



Enterprise Integration With Spring





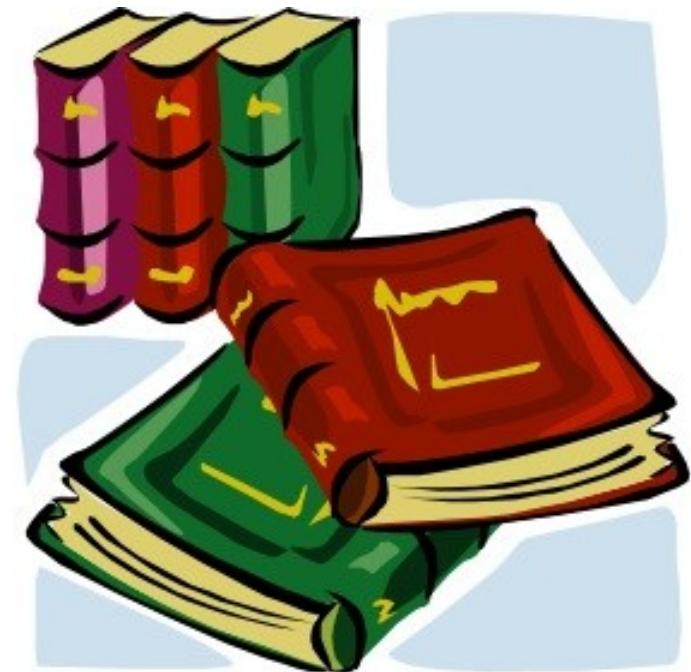
Welcome to Enterprise Integration with Spring

A 4-day, hands-on course that teaches you how
to use Spring to integrate business applications

Enterprise Integration, defined



“The goal of Enterprise Integration is to connect multiple enterprise systems with heterogeneous data formats to support business use cases”



Examples:

- Connect a new order processing system to a legacy billing system
- Consolidate two inventory systems after a company merger
- Import nightly batch of transaction data from credit card processor

The Goal of Spring in Enterprise Integration



- Spring's goal is to provide comprehensive infrastructural support for enterprise integration
 - Spring handles the “plumbing” of integration
 - So you can focus on domain logic / business use cases
- This course will cover multiple Spring projects
 - Spring Framework (Spring Core)
 - Spring Web Services
 - Spring Security
 - Spring Integration
 - Spring Batch

Format and Materials



- Course is 50% theory, 50% lab work
 - Theory covers integration concepts and how to use Spring
 - Labs provide hands-on experience
- USB keys are *yours to keep* and contain
 - State-of-the-art lab environment based on
 - SpringSource Tool Suite (STS)
 - Apache Tomcat
 - Maven
 - Lab sources, documentation and dependencies
 - PDF version of presentation slides

Approach and Philosophy



- Focus on real-world problems and solutions
 - All labs are based on realistic integration use cases
- Testing matters
 - All labs are test-driven using JUnit and Mockito
- Keep it simple
 - Integration is inherently complex; this course will show you how to keep it as simple as possible

Agenda: Day 1



- Introductions and Getting Started
- Integration Styles (*presentation only*)
- Tasks and Scheduling
- Remoting
- SOAP web services

Agenda: Day 2



- Advanced Spring Web Services
- RESTful web services
- Introduction to Messaging (*presentation only*)
- Working with JMS
- Transactional JMS

Agenda: Day 3



- Global transaction management (XA and JTA)
- Introduction to Spring Integration
- Configuring Spring Integration
- Spring Integration Advanced Features

Agenda: Day 4



- Introduction to Spring Batch
- Restart and Recovery With Spring Batch
- Spring Batch Admin and Scaling Batch Jobs

Prerequisites: Java



- Strong ability with language
 - Advanced syntax such as anonymous classes
- Understanding of Collections API
 - List, Set, Map, etc
- Annotations
 - Basic familiarity is OK
 - Helps to understand how to author your own
- Generics
 - Generics syntax will be used throughout the course

Prerequisites: Concepts



- TDD (“*Test-driven development*”)
 - Understanding the value of developer testing
 - Experience with JUnit 4
- Dependency Injection / Inversion of Control
- POJO Programming

Prerequisites: *Spring*



- Experience with <beans/> XML
- Familiarity with Spring XML namespaces
 - context:, tx:, aop:, etc.
- Familiarity with “annotation-driven” injection
 - @Autowired
 - <context:component-scan/>
- Familiarity with Spring's transaction management
 - @Transactional
 - <tx:annotation-driven/>

Prerequisites: *Tools*



- Basic experience with Eclipse
 - Navigating views and perspectives
 - Using content-assist (ctrl+space)
 - Using quick-fix (ctrl+1)
 - Running code
 - on Server (via WTP)
 - as JUnit Test

Prerequisites: *Java EE*



- Basic experience with Servlet container
 - Working with Tomcat
 - Configuring web.xml
 - <servlet>, <listener>, <filter>, <context-param>

How to get the most value



- Ask questions!
 - During presentations
 - *Especially* during labs
- Team up
 - Consider working the labs with a partner

Logistics



- Hours
- Lunch and breaks
- Restrooms, parking, etc.
- Other questions?

Feedback!

- We value your input
 - Evaluation instructions provided at end of course
- Send any additional feedback or questions to
 - training@springsource.com



Lab: ei-course-intro

Getting to know the reference domain and
courseware environment



Styles of Enterprise Integration

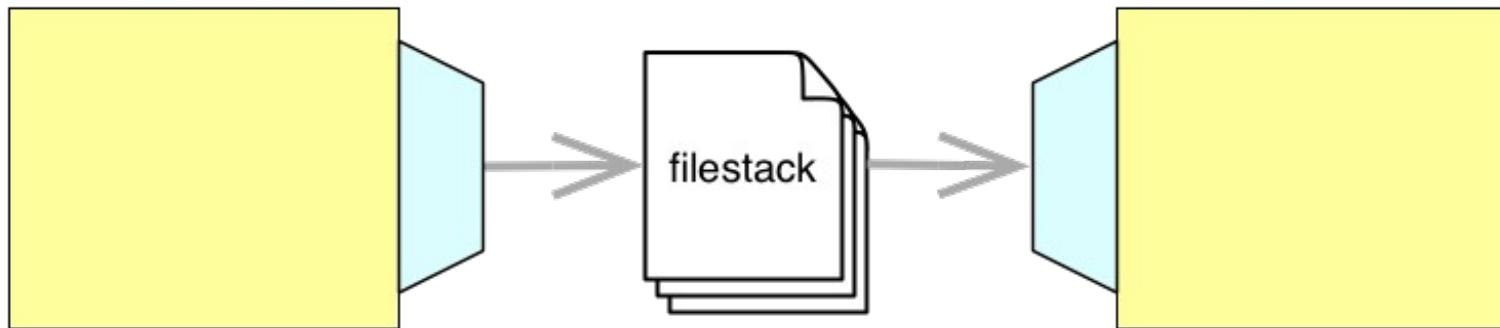
Topics in this session

- Introduction
- Integration Styles
 - File Transfer
 - Shared Database
 - Remoting
 - Messaging
- For Each Style:
 - Pros/Cons
 - Spring Support

Introduction

- Integrating Enterprise Applications can be done in many ways
- Each way has its own pros and cons
- Best solution depends on requirements
- Things to consider:
 - Coupling (logical, temporal)
 - Synchronous or asynchronous
 - Overhead
 - Data formats
 - Reliability

File Transfer: Overview



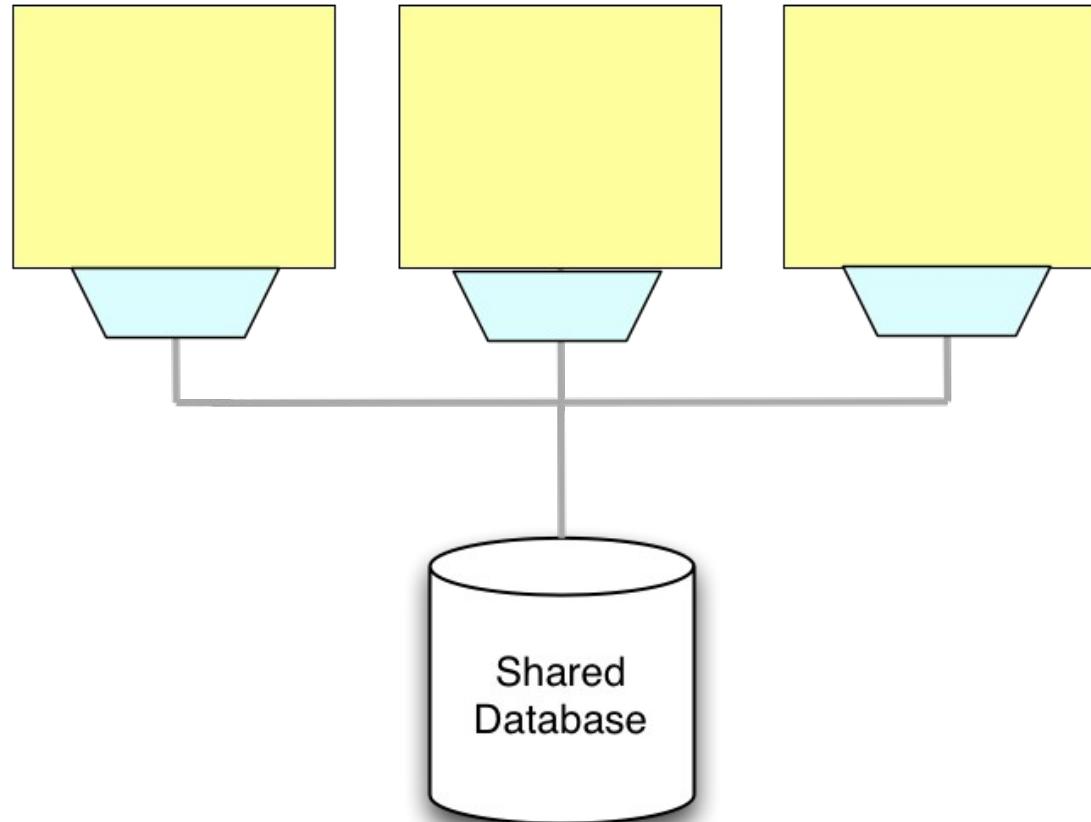
File Transfer: *Tradeoffs*



- Pros
 - Simple
 - Interoperable
 - Fast
- Cons
 - Unsafe
 - Non transactional
 - Concurrency issues
 - Security
 - Platform dependent
 - Not event-driven

- Resource API
 - Helps to simplify common I/O operations
- Spring Batch
 - Parsing of CSV- and XML-files
 - Stateful streaming of large files in and out of the app
- Spring Integration
 - Monitoring directories for new files
 - Working with (S)FTP and FTPS
 - Experimental: receiving file system notifications

Shared Database: Overview



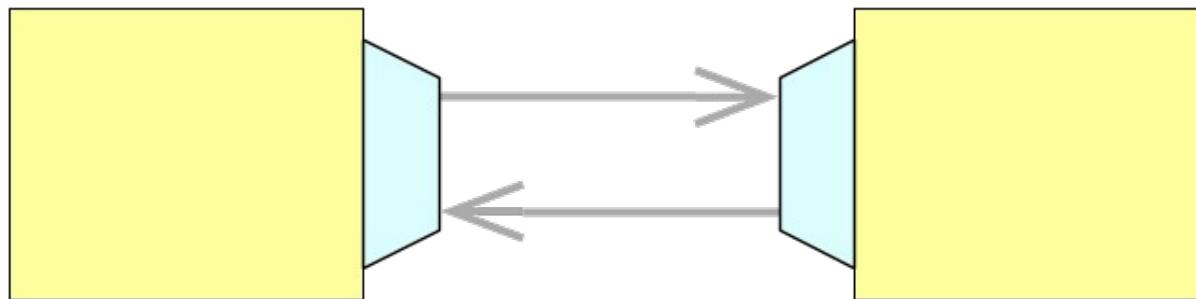
Shared Database: *Tradeoffs*



- Pros
 - Simple
 - Transactional
 - Triggers
 - But non-portable
- Cons
 - Slower
 - Impedes schema evolution
 - Less so with NoSQL DBs

- `DataAccessException` Hierarchy
- Transaction Management
- Spring JDBC
- Spring-ORM integration
 - Hibernate, JPA, iBatis
- Spring-Data umbrella project
 - Dynamic JPA repository generation
 - Support for various NoSQL solutions
 - MongoDB, Redis, Riak, Neo4J, etc.

Remoting: Overview



Remoting: *Tradeoffs*



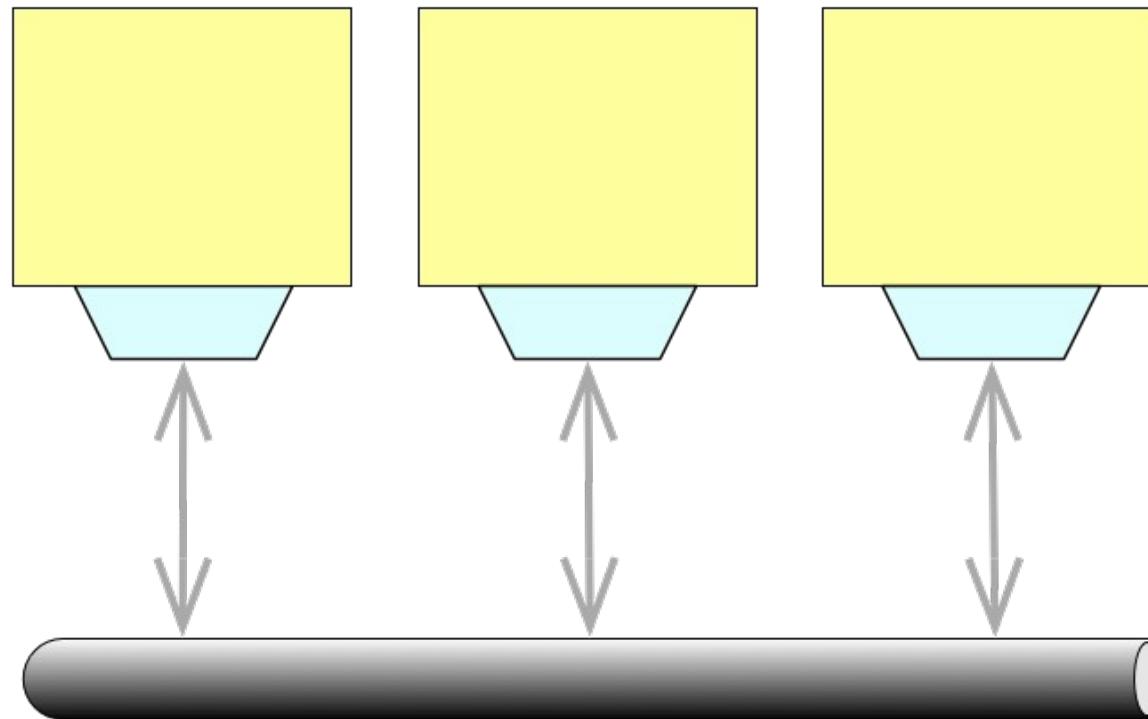
- Pros
 - Convenient
 - Stick with OO paradigm
 - Speed
- Cons
 - Not interoperable
 - Hard to version
 - Not scalable
 - Hidden complexity

Remoting: *Spring Support*



- *ProxyFactoryBean* and *Exporter* implementations for
 - RMI
 - HttpInvoker
 - Hessian / Burlap

Messaging: Overview



Messaging: *Tradeoffs*



- Pros
 - Asynchronous
 - Efficient
 - Scalable
 - Extensible
- Cons
 - Complexity
 - Longer response times
 - Loss of Transaction context
 - Loss of Security context

Messaging: *Spring Support*



- Spring JMS
 - Sending and receiving JMS messages
 - Transaction support
- Spring AMQP
 - AMQP messaging support, similar to JMS support
- Spring Web Services
 - Contract first web services
- Spring Integration
 - Supports Hohpe's *Enterprise Integration Patterns*
 - Builds on top of other Spring support for integration

What about Web Services?



- Web services can be designed to exhibit characteristics of Remoting or Messaging
 - Remoting
 - RPC/literal SOAP messages
 - XML-RPC
 - Some of the WS-* standards
 - Messaging
 - Document/literal SOAP messages
 - POX, REST
 - “Contract-first” web services
- This training focuses on loosely-coupled, contract-first, messaging-style web services



Tasks and Scheduling

Topics in this presentation



- **Introduction to concurrency**
- Java Concurrency support
- Spring's Task Scheduling support

Concurrency



- Also known as parallel processing or Multithreading / multitasking
- Allows tasks to be run simultaneously
 - Useful when applications are I/O-bound
 - incl. servers serving multiple clients
 - Maximum advantage of multicore/-processor systems
- A built-in feature of the Java language and platform
 - Unlike, for example, C/C++

A simple task



Download 100 photos from flickr.com

photos.txt

```
http://farm4.static.flickr.com/3267/2803733448_ea757aab49.jpg
http://farm3.static.flickr.com/2160/2616535176_ac7db42a78.jpg
http://farm4.static.flickr.com/3068/2545650273_6c1b07d4c3.jpg
http://farm4.static.flickr.com/3024/2530880977_ff6cf5012.jpg
```

...

Serial processing



```
public static void main(String[] args) throws IOException {
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(new FileInputStream("c:\\photos.txt")));
    String tmpDir = "c:\\temp";
    String photoUrl = null;
    while ((photoUrl = reader.readLine()) != null) {
        DownloadUtils.download(photoUrl, tmpDir);
    }
}
```

Too slow? Might be better to download all 100 photos *simultaneously*

Task Scheduling

- Another form of concurrency
- Schedule piece of work to run once or repeatedly
- Examples: poll for new email every second, start nightly batch job every working day, etc.

Topics in this presentation



- Introduction to concurrency
- **Java Concurrency support**
- Spring's Task Scheduling support

Java Concurrency Support



- Thread is basic concurrency building block Java
- Executes instance of Runnable interface
- Multiple threads can run in parallel
- Using private data or accessing shared memory
- Sharing requires synchronization when data is changed

Create an implementation of java.lang.Runnable



```
public class DownloadTask implements Runnable {  
    private final String url;  
    private final String tmpDir;  
    public DownloadTask(String url, String tmpDir) {  
        this.url = url;  
        this.tmpDir = tmpDir;  
    }  
    @Override  
    public void run() {  
        try {  
            DownloadUtils.download(url, tmpDir);  
        } catch (IOException ex) { ex.printStackTrace(System.err); }  
    }  
}
```

- Shouldn't typically use Threads directly
 - Thread management is low-level cross-cutting concern
- Java 5 and 6 provide higher-level language support
 - Transparently use thread pools, task queues, etc.
- All support is under `java.util.concurrent.*`
 - Concurrent collections, atomic wrapper types, thread pools, task types, executor framework, etc.

- Simple abstraction for executing tasks
 - Hides the details of low-level thread management
 - Some implementations may execute tasks in a separate thread
 - Other implementations execute serially

```
public interface void Executor {  
    void execute(Runnable task);  
}
```

- Spring has similar TaskExecutor interface
 - Pre-dates Java 5
 - Extends java.util.concurrent.Executor since Spring 3.0

- Extends Executor, adding lifecycle methods
 - shutdown(): accept no more new tasks
 - awaitTermination(): block until all tasks are complete

```
public interface ExecutorService extends Executor {  
  
    void shutdown();  
    boolean awaitTermination(long amount, TimeUnit unit)  
        throws InterruptedException;  
    // ...  
}
```

- Provides factory methods for commonly used ExecutorService implementations

```
public class Executors {  
    public static ExecutorService newFixedThreadPool(int size);  
    public static ExecutorService newSingleThreadExecutor();  
    public static ExecutorService newCachedThreadPool();  
    // ...  
}
```

Putting it all together



```
public class PhotoDownloader {  
    public PhotoDownloader(List<String> urls, String tmpDir) { ... }  
  
    public void download() {  
        ExecutorService service = getExecutorService();  
        for (String photoUrl : urls) {  
            service.execute(new DownloadTask(photoUrl, tmpDir));  
        }  
        service.shutdown();  
        service.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);  
    }  
  
    ExecutorService getExecutorService() {  
        return Executors.newCachedThreadPool();  
    }  
}
```

Callable

- For tasks that return a result and/or throw exceptions, Java 5 introduced Callable

```
public class NextPrimeNumberFinder implements Callable<Integer> {  
    private int number;  
  
    public NextPrimeNumberFinder(int number) { this.number = number; }  
  
    @Override  
    public Integer call() throws Exception {  
        for (;;) {  
            if (isPrime(++number)) return number;  
        }  
    }  
}
```

Future

- ExecutorService can schedule Callable and return a **Future**
 - Placeholder that contains result once it's available
 - Allows to check if result is there, cancel the task, or block and wait (with optional timeout) for the result

```
Future<Integer> futurePrime =  
    executorService.submit(new NextPrimeNumberFinder(10000));  
// do some other work until we need the result...  
// now block and wait for the result  
try {  
    Integer nextPrime = futurePrime.get();  
} catch (ExecutionException e) {  
    System.err.println("NextPrimeNumberFinder threw exception: " + e.getCause());  
}
```

ScheduledFuture



- ScheduledFuture is Future with a delay property
 - Useful for scheduling tasks with delay or repeatedly
 - Supported through ScheduledExecutorService
 - And.... by Spring!

```
ScheduledExecutorService scheduledExecutorService =
```

```
    new ScheduledThreadPoolExecutor(10);
```

```
// start the finding of the next prime in 10 seconds
```

```
ScheduledFuture<Integer> scheduledPrime =
```

```
    scheduledExecutorService.schedule(new NextPrimeNumberFinder(1000),  
                                    10, TimeUnit.SECONDS);
```

```
// run the download task repeatedly every second with initial delay of 0
```

```
scheduledExecutorService.scheduleWithFixedDelay(
```

```
    new DownloadTask(photoUrl, tmpDir), 0, 1, TimeUnit.SECONDS);
```

Topics in this presentation



- Introduction to concurrency
- Java Concurrency support
- **Spring's Task Scheduling support**

org.springframework.scheduling. TaskScheduler



```
public interface TaskScheduler {  
    ScheduledFuture schedule(Runnable task, Date startTime);  
    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);  
    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);  
    ScheduledFuture schedule(Runnable task, Trigger trigger);  
    // ...  
}
```

```
ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();  
// ...  
scheduler.scheduleWithFixedDelay(task, 5000);  
scheduler.shutdown();
```

```
public interface Trigger {  
    Date nextExecutionTime(TriggerContext triggerContext);  
}
```

```
public class CronTrigger implements Trigger {  
    public CronTrigger(String cronExpression) { ... }  
}
```

```
ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();  
// ...  
scheduler.schedule(task, new CronTrigger("*/5 * 9-17 * * MON-FRI"));  
scheduler.shutdown();
```

Calls the task every five seconds
between the hours of 9am and
5pm on weekdays

Introducing the Spring 3 <task:> namespace



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

    <!-- add <task:> elements -->

</beans>
```

Introducing the Spring 3 <task:> namespace



- Provides a simple, high-level abstraction over TaskScheduler, Trigger and other elements of the Spring scheduling API
 - Define a ThreadPoolTaskExecutor and/or ~Scheduler
 - Schedule tasks
 - Configure annotation-driven scheduling and asynchronous method execution

Spring 3 <task:> namespace



```
<beans ...>

    <bean id="photoDownloader">
        <!-- constructor args: urls, tmpDir -->
    </bean>

    <task:scheduler id="scheduler" pool-size="5"/>

    <task:scheduled-tasks scheduler="scheduler">
        <task:scheduled ref="photoDownloader"
                        method="download" fixed-delay="1000"/>
    </task:scheduled-tasks>

</beans>
```

Creates a ThreadPoolTaskScheduler

Calls the download() method every second

Cron expressions with the <task:> namespace



scheduler uses single-threaded
Executor by default

```
<task:scheduled-tasks>
  <task:scheduled ref="photoDownloader" method="download"
    cron="*/5 * 9-17 * * MON-FRI"/>
</task:scheduled-tasks>
```

Annotation-based Configuration



- Can also annotate methods instead of using XML

```
<task:scheduler id="scheduler" pool-size="5"/>
<task:annotation-driven scheduler="scheduler"/>
...
...
```

```
public class PhotoDownloader {
    public PhotoDownloader(List<String> urls, String tmpDir) { ... }

    @Scheduled(cron="*/5 * 9-17 * * MON-FRI")
    public void download() {
        ...
    }
}
```

- @Async annotated methods will run asynchronously (in a separate thread)
 - So they return immediately!
- Have to return void or a Future
 - In Future case, just wrap result in AsyncResult to keep compiler happy

```
@Async
```

```
public Future<File> download(String url, File directory) {  
    File file = DownloadUtils.downloadTo(url, directory);  
    return new AsyncResult(file);  
}
```

Running @Async Methods



- By default SimpleAsyncTaskExecutor is used
 - Creates new Thread for every task
- Override using the executor attribute
- Use namespace to define ThreadPoolTaskExecutor

```
<task:annotation-driven executor="executor"/>  
  
<task:executor id="executor" pool-size="5-10" queue-capacity="50" />
```

- Create up till 5 threads for new tasks
- When 5 threads are in use, start to queue tasks
- When queue is full, allocate up to 10 threads max
- Use plain bean definition for other TaskExecutors

- Java EE apps should not create their own threads
- Integrate with app-server managed threads through CommonJ (WebSphere & WebLogic)
 - org.sfw.scheduling.commonj.WorkManagerTaskExecutor
- Or through JCA
 - org.sfw.jca.work.glassfish.GlassFishWorkManagerTaskExecutor
 - org.sfw.jca.work.jboss.JBossWorkManagerTaskExecutor
 - org.sfw.jca.work.WorkManagerTaskExecutor for other app servers
- Configuration done in app server, not in Spring



Lab: tasks-1



Introduction to Spring Remoting

Simplifying Distributed Applications

Topics in this Session



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

Goals of Spring Remoting



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

The Problem with Plumbing Code



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

Violates a separation of concerns
Couples business logic to remoting infrastructure

Hiding the Plumbing with Spring



- Spring provides **exporters** to handle server-side requirements
 - Binding to registry or exposing an endpoint
 - Conforming to a programming model if necessary
- Spring provides FactoryBeans that generate **proxies** to handle client-side requirements
 - Communicate with the server-side endpoint
 - Convert remote exceptions to a runtime hierarchy

The Declarative Approach



- Spring's abstraction uses a configuration-based approach
- On the server side
 - Expose existing services with NO code changes
- On the client side
 - Invoke remote methods from existing code
 - Take advantage of polymorphism by using dependency injection

Consistency across Protocols



- Spring's exporters and proxy FactoryBeans bring the same approach to multiple protocols
 - Provides flexibility
 - Promotes ease of adoption
- On the server side
 - Expose a single service over multiple protocols
- On the client side
 - Switch easily between protocols
 - Migrate between remote vs. local deployments

Topics in this Session

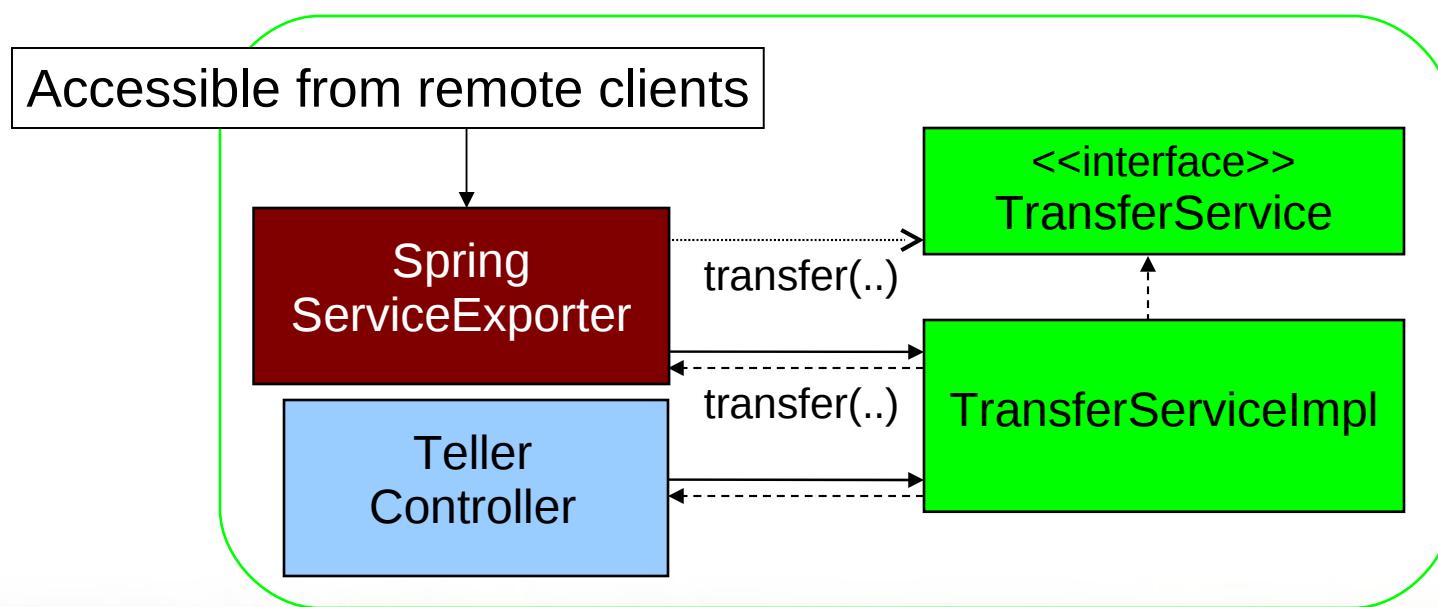


- Goals of Spring Remoting
- **Spring Remoting Overview**
- Supported Protocols
 - RMI
 - HttpInvoker
 - Hessian/Burlap

Service Exporters



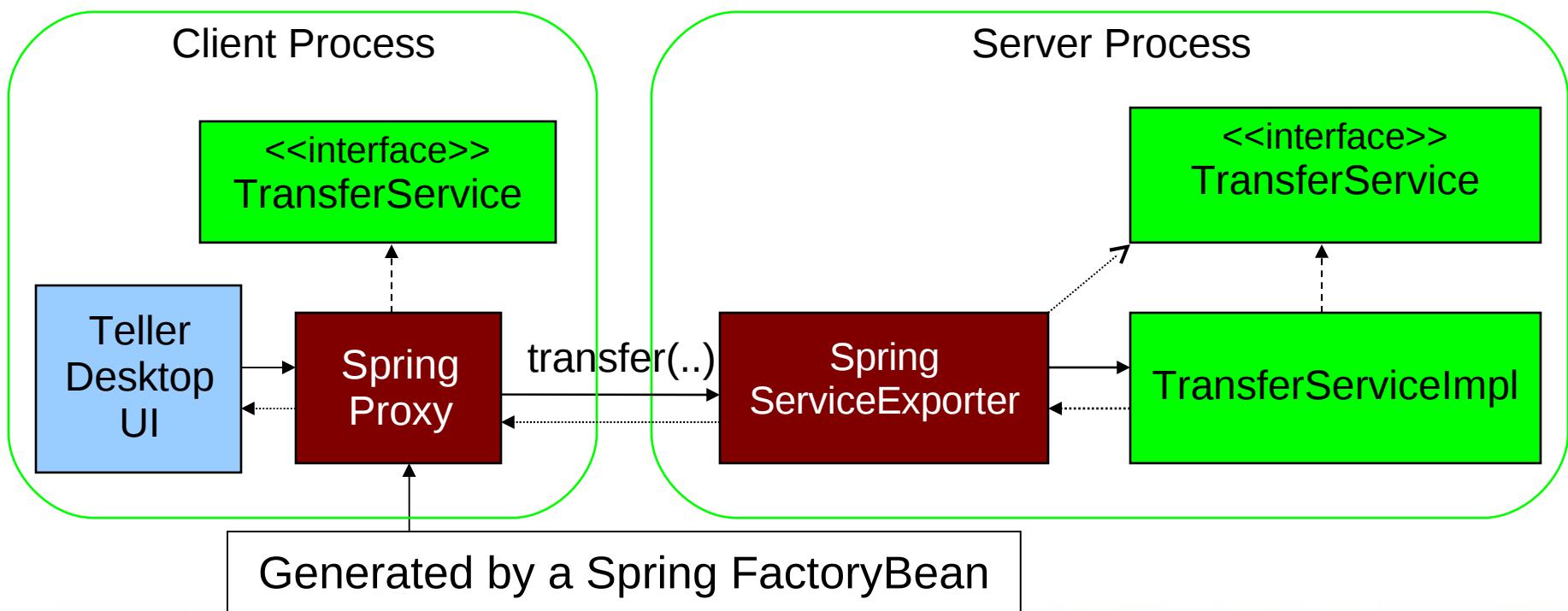
- Spring provides service exporters to enable declarative exposing of existing services



Client Proxies



- Dynamic proxies generated by Spring communicate with the service exporter



Topics in this Session



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

The RMI Protocol

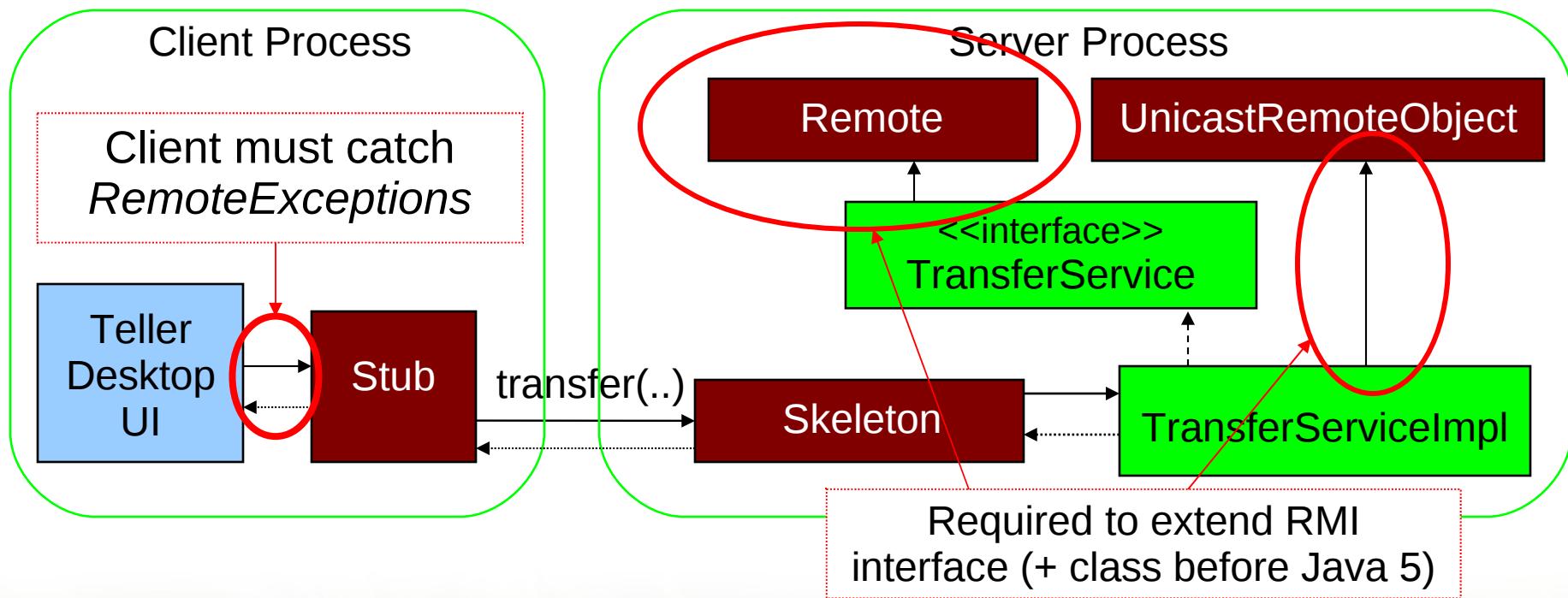


- Standard Java remoting protocol
- Server-side exposes a *skeleton*
- Client-side invokes methods on a *stub* (proxy)
- Java serialization is used for marshalling

Traditional RMI



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



Spring's RMI Service Exporter



- Transparently expose an existing POJO service to the RMI registry
 - No need to write the binding code
- Avoid traditional RMI requirements
 - Service interface does not extend **Remote**
 - Service class is a POJO
- Looks up existing RMI registry, creating it if not found

Configuring the RMI Service Exporter



- Start with an existing POJO service

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

- Define a bean to export it

Binds to rmiRegistry as “transferService”

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

Can also specify ‘registryPort’ (default is 1099)

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

Spring's RMI Proxy Generator



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

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

Configuring the RMI Proxy



- Define a factory bean to generate the proxy

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

- Inject it

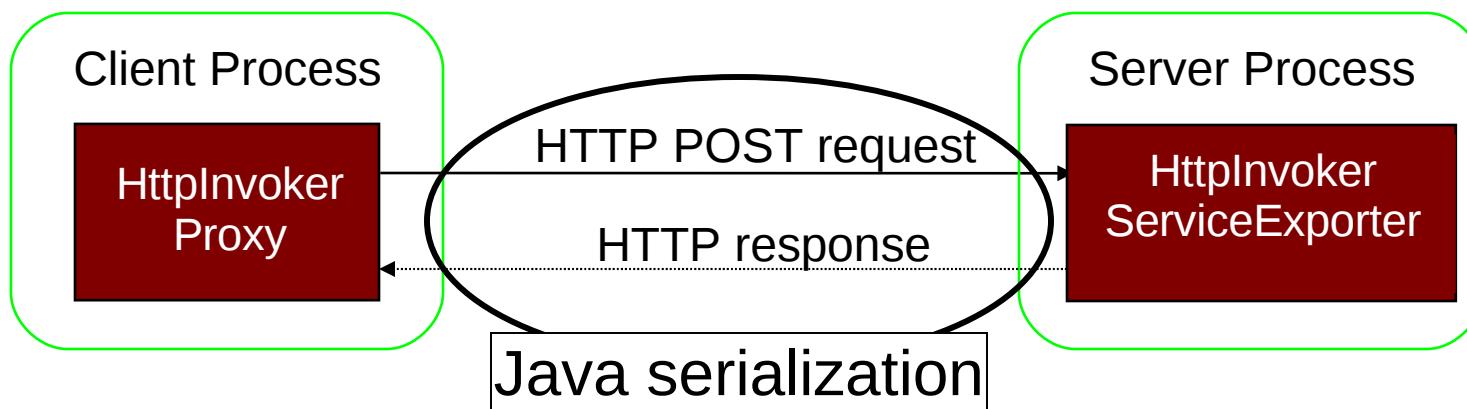
TellerDesktopUI only depends on the TransferService interface

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

Spring's HttpInvoker



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



Configuring the HttpInvoker Service Exporter (1)



- Start with an existing POJO service

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

- Define a bean to export it

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

Configuring the HttpInvoker Service Exporter (2)



- To expose the HTTP service, define either a DispatcherServlet or a HttpRequestHandlerServlet
 - Rely on BeanNameUrlHandlerMapping to map requests to service with DispatcherServlet (hence "/transfer")
 - HttpRequestHandlerServlet doesn't require Spring-MVC, so service exporter bean is defined in root context

