web Application Developer certification

# Enterprise Spring



- Building loosely coupled event-driven architectures
  - Separate processing, communications & integration
- 4 day course covering
  - Tasks, Scheduling and Concurrency
  - Advanced transaction management
  - REST Web Services with Spring MVC
  - Spring Integration
  - Spring Batch
  - Data Ingestion and Transformation



# Spring Boot Developer

- 2 day workshop covering
  - Getting started with Spring Boot
  - Spring Boot CLI
  - Configuration, auto-configuration and profiles
  - Web development and REST
  - Data Access: JDBC, JPA, Spring Data, NoSQL
  - Testing
  - Security, Messaging
  - Deployment, Metrics, Actuator
  - Microservices

# Spring Cloud Services

## Microservices With Spring



- Course topics:
  - Introduction to Spring Boot
    - Underpins all Spring Cloud projects
  - Pushing Applications to a PaaS
    - Using Pivotal Cloud Foundry
  - What are Microservices?
    - Architecting a microservices solution
  - Cloud infrastructure services and Netflix OSS
    - Service Configuration
    - Service Registration
    - Load-balancing and fault tolerance
    - Security using OAuth

# Pivotal Cloud Foundry Developer



CLOUD FOUNDRY

- 3 day course covering
  - Application deployment to Cloud Foundry
    - Deployment using *cf* tool or an IDE
    - Using the PCF Application Manager
  - Cloud Foundry Concepts
    - Logging, Continuous Integration, Monitoring
    - Accessing and defining Services
    - Using and customizing Buildpacks
  - Design considerations: “12 Factor”
    - JVM application specifics, using Spring Cloud

# Pivotal Cloud Foundry Administrator



CLOUD FOUNDRY

- 4 day course covering
  - Application deployment to Cloud Foundry
    - Logging, scaling, services
  - “Day 1 Operations”
    - Installation of PCF Ops Manager and Elastic Runtime
    - Configuring users, roles, and quotas
    - Capturing and reading logs
  - “Day 2 Operations”
    - Backing up and restoring an installation
    - Using BOSH
    - Upgrading Ops Manager and tiles.

# Pivotal Support Offerings

- 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 Support FAQ
  - <https://discuss.pivotal.io/hc/en-us/articles/225569608-Pivotal-Support-FAQ>



# 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
  - Mentoring: design, development and product management
    - <http://www.pivotal.io/agile>
    - <http://pivotallabs.com>



# Resources

- The Spring reference documentation
  - <http://spring.io/docs>
  - <http://projects.spring.io/spring-boot>
  - <http://projects.spring.io/spring-data>
  - <http://projects.spring.io/spring-security>
  - <http://projects.spring.io/spring-cloud>
- 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
- Your course is registered with the Pivotal Academy
  - Please fill out the [\*evaluation form\*](#)
  - Once you've done, login to *Pivotal Academy*
    - You can download your Attendance Certificate

*Don't forget the optional sections*

## *Optional Section*

# Object Relational Mapping

## Using OR Mapping in the Enterprise

Fundamental Concepts and Concerns

# Objectives

- After completing this lesson, you should be able to:
  - Explain the Object/Relational “impedance” mismatch
  - Evaluate ORM in a Java Application
  - Assess the benefits and challenges of O/R Mapping



# 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

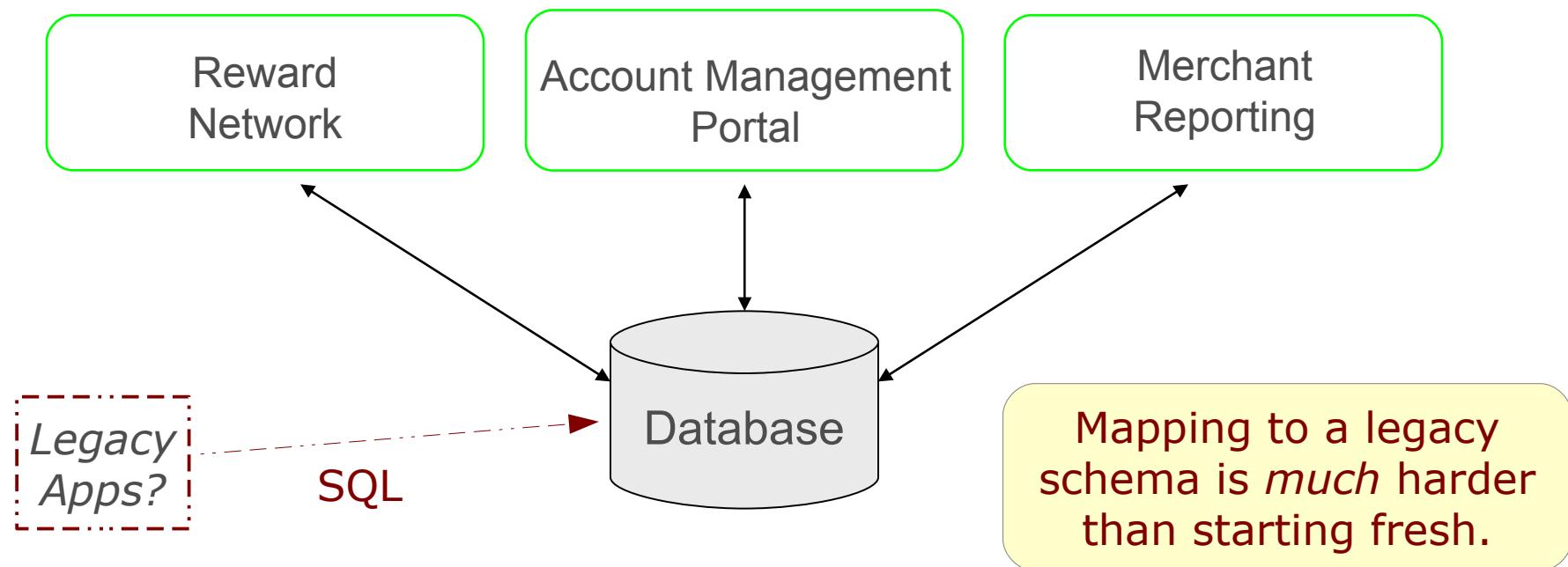
- Object/Relational Mapping (ORM) engines exist to mitigate the mismatch
- Spring supports all of the major ones:
  - Hibernate
  - EclipseLink
  - OpenJPA
  - ...

# 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



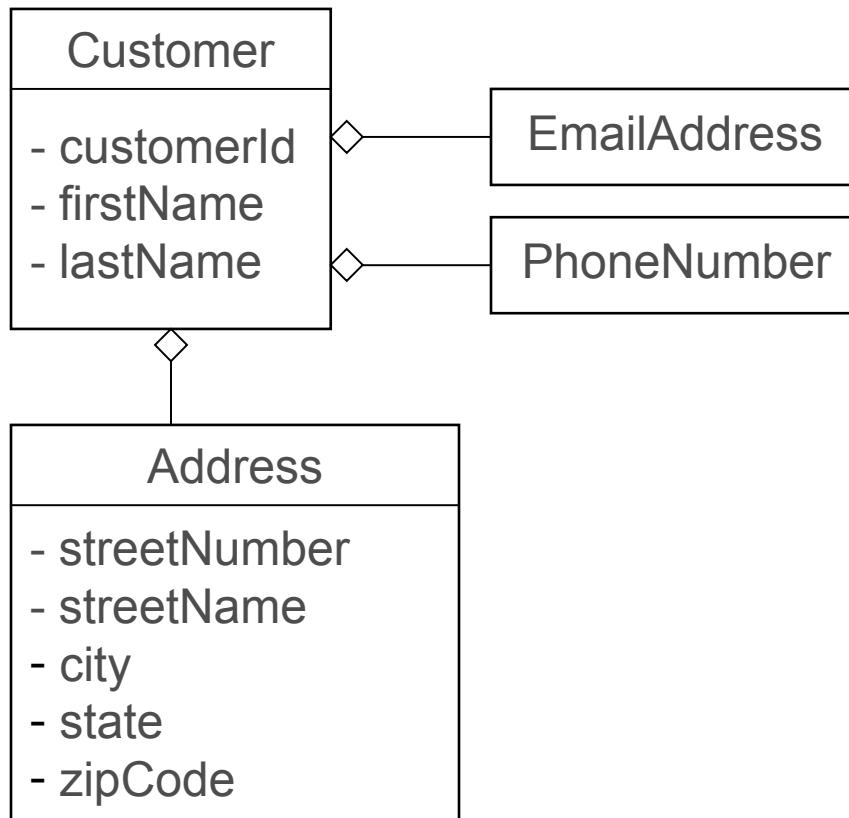
# O/R Mismatch: Granularity (1)

- 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 physical object)*
  - `x.equals(y)`    *equivalence (contain same data)*
- 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:

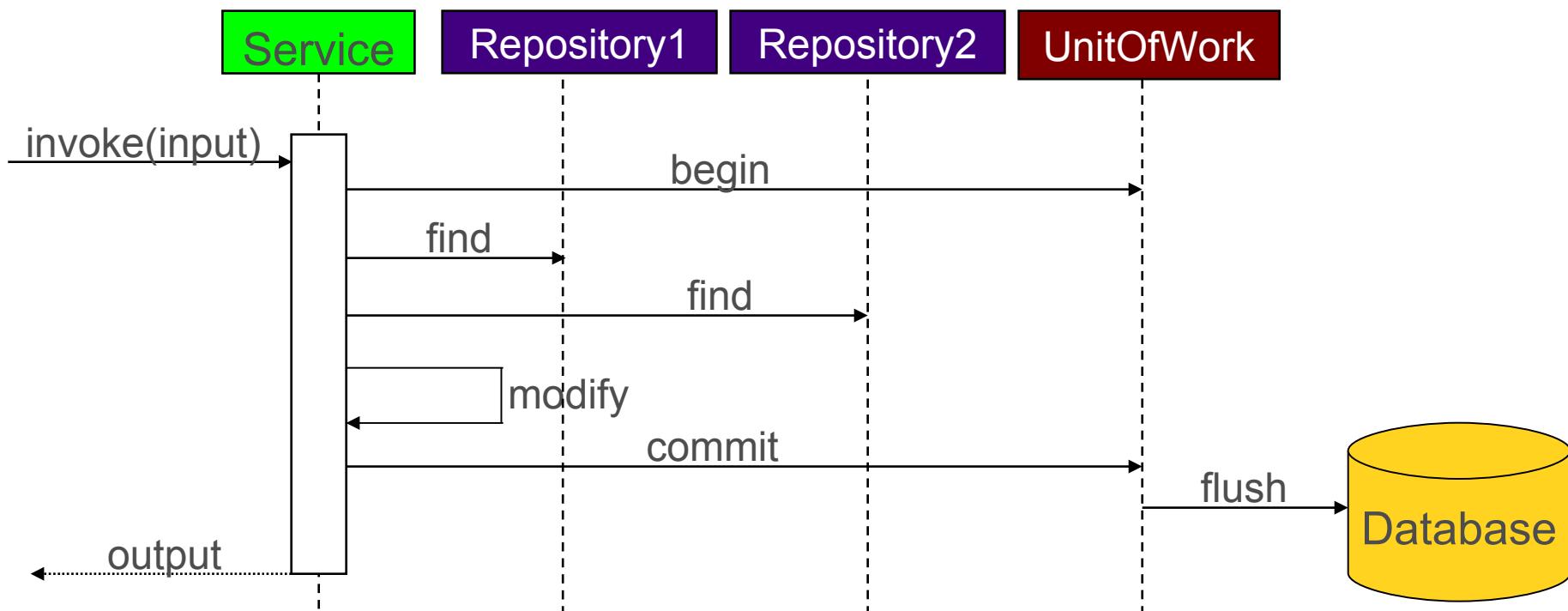
```
SELECT c.first_name, c.last_name, a.city, ...
  FROM customer c, customer_address ca, address a
 WHERE ca.customer_id = c.id
   AND ca.address_id = a.id
   AND a.zip_code = 12345
```

