



Enterprise Spring

4 Day Workshop

Building loosely coupled event-driven

Version 4.2.a

Pivotal

Copyright Notice

Copyright © 2015 Pivotal Software, Inc. All rights reserved. This manual and its accompanying materials are protected by U.S. and international copyright and intellectual property laws.

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

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

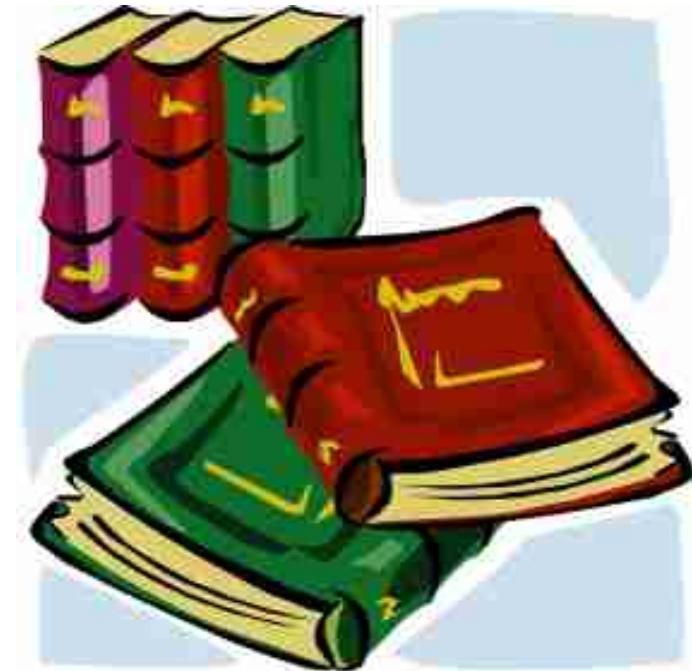
These materials and the computer programs to which it relates are the property of, and embody trade secrets and confidential information proprietary to, Pivotal Software, Inc., and may not be reproduced, copied, disclosed, transferred, adapted or modified without the express written approval of Pivotal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

Welcome to Enterprise 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 heterogenous 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, REST, JMS)
 - Spring Integration
 - Spring Batch
 - Spring XD

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
 - Lab environment based on
 - Spring Tool Suite (STS)
 - Apache Tomcat
 - Maven
 - Lab sources, documentation and dependencies

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
- RESTful web services
- Testing RESTful web services

Agenda: Day 2

- Introduction to Messaging
- Working with JMS
- Transactional JMS
- Global transaction management (XA and JTA)
- Introduction to Spring Integration

Agenda: Day 3

- Configuring Spring Integration
- Spring Integration Advanced Features
- Introduction to Spring Batch
- Restart and Recovery With Spring Batch

Agenda: Day 4

- Spring Batch Admin
- Scaling Batch Jobs
- Spring XD
 - Introduction
 - Streams
 - 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 explicit Spring configuration
 - <beans/> XML and @Configuration classes
- 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/>

These are all covered by our
Core Spring 4-day training course

Prerequisites: Tools

- Basic experience with Eclipse/STS
 - Navigating views and perspectives
 - Using content-assist (ctrl+space)
 - Using quick-fix (ctrl+1)
 - Running code
 - on a 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

- Participants list
- Self introduction
- Course registration
- Courseware
- Internet access
- Phones on silent
- Working hours
- Lunch and breaks
- Toilets/Restrooms
- Fire alarms
- Emergency exits
- Other questions?



LOGISTICS

Feedback!

- We value your input
 - Evaluation instructions provided at end of course
- Send any additional feedback or questions to
education@pivotal.io

About Pivotal

Cloud Foundry

*Cloud Independence
Microservices
Continuous Delivery
Dev Ops*



Development

*Frameworks
Services
Analytics*



Pivotal
Tracker

Big Data Suite

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



HAWQ



GREENPLUM

GEMFIRE[®]



Spring Projects

Spring Framework



Spring Social



Spring Cloud



Spring Session

Spring Android



Spring Web Flow

Spring Security



Spring Data



Spring Batch



Spring Integration



Spring (SOAP) Web Services



Spring AMQP



Spring Hateoas



Spring Mobile

Spring Reactor



Spring XD



Spring Boot

Lab: ei-course-intro

Getting to know the reference domain
and courseware environment

Styles of Enterprise Integration

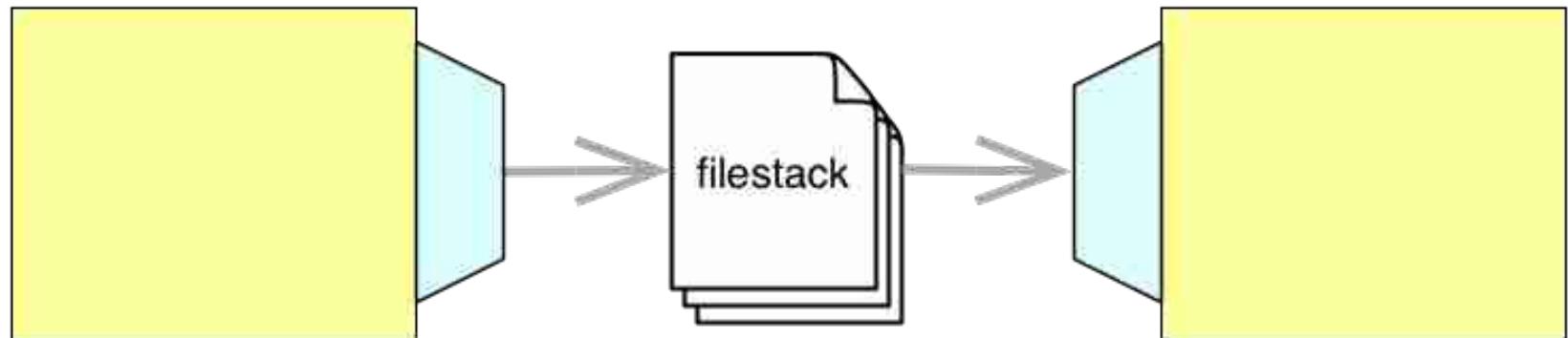
Topics in this session

- 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
 - Security

File Transfer: Overview



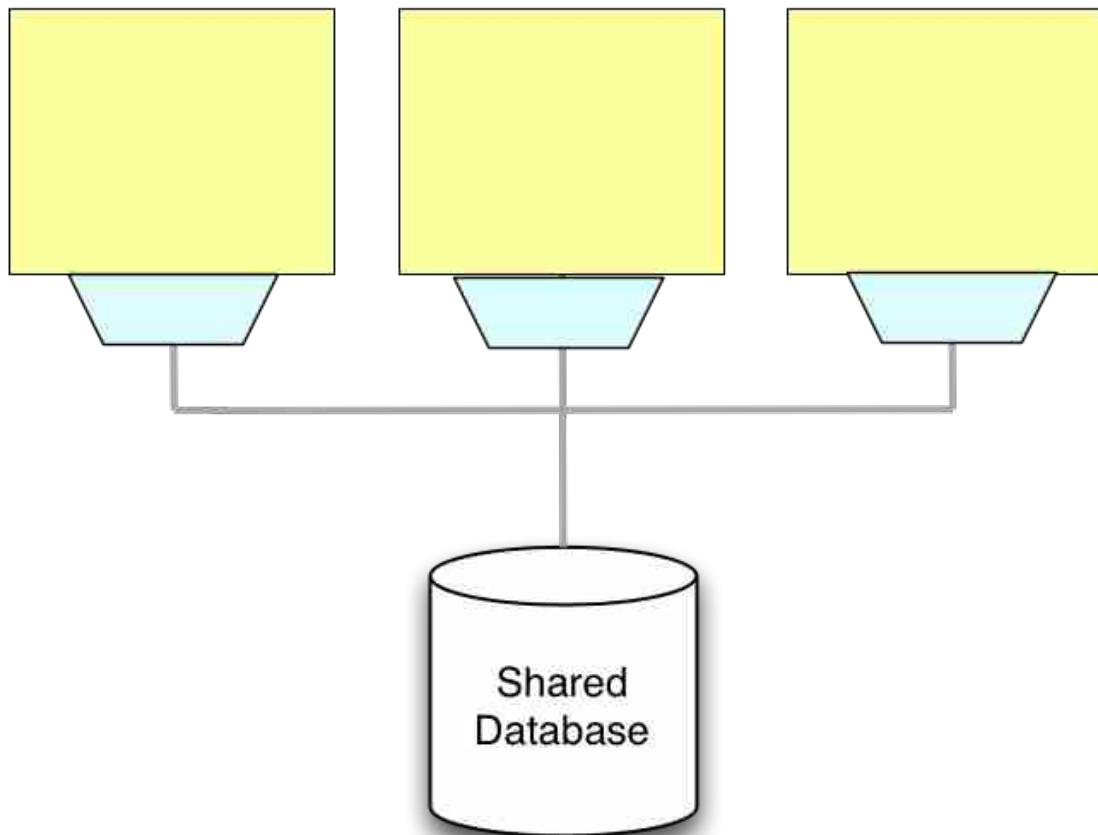
File Transfer: *Tradeoffs*

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

File Transfer: *Spring Support*

- 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

Shared Database: Overview



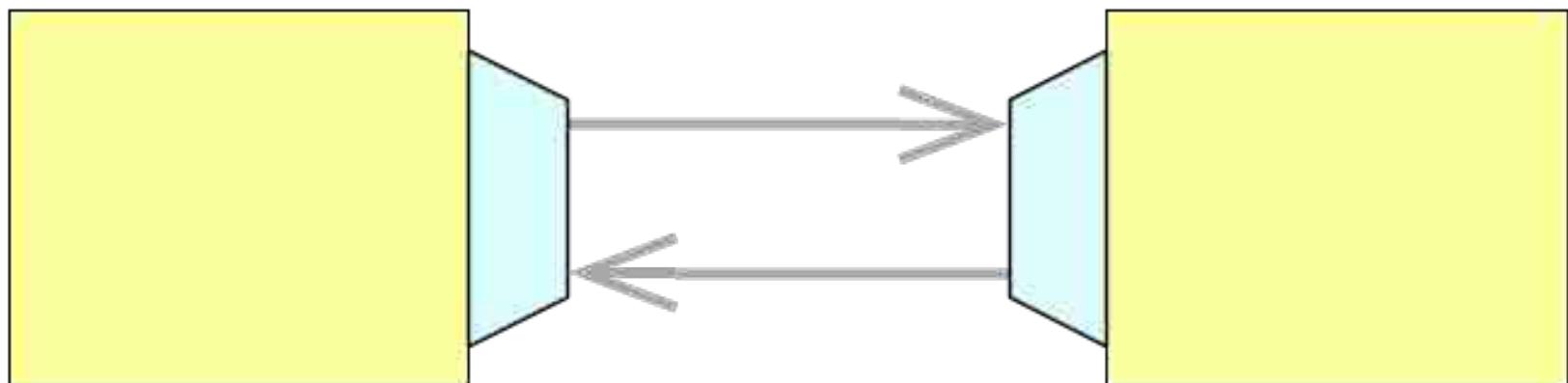
Shared Database: *Tradeoffs*

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

Shared Database: *Spring Support*

- DataAccessException Hierarchy
- Transaction Management
- Spring JDBC
- Spring-ORM integration
 - Hibernate, JPA
- 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
 - Typically not interoperable
 - Hard to version
 - Hidden complexity
 - Coupling (temporal & spatial)



Some modern remoting projects are interoperable and support versioning (e.g. Protocol Buffers, Thrift, Avro)

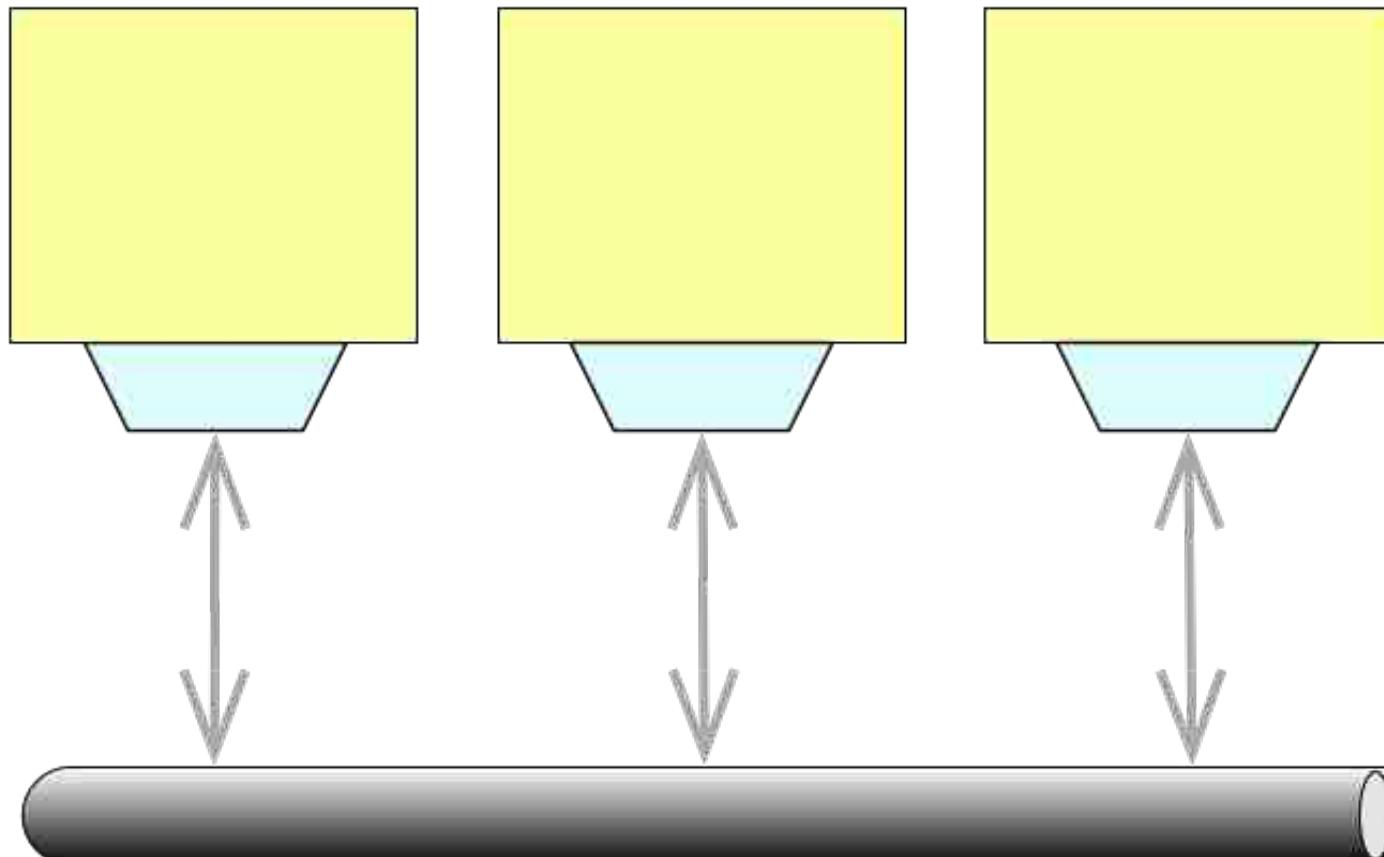
Remoting: *Spring Support*

- *ProxyFactoryBean* and *Exporter* implementations for
 - RMI
 - HttpInvoker
 - Hessian / Burlap
 - JAX-RPC
 - JAX-WS
 - JMS
 - RabbitMQ (Spring AMQP)
- Support for Google's Protocol Buffers in REST module



This course contains an optional module (slides and lab) on Spring Remoting.

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

SOAP Web Services: *Spring Support*

- Spring Web Services project
 - loosely-coupled, contract-first, messaging-style web services
 - WS-security support



This course focuses on REST Web Services, but contains optional modules (slides and labs) on Spring Web Services.

Summary

In this section we have covered

- Integration Styles
 - File Transfer
 - Shared Database
 - Remoting
 - Messaging
- Web Services
 - SOAP or REST
 - Can be used to implement remoting or messaging

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++

Example: 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_ff6fcfc5012.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

Implement `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); }  
    }  
}
```

Java Concurrency APIs

- Shouldn't typically use Threads directly
 - Thread management is low-level cross-cutting concern
- Java 5 and later 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.

java.util.concurrent.Executor

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

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

- Spring has similar *TaskExecutor* interface
 - Extends *java.util.concurrent.Executor*

java.util.concurrent.ExecutorService

- 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;  
  
    // ... any other methods  
}
```

java.util.concurrent.Executors

- 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() throws InterruptedException {  
        ExecutorService service = getExecutorService();  
        for (String photoUrl : urls) {  
            service.execute(new DownloadTask(photoUrl, tmpDir));  
        }  
        service.shutdown();  
        service.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);  
    }  
  
    private 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*, 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());  
}
```

ScheduledExecutorService / ScheduledFuture

- *ScheduledExecutorService* provides delay & repetition.
 - Use *ScheduledFuture* to obtain results.

```
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**

<task:> namespace

- A simple way to define TaskExecutors & Schedulers

```
<?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.xsd">

    <!-- add <task:> elements -->
</beans>
```

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/scheduling.html>

<task:executor />

- Defines an *Executor* instance
 - Specifically, a Spring *ThreadPoolTaskExecutor*.
 - Dependency inject wherever an *Executor* is needed:

```
<task:executor id="exec" ... />

public class PhotoDownloader {
    @Autowired Executor executor;
    ...
    ...
    for (String photoUrl : urls) {
        executor.execute(new DownloadTask(photoUrl, tmpDir));
    }
}
```

<task:executor/> examples

```
<task:executor id="exec" pool-size="5"/>
```

- Defines a thread pool, up to 5 threads

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

- Create up to 5 threads for new tasks
- Queue tasks when all threads used
- When queue full, allocate up to 10 threads max
- Reject tasks when capacity reached

```
<task:executor id="exec" pool-size="5-10" queue-capacity="50"  
    rejection-policy="CALLER_RUNS" />
```

- Run tasks in caller's thread instead of rejecting.

<task:scheduled-tasks/>

- Allows scheduled method execution:

```
<task:scheduled-tasks>
```

Calls the download() method every second

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

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

```
</task:scheduled-tasks>
```

Every 5 seconds, 9 to 5, weekdays

```
<bean id="photoDownloader">
```

```
  <!-- constructor args: urls, tmpDir -->
```

```
</bean>
```

Warning: *scheduled-tasks* uses
single-threaded Executor by default...

<task:scheduler/>

- Defines an *Executor* instance
 - Specifically, a Spring *ThreadPoolTaskScheduler*.
 - Dependency inject wherever *TaskScheduler* is needed.

```
<task:scheduler id="scheduler" pool-size="5"/>  
  
<task:scheduled-tasks scheduler="scheduler">  
  <task:scheduled ... />  
  <task:scheduled ... />  
</task:scheduled-tasks>
```

Provides a source for threads
to run 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

- @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

```
<task:annotation-driven executor="executor"/>  
  
<task:executor id="executor" ... />
```

Application Server Integration

- 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

Configure Tasks and Scheduling

REST

Representational State Transfer

An Introduction

Topics in this Session

- **What is REST?**
- Core REST Concepts
- RESTful architecture & design
- Advantages
- Appendix
 - HTTP Methods

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
- Appendix
 - HTTP Methods

Core REST Concepts

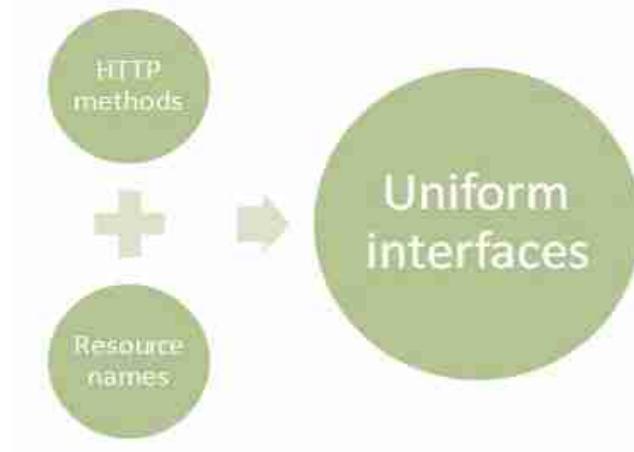
- Identifiable Resources
- Uniform interface
- Stateless conversation
- Resource representations
- Hypermedia

Identifiable Resources

- Everything is a resource
 - Customer
 - Car
 - Shopping cart
- Resources are expressed by URLs
 - 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)
- Use HTTP Methods
 - GET
 - POST
 - PUT, PATCH
 - DELETE
 - HEAD and OPTIONS
(for meta-data)



See *Appendix* at end of section for a overview of these commands

Resource representations

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

Stateless Protocol

- Server does not maintain state, client does
 - *Don't use the HTTP Session!*
- Very scalable
 - Any server instance can serve a request
 - Just put a load balancer in front
- Client maintains state through links
 - Just like a “real” application
 - Next few slides
- 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
 - Client needs some knowledge of server semantics
 - But REST has no WSDL
 - Should abstract fully: HATEOAS
 - See advanced section

REST Example: GET

```
GET /transfers/121  
Host: www.mybank.com  
Accept: application/json  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 83  
Content-Type: application/json  
  
{ id: 121, amount: 300.00,  
  credit: S123, debit: C456 }
```

**Accept header
defines representation**

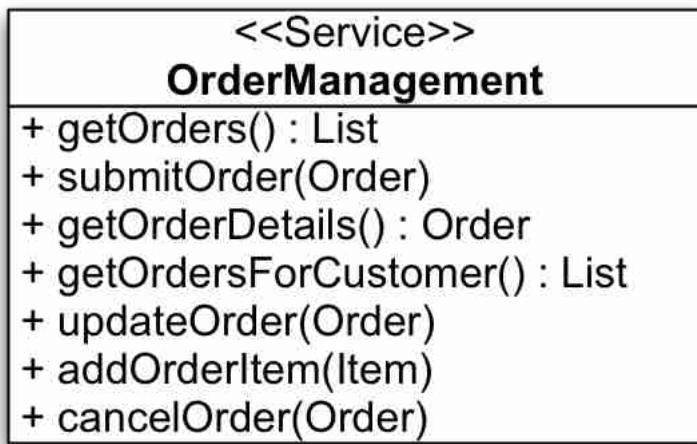
```
GET /transfers/121  
Host: www.mybank.com  
Accept: application/xml  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 1456  
Content-Type: application/xml  
  
<transfer id="121"  
          amount="300.00">  
  <credit>S123</credit>  
  <debit>C456</debit>  
</transfer>
```

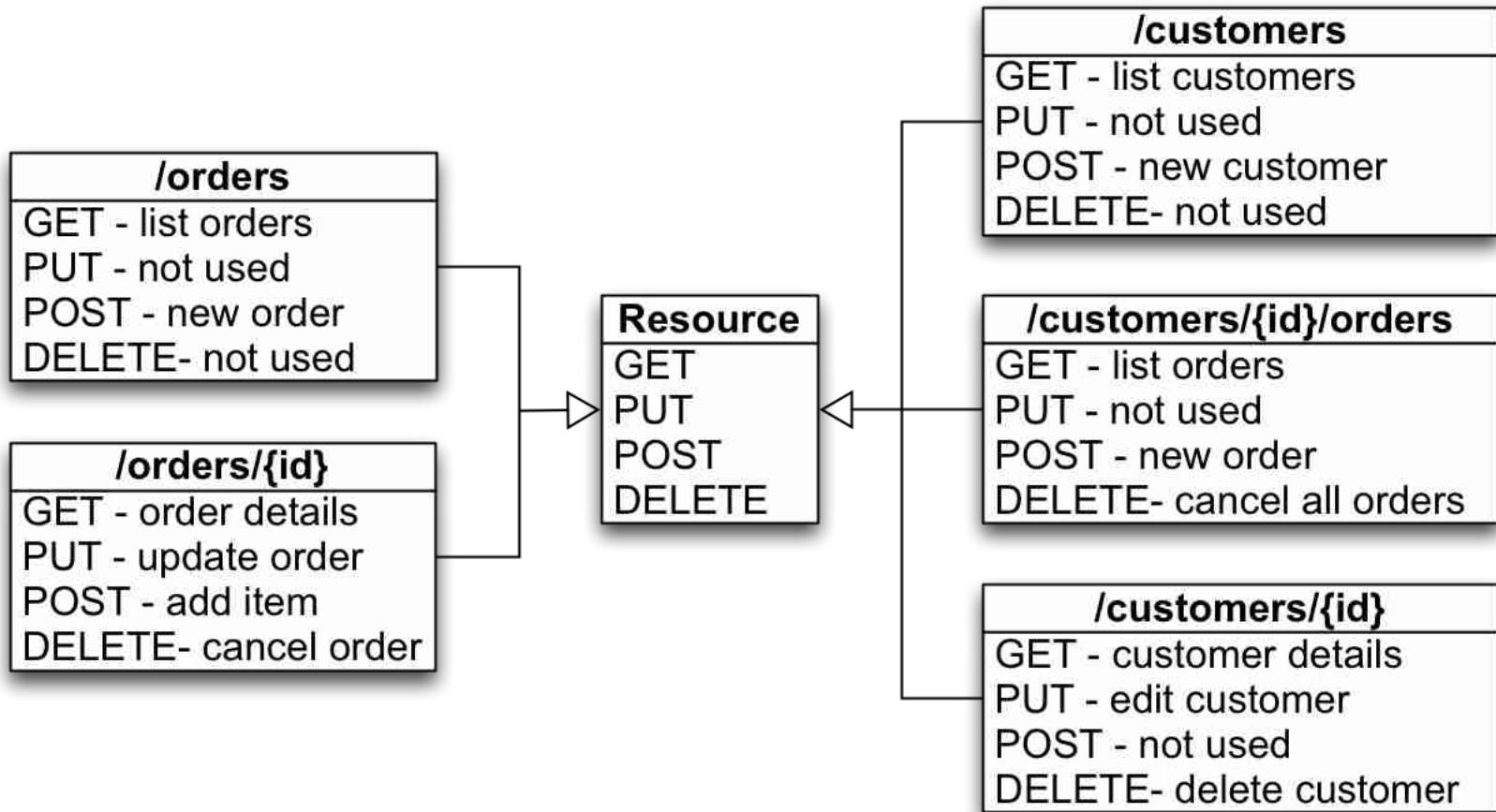
Topics in this Session

- What is REST?
- Core REST Concepts
- **RESTful architecture & design**
- Advantages
- Appendix
 - HTTP Methods

RESTful Architecture



RESTful Architecture



RESTful application design

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

HTTP Status Codes

- Web apps just use a handful of status codes
 - Success: 200 OK
 - Redirect: 302/303
 - Client Error: 404 Not Found
 - Server Error: 500 (such as unhandled Exceptions)
- RESTful applications use many additional codes to communicate with their clients

Common Response Codes

200

- After a successful GET where content is returned

201

- When new resource was created on POST or PUT
- Location header should contain URI of new resource

204

- When the response is empty
- e.g. after successful update with PUT or DELETE

404

- When requested resource was not found

405

- When HTTP method is not supported by resource

409

- When a conflict occurs while making changes
- e.g. when POSTing unique data that already exists

415

- When request body type not supported

Topics in this Session

- What is REST?
- Core REST Concepts
- RESTful architecture & design
- **Advantages**
- Appendix
 - HTTP Methods

Advantages

- Widely supported
 - Languages
 - Scripts
 - Browsers
 - only GET and POST through HTML
- Scalability
- Support for redirect, caching, different representations, resource identification, ...
- Support for JSON, but also other formats
 - XML 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
- OAuth 1 & 2 are also common solutions
 - Spring provides both client and server support

Transactions

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

Summary

You should have learned that:

- Java provides JAX-RS as standard specification
- Spring MVC adds REST support using a familiar programming model
 - Extended by `@RequestBody`, `@ResponseBody`, `@ResponseStatus`, `@PathVariable`
 - Message converters replace views
 - Automatic content-negotiation

Topics in this Session

- What is REST?
- Core REST Concepts
- RESTful architecture & design
- Advantages
- Appendix
 - HTTP Methods

Appendix: REST Methods

- Brief overview of the HTTP methods used by REST
 - GET
 - OPTIONS
 - POST
 - PUT
 - PATCH
 - DELETE

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 Examples

```
GET /transfers/121
```

Host: www.mybank.com

Accept: application/json

...

*Accept header
defines representation*

```
GET /transfers/121
```

Host: www.mybank.com

Accept: application/xml

...

```
HTTP/1.1 200 OK
```

Date: ...

Content-Length: 83

Content-Type: application/json

```
{ id: 121, amount: 300.00,  
  credit: S123, debit: C456 }
```

```
HTTP/1.1 200 OK
```

Date: ...