```
<servlet>
  <servlet-name>transfer</servlet-name>
  <servlet-class>org.sfw...HttpRequestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>transfer</servlet-name>
  <url-pattern>/rewards/transfer</url-pattern>
</servlet-mapping>
```

service exporter bean name

Configuring the HttpInvoker Proxy



- Define a factory bean to generate the proxy

```
<bean id="transferService"
      class="org.springframework.remoting.httpinvoker.
          HttpInvokerProxyFactoryBean">
    <property name="serviceInterface" value="foo.TransferService"/>
    <property name="serviceUrl" value="http://foo:8080/services/transfer"/>
</bean>
```

HTTP POST requests will be sent to this URL

- Inject it into the client

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

Hessian and Burlap



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

Hessian and Burlap Configuration



- Service exporter configuration is identical to `HttpInvokerServiceExporter` except class names
 - `org.springframework.remoting.caucho.HessianServiceExporter`
 - `org.springframework.remoting.caucho.BurlapServiceExporter`
 - Same web setup required as well
- Proxy configuration is identical to `HttpInvokerProxyFactoryBean` except class names
 - `org.springframework.remoting.caucho.HessianProxyFactoryBean`
 - `org.springframework.remoting.caucho.BurlapProxyFactoryBean`

Choosing a Remoting Protocol (1)



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

Choosing a Remoting Protocol (2)



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



LAB

Simplifying Distributed Applications with Spring
Remoting



Spring Web Services

Implementing Loosely Coupled Communication
with Spring Web Services

Topics in this Session

- **Introduction to Web Services**
 - Why use or build a web service?
 - Best practices for implementing a web service
- Spring Web Services
- Client access

Web Services enable *Loose Coupling*



*“Loosely coupled systems are considered useful when either the **source** or the **destination** computer systems are subject to frequent changes”*

Wikipedia (July 2007)

**Loose coupling increases tolerance...
changes should not cause incompatibility**

Web Services enable *Interoperability*



- XML is the lingua franca in the world of interoperability
- XML is understood by all major platforms
 - SAX, StAX or DOM in Java
 - System.XML or .NET XML Parser in .NET
 - REXML or XmlSimple in Ruby
 - Perl-XML or XML::Simple in Perl

Best practices for implementing web services



- Remember:
 - web services are not only about SOAP (e.g. REST)
 - web services != RPC
- Design contract independently from service interface
 - Generating contract from code results in tight coupling and awkward contracts
- Refrain from using stubs and skeletons
 - Code and contract should be able to evolve independently

Loose Coupling & Validation



- Consider skipping validation for incoming requests...
- ... and using Xpath to only extract what you need

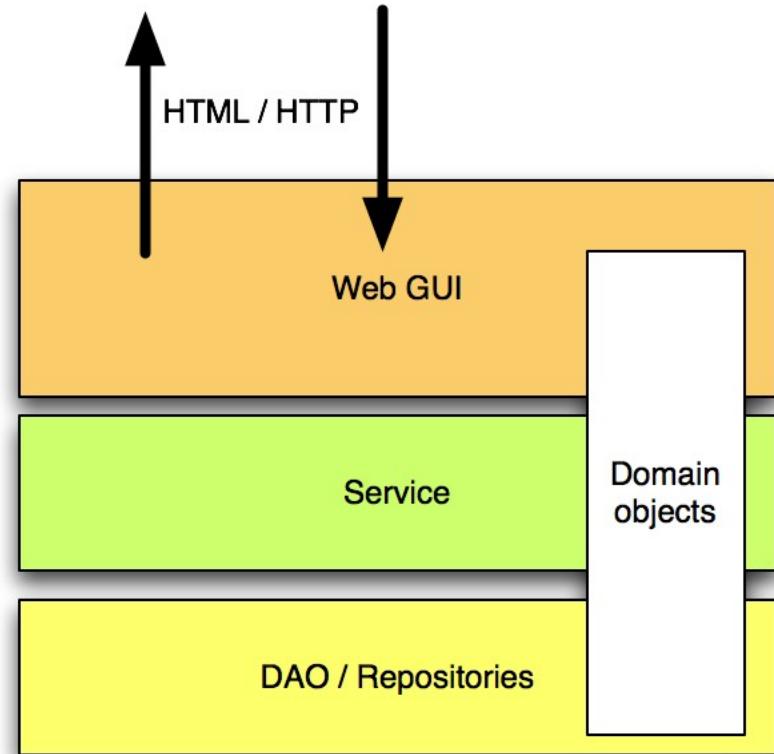
Postel's law:
“Be conservative in what you do;
be liberal in what you accept from others.”

- Easier to successfully process different requests
 - Looser coupling
 - Less versioning issues
 - Increases flexibility for clients
 - e.g. add additional contents, so request can be passed to other services as well
- Not necessarily a best practice, depends on the service

Web GUI on top of your services



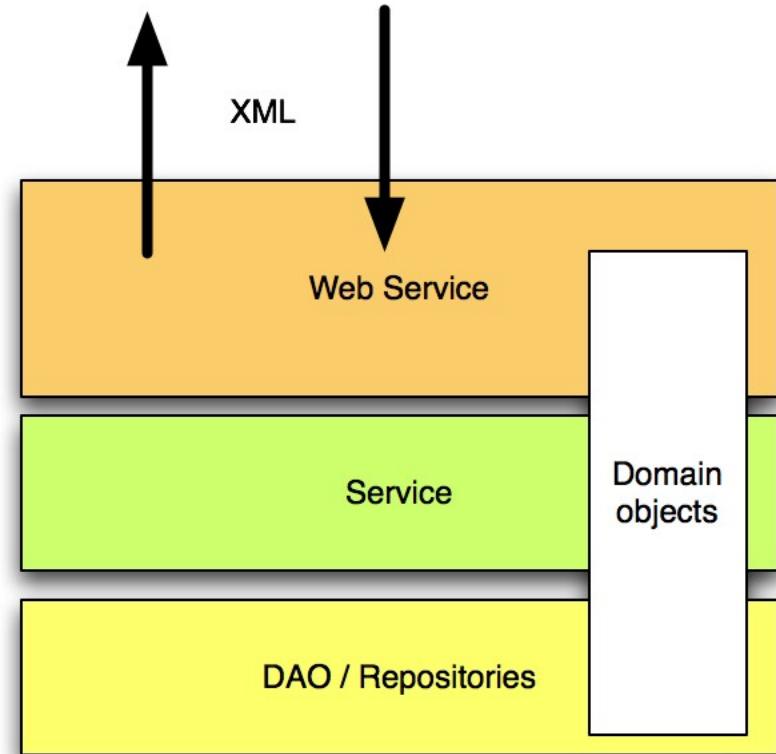
The **Web GUI layer** provides **compatibility** between **HTML-based** world of the user (the browser) and the **OO-based** world of your service



Web Service on top of your services



Web Service layer
provides **compatibility**
between **XML-based**
world of the user and
the **OO-based** world of
your service



Topics in this Session

- Introduction to Web Services
 - Why use or build a web service?
 - Best practices for implementing a web service
- **Spring Web Services**
- Client access

Define the contract

- Spring-WS uses Contract-first
 - Start with XSD/WSDL
- Widely considered a Best Practice
 - Solves many interoperability issues
- Also considered difficult
 - But isn't

Contract-first in 3 simple steps



- Create sample messages
- Infer a contract
 - Trang
 - Microsoft XML to Schema
 - XML Spy
- Tweak resulting contract

Sample Message



Namespace for this message

```
<transferRequest xmlns="http://mybank.com/schemas/tr"
    amount="1205.15">
    <credit>S123</credit>
    <debit>C456</debit>
</transferRequest>
```

Define a schema for the web service message



```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tr="http://mybank.com/schemas/tr"
    elementFormDefault="qualified"
    targetNamespace="http://mybank.com/schemas/tr">
    <xs:element name="transferRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="credit" type="xs:string"/>
                <xs:element name="debit" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="amount" type="xs:decimal"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

```
<transferRequest
    amount="1205.15">
    <credit>S123</credit>
    <debit>C456</debit>
</transferRequest>
```

Type constraints

```
<xs:element name="credit" >  
  <xs:simpleType>  
    <xs:restriction base="xs:string">  
      <xs:pattern value="\w\d{3}"/>  
    </xs:restriction>  
  </xs:simpleType>  
</xs:element>
```

1 Character + 3 Digits

SOAP Message



Envelope

Header

Security

Routing

Body

TransferRequest

Simple SOAP 1.1 Message Example

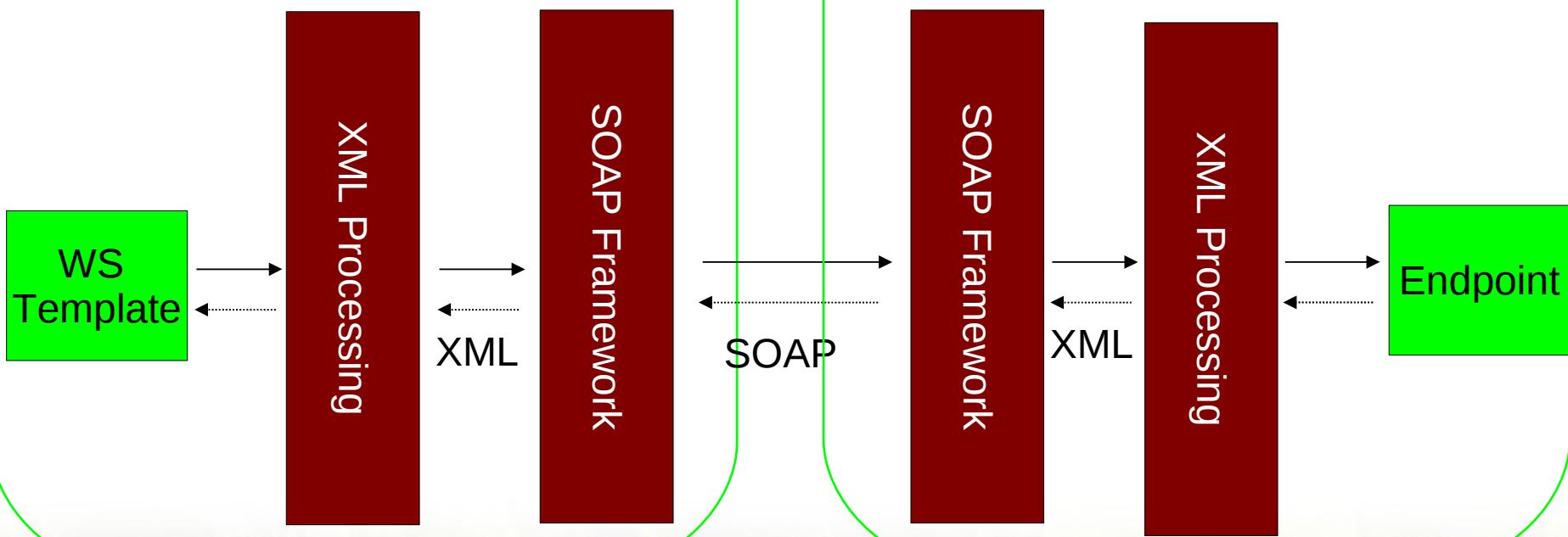


```
<SOAP-ENV:Envelope xmlns:SOAP-  
    ENV="http://schemas.xmlsoap.org/soap/envelope/">  
    <SOAP-ENV:Body>  
        <tr:transferRequest xmlns:tr="http://mybank.com/schemas/tr"  
            amount="1205.15">  
            <tr:credit>S123</tr:credit>  
            <tr:debit>C456</tr:debit>  
  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

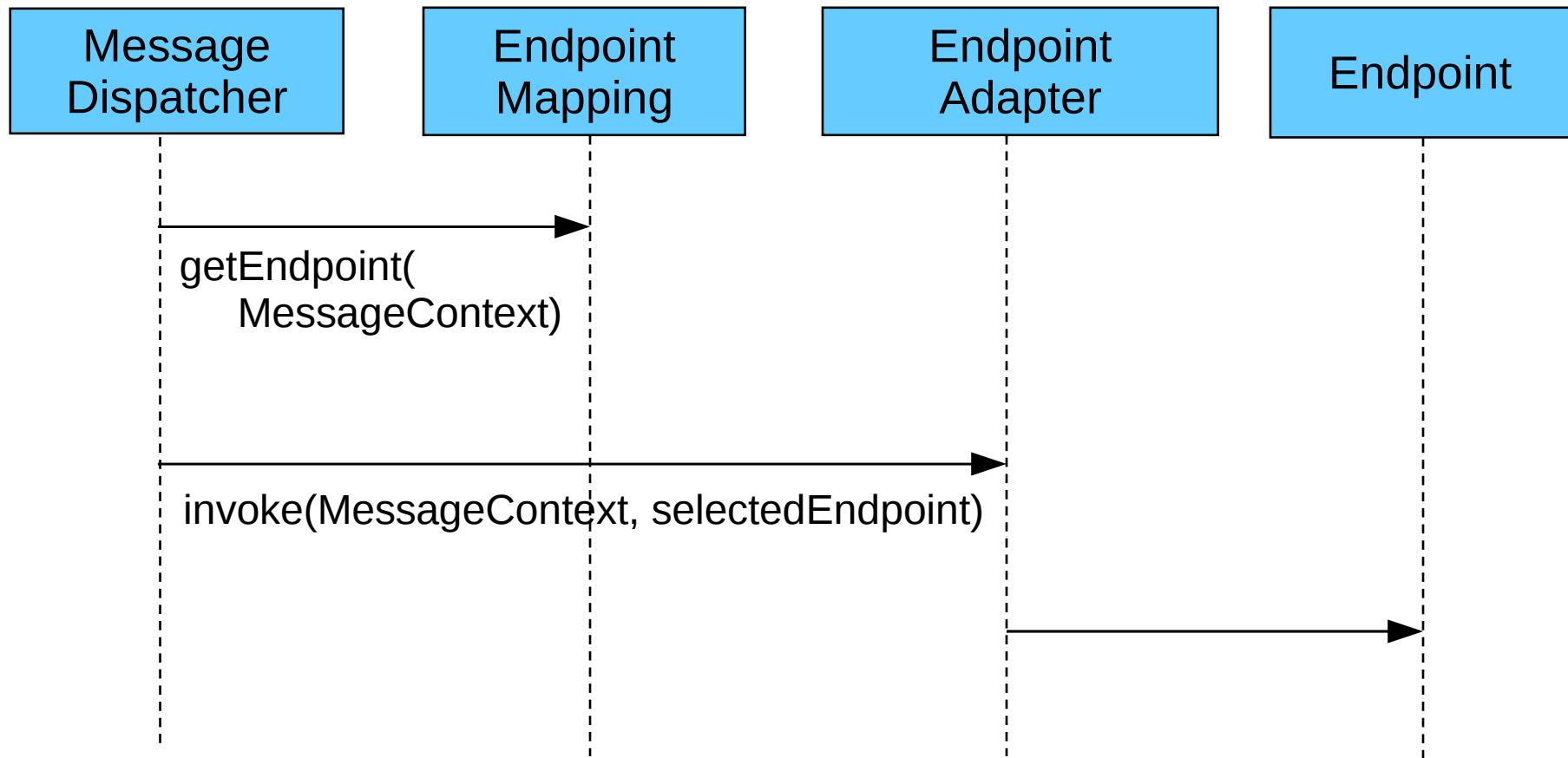
Spring Web Services



Client Process



Request Processing



Bootstrap the application tier



- Inside <webapp> within web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/transfer-app-cfg.xml
    </param-value>
</context-param>
```

The application context's configuration file(s)

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

Loads the ApplicationContext into the ServletContext
before any Servlets are initialized

Wire up the Front Controller (MessageDispatcher)



- Inside <webapp/> within web.xml

```
<servlet>
    <servlet-name>transfer-ws</servlet-name>
    <servlet-class>..ws..MessageDispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/transfer-ws-cfg.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

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

Map the Message Dispatcher Servlet



- Inside <webapp/> within web.xml

```
<servlet-mapping>
  <servlet-name>transfer-ws</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```



There might also be a web interface (GUI) that is mapped to another path

Endpoint

- Endpoints handle SOAP messages
- Similar to MVC Controllers
 - Handle input message
 - Call method on business service
 - Create response message
- With Spring-WS you can focus on the Payload
- Switching from SOAP to POX without code change

XML Handling techniques



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

- JAXB 2 is part of Java EE 5 and JDK 6
- Uses annotations for mapping metadata
- Generates classes from a schema and vice-versa
- Supported as part of Spring-OXM

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="rewards.ws.types"/>
```

- No explicit marshaller bean definition necessary to be used in Spring-WS 2.0 endpoints

```
<!-- registers all infrastructure beans needed for annotation-based  
     endpoints, incl. JAXB2 (un)marshalling: -->  
<ws:annotation-driven/>
```

Implement the Endpoint



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

The diagram illustrates the flow of data processing. A box labeled "Spring WS Endpoint" has an arrow pointing left to the "@Endpoint" annotation. Another arrow points down from this box to a box labeled "Mapping". From the "Mapping" box, two arrows point down to a box labeled "Converted with JAXB2". One arrow points vertically, and the other points diagonally up and to the right.

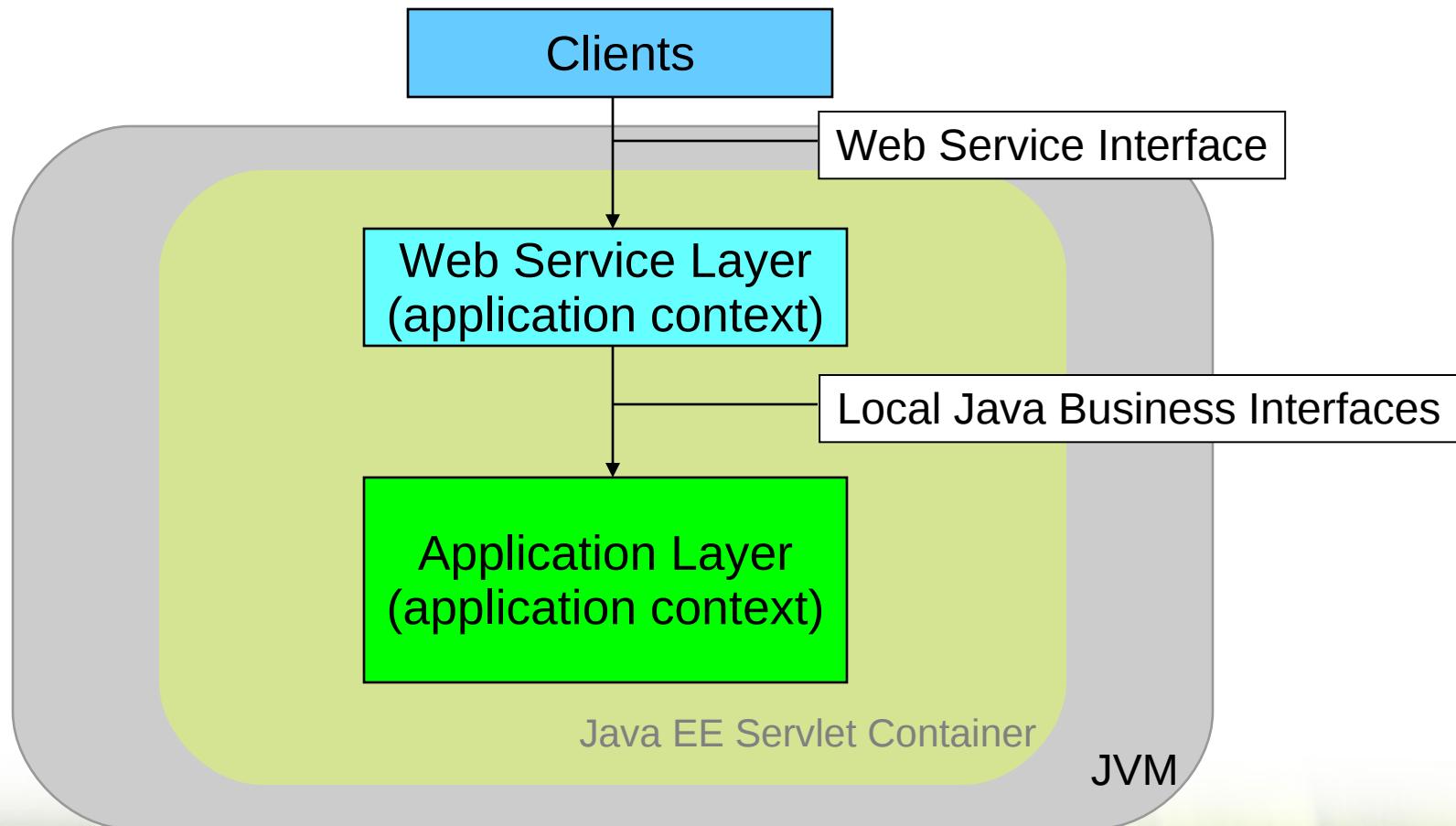
Configure Spring-WS and Endpoint beans



- @Endpoint annotated classes can be component-scanned
 - Saves writing lots of trivial XML bean definitions
 - Just make sure not to scan classes that belong in the root context instead

```
<!-- instead of defining explicit endpoints beans  
    we rely on component scanning: -->  
<context:component-scan base-package="transfers.ws"/>  
  
<!-- registers all infrastructure beans needed for  
    annotation-based endpoints, incl. JAXB2 (un)marshalling:  
-->  
<ws:annotation-driven/>
```

Architecture of our application exposed using a web service



Further Mappings

- You can also map your Endpoints in XML by
 - Message Payload
 - SOAP Action Header
 - WS-Addressing
 - XPath

Publishing the WSDL



http://somehost:8080/transferService/transferDefinition.wsdl

```
<ws:dynamic-wsdl id="transferDefinition"
    portTypeName="Transfers"
    locationUri="http://somehost:8080/transferService/"
    <ws:xsd location="/WEB-INF/transfer.xsd"/>
</ws:dynamic-wsdl>
```

- Generates WSDL based on given XSD
 - id becomes part of WSDL URL
- Most useful during development
- Just use static WSDL in production
 - Contract shouldn't change unless YOU change it

Topics in this Session

- Introduction to Web Services
 - Why use or build a web service?
 - Best practices for implementing a web service
- Spring Web Services
- **Client access**

Spring Web Services on the Client



- **WebServiceTemplate**
 - Simplifies web service access
 - Works directly with the XML payload
 - Extracts *body* of a SOAP message
 - Also works with POX (Plain Old XML)
 - Can use marshallers/unmarshallers
 - Provides convenience methods for sending and receiving web service messages
 - Provides callbacks for more sophisticated usage

Marshalling with WebServiceTemplate



```
<bean id="webServiceTemplate"
      class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="defaultUri" value="http://mybank.com/transfer"/>
    <property name="marshaller" ref="marshaller"/>
    <property name="unmarshaller" ref="marshaller"/>
</bean>

<bean id="marshaller" class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation" value="classpath:castor-mapping.xml"/>
</bean>
```

```
WebServiceTemplate ws = context.getBean(WebServiceTemplate.class);
TransferRequest request = new TransferRequest("S123", "C456", "85.00");
Receipt receipt = (Receipt) ws.marshalingSendAndReceive(request);
```

Topics in this Session

- Introduction to Web Services
 - Why use or build a web service?
 - Best practices for implementing a web service
- Spring Web Services
- Client access



LAB

Exposing SOAP Endpoints using
Spring Web Services



Advanced Spring Web Services

Interceptors, Error Handling and Testing

Topics in this Session

- **Interceptors**
- Error handling
- Out-of-container testing

Interceptors

- EndpointInterceptors receive callbacks during message handling flow
 - handleRequest:invoked before endpoint is called
 - handleResponse: invoked after endpoint (no fault)
 - handleFault: invoked after endpoint (fault)
- All methods can manipulate MessageContext (request/response) and abort further execution
- Built-in implementations for logging, validation, XSLT, WS-Security and WS-Addressing
- Add your own to customize the framework

Interceptor Registration

- Register through namespace: apply to all endpoints

```
<ws:interceptors>
    <bean class="org.sfw.ws...SoapEnvelopeLoggingInterceptor" />
</ws:interceptors>
```

- Optionally restrict to requests with certain payload roots or SOAP actions

```
<ws:interceptors>
    <ws:payloadRoot localPart="MyRequest"
        namespaceUri="http://some.domain.com/namespace">
        <ref bean="xwsSecurityInterceptor"/>
    </ws:payloadRoot>
</ws:interceptors>
```

Built-in Interceptors (1)



- Logging interceptors:
 - SoapEnvelopeLoggingInterceptor
 - Logs full request & response envelope
 - PayloadLoggingInterceptor
 - Logs request & response payload only
- PayloadValidatingInterceptor
 - Validates request and/or response payload against schema
 - Results in SOAP fault for invalid requests
- PayloadTransformingInterceptor
 - Transforms request and/or response payload using XSLT stylesheet (e.g. for different version formats)

Built-in Interceptors (2)



- `AddressingEndpointInterceptor`
 - Provides WS-Addressing support
- **WS-Security interceptors**
 - `XwsSecurityInterceptor` for Sun's XWSS
 - Requires Sun JVM and SAAJ
 - `Wss4jSecurityInterceptor` for WSS4J integration
 - Also supports non-Sun JVMs and Axiom
 - Support for Spring Security, JAAS and keystores

Client-side Interceptors

- `WebServiceTemplate` also supports interceptors
 - `ClientInterceptors`, interface comparable to server
- Built-in implementations for validation and WS-Security

```
<bean id="webServiceTemplate" class="org.sfw.ws.client.core.WebServiceTemplate">
    <property name="interceptors">
        <bean class="org.sfw.ws.client.support.interceptor.PayloadValidatingInterceptor">
            <property name="schema" value="/WEB-INF/schemas/request-schema.xsd"/>
        </bean>
    </property>
</bean>
```

Topics in this Session

- Interceptors
- **Error handling**
- Out-of-container testing

SOAP Faults

- SOAP uses faults to signal errors to a client
- Replace regular payload of response message
- Contain code, reason string and optional detail XML and actor URI

```
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <SOAP:Fault>
      <faultcode>SOAP:Server</faultcode>
      <faultstring>Oops!</faultstring>
    </SOAP:Fault>
  </SOAP:Body>
</SOAP:Envelope>
```

'Client', 'Server', 'MustUnderstand' and 'VersionMismatch' are predefined for SOAP 1.1

Error Handling

- `EndpointExceptionResolver` maps exceptions from endpoint methods to SOAP Messages

Request/response wrapper

```
public interface EndpointExceptionResolver {  
    boolean resolveException(MessageContext messageContext,  
                             Object endpoint, Exception ex);  
}
```

- Default is `SimpleSoapExceptionResolver`
 - Creates SOAP Fault with 'Server' code and exception message as fault string
- Define `EndpointExceptionResolver` to override

SoapFaultMapping-ExceptionResolver



- Maps exceptions to faults by type name

```
<bean class="org.sfw.ws...SoapFaultMappingExceptionResolver">
  <property name="exceptionMappings">
    <value>
      org.springframework.oxm.ValidationFailureException=CLIENT,Invalid request
    </value>
  </property>
  <property name="defaultFault" value="SERVER"/>
</bean>
```

- Keys: exception types
- Values: code (required), fault string (optional), locale (defaults to English)

SoapFaultAnnotation-ExceptionResolver



- Allows annotating exceptions to map to SOAP faults
 - So limited to custom application exceptions

```
<bean class="org.sfw.ws...SoapFaultAnnotationExceptionResolver"/>
```

```
@SoapFault(faultCode=FaultCode.CLIENT)
public class InvalidInputException extends Exception {
    public InvalidInputException(String message) {
        super(message);
    }
}
```

- Fault string defaults to exception message
 - Override using faultStringOrReason attribute

WebServiceTemplate



- SoapFaultMessageResolver used by default for Fault response handling by WebServiceTemplate
 - Throws SoapFaultClientExceptions which wrap the SOAP message and expose the SoapFault
- Can provide custom implementation

```
public interface FaultMessageResolver {  
    void resolveFault(WebServiceMessage message) throws IOException;  
}
```

```
<bean class="org.springframework.ws.client.core.WebServiceTemplate">  
    <property name="faultMessageResolver">  
        <bean class="com.example.CustomFaultMessageResolver"/>  
    </property>  
</bean>
```

Topics in this Session

- Interceptors
- Error handling
- **Out-of-container testing**

Out-of-container Testing



- Spring-WS 2.0 adds support for out-of-container integration testing
 - No stubbing/mocking of endpoint parameters
 - No deployments to local server
- Test endpoints with MockWebServiceClient
 - Fake client that sends request and validates response
 - Tests full MessageDispatcher configuration
- Test clients with MockWebServiceServer
 - Fake server that validates request and sends response
 - Test full WebServiceTemplate configuration

Steps to use the MockWebServiceClient:

1. Create an instance using one of its factory methods
2. Call the sendRequest method with a RequestCreator callback
3. Set up response expectations by calling andExpect method with a ResponseMatcher callback
4. Optionally repeat step 3 for multiple assertions

Creating The Mock Client



- Pass in ApplicationContext to createClient
 - Same defaults apply as with MessageDispatcher

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("spring-ws-servlet.xml")
public class ServerTest {

    @Autowired ApplicationContext applicationContext;

    MockWebServiceClient mockClient;

    @Before
    public void createClient() {
        mockClient = MockWebServiceClient.createClient(applicationContext);
    }

    ...
}
```

Send The Request

- sendRequest takes RequestCreator callback
 - Easiest to use RequestCreators.withPayload static factory methods
 - Provide Source or Resource for XML payload

```
import static org.springframework.ws.test.server.RequestCreators.withPayload;  
... // code from previous slide omitted  
@Test  
public void customerEndpoint() throws Exception {  
    Source requestPayload = new StringSource(  
        "<customerCountRequest xmlns='http://springframework.org/spring-ws'>" +  
        "  <customerName>John Doe</customerName>" +  
        "  </customerCountRequest>");  
    ...  
    mockClient.sendRequest(withPayload(requestPayload))  
        .andExpect([someResponseMatcher]);  
}
```

See next slide

Check The Response

- `andExpect` takes `ResponseMatcher` callback
 - Easiest to use `ResponseMatchers` static factory methods for payload, header and/or fault checks
 - `payload()` takes `Source` or `Resource`, like `withPayload()`

```
import static org.springframework.ws.test.server.ResponseMatchers.*;  
... // code from previous slide omitted  
Source responsePayload = new StringSource(  
    "<customerCountResponse xmlns='http://springframework.org/spring-ws'>" +  
    "  <customerCount>10</customerCount>" +  
    "</customerCountResponse>");  
  
mockClient.sendRequest(withPayload(requestPayload))  
    .andExpect(payload(responsePayload));  
}  
  
extra andExpect() calls can be chained
```

Resource Access In Tests



- Spring-WS applications use WebApplicationContext
 - Resources loaded from root of war file by default
 - e.g. "/WEB-INF/schemas/reward-network.xsd"
- Server tests use ClassPathXmlApplicationContext
 - Resources loaded from classpath by default
 - Won't work with web-relative resources!
- Advice: use classpath resources for things you also need in integration tests
 - e.g. "classpath:schemas/reward-network.xsd"
 - Works with every ApplicationContext type
 - Same tip applies to other Spring web applications

Steps to use the MockWebServiceServer:

1. Create an instance using one of its factory methods
2. Set up request expectations by calling andExpect method with a RequestMatcher callback
3. Optionally repeat step 2 for multiple assertions
4. Create a response using andRespond() with a ResponseCreator callback
5. Use the WebServiceTemplate
6. Verify expectations using verify()

Creating The Mock Server



- Call `createServer` with `WebServiceTemplate` or `ApplicationContext`
 - Latter uses template defined in context
 - Template is configured to use fake `MessageSender`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("integration-test.xml")
public class ClientTest {
    @Autowired WebServiceTemplate template;
    MockWebServiceServer mockServer;

    @Before
    public void createServer() throws Exception {
        mockServer = MockWebServiceServer.createServer(template);
    }
    ...
}
```

Set Request Expectations



- expect takes RequestMatcher callback
 - Easiest to use RequestMatchers static factory methods for payload, XPath, header or URI checks

```
import static org.springframework.ws.test.client.RequestMatchers.*;  
... // code from previous slide omitted
```

```
@Test  
public void customerClient() throws Exception {  
    Source requestPayload = new StringSource(  
        "<customerCountRequest xmlns='http://springframework.org/spring-ws'>" +  
        "  <customerName>John Doe</customerName>" +  
        "  </customerCountRequest>");  
    ...  
    mockServer.expect(payload(requestPayload))  
        .andRespond([someResponseCreator]);  
    ...  
}
```

extra expect() calls can be chained

Create A Response

- `andRespond` takes `ResponseCreator` callback
 - Easiest to use `ResponseCreators` static factory methods for payload, connection error, fault or exception

```
import static org.springframework.ws.test.client.ResponseCreators.*;  
  
... // code from previous slide omitted  
Source responsePayload = new StringSource(  
    "<customerCountResponse xmlns='http://springframework.org/spring-ws'>" +  
    "  <customerCount>10</customerCount>" +  
    "</customerCountResponse>");  
  
mockServer.expect(payload(requestPayload))  
    .andRespond(withPayload(responsePayload));  
  
...  
}
```

Call The Template

- After setting up expectations, use the template and verify the expectations
 - Use directly or through some client wrapper class

```
... // code from previous slide omitted  
CustomerCountRequest request = new CustomerCountRequest("John Doe");  
CustomerCountResponse response =  
    (CustomerCountResponse) template.marshalSendAndReceive(request);  
assertEquals(10, response.getCustomerCount());  
mockServer.verify();  
}
```

does not create real HTTP connection



LAB



REST

Representational State Transfer

Topics in this Session

- **What is REST?**
- Core REST Concepts
- RESTful architecture & design
- Advantages
- Java Frameworks
- Spring-MVC 3.0

What is REST?

- Representational State Transfer
- Term coined up by Roy Fielding
 - Author of HTTP spec
- Architectural style
- Architectural basis for HTTP
 - Defined a posteriori
- Embraces HTTP as an *application* protocol, not just as a *transport* protocol

REST is not...

- An API or framework
 - It is only an *architectural style*
- Equal to HTTP
 - REST principles can be followed using other protocols
- The opposite of SOAP
 - REST vs SOAP is a false dichotomy

Topics in this Session

- What is REST?
- **Core REST Concepts**
- RESTful architecture & design
- Advantages
- Java Frameworks
- Spring-MVC 3.0

Core REST Concepts

- Identifiable Resources
- Uniform interface
- Stateless conversation
- Resource representations
- Hypermedia

Identifiable Resources

- Everything is a resource
 - Customer
 - Car
 - Shopping cart
- Resources are expressed by URIs
 - Meaning URLs: REST community prefers the term URI
- Each URI adds value to the client
 - Don't just expose your entire domain

Uniform interface

- Interact with resources using a *constrained* set of operations
 - Many nouns (resources)
 - Few verbs (operations)
- HTTP has this kind of limited interface
 - GET
 - POST
 - PUT
 - DELETE
 - HEAD and OPTIONS for meta-data

GET

- GET retrieves a Representation of a Resource
- GET is a *safe* operation
 - Has no side effects
- GET is *cacheable*
 - Servers may return ETag header when accessed
 - Clients send this header on subsequent retrieval
 - If the resource has not changed, 304 (Not Modified) is returned, with empty body
 - Similar solution exists for Last-Modified header

GET Example

```
GET /transfers/123  
Host: www.somebank.com  
Accept: application/xml  
...
```

HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type: application/xml

```
<transfer>  
...  
</transfer>
```

ETags

```
GET /transfers  
Host: www.somebank.com  
Accept: text/html  
  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
ETag: "b4bdb3-5b0-  
43ad74ee73ec0"  
Content-Length: 1456  
Content-Type: text/html  
  
...
```

```
GET /transfers  
If-None-Match: "b4bdb3-  
5b0-43ad74ee73ec0"  
Host: www.somebank.com  
Accept: text/html  
  
...
```

```
HTTP/1.1 304 Not Modified  
Date: ...  
ETag: "b4bdb3-5b0-  
43ad74ee73ec0"  
Content-Length: 0
```

HEAD

- Identical to GET, without response body
- Typically used for getting a header and saving some bandwidth

```
HEAD /transfers  
Host: www.somebank.com  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 1456  
Content-Type: application/xml  
...
```

POST

- POST creates a new Resource
 - Typically as a child of another Resource
- Response Location header is used to indicate URI of child
- POST is powerful, don't overuse it!
 - Not safe, not *idempotent*

```
POST /transfers
Host: www.somebank.com
Content-Type:
application/xml

<transfer>
...
</transfer>
```

The diagram illustrates a POST request from a client to a server. On the left, a white rectangular box contains the client's request: a POST verb followed by the path '/transfers', the 'Host' header 'www.somebank.com', the 'Content-Type' header 'application/xml', and an XML payload starting with '<transfer>'. An arrow points from this box to a larger white rectangular box on the right, which represents the server's response. This response includes the 'HTTP/1.1 201 Created' status line, the 'Date' header, the 'Content-Length' header, and the 'Location' header in purple, which specifies the URI 'http://mybank.com/transfers/123'. Ellipses at the bottom indicate additional response content.