- 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  
}  
  
if (order.isCancelled()) {
```

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

Item becomes transient

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

# Overview of Spring Web

## Developing Modern Web Applications

Servlet Configuration, Product Overview



# Objectives

- After completing this lesson, you should be able to:
  - Describe an Effective Web Application Architecture
  - Setup Spring for Web Applications
  - Describe the basics of Spring Web
  - Explain the use of Spring with third-party Web frameworks





# Why No Spring Boot?

- Many existing Spring MVC applications were written prior to Spring Boot
- Currently in production and under maintenance
  - **Remember:** applications may be in production for *many years, even decades*
- *How are they configured if Spring Boot doesn't do it for you?*

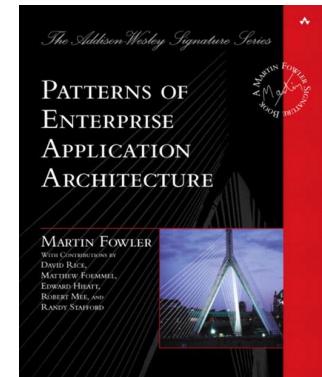
@MVC

# Topics in this Session

- **Using Spring in Web Applications**
- Configuration using `web.xml`
- Configuration using Java
- Using Spring MVC without Boot

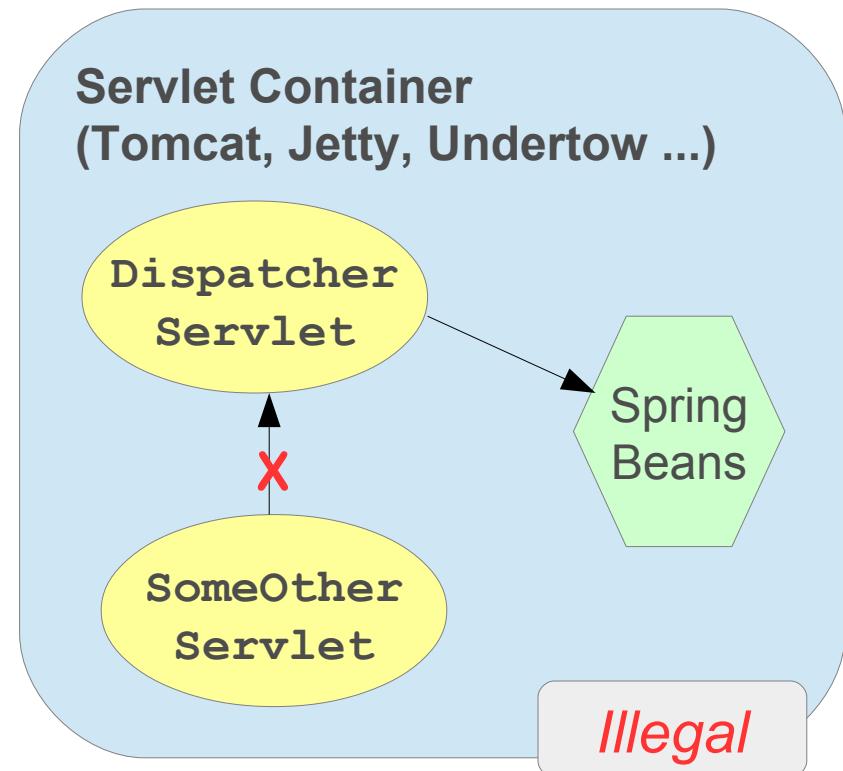
# Front Controller Pattern

- Single controller to handle all requests to a Framework
  - Spring MVC has a **DispatcherServlet**
  - JSF: **FacesServlet**
  - RestEasy: **HttpServletDispatcher**
  - Struts uses an **ActionServlet**
- All work with Spring
  - May use multiple servlets in *same* application
  - **But** don't want to load Application Context *multiple* times
  - Servlet's *can't* pass each other data



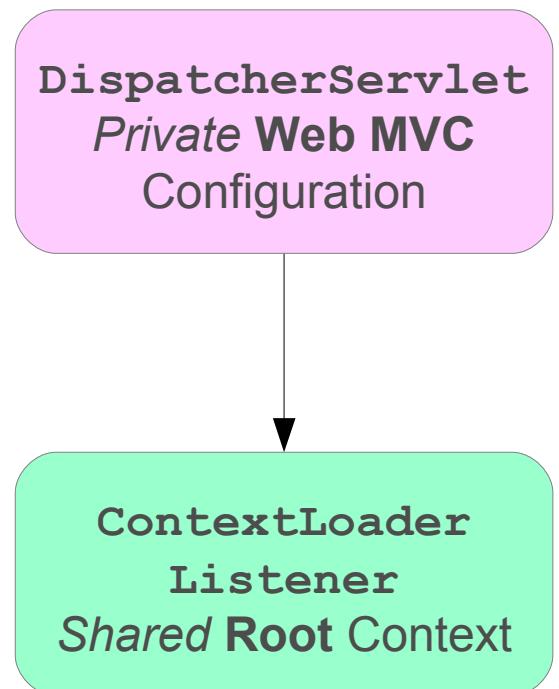
# Can't Do This ...

- Spring MVC's servlet *could* load the application context
  - But other servlets can't ask it for its beans
    - Servlets "sandboxed" for safety
  - Nor could any servlet Filters (same limitation)



# Recommendation: Separate Configurations

- Shared Configuration
  - Common to *all* servlets
    - *The Application*
      - services, repositories ...
    - Loaded by Spring into a *Root Application Context* using a custom *Listener*
- Dedicated Configuration
  - Just the beans used by each servlet
  - For Spring MVC: the *DispatcherServlet*
    - Loads its own beans – separation of concerns



# Spring Application Context

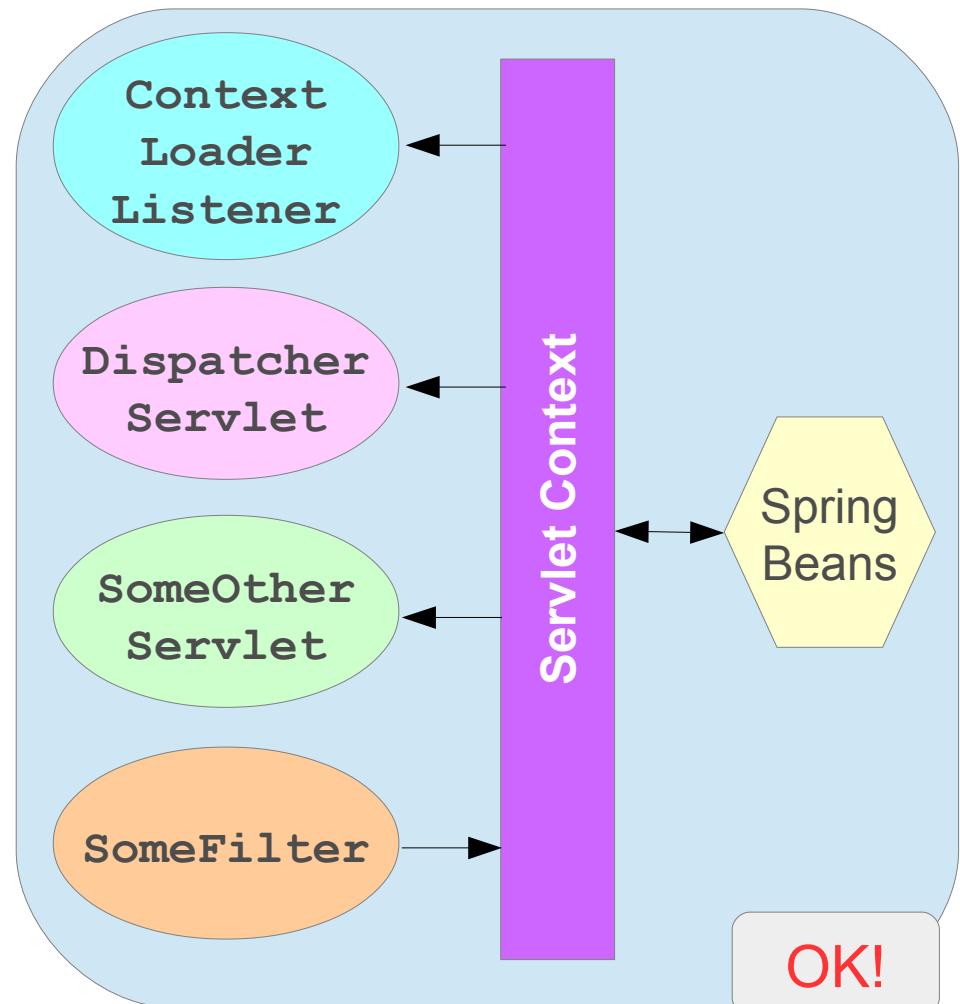
## Lifecycle in Web Applications

- Initialize Spring within a web application
  - Creates **Root** application context
  - Contains business services, repositories, ...
- Use a standard servlet listener
  - Initialization occurs before any servlets execute
  - Application ready for user requests
  - `ApplicationContext.close()` is called when the application is stopped

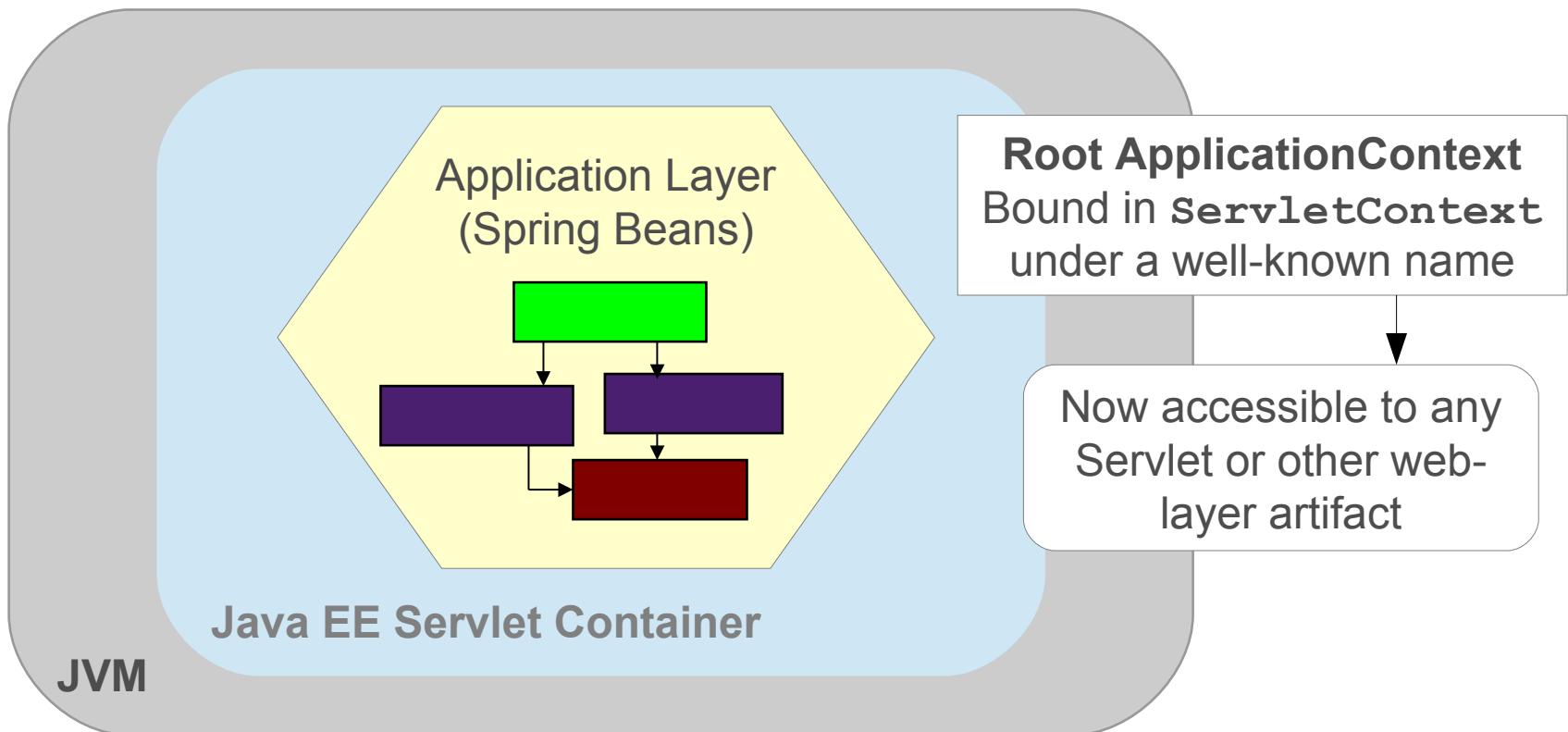
# ContextLoaderListener

- Spring Web “Bootstrap”
  - Load **shared Root Application Context** *before* any Servlets or Filters start up
  - Shared via Container's **ServletContext**

Servlet Container  
(Tomcat, Jetty, Undertow ...)



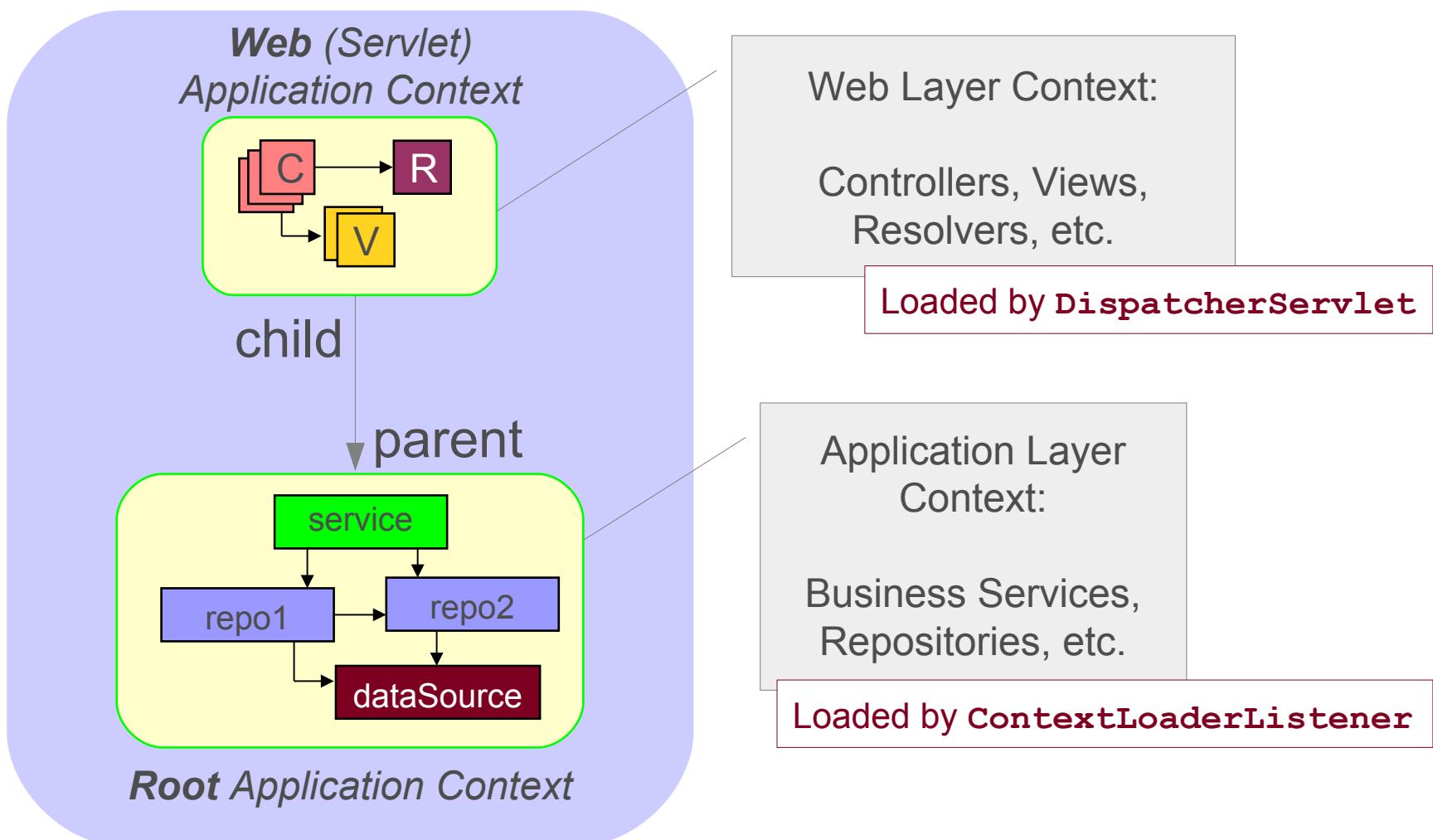
# Servlet Container After Starting Up



# DispatcherServlet

- Spring's “front controller”
  - Coordinates all request handling activities
  - Delegates to Web infrastructure beans
  - Invokes user Web components (like Controllers)
- Creates its own **WebApplicationContext**
  - Contains just web-related beans
    - Such as Controllers, Views, View Resolvers ...
  - Web-aware, supports session/request scopes
  - Has access to the “Root” application context
    - *Example:* Account Controller can call business methods on the Account Service

# Servlet Container Once Initialized





# Comparison with Spring Boot

- Spring Boot creates a *single WebApplication-Context* containing all your beans
  - Does not use a `ContextLoaderListener`
- Because Servlets are created dynamically
  - `DispatcherServlet` created first
  - Puts application context in the shared `ServletContext`
  - Other servlets *can* access your Spring Beans
    - By fetching the application context from the `ServletContext`

# Accessing Spring Beans in a Servlet

- Use *WebApplicationContextUtils*
  - gets Spring *ApplicationContext* from *ServletContext*