Content-Length: 1456

Content-Type: application/xml

```
<transfer id="121"  
          amount="300.00">  
  <credit>S123</credit>  
  <debit>C456</debit>  
</transfer>
```

ETags - Caching

```
GET /transfers/121
```

```
Host: www.mybank.com
```

```
...
```

```
HTTP/1.1 200 OK
```

```
Date: ...
```

```
ETag: "b4bdb3-5b0-  
43ad74ee73ec0"
```

```
Content-Length: 1456
```

```
Content-Type: text/html
```

```
...
```

```
GET /transfers/121
```

```
If-None-Match: "b4bdb3-  
5b0-43ad74ee73ec0"
```

```
Host: www.mybank.com
```

```
...
```

```
HTTP/1.1 304 Not Modified
```

```
Date: ...
```

```
ETag: "b4bdb3-5b0-  
43ad74ee73ec0"
```

```
Content-Length: 0
```

HEAD

- Get meta-information about a resource
 - Returns the same HTTP response headers as GET
 - But no content

```
GET /transfers/121
Host: www.mybank.com
Accept: application/json
...
```

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

OPTIONS

- What HTTP methods does the server support?
 - Not a widely supported HTTP method
 - May get a *501 Not Implemented* or *405 Method Not Allowed* response

```
OPTIONS /transfers
```

```
Host: www.mybank.com
```

```
HTTP/1.1 200 OK
```

```
Allow: GET, PUT, POST, OPTIONS, HEAD, DELETE, PATCH  
Accept-Patch: application/example, text/example
```

POST

- POST creates a new Resource
 - Usually as child of existing Resource
 - URI of child in Location response header
- POST is powerful, don't overuse it!
 - Not safe, not idempotent
 - Cannot just resend

Idempotent

Multiple invocations have same end-result

```
POST /transfers
Host: www.mybank.com
Content-Type: application/json

{amount: 300.00, credit: S123,
 debit: C456}
```

```
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)

Successful update – nothing to return

```
PUT /transfers/123
Host: www.mybank.com
Content-Type: application/json

{id: 123, amount: 300.00,
 credit: S123, debit: C456}
```

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

Results from a PUT

```
PUT /transfers/123
Host: www.mybank.com
Content-Type: ...
{id: 123, amount: 300.00,
credit: $123, ...}
```

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

Updated existing resource

Created new resource

HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://...
...

PATCH

- Just update *part* of a resource
 - A partial PUT
 - Not safe! (has side-effects), *not idempotent*
- Example below: just update amount field

```
PATCH /transfers/123
Host: www.mybank.com
Content-Type: application/json

{amount: 200.00}
```

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

DELETE

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

Successful
delete –
nothing to
return

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

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

HTTP Methods

Method	Safe	Idempotent	Cacheable
GET			
HEAD			
OPTIONS			
POST			
PUT			
PATCH			
DELETE			

Implementing REST using Spring

Representational State Transfer

Using RestTemplate and Spring MVC for REST

Topics in this Session

- **RestTemplate**
- **HttpEntity** and Subclasses
- Java Frameworks
- Spring MVC

RestTemplate Introduction

- Provides access to RESTful services
 - Encapsulates all HTTP Methods
 - Pass variables as Map or String...
 - Specify url as java.net.URI or String
- Configuration Options
 - Support for URI templates
 - Uses *HttpMessageConverters* (later slide)
 - Custom execute() with callbacks
 - Support for asynchronous interactions

RestTemplate API

HTTP	RestTemplate Method
DELETE	<code>delete(String url, Object... urlVariables)</code>
GET	<code>getForObject(String url, Class<T> responseType, Object... urlVariables)</code>
HEAD	<code>headForHeaders(String url, Object... urlVariables)</code>
OPTIONS	<code>optionsForAllow(String url, Object... urlVariables)</code>
POST	<code>postForLocation(String url, Object data, Object... urlVariables)</code> <code>postForObject(String url, Object data, Class<T> responseType, Object... uriVariables)</code>
PUT	<code>put(String url, Object data, Object... urlVariables)</code>
PATCH	Since Spring 3.2: using <code>exchange()</code> and <code>execute()</code>

Message Convertors – 1

- RestTemplate only deals in objects
 - Automated Object to/from payload conversion
 - Many defaults available “out-of-the-box”
 - Strings, JSON, XML, ...
 - Automatically used and configured if found on classpath
- Spring Version Changes
 - Jackson JSON V1 convertor deprecated by Spring 4.0, removed by Spring 4.1
 - Spring 4.1 added convertors for: Jackson data-format XML, Google's GSON and Google Prototype buffers

Message Convertors – 2

* Since Spring 4.1

Convertor	Data-type Handled	Media-Type
StringHttpMessageConverter	A string from a request or response	In: text/* Out: text/plain
MarshallingHttpMessageConverter	XML Payload using Spring OXM	application/xml
MappingJackson2XmlHttpMessageConverter	XML Payload using Spring OXM *	application/xml
MappingJackson2HttpMessageConverter	JSON Payload using Jackson V2	application/json
GsonHttpMessageConverter	JSON payload using Google's GSON library *	application/json
AtomFeedHttpMessageConverter	ATOM feed using Rome's Feed API	application/atom+xml
RssChannelHttpMessageConverter	ATOM feed using Rome's Feed API	application/rss+xml

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.springframework.web.client.RestTemplate">
    <property name="requestFactory">
        <bean class=
            "org.springframework.http.client.HttpComponentsClientHttpRequestFactory"/>
    </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);
```

{id} = 1

{id} = 1

Topics in this Session

- RestTemplate
- **HttpEntity and Subclasses**
- Java Frameworks
- Spring MVC

HttpEntity

- To handle request/response explicitly
 - Set headers, control content sent/received
 - Use `HttpEntity` and `HttpHeaders`

```
// Add our own message to the Hello website
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.TEXT_PLAIN);
HttpEntity<String> entity =
        new HttpEntity<String>("Hello Spring", headers);
URI location = template.postForLocation("http://hello.org", entity);

// Get the message back again
HttpEntity<String> entity2 =
        template.getForEntity(location, String.class);
String message = entity2.getBody();
MediaType contentType = entity2.getHeaders().getContentType();
```

RequestEntity, ResponseEntity

- Specific subclasses of `HttpEntity`
 - Fluent API since Spring 4.1
- Used by `RestTemplate.exchange()`
 - Accepts a `RequestEntity` instance
 - Allows more control over the request that is sent
 - A `ResponseEntity` is returned
- Can also return `ResponseEntity` instances from Controller methods (examples later)
 - Used to populate the `HttpServletRequest`
 - Again allow more control
 - Easier to write unit-tests for the Controller

RestTemplate.exchange()

```
// Add your own message to the 'Hello' website
ResponseEntity<Void> response = template.exchange(
    RequestEntity.post(new URI("http://hello.org"))
        .contentType(MediaType.TEXT_PLAIN)
        .body("Hello Spring"),
    null /* nothing to return */);
URI location = response.getHeaders().getLocation();

// Get the message back again
ResponseEntity<String> resp2 = template.exchange(
    RequestEntity.get(location)
        .accept(MediaType.TEXT_PLAIN).build(),
    String.class);
String text = resp2.getBody(); // Should be 'Hello Spring'
```

Topics in this Session

- RestTemplate
- ResponseEntity and Subclasses
- **Java Frameworks**
- Spring MVC

REST Frameworks

- Multiple Java frameworks for REST exist
- For Spring users, two options are particularly interesting:
 - 1) JAX-RS
 - 2) Spring MVC 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 since 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/json")
    public Reward getReward(@PathParam("number") String number) {
        return findReward(number);
    }
}
```

Spring MVC

- Spring MVC includes REST support (since 3.0)
 - 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 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

- RestTemplate
- HttpEntity and Subclasses
- Java Frameworks
- **Spring MVC**

@ResponseStatus

Will show a *much simpler* way to
Implement POST soon

- Overrides default Http response (200 - OK)
- When using @ResponseStatus, *void* methods no longer imply a default view name!
 - *There will be no View at all*
 - Code below returns a response with an *empty* body

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

Custom helper method
– see next slide

Determining Location Header

- Location header value must be full URL
 - Determine based on request URL
 - Controller shouldn't know host name or servlet path
- URL of created child resource usually a sub-path
 - POST to <http://www.myshop.com/store/orders> gives <http://www.myshop.com/store/orders/123>
 - Use Spring's *UriTemplate* to produce valid URL

```
private String getLocationForChildResource(HttpServletRequest request,  
                                         Object childIdentifier) {  
    StringBuffer url = request.getRequestURL();  
    UriTemplate template = new UriTemplate(url.append("/{childId}").toString());  
    return template.expand(childIdentifier).toASCIIString();  
}
```

Escapes illegal characters to create a valid URL

Control over the Response Status

- `@ResponseStatus` is static, how to have a conditional response status?
- Return a `ResponseEntity`

```
@RequestMapping("/rewards/{id}")
public ResponseEntity<?> get(@PathVariable String id) {
    ...
    if(found) {
        return new ResponseEntity(reward, HttpStatus.OK);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Dedicated instance for full control

Factory methods for common cases

@ResponseStatus & Exceptions

- Can also annotate your own exception classes
 - Given status code used when exception is thrown from controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
public class OrderNotFoundException extends RuntimeException {
    ...
}
```

```
@RequestMapping(value="/orders/{id}", method=GET)
public String showOrder(@PathVariable("id") long id, Model model) {
    Order order = orderRepository.findOrderById(id);
    if (order == null) throw new OrderNotFoundException(id);
    model.addAttribute(order);
    return "orderDetail";
}
```

@ExceptionHandler

- For existing exceptions you cannot annotate, use `@ExceptionHandler` method on controller
 - Method signature similar to request handling method
 - Also supports `@ResponseStatus`

```
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler({DataIntegrityViolationException.class})
public void conflict() {
    // could add the exception, response, etc. as method params
}
```

Accessing Request/Response Data

- Can explicitly inject HttpServletRequest (and HttpServletResponse)
 - But makes Controller methods hard to test
 - Consider Spring's MockHttp... classes
- Spring can automatically inject part of the request
 - @RequestParam, @PathVariable, Principal, Locale, @Value & SpEL, @RequestHeader
- To perform REST we also need:
 - @RequestBody, @ResponseBody

@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/json

{"id": 123, "items": [...], ...}

HTTP/1.1 204 No Content
Date: ...

Better Implementation of POST

- Don't depend on Servlet API!
 - Use UriComponentsBuilder

```
@RequestMapping(value="/rewards",method=POST)
public ResponseEntity<Void> create(
```

```
    @RequestBody Reward reward,
    UriComponentsBuilder builder){
```

```
    ...
    URI location = builder.path("/rewards/{rewardId}")
        .buildAndExpand(rewardId).toUri();
```

```
}
```

Passed in by
Spring MVC

Fluent API to
compute the URI

Use dedicated method to
return 201 Created



No need for *getLocationForChildResource*

@ResponseBody

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

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

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

HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type: application/json
{"id": 123, "items": [...], ...}

Content Negotiation - Accept

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    return orderRepository.findOrderById(id);
}
```

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

HTTP/1.1 200 OK
Date: ...
Content-Length: 756
Content-Type: application/json
{"id": 123, "items": [...], ... }

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>

Note on Configuration

- Spring provides many new features since 3.0
 - Such as the Message Convertors
 - But they break backwards compatibility
- They must be *enabled*
 - Either via an annotation or in the beans XML

```
<beans ... >  
  
    <!-- Enables default conversion service, validator  
        and message converters -->
```

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

```
@Configuration  
@EnableWebMvc  
public class RewardConfig { ... }
```

@RestController Simplification

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



No need for @ResponseBody on GET methods

Summary

You should have learned that:

- Spring provides RestTemplate for clients
- Java provides JAX-RS as standard specification
- Spring MVC adds REST support using a familiar programming model
 - Extended by @RequestBody, @ResponseBody, @ResponseStatus, @PathVariable
 - Automatic content-negotiation
- Spring uses message-convertors to automate content handling

Lab

Building a RESTful web service

Testing RESTful Controllers

Out-of-container testing

Topics in this Session

- **Out-of-container Testing**

The Need...

- Spring can eliminate *a lot* of the code
 - but the need for functional testing remains
- Very important to test the entire application from REST interface to DB
 - This usually involves deploying to an app server
- Spring's MVC Test Framework
 - Test entire application stack without app server
 - Suitable for both REST and non-REST Controller methods

MVC Test Framework Overview

- Began as a separate project on GitHub
 - Incorporated into Spring Framework since *Spring 3.2*
 - Found in `spring-test.jar`
- **Goal:** Provide first class *JUnit* support for testing Spring MVC code
 - Fluent API
 - Process requests *through* DispatcherServlet
 - Does ***not*** require Web container to test



SPRING TEST
FRAMEWORK

MVC Test Framework Example

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContentConfiguration("classpath:test-servlet-context.xml")
public final class AccountControllerTests {
    @Autowired
    private WebApplicationContext wac;
    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    @Test
    public void testBasicGet() {
        mockMvc.perform(get("/accounts")).andExpect(status().isOk());
    }
}
```

Tells test runner to instantiate a *WebApplicationContext*

Inject *WebApplicationContext*

Define *MockMvc* field and initialize it

Perform tests on *mockMvc* instance

Unit Testing vs using MVC Test Framework

- Unit Testing
 - Use to test *logic* of controllers
 - Create & populate model, test for proper views, model data, etc
- MVC Test Framework
 - Use to test MVC annotations (like `@RequestMapping`, `@PathVariable`,)
 - Also can perform integration tests with other Spring MVC beans (such as *View Resolvers*, *Filters*, etc)

Setting Up The Test Harness

```
import static  
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatcher.*;  
  
{@ContentConfiguration(...)  
@WebAppConfiguration  
@RunWith(SpringJUnit4ClassRunner.class)  
public final class AccountControllerTests {  
    @Autowired private WebApplicationContext wac;  
    private MockMvc mockMvc;  
  
    @Before  
    public void setup() {  
        // Initialize mockMvc using WebApplicationContext  
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();  
    }  
  
    // Now perform tests using the mockMvc object.  
}
```

Static imports make it easier to invoke Builder & Matcher static methods

Setting Up Static Imports

- Static imports are key to *fluid* builders
 - `MockMvcRequestBuilders.*` and `MockMvcResultMatchers.*`
- Eclipse/STS users can add to '*favorite static members*' in preferences
 - Java → Editor → Content Assist → Favorites
 - Add to favorite static members
 - `org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get`
 - `org.springframework.test.web.servlet.result.MockMvcResultMatchers.status`

Perform Test and Expect Results

- Argument to 'perform' dictates the action
 - 'perform' returns **ResultActions** object
 - Can chain **andExpect** methods together – *fluid syntax*

```
@Test  
public void basicAccountDetailsRequest() {  
    mockMvc.perform(get("/accounts/{acctId}", "123456001"))  
        .andExpect(status().isOk());  
    // Expect status code 200  
}
```

MockMvcRequestBuilders methods go in **perform**:
Options include *get, put, post, delete, fileUpload*

MockMvcResultMatcher
methods in *italics*

Matcher specific **assertion
methods** check result

RequestBuilder Methods

- Provides HTTP *get*, *put*, *post*, *delete* operations
 - Argument usually a URI template string
 - Embellish with *content*, *contentType*, *headers*, etc.
 - Returns a **MockHttpServletRequestBuilder** instance

```
// Setting Accept header
mockMvc.perform(get("/accounts/{acctId}", "123456001")
    .accept("application/json; charset=UTF-8")           // Request JSON Response

// Perform a put
mockMvc.perform(
    put("/accounts/{acctId}", "123456001")           // PUT request
    .content("{\"accountName\":\"json is nice\"}")        // with JSON
    .contentType(MediaType.APPLICATION_JSON)             // and content type
)
```

MockHttpServletRequestBuilder

Static Methods

- Many others available, see [JavaDoc](#)

Method	Description
header	Add a header variable to the request.
headers	Adds multiple headers
contentType	Set content type (Mime type) for body of the request
content	Actual request body to be sent
accept	Set requested type (Mime type) of expected response
locale	Set the locale for making requests

Assertions Using `andExpect`

- `andExpect()` accepts Matchers for specific assertions
 - Typically via a static method of `MockMvcResultMatcher`
 - Uses *Hamcrest* internally

```
@Test
public void testPut() {
    mockMvc.perform(
        get("/accounts/{acctId}", "123456001")           // GET request
    )
    .andExpect(status().isOk())                         // Expect status code 200
    .andExpect(content().contentType("application/json...")) // with a content type
    .andExpect(jsonPath("$.username").value("loislane")) // with specific JSON
    .andDo(print())                                     // and log to sysout too
    // ... many other static methods available ...
}
```

The diagram illustrates the annotations in the code with three callouts:

- A callout labeled "Perform ..." points to the `get()` method, indicating it performs a GET request.
- A callout labeled "... Expect" points to the `.andExpect()` methods, indicating they expect specific results.
- A callout labeled "... andDo" points to the `.andDo(print())` method, indicating it logs the response to sysout.

MockMvcResultMatcher methods

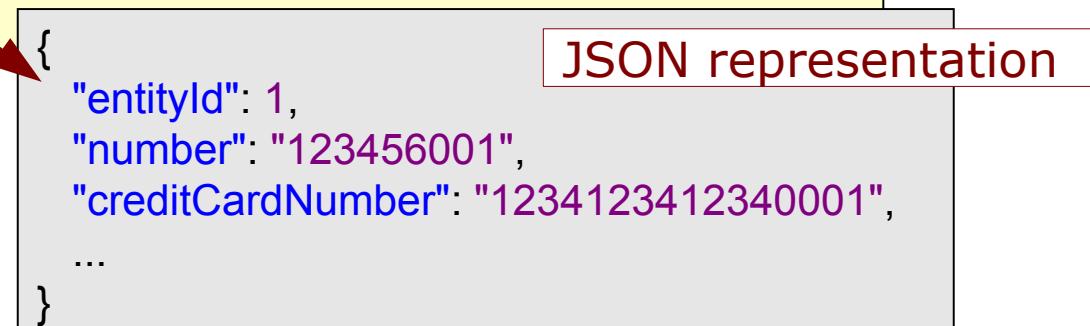
Method	Description	Examples
status()	HTTP status	<code>status().isOk(), status().isNotFound()</code>
content()	HTTP response body	<code>content() .contentType("application/json")</code>
header()	HTTP headers	<code>header() .string("content-length", 0)</code>
xPath()	Perform Xpath expressions on content	<code>Xpath("/user/@name") .string("loislane")</code>
jsonPath()	Perform JsonPath expressions on content	<code>jsonPath("\$.username") .value("loislane")</code>

Consult [JavaDoc](#) for details on various assertions

Testing RESTful Controllers I

- Can test RESTful interactions
 - Need to specify expected contentType and accept values for representation

```
@RequestMapping(method = RequestMethod.GET,  
    produces="application/json")  
public @ResponseBody Account get(@PathVariable String number) {  
    return accountManager.findAccount(number);  
}
```



Testing RESTful Controllers II

- Note use of accept() and jsonPath()
 - Assertions use JsonPath – like XPath for JSON
 - See <https://code.google.com/p/json-path/>

```
@Test  
public void testRestfulGet() throws Exception {  
    mockMvc.perform(get("/accounts/{acctId}", "123456001")  
        .accept(MediaType.APPLICATION_JSON))  
        .andExpect(content().contentType("application/json"))  
        .andExpect(jsonPath("$.number").value("123456001"));  
}
```

Printing Debug Information

- Sometimes you want to know what happened
 - `andDo()` performs action on `MvcResult`
 - `print()` sends the `MvcResult` to output stream
 - Can also use `andReturn()` to get the `MvcResult` object

```
// Use this to access the print() method
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
// Other static imports as well

// Use print() method in test to get debug information
mockMvc.perform(get("/accounts/{acctId}","123456001")
    .andDo(print()) // Add this line to print debug information to the console
    ...
    ...
```

Performing Security Testing

```
public class AccountControllerIntegrationTests {  
    @Autowired private WebApplicationContext context;  
    @Autowired private FilterChainProxy springSecurityFilterChain;  
    private MockMvc mockMvc;  
  
    @Before public void setup() {  
        mockMvc = webAppContextSetup(context).  
            addFilter(springSecurityFilterChain).build();  
    }  
  
    @Test public void requiresAuthentication() throws Exception {  
        mockMvc.perform(get("/"))  
            .andExpect(redirectedUrl("http://localhost/login"));  
    }  
}
```

Inject springSecurityFilterChain and add as a filter

Assert unauthenticated access redirects to the configured login page

Summary

You should have learned that:

- The Spring MVC Test framework offers an enhanced ability to test not just method calls but annotations, response content, status codes ... and more

Optional Advanced Section

- Implementing HATEOAS with Spring
- Using Spring Data Rest

Lab

Part 2: Out-of-container testing

Continue previous lab – STEP 2 TODOs

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

Messaging & Decoupling

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

Intra-application Integration

- 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

Use Cases

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

Use Cases

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

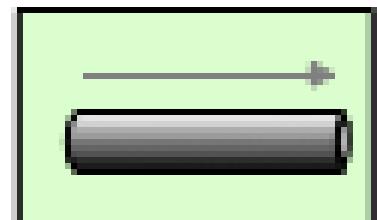
- Introduction
- Decoupling
- Use Cases
- **Enterprise Integration Patterns**

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

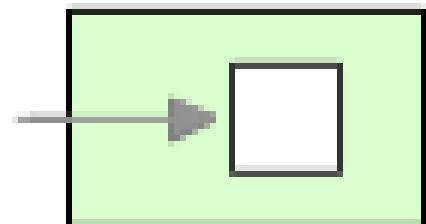
Message Channel

- 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



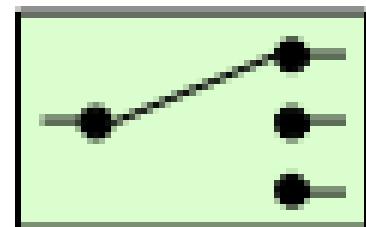
Message Endpoint

- 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



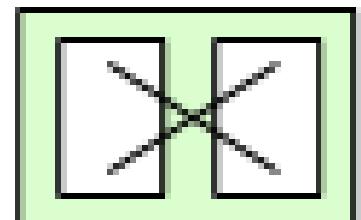
Message Router

- 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

Introducing JmsTemplate and Spring's Listener Container

Topics in this Session

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

Java Message Service (JMS)

- The JMS API provides an abstraction for accessing Message Oriented Middleware
 - Avoid vendor lock-in
 - Increase portability
- JMS does *not* enable different MOM vendors to communicate
 - Need a bridge (expensive)
 - Or use AMQP (standard msg protocol, like SMTP)
 - See RabbitMQ

JMS Core Components

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

JMS Message Types

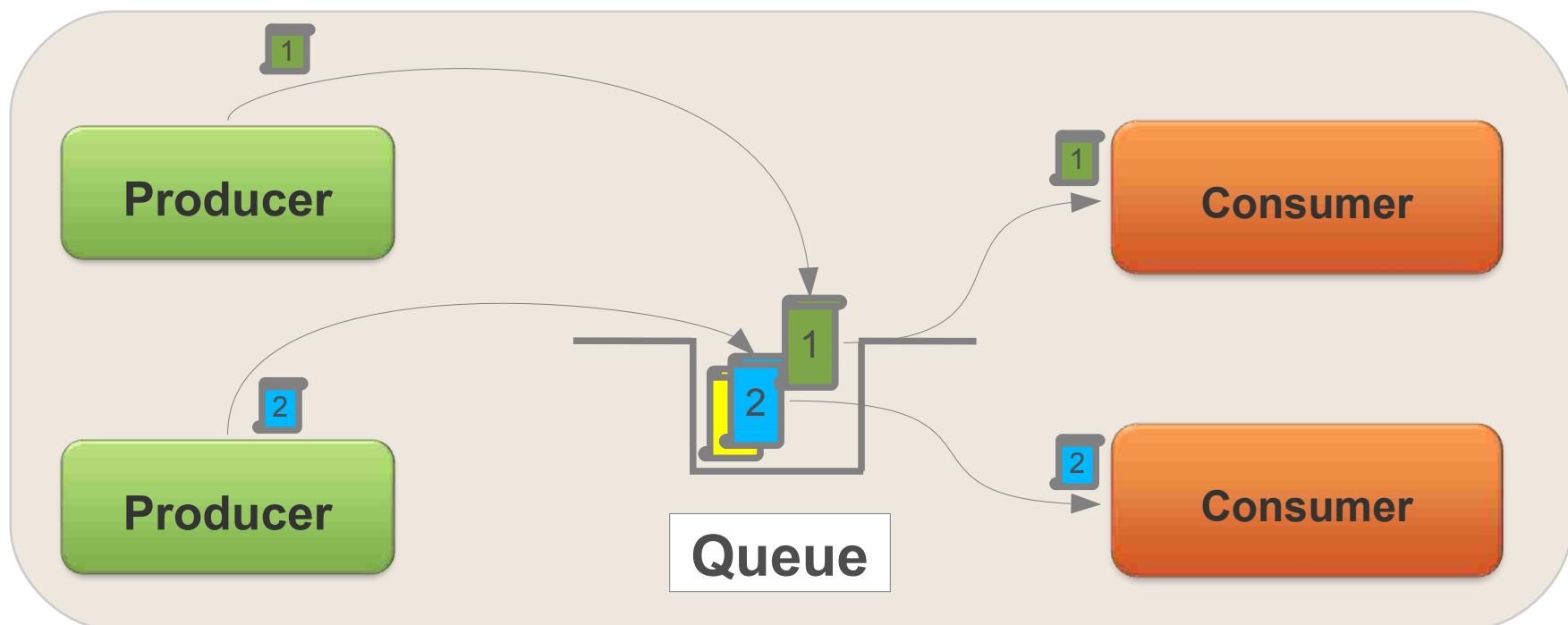
- Implementations of the Message interface
 - TextMessage
 - ObjectMessage
 - MapMessage
 - BytesMessage
 - StreamMessage

JMS Destination Types

- Implementations of the Destination interface
 - Queue
 - Point-to-point messaging
 - Topic
 - Publish/subscribe messaging
- Both support *multiple* producers and consumers
 - Messages are different
 - Let's take a closer look ...

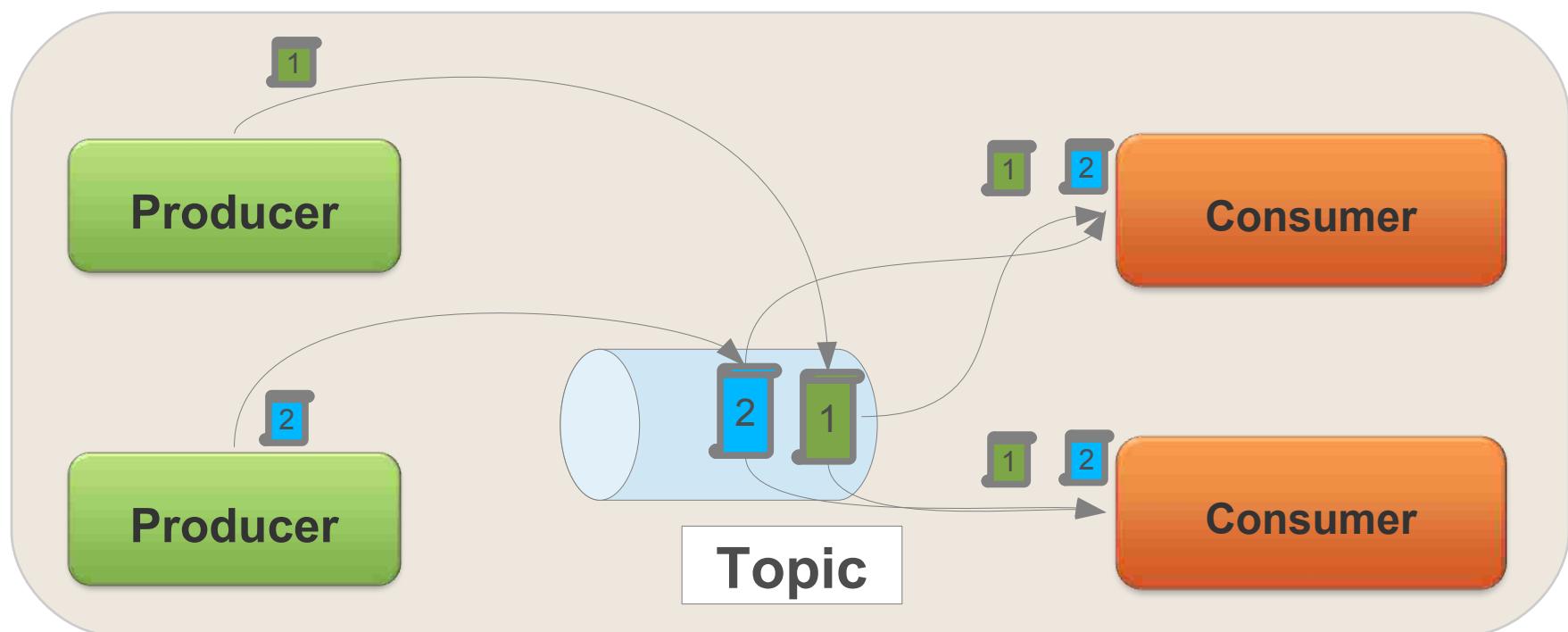
JMS Queues: Point-to-point

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



JMS Topics: Publish-subscribe

1. Message sent to topic
2. Message optionally stored
3. Message distributed to *all* subscribers



The JMS Connection

- A JMS Connection is obtained from a factory

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

- Typical enterprise application:
 - ConnectionFactory is a managed resource bound to JNDI

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

The JMS Session

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

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

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

Creating Messages

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

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

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

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

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

Producers and Consumers

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

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

Topics in this Session

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

JMS Providers

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

Apache ActiveMQ

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

Apache ActiveMQ Features

Support for:

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

Topics in this Session

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

Configuring JMS Resources with Spring

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



See: [Spring Framework Reference – Using Spring JMS](#)
<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#jms>

Configuring a ConnectionFactory Using XML

- 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 a ConnectionFactory Using Java Config

- ConnectionFactory may be standalone

```
@Bean  
public ConnectionFactory connectionFactory() {  
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();  
    cf.setBrokerURL("tcp://localhost:61616");  
    return cf;  
}
```

- Or retrieved from JNDI

```
@Bean  
public ConnectionFactory connectionFactory() throws Exception {  
    Context ctx = new InitialContext();  
    return (ConnectionFactory) ctx.lookup("jms/ConnectionFactory");  
}
```

Configuring Destinations Using XML

- Destinations may be standalone

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

- Or retrieved from JNDI

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

Configuring Destinations

Using Java Config

- Destinations may be standalone

```
@Bean  
public Destination orderQueue() {  
    return new ActiveMQQueue( "order.queue" );  
}
```

- Or retrieved from JNDI