```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://mybank.com/
           transfers/123
...
```

PUT

- PUT updates a resource or creates it with a known destination URI
- Idempotent operation
 - Same request yields same result
- Not safe! (has side-effects)

```
PUT /transfers/123
Host: www.somebank.com
Content-Type:
application/xml

<transfer>
...
</transfer>
```

The diagram illustrates a PUT request. On the left, a white rectangular box contains the HTTP method 'PUT', the endpoint '/transfers/123', the 'Host' header 'www.somebank.com', the 'Content-Type' header 'application/xml', and the XML payload '`<transfer>`', '...', and '`</transfer>`'. An arrow points from this box to a second white rectangular box on the right. This second box contains the HTTP response 'HTTP/1.1 204 No Content' and the 'Date' header '...'. Ellipses '...' are also present in both boxes.

```
HTTP/1.1 204 No Content
Date: ...
```

Idempotency

- Multiple invocations have the same end-result
- Important concept: allows operations to be retried without unwanted extra side-effects
- Helps with integration solutions
 - When no reliable transport can be used, or when server error doesn't indicate if operation succeeded, client can simply invoke operation again when in doubt
- Try to design for idempotent operations in distributed architectures
 - Not just for RESTful applications

DELETE

- DELETE deletes a resource
- Idempotent
 - Post condition is always the same
- Not safe!

```
DELETE /transfers/123  
Host: www.mybank.com
```

...

```
HTTP/1.1 204 No Content  
Date: ...  
Content-Length: 0  
...
```

HTTP Methods

	Safe	Idempotent	Cacheable
GET	✓	✓	✓
HEAD	✓	✓	✗
PUT	✗	✓	✗
POST	✗	✗	✗
DELETE	✗	✓	✗

Resource representations

- Resources are abstract
 - Only accessed through a particular *representation*
- Multiple representations are possible
 - text/html, image/gif, application/pdf
- Request specifies the desired representation using the Accept HTTP header
 - Or sometimes interpreted through file extension in URI like .xml, .pdf, ...
- Response reports the actual representation using the Content-Type HTTP header
- Best practice: use well-known media types

Stateless conversation

- Server does not maintain state
 - e.g.: Don't use the HTTP Session!
- Client maintains state through links
- Very scalable
 - Every server instance can serve a request
 - Just put a load balancer in front
- Enforces loose coupling (no shared session knowledge)

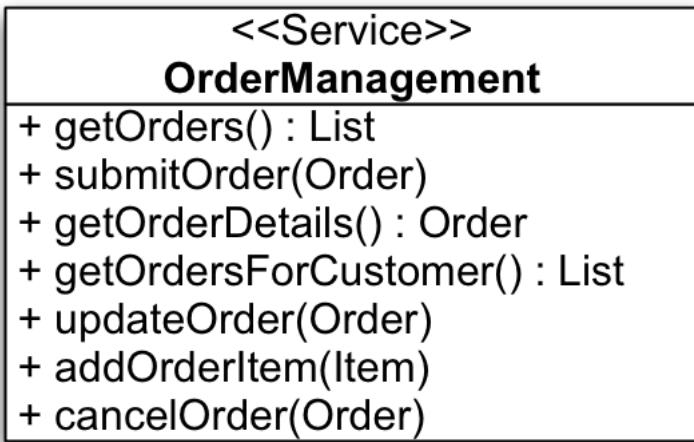
Hypermedia

- Resources contain links
- Client state transitions are made through these links
- Links are provided by server
- Seamless evolution
 - Don't have to update clients when changing the server, just send new links
- Not always easy, as client needs *some* knowledge of server semantics

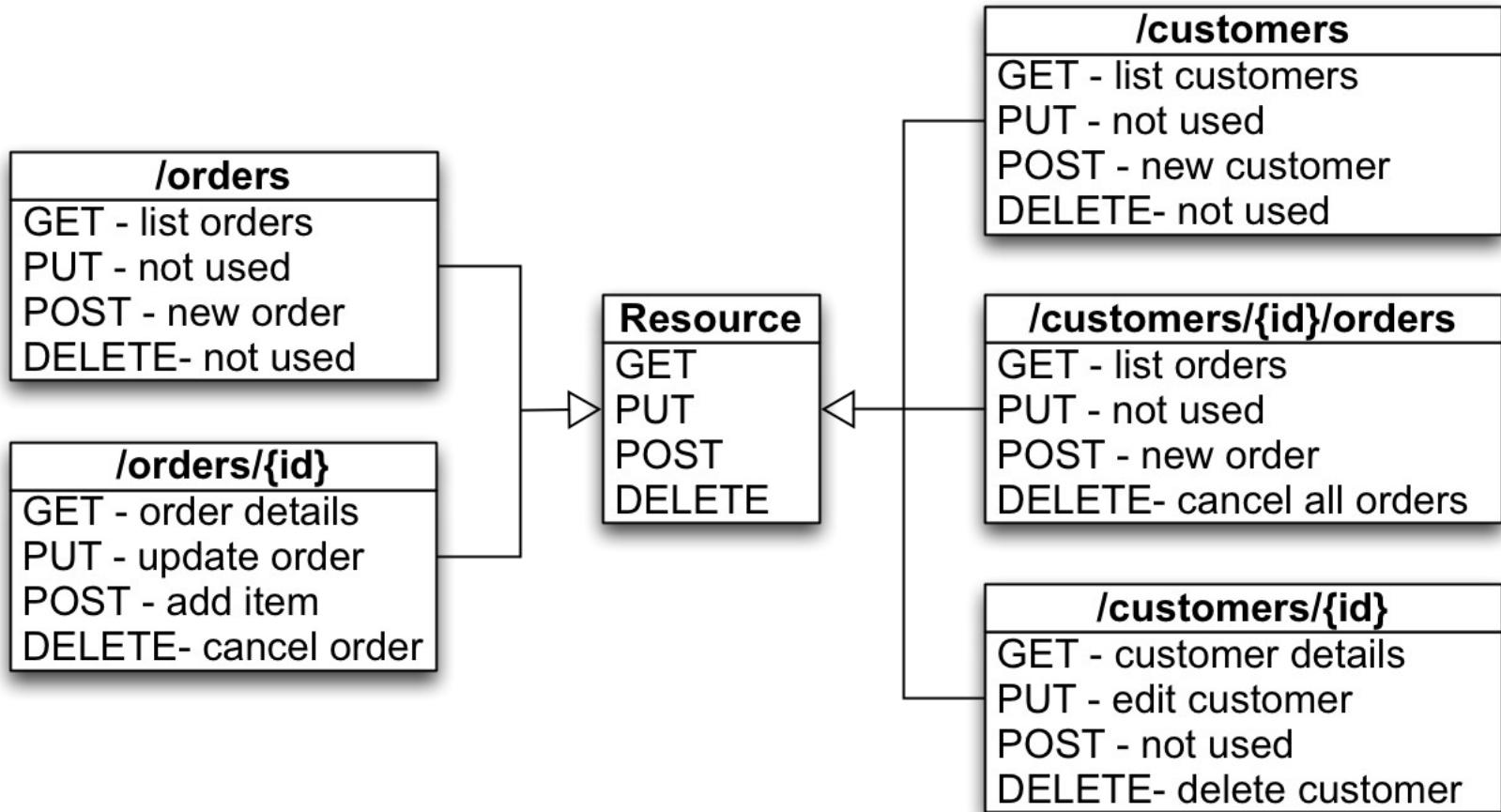
Topics in this Session

- What is REST?
- Core REST Concepts
- **RESTful architecture & design**
- Advantages
- Java Frameworks
- Spring-MVC 3.0

RESTful Architecture



RESTful Architecture



RESTful application design

- Identify resources
 - Design URIs
- Select representations
 - Or create new ones
- Identify method semantics
- Select response codes

Topics in this Session

- What is REST?
- Core REST Concepts
- RESTful architecture & design
- **Advantages**
- Java Frameworks
- Spring-MVC 3.0

Advantages

- Widely supported
 - Languages
 - Scripts
 - Browsers
 - only GET and POST through HTML
- Scalability
- Support for redirect, caching, different representations, resource identification, ...
- Support for XML, but also other formats
 - JSON and Atom are popular choices

Security

- REST uses HTTP Basic or Digest
- Sent on every request
 - REST is stateless, remember ☺
- Requires SSL (transport-level security)
- Use XML-DSIG or XML-Encryption for message-level security
 - Like WS-Security for SOAP messages

Transactions

- HTTP is not designed for long-running transactions
- Use *compensating transactions* instead
- This is also a WS-* best practice
 - WS-Business Activity

Topics in this Session

- What is REST?
- Core REST Concepts
- RESTful architecture & design
- Advantages
- **Java Frameworks**
- Spring-MVC 3.0

REST Frameworks

- Multiple Java frameworks for REST exist
- For Spring users, two options are particularly interesting:
 - 1) JAX-RS
 - 2) Spring-MVC 3.0 with updated REST-support
- Both are valid choices depending on requirements and developer experience

JAX-RS

- Java API for RESTful Web Services (JSR-311)
- Part of Java EE 6
- Standard for writing RESTful applications
- Focuses mostly on application-to-application communication
 - Less focus on browsers as RESTful clients
- Very complete
- Jersey is the reference implementation
 - Ships with Spring integration out of the box
 - RESTEasy, Restlet and CXF support Spring as well

JAX-RS Example

```
import javax.ws.rs.*;
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Scope;

@Path("/reward/{number}")
@Component
@Scope("request")
public class RewardResource {

    @GET
    @Produces("application/xml")
    public Reward getReward(@PathParam("number") String number) {
        return findReward(number);
    }
}
```

Spring MVC 3.0

- Spring MVC in Spring 3.0 includes REST support
 - URI templates
 - Message converters
 - Declaring response status codes
 - Content negotiation
 - RestTemplate for clients
 - And more
- Easier for existing MVC users than JAX-RS
 - As JAX-RS requires different programming model
- Also supports browsers as REST clients
 - Through HTTP Method conversion

Spring MVC 3.0 Example

```
@Controller
@RequestMapping("/reward/{number}")
public class RewardController {

    @RequestMapping(method=GET)
    public @ResponseBody Reward getReward(
        @PathVariable String number) {
        return findRewardByNumber(number);
    }
}
```

Topics in this Session

- What is REST?
- Core REST Concepts
- RESTful architecture & design
- Advantages
- Java Frameworks
- **Spring-MVC 3.0**

Spring-MVC 3.0 Additions

- URI template support

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
public String showOrder(@PathVariable("id") long id, Model model) {
    model.addAttribute(orderRepository.findOrderById(id));
    return "orderDetail";
}
```

- Declarative response status codes (no View)

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public void create(HttpServletRequest req, HttpServletResponse resp) {
    Order order = createOrder(req);
    resp.addHeader("Location", urlForChildResource(req, order.getId()));
}
```

Custom helper method

@ResponseBody

- Object returned by controller method can be converted directly into HTTP response body
 - Instead of Model rendered by View
 - Converter chosen based on Accept header
 - XML through JAXB2, JSON through Jackson, etc.

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseBody Order getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}
```

GET /store/orders/123
Host: www.myshop.com
Accept: application/xml
...

HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type: application/xml
<order id="123">
...
</order>

@RequestBody

- For POSTs and PUTs, HTTP request body can be converted to method parameter
 - Converter chosen based on Content-Type of request

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.PUT)
@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);
}
```

PUT /store/orders/123
Host: www.myshop.com
Content-Type: application/xml
<order>
...
</order>

HTTP/1.1 204 No Content
Date: ...

RestTemplate Introduction (1)

- Provides access to RESTful services
- Supports URI templates and HttpMessageConverters
- Direct request/response payloads
- HttpEntity (since 3.0.2)
 - Represents request or response
 - Provides access to headers in addition to payload
 - Headers copied to HTTP request on post..., put...
 - allows specification of e.g. Content-Type

RestTemplate Introduction (2)

HTTP Method	RestTemplate Method
DELETE	delete(String url, String... urlVariables)
GET	getForObject(String url, Class<T> responseType, String... urlVariables) getForEntity(String url, Class<T> responseType, String... urlVariables)
HEAD	headForHeaders(String url, String... urlVariables)
OPTIONS	optionsForAllow(String url, String... urlVariables)
POST	postForLocation(String url, Object request, String... urlVariables) postForObject(String url, Object request, Class<T> responseType, String... uriVariables) postForEntity(String url, Object request, Class<T> responseType, String... uriVariables)
PUT	put(String url, Object request, String...urlVariables)

Defining a RestTemplate

- Just call constructor in your code

```
RestTemplate template = new RestTemplate();
```

- Has default HttpMessageConverters
 - Same as on the server, depending on classpath
- Or use external configuration
 - To use Apache Commons HTTP Client, for example

```
<bean id="restTemplate" class="org.sfw.web.client.RestTemplate">
  <property name="requestFactory">
    <bean class="org.sfw.http.client.CommonsClientHttpRequestFactory"/>
  </property>
</bean>
```

RestTemplate Usage Examples

```
RestTemplate template = new RestTemplate();
String uri = "http://example.com/store/orders/{id}/items";

// 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);
```

RestTemplate Usage Examples (2)

```
RestTemplate template = new RestTemplate();
HttpEntity<String> request = new HttpEntity<String>("Hello World!",
                                                       MediaType.TEXT_PLAIN);
URI location = template.postForLocation("http://example.com/", request);
...
ResponseEntity<String> response =
    template.getForEntity("http://example.com", String.class);
String body = response.getBody();
MediaType contentType = response.getHeaders().getContentType();
HttpStatus status = response.getStatusCode();
```

Summary

- REST is a viable alternative to the WS-* stack
 - Not necessarily easier
 - Difficult in different ways
- REST builds on
 - Identifiable Resources
 - Uniform interface
 - Stateless conversation
 - Resource representations
 - Hypermedia



LAB

Building a RESTful web service



Introduction To Messaging

Characteristics and Benefits of
messaging-based communication

Topics in this Session

- **Introduction**
- Decoupling
- Use Cases
- Enterprise Integration Patterns

Introduction



- Messaging is another integration style
- Systems exchange messages via *broker*, which:
- can provide guarantees & services
 - Durability (persistent messages)
 - Atomicity (sending/receiving in single transaction)
 - Priority (some messages more important)
- can offer multiple interfaces
 - JMS driver, Stomp, AMQP, etc.
- Often called Message Oriented Middleware (MoM)
 - Available as commercial & open-source products

Messages

- Messages consist of headers and payload
- Headers are metadata
 - Used for identification, routing, etc.
 - Typically key-value pairs
 - Both pre-defined by broker or standard and custom
- Payload is actual content to exchange
 - Sometimes just string or byte array
 - Some systems support typed messages
 - Often uses common data exchange format (XML, JSON, EDI, ...)

Topics in this Session

- Introduction
- **Decoupling**
- Use Cases
- Enterprise Integration Patterns

Broker decouples sender and receiver:

- Spatial
 - Don't have to be co-located to communicate
 - Other styles offer this too
- Temporal
 - No need for other system(s) to respond immediately
 - Asynchronous interaction
- Logical
 - Sender(s) don't need to know about receiver(s)

Temporal Decoupling



- Asynchronous communication doesn't hold up sender
 - Just give message to the broker and resume
 - No need to block sender waiting for receiver
 - Broker buffers messages until receiver is ready
 - Receiver doesn't even need to be running!
- Allows for sending-only or request-reply
 - Replies delivered back as separate messages
 - Receiver can correlate with original request
 - a.k.a. out of band communication
 - vs. in-band like with RPC, REST or SOAP over HTTP
 - cf. email vs. phone conversation

Logical Decoupling



- Senders don't target receivers directly
 - And receivers don't know about senders
- Use broker-defined destinations instead
- Allows extra logic between sender and receiver
 - Routing: determining receiver(s) for a message
 - Filtering: dropping messages
 - Transforming: changing messages
- Topologies can change without affecting existing senders/receivers

Asynchronous Characteristics



- Asynchronous messaging has pros and cons
- Pros:
 - Better scalability
 - Looser coupling (temporal & logical)
- Cons:
 - Overhead caused by broker
 - Extra complexity
- For many integration use cases pros outweigh cons
 - But always consider on case-by-case basis

- Some pros apply to intra-application integration as well
 - Logically decoupled components exchanging data through in-memory messages
 - Event-driven system, no hard-coded control flow
 - Allows same patterns as with MoM within applications
- Frameworks like Spring Integration and Apache Camel enable this
 - Combined with adapters to connect to external systems

Topics in this Session

- Introduction
- Decoupling
- **Use Cases**
- Enterprise Integration Patterns

Simple producer – consumer

- Unidirectional: send message off for further processing
 - Web application places order with back-end system
 - Trigger messages, e.g. to set off a batch process
- Bi-directional: request – reply
 - Retrieve data from system with limited availability
 - Throttle messages so receiver isn't overloaded

Use Cases

Pipelining

- Sending message through multiple systems
 - Order chain and other process flows
 - Content enrichment
- Often called pipes and filters architecture
 - Pipes contain the messages,
filters are the components sending and receiving
 - cf. Unix pipes

Message Distribution

- Same message received by multiple receivers
 - Publish-subscribe
 - Event broadcasting
- Competing receivers
 - Scalable workload distribution
 - Optimizing resource usage

Use Cases

Message Aggregation

- Handling similar requests from many clients
 - Each client usually has dedicated reply destination
- Gathering events
 - Complex Event Processing (CEP)
 - Centralized logging

Topics in this Session

- Introduction
- Decoupling
- Use Cases
- **Enterprise Integration Patterns**

Enterprise Integration Patterns [EIP]

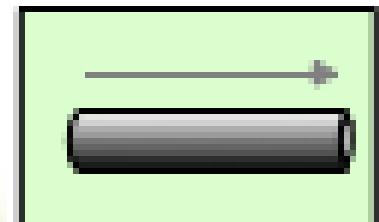


- Book that catalogs messaging patterns for enterprise integration architecture
 - Written by Gregor Hohpe & Bobby Woolf
 - <http://eaipatterns.com/>
- Standard names for common MoM concepts
 - Channel, Endpoint, etc.
- Design Patterns for architecting solutions based on these building blocks
 - Filter, Router, Splitter/aggregator, Resequencer, etc.
- Includes graphical icons for every pattern

Message Channel

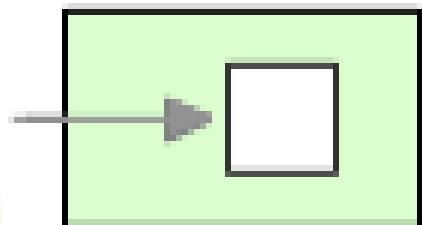


- Allows two applications to communicate
 - One application writes to the channel
 - Another reads from the channel
 - Point-to-point
 - Or multiple applications receive a copy
 - Publish-subscribe
- A channel can buffer messages



Message Endpoint

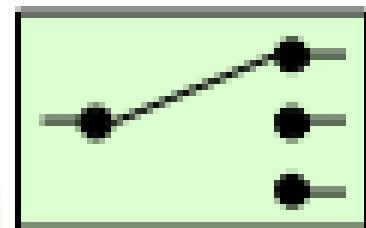
- Connects an application to a Message Channel
 - For sending or receiving
- Can take care of *polling* if necessary
 - i.e. checking for new messages
- Isolates application from the messaging system
- May wrap and unwrap messages
 - Application may only need to deal with the payload



Message Router



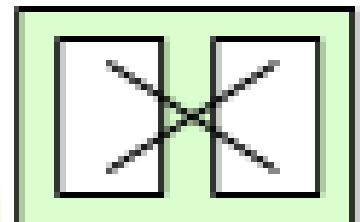
- Dynamically forwards messages to specific channel(s) based on a set of conditions
- *Content-Based* Routers use information within the Message (payload or headers) to determine channel(s) to forward to



Message Translator



- Translates messages from one data format to another
- May also *enrich* content in a payload or add headers



Conclusion

- Asynchronous messaging brings many benefits
 - Flexibility
 - Resilience
 - Scale
 - Performance
 - compared to e.g. file- or DB-based integration
- Integration style that applies to many scenarios
 - Even to integrating components within an application
- Well established with many mature solutions
 - But innovation still happening (e.g. AMQP / RabbitMQ)



Spring JMS

Simplifying Messaging Applications

Topics in this Session



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

Java Message Service (JMS)



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

JMS Core Components



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

JMS Message Types



- Implementations of the Message interface
 - TextMessage
 - ObjectMessage
 - MapMessage
 - BytesMessage
 - StreamMessage

JMS Destination Types

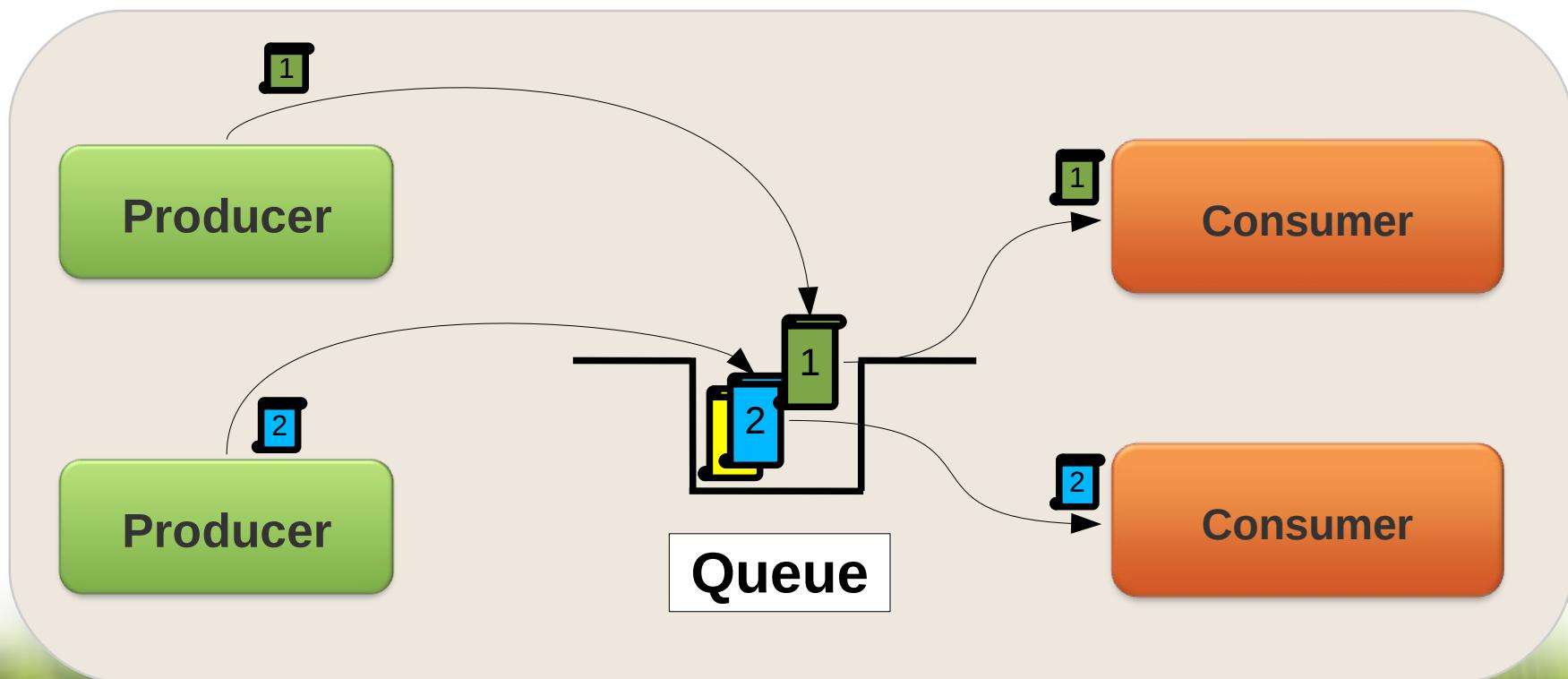


- Implementations of the Destination interface
 - Queue
 - Point-to-point messaging
 - Topic
 - Publish/subscribe messaging

JMS Queues: Point-to-point



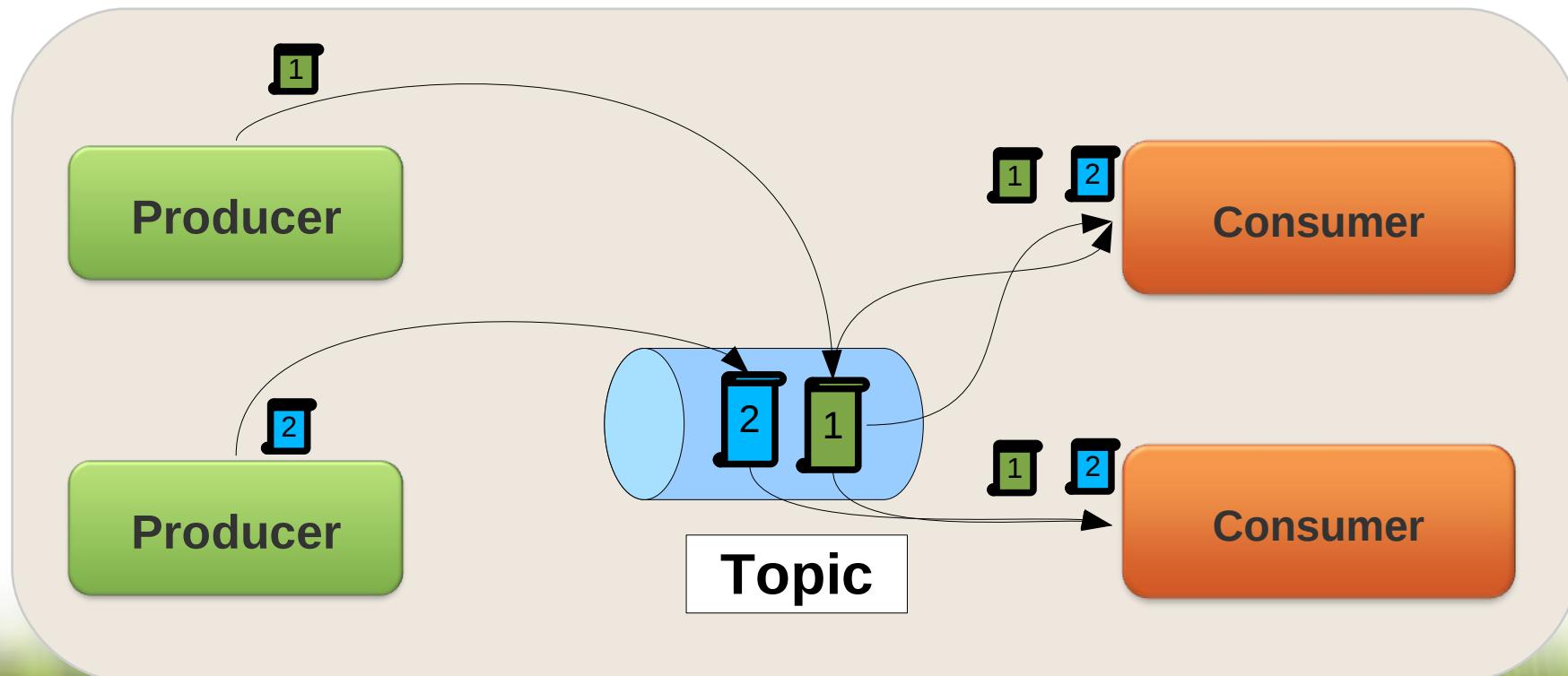
1. Message sent to queue
2. Message queued
3. Message consumed by *single* consumer



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();
```
- In a typical enterprise application, the ConnectionFactory is a managed resource and bound to JNDI

```
Properties env = new Properties();
// provide JNDI environment properties
Context ctx = new InitialContext(env);
ConnectionFactory connectionFactory =
    (ConnectionFactory) ctx.lookup("connFactory");
```

The JMS Session



- A Session is created from the Connection
 - Represents a unit-of-work
 - Provides transactional capability

```
Session session = conn.createSession(  
    boolean transacted, int acknowledgeMode);
```

```
// use session  
if (everythingOkay) {  
    session.commit();  
} else {  
    session.rollback();  
}
```

Creating Messages



- The Session is responsible for the creation of various JMS Message types

```
session.createTextMessage("Some Message Content");
```

```
session.createObjectMessage(someSerializableObject);
```

```
MapMessage message = session.createMapMessage();
message.setInt("someKey", 123);
```

```
BytesMessage message = session.createBytesMessage();
message.writeBytes(someByteArray);
```

Producers and Consumers



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

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

Topics in this Session



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

JMS Providers

- Most providers of Message Oriented Middleware (MoM) support JMS
 - WebSphere MQ, Tibco EMS, Oracle EMS, Jboss AP, SwiftMQ, etc.
 - Some are Open Source, some commercial
 - Some are implemented in Java themselves
- The lab for this module uses Apache ActiveMQ

- Open source message broker written in Java
- Supports JMS and many other APIs
 - Including non-Java clients!
- Can be used stand-alone in production environment
 - 'activemq' script in download starts with default config
 - SpringSource provides support
- Can also be used embedded in an application
 - Configured through ActiveMQ or Spring xml files
 - What we use in the labs

Support for:

- Many cross language clients & transport protocols
 - Incl. excellent Spring integration
- Flexible & powerful deployment configuration
 - Clustering incl. load-balancing & failover, ...
- Advanced messaging features
 - Message groups, virtual & composite destinations, wildcards, etc.
- Enterprise Integration Patterns when combined with Spring Integration or Apache Camel
 - from the book by Gregor Hohpe & Bobby Woolf

Topics in this Session



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

Configuring JMS Resources with Spring



- Spring enables decoupling of your application code from the underlying infrastructure
 - Container provides the resources
 - Application is simply coded against the API
- Provides deployment flexibility
 - use a standalone JMS provider
 - use an application server to manage JMS resources

Configuring a ConnectionFactory



- ConnectionFactory may be standalone

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

- Or retrieved from JNDI

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

Configuring Destinations



- Destinations may be standalone

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

- Or retrieved from JNDI

```
<jee:jndi-lookup id="orderQueue"
                  jndi-name="jms/OrderQueue"/>
```

Topics in this Session



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

Spring's JmsTemplate



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

JmsTemplate Strategies



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

MessageConverter



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

Implementing MessageConverter



- It is sometimes desirable to provide your own conversion strategy
 - Reuse existing code
 - Delegate to OXM marshaller (available in Spring 3.0)
- Implement the two necessary methods

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

- Provide the implementation to JmsTemplate via dependency injection

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

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

Defining a JmsTemplate Bean



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

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

CachingConnectionFactory



- JmsTemplate aggressively closes and reopens JMS resources like Sessions and Connections
 - Assumes these are cached by ConnectionFactory
- Without caching this causes lots of overhead
 - Resulting in poor performance
- Use CachingConnectionFactory to add caching within the application if needed

```
<bean id="connectionFactory"
      class="org.springframework.jms.connection.CachingConnectionFactory">
  <property name="targetConnectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://embedded?broker.persistent=false"/>
    </bean>
  </property>
</bean>
```

Topics in this Session



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

Sending Messages



- The template provides options
 - One line methods that leverage the template's MessageConverter
 - Callback-accepting methods that reveal more of the JMS API
- Use the simplest option for the task at hand

Sending Messages with Conversion



- Leveraging the template's MessageConverter

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

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

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

Sending Messages with Callbacks



- When more control is needed, use callbacks

```
public void convertAndSend(Object message,  
                           MessagePostProcessor mpp);
```

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

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

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

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

SessionCallback Example: Synchronous Request-Reply



```
RewardConfirmation confirmation =  
    jmsTemplate.execute(new SessionCallback() {  
  
    public Object doInJms(Session session) throws JMSEException {  
        TemporaryQueue replyQueue = session.createTemporaryQueue();  
        Message request = session.createObjectMessage(dining);  
        request.setJMSReplyTo(replyQueue);  
        MessageProducer producer = session.createProducer(diningQueue);  
        producer.send(request);  
  
        MessageConsumer consumer = session.createConsumer(replyQueue);  
        ObjectMessage reply = (ObjectMessage) consumer.receive(60000);  
        replyQueue.delete(); // delete here, since Connection might be cached  
        return reply != null ? (RewardConfirmation) reply.getObject : null;  
    }  
});
```

Topics in this Session



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

Synchronous Message Reception



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

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

The JMS MessageListener



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

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

Spring's MessageListener Containers



- Traditionally, use of MessageListener implementations required an EJB container
- Spring provides lightweight alternatives
 - SimpleMessageListenerContainer
 - Uses plain JMS client API
 - Creates a fixed number of Sessions
 - DefaultMessageListenerContainer
 - Adds transactional capability
- Advanced scheduling and endpoint management options available for each container option

Defining a plain JMS Message Listener



- Define listeners using jms:listener elements

```
<jms:listener-container connection-factory="myConnectionFactory">
    <jms:listener destination="queue.order" ref="myOrderListener"/>
    <jms:listener destination="queue.conf" ref="myConfListener"/>
</jms:listener-container>
```

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

Spring's message-driven objects



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

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

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

- MessageConverter provides parameter
- Return value sent to response-destination after conversion



LAB

Sending and Receiving Messages in a Spring Environment



JMS Transactions

Transactional Messaging Applications with
Spring

Topics in this Session



- **Why use JMS transactions**
- JMS Session as Unit of Work
- Overview of transactional options
- Transactional JMS Resources with Spring
- Duplicate Message Handling

Why use JMS Transactions



- Interaction with JMS is by definition stateful
- Adds or removes messages from a destination
- In composite stateful interactions we need to account for ACID principles
 - Prevent Partial failures or Duplication
- Transactions is one way to do it

Topics in this Session



- Why use JMS transactions
- **JMS Session as Unit of Work**
- Overview of transactional options
- Transactional JMS Resources with Spring
- Duplicate Message Handling

The JMS Session



- JMS Session acts as unit of work
 - Keeps track of the performed operations
- Created through the JMS Connection

```
public interface Connection {  
    Session createSession(boolean transacted, int acknowledgeMode)  
        throws JMSException;  
    // ...
```

- Pass in **true** for transacted Session
 - Call **commit()** or **rollback()** to end transaction
 - All JMS operations must use the same Session!
- Acknowledge mode ignored for transacted Session

Acknowledge Mode

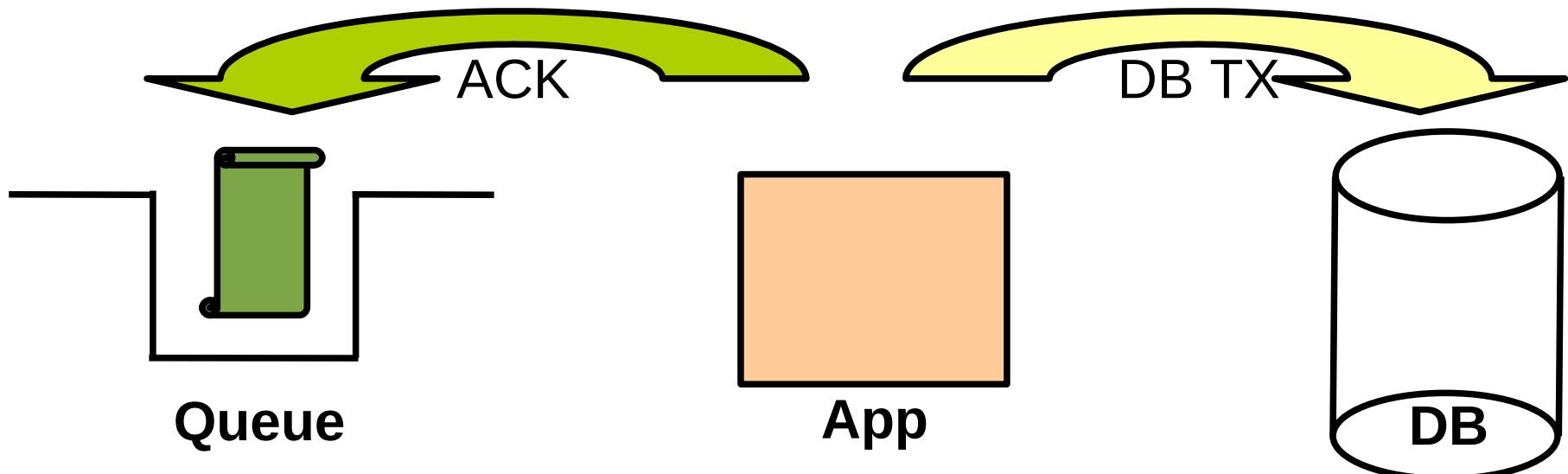


- Indicates when message is considered successfully delivered outside a transaction
- One of the following:
 - AUTO_ACKNOWLEDGE
after every successful `receive()` or `onMessage()`
 - CLIENT_ACKNOWLEDGE
client has to call `Message.acknowledge()` itself
 - DUPS_OK_ACKNOWLEDGE
lazily acknowledge delivery
- Defined as `int` constants on `Session`

- Acknowledge after every message delivery
 - Used as default in Spring without further configuration
- Means successful reception will always mark a message as delivered
 - So it is removed from the queue
 - Message will not be redelivered, so no duplicates
- Can be slower than other acknowledge modes
 - Acknowledgement sent after every delivered message!
- Results in message loss if processing after reception fails!