```
public class TopSpendersReportGenerator extends HttpServlet {  
    private ClientService clientService;  
  
    public void init() {  
        ApplicationContext context = WebApplicationContextUtils.  
            getRequiredWebApplicationContext(getServletContext());  
        clientService = (ClientService) context.getBean("clientService");  
    }  
    ...  
}
```

Your web framework's servlet typically does this for you

# Spring MVC Supports 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



Spring MVC's own servlet accesses *ApplicationContext* for you  
No need for low-level *WebApplicationContextUtils*

# Configuration Options

- **Servlet 2 XML**
  - Define as `<listener>` and `<servlet>` in `web.xml`
- **Servlet 3 Initializer**
  - Spring automatically loads any class implementing `WebApplicationInitializer`

# Topics in this Session

- Using Spring in Web Applications
- Configuration using `web.xml`
- Configuration using Java
- Using Spring MVC without Boot

# Listener Configuration

- Only option prior to Servlet 3.0
  - Just use Spring's servlet listener

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

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

The application context's configuration file(s)

Loads the **Root** ApplicationContext into the  
ServletContext *before* any Servlets are initialized

# web.xml Configuration Options

- Default resource location is document-root
  - Can use *classpath:* designator

```
<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>jpa</param-value>
</context-param>
```

web.xml

Multiple files  
may be listed

Optionally specify  
profile(s) to use

# Dispatcher Servlet

```
<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>/rewardsadmin/*</url-pattern>
</servlet-mapping>
```

Pre-Servlet 3.0

**web .xml**

Beans defined in **Web** context have access to beans defined in **Root ApplicationContext**

# Topics in this Session

- Using Spring in Web Applications
- Configuration using `web.xml`
- **Configuration using Java**
- Using Spring MVC without Boot

# Spring Integration with Servlet 3

- Servlet 3 Specification:
  - Container loads class implementing **ServletContainerInitializer**
- Spring defines its own implementation
  - **SpringServletContainerInitializer**
  - This in turn loads all classes implementing Spring's **WebApplicationInitializer** interface
    - Can have multiple initializers for different subsystems instead of one giant initializer

# Configuration: ContextLoaderListener

```
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.

Multiple classes  
may be listed

Sets up **ContextLoaderListener** using context returned.  
Spring's **ApplicationContext** loaded into **ServletContext**  
before any Servlets are initialized

# Easier All-In-One Solution

## ContextLoaderListener and DispatcherServlet

```
public class MyWebInitializer  
    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[]{"rewardsonline/*"};  
    }  
}
```

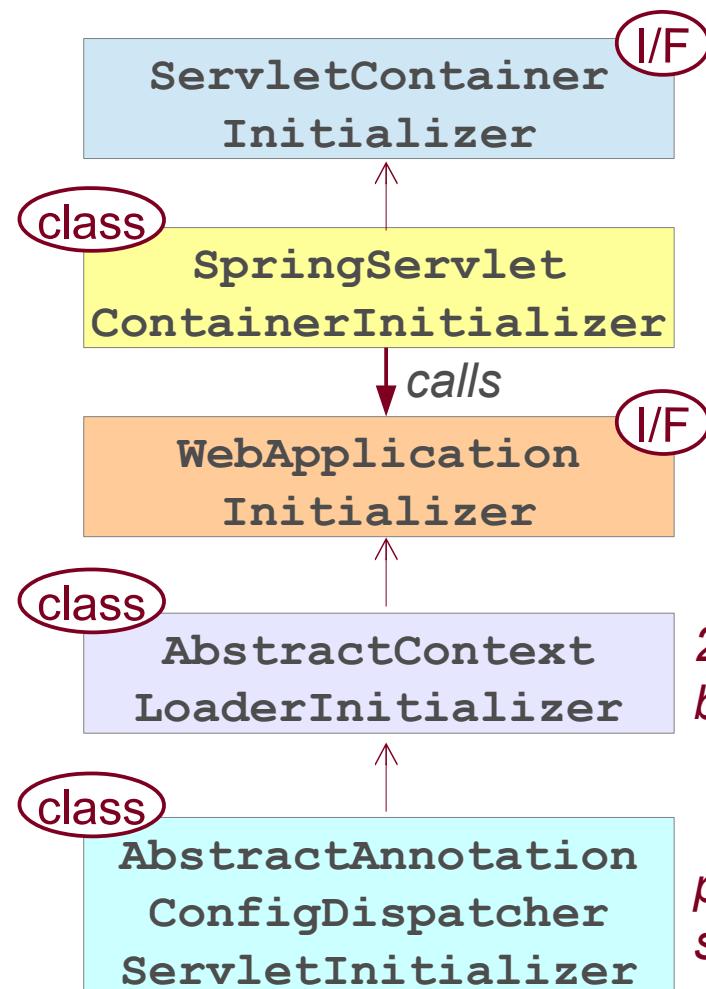
“Root” configuration

MVC configuration

Beans defined in  
MVC Context  
have access to  
beans defined in  
Root Context.

# About Web Initializer Classes

Key ↑ inherit ↓ invoke



- Interface from *Servlet 3 specification*, implement to initialize servlet system
- Spring's implementation which, in turn, delegates to one or more ...
- Base-class for all Spring MVC apps to implement for servlet configuration *without* web.xml
  - Sets up a **ContextLoaderListener**, you provide root **ApplicationContext**
  - Also defines a **DispatcherServlet**, assumes Java Config. You provide root and servlet Java config classes

2 slides  
back

prev  
slide

# Topics in this Session

- Using Spring in Web Applications
- Configuration using `web.xml`
- Configuration using Java
- **Using Spring MVC without Boot**

# Setting up Spring MVC

- Spring MVC is highly backwards compatible
  - Most default settings have remained unchanged since Spring 2.5 (versions 3.0-3.2, 4.0-4.3, 5.0- ... !)
- Annotated and legacy Controllers enabled by default
- **BUT** Newer 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

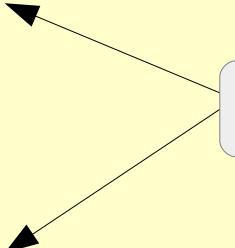
- Place on your MVC Configuration class
  - Initializes Spring MVC
    - Sets up default configuration for `@Controllers`
    - Defines useful default internal mappings and adapters
    - *Fewer* defaults than you get with Spring Boot
  - Also enables many new features since Spring 3.0
    - Validation, formatting, REST and type conversion



Full list of features: [http://docs.spring.io/spring/docs/4.3.x/  
Spring-framework-reference.html/mvc.html#mvc-config-enable](http://docs.spring.io/spring/docs/4.3.x/Spring-framework-reference.html/mvc.html#mvc-config-enable)

# @EnableWebMvc Example

```
@Configuration  
@EnableWebMvc // Add to your MVC @Configuration class  
public class MyWebConfig {  
    @Bean  
    public ViewResolver jspViewResolver() {  
        InternalResourceViewResolver r = new InternalResourceViewResolver();  
        r.setPrefix("/WEB-INF/views");  
        r.setSuffix("jsp");  
        return r;  
    }  
  
    @Bean  
    public rewardController(RewardLookupService service) {  
        return new RewardController(service);  
    }  
}
```



Define your MVC related beans

Equivalent to `<mvc:annotation-driven/>`

# Request Mapping without Spring Boot

- When Spring MVC is deployed as a classic WAR *without* Spring Boot
  - URLs are longer, but still map to controllers
- In the example below
  - We deployed **mvc.war**
  - Mapped our **DispatcherServlet** to **rewardsonline**
  - So **/accounts** maps to an @Controller method

Example URL: `http://localhost:8080 / mvc / rewardsonline / accounts`



# Summary

- Spring can be used with *any* web framework
  - Spring provides the **ContextLoaderListener** that loads your Spring Beans
- Spring MVC also requires the **DispatcherServlet**
  - Can be configured using XML or Java
  - Use **@EnableWebMvc** when not using Spring Boot
  - Note the format of URLs
- Spring has several web-related projects
  - See <http://spring.io/projects>

## *Optional Section*

# Spring JMS

Simplifying Messaging Applications

JmsTemplate and Spring's Listener Container

# Objectives

- After completing this lesson, you should be able to:
  - Explain the basic Concepts behind JMS
  - Configure JMS Resources with Spring
  - Use Spring's `JmsTemplate`
  - Send and Receive Messages
  - Implement Asynchronous Messaging using a 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

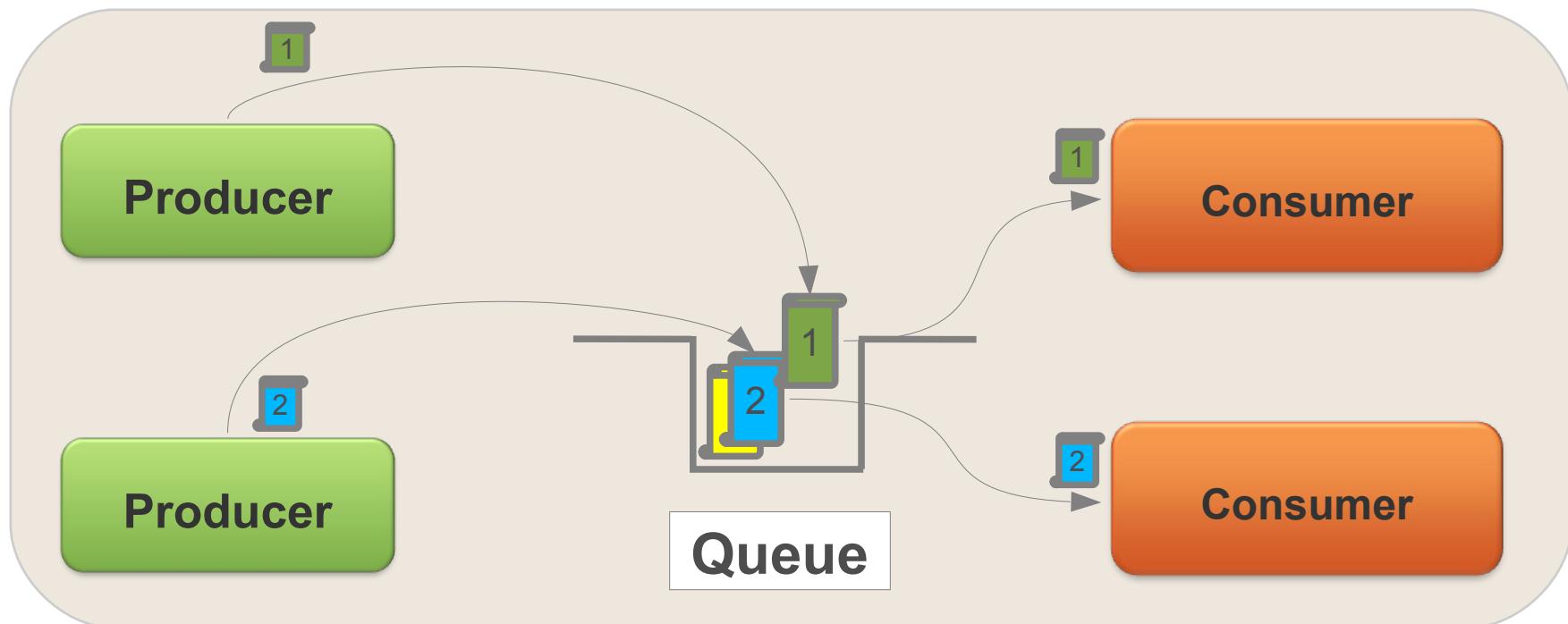
- Implementations of the Message interface
  - TextMessage
  - ObjectMessage
  - MapMessage
  - BytesMessage
  - StreamMessage