```
@Bean  
public Destination connectionFactory() throws Exception {  
    Context ctx = new InitialContext();  
    return (Destination) ctx.lookup("jms/OrderQueue");  
}
```

Topics in this Session

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

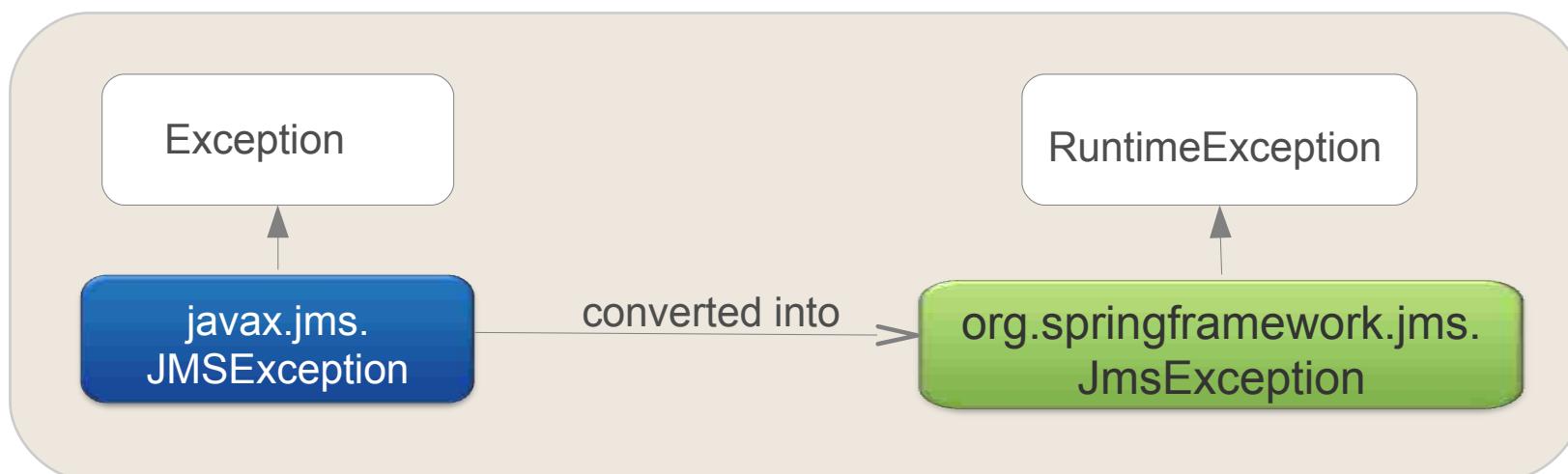
Spring's JmsTemplate

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

NOTE: The *AmqpTemplate* (used with RabbitMQ) has an almost identical API to the *JmsTemplate* – they offer similar abstractions over very different products

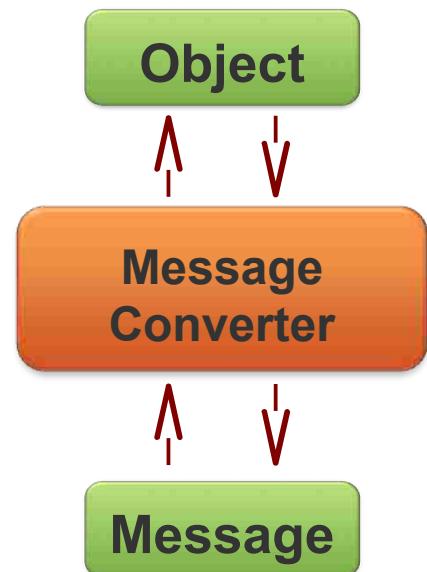
Exception Handling

- Exceptions in JMS are checked by default
- JmsTemplate converts checked exceptions to runtime equivalents



MessageConverter

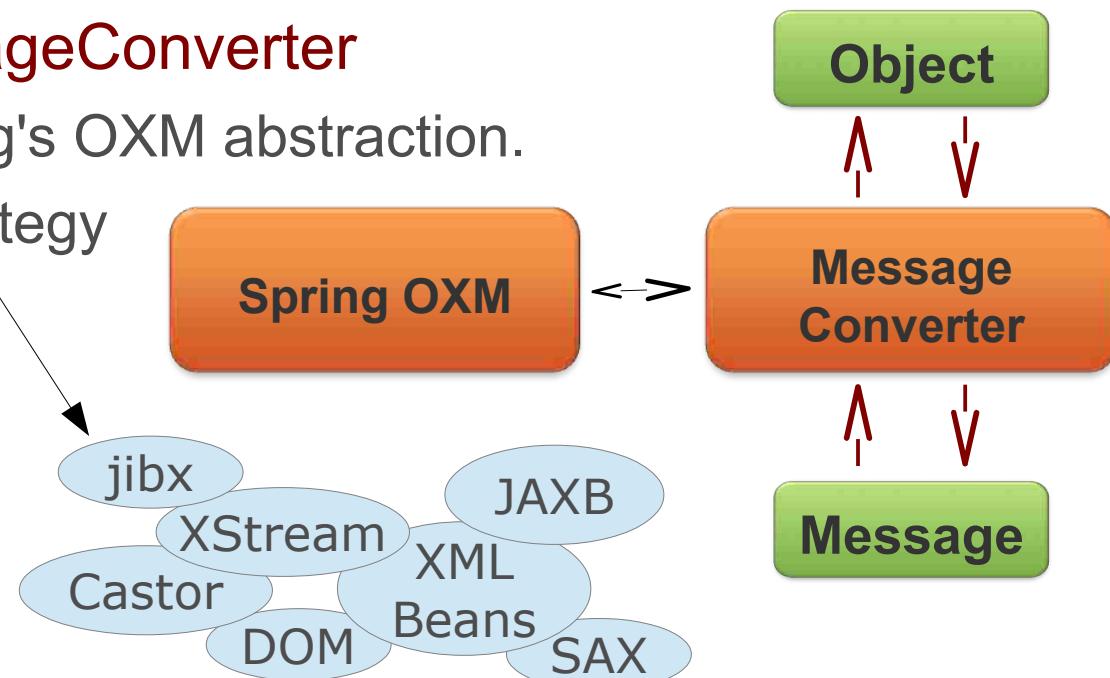
- The JmsTemplate uses a **MessageConverter** to convert between objects and messages
 - You only send and receive objects
- The default **SimpleMessageConverter** handles basic types
 - String to TextMessage
 - Map to MapMessage
 - byte[] to BytesMessage
 - Serializable to ObjectMessage



NOTE: It is possible to implement custom converters by implementing the *MessageConverter* interface

XML MessageConverter

- XML is a common message payload
 - ...but there is no “XmlMessage” in JMS
 - Use TextMessage instead.
- **MarshallingMessageConverter**
 - Plugs into Spring's OXM abstraction.
 - You choose strategy



MarshallingMessageConverter (XML)

```
<bean id="jmsTemplate" class="o.s.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="messageConverter" ref="msgConverter"/>
</bean>

<bean id="msgConverter"
      class="o.s.jms.support.converter.MarshallingMessageConverter">
    <property name="marshaller" ref="marshaller"/>
</bean>

<oxm:jaxb2-marshaller id="marshaller"
    contextPath="example.app.schema">
```

JAXB2 Illustrated here,
other marshallers available

MarshallingMessageConverter (Java)

```
@Bean public JmsTemplate jmsTemplate () {  
    JmsTemplate template = new JmsTemplate( connectionFactory );  
    template.setMessageConverter ( msgConverter() );  
    return template;  
}
```

```
@Bean public MessageConverter msgConverter() {  
    MessageConverter converter = new MarshallingMessageConverter();  
    converter.setMarshaller ( marshaller() );  
    return converter;  
}
```

```
@Bean public Marshaller marshaller() {  
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();  
    marshaller.setContextPath ( "example.app.schema" );  
    return marshaller;  
}
```

JAXB2 Illustrated here,
other marshallers available

DestinationResolver

- Convenient to use destination names at runtime
- DynamicDestinationResolver used by default
 - Resolves topic and queue names
 - Not their Spring bean names
- JndiDestinationResolver also available



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

publish-subscribe?
true q Topic
false q Queue

JmsTemplate Configuration – I

- *Must* provide reference to ConnectionFactory
 - via either constructor or setter injection
- Optionally provide delegates to handle some of the work using any of the following properties
 - messageConverter
 - destinationResolver
 - defaultDestination
 - defaultDestinationName

JmsTemplate Configuration – II

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <constructor-arg ref="connectionFactory"/>
    <property name="messageConverter" ref="...."/>
    <property name="destinationResolver" ref="...."/>
</bean>
```

XML Config

```
@Bean
public JmsTemplate jmsTemplate () {
    JmsTemplate template = new JmsTemplate( connectionFactory );
    template.setMessageConverter ( ... );
    template.setDestinationResolver ( ... );
    return template;
}
```

Java Config

Topics in this Session

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

Sending Messages

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

Sending POJO

- A message can be sent in one single line

```
public class JmsOrderManager implements OrderManager {  
    @Autowired private JmsTemplate jmsTemplate;  
    @Autowired private Destination orderQueue; → No @Qualifier so Destination  
                                                is wired by bean name  
    public void placeOrder(Order order) {  
        String stringMessage = "New order " + order.getNumber();  
        jmsTemplate.convertAndSend("messageQueue", stringMessage );  
                                                // use destination resolver and message converter  
  
        jmsTemplate.convertAndSend(orderQueue, order); // use message converter  
  
        jmsTemplate.convertAndSend(order); // use converter and default destination  
    }  
}
```

Sending JMS Messages

- Useful when you need to access JMS API
 - eg. set expiration, redelivery mode, reply-to ...

```
public void sendMessage(final String msg) {  
    MessageCreator messageCreator = new MessageCreator() {  
  
        public Message createMessage(Session session) throws JMSException {  
            TextMessage message = session.createTextMessage(msg);  
            message.setJMSExpiration(2000); // 2 seconds  
            return message;  
        }  
    };  
  
    this.jmsTemplate.send(messageCreator);  
}
```

Topics in this Session

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

Synchronous Message Reception

- JmsTemplate can also receive messages
 - but methods are blocking (synchronous)
 - optional timeout: **setReceiveTimeout()**

```
public void receiveMessages() {  
  
    // use message converter and destination resolver  
    String s = (String) jmsTemplate.receiveAndConvert("message.queue");  
    // use message converter  
    Order order1 = (Order) jmsTemplate.receiveAndConvert(orderQueue);  
  
    // handle JMS native message from default destination  
    ObjectMessage orderMessage = (ObjectMessage) jmsTemplate.receive();  
    Order order2 = (Order) orderMessage.getObject();  
}
```

Synchronous Message Exchange

- JmsTemplate also implements a request/reply pattern
 - Sending a message and blocking until a reply has been received (also uses receiveTimeout)
 - Manage a temporary reply queue automatically by default

```
public void processMessage(String msg) {  
  
    Message reply = jmsTemplate.sendAndReceive("message.queue",  
        (session) -> {  
            return session.createTextMessage(msg);  
        });  
    // handle reply  
}
```

The JMS MessageListener

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

```
import javax.jms.Message;
import javax.jms.MessageListener;

public class OrderListener implements MessageListener {
    public void onMessage(Message message) { // ...
}
```

- Traditionally, using a MessageListener required an EJB container
 - Message Driven Beans

Spring's MessageListener Containers

- 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 JMS Message Listener: XML

- 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 Spring's `SessionAwareMessageListener`
- `jms:listener-container` is configurable
 - task execution strategy, concurrency, container type, transaction manager and more

Spring's Message-Driven POJO: XML

- Spring also allows you to specify a plain Java object that can serve as a listener
 - MessageConverter provides parameter
 - Any return value sent to response-destination after conversion

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

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

Using Annotations

1 – POJO to Process Message

- Define a POJO to process message
 - Note: No references to JMS

```
public class OrderServiceImpl {  
    @JmsListener(destination="queue.order")  
    @SendTo("queue.confirmation")  
    public OrderConfirmation order(Order o) { ... }  
}
```

- Define as a Spring bean using XML, JavaConfig, or annotations as preferred
- **@JmsListener** enables a JMS message consumer for the method
- **@SendTo** defines response destination (optional)

Using Annotations

2 – Define JmsListenerContainerFactory

- JmsListenerContainerFactory: separates JMS API from your POJO:

```
@Configuration @EnableJms  
public class MyConfiguration {
```

```
    @Bean
```

```
    public DefaultJmsListenerContainerFactory  
        jmsListenerContainerFactory () {
```

```
        DefaultJmsListenerContainerFactory cf =  
            new DefaultJmsListenerContainerFactory( );  
        cf.setConnectionFactory(connectionFactory());
```

```
        ...  
        return cf;
```

Enable annotations

Default container name

Set
ConnectionFactory

Many settings available:
TransactionManager, TaskExecutor, ContainerType ...

@JmsListener features

- Container with name **jmsListenerContainerFactory** is used by default – or specify explicitly

```
public class OrderServiceImpl {  
    @JmsListener(containerFactory="myFactory",  
                 destination="orderConfirmation")  
    public void process(OrderConfirmation o) { ... }  
}
```

- Can also set a custom concurrency or a selector

```
public class OrderServiceImpl {  
    @JmsListener(selector="type = 'Order'",  
                 concurrency="2-10", destination = "order")  
    public OrderConfirmation order(Order o) { ... }  
}
```

Messaging: Pros and Cons

- Advantages
 - Resilience, guaranteed delivery
 - Asynchronous support
 - Application freed from messaging concerns
 - Interoperable – languages, environments
- Disadvantages
 - Requires additional third-party software
 - Can be expensive to install and maintain
 - More complex to use – *but not with JmsTemplate!*

Lab

Sending and Receiving Messages in
a Spring Application

Topics in this Session

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

SessionCallback Example

Synchronous Request-Reply

```
RewardConfirmation confirmation =
    jmsTemplate.execute(new SessionCallback() {

    public Object doInJms(Session session) throws JMSException {
        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;
    }
});
```

Advanced Option: CachingConnectionFactory

- JmsTemplate aggressively closes and reopens resources like Sessions and Connections
 - Normally these are cached by connection factory
 - Without caching can cause lots of overhead
 - Resulting in poor performance
- Use our CachingConnectionFactory to add caching within the application if needed

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

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 JMSEException;  
    // ...
```

- 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

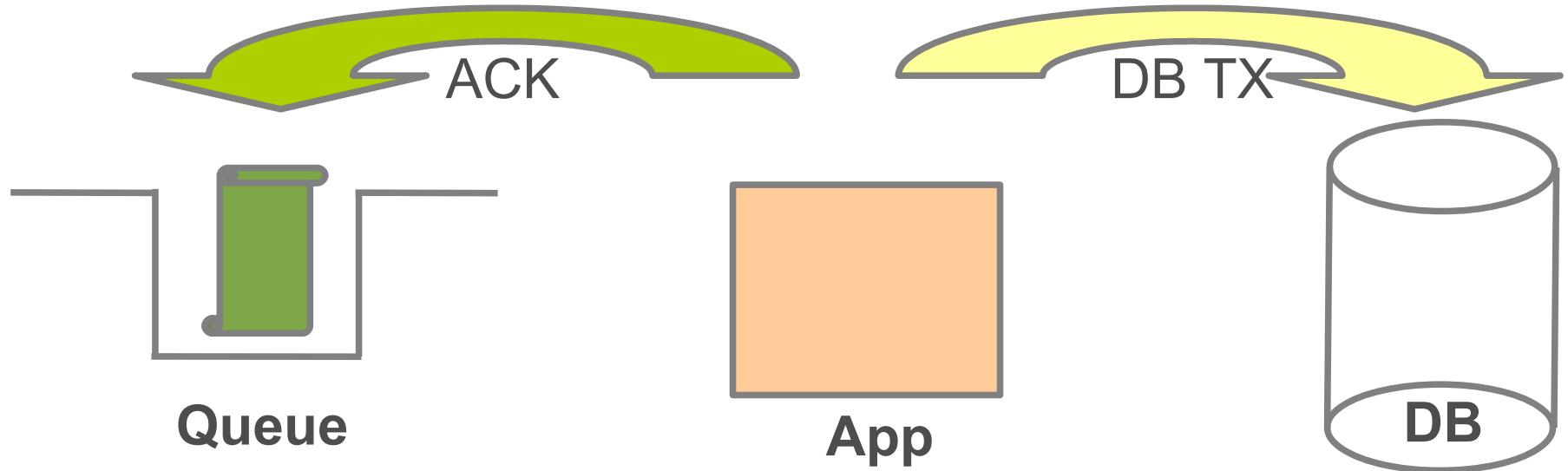
- Indicates when message is considered successfully delivered outside a transaction
- One of the following:
 - `AUTO_ACKNOWLEDGE`
after every successful `receive()` or `onMessage()`
 - `CLIENT_ACKNOWLEDGE`
client has to call `Message.acknowledge()` itself
 - `DUPS_OK_ACKNOWLEDGE`
lazily acknowledge delivery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

AUTO_ACKNOWLEDGE

- 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

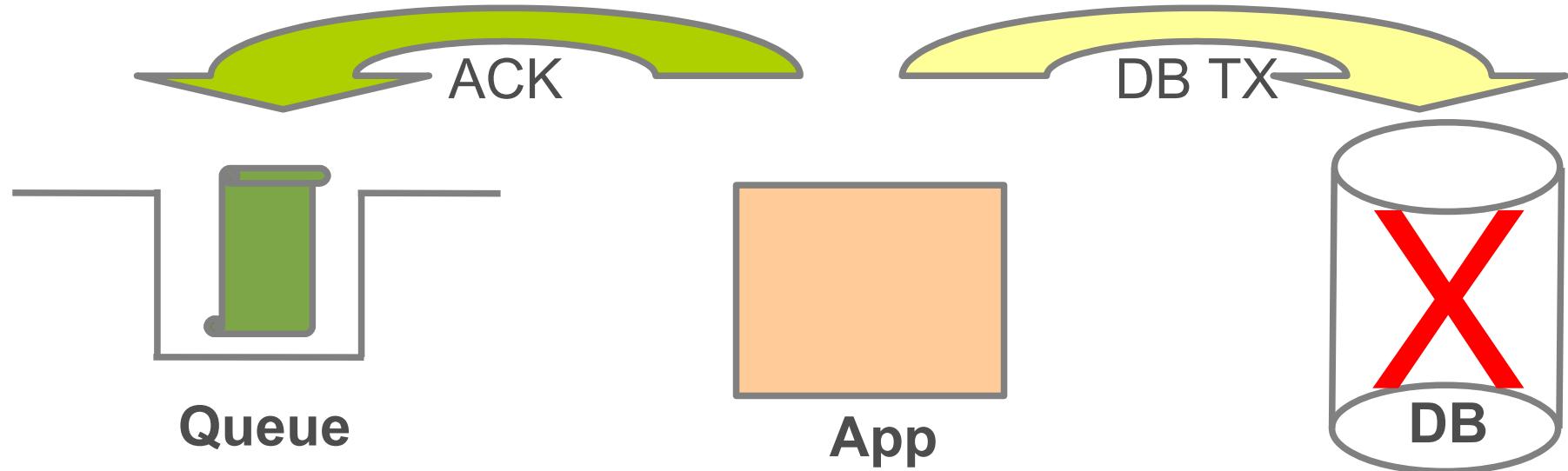


1. Receive & ack message
2. DB TX starts
3. Store message data
4. DB commits

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

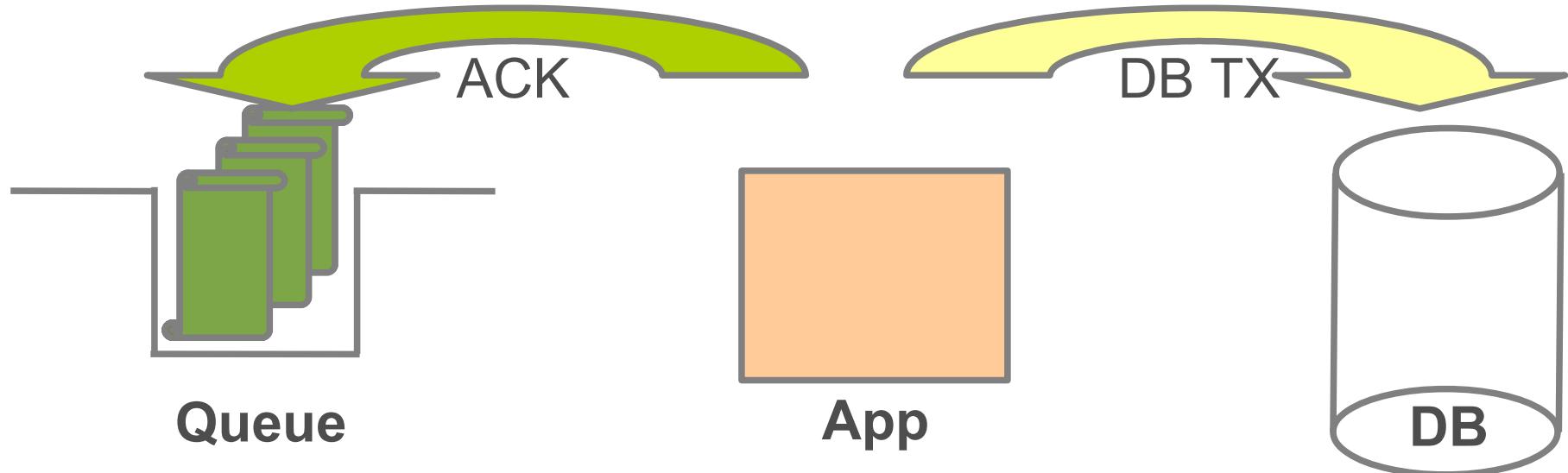
**Message not in DB and
no longer in Queue: lost!**

CLIENT_ACKNOWLEDGE

- 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!

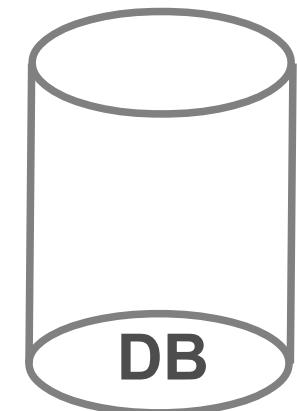
CLIENT_ACKNOWLEDGE

Happy Case Scenario



Queue

App

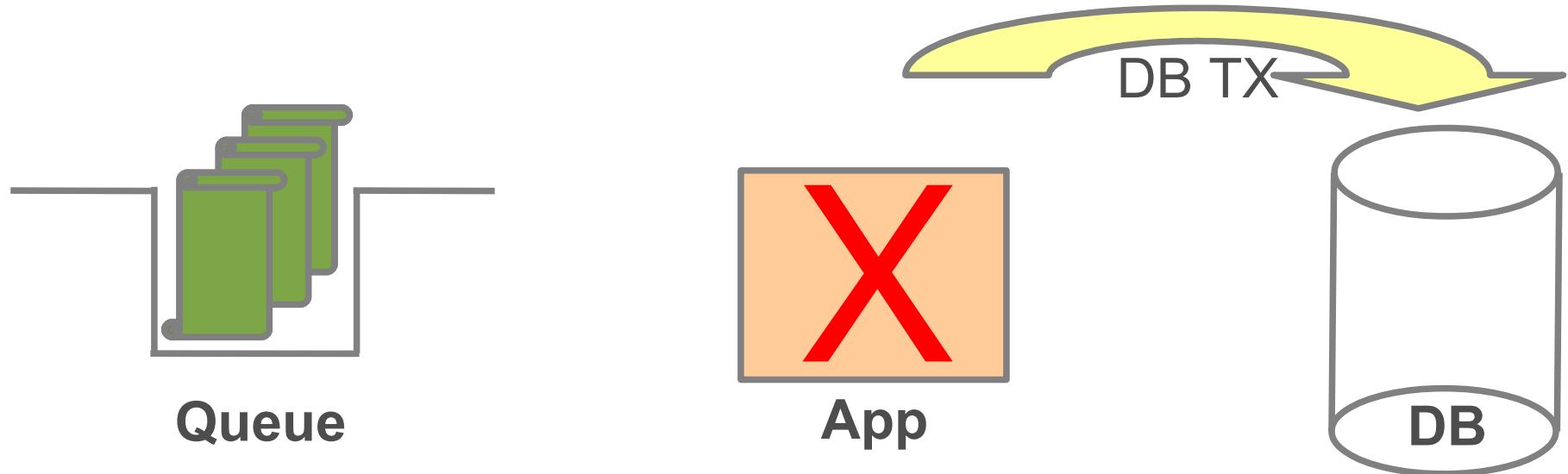


1. Receives
2. DB TX starts
3. Store message data
4. DB commits
5. Messages acknowledged

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

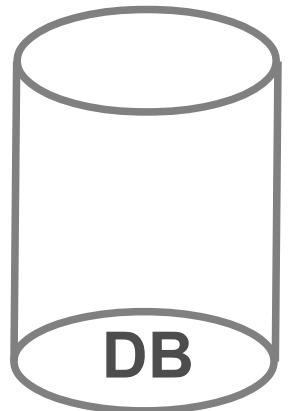
CLIENT_ACKNOWLEDGE

Duplicates On Error



Queue

App



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**

DUPS_OK_ACKNOWLEDGE

- 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
 - 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)`
 - Messages acknowledged on TX commit
- Global transaction (next chapter)
 - Multiple transactional resources managed together.
 - Values passed to `createSession` are ignored

Local JMS Transactions

- Only apply to JMS
 - Ignores other transactional resources
 - Transactions spanning multiple resources requires *global*
- Several 'best effort' strategies will work without and may 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

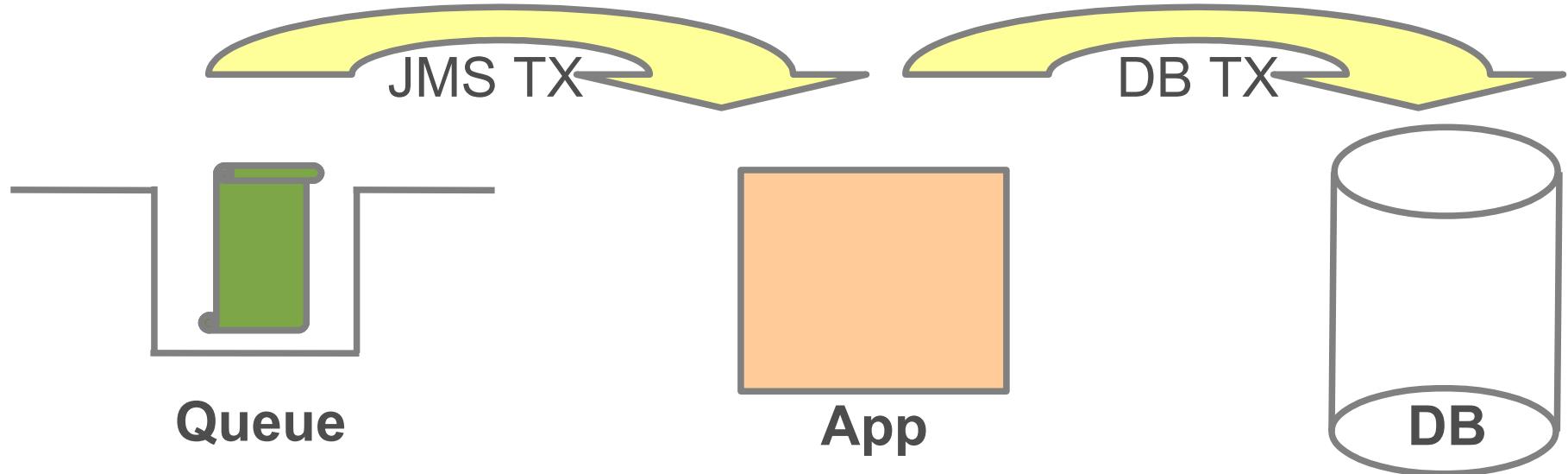
Spring Namespace Conventions

- Transacted Session and Acknowledge Mode are mutually exclusive in JMS ...
- ... so Spring's jms Namespace specifies them via a single attribute called **acknowledge**:
 - Can be `auto`, `client` or `dups_ok` for acknowledge mode
 - Or `transacted` for transacted session
- **transacted** starts local JMS Transaction
 - Assuming no global transaction is in progress
 - Will be synchronized with existing transaction

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

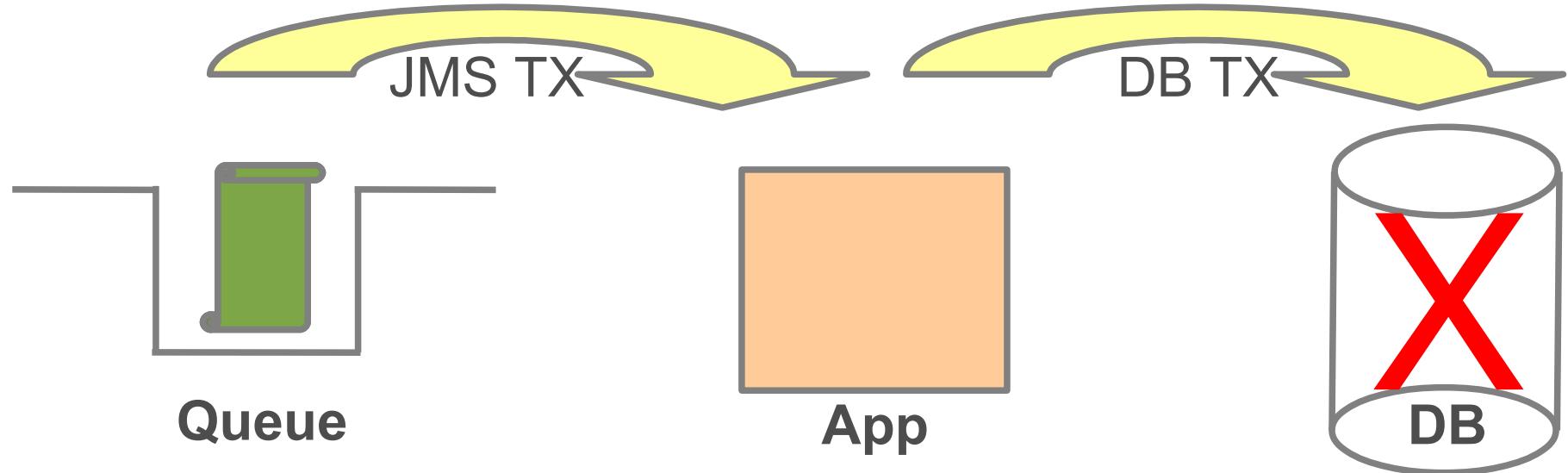
JMS+DB TX Synchronization: Happy Case Scenario



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

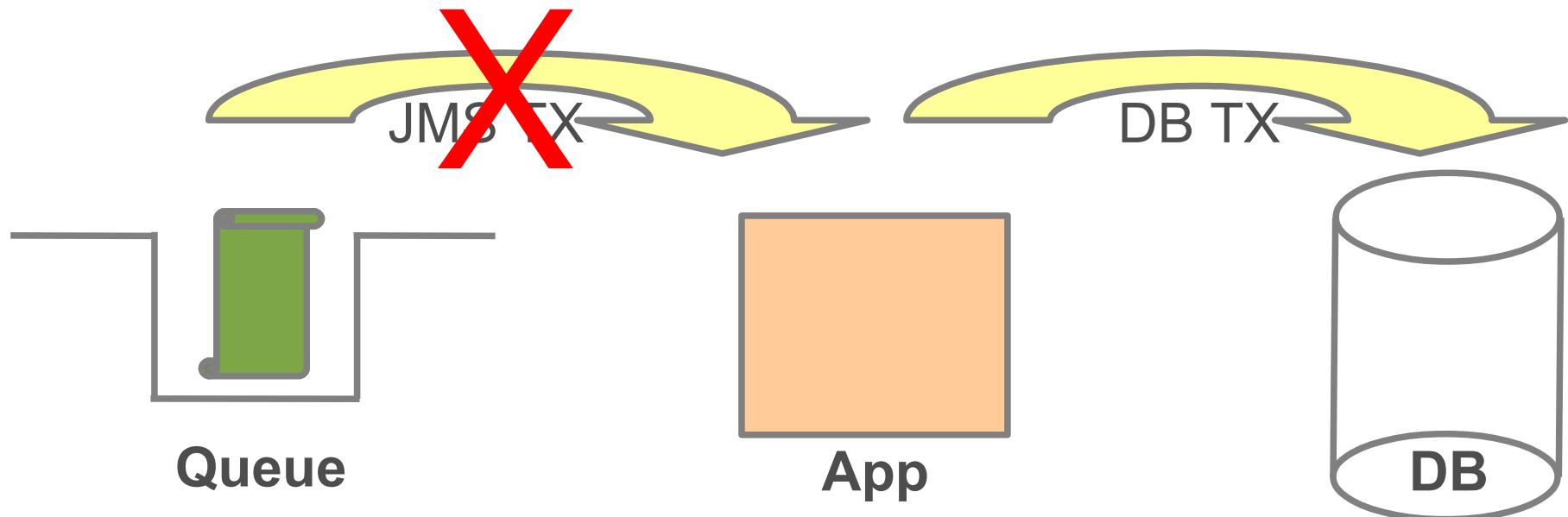
JMS+DB TX Synchronization: DB Rollback



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: results 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 (XML)

- 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 global, could also be JMS

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

Transacted Listener Container (Java)

- Set sessionTransacted flag to true
 - It gives you local JMS transaction