AUTO_ACKNOWLEDGE

Happy Case Scenario

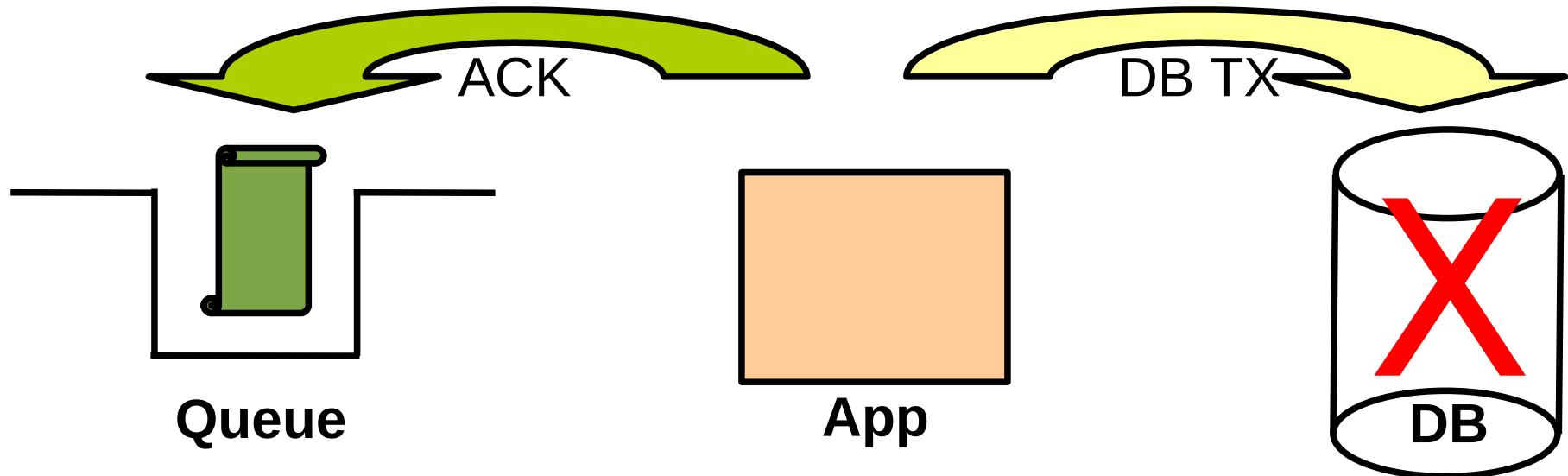


1. Receive & ack message
2. DB TX starts
3. Store message data
4. DB commits

Message in DB, no longer in Queue

AUTO_ACKNOWLEDGE

Message Loss On Error



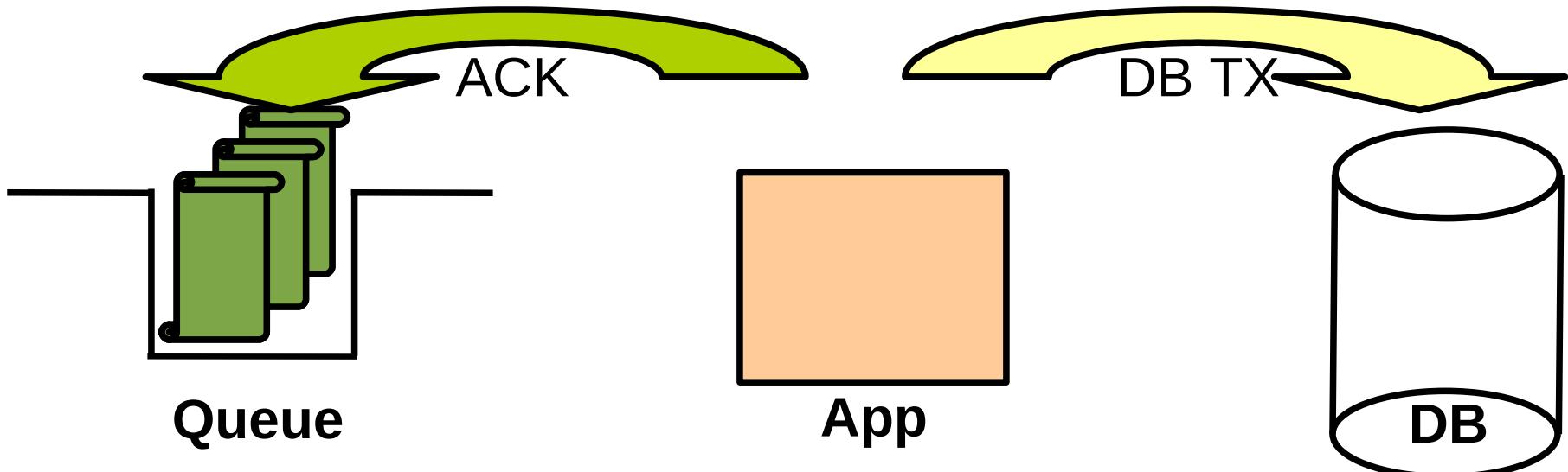
1. Receive & ack message
2. DB TX starts
3. Store message data: error!
4. DB rolls back

**Message not in DB and
no longer in Queue: lost!**

- Client takes responsibility for acknowledging
 - Requires manually calling **Message.acknowledge()**
- Acknowledge multiple messages after processing
 - **Message.acknowledge()** acts on *all* consumed messages of current Session!
 - Optimize nr. of calls MessageConsumer makes to broker
- If client fails before acknowledging, messages will be redelivered
 - Requires call to **Session.recover()** for same Session
 - Results in *duplicates* if processing succeeded already!

CLIENT_ACKNOWLEDGE

Happy Case Scenario

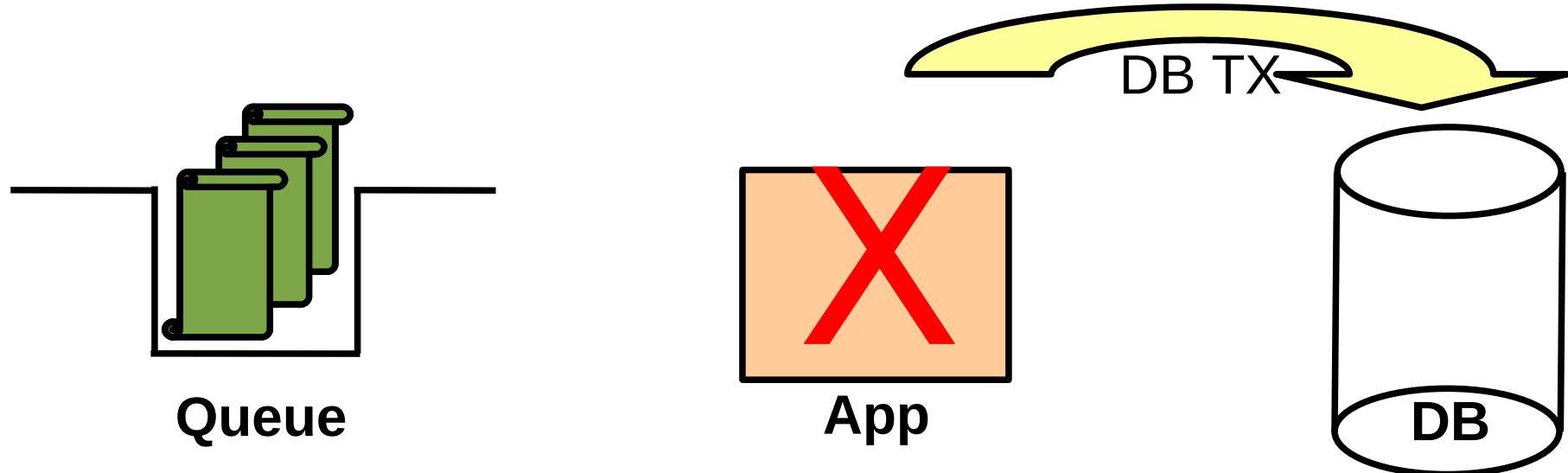


1. Receives
2. DB TX starts
3. Store message data
4. DB commits
5. Messages acknowledged

**Messages in DB,
no longer in Queue**

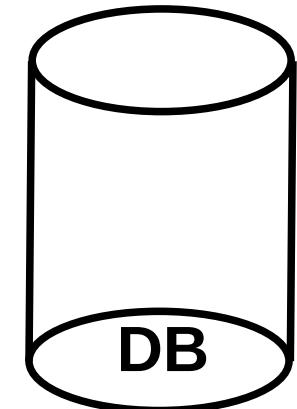
CLIENT_ACKNOWLEDGE

Duplicates On Error



Queue

App



DB

1. Receives
2. DB TX starts
3. Store message data
4. DB commits
5. Error in app before ack

**Messages in DB and still in Queue:
will result in duplicate messages**

- Message delivery is lazily acknowledged
 - Not directly after successful delivery!
 - But still automatically, i.e. not manually by client code
- Can result in duplicates
 - If JMS provider fails before acknowledgement
 - Same as with CLIENT_ACKNOWLEDGE really
 - Means application needs to be able to handle this
 - Good for idempotent consumers
- May result in better performance than auto-ack
 - Lowers overhead for Session to prevent duplicates and for MessageConsumer to communicate with broker

Topics in this Session



- Why use JMS transactions
- JMS Session as Unit of Work
- **Overview of transactional options**
- Transactional JMS Resources with Spring
- Duplicate Message Handling

Flavors of JMS Transactions



- No transaction
 - Acknowledge mode applies
- Local transaction
 - Pass **true** to **Connection.createSession(boolean, int)** to start new transaction
 - Messages acknowledged on TX commit
- XA transactions (not in this presentation)
 - Participate in global transaction
 - Values passed to **createSession** are ignored

Local JMS Transactions



- Only apply to JMS
 - ignores other transactional resources
- True TXs spanning multiple resources require JTA
- Several 'best effort' strategies will work without and could be sufficient:
 - Commit database before JMS
 - Never loses messages
 - Causes duplicate messages if JMS commit fails, like with client-ack/dups_ok
 - Put commits close together to reduce failure opportunity
 - no *application* errors possible after 1st commit, only external errors

Spring Namespace Conventions



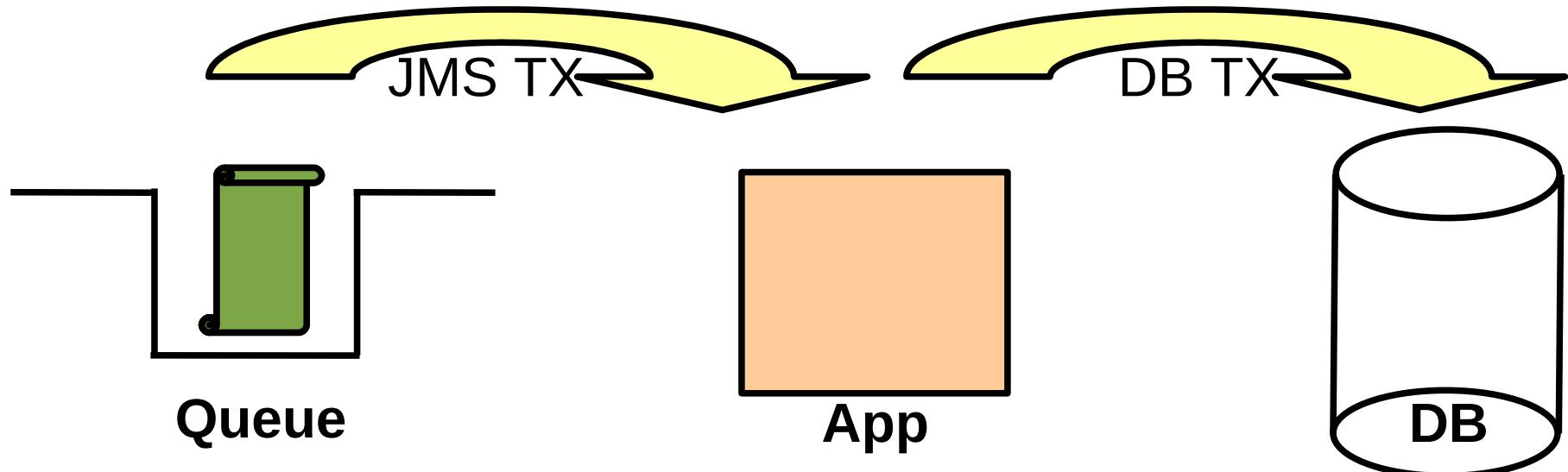
- Transacted Session and Acknowledge Mode are mutually exclusive
- So Spring Namespace specifies them in a single attribute called **acknowledge**:
 - Can be **auto**, **client** or **dups_ok** for acknowledge mode
 - Or **transacted** for transacted session
- **transacted** starts local JMS TX when no managed JMS or JTA TX is in progress
 - Will be synchronized with existing (non-JTA) TX

Participating in a Transaction



- **acknowledge="transacted"** is often enough
 - Assuming listener container started the TX
 - JmsTemplate will automatically use the same Session
 - Synchronizes with TX from other local TX manager (e.g. DataSourceTransactionManager)
 - Commit Session only *after* successful database commit
- Spring also provides **JmsTransactionManager** for when **acknowledge="transacted"** cannot be used or isn't sufficient with local JMX TXs

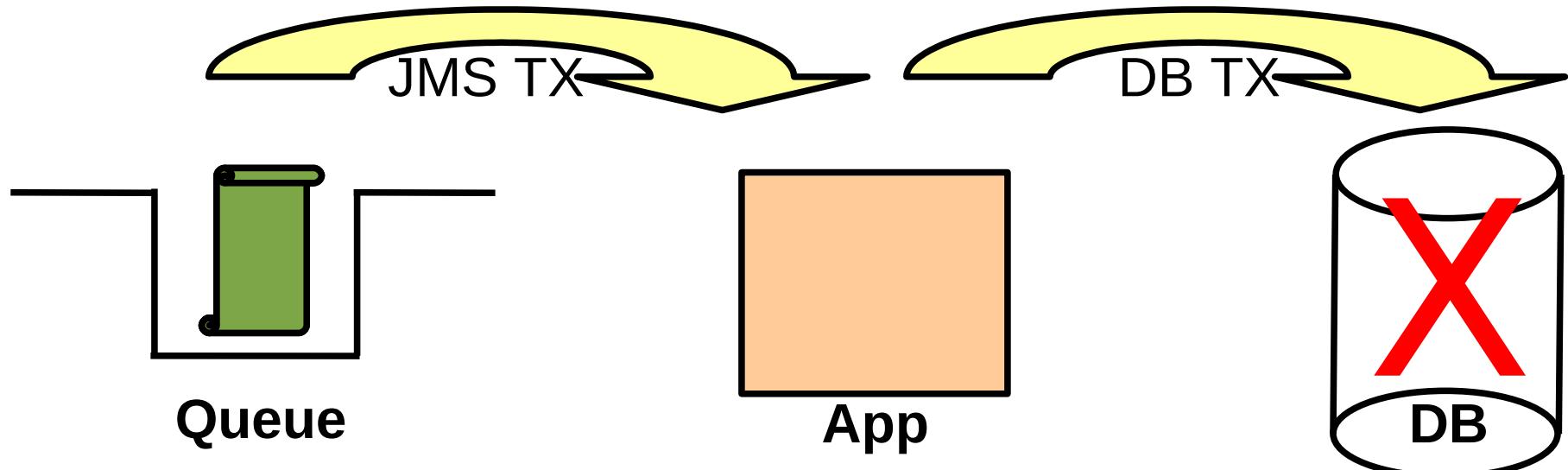
JMS+DB TX Synchronization: Happy Case Scenario



1. JMS TX starts
2. Receive message
3. DB TX starts
4. Store message data
5. DB TX commits
6. JMS TX commits

Message in DB, no longer in Queue

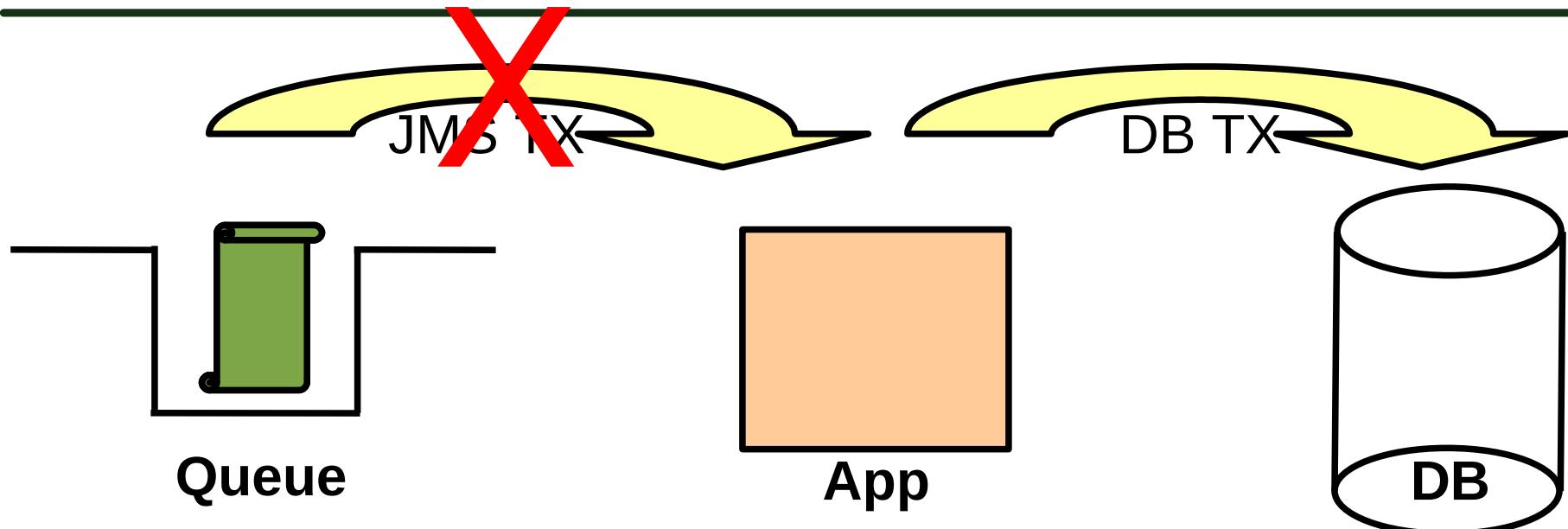
JMS+DB TX Synchronization: DB Rollback



1. JMS TX starts
2. Receive message
3. DB TX starts
4. Store message data: error!
5. DB TX rolls back
6. JMS TX rolls back

Message not in DB, still in Queue

JMS+DB TX Synchronization: JMS Commit Failure



1. JMS TX starts
2. Receive message
3. DB TX starts
4. Store message data
5. DB TX commits
6. JMS TX fails to commit
→ rollback

**Message in DB and still in Queue:
will result in duplicate message**

Topics in this Session



- Why use JMS transactions
- JMS Session as Unit of Work
- Overview of transactional options
- **Transactional JMS Resources with Spring**
- Duplicate Message Handling

Transacted Listener Container



- Specify acknowledge attribute
 - **transacted** gives you local JMS transaction

```
<jms:listener-container acknowledge="transacted">
    <jms:listener ... />
</jms:listener-container>
```

- Or specify a transaction manager
- Delegates to specified transaction manager
 - Typically JTA, could also be JMS

```
<jms:listener-container transaction-manager="transactionManager">
    <jms:listener ... />
</jms:listener-container>
```

Listener Container Transactions



- Transaction starts when message is received
- Setting a JTA transaction manager on the listener container enables Full XA
 - Always consider if you really need XA
 - Necessary if once-and-once-only delivery must be guaranteed at all times
 - If application can handle duplicates, two local TXs are much faster and easier to set up

Transacted JmsTemplate



- Specify default modes for native Sessions
- Will be ignored if the Session was created within an active transaction already

```
<bean class="...JmsTemplate">
    <property name="sessionTransacted" .../>
    <property name="sessionAcknowledgeMode" .../>
</bean>
```

- Don't specify acknowledge mode for transacted sessions (will be ignored)

JmsTemplate Usage



- Once set up nothing changes in the usage
- Read and write operations use the same Session when transaction is in progress
 - Obtained through
ConnectionFactoryUtils.doGetTransactionalSession
- Transactional behavior is defined in configuration

```
// participates in the Session and therefore the transaction
jmsTemplate.convertAndSend(order);
```

```
// participates in the Session and therefore the transaction
jmsTemplate.receiveAndConvert();
```

Topics in this Session

- Why use JMS transactions
- JMS Session as Unit of Work
- Overview of transactional options
- Transactional JMS Resources with Spring
- **Duplicate Message Handling**

Dealing With Duplicates



- Not using XA transactions can cause duplicate messages in many scenarios
 - Only XA guarantees once-and-once-only delivery
- Not always a problem
 - If processing is idempotent, just process it again
 - Otherwise, recognize that you already processed the message and ignore it
 - This is recommended if this suffices: faster & easier
- But how do you recognize a duplicate?

Detecting Duplicates



- First check if message is a redelivery

```
if (message.getJMSRedelivered()) { ... }
```

- If not, just process it
- If so, check if you processed it already
 - Query for a result based on the message payload

```
String orderNumber = extractOrderNumber(message);
if (existingOrder(orderNumber)) { ... }
```
 - You cannot tell this by looking at the message!
 - Could have been rolled back before or after processing
- Redeliveries are rare, so overhead of extra check is not incurred often

Summary



- To avoid duplicating or losing messages transactions can be used
- Without XA, JMS and database transactions need to be synchronized
 - Prevents losing messages
 - Only XA completely avoids duplicates
- This is never foolproof
- Think about failure scenarios carefully



LAB

Resolving transactional issues with Spring JMS



Global transactions using XA, JTA and Spring

Enabling ACID transactions across
multiple transaction resources

Topics in this Session

- **Introduction**
- Two Phase Commit and XA
- JTA and Spring
- Transaction Demarcation

Introduction

- Transactions manage changes made to stateful resources
 - Databases, message queues, etc.
- Provide ACID guarantees for all operations performed in a single transaction
 - Atomic
 - Consistent
 - Isolated
 - Durable
- Two ways to manage: *local* or *global*

Local Transactions



- Manage TXs on the resource itself
 - JDBC Connection, JMS Session, ...
 - AKA *Native* transaction
- Best solution if TX involves a single resource
 - Works in all environments
 - Very fast
- Can cause issues with multiple resources
 - ACID properties no longer guaranteed
 - Something needs to coordinate the TX across resources

- Use a dedicated **transaction manager**
 - Allows for ACID guarantees with multiple transactional resources in a single TX
 - Coordinates TXs across all participating resources
- Sometimes called *distributed* transaction
 - Same term is also used for propagating TX contexts to other transaction managers
 - We'll use *global* in this presentation to avoid confusion

Committing a Global TX



- With multiple resources, TXs can't simply commit in one step
 - Subsequent resources might fail to commit after the first resource successfully commits
 - Prevents ACID guarantees for entire TX
- This means global transaction committing needs multiple, coordinated steps

Topics in this Session

- Introduction
- **Two Phase Commit and XA**
- JTA and Spring
- Transaction Demarcation

Two Phase Commit



- Global TXs use **Two Phase Commit** (2PC)
- Driven by the transaction manager
- Phase 1: First check with all participating resources if they're ready to commit (*prepare*)
- Phase 2: Only if they all are, do the commit
- Otherwise perform a rollback

- 2PC requires a common interface between transaction manager and the resources
 - How else can they communicate?
- The common standard to do this is **XA**
- Specification from the X/Open group
- Very complicated
 - You don't need to read the spec to use global TXs
 - You do need to understand the basics, though!

- TX manager performs the following operations on participating resources:
 - 1) start(Xid): join the TX
 - 2) end(Xid): no more TX work for this resource
 - 3) prepare(Xid): is resource ready to commit?
 - 4) commit(Xid): perform the actual commit
- where Xid is transaction id for a resource
 - Contains global TX id, same for all resources in TX
 - Also has *branch qualifier*: part of the TX that the resource needs to care about

- This means you need two things:
 - An XA-aware transaction manager
 - XA-aware transactional resources
- First is provided by JTA transaction manager
 - We'll discuss JTA in a moment
- Second needs to be provided by resource's vendor
 - Often in different class than used with non-XA TXs

Two Phase Commit Failures



- 2PC presents some hard cases to deal with:
 - Resource becomes unavailable after 1st phase
 - Transaction manager itself fails during execution
- Transaction manager is ultimately responsible
- Keeps detailed log on disk while TX in progress
 - As do the XA resources themselves!
- Allows TX to be *recovered* later
 - Guarantees consistent state between all resources
 - At the cost of a *lot* of additional overhead
 - XA Specification covers this in detail

Topics in this Session

- Introduction
- Two Phase Commit and XA
- **JTA and Spring**
- Transaction Demarcation

- Java Transaction API
- Allows global transactions using XA in Java
 - JTA can also be used without XA
 - But XA requires JTA!
- JTA support built-in with J2EE / Java EE servers
 - Stand-alone implementations exist as well
 - e.g.: Atomikos, JOTM, Arjuna (now JBoss Transactions)
 - Prefer built-in if available
- Typically not used by application code directly

Using JTA

- Required if you use EJBs for TX management
 - Even if you don't need global TXs
 - Means Container-Managed Transactions always incur JTA overhead
- Optional if you use Spring
 - Just use when you need global TXs
 - Switching from local to global is easy

Using JTA with Spring



- `JtaTransactionManager` is part of Spring's transaction management support
- Integrates with external JTA TX manager
 - Does NOT provide JTA support by itself!
- Used instead of local implementation of `PlatformTransactionManager`
- Doesn't require any code changes and very little configuration changes

Configuring JTA with Spring



- Within a Java EE server, just ask Spring to look up the JTA TX manager:

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

- Creates bean called transactionManager
- Uses server-specific subclass if present
 - Allows things not in JTA spec, like TX suspension
- Combined with JNDI lookups for TX resources

```
<jee:jndi-lookup id="dataSource"  
                 jndi-name="java:comp/env/jdbc/myDS" />  
<jee:jndi-lookup id="connectionFactory"  
                 jndi-name="java:comp/env/jms/myConnFact" />
```

Configuring JTA with Spring



- For stand-alone applications, define plain bean:

```
<bean id="transactionManager"
      class="org.springframework.tx.jta.JtaTransactionManager">
    <property name="transactionManager" ref="jtaTxMgr"/>
    <property name="userTransaction" ref="userTx"/>
</bean>
```

- Note: 'transactionManager' as id is a best practice
 - allows many Spring-components to find the TX manager automatically without wiring instructions, esp. when using namespace support

Configuring TX Resources



- For JDBC or JMS, using XA-aware types is typically enough
 - Not configured in application with real app server
- Other code may need instructions to find JTA TX manager (note: this is *not* Spring-specific!)

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<property name="hibernateProperties">
<value>
  hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
  hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.WeblogicTransactionManagerLookup
  hibernate.current_session_context_class = jta
</value>
...
```

Topics in this Session

- Introduction
- Two Phase Commit and XA
- JTA and Spring
- **Transaction Demarcation**

Transaction Demarcation



- Indicating *where* TXs should start / stop
- Typically done declaratively with Spring
 - using @Transactional annotations or AOP pointcuts
- Not directly related to *how* TXs need to be managed! (local or global)
 - Orthogonal concerns
- Means that with Spring, global TXs don't require code changes

- For databases or synchronous JMS, just use @Transactional with <tx:annotation-driven /> or use <tx:advice>
- For asynchronous JMS, pass JTA transaction manager to listener container:

```
<jms:listener-container transaction-manager="transactionManager">
    <jms:listener ref="jmsListener" destination="queue.name"/>
</jms:listener-container>
```

- Note: for local TXs you would use acknowledge="transacted" instead

Summary

- Global TXs enable multiple resources to participate in a single transaction
- Requires JTA transaction manager and XA-aware resources
- Spring supports JTA, but doesn't require it
- Just a matter of changing configuration
 - use Spring's JtaTransactionManager
 - may also need to reconfigure resources like Hibernate
 - no code changes necessary



Lab

tx-xa-1: Combining JMS and JDBC operations in a global transaction

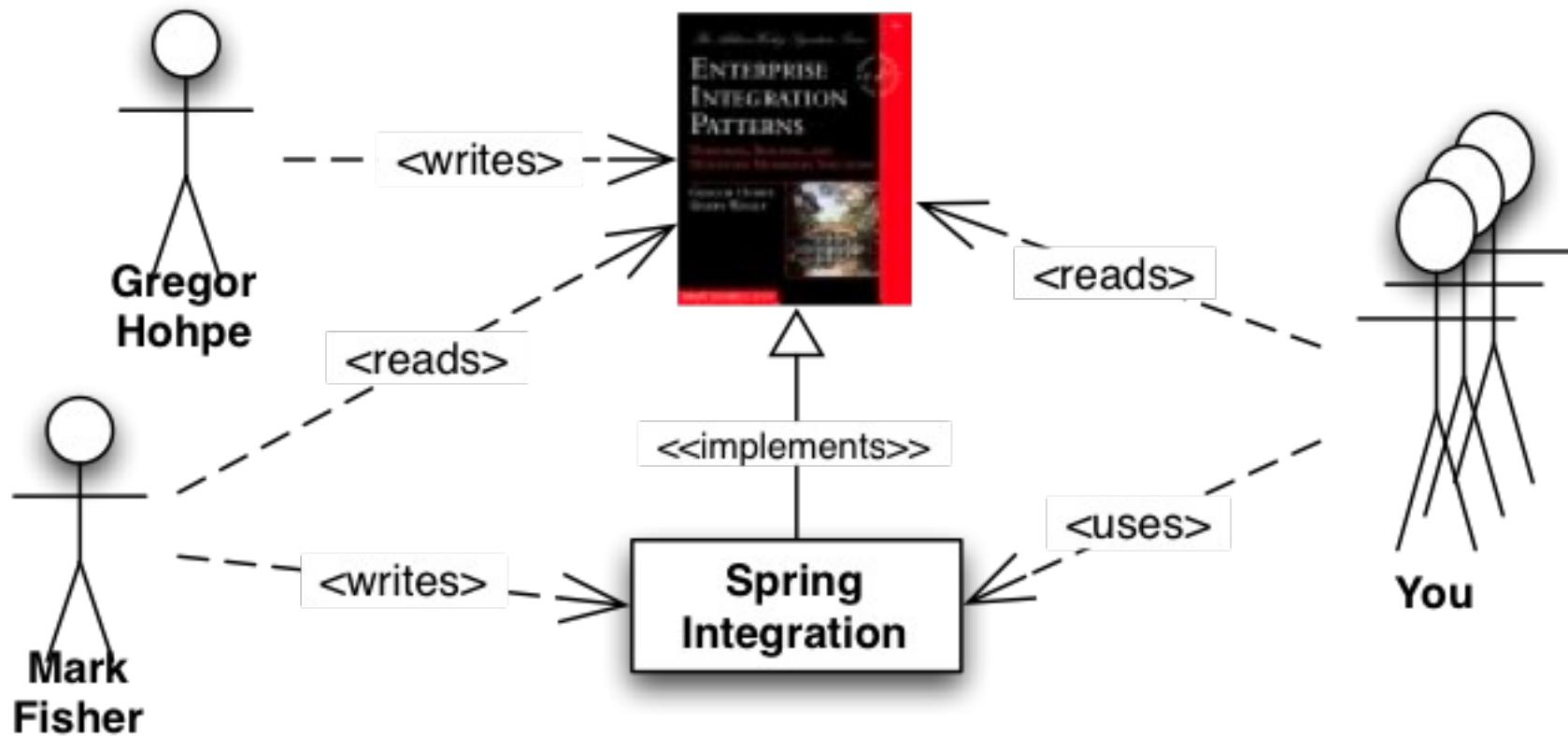
Introduction to Spring Integration

Enterprise Integration Patterns with Spring Integration

Topics in this session

- **Spring Integration – Goals**
- Spring Integration
 - Basics
 - External integration
 - Visual Editor (STS)
- Summary

Spring Integration



Spring Integration allows you to:

- 1) Let application components exchange data through in-memory messaging
 - 2) Integrate with external applications in a variety of ways through adapters
-
- Builds on Enterprise Integration Patterns for both
 - Builds on the Spring portfolio & programming model

Spring Integration - Benefits

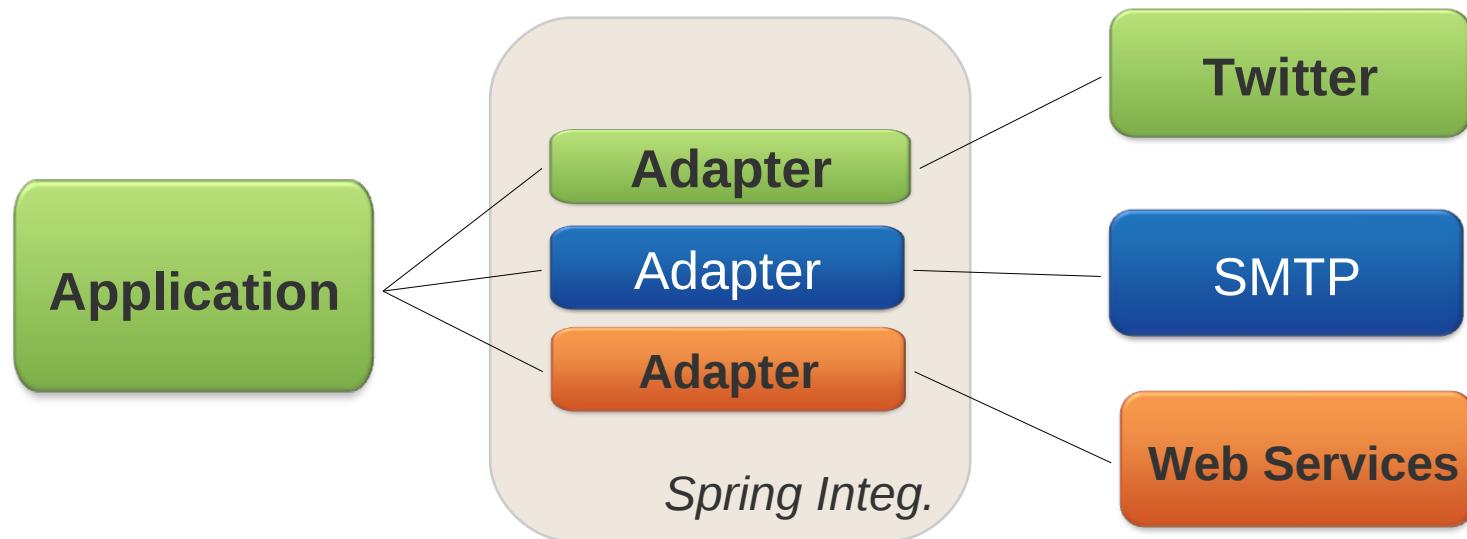


- Loose coupling between components
 - Small, focused components
 - Eases testing, reuse, etc.
- Event-driven architecture
 - No hard-coded process flow
 - Easy to change or expand
- Separates integration and processing logic
 - Framework handles routing, transformation, etc.
 - Easily switch between sync & async processing

Spring Integration - Adapters



- Connect your application to the outside world
 - Remoting, REST, WS, File & FTP, SMTP, Twitter, ...
 - For accepting input and producing output



- Adapters shield application components from integration details
 - External events produce incoming message
 - Incoming email, new file, SOAP request, ...
 - Components just deal with (payload of) messages
 - Don't care about message origin nor destination
 - Internal message can trigger external event
 - Calling a service, sending JMS message or email, ...

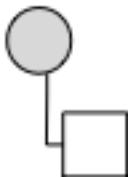
Topics in this session

- EIP Reminder
- Spring Integration – Goals
- Event driven architecture
- **Spring Integration**
 - **Basics**
 - External integration
 - Visual Editor (STS)
- Summary

Ground rules

- Simple core API:
- A **Message** is sent by an **endpoint**
- **Endpoints** are connected to each other using **MessageChannels**
- An **endpoint** can receive **Messages** from a **MessageChannel**
 - By subscribing (passive) or polling (active)

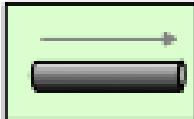




Message



- A **Message** consists of **MessageHeaders** and a **payload**
 - Some headers are pre-defined
 - Payload is just a Java object
- A Message is immutable
 - Let framework wrap payload for you or use a **MessageBuilder** to create it
- Each Message is created with unique ID



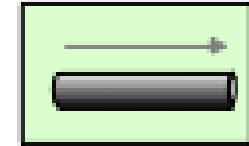
MessageChannels



- Central to loose coupling
 - Runtime IoC
- Connect message endpoints
- Optional buffering, interception
- Just Spring Beans
 - No broker needed
 - No persistence by default
 - May be backed by JMS or JDBC Message Store

Point-to-point

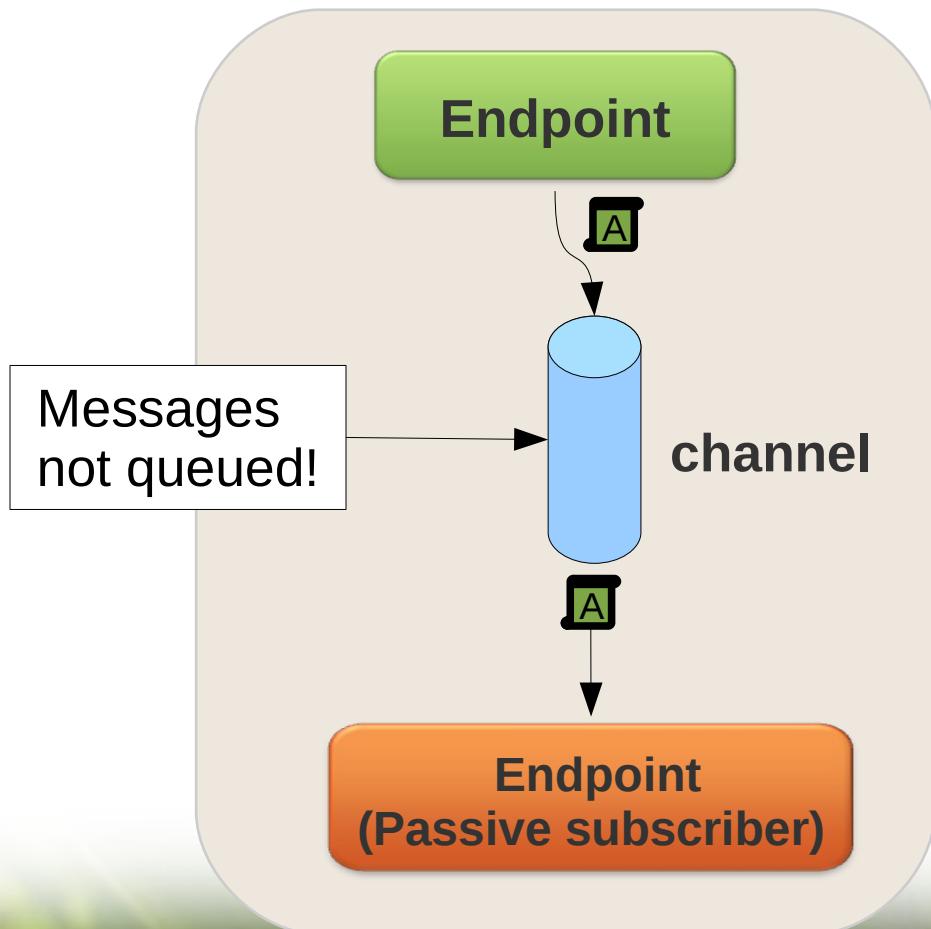
- Only one receiver per message
- DirectChannel
 - Message passed to receiver in sender's thread
 - Sending blocks, not asynchronous!
- QueueChannel
 - Message is queued, sending doesn't block
 - Receiver polls from a different thread



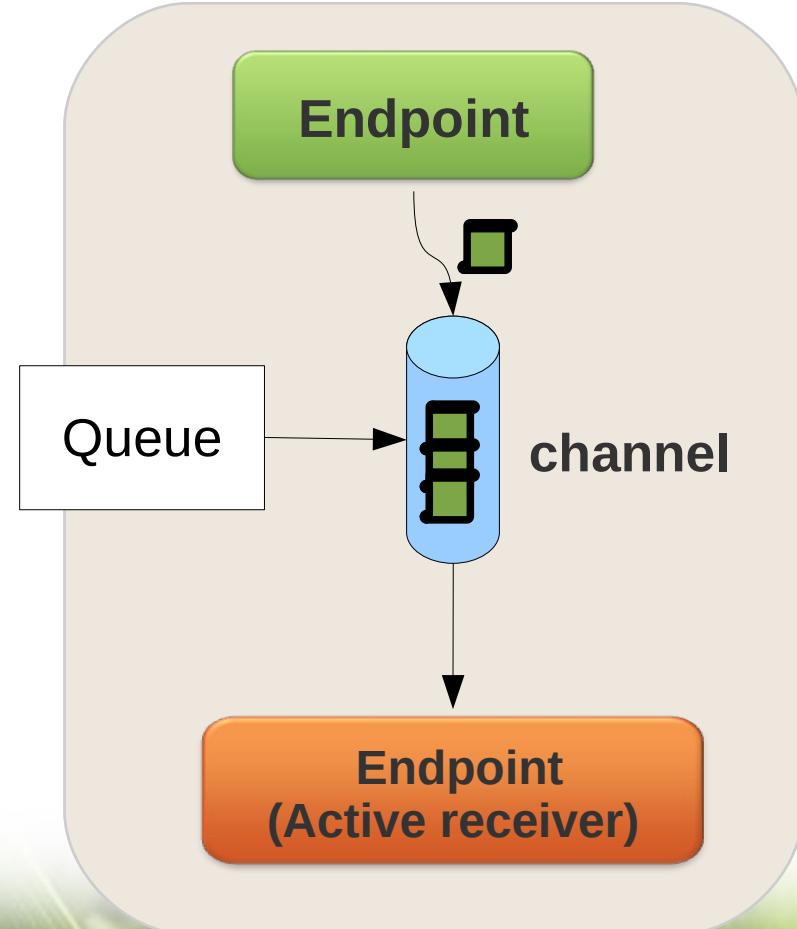
Direct vs Queue Channel



DIRECT CHANNEL

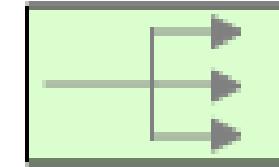


QUEUE CHANNEL



Publish-subscribe

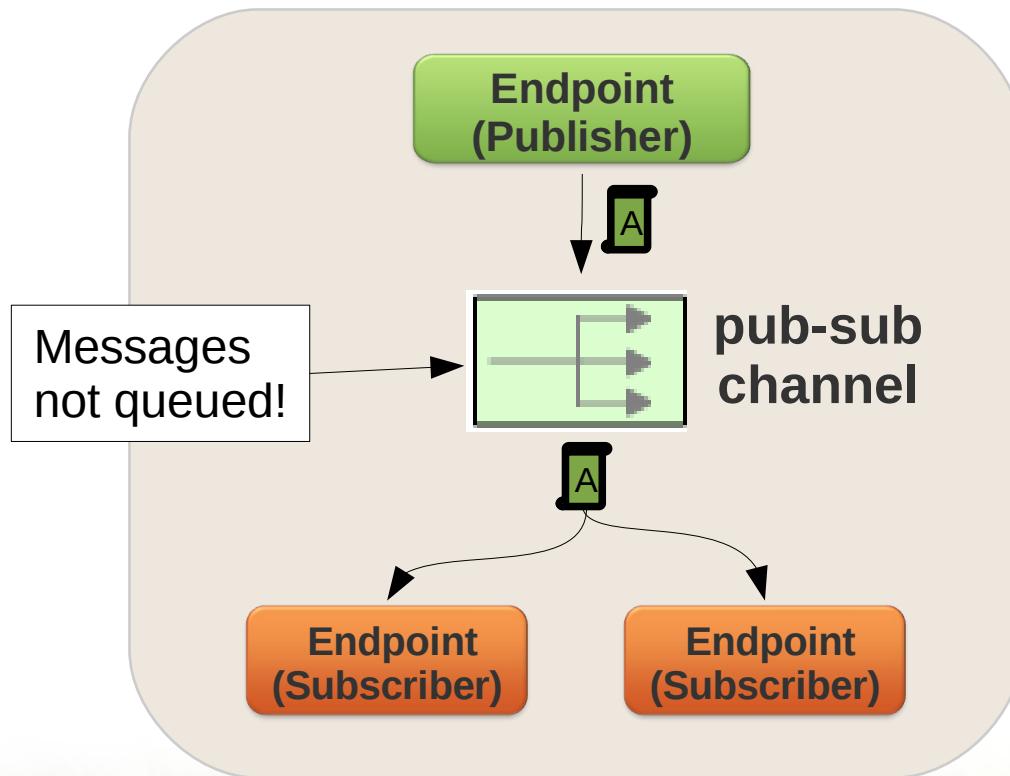
- Multiple receivers per message
- PublishSubscribeChannel
 - Receivers invoked consecutively in sender's thread
 - Or invoked in parallel on different threads using TaskExecutor



MessageChannel Types



Publish-subscribe



Defining Channels



DirectChannel (sync)

```
<channel id="incoming"/>
```

QueueChannel (async)

```
<channel id="orderedNotifications">
    <queue capacity="10"/>
</channel>
```

PublishSubscribeChannel (sync)

```
<publish-subscribe-channel id="statistics"/>
```

PublishSubscribeChannel (async)

```
<publish-subscribe-channel id="appEvents"
    task-executor="pubSubExecutor"/>
<task:executor id="pubSubExecutor" pool-size="10"/>
```



Samples assume spring-integration namespace is the default

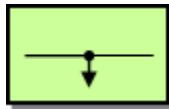
Channel Interceptors

- Can operate on Messages
 - pre-/postSend, pre-/postReceive

```
<channel id="intercepted">
    <interceptors>
        <ref bean="someInterceptorImplementation"/>
    <interceptors>
</channel>
```

- Selective global interceptor

```
<channel-interceptor ref="interceptorForXChannels"
    pattern="x*" />
```



Wire Tap



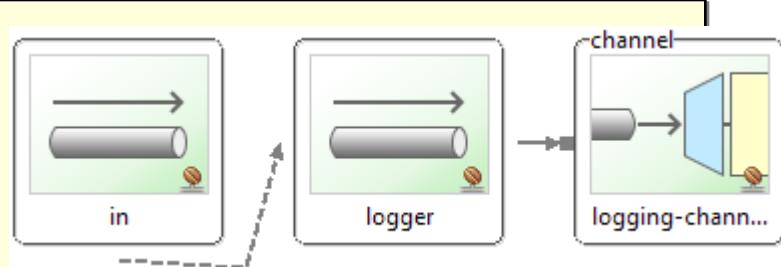
- Standard pattern implemented as interceptor
- Copies incoming messages to given channel
 - 'Spy' on channel, good for debugging and monitoring
 - Often used with logging channel adapter

```
<channel id="in">
    <interceptors>
        <wire-tap channel="logger"/>
    </interceptors>
</channel>

<channel id="logger"/>

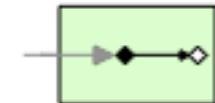
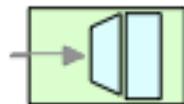
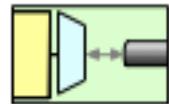
<logging-channel-adapter channel="logger" level="DEBUG"
    log-full-message="true"/>
```

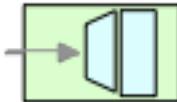
not just payload



Message Endpoints

- **Channel Adapter:** One way integration
 - message enters or leaves application
 - Called 'inbound' or 'outbound'
- **Gateway:** Two way integration
 - Bring message into application and wait for response (inbound), or invoke external system and feed response back into application (outbound)
- **Service Activator**
 - Call method and wrap result in response message
 - Basically outbound gateway for invoking bean method





Messaging Gateway



```
<gateway id="orderService"
    service-interface="com.example.OrderService"
    default-request-channel="orders" />
```

```
public interface OrderService {
    public OrderConfirmation submitOrder(Order order);
}
```

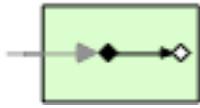
- Proxy for sending new messages
 - Code doesn't depend on SI API
- Temporary reply channel created automatically
 - Can also specify default-response-channel

Gateway Method Signatures



- Gateway interface methods may return Future
 - Becomes async, non-blocking gateway then
- 'void' methods also supported
 - Still called messaging *gateway*, even though that's technically a passive inbound *adapter* (as it's one-way)
- Use annotations for per-method configuration

```
public interface OrderService {  
    @Gateway(requestChannel="orderChannel")  
    public Future<Confirmation> submitOrder(Order order);  
  
    @Gateway(requestChannel="cancellationChannel")  
    public void cancelOrder(Order order);  
}
```



Service Activator



```
<service-activator ref="orderProcessor"  
    input-channel="orders" output-channel="confirmations" />  
  
<beans:bean id="orderProcessor" class="broker.OrderProcessor" />
```

```
public class OrderProcessor {  
    public OrderConfirmation processOrder(Order order) {  
        ...  
    }  
}
```

- Invoke any bean method for incoming message
 - Specify method attribute or @ServiceActivator if there are multiple methods

Service Activator Methods



- 'void' and null-returning methods also supported
 - No response message then (one-way)
 - <outbound-channel-adaptor> can be used as alternative for void methods
- Can cause problems when inbound gateway expects reply message
 - Set **requires-reply** to **true** to throw an exception on null

Topics in this session

- EIP Reminder
- Spring Integration – Goals
- Event driven architecture
- **Spring Integration**
 - Basics
 - **External integration**
 - Visual Editor (STS)
- Summary

Gateways and Adapters



- Remember the difference:
 - Channel Adapter is one way (in or out)
 - Inbound Gateway awaits internal reply and returns it in-band
 - Outbound Gateway awaits external response and puts it on a channel in invoking Thread
- Often use or add message headers
 - inbound HTTP: copies request header to SI headers
 - outbound JMS: copies SI headers to JMS headers

Temporary Reply Channels (1)



- Temporary reply channels created automatically for inbound gateways if not explicitly defined
 - Anonymous point-to-point channel
 - Set as 'replyChannel' message header
- Used by components that produce output when no explicit output channel is provided
 - e.g. outbound gateways, service activators
 - Output becomes reply message
- Reply channel removed automatically after receiving reply message

Temporary Reply Channels (2)



- Advice: only define explicit reply channel when you need a channel definition
 - i.e. to refer to the channel by name and/or to change its type from point-to-point to publish-subscribe
- Rely on default temporary reply channel otherwise

Integration Namespaces

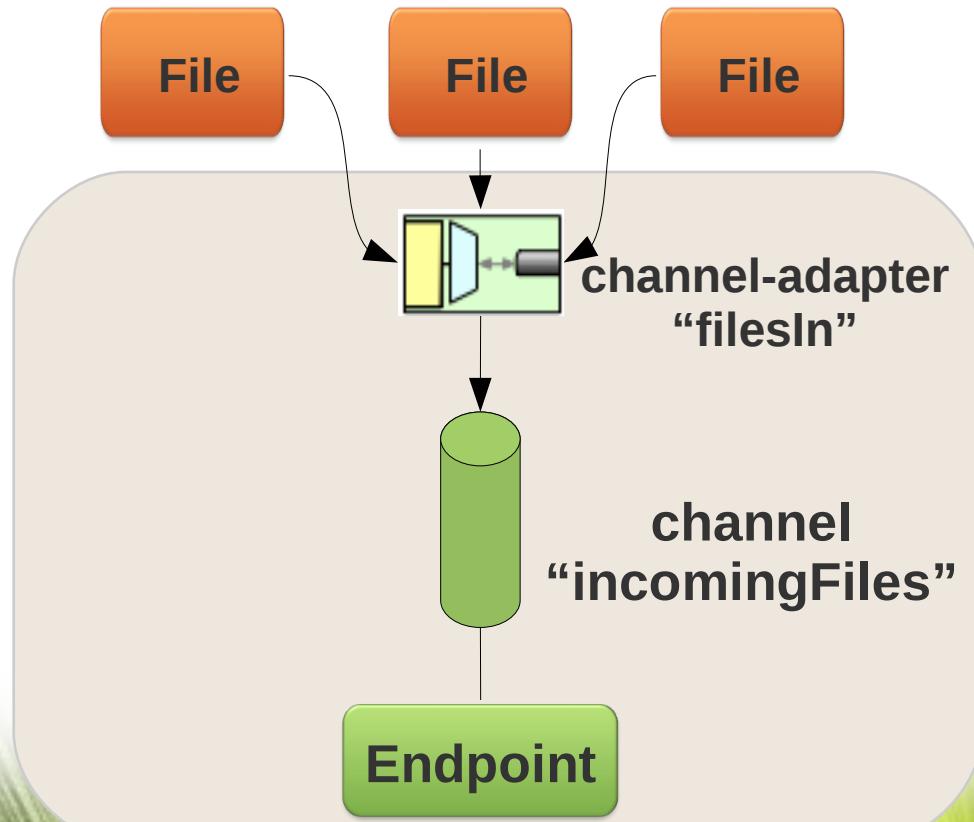


- Spring Integration has dedicated namespaces for different integration types
 - `http://..sfw../schema/integration/file`
 - `http://..sfw../schema/integration/http`
 - `http://..sfw../schema/integration/xml`
 - `http://..sfw../schema/integration/jms`
 - `http://..sfw../schema/integration/ip`
 - `http://..sfw../schema/integration/xmpp`
 - `http://..sfw../schema/integration/twitter`
 - ...

Sample: Inbound File Adapter



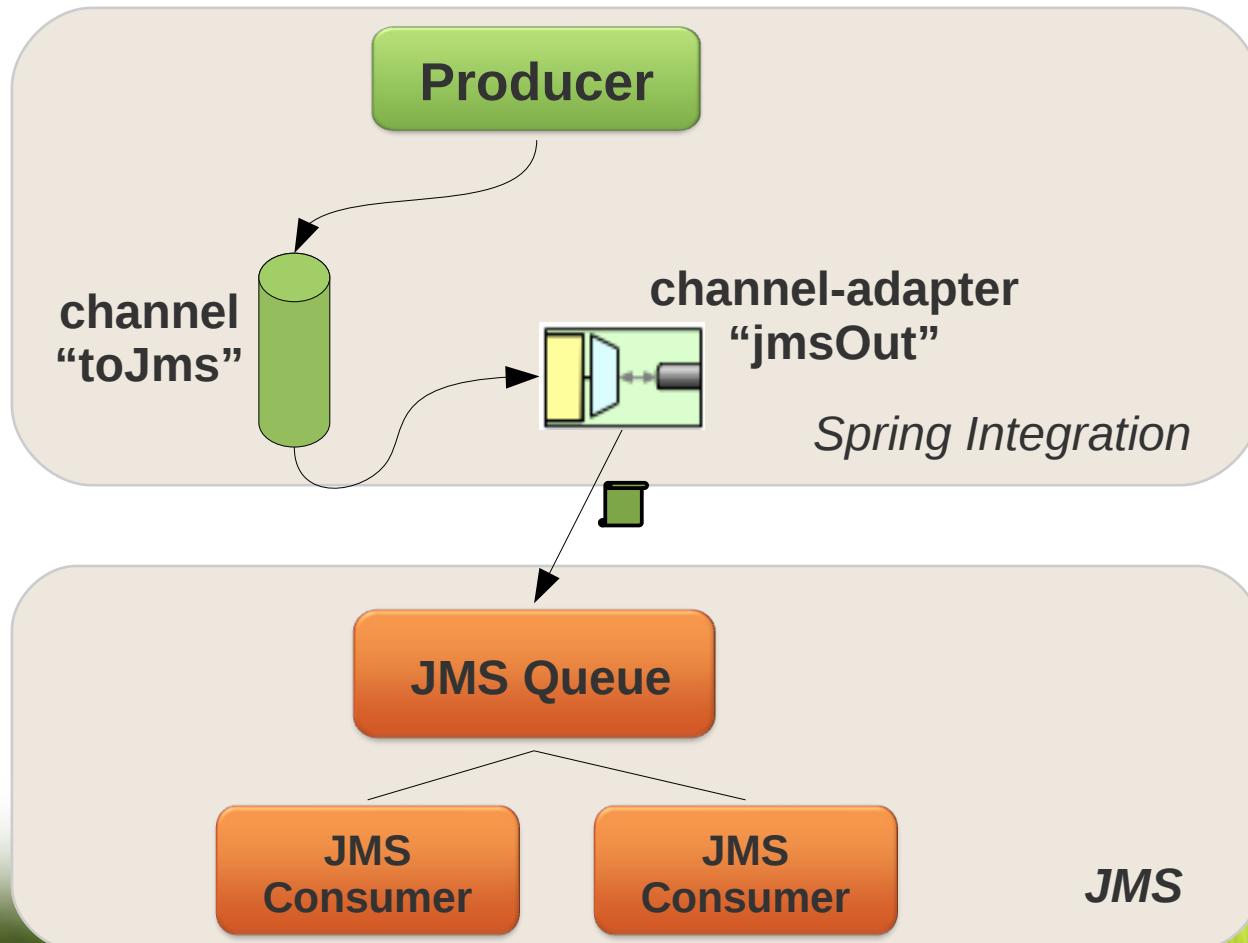
```
<int-file:inbound-channel-adapter id="filesIn"  
    channel="incomingFiles"  
    directory="file:C:/inputResource/" />
```



Sample: Outbound JMS Adapter



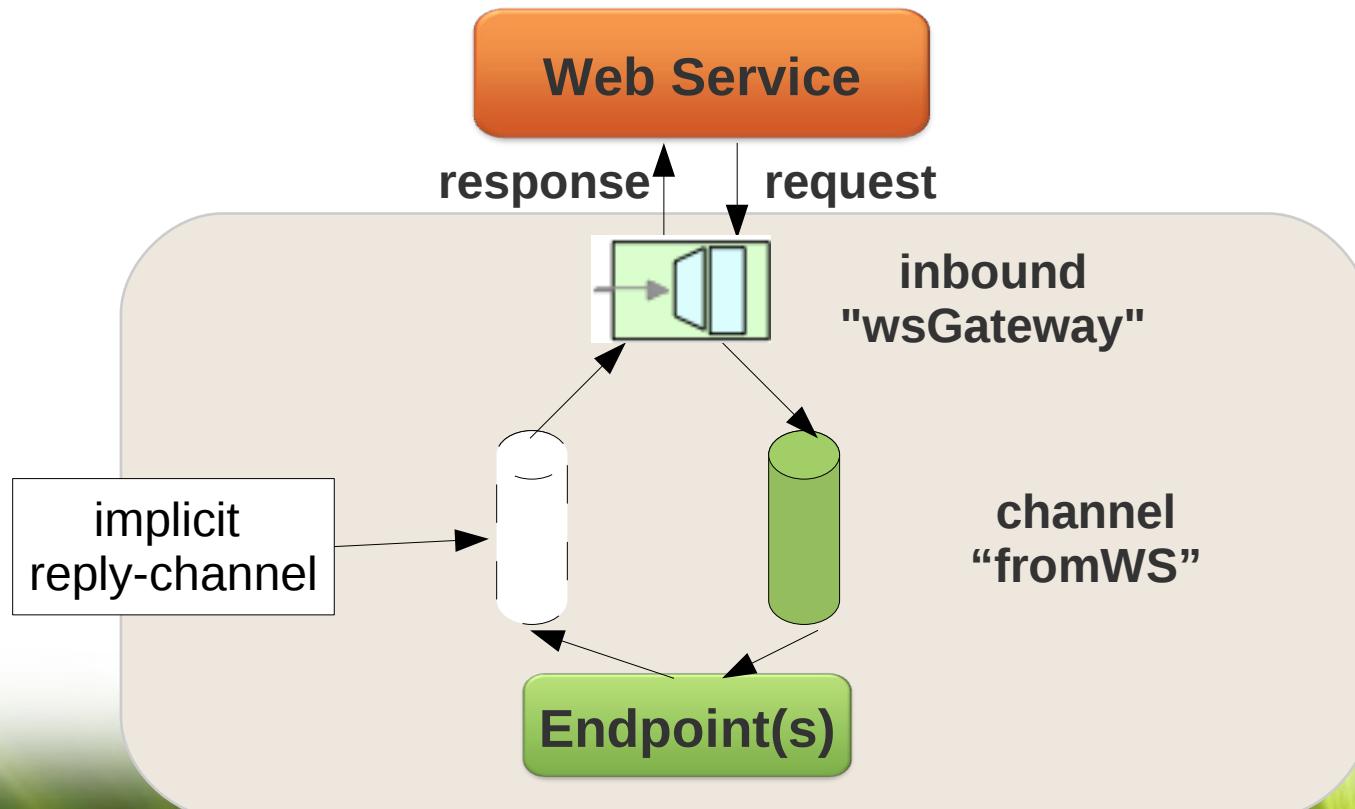
```
<int-jms:outbound-channel-adapter id="jmsOut"  
channel="toJms" destination="jmsQueue" />
```



Sample: Inbound Web Service Gateway



```
<int-ws:inbound-gateway id="wsGateway" channel="fromWS"  
marshaller="jaxb2" unmarshaller="jaxb2" />  
<oxm:jaxb2-marshaller id="jaxb2" contextPath="com.example.xml"/>
```



Sample: Combining Components



```
<gateway default-request-channel="new-orders"  
        service-interface="com.example.OrderService"/>  
  
<publish-subscribe-channel id="new-orders"/>  
  
<service-activator input-channel="new-orders" requires-reply="true"  
        ref="orderProcessor" method="processOrder" />  
  
<int-jms:outbound-channel-adapter channel="new-orders"  
        destination-name="queue.orders"/>
```

Send confirmation to
implicit reply-channel

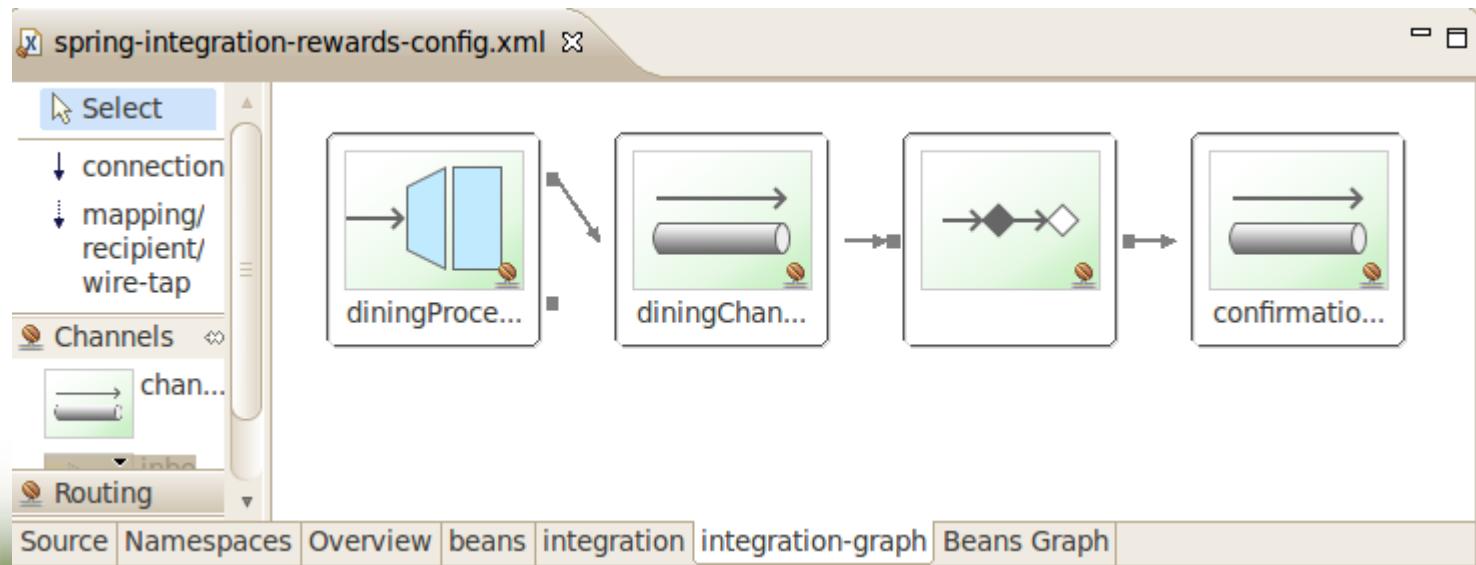
```
@Controller  
public class OrderController {  
  
    @Autowired OrderService orderService; ← Use gateway to interact with SI  
  
    @RequestMapping(value="/orders", method=POST)  
    @ResponseStatus(CREATED)  
    public void placeOrder(@RequestBody Order order,  
                          HttpServletRequest req, HttpServletResponse resp) {  
  
        Confirmation conf = orderService.submitOrder(order);  
  
        ...  
    }
```

Send additional JMS message

Topics in this session

- EIP Reminder
- Spring Integration – Goals
- Event driven architecture
- **Spring Integration**
 - Basics
 - External integration
 - **Visual Editor (STS)**
- Summary

- SpringSource Tool Suite includes a visual editor for Spring Integration flows
- Select 'integration-graph' tab on open Spring Integration configuration file



Summary

- Spring Integration provides the base components to implement EIP
 - To integrate application components
 - To integrate with external systems
- Allows for loosely coupled, event-driven architecture and separation of integration and processing logic

Lab

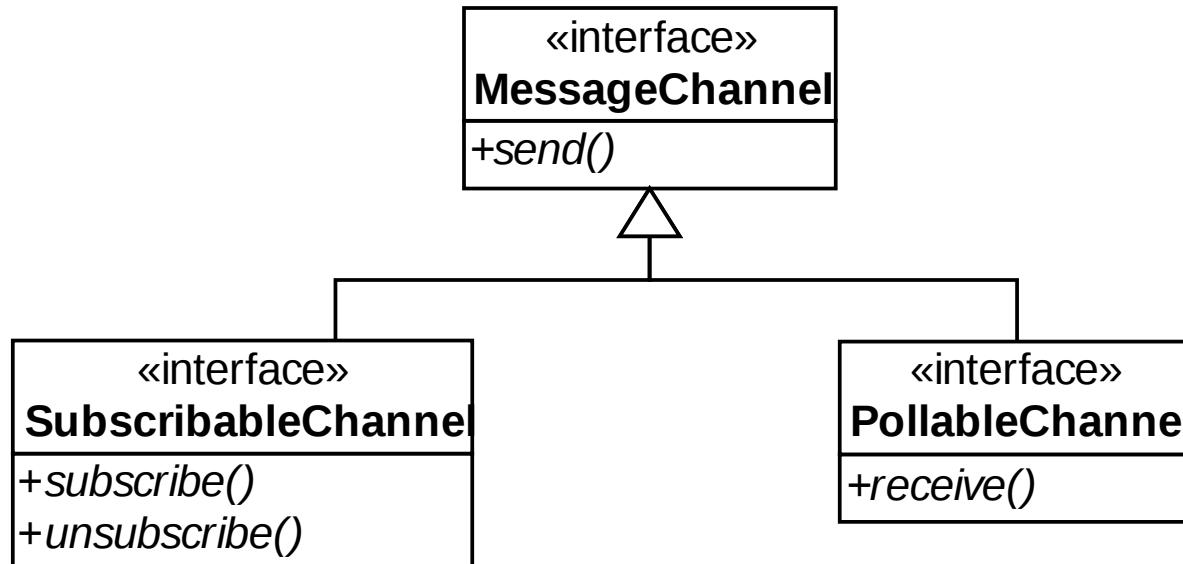
Spring Integration Configuration Overview

Configuring a working Pipes and Filters
architecture

Topics

- **Channel Types and Polling**
- Synchronous vs. asynchronous handoff
- Error handling
- More Endpoint types
- Simplifying configuration

Types of MessageChannels



- SubscribableChannels invoke subscribers on send()
- PollableChannels wait for receive() method to be called (typically from another thread)

MessageChannel Implementations



- SubscribableChannel
 - DirectChannel direct point-to-point
 - PublishSubscribeChannel pub-sub, optional TaskExecutor
 - ExecutorChannel point-to-point via TaskExecutor
- PollableChannel (always point-to-point)
 - Queue Channel
 - Priority Channel FIFO buffering queue
 - Rendezvous Channel priority-ordered
 - NullChannel zero-capacity, ensures handoff /dev/null

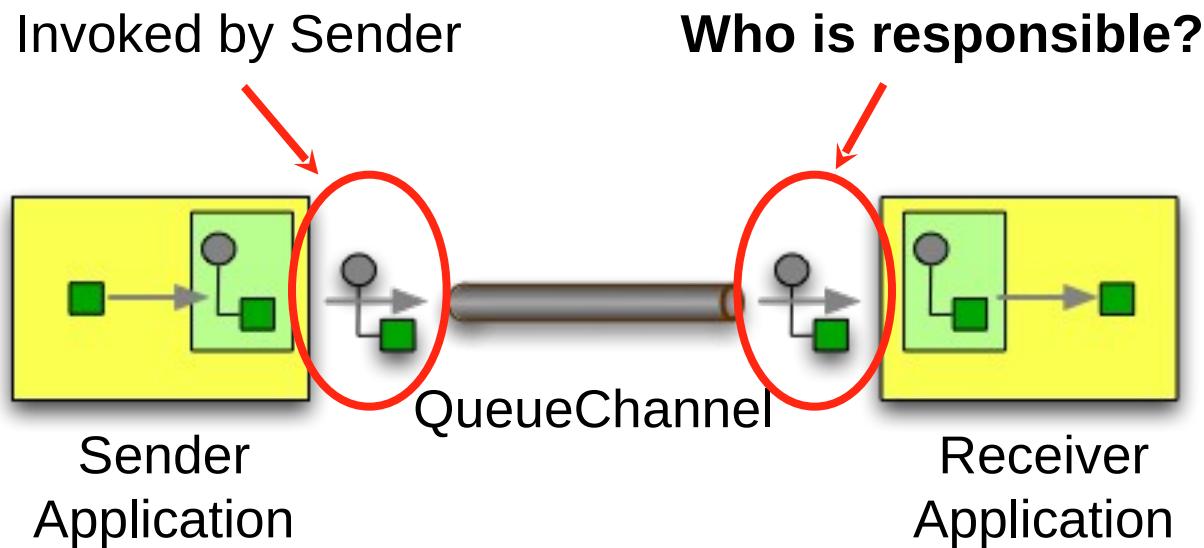


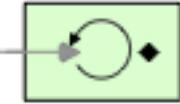
This overview does not show the complete hierarchy

Passive Components

- Most application components are passive
 - Controllers, services, repositories etc.
 - Just wait to be invoked
 - Container takes care of threading
- Channels are also passive!
 - SubscribableChannels call passive subscriber(s) directly when send(Message) is called
 - But when sending to a PollableChannel, something must actively receive the message

Missing Link: Who Picks Up New Messages?





Active Components



- Endpoints can be made active using a **poller**
- By default single thread does the polling
 - Optionally use a thread pool

```
<poller default="true" task-executor="pool"  
fixed-delay="500"/>  
  
<task:executor id="pool" pool-size="9" />
```

milliseconds by default

Endpoints Wired with Poller



- Endpoints use default poller when needed
 - i.e. for PollableChannels as input
- Overridable per endpoint

```
<service-activator ... >
  <poller task-executor="smallPool" fixed-delay="500"/>
</service-activator>

<task:executor id="smallPool" pool-size="3" />
```

Transactional Pollers

- Transactional pollers make message handling flow atomic
 - Assuming flow is single threaded
 - TX spans receive() call and handler invocation
- Configurable on the poller
 - Same configuration options and defaults as Spring

```
<service-activator ... >
  <poller fixed-rate="1000">
    <transactional/>
  </poller>
</service-activator>
```

Topics

- Channel Types and Polling
- **Synchronous vs. asynchronous handoff**
- Error handling
- More Endpoint types
- Simplifying configuration

Synchronous Handoff

- DirectChannels and PublishSubscribeChannels without Executors perform synchronous handoff
- Sending thread invokes receiver(s) directly
- Just like a regular method call:
 - Transactions work
 - Security context available
 - Exceptions are simply propagated back to caller
 - Low overhead
 - But may not scale as sending thread is held up

```
<channel id="direct"/>
```

```
<publish-subscribe-channel id="pubsub"/>
```

- With other channel types receiver runs on different thread than sender
- When breaking thread boundary:
 - Transaction and security context are lost
 - Exceptions can typically not be propagated to caller
- Make no assumptions about temporal (happens-before) relations
 - Avoid relying on them
 - Can still wait for reply message of course

From Sync to Async

- To switch from sync to async simply add a queue or task-executor
- Small change has **profound** effects
 - ensure you understand them well

```
<channel id="queueChannel">  
    <queue capacity="10"/>  
</channel>
```

```
<publish-subscribe-channel id="pubsub"  
    task-executor="taskExecutor"/>
```

Topics

- Channel Types and Polling
- Synchronous vs. asynchronous handoff
- **Error handling**
- More Endpoint types
- Simplifying configuration

Synchronous Error Handling



- Sender receives a `MessageHandlingException` when receiving endpoint throws exception in sender's thread
 - Wraps original exception and failed Message

```
<gateway default-request-channel="in" service-interface="foo.SimpleGateway" />
<service-activator input-channel="in" expression="1/0" />
```

```
SimpleGateway gateway = context.getBean(SimpleGateway.class);
gateway.send("test");
```

```
Exception in thread "main" org.springframework.integration.MessageHandlingException:
Expression evaluation failed: 1/0
```

...