# 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 ...

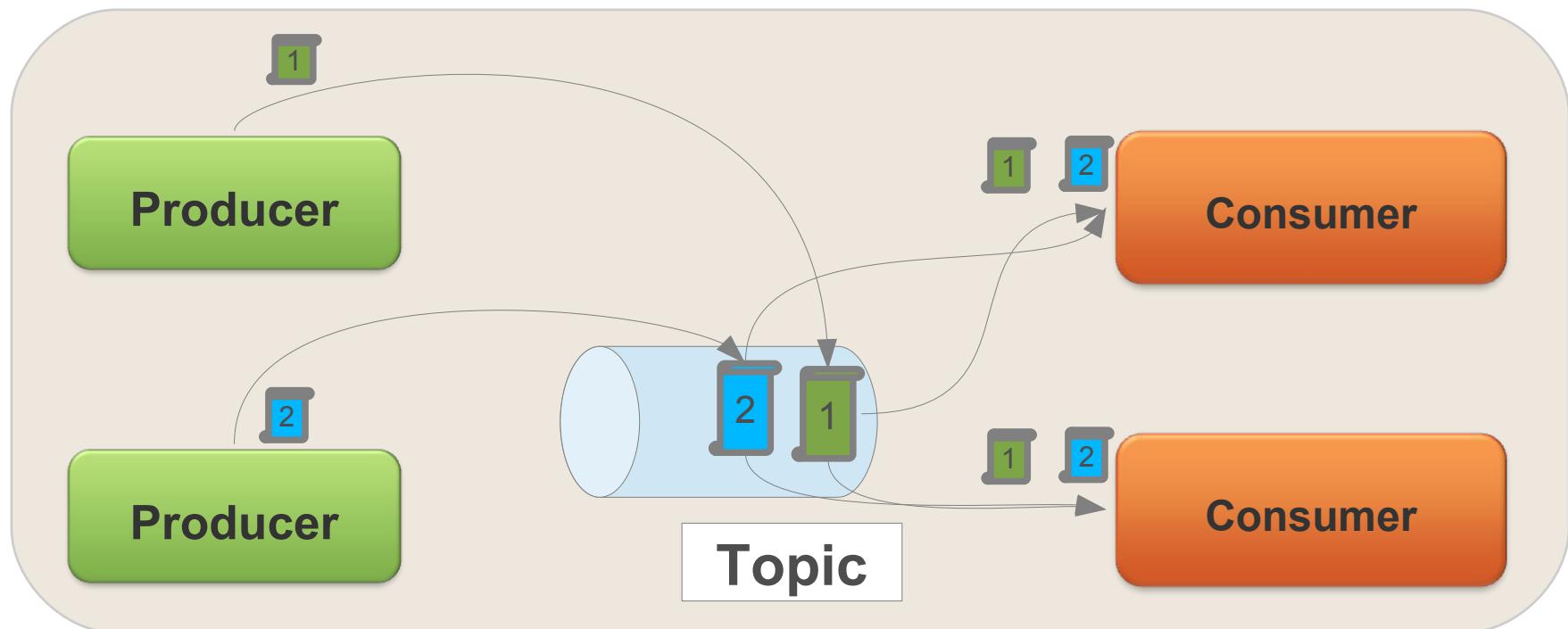
# JMS Queues: Point-to-point

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 acknowledgementMode);
```

```
// 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, ...
  - 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**

<https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#remoting-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 orderQueue() 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

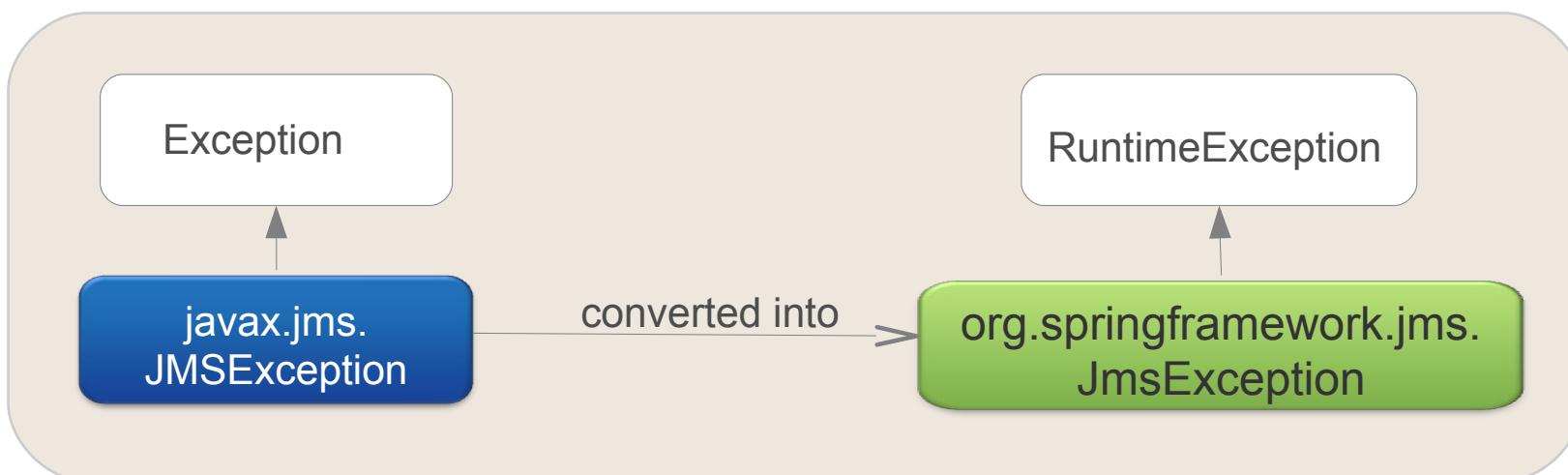
# Spring's JmsTemplate

- 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 configuration

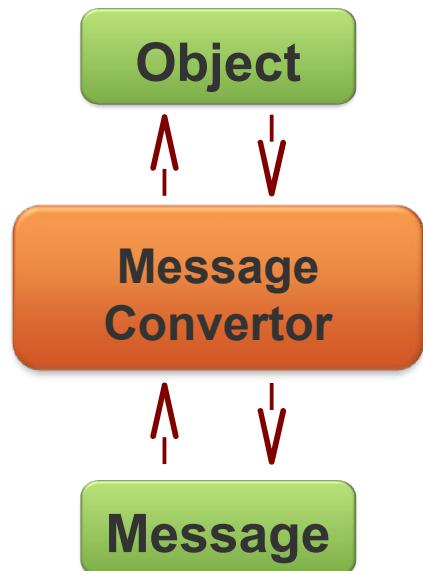
- Must provide reference to ConnectionFactory
  - via either constructor or setter injection
- Optionally provide other facilities
  - `setMessageConverter` (1)
  - `setDestinationResolver` (2)
  - `setDefaultDestination` or `setDefaultDestinationName` (3)

```
@Bean  
public JmsTemplate jmsTemplate () {  
    JmsTemplate template = new JmsTemplate( connectionFactory() );  
    template.setMessageConverter ( ... );  
    template.setDestinationResolver ( ... );  
    return template;  
}
```

(1), (2), (3) – see next few slides

# (1) MessageConverter

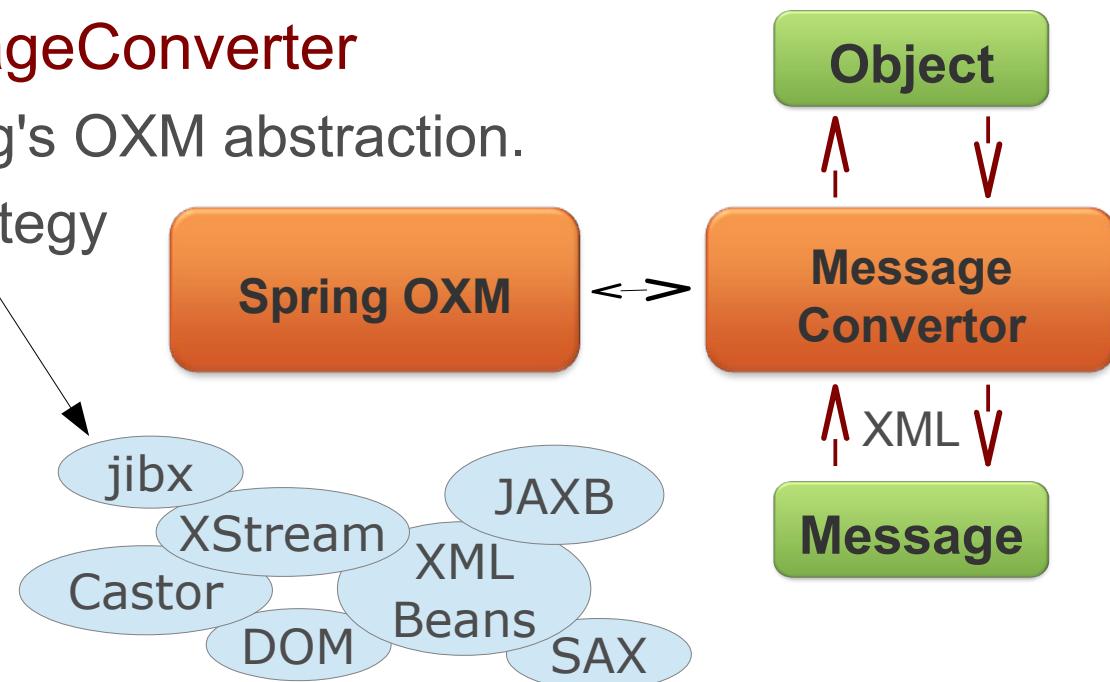
- The JmsTemplate uses a **MessageConverter** to convert between objects and messages
  - You only send and receive objects
- The default **SimpleMessageConverter** handles basic types
  - String to TextMessage
  - Map to MapMessage
  - byte[] to BytesMessage
  - Serializable to ObjectMessage



**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.

## (2) 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

# (3) Default Destination

- Used by default when sending *or* receiving messages

```
@Bean
```

```
public JmsTemplate orderTemplate () {  
    JmsTemplate template = new JmsTemplate ( connectionFactory() );  
    template.setDefaultDestination ( orderQueue() );  
    return template;  
}
```

Specify by Object

```
@Bean public JmsTemplate orderTemplate () {
```

```
    JmsTemplate template = new JmsTemplate ( connectionFactory() );  
    template.setDefaultDestinationName ("order.queue");  
    return template;  
}
```

Specify by Name

# 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

# Receiving Objects

- JmsTemplate can also *receive* data
  - Automatically converted using MessageConverter
  - Underlying messages hidden

```
public void receiveData() {  
  
    // use message converter and destination resolver  
    String s = (String) jmsTemplate.receiveAndConvert("message.queue");  
    // use message converter  
    Order order1 = (Order) jmsTemplate.receiveAndConvert(orderQueue);  
    // use message converter and default destination  
    Order order2 = (Order) jmsTemplate.receiveAndConvert();  
}
```

# Receiving Messages

- Or you may access the underlying message
  - Gives you access to message properties

```
public void receiveMessages() {  
  
    // handle JMS native message from default destination  
    ObjectMessage orderMessage = (ObjectMessage) jmsTemplate.receive();  
    Order order2 = (Order) orderMessage.getObject();  
  
    // receive(destination) and receive(destinationName) also available  
}
```

# Synchronous Message Exchange

- JmsTemplate also implements a request/reply pattern
  - Using `sendAndReceive()`
  - 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  
}
```

# Asynchronous or Synchronous



- Sending messages is asynchronous
  - The send methods return immediately
    - Even if the message takes time to be delivered
    - Recall the acknowledgement modes in `createSession()`
- But `receive()` and `receiveAndConvert()` are blocking
  - Synchronous – will wait for ever for a new message
    - optional timeout: `setReceiveTimeout()`
- How can we receive data asynchronously?
  - JMS defines *Message Driven Beans*
  - But you normally need a full JEE container to use them

# 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
- (3) Annotate POJO to be message-driven

# 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

Spring 4.1

- 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 ...

# Step (3)

## Define Receiving Method with @JmsListener

- Container with name **jmsListenerContainerFactory** is used by default

```
public class OrderServiceImpl {  
    @JmsListener(containerFactory="myFactory",  
        destination="orderConfirmation")  
    public void process(OrderConfirmation o) { ... }  
}
```

- Can also set a custom concurrency or a payload selector

```
public class OrderServiceImpl {  
    @JmsListener(selector="type = 'Order'",  
        concurrency="2-10", destination = "order")  
    public OrderConfirmation order(Order o) { ... }  
}
```

# Using JMS: Pros and Cons

- Advantages
  - Application freed from messaging concerns
    - Resilience, guaranteed delivery (compare to REST)
  - Asynchronous support built-in
  - 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

**Coming Up:** Spring's Caching Connection Factory



# Topics in this Session

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

# Alternative Step (2)

## Use 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*
  - Supports multiple listeners in a single declaration
  - 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"/>
```

No need for `@JmsListener`

# Message-Driven POJO in XML

- Listener unpacks incoming payload
  - Uses the MessageConverter
  - Invokes method on POJO
  - Return value sent to response-destination after conversion