```
DefaultJmsListenerContainerFactory cf =  
    new DefaultJmsListenerContainerFactory( );  
cf.setSessionTransacted(true);
```

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

```
DefaultJmsListenerContainerFactory cf =  
    new DefaultJmsListenerContainerFactory( );  
cf.setTransactionManager(transactionManager);
```

Listener Container Transactions

- Transaction starts when message is received
- Setting the listener container's transaction manager to a global transaction manager enables global transactions.
 - Always consider if you really need global
 - 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

- Coordinating local transactions can cause duplicate messages in many scenarios
 - Only global 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)) { ... }
```

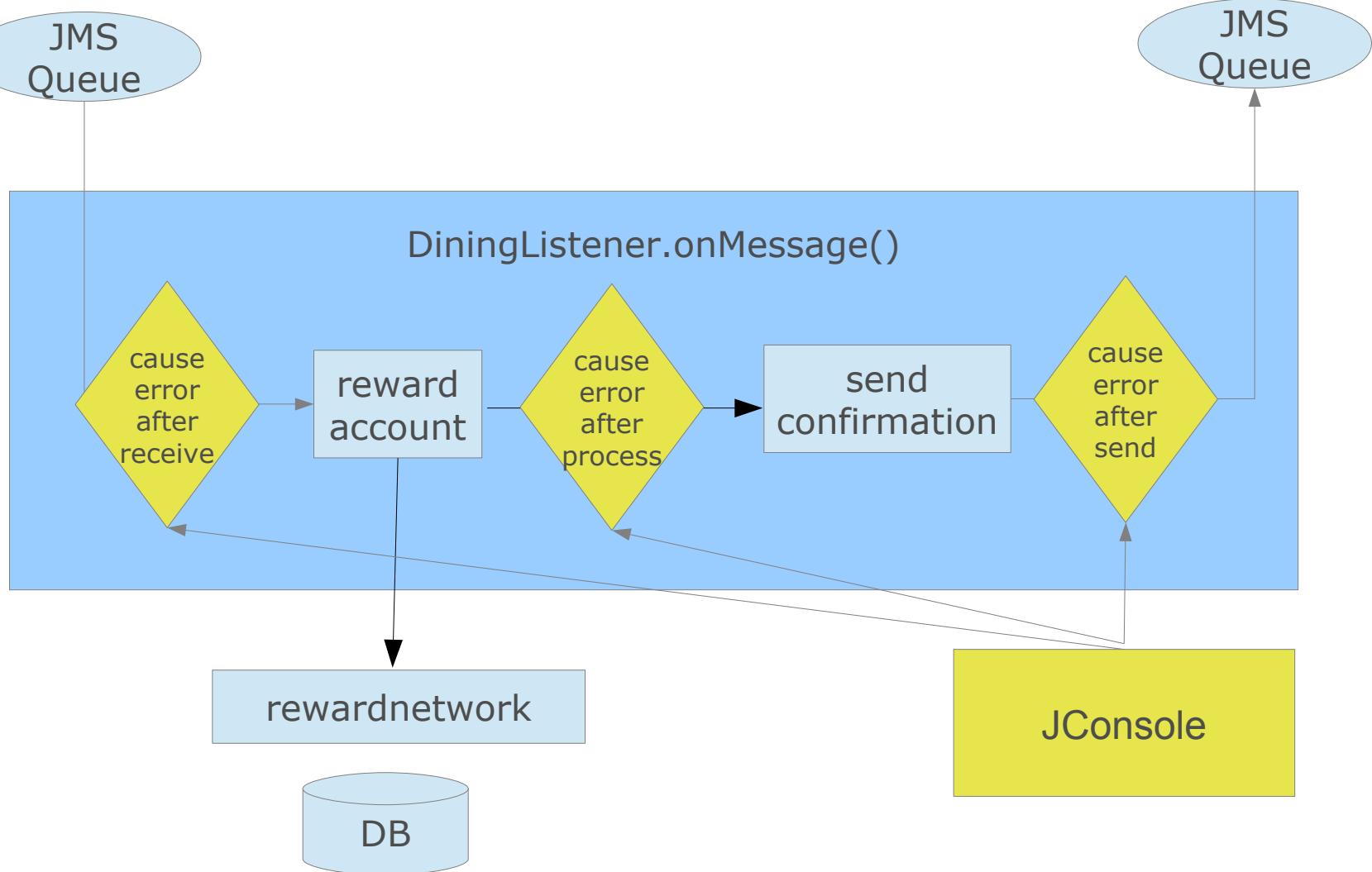
- You cannot tell this by looking at the message!
 - Could have been rolled back before or after processing
- Redeliveries are rare, so overhead of extra check is not incurred often

Summary

- To avoid duplicating or losing messages transactions can be used
- JMS and database transactions need to be synchronized
 - Prevents losing messages
 - Only global transaction management 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

Global Transactions

- 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

XA

- 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!

XA Commit Process

- 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

XA Aware Resources

- 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

JTA

- 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, Bitronix, JOTM, Narayana (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>
```

JTA components

- 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

TX Demarcation in Spring for JTA (XML)

- 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

TX Demarcation in Spring for JTA (Java)

- For databases or synchronous JMS, just use
 @Transactional with
 @EnableTransactionManagement
- For asynchronous JMS, pass JTA transaction manager to JMS listener container factory:

```
DefaultJmsListenerContainerFactory cf =  
    new DefaultJmsListenerContainerFactory( );  
cf.setTransactionManager(transactionManager);
```

Note: for local TXs you would use

cf.setSessionTransacted(true); 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 and Concepts
- Spring Integration
 - Basics
 - External integration
 - Visual Editor (STS)
- Summary

Spring Integration



Spring Integration

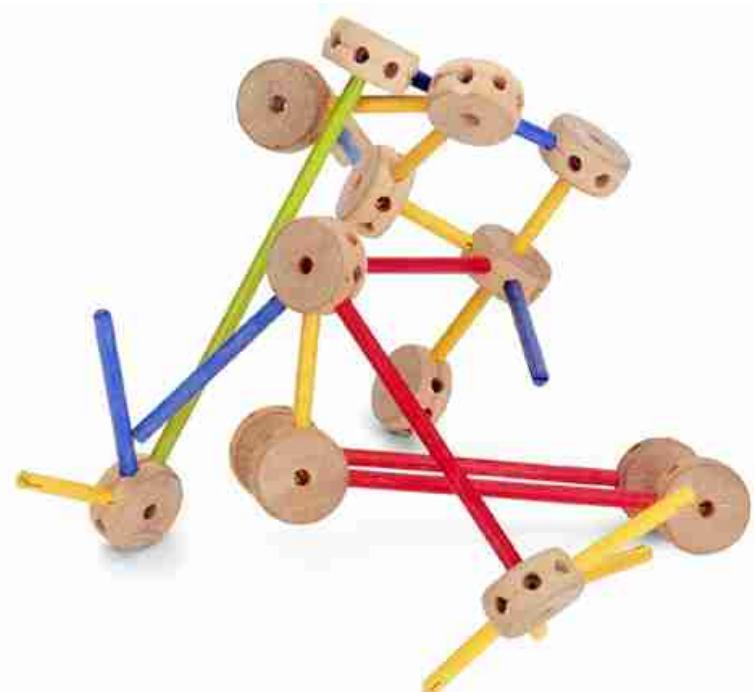
- Spring Integration allows you to
 - Let application components exchange data through in-memory messaging
 - Integrate with external applications in a variety of ways through adapters
- Origins
 - Builds on Enterprise Integration Patterns for both
 - Builds on the Spring portfolio & programming model
 - See <http://spring.io/spring-integration>



SPRING INTEGRATION

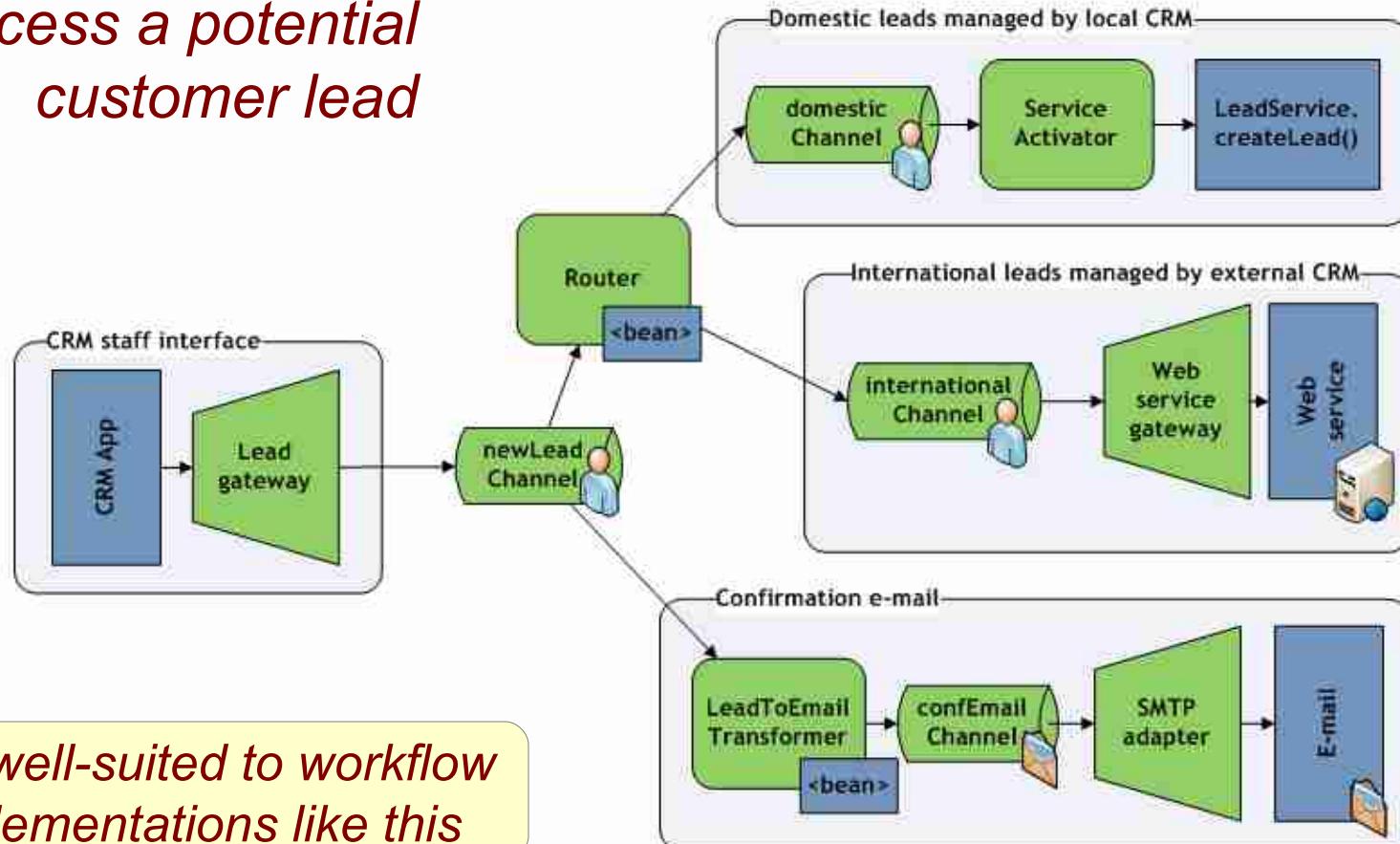
Tinker Toy™ Analogy

- Hub and spoke models
 - The spokes are communication *channels*
 - The hubs are your Java *components* or *endpoints*
- Use Spring Integration to build applications the same way
 - Architectural decision up-front



Example System using S.I.

Process a potential customer lead



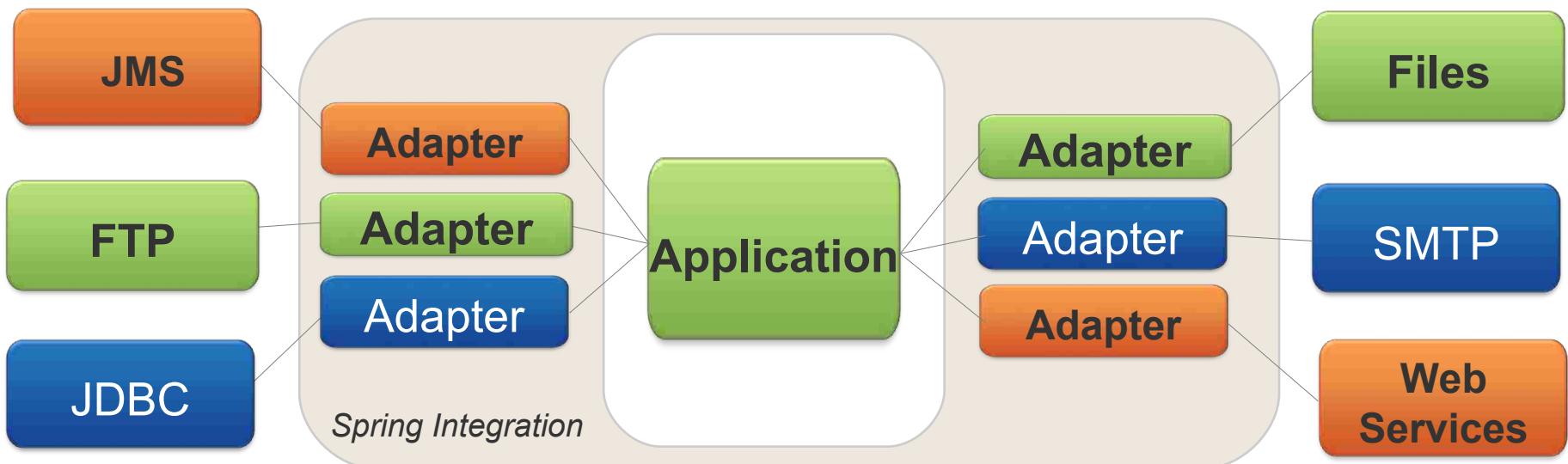
SI is well-suited to workflow implementations like this

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



Spring Integration - Adapters

- 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, ...

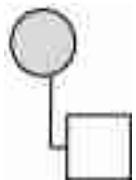
Topics in this session

- 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)

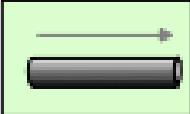




Message

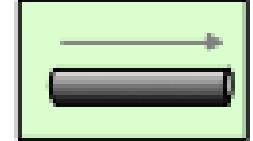
- Consists of **MessageHeaders** and a **Payload**
 - Some headers are pre-defined
 - Payload is just a Java object
- A Message is immutable
 - Let framework wrap payload for you or use a **MessageBuilder** to create it
- Each Message has a unique ID (header property)

```
public interface Message<T> {  
    T getPayload();  
    MessageHeaders getHeaders();  
}
```



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



MessageChannel Types

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

Endpoint
invoked in
same
thread

DIRECT CHANNEL

Endpoint

A



channel

Messages
not queued!

A

Endpoint
(Passive subscriber)

QUEUE CHANNEL

Endpoint

A

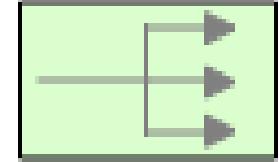


channel

Endpoint
invoked by
another
thread

Queue

Endpoint
(Active receiver)



MessageChannel Types

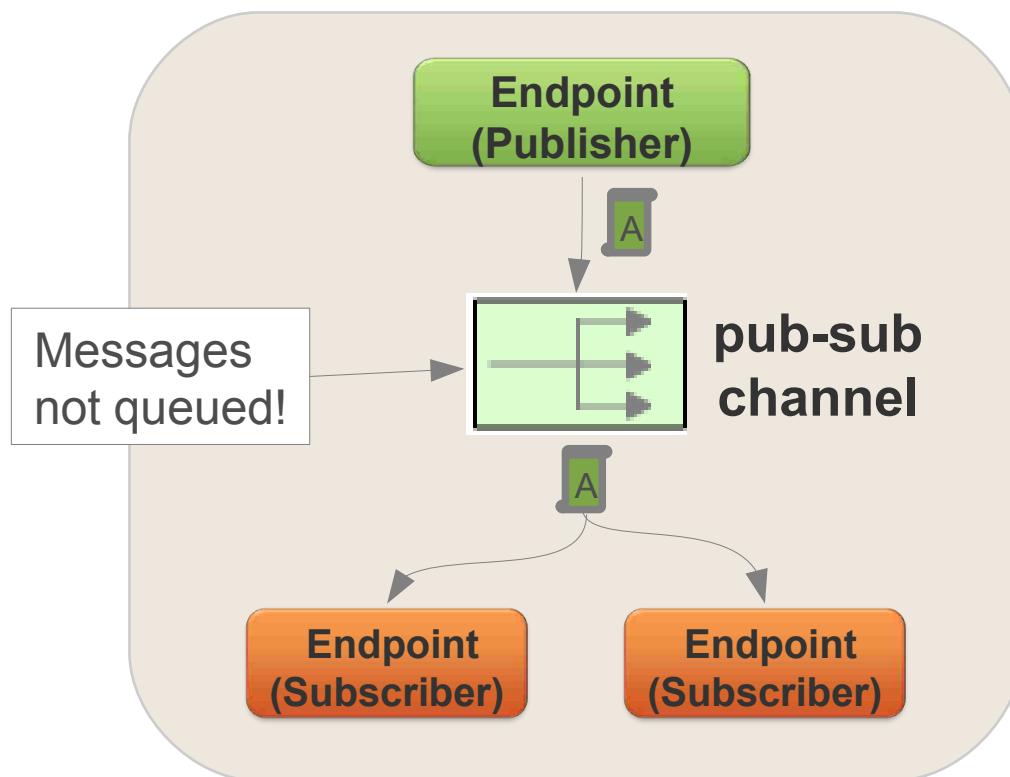
Publish-subscribe

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

MessageChannel Types

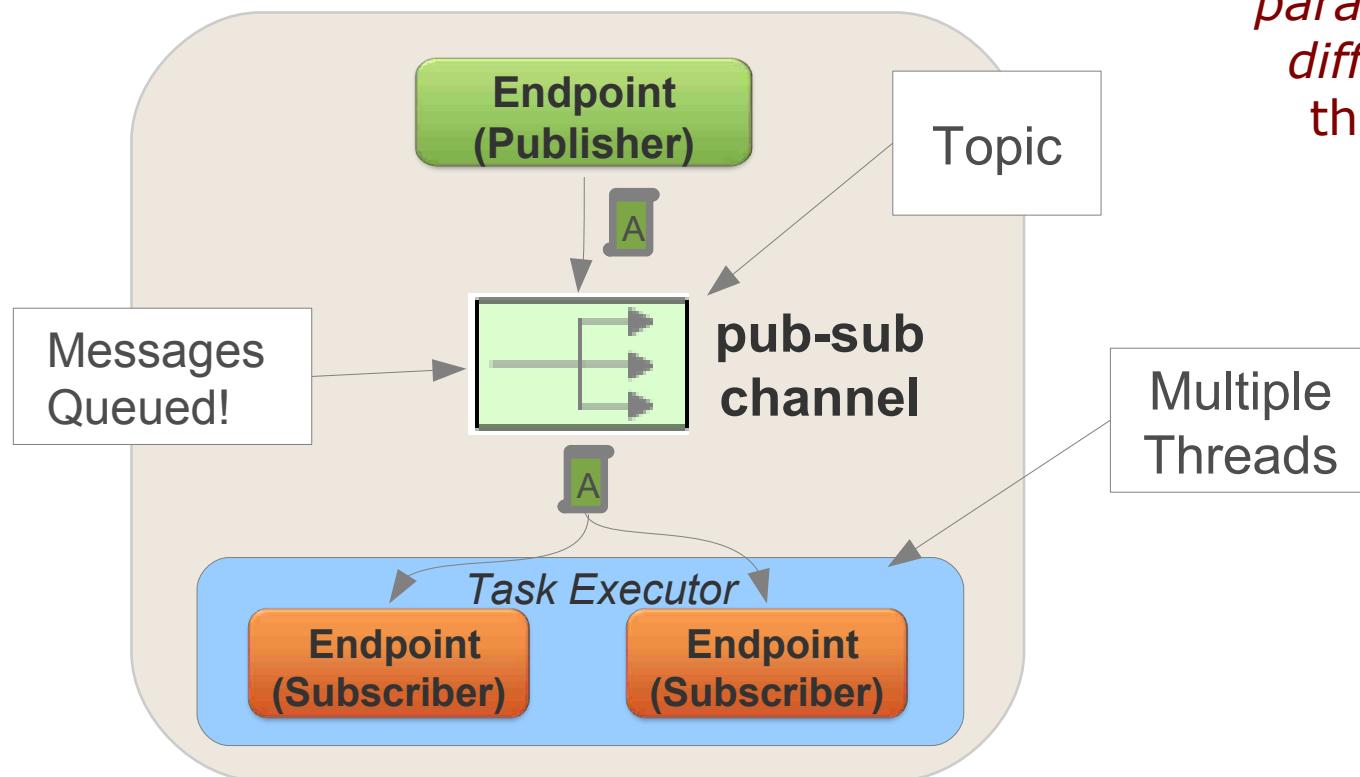
Publish-subscribe - 1

Endpoints invoked
invoked
serially in
same thread



MessageChannel Types

Publish-subscribe - 2



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

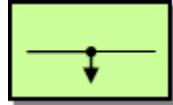
Channel Interceptors

- 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"/>
```

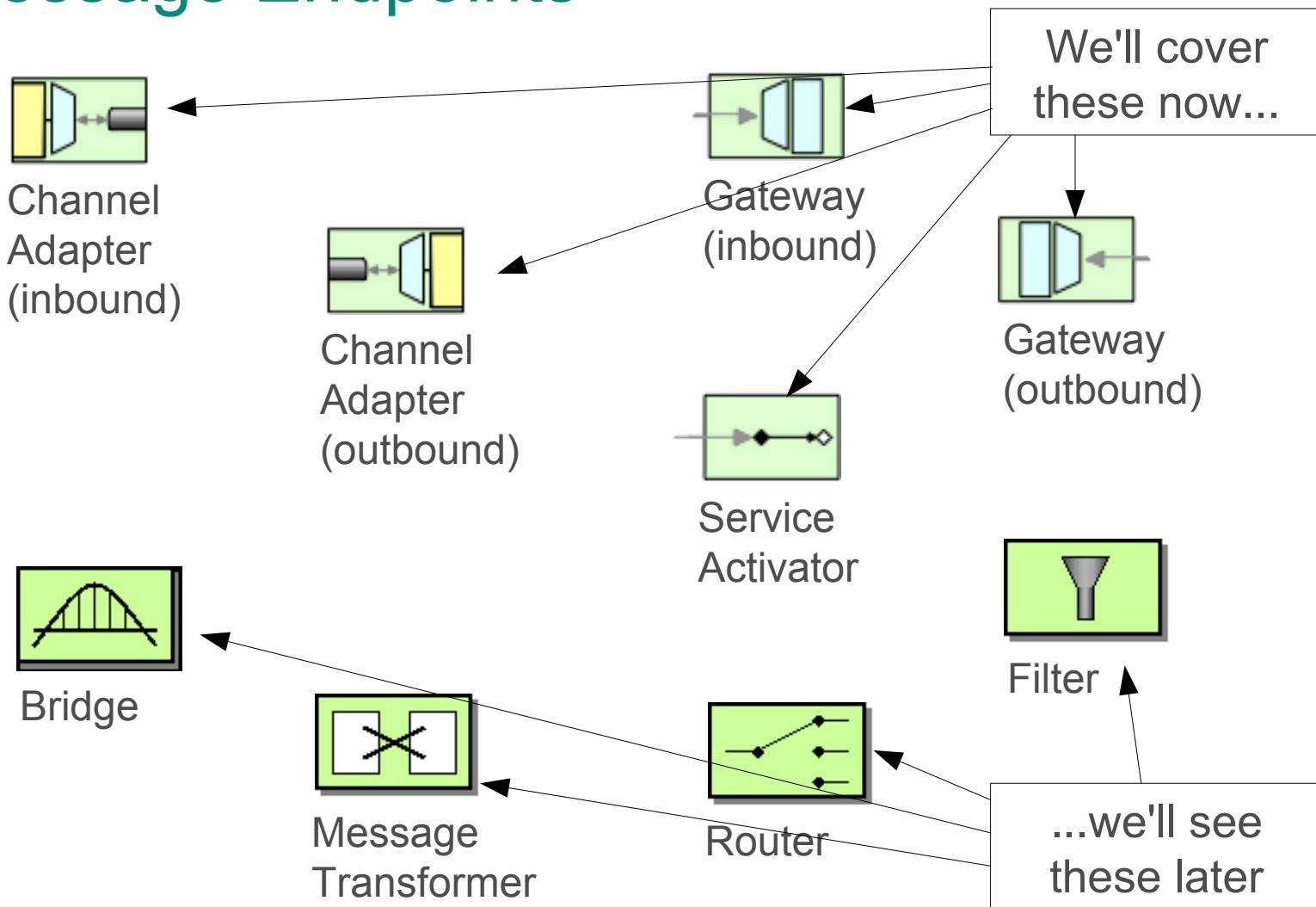
```
<logging-channel-adapter channel="logger" level="DEBUG"
```

not just payload

log-full-message="true"/>

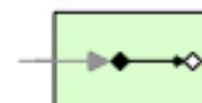
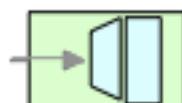
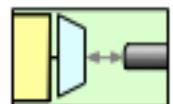


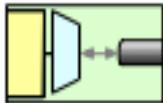
Message Endpoints



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





Channel Adapters

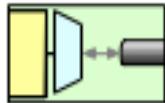
- Typically used to accept external input or send output
 - *Example:* incoming HTTP request

```
<int-http:inbound-channel-adapter id="httpAdapter"
    channel="updates" supported-methods="PUT,DELETE" />
```

- *Example:* Send to a JMS queue

```
<int-jms:outbound-channel-adapter channel="new-orders"
    destination-name="queue.orders"/>
```

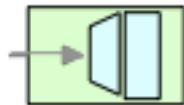
- Many, sources, sinks and protocols supported
 - Files, HTTP, JMS, Twitter ...