```
Caused by: java.lang.ArithmetricException: / by zero
```

- What if async receiver throws Exception?
 - Sender has already moved on
- MessageHandlingException is sent to error channel
 - Use errorChannel header from request message, set by some endpoint
 - If absent, use global channel called "errorChannel"

```
<gateway default-request-channel="gatewayChannel"  
        service-interface="foo.SimpleGateway"  
        error-channel="exceptionTransformationChannel"/>
```

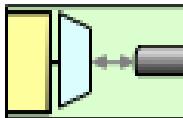
errorChannel

- Created as publish-subscribe-channel by default
 - Define your own "errorChannel" to override, e.g. for asynchronous handling
- Register handler for global error handling
 - Could be simple logger or complete custom flow
 - Built-in router based on exception type:

```
<exception-type-router input-channel="errorChannel"
                      default-output-channel="genericErrors">
    <mapping exception-type="example.OutOfStockException"
            channel="resupplyChannel"/>
    <mapping exception-type="example.InsufficientFundsException"
            channel="loanProposalChannel"/>
</exception-type-router>
```

Topics

- Channel Types and Polling
- Synchronous vs. asynchronous handoff
- Error handling
- **More Endpoint types**
- Simplifying configuration

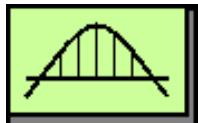


Inbound Channel Adapter



- Method invoking adapter, or transport related implementation
- Often requires polling
 - Unless transport calls subscribed adapter

```
<inbound-channel-adapter channel="inputChannel"
    ref="newWorkSource"
    method="getJob"/>
<poller cron="0 0-5 14 * * *"/>
</inbound-channel-adapter>
```

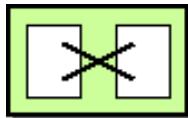


Bridge



- Simply connects two channels or channel adapters
 - e.g. to connect pollable to subscribable channel
- Can have a <poller /> - used to throttle msgs

```
<bridge input-channel="input" output-channel="output" />  
<bridge input-channel="input" output-channel="output" >  
    <poller .../>  
</bridge>  
  
<stream:stdin-channel-adapter id="stdin" />  
<stream:stdout-channel-adapter id="stdout" />  
<bridge id="echo" input-channel="stdin" output-channel="stdout" />
```



Message Transformer



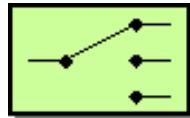
- Basically service activator with specific intent

```
<transformer input-channel="inputChannel"  
            output-channel="outputChannel"  
            ref="anyPOJO" method="doTransform"/>
```

- Common use cases:
 - Payload format *conversion*, incl. (un)marshalling
 - Payload or header *enriching*



Message is immutable, so transformer always creates new Message!



Router



- Decides next channel(s) to send message to
- Typically based on payload or headers
 - Several default implementations available

```
public class OrderRouter {  
    public String routeOrder(Order order) {  
        return stockChecker.inStock(order) ? "warehouse" : "resupply";  
    }  
}
```

```
<router input-channel="orders" ref="orderRouter" method="routeOrder"/>  
<beans:bean id="orderRouter" class="example.OrderRouter" />
```



Filter



- Decides whether to pass or drop a message
- Usually implemented as boolean method
 - True means pass, false means drop

```
public boolean filter(Order order) {  
    return order.isValid();  
}
```

```
<filter ref="orderFilter" method="filter"  
       input-channel="orders" output-channel="validOrders"/>
```

Filter Options

- Default is to silently drop message
- Alternatives:
 - Throw exception on rejection
 - Send message to discard channel
 - 2-way router, or *switch*

```
<filter ref="orderFilter" method="filter" input-channel="orders"
       output-channel="validOrders" throw-exception-on-rejection="true"/>
```

```
<filter ref="orderFilter" method="filter" input-channel="orders"
       output-channel="validOrders" discard-channel="invalidOrders"/>
```

MessagingTemplate



- Using Spring Integration API often not necessary
 - Code shielded from framework through gateways, service activators, etc.
- Sometimes direct Message and/or Message-Channel usage is more convenient
 - Test code, custom Message creation, etc.
- Framework offers MessagingTemplate for this
 - Sending & receiving
 - Similar to JmsTemplate
 - incl. MessageConverter and channel name resolving
 - Uses payload as-is and resolves bean names by default

MessagingTemplate Usage



```
<bean class="org.springframework.integration.core.MessagingTemplate">
  <property name="receiveTimeout" value="1000"/>
</bean>
```

```
@Autowired MessagingTemplate template;
@Autowired MessageChannel someChannel;
```

```
Message<String> msg = MessageBuilder.withPayload("Hello")
  .setHeader("customHeader", someObject)
  .setErrorChannelName("someErrorChannel").build();
template.send(someChannel, msg);
```

```
RewardConfirmation confirmation =
  (RewardConfirmation) template.receiveAndConvert("confirmations");
```

```
// synchronous request-response:
Message responseMsg = template.sendAndReceive("requests", requestMsg);
```

message conversion

named channels

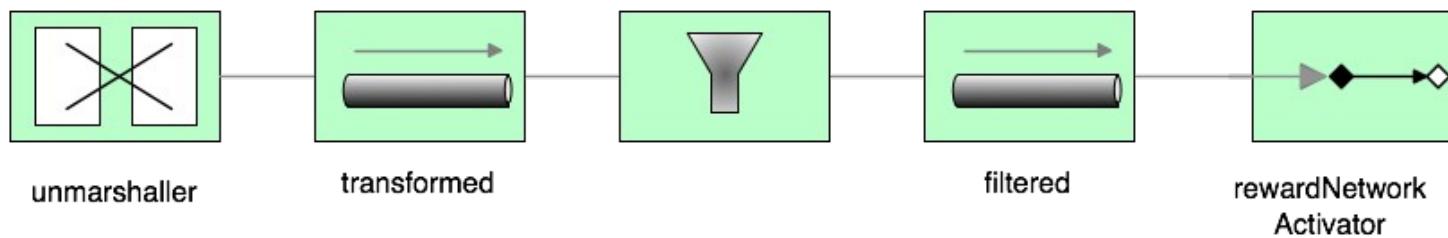
Topics

- Channel Types and Polling
- Synchronous vs. asynchronous handoff
- Error handling
- More Endpoint types
- **Simplifying configuration**

Chaining Endpoints

- Many endpoints work together in a chain
 - Unmarshal
 - Filter out redundant messages
 - Invoke service
- Requires multiple channels to tie them together
- Reduce boilerplate with chains

Consider This Setup



Without Chain

```
<transformer input-channel="input"
             output-channel="transformed"
             ref="unmarshallingTransformer"/>

<channel id="transformed"/>

<filter input-channel="transformed"
        output-channel="filtered" ref="filterBean"/>

<channel id="filtered"/>

<service-activator input-channel="filtered"
                    output-channel="output"
                    ref="rewardNetwork" method="rewardAccountFor" />
```

With Chain

```
<chain input-channel="input" output-channel="output" >
    <transformer ref="unmarshallingTransformer"/>
    <filter ref="filterBean"/>
    <service-activator ref="rewardNetwork"
        method="rewardAccountFor" />
</chain>
```

Chaining Contract

- All endpoints wired with implicit DirectChannels in order
- All endpoints but the last MUST return output
 - Endpoint can filter a message by returning null
- If last endpoint returns something, either output-channel or replyChannel must be set

Endpoint Configuration



- So far most configuration was XML
- Annotations are also supported
 - @MessageEndpoint stereotype
 - Class can be picked up by component-scanning
 - Method annotations for various endpoint types
 - @Gateway, @ServiceActivator, @Router, @Filter, etc.
- As well as Spring Expression Language and Groovy
 - Can even monitor file for dynamic changes
 - Groovy not covered in this course

Method Annotations

- Disambiguate between multiple endpoint methods
 - when XML only has ref attribute
- Specify internal configuration, like channels
- Pass certain or all headers to endpoint methods

```
@MessageEndpoint
public class QuoteService {
    @ServiceActivator(inputChannel="tickers", outputChannel="quotes")
    public Quote lookupQuote(String ticker) { ... }
}
```

```
@Router
public List<String> route(@Header("status") OrderStatus status) { ... }
```

SpEL Expressions



- Many endpoints support expression attribute
- Inline handler logic to avoid writing trivial Java code
- SI variables available: headers, payload

```
<router input-channel="in" expression="payload + 'Channel'">

<filter input-channel="in"
        expression="payload.equals('nonsense')"/>

<service-activator input-channel="in" output-channel="out"
        expression="@accountService.processAccount(payload,
        headers.accountId)">
```

named Spring bean

Summary

- Spring Integration takes care of messaging/threading plumbing code
 - Transparently configures polling
 - Allows passive programming model for active endpoints
- Simple but powerful scheduling
- Declarative configuration for endpoints

LAB

Optimize the Messaging application
using an Idempotent Receiver

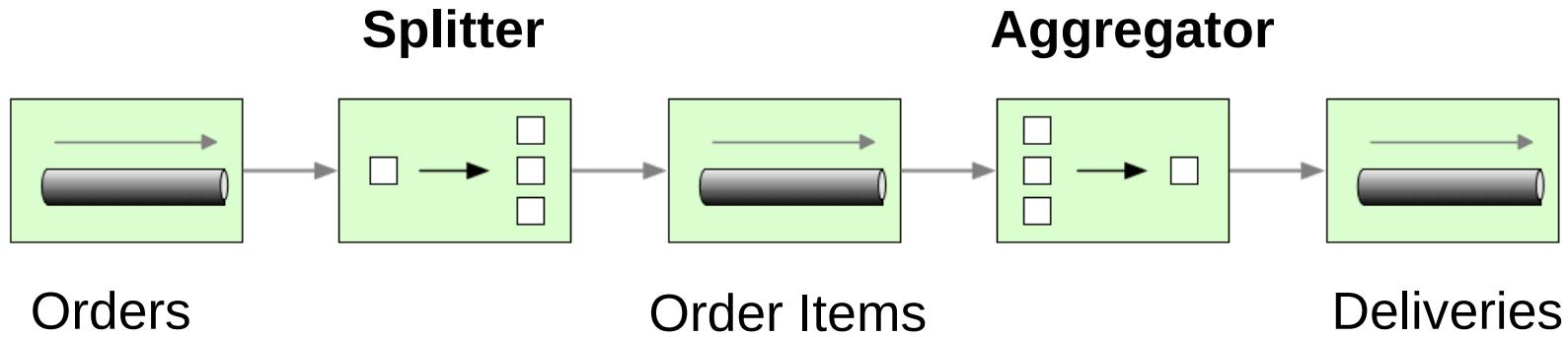
Spring Integration Advanced Features

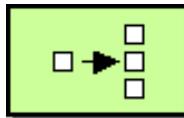
Splitting & aggregating, dispatching
and XML support

Topics

- **Splitting and aggregating**
- Dispatcher configuration
- XML support

Splitting and Aggregating





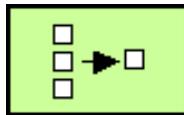
Splitter



- Input: a single message
- Output: multiple messages
- Splitting strategy should be provided in a dedicated method

```
@Splitter  
public List<OrderItem> split(Order order) {  
    return order.getItems();  
}
```

```
<splitter ref="orderSplitter" ... />
```



Aggregator



- Input: multiple messages
- Output: a single message
- Aggregating strategy should be provided in a dedicated method

```
@Aggregator  
public Order combine(List<OrderItem> items) {  
    return Order.withItems(items);  
}
```

```
<aggregator ref="itemAggregator" ... />
```

- Aggregator holds messages until a set of **correlated** messages is ready to **release**
- Correlation strategy
 - What correlates this message?
- Release strategy
 - Is this list of messages complete and ready to be released?



In Spring Integration 1.0, “Release strategy” used to be called “Completion strategy”

Correlation Strategy

- Default: CORRELATION_ID header
- Inject implementation of CorrelationStrategy into aggregator
 - also correlation-method
 - also correlation-expression

```
<aggregator ref="itemAggregator"  
correlation-strategy="correlationStrategyImpl" ... />
```

```
<aggregator ref="personIdAggregator"  
correlation-strategy-expression="payload.person.id" ... />
```

Release Strategy

- Default: SequenceSizeReleaseStrategy
 - SEQUENCE_SIZE header
- Inject implementation of ReleaseStrategy into aggregator
 - also release-strategy-method
 - also release-strategy-expression

```
<aggregator ref="itemAggregator"  
release-strategy="releaseStrategyImpl" ... />
```

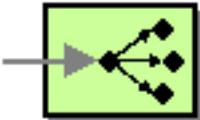
```
<aggregator ref="msgGroupSizeSpELAggregator"  
release-strategy-expression="payload.size() gt 5" ... />
```

Splitter and Aggregator

- Splitter sets CORRELATION_ID to MESSAGE_ID of original message
- Also sets SEQUENCE_* headers
- If you don't change the headers Aggregator **just works** with the products of a Splitter
- Since 2.0, nested splitters work too

Topics

- Splitting and aggregating
- **Dispatcher configuration**
- XML support



Point-to-point Dispatch



- Point-to-point SubscribableChannel ensures single handler per message
- But can still have multiple subscribers
- How is handler determined?
- What happens if handler fails?

Dispatching Defaults

- Default is round-robin load balancer with failover
 - Rotate over subscribers for subsequent messages
 - Call next subscriber if current one throws Exception
- Handlers can have 'order' property set for failover
 - If not, order of subscribing is used

```
<inbound-channel-adapter channel="in" expression=" 'foo' ">
    <poller fixed-rate="100"/>
</inbound-channel-adapter>
<channel id="in"/>
<logging-channel-adapter channel="in" expression=" 'first' "/>
<logging-channel-adapter channel="in" expression=" 'second' "/>
```

Dispatcher Configuration



- Can disable load balancing and/or failover

```
<channel id="in">
  <dispatcher failover="false"/>
</channel>
```

```
<channel id="in">
  <dispatcher load-balancer="none"/>
</channel>
```

- Without failover exception propagates back to caller immediately
- Without load balancer, subscribers are always called in same order
 - Good for 'primary' handlers with fallback
 - Only makes sense with failover enabled

Asynchronous Dispatching



- Can also dispatch using TaskExecutor
 - Defines ExecutorChannel instead of DirectChannel

```
<channel id="in">
    <dispatcher task-executor="someExecutor"/>
</channel>
```

- Sending doesn't block (assuming thread available)
- Subscribers still called from a single thread
 - Which is no longer the caller's thread!
 - failover and/or load-balancer still supported

Topics

- Splitting and aggregating
- Dispatcher configuration
- **XML support**

- XPath:
 - Splitting
 - Routing
 - Filtering
- XSLT
- OXM
- XML Payloads

XPath Support

```
<!-- splits Node into Nodes, String and File into Strings -->
<xml:xpath-splitter input-channel="messagesContainingOrders"
                     output-channel="orders"
                     create-documents="true">
    <xml>xpath-expression expression="//order"/>
</xml:xpath-splitter>

<xml:xpath-router input-channel="orders">
    <xml>xpath-expression expression="/order/@type"/>
</xml:xpath-router>
```

A callout box with a black border and white background points from the 'create-documents="true"' attribute to the text 'turns Nodes into Documents'.

turns Nodes into Documents

XPath Support

```
<filter input-channel="orders" output-channel="typedOrders"
       ref="selector"/>

<xml:xpath-selector id="selector" evaluation-result-type="boolean">
    <xml:xpath-expression expression="boolean(/order/@type)" />
</xml:xpath-selector>
```

XSLT Support

```
<xml:xslt-transformer id="usingResource"  
    input-channel="orders10"  
    output-channel="orders"  
    xsl-resource="file:${xsl-dir}/order1.0to1.1translation.xsl"/>
```

```
<xml:xslt-transformer id="usingTemplates"  
    input-channel="orders10"  
    output-channel="orders"  
    xsl-templates="templatesBean"  
    result-transformer="resultTransformerBean"/>
```

javax.xml.transform.Templates

OXM Support

- Spring Integration builds on top of Spring OXM
- Spring 3.0 object-to-XML abstraction
 - Marshaller and Unmarshaller interfaces
 - Generic XmlMappingException hierarchy
 - Implementations for JAXB 1 & 2, JiBX, XMLBeans, XStream and Castor
 - With namespace support for some

Spring OXM Interfaces



```
public interface Marshaller {  
    void marshal(Object graph, Result result)  
        throws XmlMappingException, IOException;  
}
```

javax.xml.transform.Result

```
public interface Unmarshaller {  
    Object unmarshal(Source source)  
        throws XmlMappingException, IOException;  
}
```

javax.xml.transform.Source

Result and Source may be
DOM, SAX, or Stream

- Marshallers write XML to Result instance
 - DOMResult by default, can be changed
- 'Result' typically bad XML message payload type
 - Needs instanceof checks and downcasting to process
- ResultTransformer converts to suitable type
 - ResultToStringTransformer
 - ResultToDocumentTransformer

Marshalling and Unmarshalling



```
<si-xml:unmarshalling-transformer  
    input-channel="xmlInputChannel"  
    output-channel="objectOutputChannel"  
    unmarshaller="unmarshaller"/>  
  
<si-xml:marshalling-transformer  
    input-channel="objectInputChannel"  
    output-channel="xmlOutputChannel"  
    marshaller="marshaller"  
    result-transformer="resultToStringTransformer"/>  
  
<bean id="resultToStringTransformer"  
    class="org.sfw...xml.transformer.ResultToStringTransformer" />
```

XML Payloads

- XML Message payloads can have various types
- XML endpoints convert where needed
 - All Support Node and String
 - XPath Splitter: plus File
 - XPath Router, Selector, Transformer and Header Enricher:
plus File and DOMSource
 - XML Validating Selector: plus Source
 - XML Unmarshaller: plus File and Source
- For other payloads use explicit transformer

Summary

- Splitting and Aggregating
 - symmetric, customizable
- Configurable dispatching for point-to-point subscribable channels
- XPath, XSLT and OXM support for dealing with XML



Introducing Spring Batch

A Quickstart guide to offline processing
with Spring Batch

Topics in this Session



- **Batch and offline processing**
- Spring Batch high-level overview
- Job parameters and job identity
- Quick start using Spring Batch
- Readers, Writers & Processors
- JDBC Item Readers

Batch Jobs

Differ from online/real-time processing applications:

- Long-running
 - Often outside office hours
- Non-interactive
 - Often include logic for handling errors or restarts
- Process large volumes of data
 - More than fits in memory or a single transaction

Batch and offline processing



- Close of business processing
 - Order processing
 - Business reporting
 - Account reconciliation
- Import/export handling
 - a.k.a. ETL jobs (Extract-Transform-Load)
 - Instrument/position import
 - Data warehouse synchronization
- Large-scale output jobs
 - Loyalty scheme emails
 - Bank statements

- The batch domain adds some value to a plain business process by introducing new concepts:
 - A **job** has an identity – defines what needs to be done
 - A **job** has **steps**
 - A **job instance** can be **restart**ed after a failure – a new **execution**
 - Each **execution** has a **start** time, **stop** time, **status**
 - The **job instance** has an overall **status**
 - Each **execution** can tell us how many **items** were processed, how many **commits**, **rollbacks**, **skips**
- Add value through robustness, reliability, traceability (SLA)

Topics in this Session

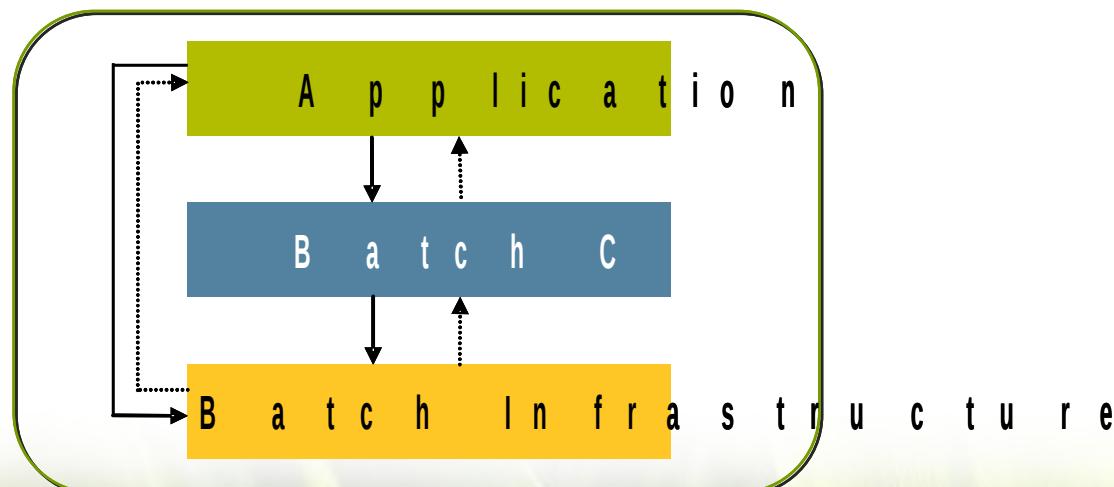


- Batch and offline processing
- **Spring Batch high-level overview**
- Job parameters and job identity
- Quick start using Spring Batch
- Readers, Writers & Processors
- JDBC Item Readers

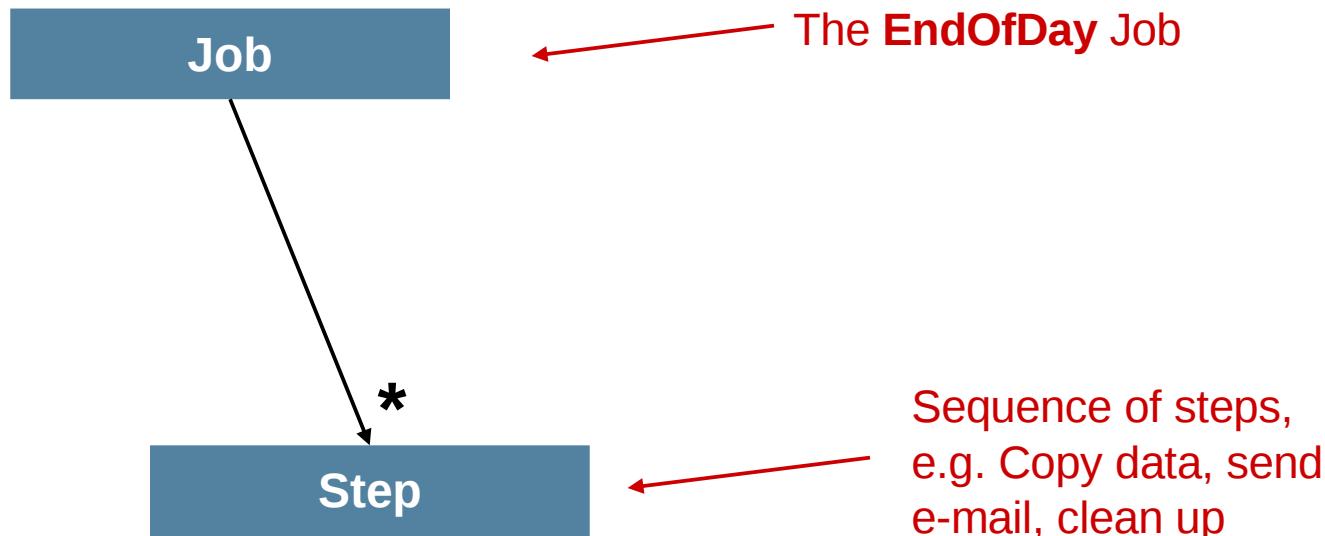
Spring Batch Overview



- The Spring programming model applied to batch processing
- Don't write code that doesn't make you money
- Download
<http://www.springframework.org/spring-batch>



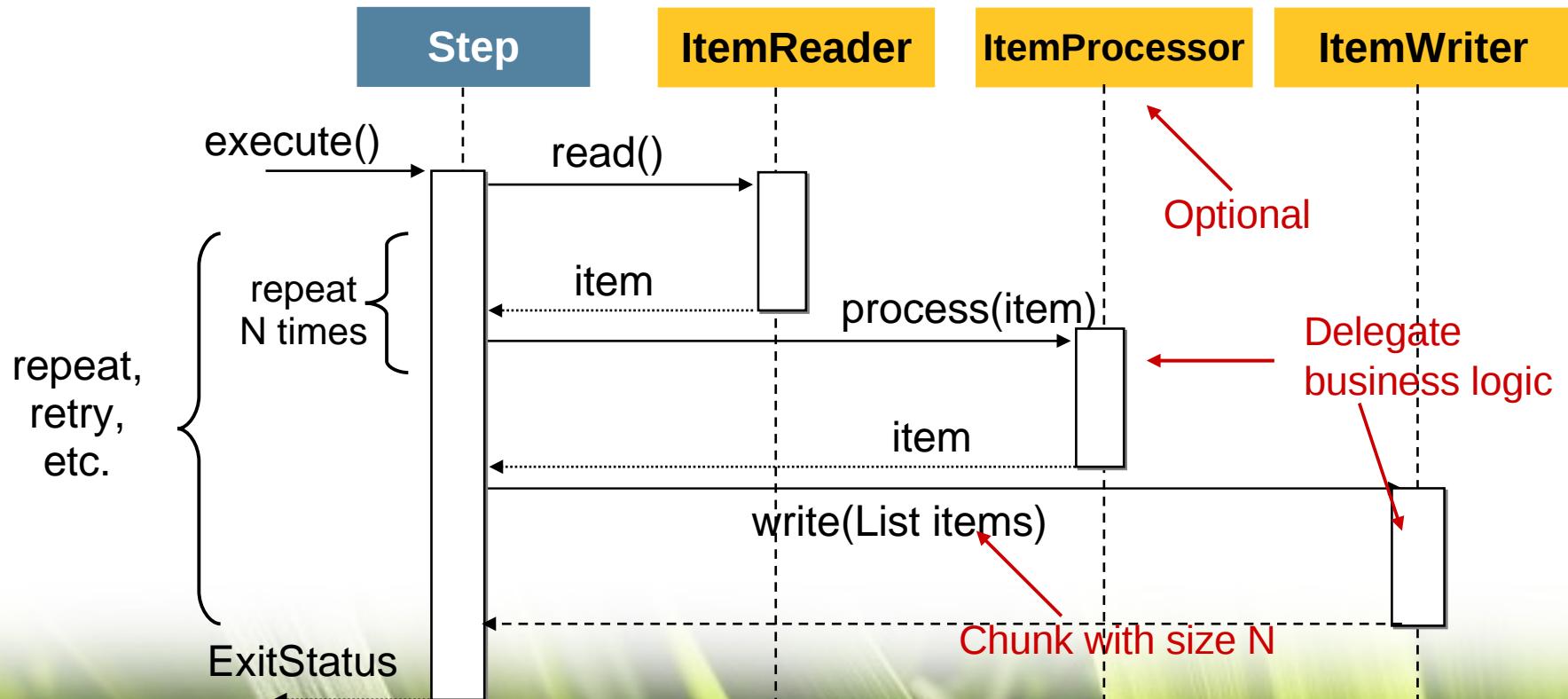
Job and Step



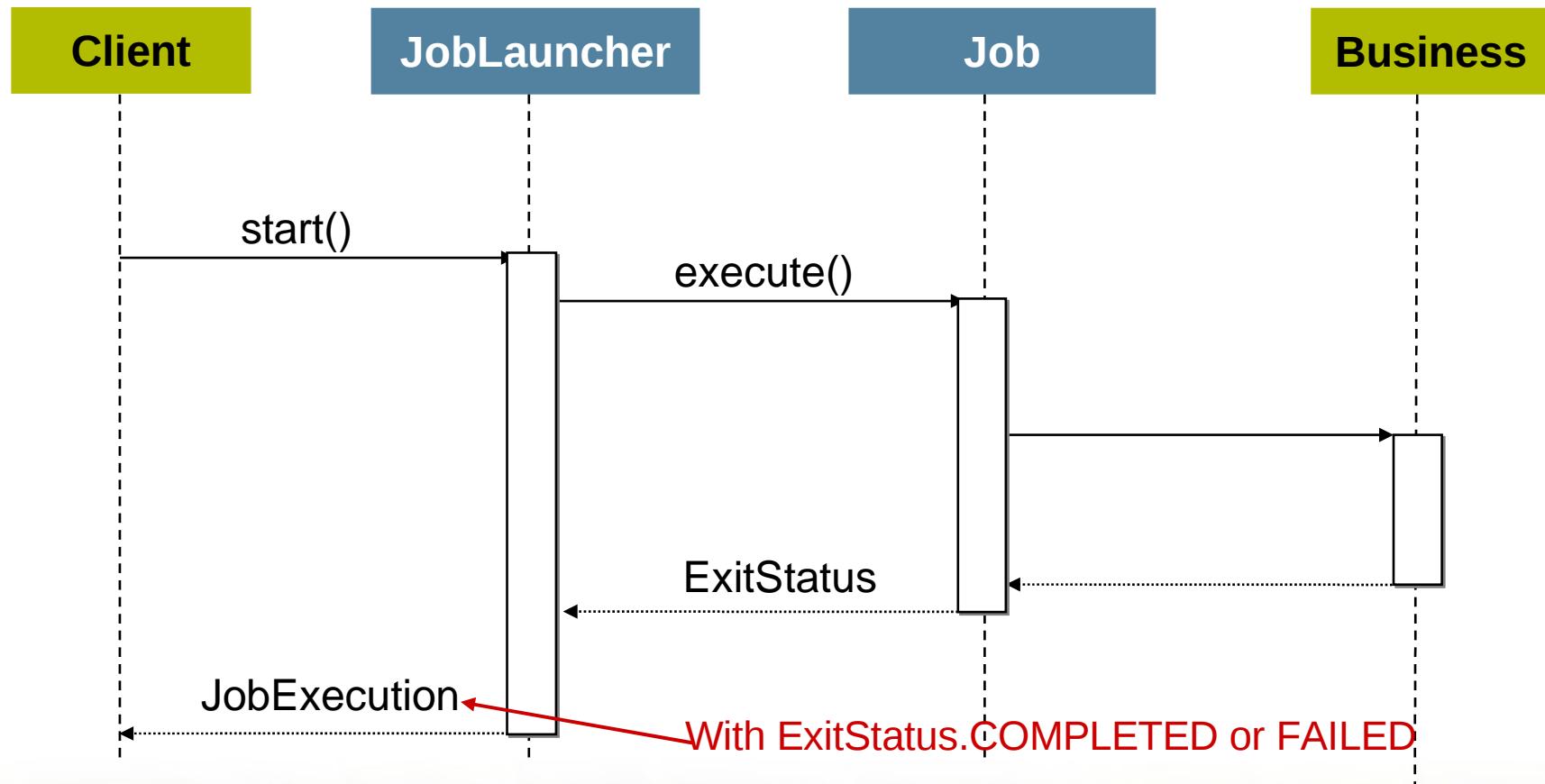
Chunk-Oriented Processing



- Input-output can be grouped together
- Input collects Items before outputting:
Chunk-Oriented Processing
- Optional ItemProcessor



JobLauncher



Topics in this Session



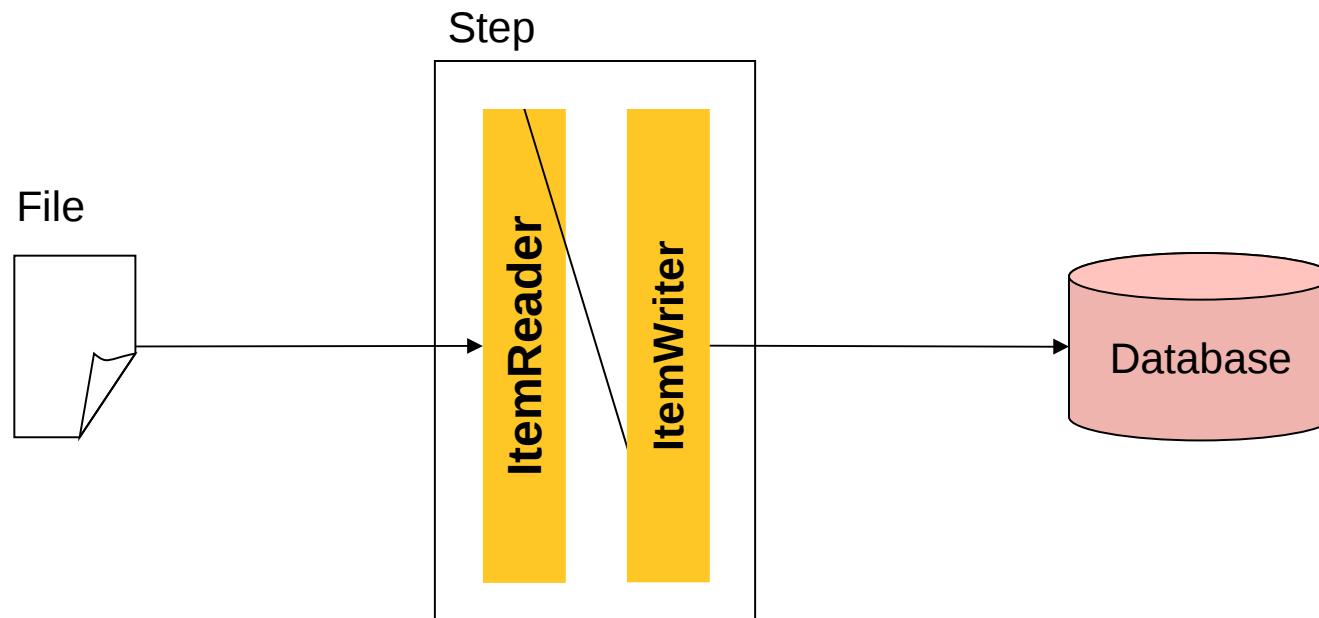
- Batch and offline processing
- Spring Batch high-level overview
- **Readers, Writers & Processors**
- Job parameters and job identity
- Quick start using Spring Batch
- JDBC Item Readers

Readers and Writers

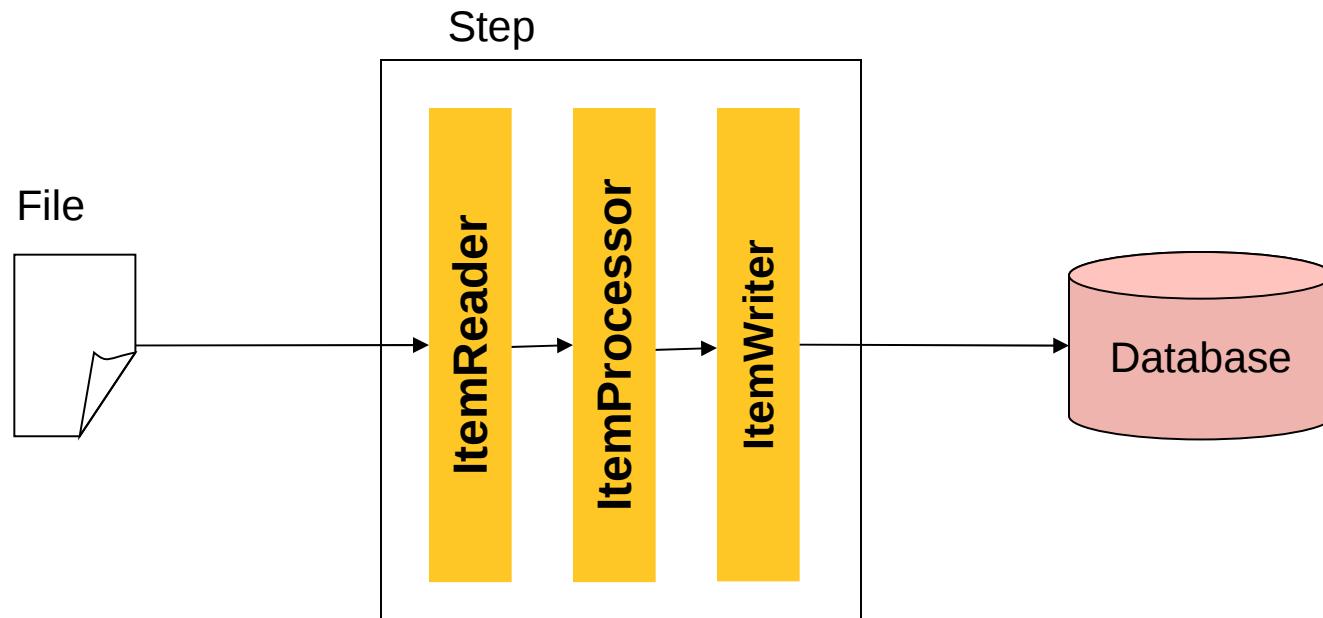


- Spring Batch provides many implementations of ItemReader and ItemWriter, e.g.
 - Flat files
 - XML
 - JDBC: cursor & driving query
 - Hibernate
 - JMS
- Some simple jobs can be implemented with off-the-shelf components

Simple Copy Job



Copy Job with Processing



More Complex Use Cases



- It's very common to use an off-the-shelf reader and writer
- More complex jobs often require custom readers or writers
- ItemProcessor is often used if there's a need to delegate to existing business logic
- Use a writer if it's more efficient to process a complete chunk

ItemReader



- The interface is parameterised, so e.g.