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

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

# CachingConnectionFactory

- JmsTemplate aggressively closes and reopens resources like Sessions and Connections
  - Lots of overhead and poor performance
  - Normally these are cached by connection factory
- 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>
```

## *Optional Section*

# Performance and Operations

Management and Monitoring of Spring Java Applications

Exporting Spring Beans to JMX

# Objectives

- After completing this lesson, you should be able to:
  - Explain Basic JMX Concepts and Architecture
  - Automatically export Spring Beans as MBeans
  - Setup and Use Spring Insight



# 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

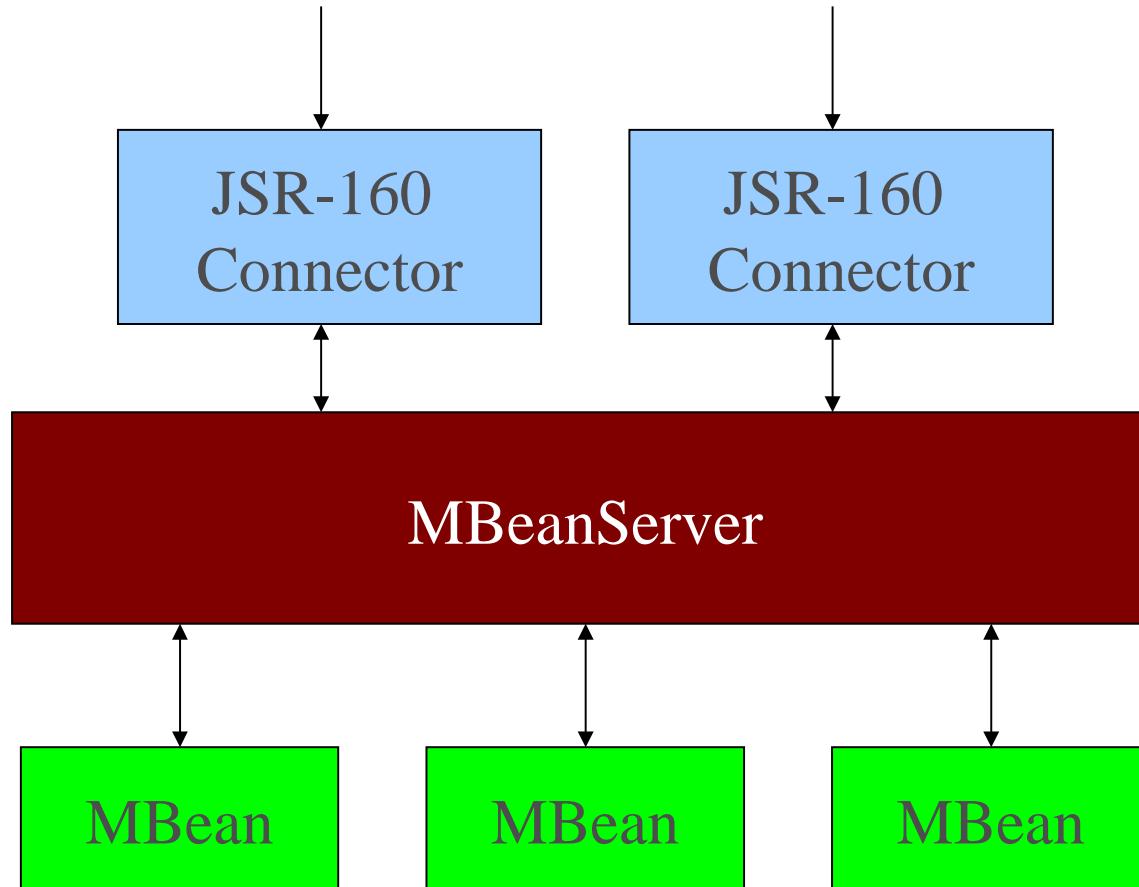
# What is JMX?

- 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:

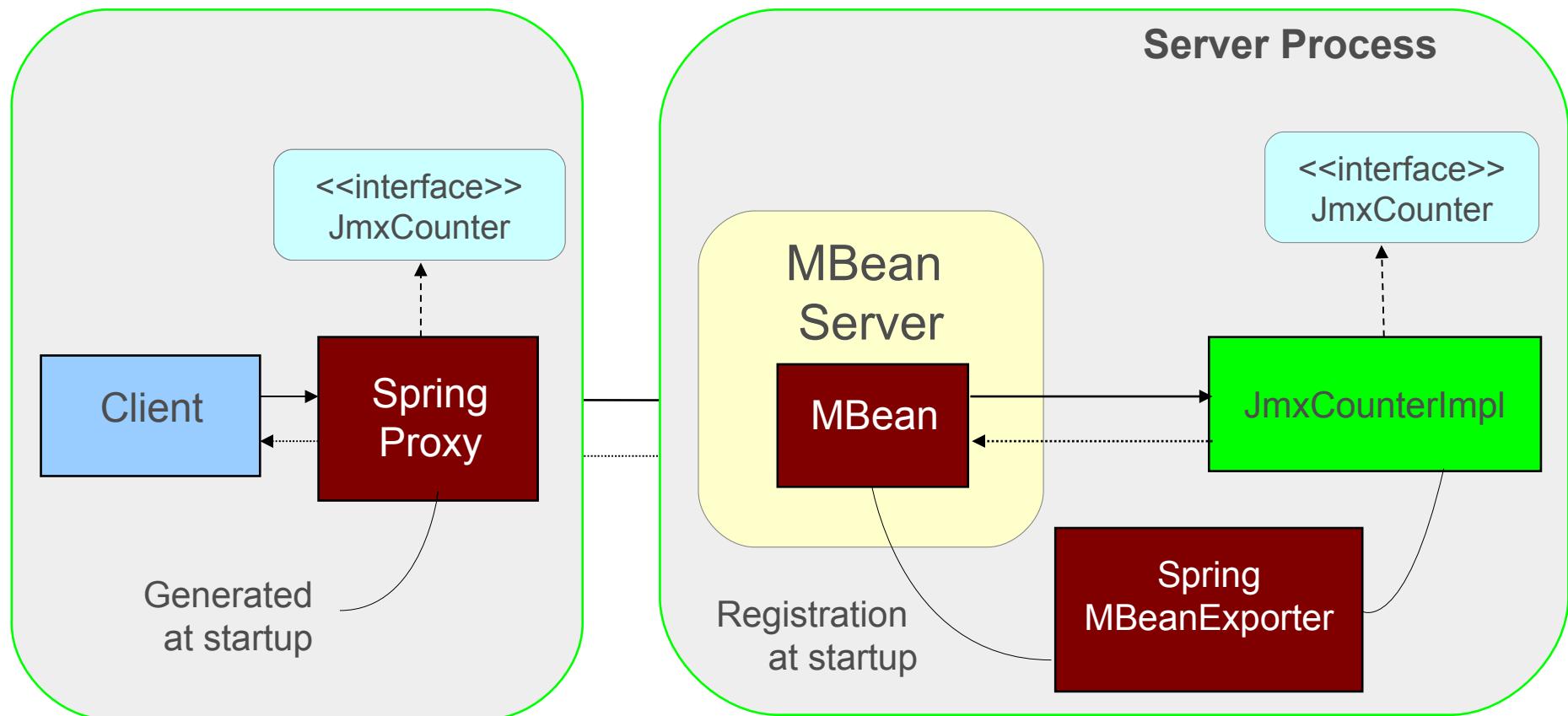
- Combine Annotations with Exporter:
  - Only annotated attributes/operations exposed.

```
@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

```
@EnableMBeanExport  
 @Configuration  
 class ContextConfiguration {  
     @Bean  
     public LoggerDynamicMBean loggerDynamicMBean() {  
         Logger logger = Logger.getLogger("org.springframework.jmx");  
         return new LoggerDynamicMBean(logger);  
     }  
 }
```

# Automatically Exporting Pre-existing Mbeans – XML

- Same configuration using XML

```
<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>
```

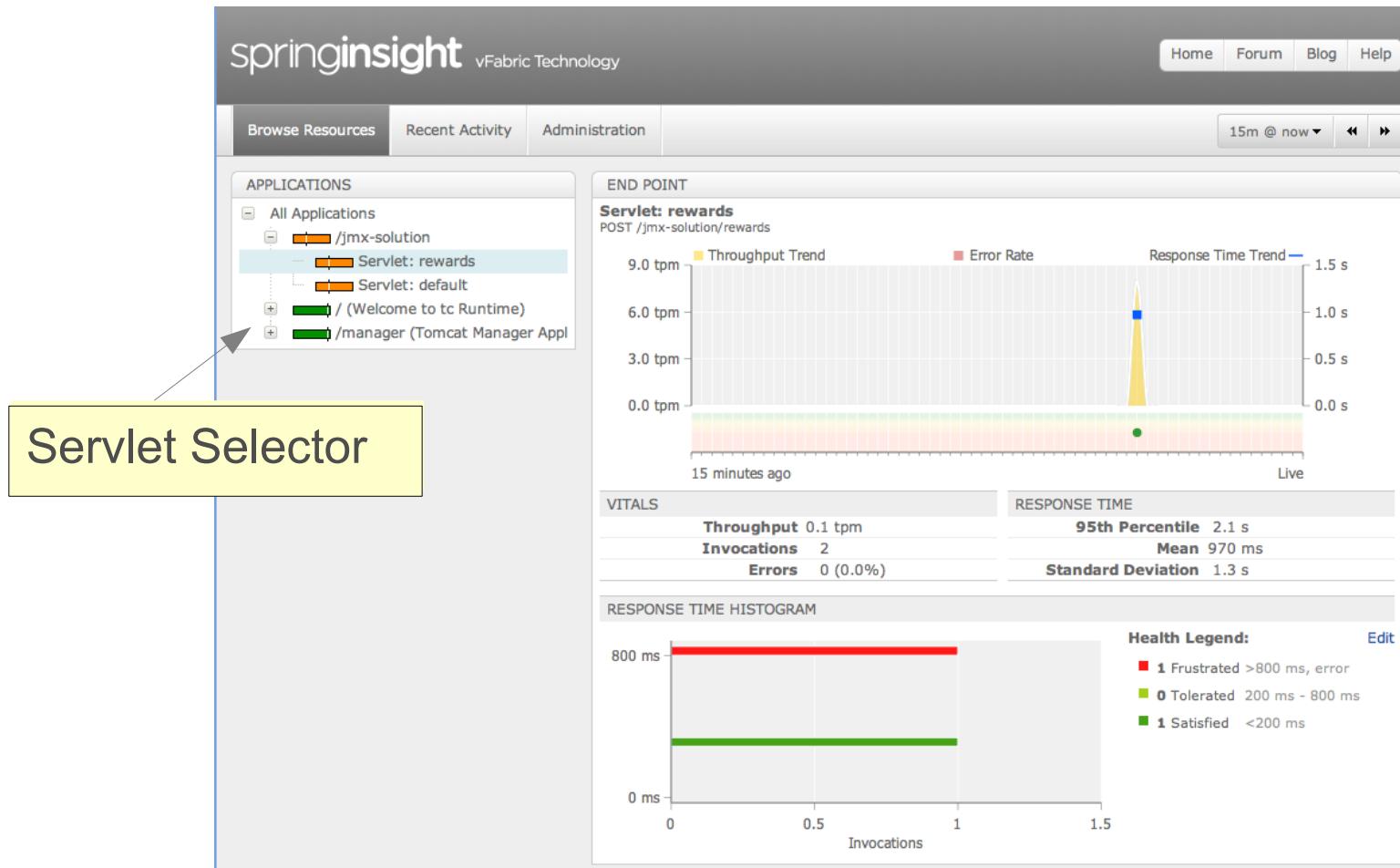
# Topics in this session

- Introduction
- JMX
- Introducing Spring JMX
- Automatically exporting existing Mbeans
- **Spring Insight**

# Spring Insight Overview

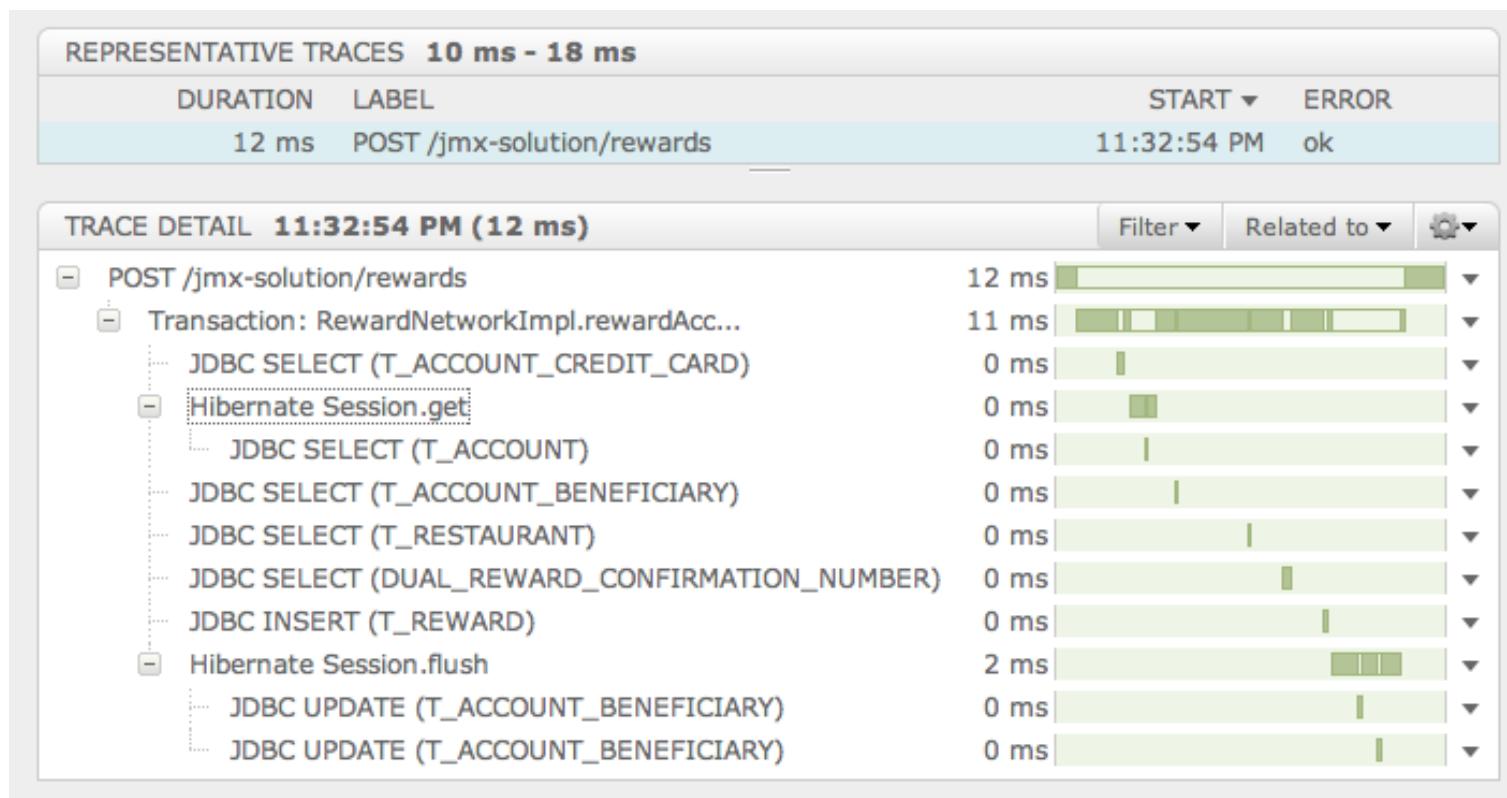
- Use with *tc Server Developer Edition*
  - Comes with STS
  - Monitors web applications deployed to tc Server
  - <http://localhost:8080/insight>
- Focuses on what's relevant
  - esp. performance related parts of the application
- Detects performance issues during development
  - Commercial version for production: *vFabric APM*