Event Generation

- Special case used for generating new messages
 - For testing
 - To process data periodically
- Needs a poller to activate it on a schedule

```
<int:inbound-channel-adapter id="generator"
    channel="events" ref="eventGenerator" method="generate" >
    <poller .../>
</int:inbound-channel-adapter>
```



Messaging Gateway

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

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

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

Java → 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)

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

Use annotations
for *per-method*
configuration



Service Activator

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

```
<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) {  
        ...  
    }  
}
```

Channel → Java

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

- Spring Integration – Goals
- **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)

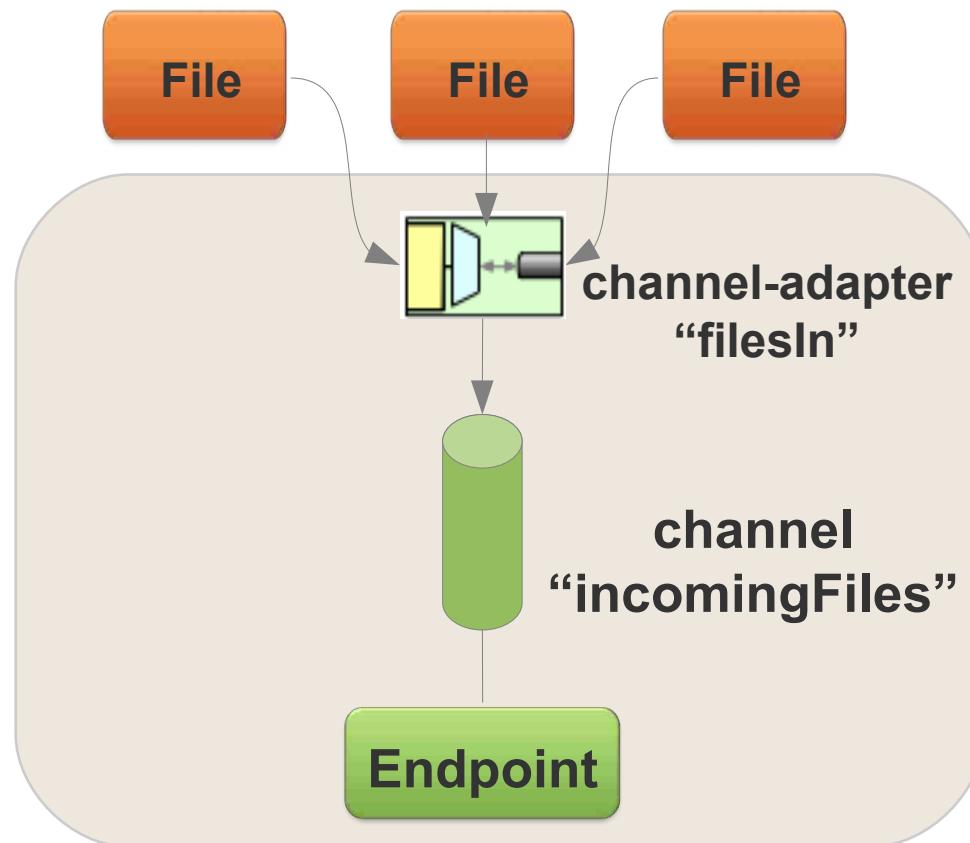
- Advice: only define explicit reply channel when you need a channel definition
 - i.e. to refer to the channel by name and/or to change its type from point-to-point to publish-subscribe
- Rely on default temporary reply channel otherwise

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`
 - ...

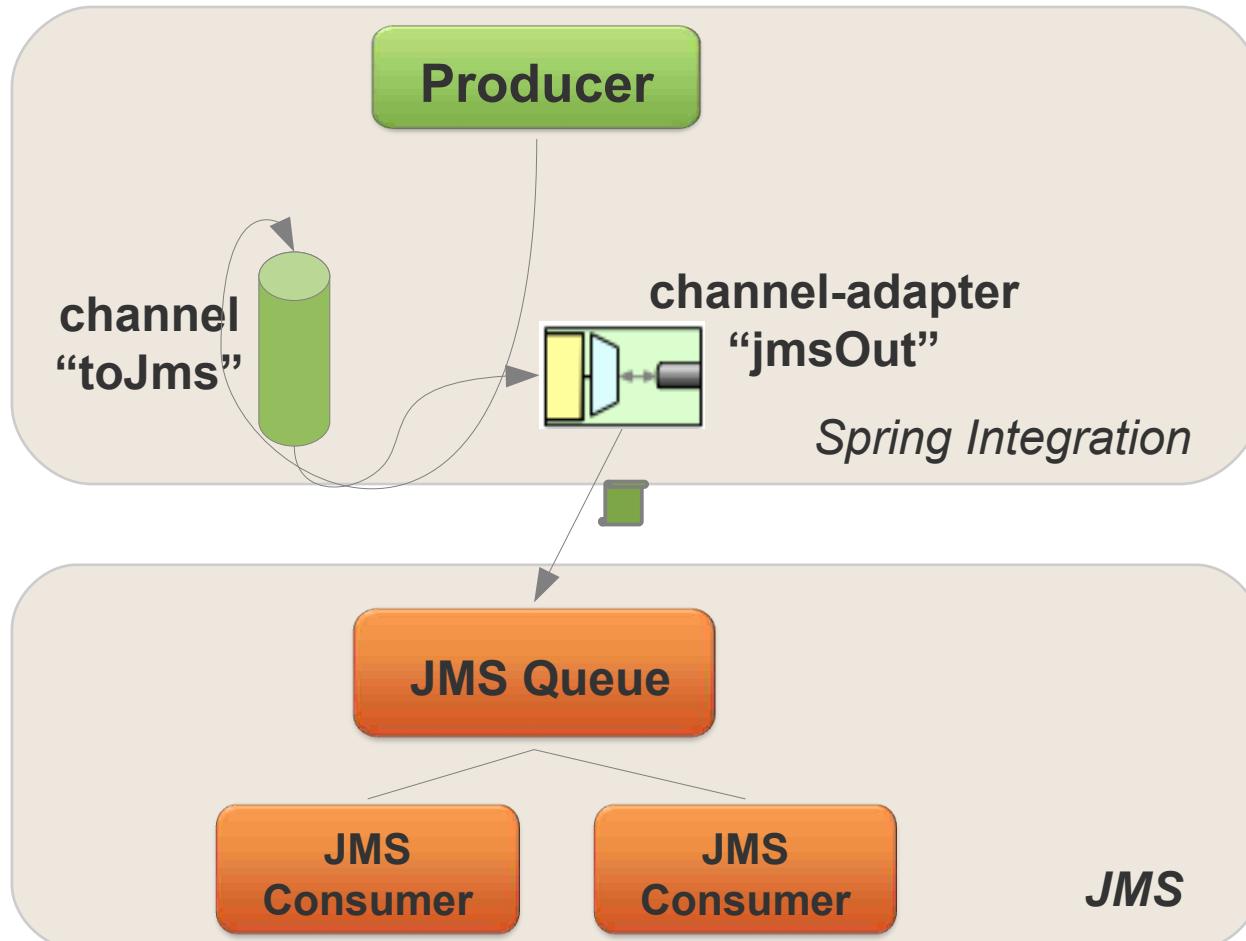
Sample: Inbound File Adapter

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



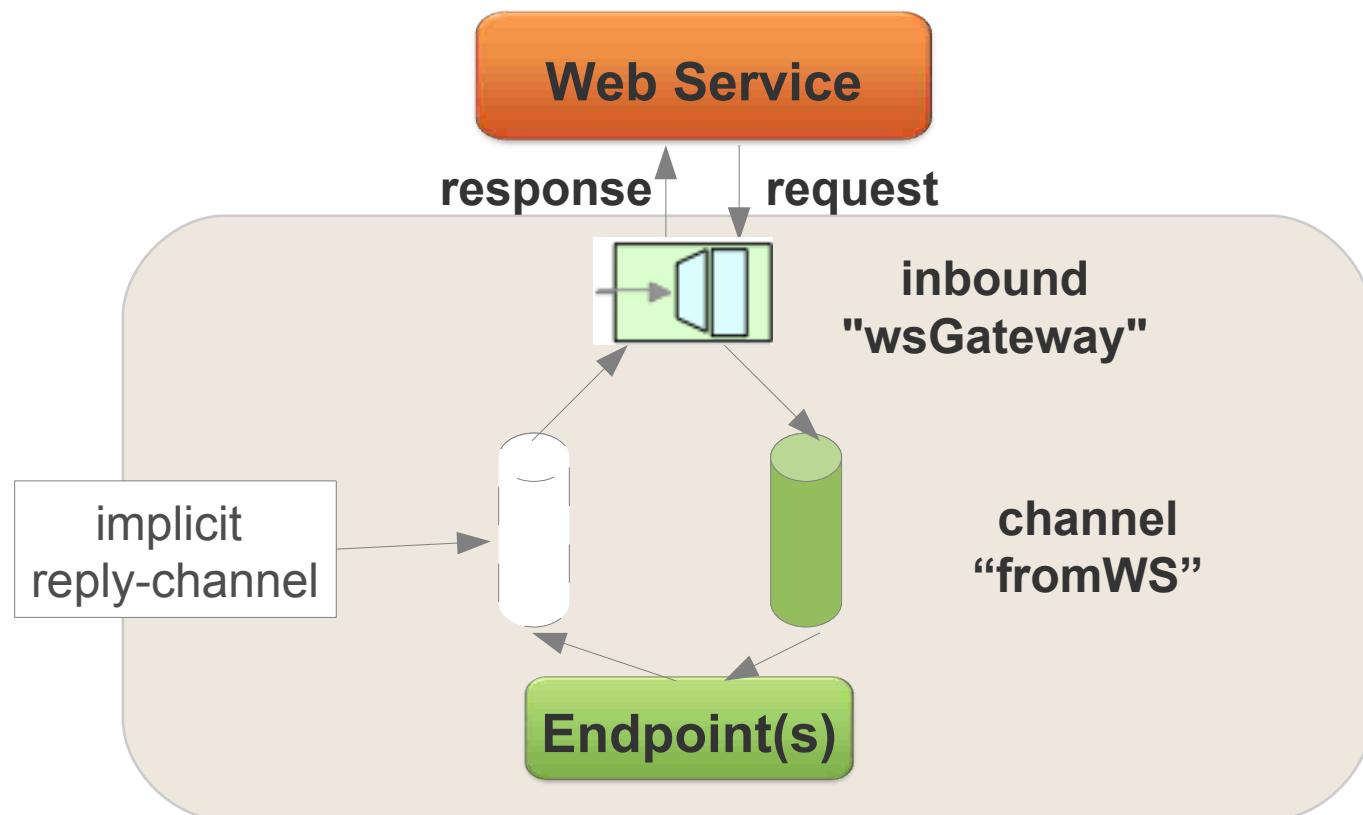
Sample: Outbound JMS Adapter

```
<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

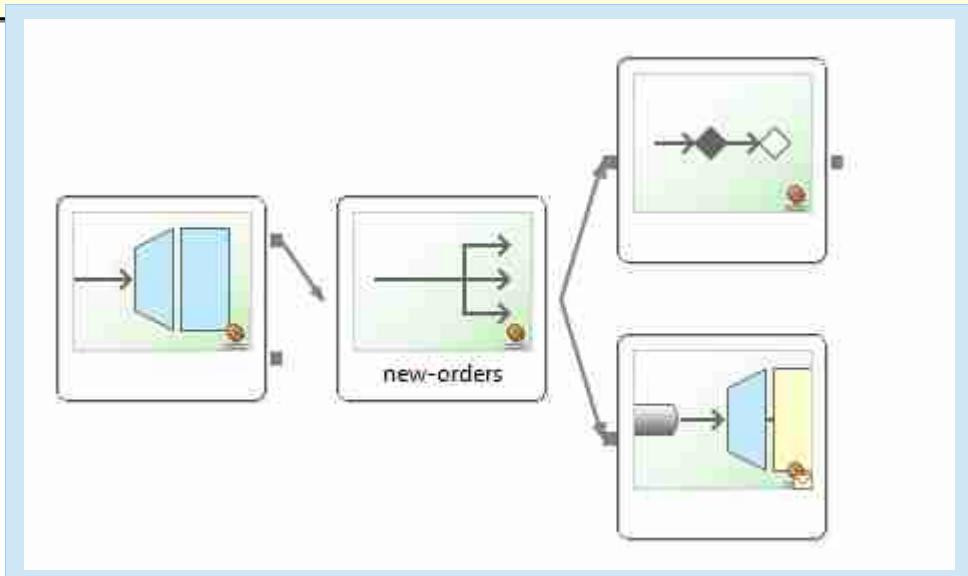
```
<gateway default-request-channel="new-orders"  
        service-interface="com.example.OrderService"/>
```

Send confirmation to
implicit reply-channel

```
<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 additional
JMS message

Sample: Combining Components

```
@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);  
        ...  
    }  
}
```

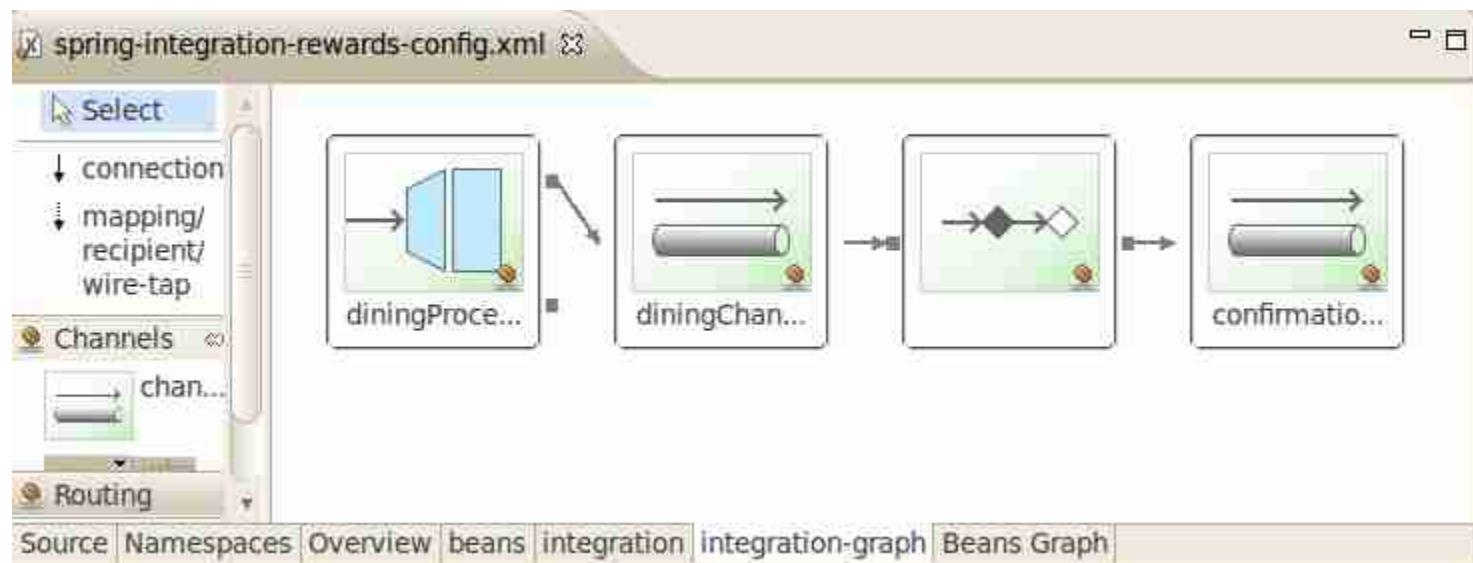
```
<gateway default-request-channel="new-orders"  
        service-interface="com.example.OrderService"/>  
    ...
```

Topics in this session

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

STS Visual Editor

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



Summary

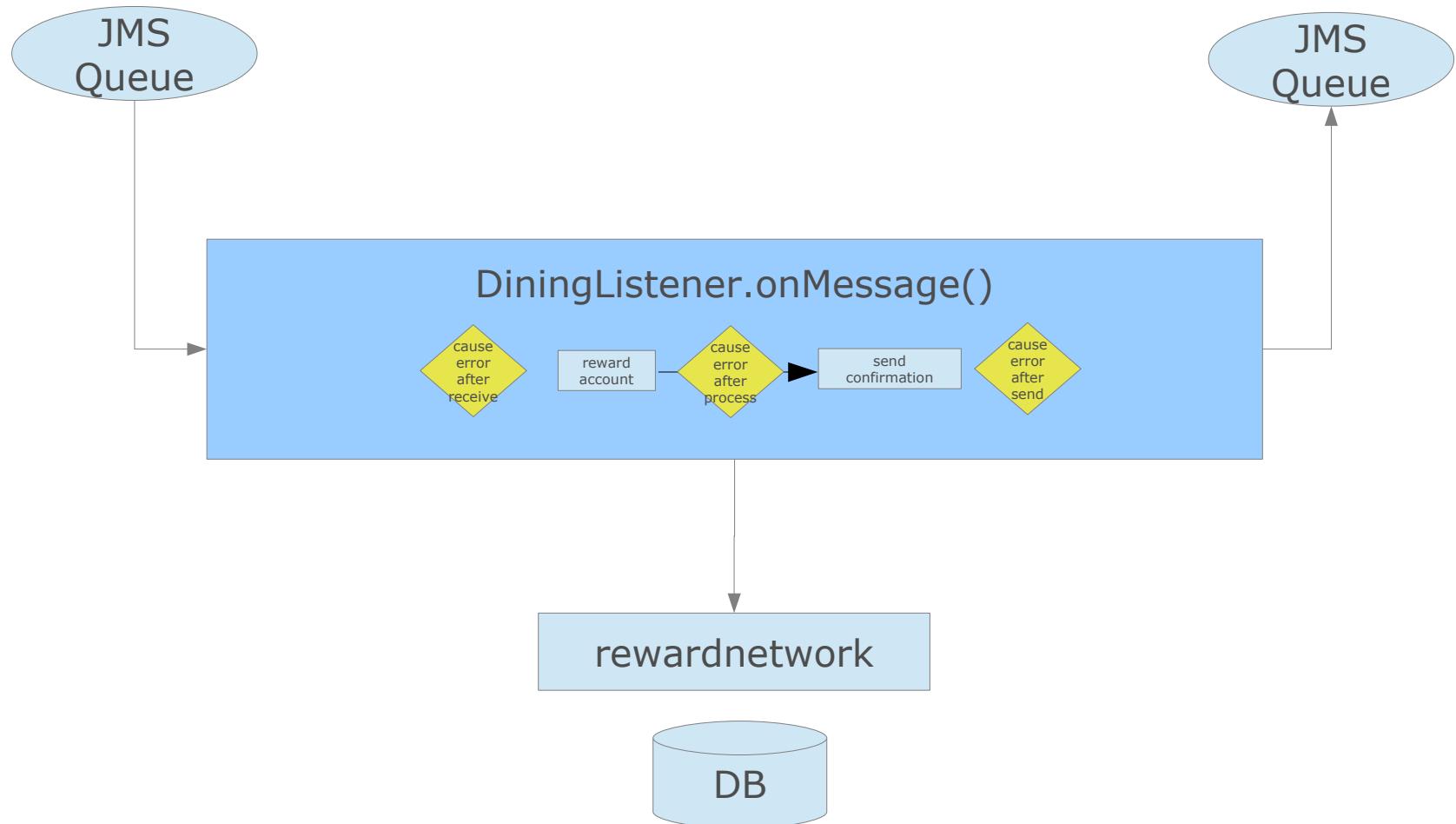
- Spring Integration provides the base components to implement EIP
 - To integrate application components
 - To integrate with external systems
- Allows for loosely coupled, event-driven architecture and separation of integration and processing logic

Lab

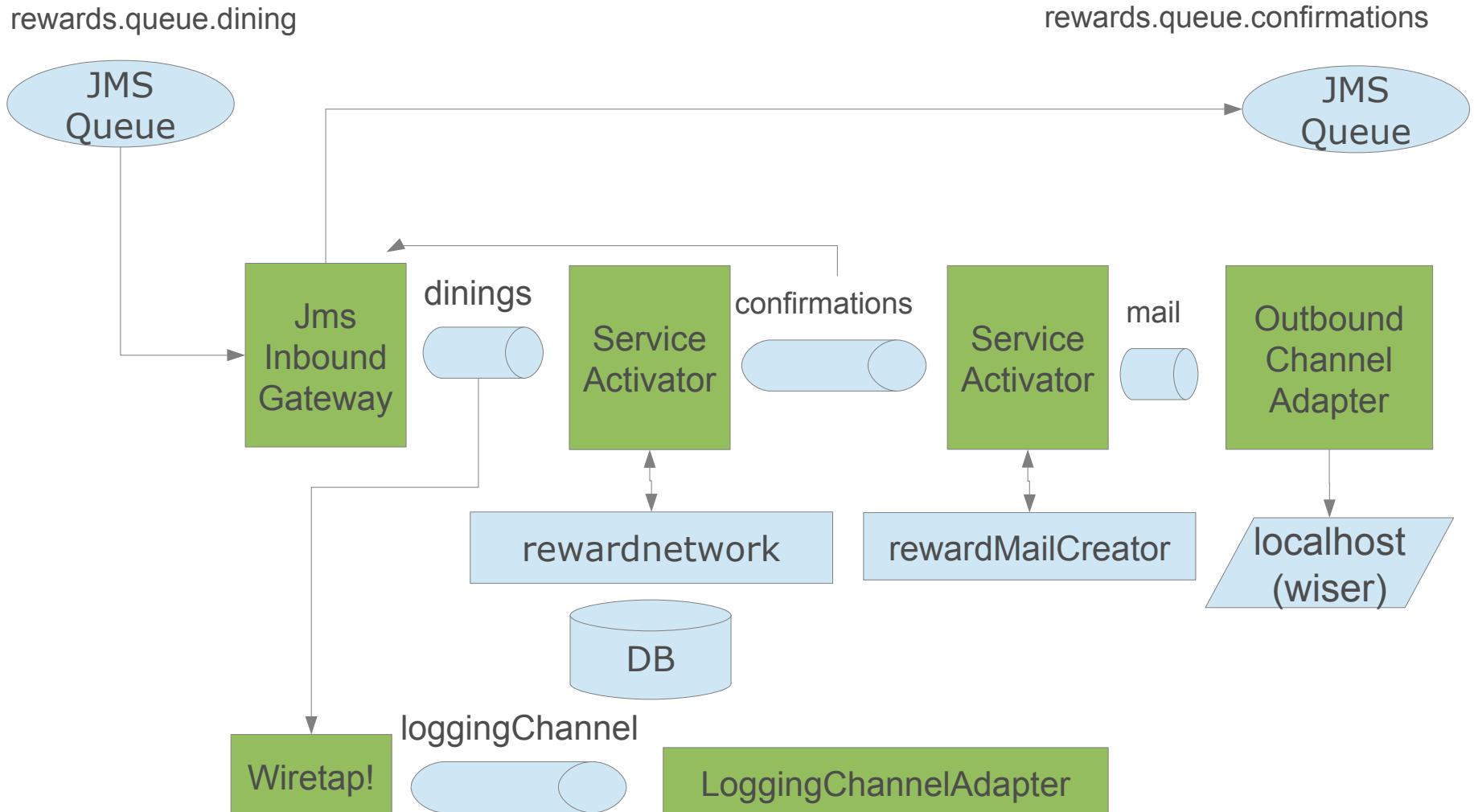
Previous Exercise

rewards.queue.dining

rewards.queue.confirmations



Converted to Spring Integration



Spring Integration Configuration Overview

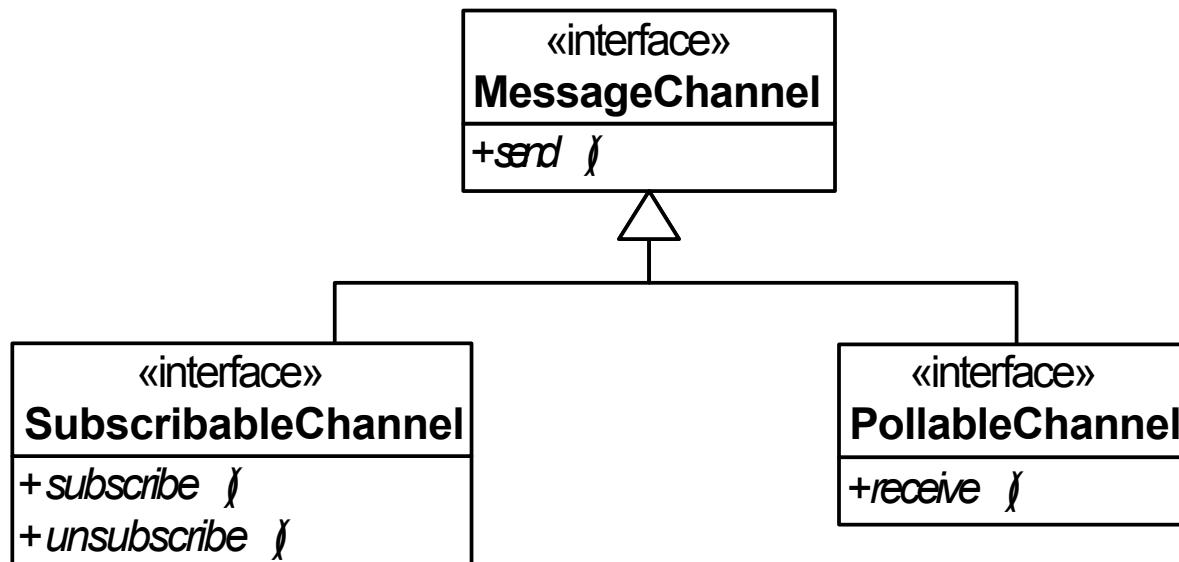
Configuring a working Pipes and Filters
architecture

Topics

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

Types of MessageChannels

- *SubscribableChannels* invoke subscribers on *send()*
- *PollableChannels* wait for *receive()* 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 FIFO buffering queue
 - Priority Channel priority-ordered
 - Rendezvous Channel zero-capacity, ensures handoff
 - NullChannel /dev/null

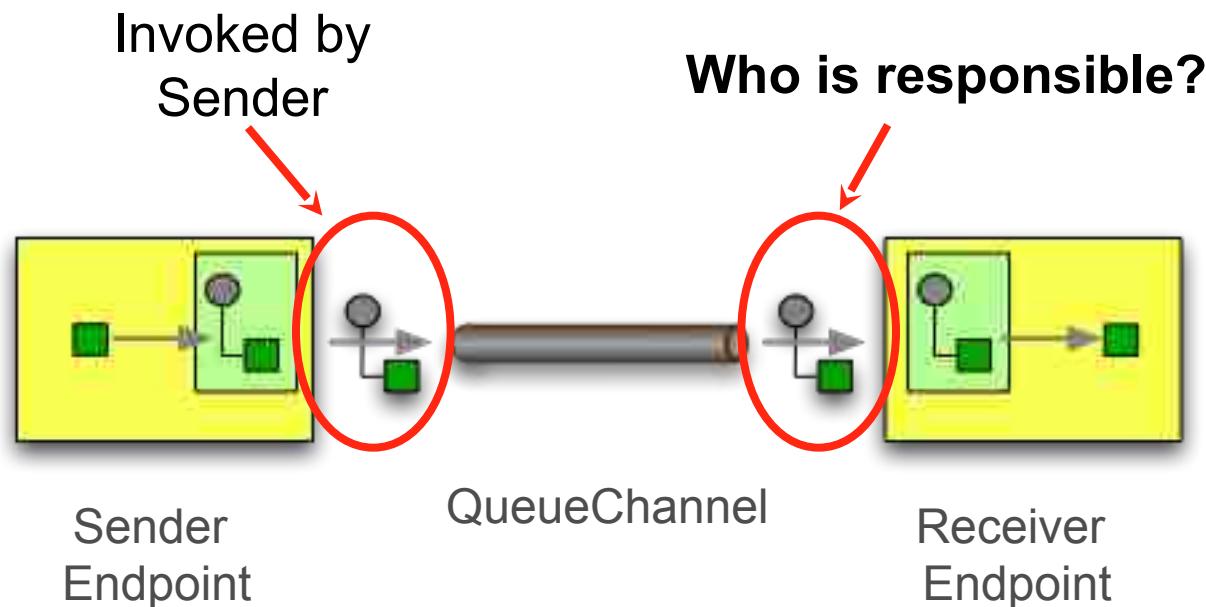


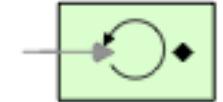
See [Spring Integration Reference – Message Channel Implementations](#)
for other kinds of channels

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"/>
```

milliseconds by default

```
<task:executor id="pool" pool-size="9" />
```

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
- Java DSL 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"/>
```

Asynchronous Handoff

- 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
- Java 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
```

Asynchronous Error Handling

- 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"/>
```

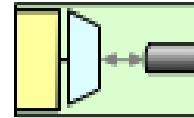
errorChannel

- 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

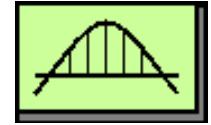
- Channel Types and Polling
- Synchronous vs. asynchronous handoff
- Error handling
- **More Endpoint types**
- Simplifying configuration
- Java DSL configuration



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>
```



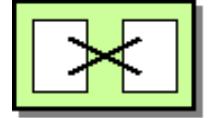
Bridge

- 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

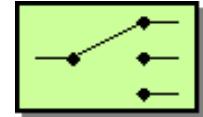
- Basically service activator with specific intent

```
<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!

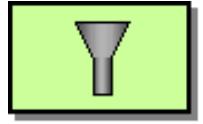


Router

- 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;
```

Dependency injection
by name – bean with
`id="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

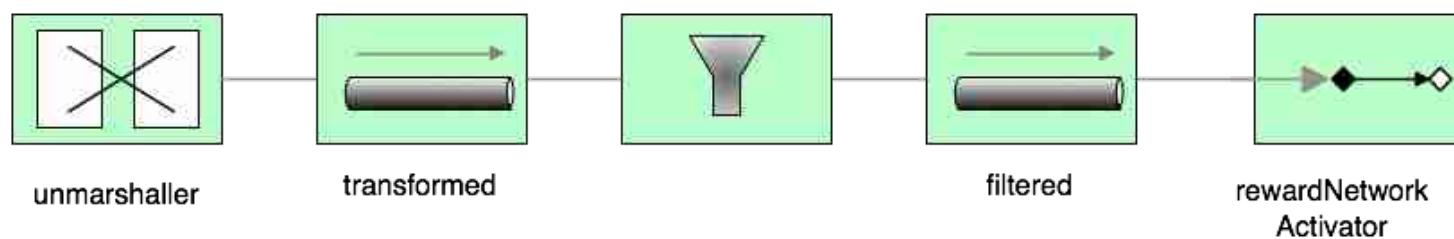
Topics

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

Topics

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

Java configuration with Spring Integration DSL

- Spring Integration has Java configuration support with an additional project **spring-integration-java-dsl**
- Consists of a Java-based facade with a fluent API
 - IntegrationFlows, MessageChannels, etc.
- Benefits
 - Type-safety
 - XML IDE support not longer needed
 - Less verbose than XML
 - One language for all

Initializing Java DSL Configuration

- Initialize Spring Integration infrastructure

```
@EnableIntegration  
@IntegrationComponentScan  
@Configuration  
public class SpringIntegrationConfig {  
}
```

Initialize Spring integration for Java configuration

Discover specific Spring Integration components (e.g. @MessagingGateway)

- Add dependency **spring-integration-java-dsl**

```
<dependency>  
    <groupId>org.springframework.integration</groupId>  
    <artifactId>spring-integration-java-dsl</artifactId>  
</dependency>
```

Declaring a gateway (xml vs annotations)

```
<int:gateway service-interface="rewards.messaging.ConfirmationProcessor"  
           id="confirmationProcessor"  
           default-request-channel="confirmations" />
```

The diagram illustrates the mapping between XML configuration and Java annotations. A blue downward arrow points from the XML code in the top box to the Java code in the bottom box. A grey arrow points from the text "Detected by @IntegrationComponentScan" in a callout box to the "@MessagingGateway" annotation in the Java code.