```
public class DiningItemReader implements ItemReader<Dining> {  
  
    public Dining read() {  
        // Read a Dining record from somewhere...  
        return dining;  
    }  
    ...  
}
```

ItemProcessor



- The interface is parameterised
 - perhaps the item that was read is transformed, or other business logic delegation occurred, resulting in an object of a different type

```
public class DiningItemProcessor implements  
    ItemProcessor<XMLDining, Dining> {  
  
    public Dining process(XMLDining xmlDining) {  
        // transform the data...  
        return dining;  
    }  
    ...  
}
```

ItemWriter



- Also parameterised; notice the chunk (i.e. a List of items) as the parameter!

```
public class DiningItemWriter implements ItemWriter<Dining> {

    public void write(List<? extends Dining> dinings) {
        // Business logic here and write out to e.g. database
    }
    ...
}
```

Topics in this Session

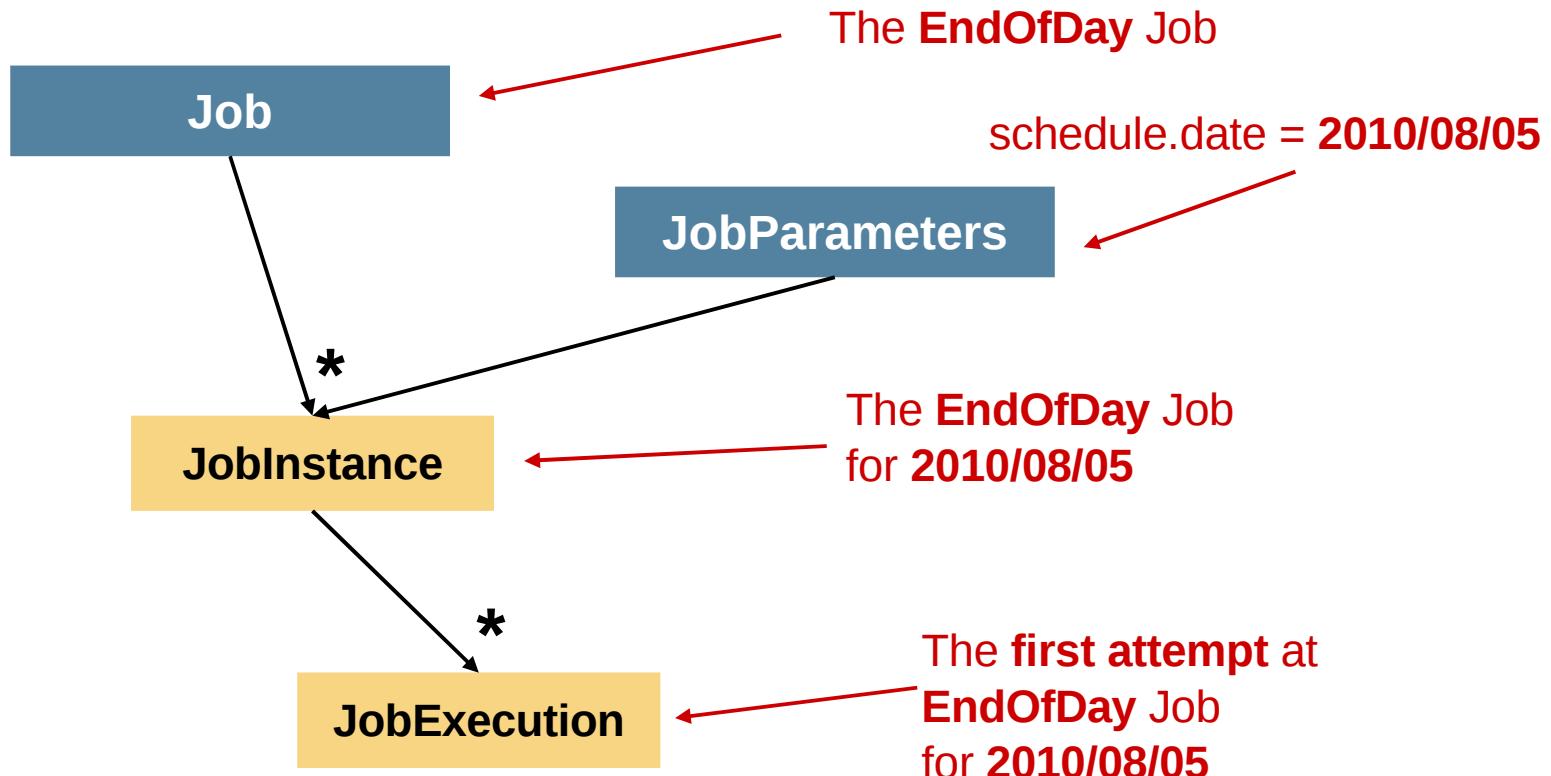


- Batch and offline processing
- Spring Batch high-level overview
- Readers, Writers & Processors
- **Job parameters and job identity**
- Quick start using Spring Batch
- JDBC Item Readers

Job Identity

- When you launch a job, what is it that you launch?
- If it fails how would you identify it?
- How can we track the eventual successful execution?

Job Identity and Parameters



Job Parameters

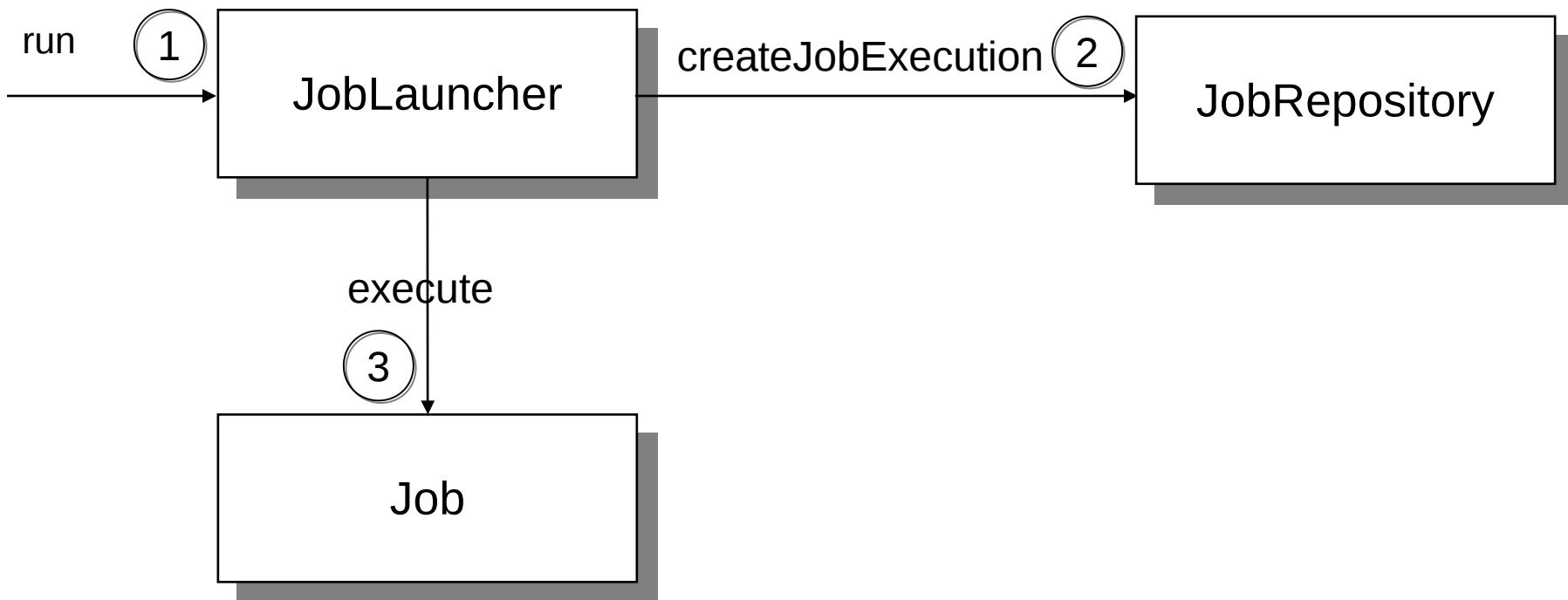
- Unique for each Job Instance
 - JobInstanceAlreadyCompletedException
- Often a natural unique parameter such as run Date/Time
- Can use a JobParametersIncrementer to create uniqueness

Job Persistence



- Need to track status of job executions
- Readers and writers are often stateful, so need to persist e.g. intermediate state
- Batch meta-data stored in
 - Relational database
 - Volatile (map)
- Core interface = JobRepository

JobLauncher and JobRepository



JobRepository Practicalities



- JobRepository is really an **internal** interface
 - no need for users to know details
- Still need to create an instance of JobRepository
- Spring Batch provides namespace support:

```
<!-- defaults to 'transactionManager' for transaction-manager,  
    'dataSource' for data-source and 'BATCH_' for table-prefix -->  
<batch:job-repository id="jobRepository" />
```

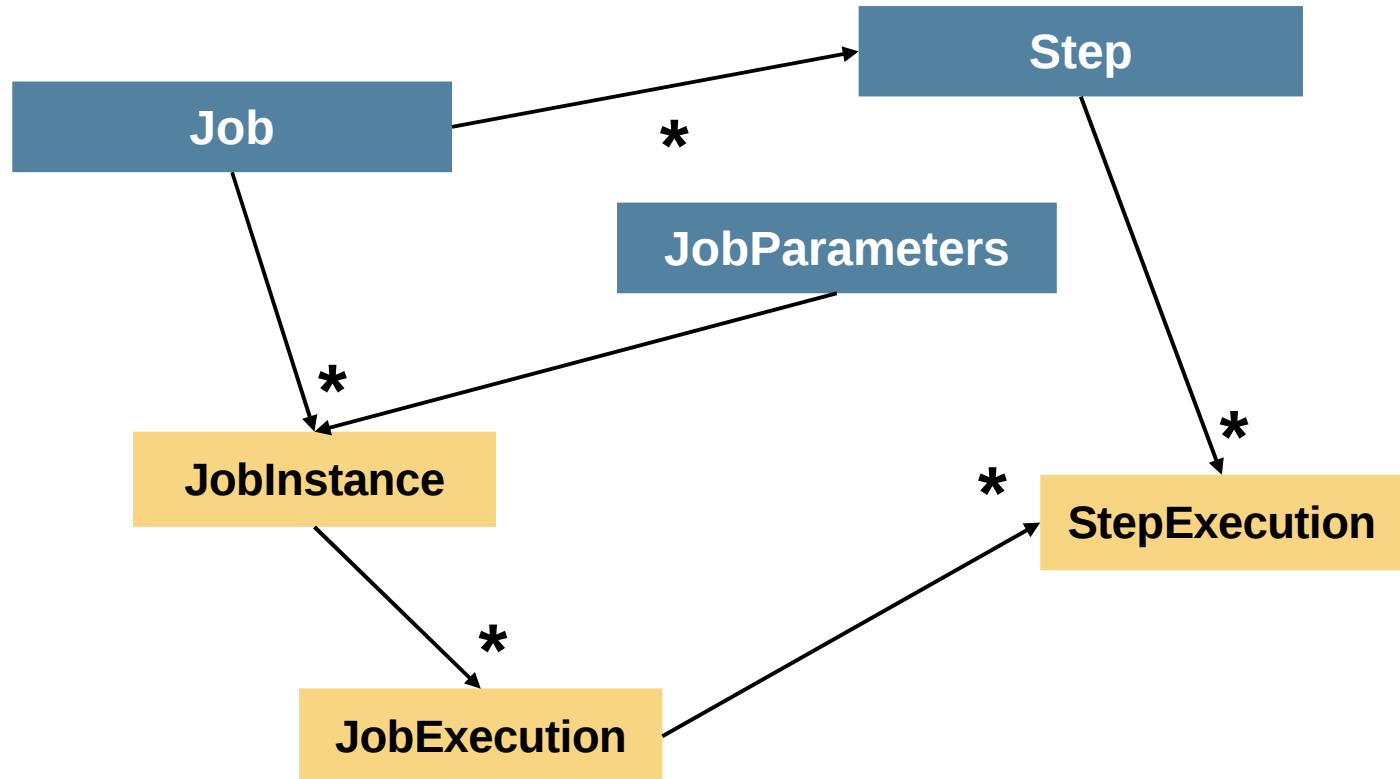
Batch Meta-Data Schema



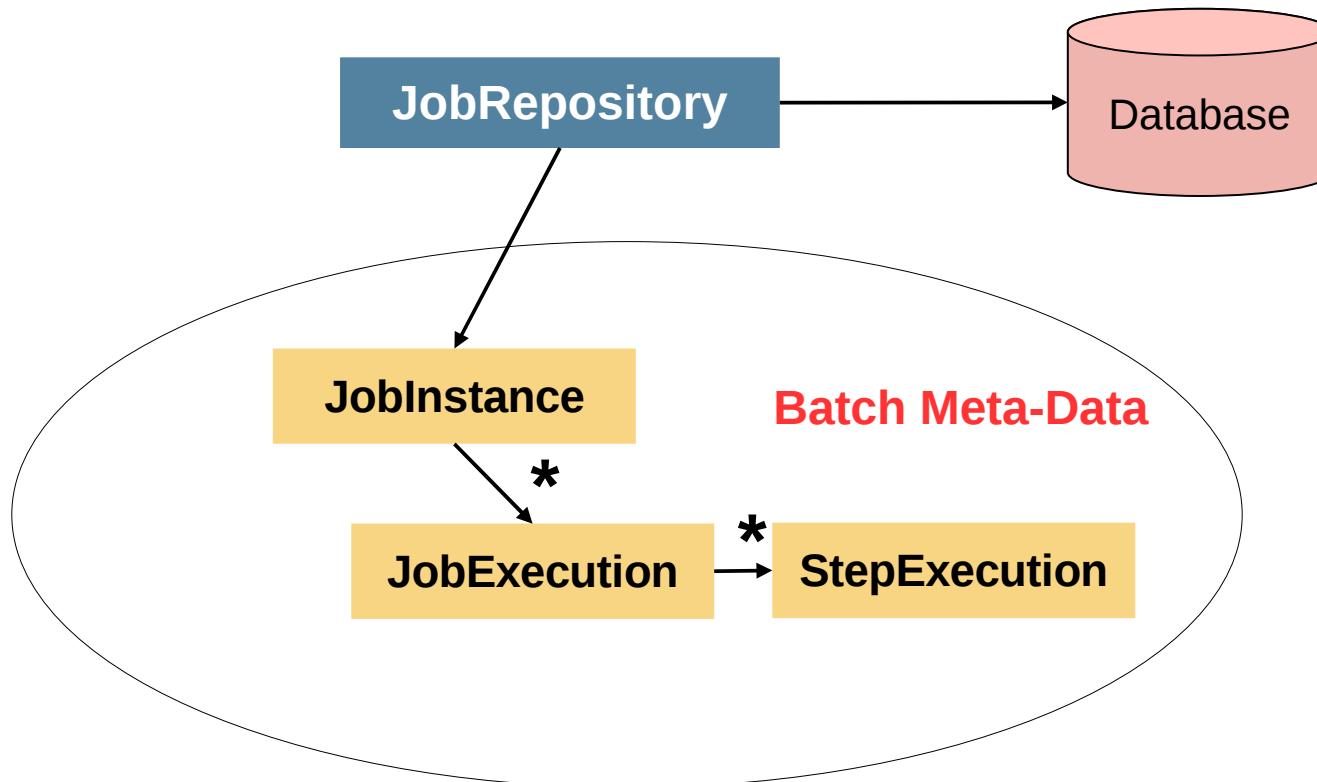
The screenshot shows the HSQL Database Manager interface. The left pane displays a tree view of database objects under the schema 'PUBLIC'. The right pane shows the results of a SQL query: 'select * from BATCH_STEP_EXECUTION'. The results are presented in a table with three columns: START_TIME, END_TIME, and STATUS. One row is visible, showing the values: 2008-08-01 11:26:18.679, 2008-08-01 11:26:19.867, and COMPLETED respectively.

START_TIME	END_TIME	STATUS
2008-08-01 11:26:18.679	2008-08-01 11:26:19.867	COMPLETED

Extended Picture: Job and Step



JobRepository and Batch Meta-Data



Topics in this Session



- Batch and offline processing
- Spring Batch high-level overview
- Readers, Writers & Processors
- Job parameters and job identity
- **Quick start using Spring Batch**
- JDBC Item Readers

Spring Batch Quickstart



- Configure the readers, processors and writers
- Configure a Job with Step(s)
- Configure a JobLauncher
- Load and run the job
 - Write a wrapper for the JobLauncher
 - Use off-the-shelf wrapper (CommandLineJobRunner, ...)
- Result is in return value from the JobLauncher

Configure JobLauncher



```
<beans>

<bean id="jobLauncher" class="org.springframework.batch.core.launcher.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository" />
</bean>

<batch:job-repository id="jobRepository" />

</beans>
```

Configure Reader and Writer



```
<beans>

    <bean id="itemReader" class="org.sfw..FlatFileItemReader">
        <property name="fieldSetMapper" ref="customMapper" />
        <property name="resource" value="file://home/jobs/data/input.csv"/>
        ...
    </bean>

    <bean id="itemWriter" class="org.sfw..FlatFileItemWriter">
        <property name="fieldSetCreator" ref="customCreator" />
        ...
    </bean>

</beans>
```

Configure a Job with Step(s)



- Using the Spring Batch namespace support :

```
<beans:beans>

    <job id="endOfDayJob" >
        <step id="copyStep">
            <tasklet>
                <chunk reader="itemReader" writer="itemWriter"
                      commit-interval="${chunk.size}">
                </tasklet>
            </step>
            ...
        </job>

    </beans:beans>
```

Launch the Job

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

// Look up the job launcher and job
jobLauncher = context.getBean(JobLauncher.class);
job = context.getBean("endOfDayJob", Job.class);

// Launch the job
JobExecution execution =
    jobLauncher.run(job, new JobParameters());
```

Use this to see the exit status



CommandLineJobRunner



- Framework-provided command line utility
- 2 mandatory parameters: XML config location and job id

```
# Command line prompt:
```

```
$ java -classpath ... org.sfw...CommandLineJobRunner  
application-config.xml endOfDayJob
```

(All on the same line)

Adding Parameters

- With JobLauncher

```
JobParametersBuilder builder = new JobParametersBuilder();
builder.addString("file.locator", "2008-05-05");
...
JobExecution execution =
    jobLauncher.run(job, builder.toJobParameters());
```

Adding Parameters

- With CommandLineJobRunner

```
$ java -classpath ... org.sfw...CommandLineJobRunner  
application-config.xml endOfDayJob  
run.id=1003AHQ7  
schedule.date(date)=2008/05/05
```

key(*type*)=value pairs at end of line
where *type* = string | date | long

Topics in this Session



- Batch and offline processing
- Spring Batch high-level overview
- Readers, Writers & Processors
- Job parameters and job identity
- Quick start using Spring Batch
- **JDBC Item Readers**

- Useful when DB is primary input/output or to stage file-based I/O in temporary DB table
 - Flat file I/O in batch processes can cause difficulties
 - Read / discard much data during restart for files not formatted with fixed record lengths
 - Difficult to synchronize on output (non-transactional)
- Spring Batch provides DB ItemReaders and -Writers to help with this

We'll just talk about readers here

- Cursor based
 - JdbcCursorItemReader
 - HibernateCursorItemReader
 - StoredProcedureItemReader
- Paging
 - JdbcPagingItemReader
 - JpaPagingItemReader
 - IbatisPagingItemReader

*We'll just talk about the JDBC Implementations here;
refer to the Spring Batch Documentation for more information*

Cursor Based Item Readers



- JdbcTemplate + RowMapper callback load all data into memory
 - Not practical for large batch data sets
- Cursor-based Item Reader maps row at a time, using same RowMapper interface
- Reader's read() method returns mapped domain object
 - Data is 'streamed' - written and discarded according to the step's commit-interval
- May not be memory efficient – check JDBC driver for proper server-side cursor support

JdbcCursorItemReader



```
<bean id="itemReader" class="org.spr...JdbcCursorItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql"
    value="select ID, NAME, CREDIT from CUSTOMER"/>
  <property name="rowMapper">
    <bean class="com.acme.domain.CustomerCreditRowMapper"/>
  </property>
</bean>
```

Additional properties, including:
fetchSize, *maxRows*, *driverSupportsAbsolute*, ...
Refer to documentation

Paging Item Readers



- Alternative technique to cursor – uses distinct queries for each set (or page) of data
 - Start and end included in query
 - Syntax depends on DBMS
- JdbcPagingItemReader delegates to a PagingQueryProvider implementation
 - SqlPagingQueryProviderFactoryBean detects DBMS to create the appropriate provider
- More control over memory utilization than cursor based readers

JdbcPagingItemReader



- You must provide
 - SELECT clause
 - FROM clause
 - Optional WHERE clause
 - Sort Key
- The sort key is used as the start/end
 - Reader keeps track of the last read sort key and adds it to the where clause for the next query
 - **MUST BE UNIQUE!**

JdbcPagingItemReader



```
<bean id="itemReader" class="org.spr...JdbcPagingItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="queryProvider">
        <bean class="org.spr...SqlPagingQueryProviderFactoryBean">
            <property name="dataSource" ref="dataSource"/>
            <property name="selectClause" value="select id, name, credit"/>
            <property name="fromClause" value="from customer"/>
            <property name="whereClause" value="where status='NEW'"/>
            <property name="sortKey" value="id"/>
        </bean>
    </property>
    <property name="pageSize" value="1000"/>
    <property name="rowMapper">
        <bean class="com.example.CustomerMapper"/>
    </property>
</bean>
```



Lab



Spring Batch Restart and Recovery

State & execution management for
restarting and recovering from failures
with Spring Batch

Topics in this Session



- **ExecutionContext**
- Stateful ItemReaders/Writers
- Reading Flat Files
- Sharing State Between Steps
- Intro to Skip, Retry, Repeat, Restart
- Other Listeners
- Business Logic Delegation

- We need to know where a failure occurred to restart a batch process
- Job Repository meta data is used to determine the step at which the failure occurred
- Application Code (in reader/writer) needs to maintain state within a step (e.g. current chunk)
- Spring Batch can supply that data during restart

ExecutionContext



- Key/Value pairs persisted by the framework
- During restart, provides initial state
- Job ExecutionContext
 - Committed at end of step
 - Can also be used to pass state between steps
- Step ExecutionContext
 - Committed at end of each chunk

Topics in this Session

- ExecutionContext
- **Stateful ItemReaders/Writers**
- Reading Flat Files
- Sharing State Between Steps
- Intro to Skip, Retry, Repeat, Restart
- Other Listeners
- Business Logic Delegation

- StepExecutionListener
 - Get a reference to the execution context to store state within the reader/writer
- @BeforeStep annotation
 - Same as StepExecutionListener
- ItemStream
 - Provides access to the context before the first read and just before each commit
- Care must be taken if step is multi-threaded

StepExecutionListener

```
public class DiningReader implements ItemReader<Dining>, StepExecutionListener {  
    private ExecutionContext executionContext;  
    private int filePosition;  
    public Dining read() throws Exception {  
        ...  
        filePosition++;  
        executionContext.put("position", filePosition);  
        return dining;  
    }  
  
    public void beforeStep(StepExecution stepExecution) {  
        this.executionContext = stepExecution.getExecutionContext();  
        filePosition = this.executionContext.getInt("position", 0);  
    }  
    ...  
}
```

Initialize File Position from Last Run

Also need implementation for afterStep()

Assumes single-threaded step

StepExecutionListener



- Must also implement afterStep()

```
public class DiningReader implements ItemReader<Dining>, StepExecutionListener {  
    ...  
  
    public ExitStatus afterStep(StepExecution stepExecution) {  
        return null;  
    }  
}
```

- Can alter exit status
 - will be merged using ExitStatus.and()
 - return null to preserve the original step exit status

Annotations

```
public class DiningReader implements ItemReader<Dining> {  
    private ExecutionContext executionContext;  
    private int filePosition;  
  
    public Dining read() throws Exception {  
        ...  
        filePosition++;  
        executionContext.put("position", filePosition);  
        return dining;  
    }  
  
    @BeforeStep  
    public void saveContext(StepExecution stepExecution) {  
        this.executionContext = stepExecution.getExecutionContext();  
        filePosition = this.executionContext.getInt("position", 0);  
    }  
}
```

Also @AfterStep, but not required

ItemStream

- If ItemReader implements ItemStream interface
 - open() - called before any read() calls
 - update() - called at the end of each chunk, before commit
 - close() - called at the end of the step
- Implemented by many built-in ItemReaders and -Writers

ItemStream



```
public class DiningReader implements ItemReader<Dining>, ItemStream {  
    private int filePosition;  
    public Dining read() throws Exception {  
        ...  
        filePosition++;  
        return dining;  
    }  
    public void open(ExecutionContext executionContext) throws  
        ItemStreamException {  
        filePosition = executionContext.getInt("position", 0);  
    }  
    public void update(ExecutionContext executionContext) throws  
        ItemStreamException {  
        executionContext.putInt("position", filePosition);  
    }  
    public void close() throws ItemStreamException { }  
}
```

Transactional ItemReaders and -Processors



- Items read and processed are cached by default
 - Prevents losing data in case of error
- Not needed for transactional reading/processing
 - e.g. when reading from JMS Queue
 - Rollback will restore original state automatically
- Configure this on the chunk

```
<step id="step1">
  <tasklet>
    <chunk reader="itemReader" writer="itemWriter" commit-interval="10"
          reader-transactional-queue="true" processor-transactional="true">
      </chunk>
    </tasklet>
  </step>
```

Topics in this Session

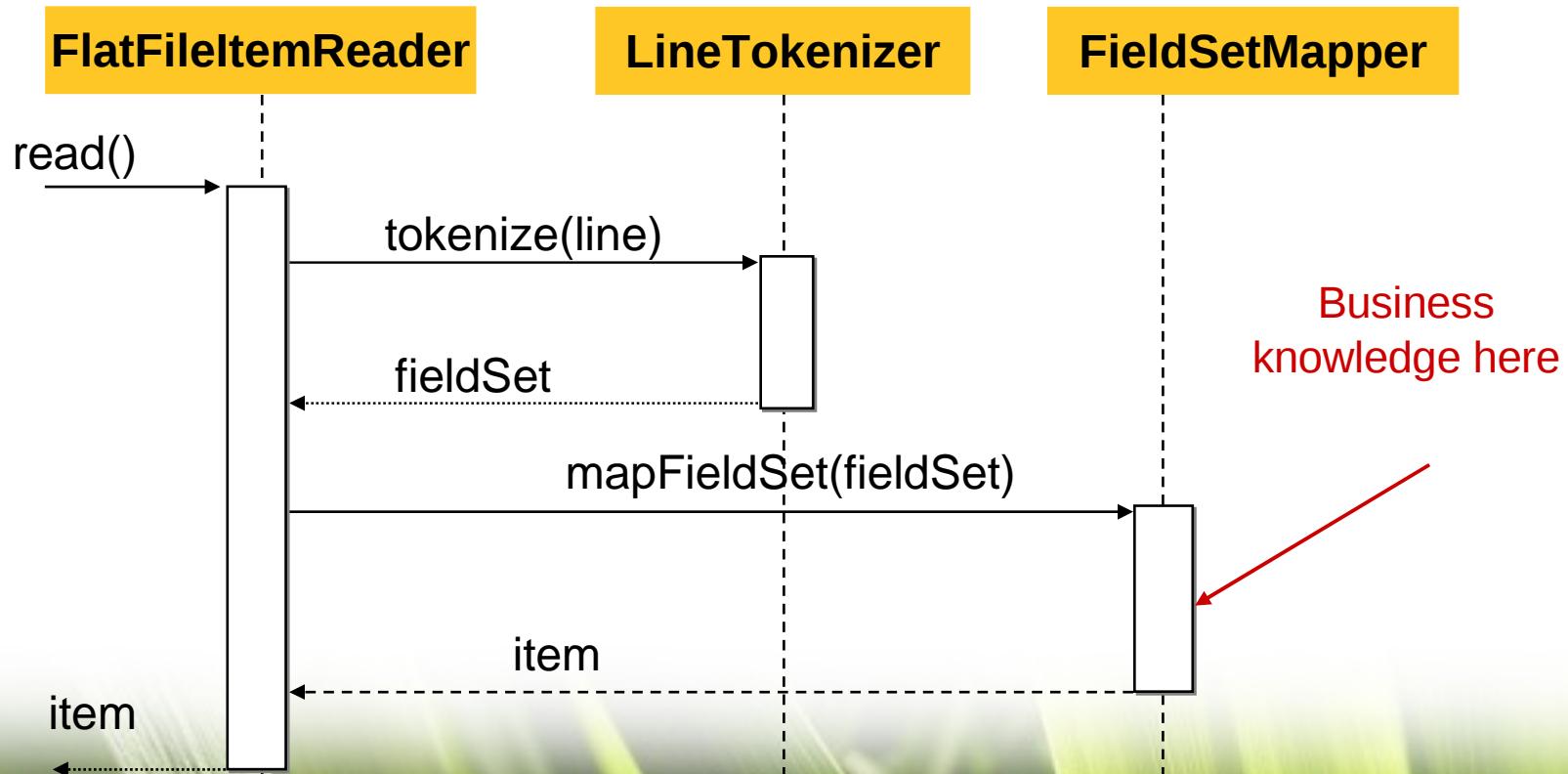


- ExecutionContext
- Stateful ItemReaders/Writers
- **Reading Flat Files**
- Sharing State Between Steps
- Intro to Skip, Retry, Repeat, Restart
- Other Listeners
- Business Logic Delegation

FlatFileItemReader



- Stateful, remembers nr. of lines read
- Delegates to LineTokenizer and FieldSetMapper for parsing when used with DefaultLineMapper



FieldSet



- Intermediate between line (record) in file and domain object
- Immutable wrapper for array of Strings
- Access fields by name or index, and by type

```
FieldSet fs = ... ;
```

```
Date date = fs.readDate(0, "dd/MM/yyyy"); // first field  
Long number = fs.readLong(1);           // second field  
String value = fs.readString("city");   // named field  
String[] values = fs.getValues();        // all fields
```

FieldSetMapper

- Maps a FieldSet to a domain Object

```
public class DiningFieldSetMapper  
    implements FieldSetMapper<Dining> {  
  
    public Dining mapFieldSet(FieldSet fs) {  
        // Construct and return a Dining object  
    }  
  
}
```

FlatFileItemReader Configuration



```
<bean id="diningRequestsReader" class="org.sfw.batch.item.file.FlatFileItemReader">
<property name="resource" value="file:batch-input.csv"/> File to read
<property name="lineMapper">
<bean class="org.springframework.batch.item.mapping.DefaultLineMapper">
<property name="lineTokenizer">
<bean class="org.sfw.batch.item.transform.DelimitedLineTokenizer">
<property name="names"> for use in FieldSet
<list>
<value>creditCardNumber</value>
<value>merchantNumber</value>
<value>amount</value>
<value>date</value>
</list>
</property>
</bean>
</property>
<property name="fieldSetMapper">
<bean class="rewards.batch.DiningFieldSetMapper"/>
</property>
</bean>
</property>
</bean>
```

default impl

For use with CSV-like files

FieldSetMappers in Spring Batch



- Spring Batch provides a couple of mappers for common use cases:
- PassthroughFieldSetMapper
 - just returns the FieldSet as-is
 - simple use cases with no domain model, e.g. copy straight from file to database
- BeanWrapperFieldSetMapper
 - uses field names to bind to a prototype object
 - Binds using Spring TypeConverter (a.k.a. PropertyEditor) support

Topics in this Session



- ExecutionContext
- Stateful ItemReaders/Writers
- Reading Flat Files
- **Sharing State Between Steps**
- Intro to Skip, Retry, Repeat, Restart
- Other Listeners
- Business Logic Delegation

Sharing State Between Steps



- Data in step execution context can be “promoted” to job execution context
- Downstream steps can then retrieve those data
- Prevents state-storing code from having to interact with Job ExecutionContext directly
- ExecutionContextPromotionListener does this for specified key names

Sharing State Between Steps



```
<job id="job1">
  <step id="step1" next="step2">
    <tasklet>
      <chunk reader="reader" writer="savingWriter" commit-interval="10"/>
      <listeners>
        <listener ref="promotionListener"/>
      </listeners>
    </tasklet>
  </step>
  <step id="step2">
    ...
  </step>
</job>

<beans:bean id="promotionListener"
  class="org.spr....ExecutionContextPromotionListener">
  <beans:property name="keys" value="someKey"/>
</beans:bean>
```

Sharing State Between Steps



```
// Used in step 2:
```

```
public class RetrievingItemWriter implements ItemWriter<Object> {  
    private Object someObject;  
    public void write(List<? extends Object> items) throws Exception {  
        // ...  
    }  
  
    @BeforeStep  
    public void retrieveInterstepData(StepExecution stepExecution) {  
        JobExecution jobExecution = stepExecution.getJobExecution();  
        ExecutionContext jobContext = jobExecution.getExecutionContext();  
        this.someObject = jobContext.get("someKey");  
    }  
}
```

Step Scope

- Spring Batch defines custom 'step' scope
- Useful for just-in-time initialization
 - To pass parameters from JobParameters or Job's ExecutionContext to step using SpEL expressions
 - jobParameters variable available automatically

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

Topics in this Session

- ExecutionContext
- Stateful ItemReaders/Writers
- Reading Flat Files
- Sharing State Between Steps
- **Intro to Skip, Retry, Repeat, Restart**
- Other Listeners
- Business Logic Delegation

Common Batch Idioms



- Batch jobs typically process large amounts of homogeneous input
- Makes iteration a common concern: **Repeat**
- Transient errors during processing may require a **Retry** of an input item
- Some input may not be valid, may want to **Skip** it without failing
- Some errors should fail the job execution, allowing to fix the problem and **Restart** the job instance where it left off

Spring Batch Support



- Spring Batch supports these common concerns
- Abstracts them in the framework
 - Job business logic doesn't need to care about details
- Allows for simple configuration with pluggable strategies

Repeat

- Most batch steps read items to process from some input source
- Framework can perform iteration over the input
 - Done using RepeatTemplate with RepeatCallback
 - Chunked step invokes ItemReader.read() in callback impl
 - Means no direct interaction with template is needed
- When there's no more input the step is complete
 - For chunked steps, ItemReader.read() simply returns null

Retry

- A processing error could be *transient*
 - intermittent failure caused by external circumstances
- Simply retrying the operation might succeed
 - No need to fail the entire execution immediately
- Can mark certain exceptions as retryable
 - With max. # of retries per step execution

Retry Configuration



```
<step id="step1">
  <tasklet>
    <chunk reader="itemReader" writer="itemWriter" commit-interval="20"
          retry-limit="3">
      <retryable-exception-classes>
        <include class="org.sfw.dao.DeadlockLoserDataAccessException"/>
      </retryable-exception-classes>
    </chunk>
  </tasklet>
</step>
```

- Can also use excludes
 - For example, to include java.lang.Exception and then exclude specific exceptions that are not retryable

Skip

- Not all processing errors should cause a failure
 - Input often contains small percentage of invalid items
 - Just log the error and continue
- Can mark certain exceptions as skippable
 - With max. # of items to skip per step execution

Skip Configuration

```
<step id="step1">
  <tasklet>
    <chunk reader="flatFileItemReader" writer="itemWriter"
           commit-interval="10" skip-limit="10">
      <skippable-exception-classes>
        <include class="org.sfw.batch.item.file.FlatFileParseException"/>
      </skippable-exception-classes>
    </chunk>
  </tasklet>
</step>
```