# Spring Insight Overview



# Spring Insight Overview

- A request trace from HTTP POST to SQL



# Setting up Spring Insight

- *tcServer* comes with a number of preconfigured setups
  - Spring Insight is *no longer* one of them since 3.2.x
- Instead
  - Download from our *Template Repository* 1
  - Create the Insight template 2

```
./tcruntime-admin.sh get-template spring-insight-developer
./tcruntime-instance.sh create Insight -t insight
```

12
  - Create a new Server in STS
    - Ask for Pivotal → tc Server
    - Point it to the *Insight* instance when prompted

# 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

## *Optional Section*

# Dependency Injection Using XML

Spring's XML Configuration Language

Using <bean> definitions and namespaces

# Objectives

- After completing this lesson, you should be able to:
  - Write Bean definitions using XML
  - Create a Spring Application Context using XML
  - Use XML Attributes to Control Bean Behavior
  - Explain purpose of Factory Beans and be able to use them
  - Configure Spring XML using Namespaces



# Topics in this session

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

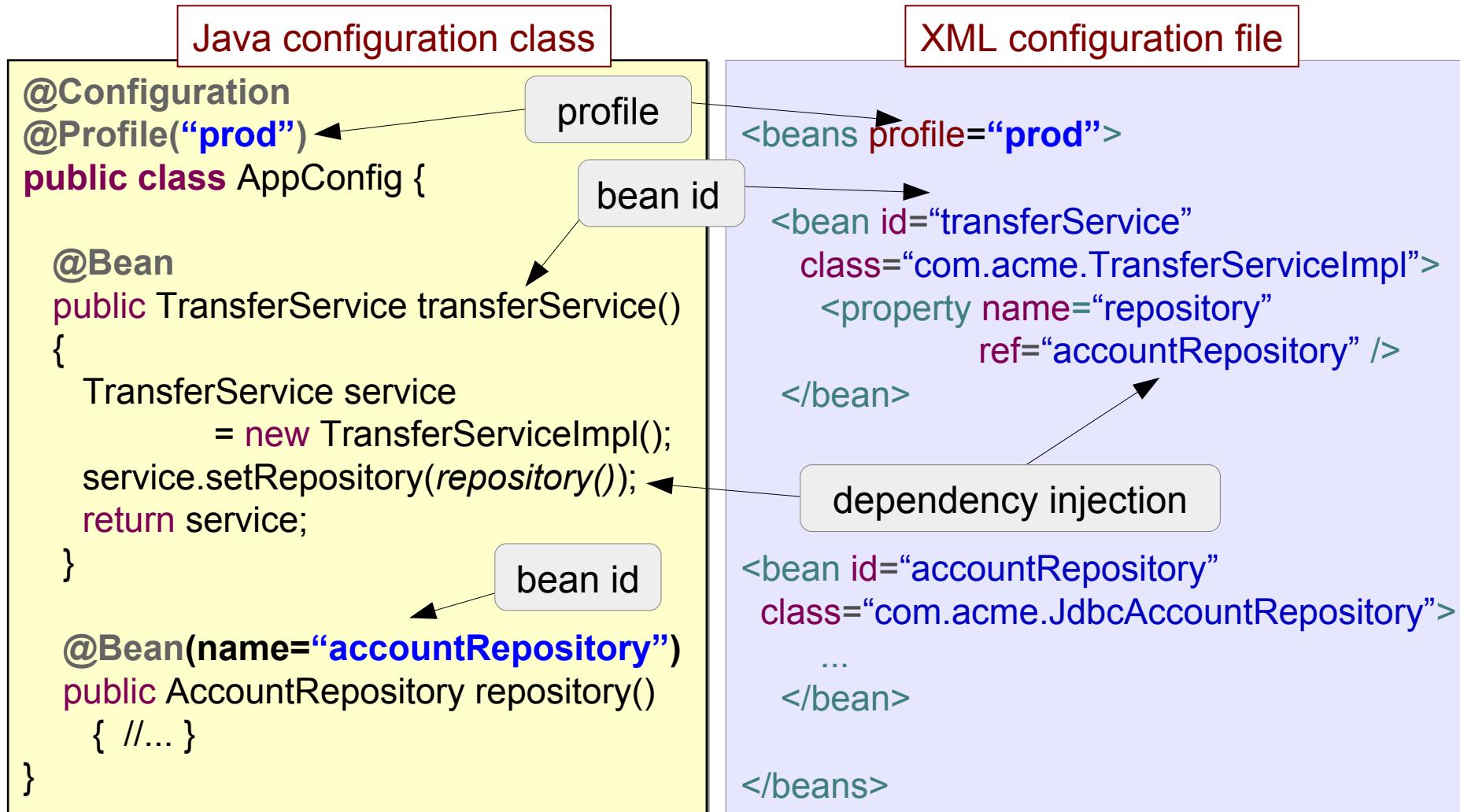
Note: XML is in the certification exam



# 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 *explicit* configuration as with Java Config
  - Uses custom XML instead of Java

# @Configuration Comparison



# 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 RepositoryImpl repository() {
    return new RepositoryImpl();
}

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

```
@Bean public ServiceImpl 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 ServiceImpl service() {
    ServiceImpl 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 ServiceImpl service() {
    ServiceImpl 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 ServiceImpl service() {
    ServiceImpl 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
- Factory Beans
- Namespaces
- Lab
- Advanced Topics

# Creating an ApplicationContext using XML

- Use a Java Configuration class
  - `@ImportResource` to define XML file(s):

```
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.
- Valid prefixes are  
classpath: (default), file:, http:

Can combine with  
`@Configuration` imports

# 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)

# Creating the ApplicationContext – I

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

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

- This is actually a Spring Boot class
  - But it works well for *any* Spring application
  - More general purpose than previous alternatives

# Creating the ApplicationContext – 2

- Older “classic” techniques available as well
  - Context type defines *where* XML files are loaded from
  - Existing code (and many online examples) do it this way

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

# Topics in this session

- Writing bean definitions
- Creating an application context
- **Controlling Bean Behavior**
- Factory Beans
- 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
  - But called “*lazy-init*”

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

# Profile Configuration in XML

- All bean definitions

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

Profile applies to all beans in the file

- 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>
```

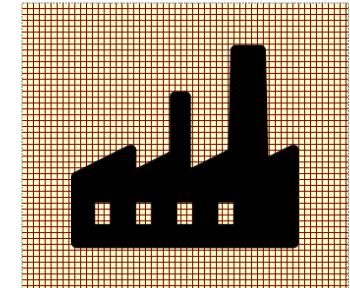
Different subset of beans for each profile, plus some *shared* beans

# Topics in this session

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



# Advanced XML Bean Instantiation



- Conditional configuration
  - @Bean methods can use *any* Java you need
    - Do property lookups
    - Use if-then-else and iterative logic
- No equivalent in XML
  - We did *not* implement <if>, <for-each>
- Instead Spring XML relies on the *Factory Pattern*
  - Use a factory to create the bean(s) we want
  - Put *any* complex Java code we need in the factory's internal logic

# The Spring FactoryBean interface



- Fall-back for complex configuration in XML
  - Used long before @Bean methods introduced

```
public class AccountServiceFactoryBean  
    implements FactoryBean <AccountService>
```

```
{
```

```
    public AccountService getObject() throws Exception {  
        // Conditional logic – for example: selecting the right  
        // implementation or sub-class of AccountService to create  
        return accountService;  
    }
```

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

```
}
```

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

**Note:** Some Java Configuration also uses factory beans

# 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 *internally*

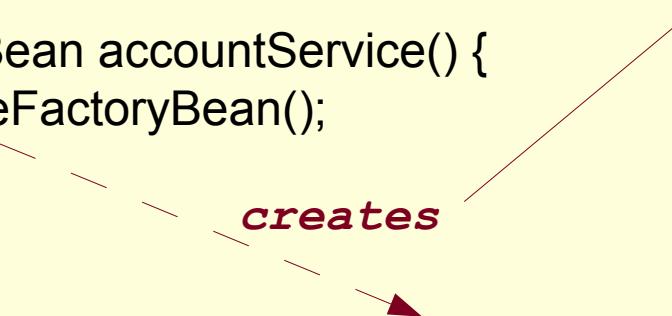
# Using FactoryBeans with Java Configuration

- Works exactly the same way

```
@Configuration  
public class ServiceConfig {  
  
    @Bean  
    public AccountServiceFactoryBean accountService() {  
        return new AccountServiceFactoryBean();  
    }  
  
    @Bean  
    public CustomerService customerService(AccountService accountService) {  
        return new CustomerService(accountService);  
    }  
}
```

getObject() called by Spring *internally*

*creates*



# FactoryBeans in Spring

- FactoryBeans are widely used within Spring
  - EmbeddedDatabaseFactoryBean\*\*
  - JndiObjectFactoryBean
    - One option for looking up JNDI objects
  - Creating Remoting proxies
  - Creating Caching proxies\*\*
  - For configuring data access technologies\*\*
    - JPA, Hibernate or MyBatis
- In XML, often hidden behind *namespaces*

\*\* These will appear later in the course

# Topics in this session

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

# 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
  - Define “programming instructions” for bean files
  - Greatly reduce size of bean files (see next slides)

# Namespace Example 1

## In-Memory DataStore using Bean XML

- Creating an in-memory test database

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.  
    embedded.EmbeddedDatabaseFactoryBean">  
    <property name="databasePopulator" ref="populator"/>  
    <property name="databaseType" ref="HSQL"/>  
</bean>
```

FactoryBean

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

```
    <property name="scripts">  
        <list>  
            <value>classpath:rewards/testdb/schema.sql</value>  
            <value>classpath:rewards/testdb/data.sql</value>  
        </list>  
    </property>  
</bean>
```

Populate with  
test-data

Bean XML requires two beans and  
knowledge of the classes being used

# Namespace Example 1

## In-Memory DataStore using jdbc Namespace

- Simplify using jdbc namespace

```
<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="classpath:rewards/testdb/schema.db"/>
    <jdbc:script location=""classpath:rewards/testdb/test-data.db"/>
</jdbc:embedded-database>
```

@Bean

```
public DataSource dataSource() {
```

```
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
```

```
    return builder.setName("testdb")
```

```
        .setType(EmbeddedDatabaseType.HSQL)
```

```
        .addScript("classpath:rewards/testdb/schema.db")
```

```
        .addScript("classpath:rewards/testdb/test-data.db").build();
```

```
}
```

Equivalent to ...

# Namespace Example 2

## Property Placeholders

- Property Placeholders define property sources
  - XML Equivalent of @PropertySource
- Namespace just an elegant way to hide the underlying bean declaration
  - Same functionality, less typing

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



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

# Accessing Properties in XML

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

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



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

db-config.properties



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

# 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" )
```

# Typical Profiles & Namespaces Example

```
<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>
```

# 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!



*Transactions and AOP will be discussed later*

# 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

# XML and Annotations

- XML supports defining annotated beans
  - Just define bean
  - Let annotations do the rest
- Must enable annotation-config

```
public class ServiceImpl {  
    @Autowired  
    public void ServiceImpl(String s) { ... }  
  