```
@MessagingGateway  
public interface ConfirmationProcessor {  
  
    @Gateway(requestChannel = "confirmations")  
    void process(RewardConfirmation confirmation);  
}
```

Declaring a channel

```
@Bean  
public MessageChannel myDirectChannel() {  
    return MessageChannels.direct().get();  
}
```

Method name = Channel name

Creates the bean

Dsl Factory builder

Variant of channel

```
@Bean  
public MessageChannel myQueueChannel() {  
    return MessageChannels.queue( 10 ).get();  
}
```

Capacity (*unlimited if not specified*)

```
@Bean  
public MessageChannel myPublishSubscribeChannel() {  
    return MessageChannels.publishSubscribe().ignoreFailures(true).get();  
}
```

Custom settings

Declaring a simple Integration Flow

```
@Bean  
public IntegrationFlow myFlow() {  
    return IntegrationFlows  
        .from(myDirectChannel())  
        .handle(...)  
        .channel(myQueueChannel())  
        .get();  
}
```

Input channel bean

Dsl Factory builder

Service Activator
(see next slide)

Output channel bean

Declaring a service activator

```
@Bean
```

```
public IntegrationFlow myFlow() {
```

```
    return IntegrationFlows
```

```
        .from(myDirectChannel())
```

```
Payload type
```

```
.<Dining>handle((dining,headers) ->  
    rewardNetwork.rewardAccountFor(dining))
```

```
Payload
```

```
Injected Spring bean
```

```
.channel(myQueueChannel())
```

```
Business logic
```

```
.get();
```

```
}
```

Declaring transformers and filters

```
@Bean
```

```
public IntegrationFlow myFlowWithFilter() {  
    return IntegrationFlows
```

```
        ...  
        .transform(Long.class, diningId -> loadDining(diningId))
```

Transform Java 8 lambda

```
.filter(Dining.class, payload -> payload.getAmount() != null)
```

Filter with java 8 lambda

```
.filter(Dining.class, payload -> payload.getAmount() != null,  
       spec -> spec.throwExceptionOnRejection(true) )
```

```
...
```

```
.get();
```

```
}
```

Filter with custom behavior

Declaring a poller

```
@Bean(name = PollerMetadata.DEFAULT_POLLER)  
public PollerMetadata poller() {  
    return Pollers.fixedDelay( 250 )  
        .get();  
}
```

The diagram illustrates the mapping of annotations in the Java code to their corresponding configuration elements. The code uses the `@Bean` annotation from Spring, the `PollerMetadata` class from the Quartz library, and the `Pollers` factory from the Spring Integration library.

- An annotation `@Bean` is annotated with the value `name = PollerMetadata.DEFAULT_POLLER`. A callout box labeled "Default poller name" points to this value.
- The code uses the `PollerMetadata` class. A callout box labeled "Delay in ms" points to the argument of the `fixedDelay` method.
- The code uses the `Pollers` factory. A callout box labeled "Dsl Factory builder" points to the type of the `poller` method.

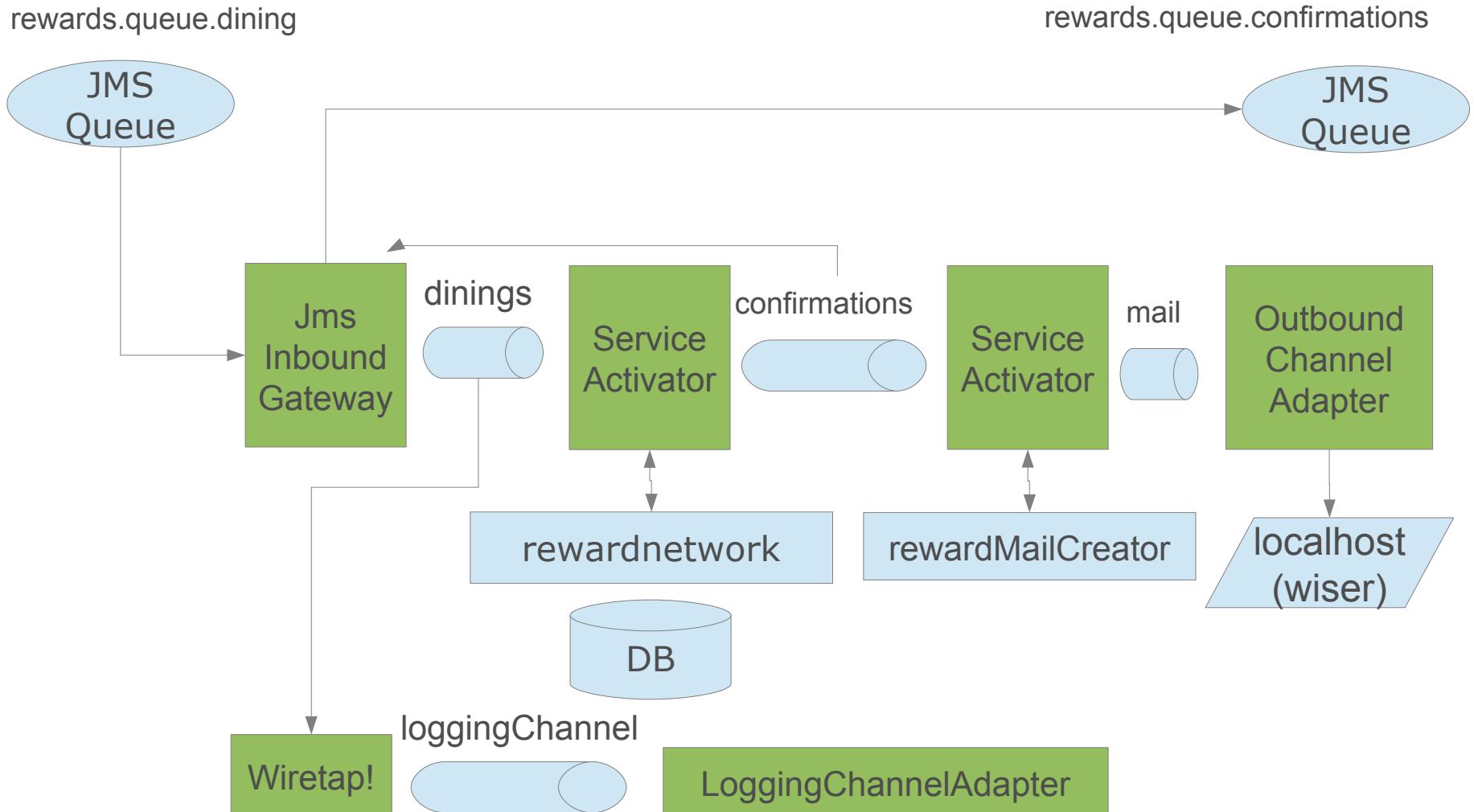
Summary

- 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
- Configuration with a java configuration

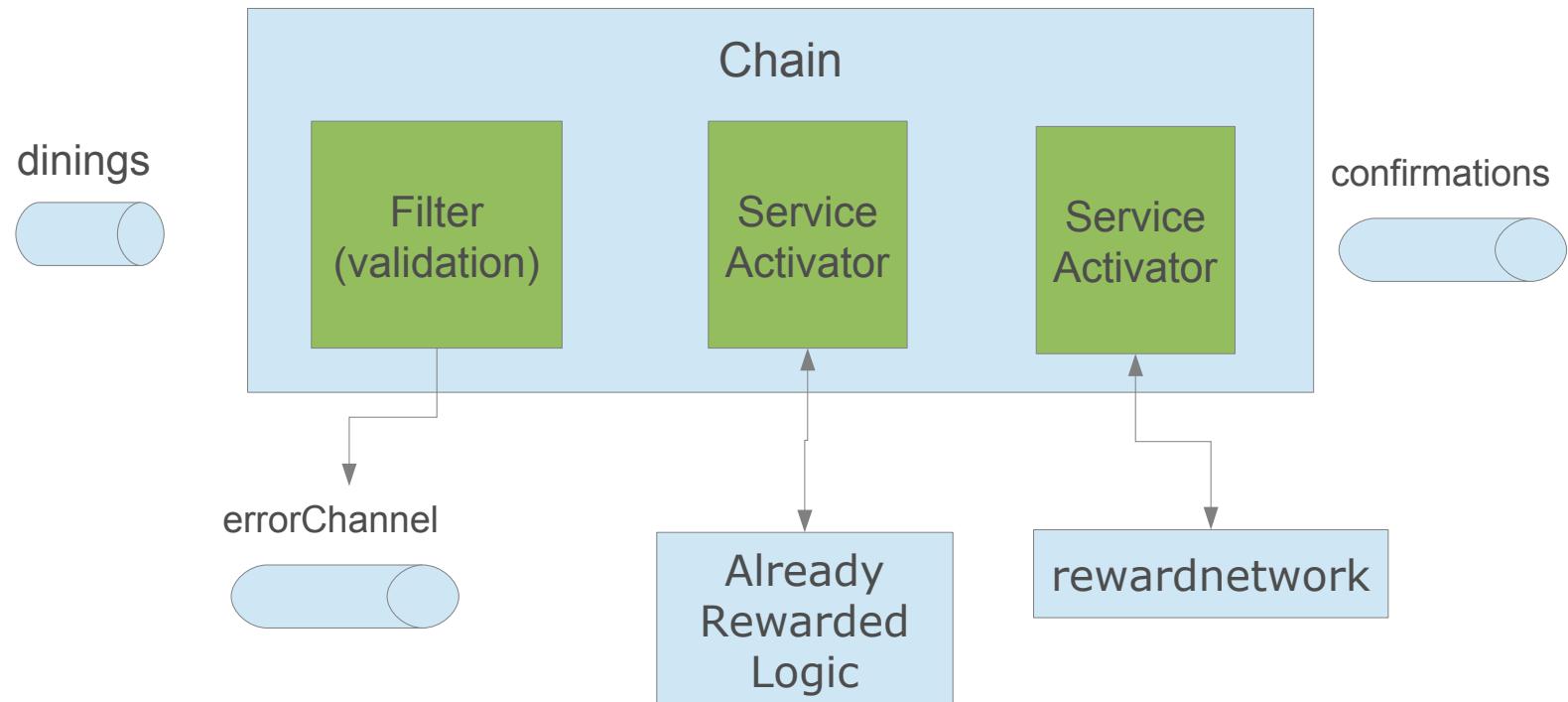
Lab

Optimize the Messaging application
using an Idempotent Receiver

Converted to Spring Integration



Previous Exercise



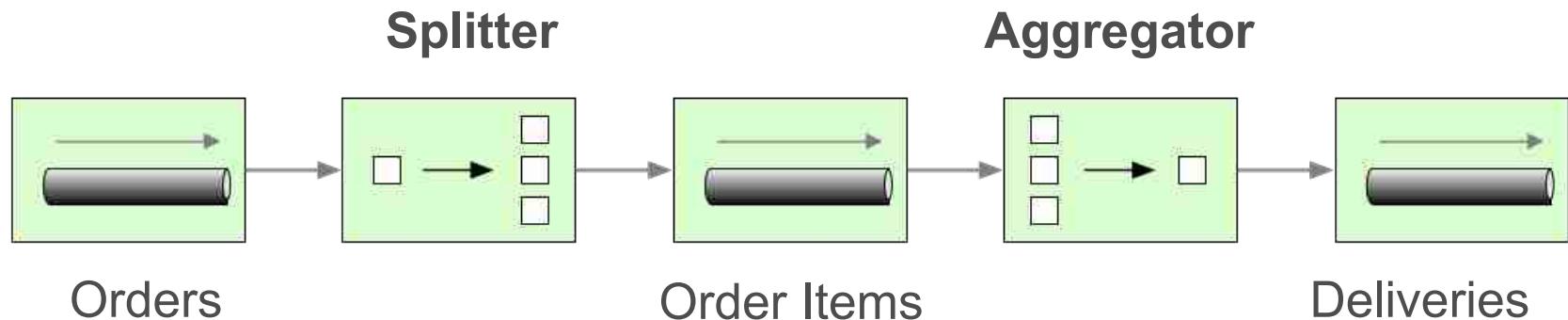
Spring Integration Advanced Features - 1

Splitting & aggregating, dispatching
and XML support

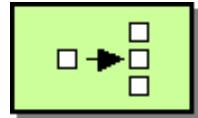
Topics

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

Splitting and Aggregating



Splitter

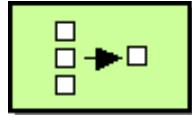


- Input: a single message
- Output: multiple messages
- Splitting strategy should be provided in a dedicated method

```
@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" ... />
```

Correlation and Release

- 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?

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" ... />
```

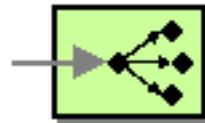
Splitter and Aggregator

- 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**

XML Support – Overview

- 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>
```

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"           → javax.xml.transform.Templates
    result-transformer="resultTransformerBean"/>
```

OXM Support

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

ResultTransformer

- 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

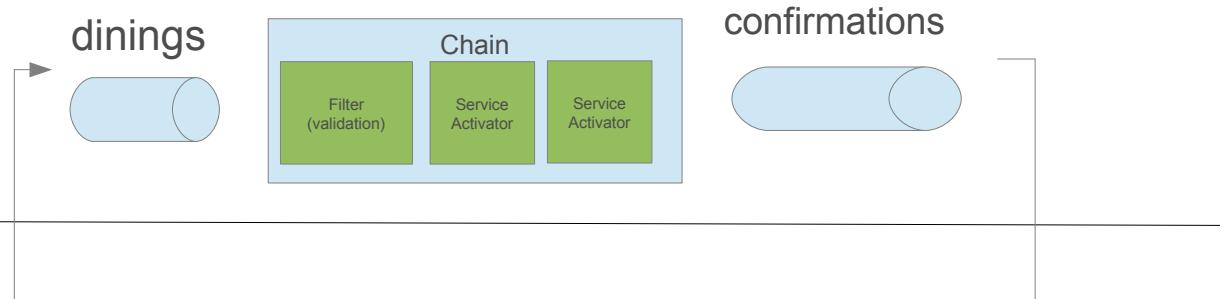
- Splitting and Aggregating
 - symmetric, customizable
- Configurable dispatching for point-to-point subscribable channels
- XPath, XSLT and OXM support for dealing with XML

Lab

XML Transformation and Splitting

Exercise

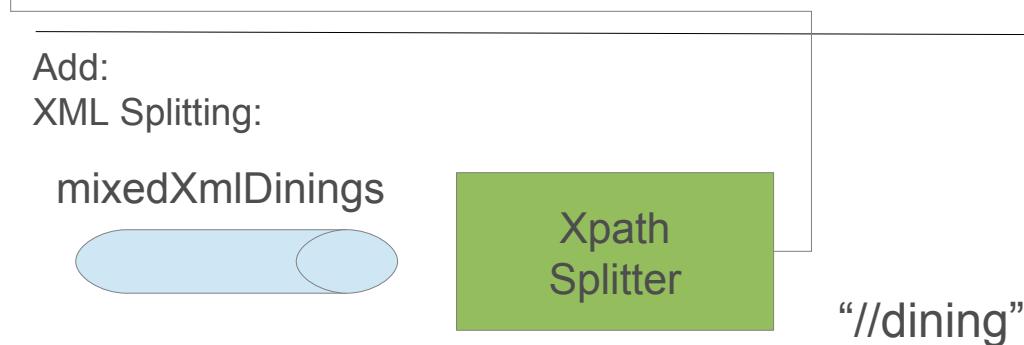
Previous Exercise:
Idempotent Receiver



Add:
XML Transform:



Add:
XML Splitting:

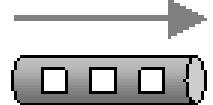


Spring Integration Advanced Features - 2

Datatype Channels,
Persisting Messages,
Control Bus and Message History

Topics

- **Datatype Channels**
- Persisting Messages
- Control Bus
- Message History



Datatype Channels

- By default channels can hold any type of Message
- Can optionally be typed
 - Restricts acceptable messages to those with payload of the given type(s)
 - Attempts conversion for messages of different type

```
<channel id="dinings" datatype="rewards.Dining"/>
```

```
<channel id="wholeNumberChannel"  
datatype="java.lang.Integer,  
java.math.BigInteger,  
java.util.concurrent.atomic.AtomicInteger"/>
```

Message Conversion

- SI uses **integrationConversionService** bean (named `ConversionService`) when defined
 - Must have suitable Converter

```
<bean id="integrationConversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
      <bean class="com.example.StringToFooConverter"/>
    </property>
</bean>
```

must have this name

Allows sending Message<String> to channel of type Foo

```
public interface Converter<S, T> {
    T convert(S source);
}
```

Spring 3.0 introduced
Converter interface

Integration ConversionService

- `ConversionServiceFactoryBean` registers many default Converters
 - See `ConversionServiceFactory.addDefaultConverters()`
- Shortcut to ensure `integrationConversionService` exists and add a custom converter to it
 - Multiple converter elements can be used

```
<int:converter>
  <bean class="com.example.StringToFooConverter"/>
</int:converter>
```

'ref' attribute also supported



Rely on `ObjectToObjectConverter` conventions to prevent having to code dedicated Converters for your classes

Topics

- Datatype Channels
- **Persisting Messages**
- Control Bus
- Message History

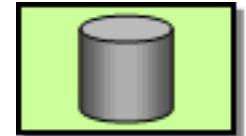
Persistent Messages

- QueueChannels keep messages in memory by default
- As do stateful endpoints like aggregators
- Custom Message Store or channel types allow persisting messages in a database or message broker

Persistent Messages

Use Cases

- Preventing message loss on application shutdown
- Transactional message processing
- Dealing with large message payloads
- Using pub-sub facilities of message broker
 - e.g. to address all nodes in a cluster
- All internal to application
 - Entire Message stored, not just payload
 - Use adapters and gateways for integration with other applications through databases and brokers



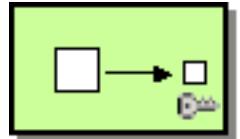
JDBC Message Store

- JdbcMessageStore stores messages in relational DB
 - DB schema ships with spring-integration-jdbc jar
 - (De)serializes messages to single column
 - Uses Java Serialization by default, so ensure headers and payload implement Serializable!

```
<int-jdbc:message-store id="messageStore" data-source="dataSource" />

<channel id="persistentChannel">
    <queue message-store="messageStore"/>
</channel>

<aggregator id="longLivedAggregator" message-store="messageStore" ... />
```



Claim Check Pattern

- Deals with large payloads using Message Store
 - Payload stored using identifier (UUID)
 - SI payload set to identifier
 - Identifier acts as 'claim check' to query Store for real payload when needed
- Supported using pair of transformers

```
<claim-check-in id="claim-in" message-store="jdbcMessageStore"  
    input-channel="checkin" output-channel="checkedin"/>  
  
<claim-check-out id="claim-out" message-store="jdbcMessageStore"  
    input-channel="checkout" output-channel="checkedout"  
    remove-message="true"/>
```

Claim only once: defaults to false

JMS Channels

- Channels can be backed by JMS Queue or Topic
 - For Queues, channel can be pollable or subscribable
 - Topics are always subscribable
- Uses regular Spring JMS support
 - Same configuration options, but defaults to local TXs
 - Defaults to ObjectMessages (SimpleMessageConverter)
 - So implement Serializable for headers and payload!

```
<int-jms:channel id="pollableQueueChannel" queue-name="si-queue"  
message-driven="false"/>
```

pollable (default is true) subscribable

```
<int-jms:publish-subscribe-channel id="pubsubChannel"  
topic-name="si-topic"/>
```

Topics

- Datatype Channels
- Persisting Messages
- **Control Bus**
- Message History



Control Bus

- Manage and monitor channels and endpoints by sending messages to a special component
 - Payloads are SpEL expressions
 - Access to any Spring bean in ApplicationContext
 - But only @ManagedOperation/-Attribute annotated methods
 - As well as Spring's Lifecycle start() & stop() and CustomizableThreadCreator subtype methods
 - Acts as dynamic service activator with built-in restrictions

```
<control-bus input-channel="operationChannel" />
```

optional output-channel for results

```
messagingTemplate.convertAndSend("operationChannel",  
                                "@jmsListenerContainer.stop()");
```

Groovy Control Bus

- Groovy instead of SpEL supported as well
 - Auto-exposes beans annotated with `@ManagedResource` and instances of `LifeCycle` or `CustomizableThreadCreator`

```
<int-groovy:control-bus input-channel="operationChannel" />
```

```
messagingTemplate.convertAndSend("operationChannel",
                                  "jmsListenerContainer.stop()");
```

Topics

- Datatype Channels
- Persisting Messages
- Control Bus
- **Message History**

Message History

- Track components a message passes through
 - For auditing, debugging, profiling, etc.
- Enabled using single tag: <message-history />
- Adds List<Properties> 'history' header
 - With 'name', 'type' and 'timestamp' Properties keys for each channel or endpoint
 - Defined as String constants of MessageHistory class
- Optionally limit tracked components by name

```
<message-history tracked-components="*Gateway, sample*, foo" />
```

Message History Sample

- Log history of every message for given channel:

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

<message-history />

<channel-interceptor pattern="someChannel">
    <wire-tap channel="historyLogChannel"/>
</channel-interceptor>

<logging-channel-adapter id="historyLogChannel"
    expression="payload + ':' + headers.history.[name + ' - ' + timestamp]"/>
```

Collection projection