- Can also use excludes, like with retry
 - For example, to include java.lang.Exception and then exclude specific exceptions that should cause failure

- Jobs defined via batch namespace considered restartable after failure by default
 - Disable with `restartable="false"`
- Requires execution context state to be persisted
 - For both job & step executions
 - Or executions would always start from the beginning
 - Various timestamps, status, etc. also persisted
 - See section on stateful readers/writers earlier
- Spring Batch checks for existing job execution
 - If found, it's a restart of an existing job instance
 - If not, it's a regular first execution

Restarting a Job Instance



- Step execution context persisted at commit
 - incl. # of items read, written, skipped
 - incl. # of TXs committed and rolled back
- On restart ItemReader/-Writer can access that state
- Uses it do determine if and where to resume
 - Built in for many supplied ItemReaders and -Writers
 - Implement ItemStream for your own implementations

Restart Configuration



- Allow completed steps to run again on restart
 - not allowed by default
- Limit the number of step restarts allowed
 - default is unlimited

```
<step id="gameLoad" next="playerSummarization">
    <tasklet allow-start-if-complete="true">
        <chunk reader="gameFileItemReader" writer="gameWriter"
               commit-interval="10" />
    </tasklet>
</step>
<step id="playerSummarization">
    <tasklet start-limit="3">
        <chunk reader="playerSummarizationSource"
               writer="summaryWriter" commit-interval="10" />
    </tasklet>
</step>
```

Topics in this Session

- ExecutionContext
- Stateful ItemReaders/Writers
- Reading Flat Files
- Sharing State Between Steps
- Intro to Skip, Retry, Repeat, Restart
- **Other Listeners**
- Business Logic Delegation

- Register to get callbacks during execution
 - For logging/auditing, state & error handling, ...
 - JobExecutionListener
 - StepListener (marker interface)
 - StepExecution-, Chunk-, Item(Read|Process|Write)- & SkipListener
 - Annotation-based approach supported as alternative
- Implemented by ItemReader/-Processor/-Writer or in dedicated class
 - Explicit registration often not needed without dedicated class
- SkipListener covered here, check reference for info on others

SkipListener



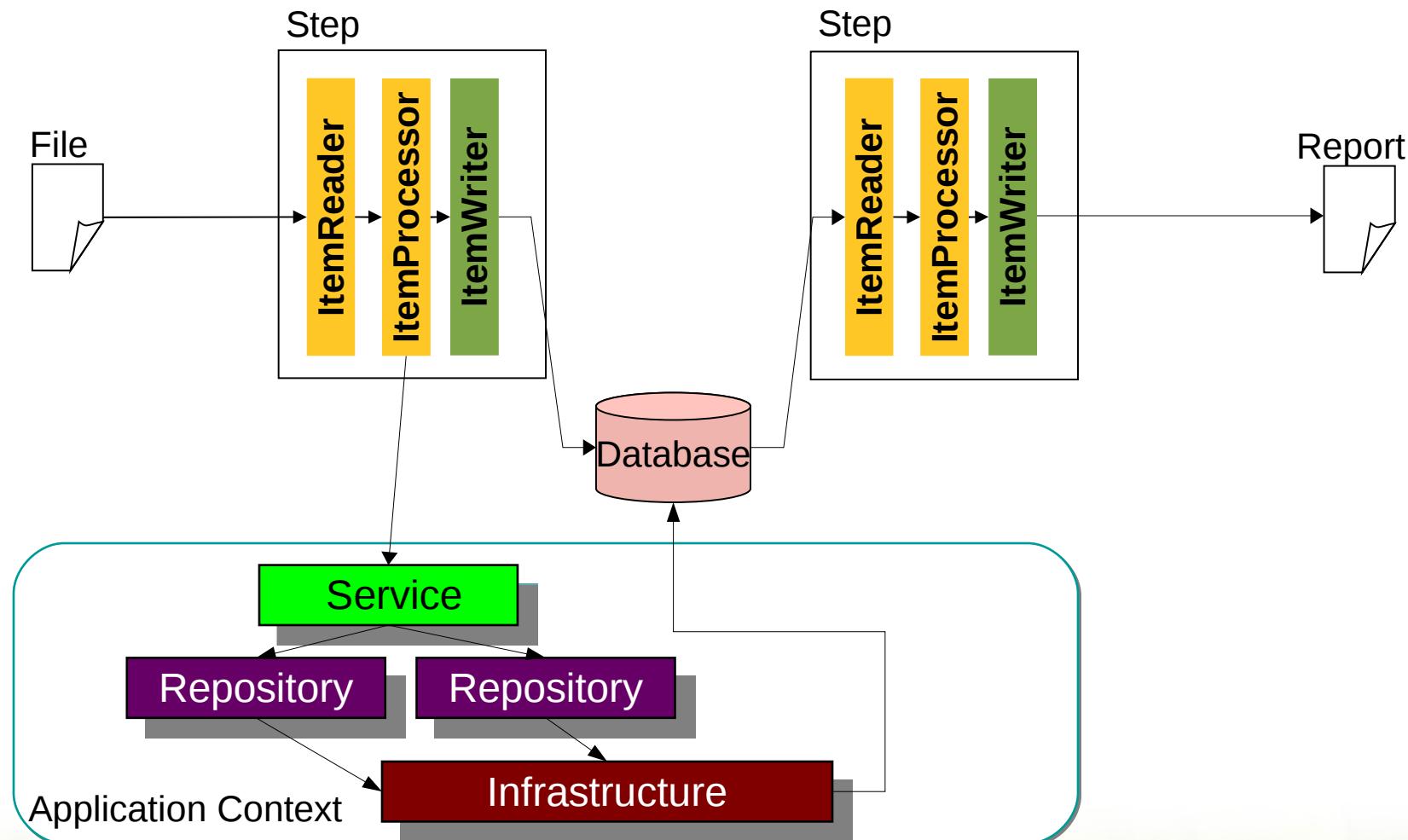
```
public interface SkipListener<T,S> extends StepListener {  
    void onSkipInRead(Throwable t);  
  
    void onSkipInProcess(T item, Throwable t);  
  
    void onSkipInWrite(S item, Throwable t);  
}
```

- Skipped item is available unless skip occurred on read
- Appropriate skip method only called once per item
 - depending on when the error occurred
- Methods are NOT called when actual error occurred – called immediately before tx commit, after all chunk processing complete
- @OnSkipInRead, @OnSkipInWrite, @OnSkipInProcess annotations also available

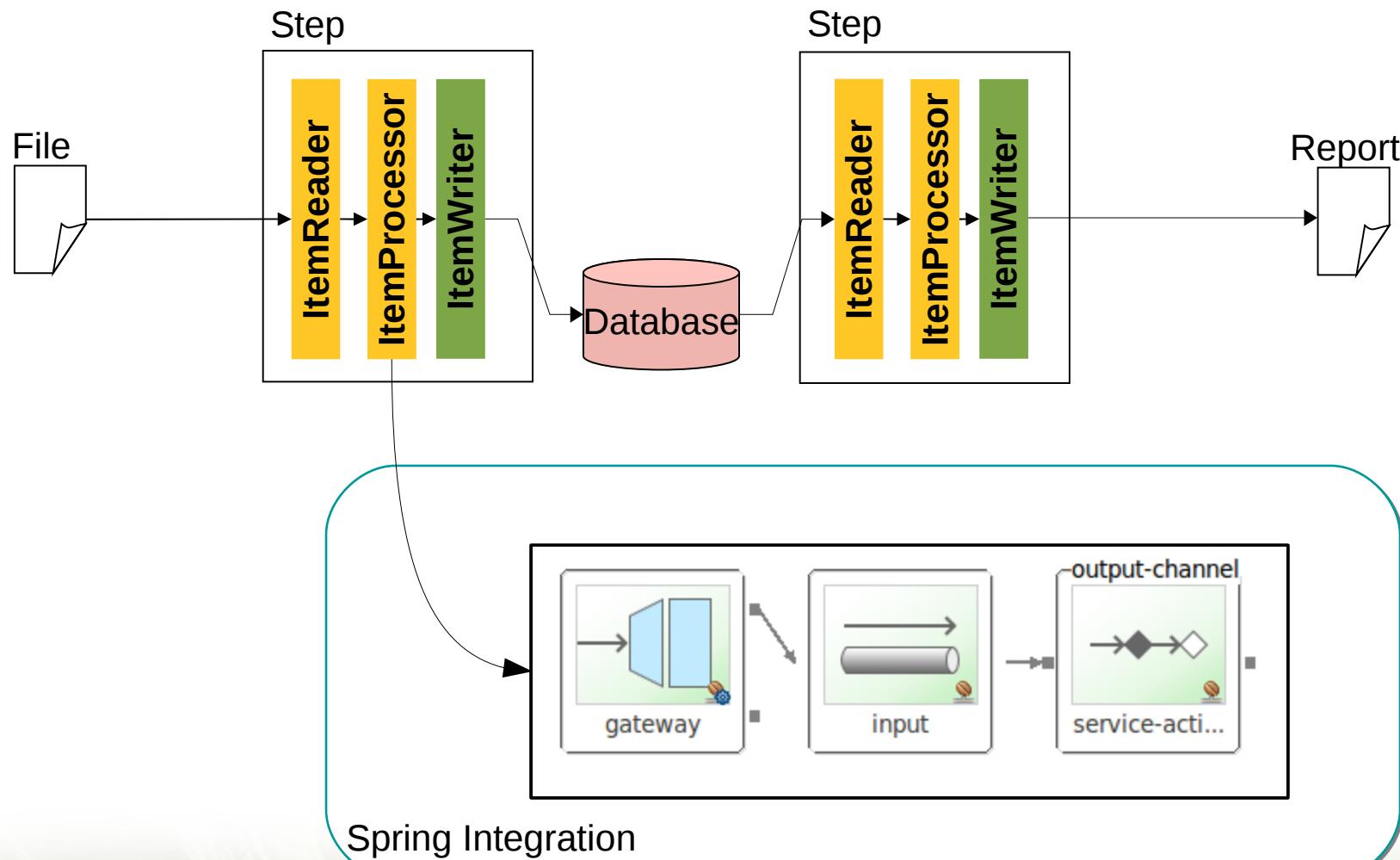
Topics in this Session

- ExecutionContext
- Stateful ItemReaders/Writers
- Reading Flat Files
- Sharing State Between Steps
- Intro to Skip, Retry, Repeat, Restart
- Other Listeners
- **Business Logic Delegation**

Business Logic Delegation – Spring Application



Business Logic Delegation – Spring Integration





Lab

Restarting and Recovering a Failed Batch Job



Spring Batch Admin and Parallel processing

A guide to Spring Batch Admin
And
Parallel processing in Batch jobs

Topics in this Session

- **Batch Admin**
- Scaling and Parallel Processing

Spring Batch Admin



- Sub project of Spring Batch
- Provides Web UI and ReSTFul interface to manage batch processes

<http://static.springsource.org/spring-batch-admin/index.html>

- Manager, Resources, Sample WAR
 - Deployed with batch job(s) as single app to be able to control & monitor jobs
 - Or monitors external jobs only via shared database

Home Page

A screenshot of a web browser window displaying the Spring Batch Admin interface. The URL in the address bar is "http://localhost:8080/batch-2-solution/".

The browser toolbar includes File, Edit, View, History, Bookmarks, Tools, and Help. The address bar shows the URL. The title bar says "Most Visited" and "Spring Batch Admin". The main content area has a green header with the "Spring source" logo. Below the header is a navigation menu with Home, Jobs, Executions, and Files tabs. To the right of the menu are links for SpringSource and Spring Batch. The main content is a table listing API resources and their methods:

Resource	Method	Description
/configuration	GET	
/files	GET	
/files/**	GET	
/files/{path}	POST	
/home	GET	
/job-configuration-files	GET	
/job-configuration-multipart	GET	
/job-requests	GET	
/job-restarts	GET	
/jobs	GET	
/jobs/executions	GET	
/jobs/executions	DELETE	
/jobs/executions/{jobExecutionId}	GET	
/jobs/executions/{jobExecutionId}	DELETE	
/jobs/executions/{jobExecutionId}/steps	GET	
/jobs/executions/{jobExecutionId}/steps/{stepExecutionId}	GET	
/jobs/executions/{jobExecutionId}/steps/{stepExecutionId}/progress	GET	
/jobs/{jobName}	GET	
/jobs/{jobName}	POST	
/jobs/{jobName}/executions	GET	
/jobs/{jobName}/{jobInstanceId}/executions	GET	
/jobs/{jobName}/{jobInstanceId}/executions	POST	
/steps/step1	GET	

At the bottom left is a "Done" button, and at the bottom right is a "S3Fox" logo.

Registered Jobs



A screenshot of a web browser displaying the Spring Batch Admin interface. The URL in the address bar is <http://localhost:8080/batch-2-solution/batch/job>. The page title is "Spring Batch Admin". The main content area shows a table of registered jobs:

Name	Description	Execution Count	Launchable	Incrementable
infinite	No description	0	true	true
job1	No description	0	true	true
job2	No description	0	true	false

Below the table, the text "Rows: 1-3 of 3 Page Size: 20" is visible. At the bottom of the page, there is a copyright notice "© Copyright 2009-2010 SpringSource. All Rights Reserved.", a "Contact SpringSource" link, and a "Done" button. A small "S3Fox" watermark is in the bottom right corner.

Launch



The screenshot shows a web browser window with the URL <http://localhost:8080/batch-2-solution/batch/job>. The page is titled "Spring Batch Admin". A blue oval highlights the "Job name=job1: Launch" section, which includes a "Launch" button and a "Job Parameters" input field containing "fail=false". The input field is described as "(Incrementable) (key=value pairs)". Below this, a note states: "If the parameters are marked as "Not incrementable" then the launch button launches an instance of the job with the parameters shown. You can always add new parameters if you want to." A message below says: "There are no job instances for this job." The footer includes copyright information, a "Contact SpringSource" link, and the "S3Fox" logo.

File Edit View History Bookmarks Tools Help

← → ⌘ × ⌘ http://localhost:8080/batch-2-solution/batch/job ⌘ Google

Most Visited

Spring Batch Admin: Job Summ... +

Spring Batch Admin

spring source

Home Jobs Executions Files SpringSource Spring Batch

Job name=job1: **Launch**

Job Parameters fail=false (Incrementable)
(key=value pairs):

If the parameters are marked as "Not incrementable" then the launch button launches an instance of the job with the parameters shown. You can always add new parameters if you want to.

There are no job instances for this job.

© Copyright 2009-2010 SpringSource. All Rights Reserved.

Contact SpringSource

Done

S3Fox

Launched



The screenshot shows a web browser window with the URL <http://localhost:8080/batch-2-solution/batch/job>. The page title is "Spring Batch Admin". The main content area displays job parameters and execution details for a job named "job1". A blue oval highlights the table showing "Job Instances for Job (job1)".

Job name=job1:

Job Parameters (Incrementable)
(key=value pairs):

If the parameters are marked as "Not incrementable" then the launch button launches an instance of the job with the parameters shown. You can always add new parameters if you want to.

Job Instances for Job (job1)

ID	JobExecution Count	Last JobExecution	Parameters
0	executions 1	STARTED	{fail=false, run.count=0}

Rows: 1-1 of 1 Page Size: 20

The table above shows instances of this job with an indication of the status of the last execution. If you want to look at all executions for [see here](#).

Done

Completed



File Edit View History Bookmarks Tools Help

← → ⌘ × ⌂ http://localhost:8080/batch-2-solution/batch/ Google

Most Visited ▾

Spring Batch Admin: Job Execut... +

Spring Batch Admin

SpringSource Spring Batch

Recent and Current Job Executions for Job = job1, instanceId = 0

Stop All

ID	Instance	Name	Date	Start	Duration	Status	ExitCode
0	0	job1	2010-09-01	14:05:58	00:00:00	COMPLETED	COMPLETED

© Copyright 2009-2010 SpringSource. All Rights Reserved. Contact SpringSource

Done

S3Fox

A screenshot of a web browser displaying the Spring Batch Admin interface. The title bar shows the URL as http://localhost:8080/batch-2-solution/batch/. The main content area is titled 'Recent and Current Job Executions for Job = job1, instanceId = 0'. A table lists one job execution with ID 0, instance 0, name job1, date 2010-09-01, start time 14:05:58, duration 00:00:00, status COMPLETED, and exit code COMPLETED. The 'Status' column for this row is circled in blue. At the bottom of the page, there is copyright information for SpringSource and a 'Contact SpringSource' link.

Job Execution Details



The screenshot shows the Spring Batch Admin interface with a green header bar. On the left, it says "Spring Batch Admin". On the right, there is a "Spring source" logo. Below the header is a navigation bar with tabs: Home, Jobs, Executions, Files, SpringSource, and Spring Batch. The main content area is titled "Details for Job Execution". It features a "Stop" button and a table of job properties:

Property	Value
ID	0
Job Name	<u>job1</u>
Job Instance	<u>0</u>
Job Parameters	run.count(long)=0,fail=false
Start Date	2010-09-01
Start Time	14:05:58
Duration	00:00:00
Status	COMPLETED
Exit Code	COMPLETED
Step Executions Count	1
Step Executions	[j1.s1]

Step Execution



File Edit View History Bookmarks Tools Help

← → ⌂ ⌂ ⌂ ⌂ ⌂ http://localhost:8080/batch-2-solution/batch/jobs/exec ⌂ Google

Most Visited ▾

Spring Batch Admin: Step Exec...

Spring Batch Admin

spring source

Home Jobs Executions Files SpringSource Spring Batch

Step Executions for Job = job1, JobExecution = 0

ID	Job Execution	Name	Date	Start	Duration	Status	Reads	Writes	Skips	ExitCode	
0	detail	0	j1.s1	2010-09-01	18:05:58	00:00:00	COMPLETED	5	5	0	COMPLETED

© Copyright 2009-2010 SpringSource. All Rights Reserved. Contact SpringSource

Done ↴ S3Fox

Step Execution Details



Spring Batch Admin

Home Jobs Executions Files

Step Execution Progress

This execution is estimated to be 100% complete after 164 ms based on end time (already finished).

History of Step Execution for Step=j1.s1

Summary after total of 1 executions:

Property	Min	Max	Mean	Sigma
Duration per Read	32	32	32	0
Duration	164	164	164	0
Commits	6	6	6	0
Rollbacks	0	0	0	0
Reads	5	5	5	0
Writes	5	5	5	0
Filters	0	0	0	0
Read Skips	0	0	0	0
Write Skips	0	0	0	0
Process Skips	0	0	0	0

Details for Step Execution

Property	Value
ID	0
Job Execution	0
Job Name	job1
Step Name	j1.s1
Start Date	2010-09-01
Start Time	18:05:58
Duration	00:00:00
Status	COMPLETED
Reads	5
Writes	5
Filters	0
Read Skips	0
Write Skips	0
Process Skips	0
Commits	6
Rollbacks	0
Exit Code	COMPLETED
Exit Message	

Done

spring source

Details for Step Execution

Property	Value
ID	0
Job Execution	0
Job Name	job1
Step Name	j1.s1
Start Date	2010-09-01
Start Time	18:05:58
Duration	00:00:00
Status	COMPLETED
Reads	5
Writes	5
Filters	0
Read Skips	0
Write Skips	0
Process Skips	0
Commits	6
Rollbacks	0
Exit Code	COMPLETED
Exit Message	

Done

Failures



Spring Batch Admin Step Details	
Step Name	j1.s1
Start Date	2010-09-01
Start Time	18:19:16
Duration	00:00:00
Status	FAILED
Reads	1
Writes	0
Filters	0
Read Skips	0
Write Skips	0
Process Skips	0
Commits	0
Rollbacks	1
Exit Code	FAILED
Exit Message	<pre>java.lang.RuntimeException: Planned failure at org.springframework.batch.admin.sample.ExampleItemWriter.write(ExampleItemWriter.java:42) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39) at</pre>

Restart or Abandon



The screenshot shows the 'Spring Batch Admin' interface. At the top, there's a navigation bar with tabs: Home, Jobs, Executions, Files, SpringSource, and Spring Batch. Below the navigation bar, the main content area has a title 'Details for Job Execution'. On the left, there are two buttons: 'Abandon' and 'Restart', which are circled in blue. To the right of these buttons is a table with the following data:

Property	Value
ID	1
Job Name	job1
Job Instance	1
Job Parameters	run.count(long)=1,fail=true
Start Date	2010-09-01
Start Time	14:19:16
Duration	00:00:00
Status	FAILED
Exit Code	FAILED

At the bottom left of the table, there's a 'Done' button. A cursor arrow is visible on the right side of the table. In the bottom right corner of the interface, there's a small 'S3Fox' logo.

Stop a Running Job



The screenshot shows the Spring Batch Admin interface. At the top, there's a navigation bar with links for Home, Jobs, Executions, Files, SpringSource, and Spring Batch. Below the navigation bar, the main content area has a title "Details for Job Execution". A "Stop" button is located above a table. A blue circle highlights the "Stop" button. The table below lists various properties of the job execution:

Property	Value
ID	2
Job Name	<u>job1</u>
Job Instance	<u>2</u>
Job Parameters	run.count(long)=2,fail=false
Start Date	2010-09-01
Start Time	14:33:24
Duration	00:00:19
Status	STARTED
Exit Code	UNKNOWN
Step Executions Count	1
<u>Step Executions</u>	[j1.s1]

Stopped



Spring Batch Admin

Home Jobs Executions Files

Job name=job1: **Launch**

Job Parameters (key=value pairs): `run.count(long)=2,fail=fal`

If the parameters are marked as "Not incrementable" the shown. You can always add new parameters if you want.

Job Instances for Job (job1)

ID	JobExecution Count	Last JobExe
2	executions	1 STOPPED
1	executions	1 FAILED
0	executions	1 COMPLETED

Rows: 1-3 of 3 Page Size: 20

The table above shows instances of this job with an indicator. [see here.](#)

Done

Details for Step Execution

Property	Value
ID	2
Job Execution	2
Job Name	job1
Step Name	j1.s1
Start Date	2010-09-01
Start Time	18:33:24
Duration	00:01:35
Status	STOPPED
Reads	1
Writes	1
Filters	0
Read Skips	0
Write Skips	0
Process Skips	0
Commits	1
Pollbacks	0
Exit Code	STOPPED
Exit Message	<code>org.springframework.batch.core.JobInterruptedException</code>

Done

Restart or Abandon



Spring Batch Admin

SpringSource Spring Batch

Details for Job Execution

Abandon

Restart

Property	Value
ID	2
Job Name	<u>job1</u>
Job Instance	2
Job Parameters	run.count(long)=2,fail=false
Start Date	2010-09-01
Start Time	14:33:24
Duration	00:01:35
Status	STOPPED
Exit Code	STOPPED
Step Executions Count	1
Step Executions	12-s11

Done

S3Fox

The screenshot shows the Spring Batch Admin interface. At the top, there's a navigation bar with tabs: Home, Jobs, Executions, Files, SpringSource, and Spring Batch. Below the navigation bar, the title 'Details for Job Execution' is displayed. On the left, there are two buttons: 'Abandon' and 'Restart', both enclosed in a blue oval. To the right of these buttons is a table with various job parameters. One specific row in the table, 'Status', is also highlighted with a blue oval. The table data includes: ID (2), Job Name (job1), Job Instance (2), Job Parameters (run.count(long)=2,fail=false), Start Date (2010-09-01), Start Time (14:33:24), Duration (00:01:35), Status (STOPPED), Exit Code (STOPPED), Step Executions Count (1), and Step Executions (12-s11). At the bottom left, there's a 'Done' button, and at the bottom right, there's a small 'S3Fox' logo.

Restarting a Stopped/Failed Job



Spring Batch Admin

Recent and Current Job Executions

Stop All

ID	Instance	Name	Date	Start	Duration	Status	ExitCode
3	2	job1	2010-09-01	14:39:00	00:00:13	COMPLETED	COMPLETED
2	2	job1	2010-09-01	14:33:24	00:01:35	STOPPED	STOPPED
1	1	job1	2010-09-01	14:32:36	00:00:00	FAILED	FAILED
0	0	job1	2010-09-01	14:32:19	00:00:00	COMPLETED	COMPLETED

Rows: 1-4 of 4 Page Size: 20

© Copyright 2009-2010 SpringSource. All Rights Reserved.

Contact SpringSource

Done

S3Fox

Restart is a new Execution of the same Instance

Topics in this Session

- Batch Admin
- **Scaling and Parallel Processing**

Scaling and Parallel Processing



- First Rule:
 - Use the simplest technique to get the job done in the required time
 - Do not optimize/parallelize unnecessarily
- Options:
 - Multi-threaded Step (single process)
 - Parallel Steps (single process)
 - Remote Chunking of Step (multi process)
 - Partitioning a Step (single or multi process)

Multi-threaded Step

- Just add a TaskExecutor to the tasklet

```
<step id="loading">
    <tasklet task-executor="taskExecutor" throttle-limit="20" ... />
</step>
```

- ItemReaders etc. must be stateless or thread-safe
 - In particular be careful with restart information in the step execution context
 - Most off-the-shelf ItemReaders etc. are NOT thread-safe!
- Number of threads limited by task executor and throttle-limit (which defaults to 4)
 - Throughput may be further limited by other components; e.g. pooled DataSources

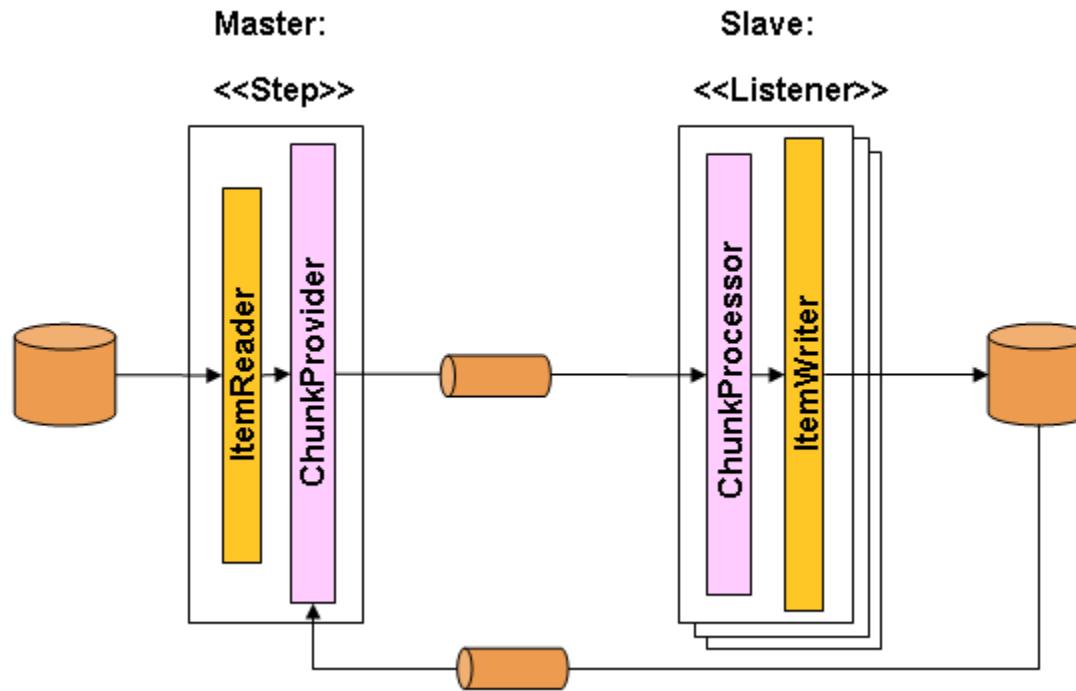
Parallel Steps

```
<job id="job1">
  <split id="split1" task-executor="taskExecutor" next="step4">
    <flow>
      <step id="step1" parent="s1" next="step2"/>
      <step id="step2" parent="s2"/>
    </flow>
    <flow>
      <step id="step3" parent="s3"/>
    </flow>
  </split>
  <step id="step4" parent="s4"/>
</job>

<beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>
```

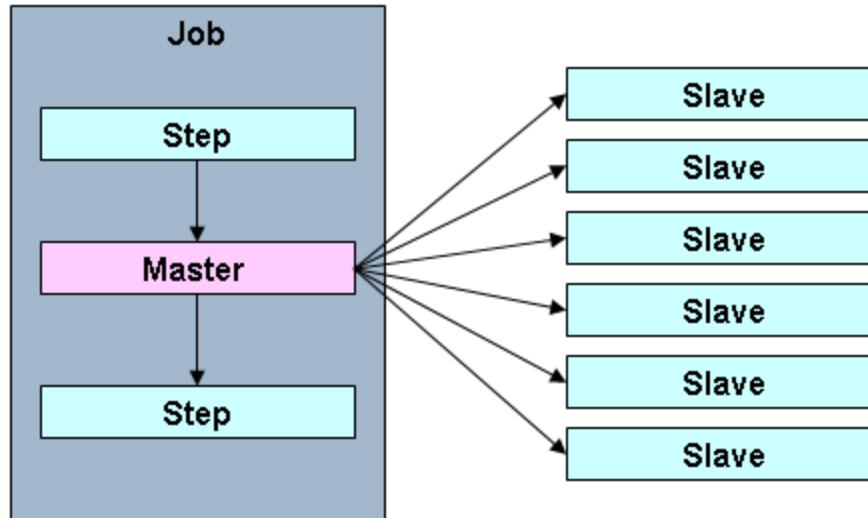
- (step1, step2) and (step3) run in parallel (two threads)
- All flows in split complete before aggregating exit statuses and transitioning to step 4

Remote Chunking



- Reader reads chunks; they are sent for remote processing
- Requires durable middleware (e.g. JMS) for communication between master and slaves
- See: spring-batch-integration project in Batch Admin

Partitioning

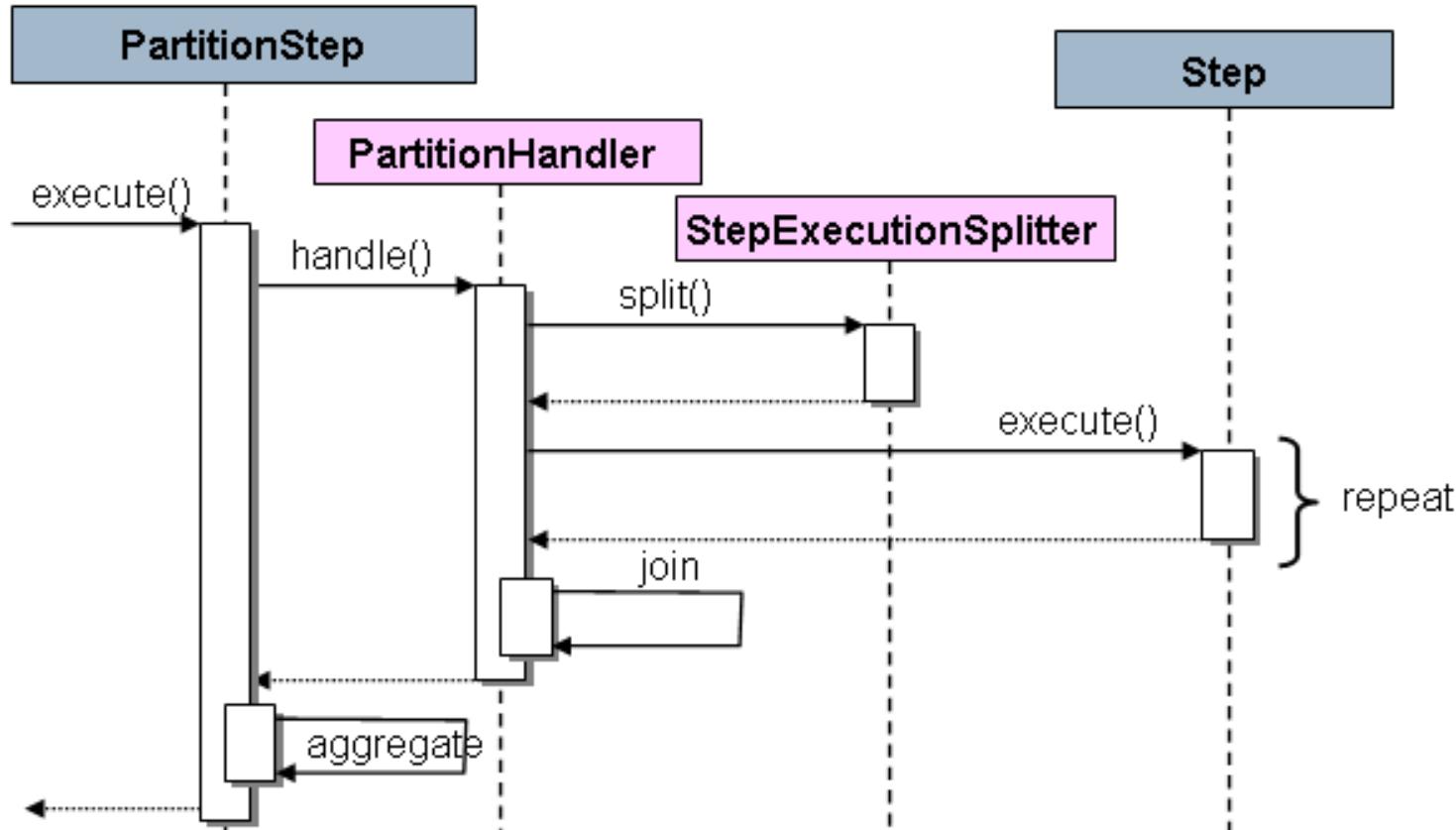


- Slaves can be local threads or remote
- Communication does not have to be durable because each partitioned step has its own execution context
- Spring Batch will ensure each is run once and once only per execution

Partitioning SPI

- The Partitioning SPI consists of
 - An implementation of Step (PartitionStep)
 - PartitionHandler
 - StepExecutionSplitter

Partition SPI Sequence Diagram



PartitionStep

```
<step name="step1:master">
  <partition step="step1" partitioner="partitioner">
    <handler grid-size="10" task-executor="taskExecutor"/>
  </partition>
</step>

<bean id="partitioner" class="org.spr...sample.common.ColumnRangePartitioner">
  <property name="dataSource" ref="dataSource" />
  <property name="table" value="CUSTOMER" />
  <property name="column" value="ID" />
</bean>
```

- The input data is partitioned (in this case into 10 partitions)
- Each partition is processed by a step named step1.partition0, ..., step1.partition9



Lab

Using Spring Batch Admin to Manage Jobs

What's next?

- Certification
- Other courses
- Resources
- Evaluation

Certification



- Computer-based exam
 - 50 multiple-choice questions
 - 90 minutes
 - Passing score: 76% (38 questions answered successfully)
- Where?
 - In any Pearson VUE Test Center
 - There are some in most big or medium-sized cities
 - See <http://www.pearsonvue.com/vtclocator/>



Certification (2)

- How to prepare for the exam?
 - See the EIwS 1.x certification guide here:
springsource.com/training/certification/enterpriseintegrationspecialist
 - Review all the slides
 - Redo the labs
- How to register?
 - At the end of the class, you will receive a voucher by email
 - For any further inquiry, you can write to
springsourceuniversity@vmware.com

Other courses

- Many courses available
 - Core Spring
 - Rich Web Applications with Spring
 - Hibernate with Spring
 - Groovy and Grails
 - tc Server, Tomcat, Hyperic
 - ...
- More details here:
<http://www.springsource.com/training/curriculum>

- 4 day course covering the Spring essentials
- Learn to build Enterprise Java applications with Spring
 - Dependency Injection
 - Testing
 - Aspect Oriented Programming
 - Data Access
 - Integration basics (some EIwS overlap)
 - Management and Monitoring
- Spring Professional Certification



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



Hibernate with Spring



- 3 day course that will give you the opportunity to:
 - Configure Hibernate applications with Spring and Spring Transactions
 - Implement inheritance and relationships with JPA and Hibernate
 - Discover how Hibernate manages objects
 - Go more in depth on locking with Hibernate
 - Advanced features such as interceptors, caching and batch updates

Consulting Offerings

- Quick Scan
 - Expert reviews project or architecture and shows how to improve
- Hyperic Jump Start
 - Helps you starting with Hyperic to monitor your environment
- Java EE to tc Server Migration
- Tomcat to tc Server Migration
 - Migrate your application and production environment to tc Server
- And custom consulting engagements that fits your specific needs

Resources

- The Spring reference documentation
 - <http://www.springsource.org/documentation>
 - For Spring Batch and Integration, check their own sites under <http://www.springsource.org/projects>
- The official technical blog
 - <http://blog.springsource.com/>
- The Spring community forums
 - <http://forum.springframework.org/>

Resources (2)

- You can register issues on our Jira repository
 - <http://jira.springframework.org/>
- The source code is available
 - SVN or Git, depending on the project
- Follow Spring development via RSS
 - Through fisheye.springsource.org, git.springsource.org or [github](http://github.com)

Thank You!

We hope you enjoyed the course

Please fill out the evaluation form

- **Go to <http://www.vmware.com/education>**
- **Click on “Class Evaluation”**
- **Follow your trainer's instructions**