    @PostConstruct  
    public void setup() { ... }  
}
```

```
<bean id="Service" class="com.acme.ServiceImpl" />
```

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

*Or ... automatically enabled by context:component-scan*

# Namespaces Declarations are Tedious!

- What you need for beans, context and jdbc:

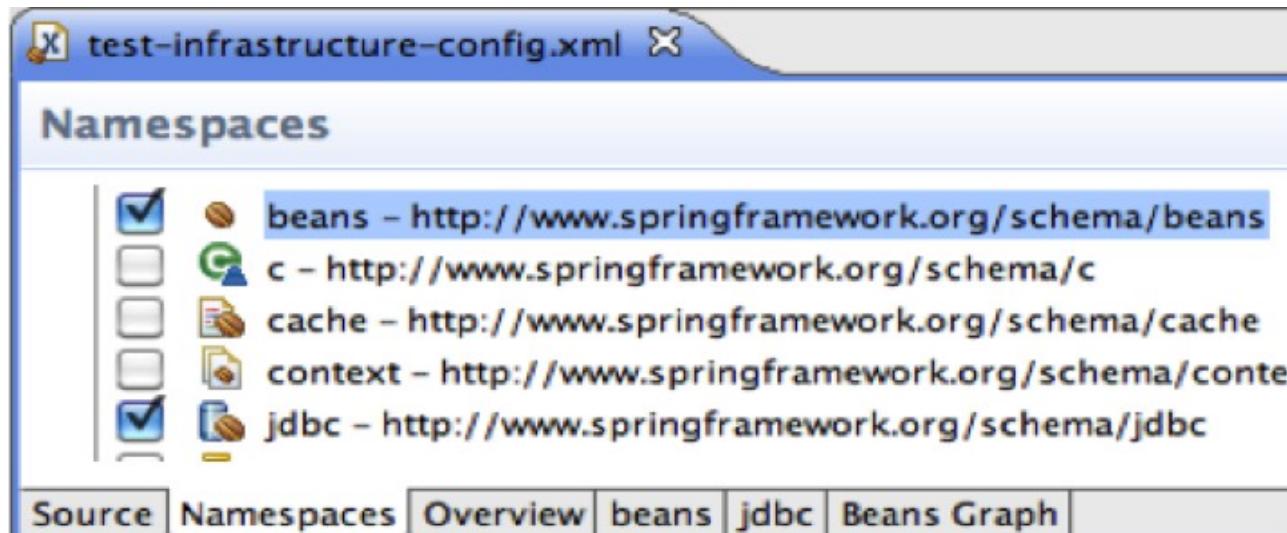
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans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

- A typo waiting to happen!
  - Fortunately there is an easier way ... (next slide)

# Adding namespace declaration

- XML syntax is error-prone
  - Use the dedicated STS XML editor Namespaces tab!

xsi:schemaLocation="..."



Click here and select appropriate namespaces

# Schema Version Numbers

spring-beans-4.2.xsd

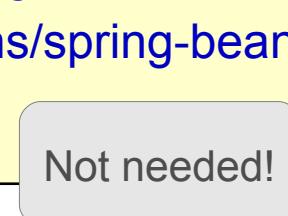
OR

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

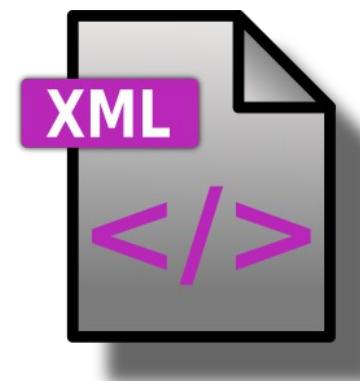
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-4.2.xsd">
    <!-- ... -->
</beans>
```



Not needed!

# Summary

- Spring's XML definition language provides *explicit* bean definitions using `<bean/>` syntax
  - Pre-dates Java Configuration
  - Provides many of the same options via attributes
- Factory Beans
  - Important configuration device
  - Understand how `getObject()` works
- Namespaces reduce verbosity, hide internal details



# Spring XML Best Practices

- XML has been around for a long time
  - Many shortcuts and useful techniques exist
    - Singleton and Factory Beans
    - Bean Definition Inheritance
    - Inner Beans
    - p and c namespaces
    - Using collections as Spring beans
- Optional Section at back of handout
  - **XML Dependency Injection Best Practices**
    - Optional lab also

# Lab (optional)

Using XML to Configure an Application

## *Optional Section*

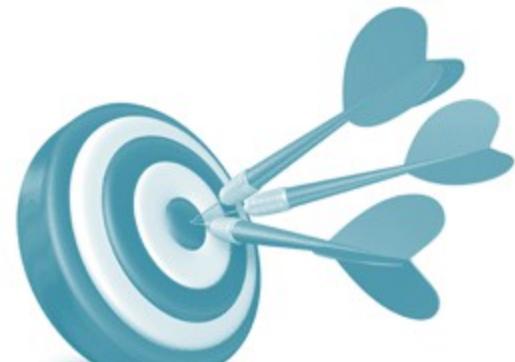
# XML Dependency Injection

## Advanced Features & Best Practices

Techniques for Creating Reusable and Concise  
Bean Definitions

# Objectives

- After completing this lesson, you should be able to:
  - Implement Singletons and Factory Beans
  - Handle Constructor Arguments
  - Use the 'p' and 'c' namespaces
  - Setup Bean definition inheritance
  - Define Inner Beans



# Topics in this session

- **Singletons and Factory Beans**
- Constructor Arguments
- 'p' and 'c' namespaces
- Using Bean definition inheritance
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections



# Advanced Bean Instantiation

- Four techniques:
  - @Bean method in @Configuration class
    - 100% Java code available, write whatever code you need
  - Beans implementing Spring's FactoryBean interface
  - Use XML factory-method attribute for Singletons
  - Define your own factories as Spring Beans in XML



# Using a Java Singleton

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

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

# The factory-method Attribute

- Non-intrusive
  - Useful for existing Singletons or Factories

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

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

*Spring configuration*

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

*Test class*

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

# Using Your Own Factories

- Spring allows one bean to create another
  - Create an instance of your factory as a bean
  - Use it to create another bean

*Spring configuration*

```
<bean id="accountServiceFactory" class="com.acme.AccountServiceFactory">  
    <!-- any constructor-arg or property elements you need -->  
</bean>  
  
<bean id="accountService" factory-bean="accountServiceFactory"  
      factory-method="create" />
```

The *class* attribute is *illegal* here  
Will be determined by the factory

# Topics in this session

- Singletons and Factory Beans
- **Constructor Arguments**
- 'p' and 'c' namespaces
- Using Bean definition inheritance
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

# 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(String username, String email) { ... }
```

Both are Strings

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

Index from zero

# Using Constructor Types

- Can also specify the type
  - Typically when class has multiple ambiguous constructors

```
class MailService {  
    public MailService(String username) { ... }  
    public MailService(int maxMessages) { ... }
```

```
<bean name="example" class="com.app.MailService">  
    <constructor-arg type="int" value="2000"/>  
</bean>
```

Is a String in XML

Force use of second constructor  
Without *type*, Spring passes “2000” as the username

# Using Constructor Names

- Constructor args can have names for matching
- Must be using Java 8 or later
  - OR: Need to compile with debug-symbols enabled
  - OR: Use **@java.beans.ConstructorProperties**

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

Specify arg  
names *in order*

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

No *index* needed

# Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- '**p**' and '**c**' namespaces
- Using Bean definition inheritance
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

# The *c* and *p* namespaces

- Before

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

- After

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

Use camel case or hyphens



c namespace is newer, 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="otherService" />
</beans>
```

Inject value for property 'url'

Inject reference to bean 'otherService'

# 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

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- **Using Bean definition inheritance**
- Inner Beans
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

# 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>
```

Overrides URL property

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

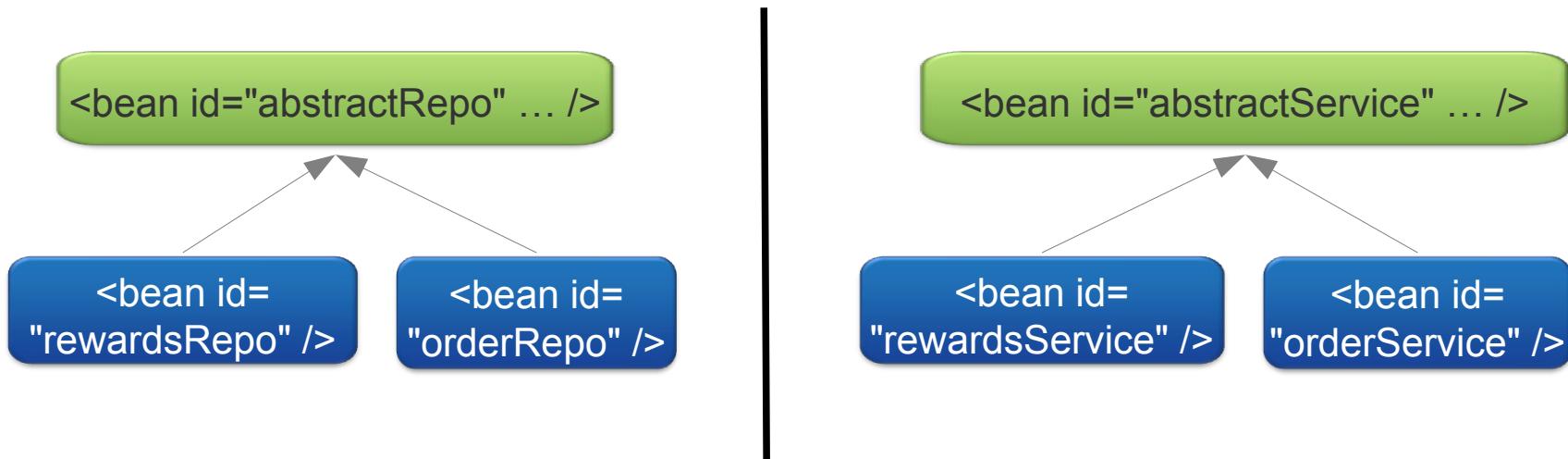
```
  <bean id="pool-C" parent="defaultPool" class="example.SomeOtherPool">
    <property name="URL"
      value="jdbc:postgresql://server-c/transfer" />
  </bean>
```

Overrides class as well

```
</beans>
```

# Inheritance for service and repository beans

- Bean inheritance commonly used for definition of Repository (or DAO) beans and (less often) Services



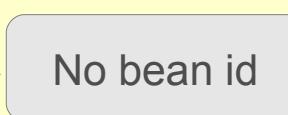
# Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- Using Bean definition inheritance
- **Inner Beans**
- Lab
- Advanced Features
  - SpEL, Autowiring, Collections

# Inner beans

- Inner bean only visible from surrounding bean

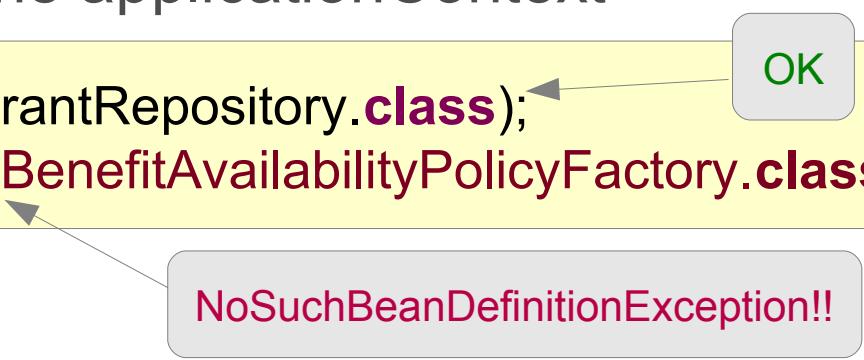
```
<bean id="restaurantRepository"
      class="rewards.internal.restaurant.JdbcRestaurantRepository">
    <property name="benefitAvailabilityPolicy">
      <bean class="rewards...DefaultBenefitAvailabilityPolicyFactory" />
    </property>
</bean>
```



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
  - Very commonly used technique in online examples
- Cons
  - Can be harder to read
  - Avoid really deep nesting
- General recommendation
  - Use them when it makes sense
    - As for inner classes in Java
    - Complex "infrastructure beans" configuration



# Lab (Optional)

Using Bean Definition Inheritance, Property  
Placeholders and Namespaces

# Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- More on Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  - SpEL, Autowiring, Collections

# Spring Expression Language

- Can also be used in bean XML files
  - Same syntax that you have seen with @Value
  - Expressions in {} preceded by #
- Recall:
  - Can access System properties and environment
  - Properties of Spring beans

# 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 – Other Spring Projects

- 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

# Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  - SpEL, Autowiring, Collections

# Autowiring in XML

- XML had automatic wiring (setting) of dependencies before @Autowired – it's where the name comes from
- Can select *byType* or *byName* or *byConstructor*
  - *Cannot autowire both properties and constructor-args*
  - *Is inherently confusing and limited due to this difference*

```
<!-- Autowire properties (setters) by type matching just like @Autowired -->
<bean id="rewardsDb" autowire="byType" ... />

<!-- Autowire properties by name – just like @Resource -->
<bean id="accountManager" autowire="byName"/ ... />

<!-- Autowire constructors only by type – just like @Autowired -->
<bean id="accountManager" autowire="byConstructor" ... />
```

# Topics in this session

- Singletons and Factory Beans
- Constructor Arguments
- 'p' and 'c' namespaces
- Profiles
- Externalizing values into properties files
- Using Bean definition inheritance
- Lab
- **Advanced Features**
  - SpEL, Autowiring, Collections

# *beans* and *util* collections

- *beans* collections
  - From the default *beans* namespace
  - Simple and easy, legacy from Spring V1
- *util* collections
  - From the *util* namespace
    - Requires additional namespace declaration
  - More features available, since Spring V2



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
  - Example: `java.util.List` implementation is always `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

value also supported

# 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>
```

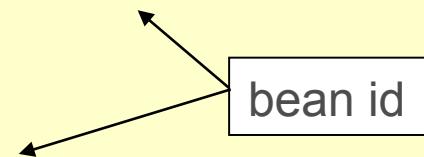
```
<property name="config">  
  <props>  
    <prop key="server.host">mailer</prop>  
    <prop key="server.port">1010</prop>  
  </props>  
</property>
```

```
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 collection elements in the beans namespace will suffice
  - But can *only* be inner beans
- Just remember the additional collection features in the **<util/>** namespace, in case you might need them
  - Declare a collection as a top-level bean
  - Specify collection implementation-type
- In the long-run, simpler to always use the **<util/>** namespace collection elements

# 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 are used widely by many existing Spring XML projects
  - Imports, Bean Inheritance, Inner Beans ...
- Other features are more specialized



# *Optional Section*

## More XML Configuration

### Useful XML Examples

Equivalent XML for the Java Configuration examples shown in this course

# Objectives

- After completing this lesson, you should be able to:
  - Configure Spring features covered on this course using XML



# Topics in this session

- AOP Configuration
- Testing Support
- Data Management and Caching
- Transactions
- JPA & Spring Data Repositories
- Spring MVC
- Spring Boot
- Spring Security

# 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" />
```

# 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" />
```

# Topics in this session

- AOP Configuration
- **Testing Support**
- Data Management and Caching
- Transactions
- JPA & Spring Data Repositories
- Spring MVC
- Spring Boot
- Spring Security

# Use @ContextConfiguration

- Tests using XML based configuration
  - JUnit 5

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

- JUnit 4

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

# @ContextConfiguration – Options

- Alternative configuration options

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration("file:system-test-config.xml")
public class TransferServiceTests { ... }
```

Local XML file

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

Multiple files

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration
public TransferServiceTests { ... }
```

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

Looks for **TransferServiceTests-context.xml**

# 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

- AOP Configuration
- Testing Support
- **Data Management and Caching**
- Transactions
- JPA & Spring Data Repositories
- Spring MVC
- Spring Boot
- Spring Security

# JDBC Namespace Equivalent

- 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>
```

Initialize in-memory  
database  
(created at startup)

# Initializing an Existing Test Database

XML provides `jdbc:initialize-database`

- Namespace supports 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  
*existing*  
database

# XML Cache Setup

```
<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="topBooks">  
        <cache:cacheable method="findBook" key="#refId"/>  
        <cache:cache-evict method="loadBooks" all-entries="true" />  
    </cache:caching>  
</cache:advice>
```

No annotations  
on this class

XML Cache Setup – no @Cachable

# Enabling Caching Proxy

- Caching must be enabled ...