```
messagingTemplate.convertAndSend("someChannel", "hello");
```

```
INFO : org.springframework.integration.handler.LoggingHandler -
hello: [someChannel - 1312462840910, historyLogChannel - 1312462840915,
historyLogChannel.adapter - 1312462840916]
```

Summary

- Spring Integration supports many advanced Enterprise Integration Patterns out of the box
 - Datatype Channels, Message Store, Claim Check, Control Bus, Message History
- Adding support requires no code changes
 - But keep serialization requirements in mind

Introducing Spring Batch

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

Batch Domain

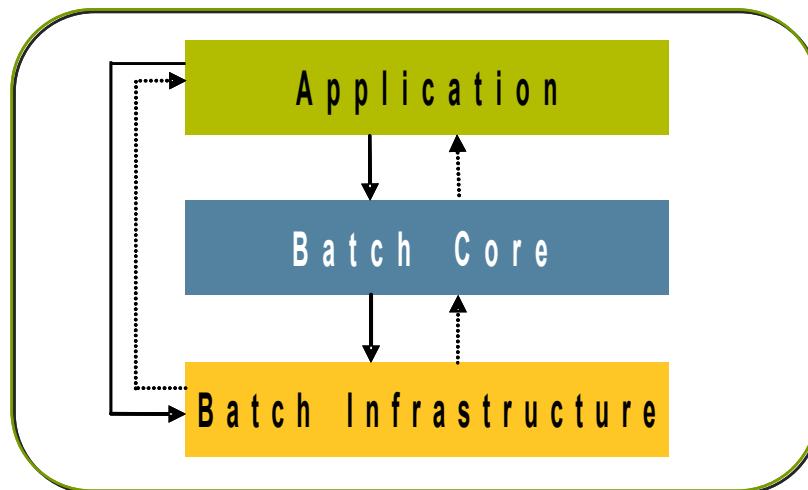
- The batch domain adds some value to a plain business process by introducing new concepts:
 - A **job** describes a series of **steps** to be done.
 - A **step** is an independent unit of processing
 - A **job instance** can be **restarted** after a failure – a new **job execution**
 - Each **job execution** has a **start** time, **stop** time, **status**
 - The **job instance's status** = the status of its latest **job execution**
 - Each **execution** can tell us how many **items** were processed, how many **commits**, **rollbacks**, **skips**

Topics in this Session

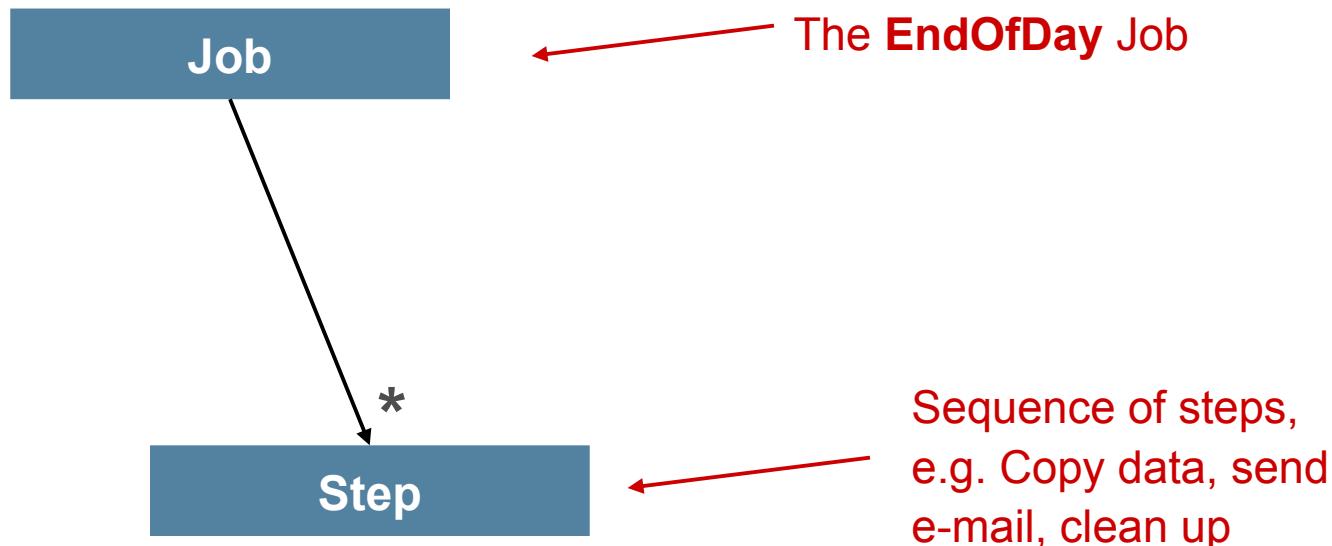
- Batch and offline processing
- **Spring Batch high-level overview**
- Job parameters and job identity
- Quick start using Spring Batch
- Readers, Writers & Processors
- JDBC Item Readers

Spring Batch Overview

- The Spring programming model applied to batch processing
 - Don't write code that doesn't make you money
- Project page: <http://projects.spring.io/spring-batch/>
- Implementation of JSR-352

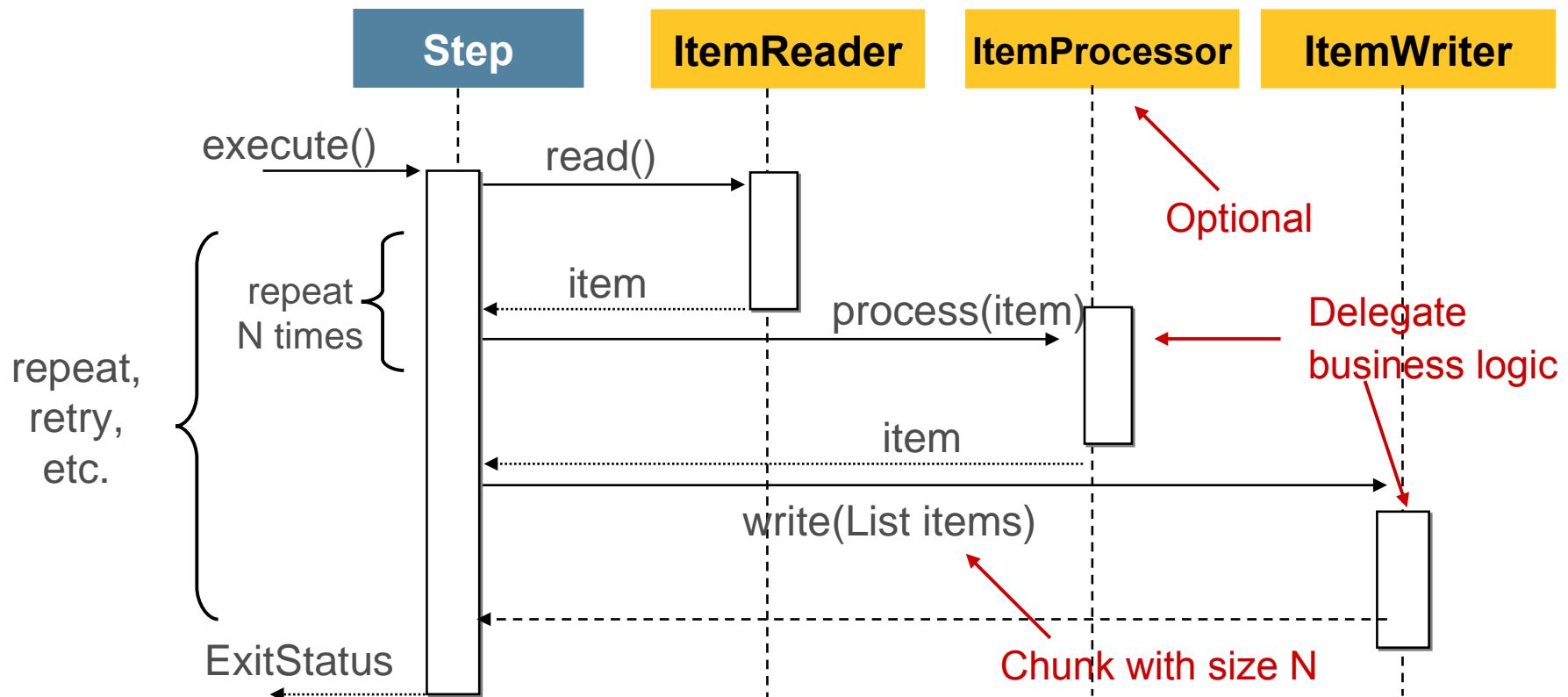


Job and Step

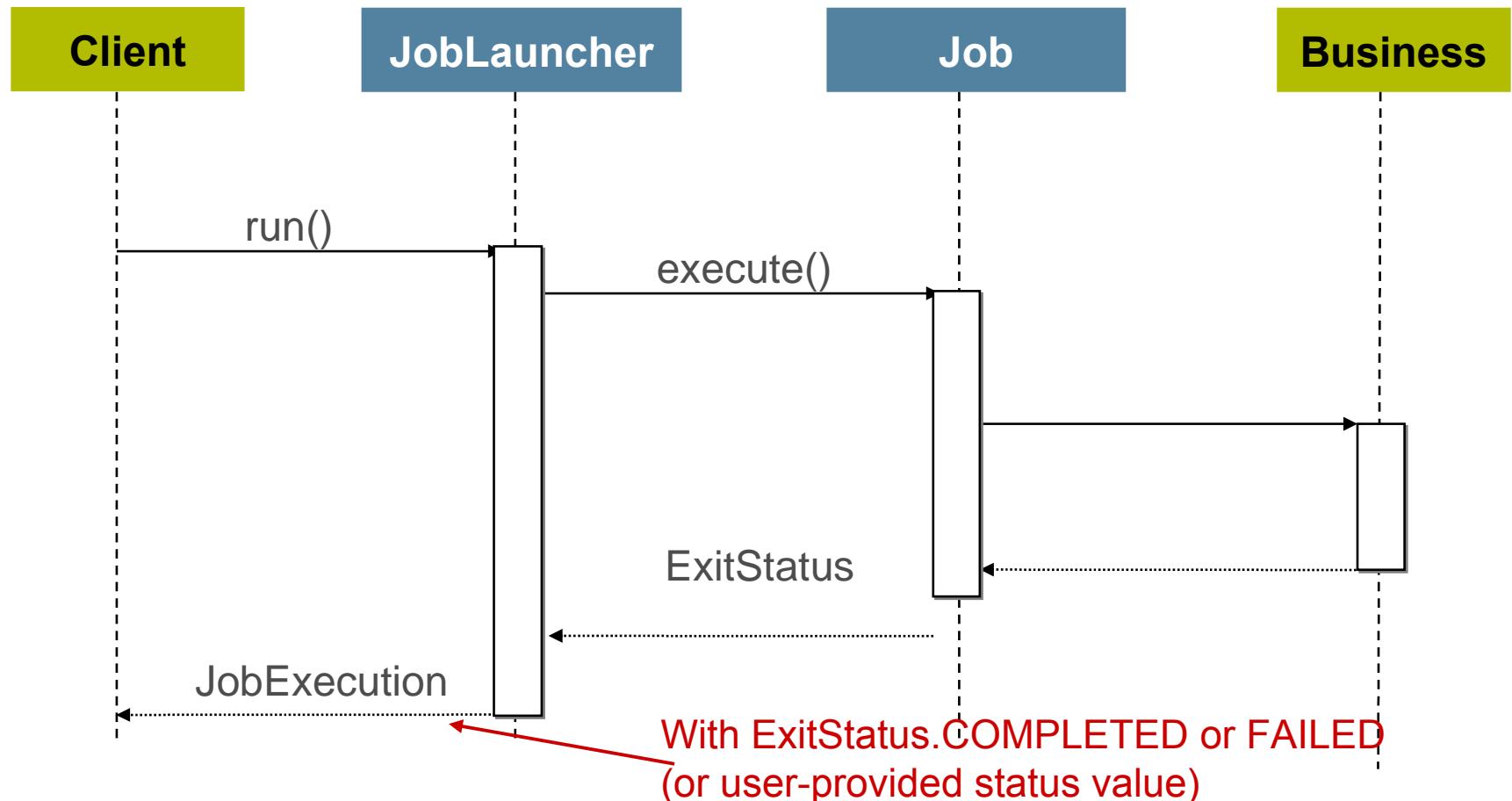


Chunk-Oriented Processing

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



JobLauncher



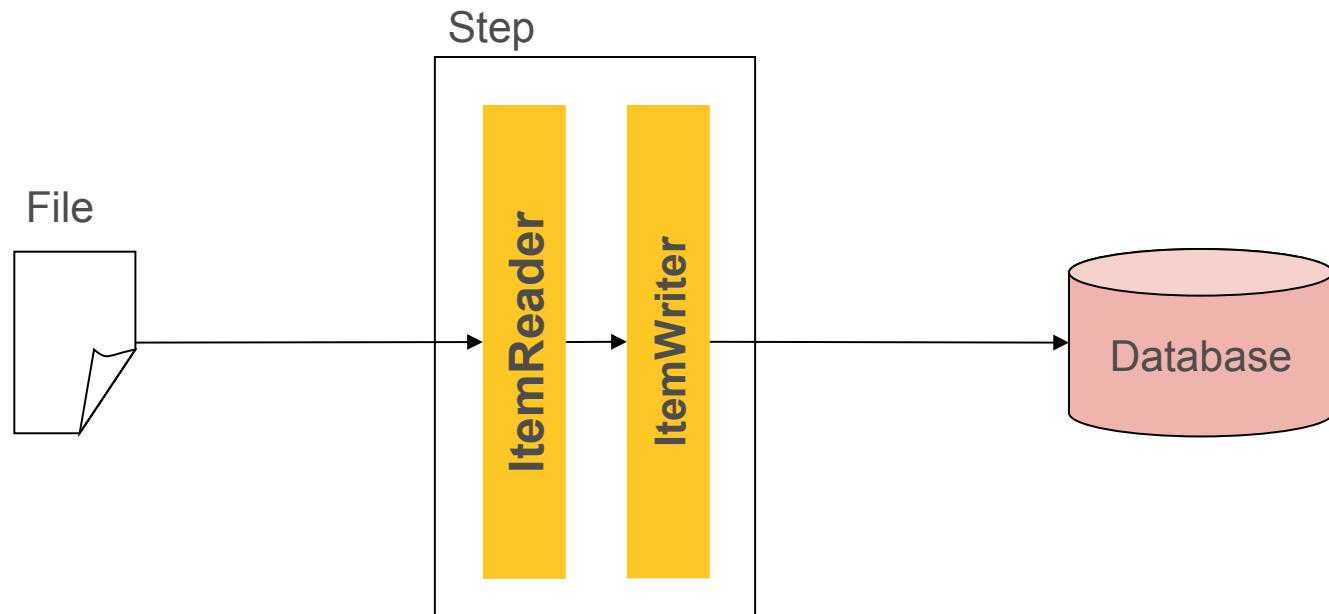
Topics in this Session

- 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

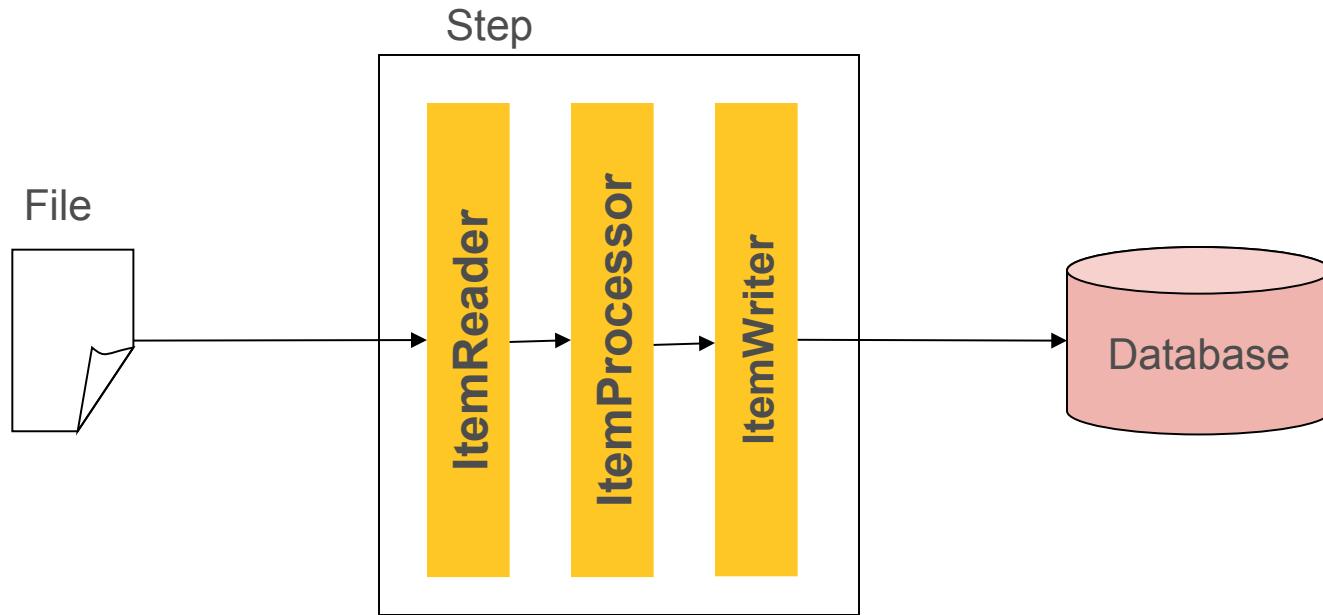
Readers and Writers

- 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 parameterized, so e.g.

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

ItemProcessor

- The interface is parameterized
 - 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 parameterized; 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  
    }  
    ...  
}
```

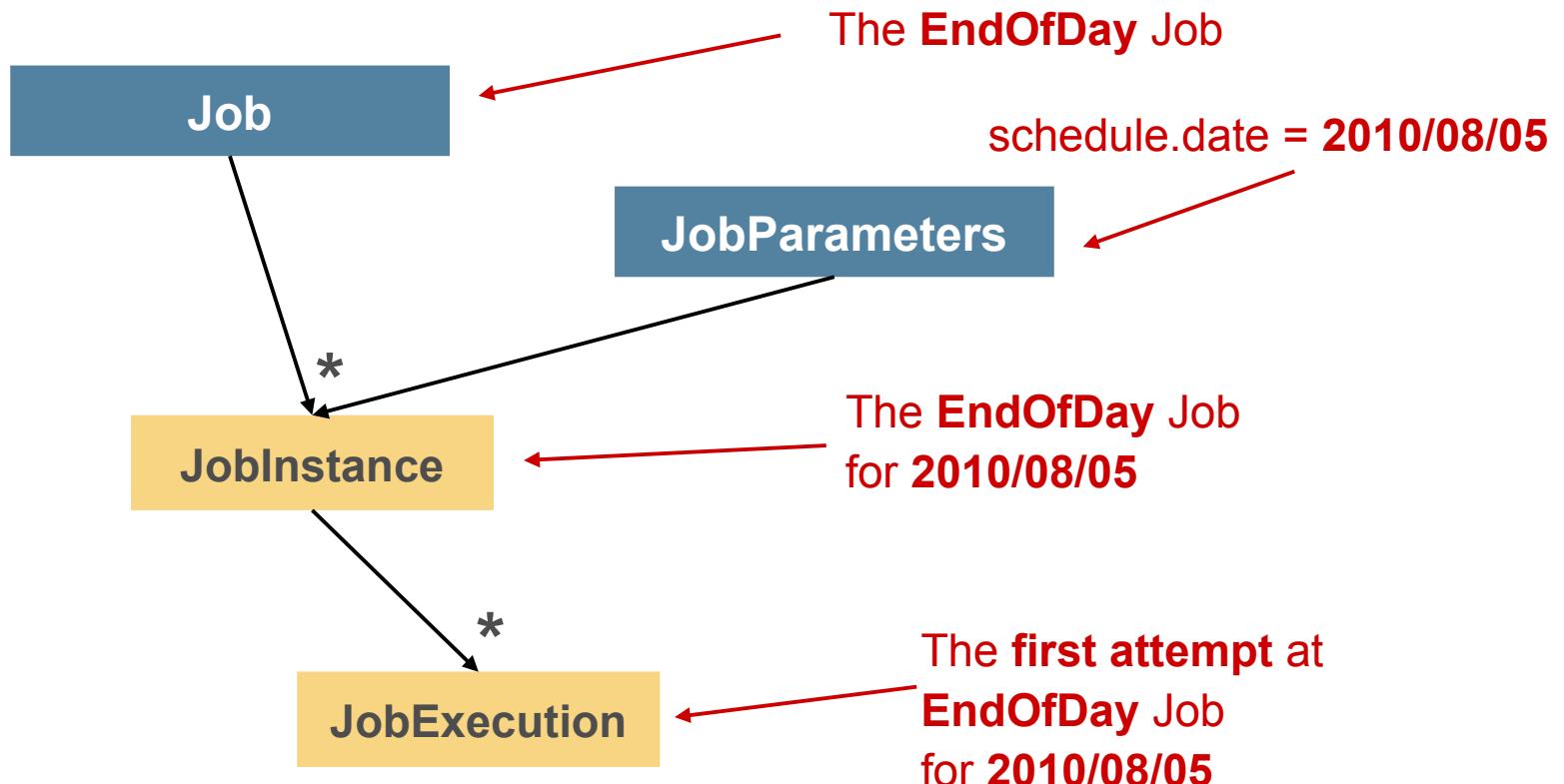
Topics in this Session

- 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



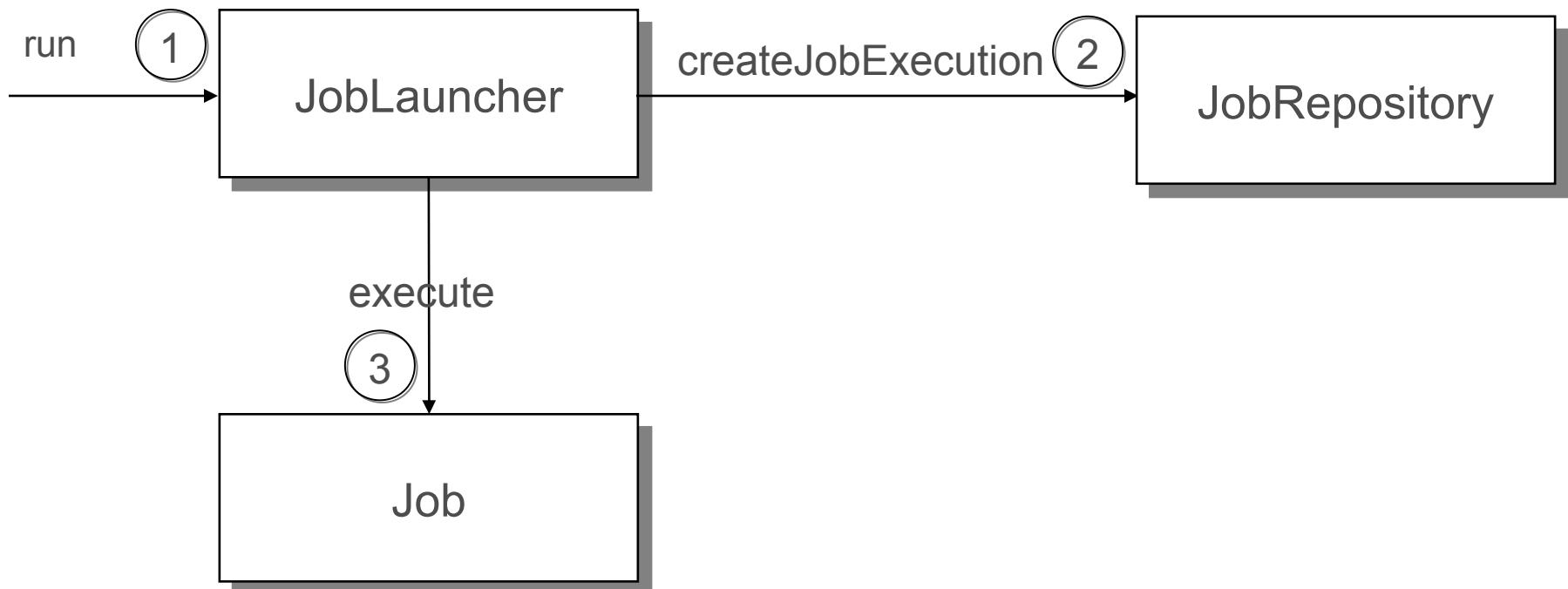
Job Parameters

- Provide uniqueness for each Job Instance
 - JobInstanceAlreadyCompletedException
 - Often a natural unique parameter such as run Date/Time
 - Can use a JobParametersIncrementer to create uniqueness
- Framework allows for non-identifying parameters
 - ...Such as folder to read files from

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 usually stored in Database
 - Use in-memory DB if no need for permanent storage.
- Core interface = JobRepository

JobLauncher and JobRepository

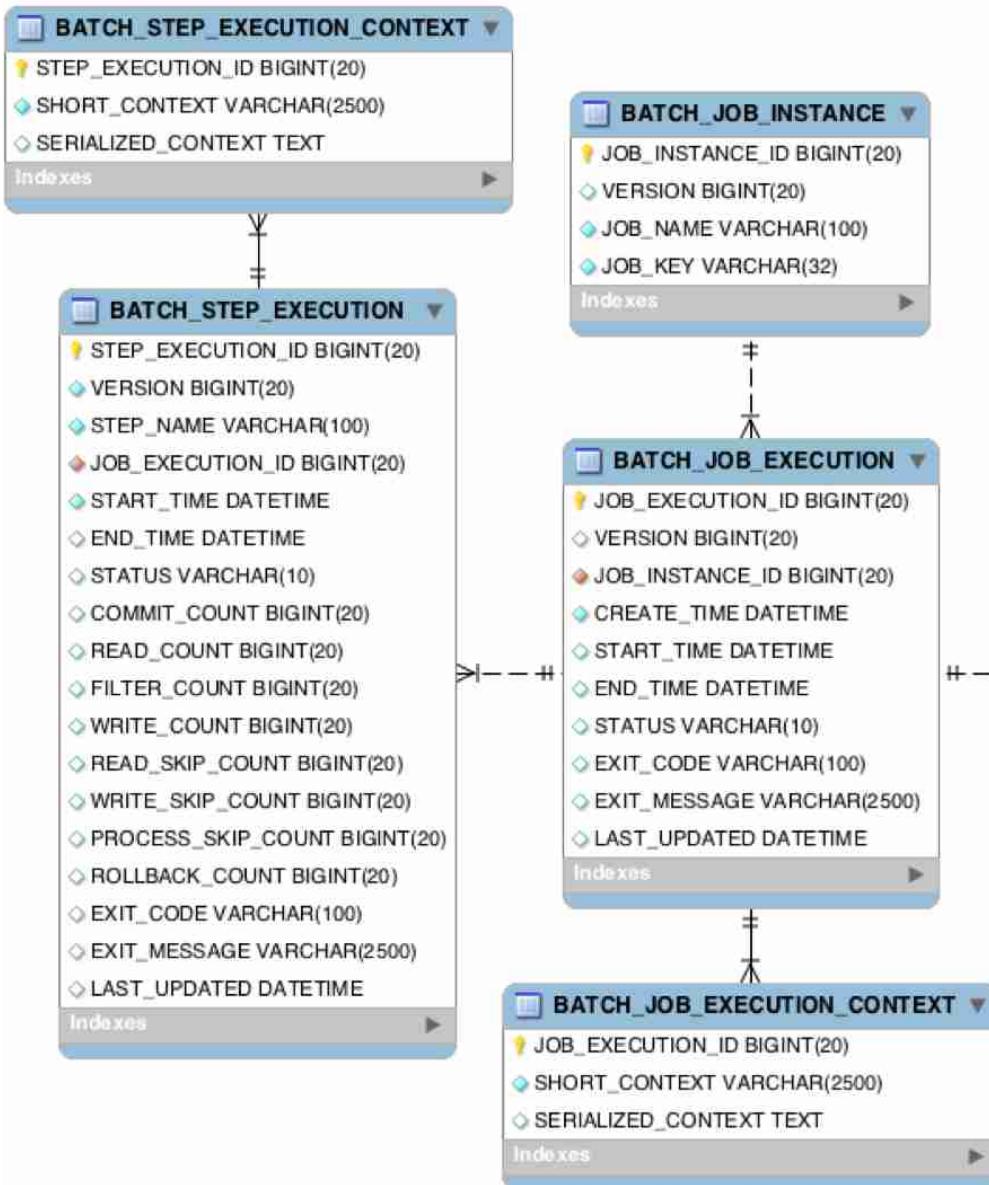


JobRepository Practicalities

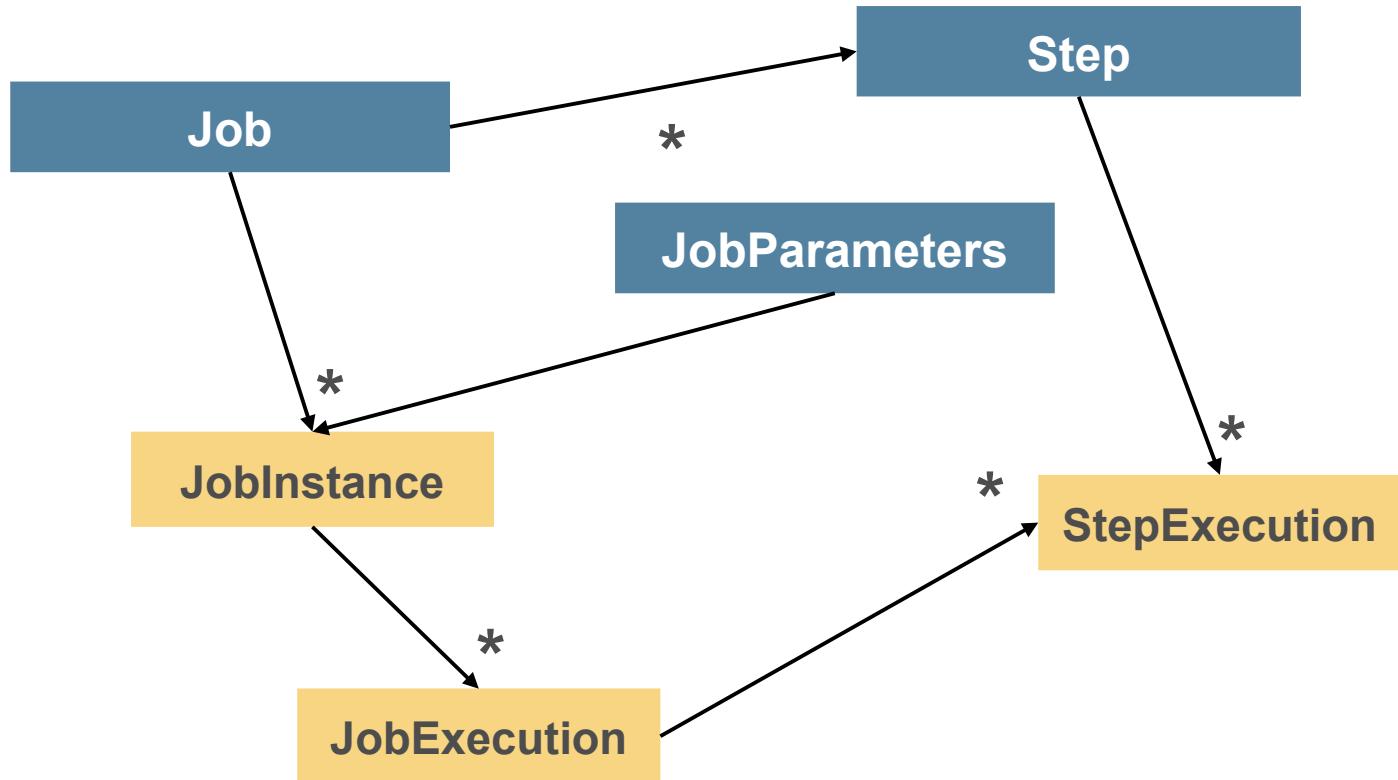
- JobRepository is really an **internal** interface
 - no need for users to know details
- Still need to create an instance of JobRepository
- Spring Batch provides namespace support:

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

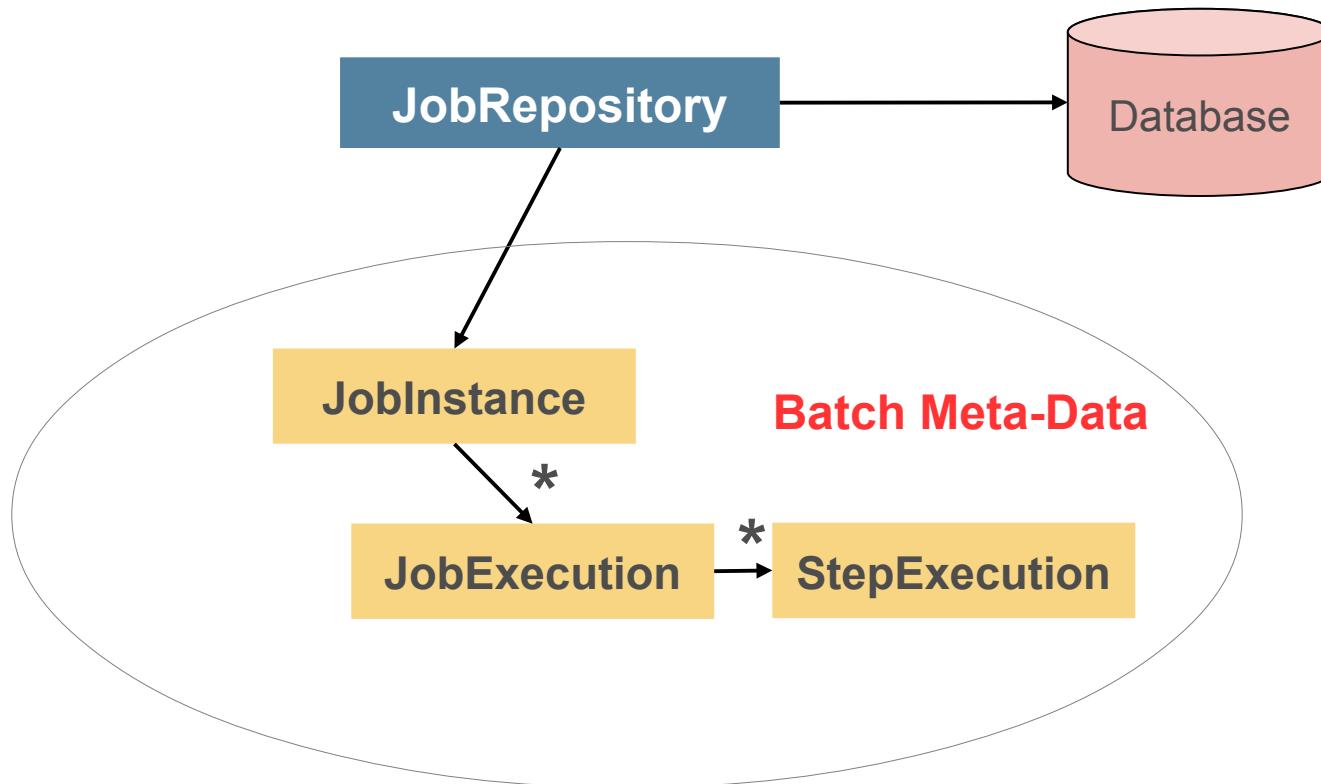
Batch Meta-Data Schema



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.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  
@Autowired JobLauncher jobLauncher;  
@Autowired @Qualifier("endOfDayJob") Job job;  
  
// ...  
  