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

# Topics in this session

- AOP Configuration
- Testing Support
- Data Management and Caching
- **Transactions**
- JPA & Spring Data Repositories
- Spring MVC
- Spring Boot
- Spring Security

# Deploying the Transaction Manager

- Declare as a Spring Bean

```
<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 elsewhere – easier not to!

# Automatic JTA Implementation Resolution

- For JTA, also possible to use custom XML tag:

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

- Resolves to appropriate implementation for the environment
  - `WebLogicJtaTransactionManager`
  - `WebSphereUowTransactionManager`
  - `JtaTransactionManager`

# @Transactional Configuration Using XML

- Annotate classes and methods with `@Transactional` in usual way
- Enable using `tx` namespace in the configuration:

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

Defines a Bean Post-Processor  
– proxies `@Transactional` beans

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

# 100% XML-based Spring Transactions

- `@Transactional` not always an option
  - Someone else may have written the service (without annotations)
  - Legacy code written before `@Transactional`
- Spring provides support for 100% XML
  - Predates annotations
  - 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 via transactional advice

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

# Multiple Transaction Managers

- Mark *one* as primary

Bean  
XML

```
<bean id="transactionManager" primary="true" ... > ... </bean>
```

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

# Topics in this session

- AOP Configuration
- Testing Support
- Data Management and Caching
- Transactions
- **JPA & Spring Data Repositories**
- Spring MVC
- Spring Boot
- Spring Security

# JPA Configuration

```
<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>
```

# Configuration – Spring and Persistence Unit

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="persistenceUnitName" value="rewardsNetwork"/>
</bean>
```

Minimal Spring config

```
<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>
```

Do JPA config in  
*persistence.xml*

If using JTA – declare `<jta-data-source>` in the persistence-unit

# JNDI Lookups

- A `jee:jndi-lookup` can be used to retrieve an *EntityManagerFactory* from an application server
- Use when deploying to JEE Application Servers
  - WebSphere, WebLogic, JBoss, Glassfish ...

```
<jee:jndi-lookup id="entityManagerFactory"  
                  jndi-name="persistence/rewardNetwork"/>
```

# Transparent Exception Translation (1)

- Spring provides this capability out of the box
  - JPA exceptions rethrown as **DataAccessExceptions**
  - Annotate with **@Repository**
  - Define a Spring-provided BeanPostProcessor

Annotationed

```
@Repository  
public class JpaCustomerRepository implements CustomerRepository {  
    ...  
}
```

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

# Transparent Exception Translation (2)

- Or use XML configuration:

```
public class JpaCustomerRepository implements CustomerRepository {  
    ...  
}
```

No annotations

```
<bean id="persistenceExceptionInterceptor"  
      class="org.springframework.dao.support.  
          PersistenceExceptionTranslationInterceptor"/>  
  
<aop:config>  
    <aop:advisor pointcut="execution(* *..CustomerRepository+.*(..))"  
                 advice-ref="persistenceExceptionInterceptor" />  
</aop:config>
```



# Generating Spring Data Repositories

- Spring scans for Repository<T,K> interfaces
  - Implements them and creates as a Spring bean
- Defining packages to scan

```
<jpa:repositories base-package="com.acme.**.repository" />
<mongo:repositories base-package="com.acme.**.repository" />
<gfe:repositories base-package="com.acme.**.repository" />
```

# Topics in this session

- AOP Configuration
- Testing Support
- Data Management and Caching
- Transactions
- JPA & Spring Data Repositories
- **Spring MVC**
- Spring Boot
- Spring Security

# 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*  
Using Spring for REST and Web applications

# Topics in this session

- AOP Configuration
- Testing Support
- Data Management and Caching
- Transactions
- JPA & Spring Data Repositories
- Spring MVC
- **Spring Boot**
- Spring Security



# Using XML with Spring Boot

- *@ImportResources* to import XML configuration files

```
@SpringBootApplication
@ImportResources("/org/test/myconfig.xml")
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

*application.java*

# Topics in this session

- AOP Configuration
- Testing Support
- Data Management and Caching
- Transactions
- JPA & Spring Data Repositories
- Spring MVC
- Spring Boot
- **Spring Security**

# Spring Security



- Extensive XML support
  - Using <security> namespace
  - See dedicated slide set

# Summary

- In this section we covered the use of XML for
  - AOP Configuration
  - Testing
  - Data Management and Caching
  - Transactions
  - JPA & Spring Data Repositories
  - Spring MVC
  - Spring Boot
  - Spring Security

## *Optional Section*

# Spring Security XML Configuration

Classic configuration options for  
Web Application Security

Addressing Common Security Requirements

# Objectives

- After completing this lesson, you should be able to:
  - Configure Spring Security using XML



# Spring Security – XML Configuration

- Spring Security is also configurable via XML
  - Most common in older code bases
  - Some default behaviors are different

# Configuration in web.xml

- web.xml configuration remains the same
  - springSecurityFilterChain
  - May also use Servlet 3.0 initializers

```
<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>
```

# intercept-url

- intercept-urls are evaluated in the order listed
  - first match is used, put specific matches first

```
<beans>
  <security:http>

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

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



Syntax available since Spring Security 2.0

# Security EL expressions

- `hasRole('role')`
  - Checks whether the principal has the given role
- `hasAnyRole('role1', 'role2', ...)`
  - Checks whether the principal has any of the given roles
- `isAnonymous()`
  - Allows access for unauthenticated principals
- `isAuthenticated()`
  - Allows access for authenticated or remembered principals



Available from Spring Security 3.0  
Previous syntax still works in Spring Security 3.0

# Intercept-url and Expression Language

- Expression Language provides more flexibility
  - Many built-in expressions available

*Spring configuration file*

```
<beans>
  <security:http use-expressions="true"> ← Expression Language needs
    <security:intercept-url pattern="/accounts/edit*"      to be enabled explicitly
      access="hasRole('ROLE_ADMIN')"/>
    <security:intercept-url pattern="/accounts/account*" 
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_USER')"/>
    <security:intercept-url pattern="/accounts/**"
      access="isAuthenticated() and hasIpAddress('192.168.1.0/24')"/>

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

 Syntax available from Spring Security 3.0

# Working with roles

- Checking if the user has one single role

```
<security:intercept-url pattern="/accounts/update*" access="hasRole('ROLE_ADMIN')"/>
```

- “or” clause

```
<security:intercept-url pattern="/accounts/update*"  
access="hasAnyRole('ROLE_ADMIN', 'ROLE_MANAGER')"/>
```

- “and” clause

```
<security:intercept-url pattern="/accounts/update*"  
access="hasRole('ROLE_ADMIN') and hasRole('ROLE_MANAGER')"/>
```

- Previous and new syntax can't be mixed

```
<security:intercept-url pattern="/accounts/update*"  
access="hasRole('ROLE_MANAGER')"/>  
<security:intercept-url pattern="/accounts/update*" access="ROLE_ADMIN"/>
```

Not correct!!

# Specifying login and logout

```
<beans ...>
  <security:http pattern="/accounts/login" security="none"/>

  <security:http use-expressions="true">
    <security:form-login login-page="/accounts/login"
      default-target-url="/accounts/home"/>

    <security:intercept-url pattern="/accounts/update*"
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_MANAGER') />

    <security:intercept-url pattern="/accounts/**"
      access="hasRole('ROLE_ADMIN') />

    <security:logout logout-success-url="/home.html"/>
  </security:http>
  ...

```

Exempt login page  
(Spring Security 3.1)

Specify login options

Must be declared explicitly  
or no logout possible

*Spring XML configuration file*

# Setting up User Login

- Default auth. provider assumes form-based login
  - This is *web* security after all
  - *Must* specify form-login element
  - A basic form is provided
  - Configure to use your own login-page

```
<security:http>
  <security:form-login/>
  ...
</security:http>

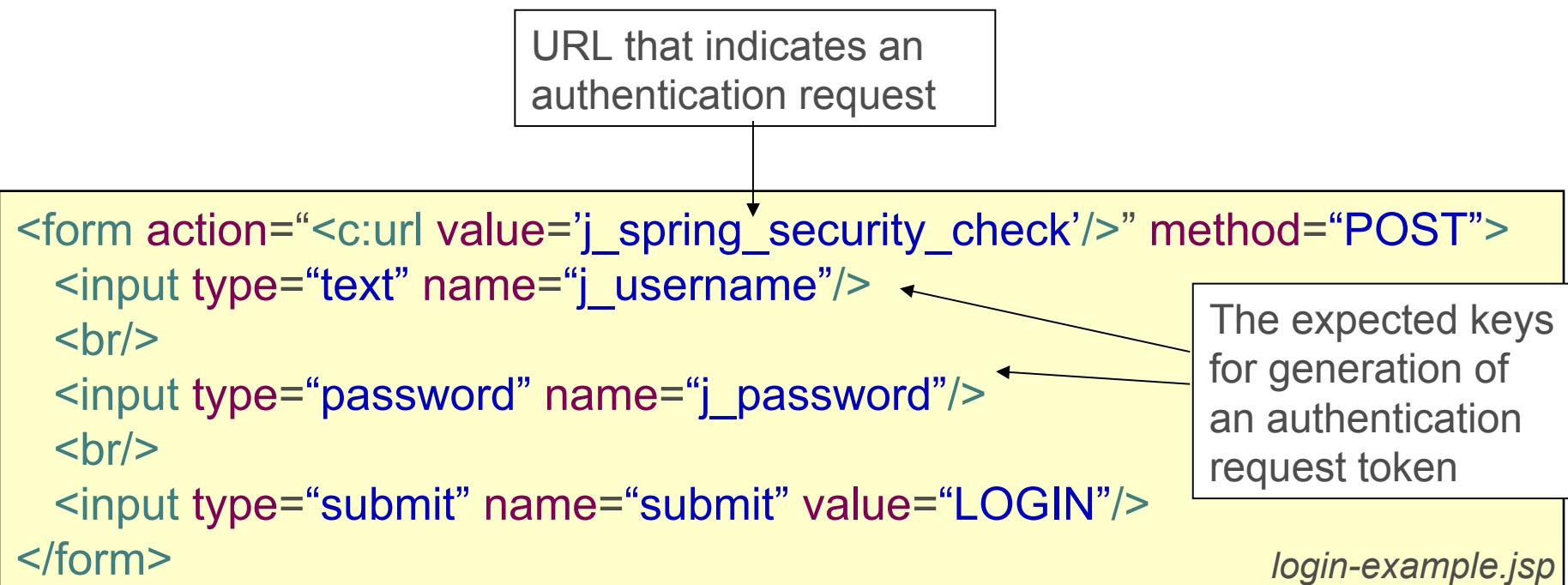
<security:authentication-manager>
  <security:authentication-provider>
  ...
  </security:authentication-provider>
<security:authentication-manager>
```

**Login with Username and Password**

User:

Password:

# An Example Login Page



Above example shows default values (*j\_spring\_security\_check*, *j\_username*, *j\_password*). All of them can be redefined using `<security:form-login/>`

# The In-Memory User Service

- Useful for development and testing
  - Note: must restart system to reload properties

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

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

List of roles separated by commas

login

password

# The JDBC user service (2/2)

- Configuration:

```
<beans>
  <security:http> ... <security:http>

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

*Spring configuration file*

Can customize queries using attributes:  
**users-by-username-query**  
**authorities-by-username-query**  
**groupAuthorities-by-username-query**

# Password Encoding

- Can encode passwords using a hash
  - sha, md5, ...

```
<security:authentication-provider>
    <security:password-encoder hash="sha-256" /> ← simple encoding
        <security:user-service properties="/WEB-INF/users.properties" />
    </security:password-encoder>
</security:authentication-provider>
```

- Secure passwords using a well-known string
  - Known as a 'salt', makes brute force attacks harder

```
<security:authentication-provider>
    <security:password-encoder hash="sha-256"> ← encoding with salt
        <security:salt-source system-wide="MySalt" />
    </security:password-encoder>
        <security:user-service properties="/WEB-INF/users.properties" />
    </security:password-encoder>
</security:authentication-provider>
```

# Method Security using XML

- Can apply security to multiple beans with only a simple declaration

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

*Spring configuration file*



*Spring Security 2 syntax only. SpEL not supported here.*

# Custom Filter Chain

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

```
<security:http>
    <security:custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>

<bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter"/>
```

- Filter can be **added** to the chain

```
<security:http>
    <security:custom-filter after="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>

<bean id="myFilter" class="com.mycompany.MySpecialFilter"/>
```

# Summary

- Spring Security supports XML configuration
  - Common in existing applications
- Support for:
  - URL Authorization Restrictions
  - Authentication Mechanisms
  - Java Method Authorization
  - Filter Customization