// 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**

DB Item Readers & Writers

- 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

DB Item Readers

- 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
- Check JDBC driver for 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

Creating a simple 2-step job
using Spring Batch

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
- Step scope
- Intro to Skip, Retry, Restart
- Other Listeners
- Business Logic Delegation
- Java Configuration

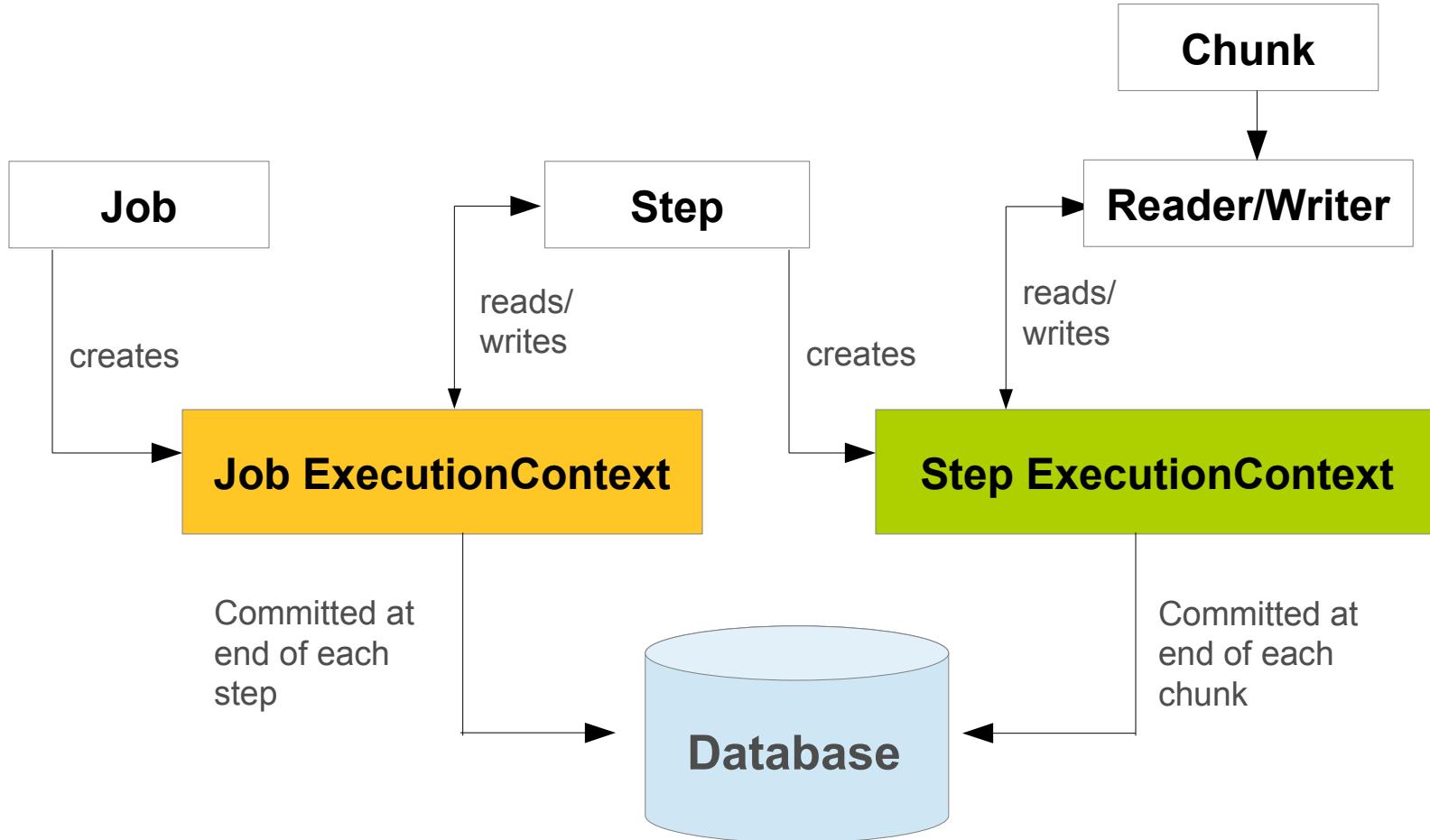
ExecutionContext

- 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

ExecutionContext



Topics in this Session

- ExecutionContext
- **Stateful ItemReaders/Writers**
- Reading Flat Files
- Step scope
- Intro to Skip, Retry, Restart
- Other Listeners
- Business Logic Delegation
- Java Configuration

Stateful ItemReaders/-Writers

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 { }  
}
```

Initialize File Position
from Last Run

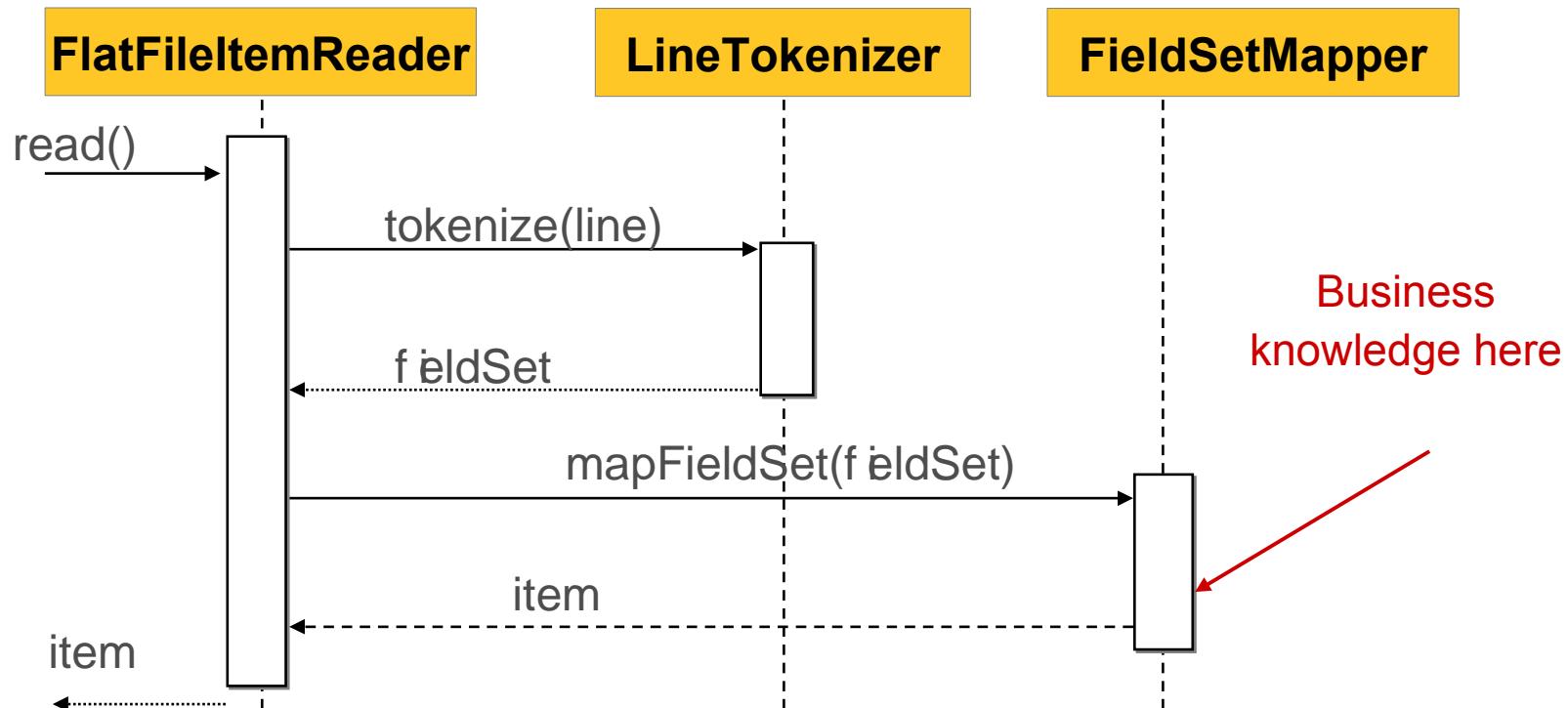
Assumes single-threaded step

Topics in this Session

- ExecutionContext
- Stateful ItemReaders/Writers
- **Reading Flat Files**
- Step scope
- Intro to Skip, Retry, Restart
- Other Listeners
- Business Logic Delegation
- Java Configuration

FlatFileItemReader

- Stateful, remembers number 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.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean class="org.sfw.batch.item.file.transform.DelimitedLineTokenizer">
                    <property name="names"> <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>
```

File to read

default impl

For use with CSV-like files

for use in FieldSet

FieldSetMappers in Spring Batch

- Spring Batch provides 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
- **Step scope**
- Intro to Skip, Retry, Restart
- Other Listeners
- Business Logic Delegation
- Java Configuration

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
- Step scope
- **Intro to Skip, Retry, Restart**
- Other Listeners
- Business Logic Delegation
- Java Configuration

Common Batch Idioms

- Batch jobs typically process large amounts of homogeneous input
- 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

Retry

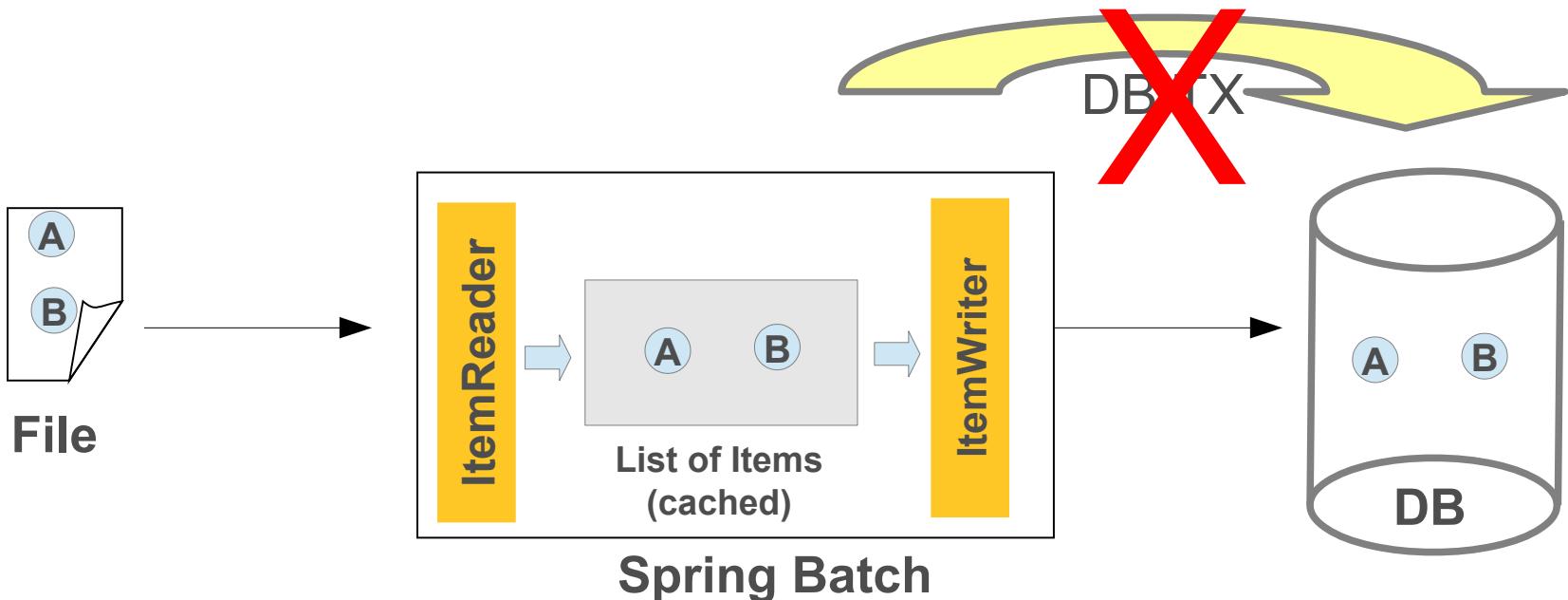
- A processing error could be *transient*
 - intermittent failure caused by external circumstances
- Simply retrying the operation might succeed
 - No need to fail the entire execution immediately
- Can mark certain exceptions as retryable
 - With max. # of retries per item

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

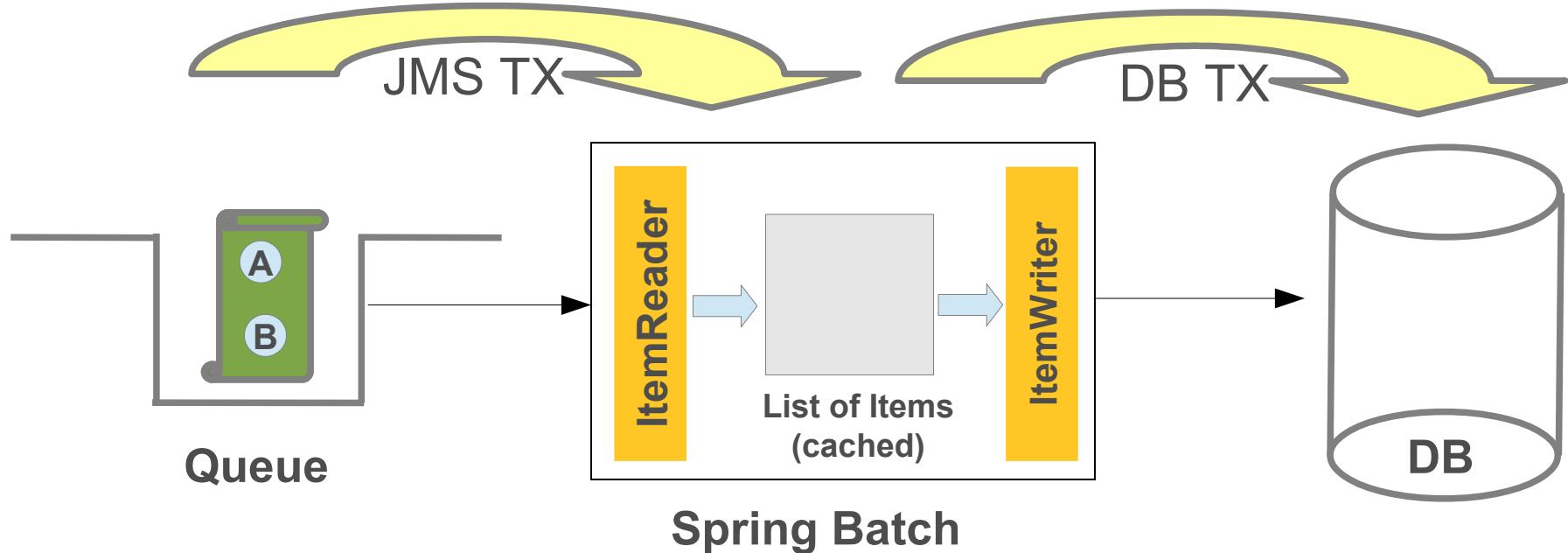
Transactional ItemReaders



1. Reads from file
2. Receive message
3. DB TX starts
4. DB rollbacks
5. Message cached
6. Retries to commit

Message in cache, not required to read from file again

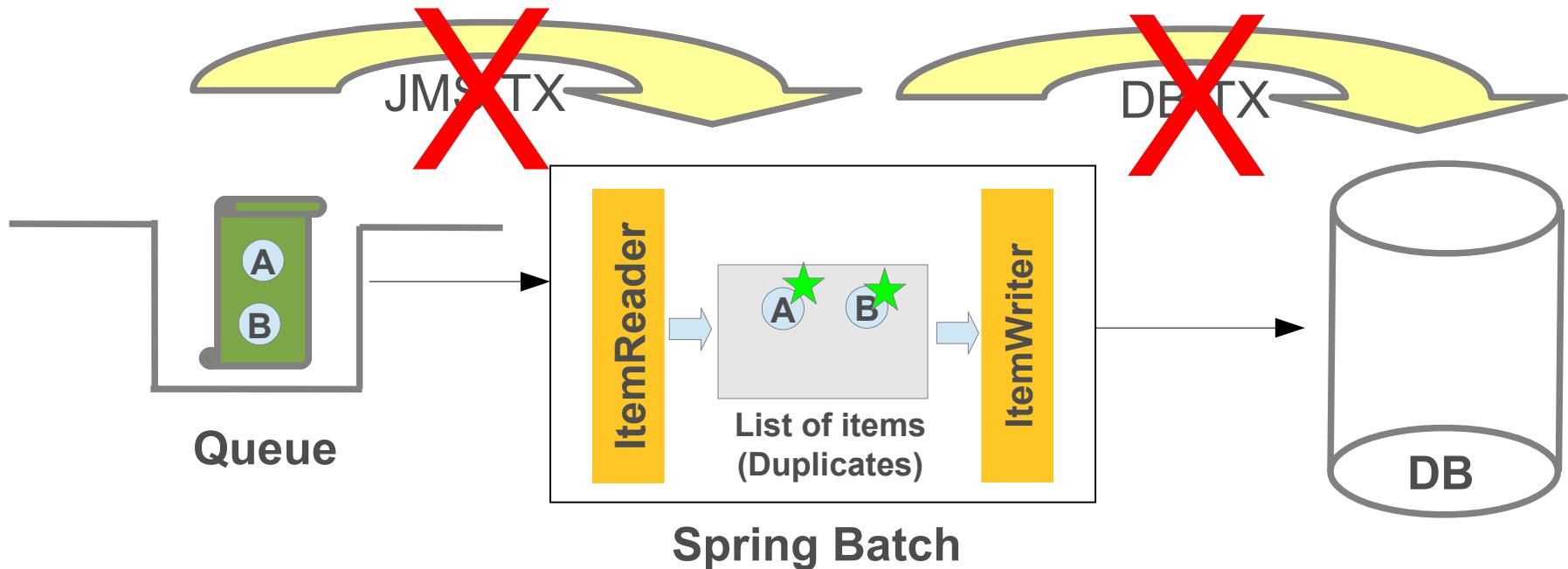
Transactional ItemReaders: JMS–DB Best Case Scenario



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

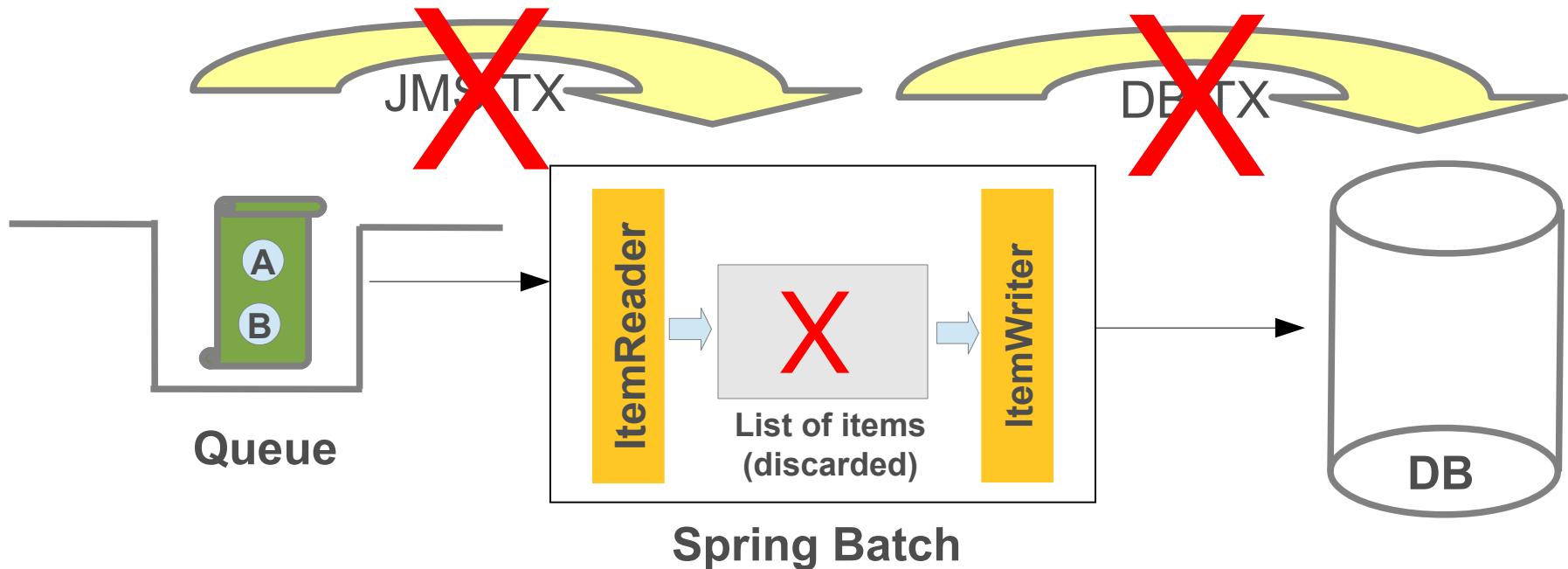
Transactional ItemReaders: JMS-DB Rollback



1. JMS TX starts
2. Receive message
3. DB TX starts
4. DB TX rollbacks
5. Items in cache
6. JMS TX rollbacks

Message in queue as well as in cache, resulting in duplicates

Transactional ItemReaders: JMS-DB Rollback



1. JMS TX starts
2. Receive message
3. DB TX starts
4. DB TX rollbacks
5. Items in cache discarded
6. JMS TX rollbacks

Message in queue, ItemReader is invoked again

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>
```

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

Restart

- 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 number of step restarts allowed (default: 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
- Step scope
- Intro to Skip, Retry, Restart
- **Other Listeners**
- Business Logic Delegation
- Java Configuration

Listeners

- 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

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

SkipListener

- 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

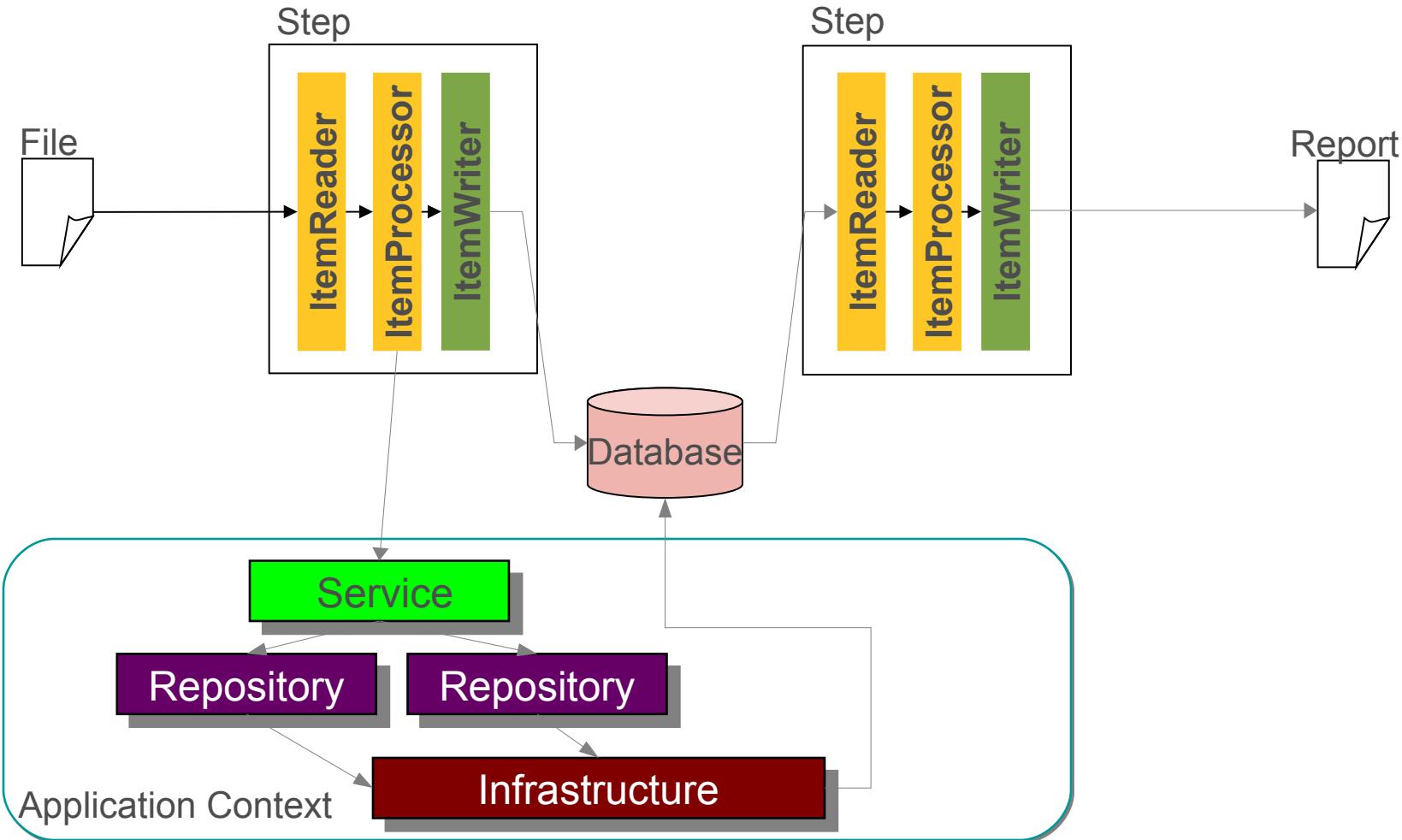


Note: The item responsible for exceeding the skip limit is not received by the SkipListener

Topics in this Session

- ExecutionContext
- Stateful ItemReaders/Writers
- Reading Flat Files
- Step scope
- Intro to Skip, Retry, Restart
- Other Listeners
- **Business Logic Delegation**
- Java Configuration

Business Logic Delegation – Spring Application

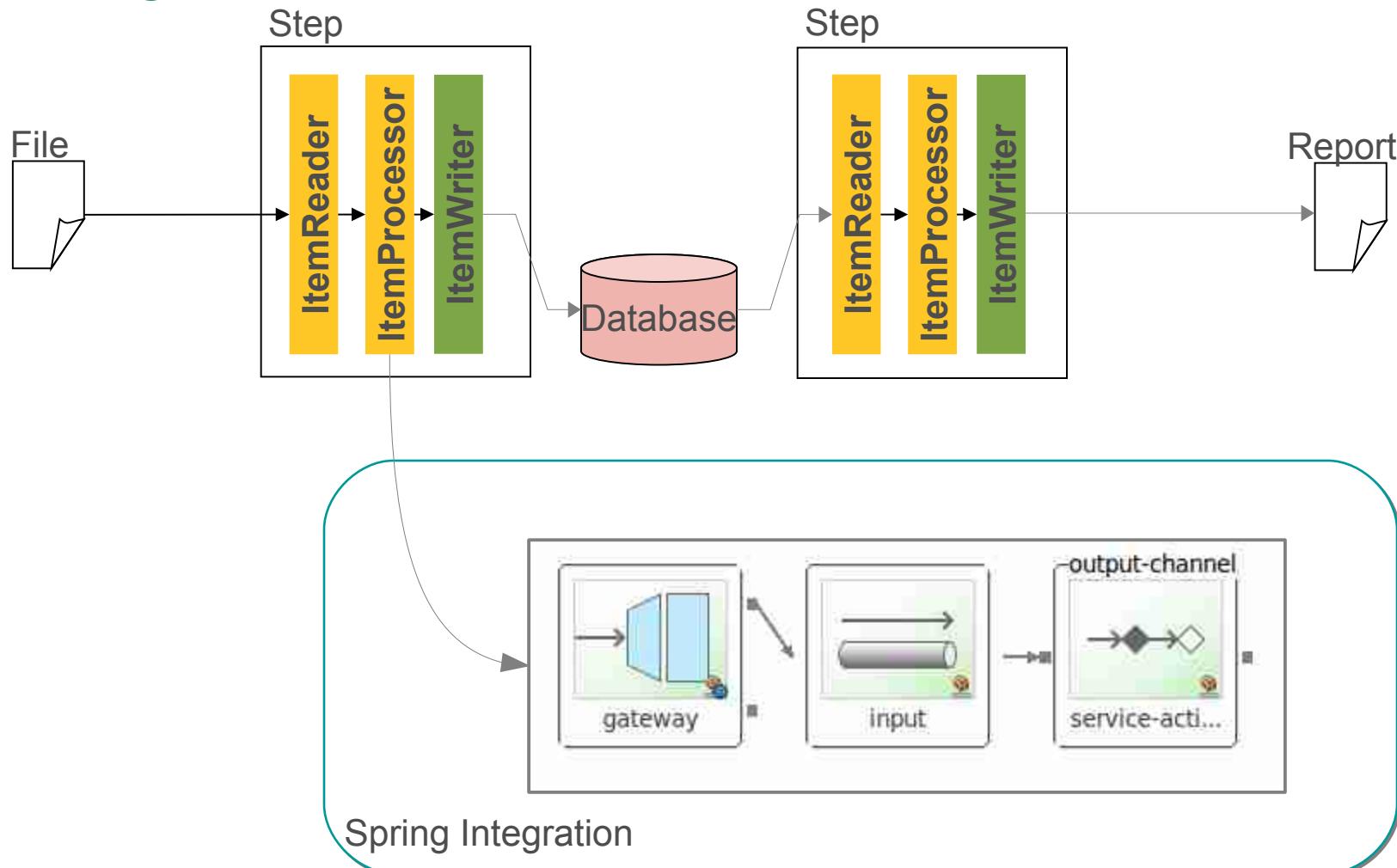


Business Logic Delegation - adapter

```
<tasklet>
    <chunk reader="diningRequestsReader"
          writer="reportWriter"
          commit-interval="10">
        <processor adapter-method="rewardAccountFor">
            <beans:ref bean="rewardNetwork"/>
        </processor>
    </chunk>
</tasklet>
```

Item processing
is delegated to
a business service

Business Logic Delegation – Spring Integration



Topics in this Session

- ExecutionContext
- Stateful ItemReaders/Writers
- Reading Flat Files
- Step scope
- Intro to Skip, Retry, Restart
- Other Listeners
- Business Logic Delegation
- **Java Configuration**

Java configuration with Spring Batch

- Spring Batch has Java configuration support
 - As of Spring Batch 2.2
- Consists of a Java-based builder API
 - Job builder, step builder, etc.
- Benefits
 - Type-safety
 - XML IDE support not longer needed
 - Less verbose than XML

@EnableBatchProcessing

- Registers the Spring Batch infrastructure
- Just needs a DataSource bean

Creates job repository,
job launcher, transaction manager,
and batch artifacts factories

```
@EnableBatchProcessing  
@Configuration  
public class BatchExecutionConfig {  
  
}
```

Declaring the Job

- Inside an `@Configuration` class

Created by `@EnableBatchProcessing`

```
@Autowired  
private JobBuilderFactory jobBuilderFactory;
```

```
@Bean  
public Job myJob(Step firstStep) {  
    return jobBuilderFactory.get("myJob")  
        .flow(firstStep).end()  
        ...  
        .build();
```

Declares the job

Configures the job

Creates the job bean

Create a Step

```
@Autowired
```

```
private StepBuilderFactory stepBuilderFactory;
```

Created by @EnableBatchProcessing

```
@Bean
```

```
public Step firstStep(ItemReader<Person> reader,  
                      ItemProcessor<Person, Person> processor,  
                      ItemWriter<Person> writer) {  
    return stepBuilderFactory.get("firstStep")  
        .<Person, Person>chunk(10)  
        .reader( reader )  
        .processor( processor )  
        .writer( writer )  
        ...  
        .build();  
}
```

Declared elsewhere and automatically injected

Declares, configures, and creates a chunk-oriented step

Create an ItemReader

- Simple Spring bean with Java configuration

```
@Bean  
public ItemReader<Person> reader() {  
    FlatFileItemReader<Person> reader = new FlatFileItemReader<>();  
    reader.setResource( new ClassPathResource("sample-data.csv") );  
    reader.setLineMapper( new DefaultLineMapper<Person>() {{  
        setLineTokenizer( new DelimitedLineTokenizer() {{  
            setNames( new String[]{"firstName", "lastName"});  
        }});  
        setFieldSetMapper(new BeanWrapperFieldSetMapper<Person>() {{  
            setTargetType(Person.class);  
        }});  
    }});  
    return reader;  
}
```

Create an ItemProcessor

- Simple Spring bean with Java configuration

```
@Bean
```

```
public ItemProcessor<Person, Person> processor(PersonService personService) {  
    ItemProcessorAdapter<Person, Person> processorAdapter =  
        new ItemProcessorAdapter<>();  
    processorAdapter.setTargetObject(personService);  
    processorAdapter.setTargetMethod("validate");  
  
    return processorAdapter;  
}
```

Create an ItemWriter

- Simple Spring bean with Java configuration

```
@Bean
public ItemWriter<Person> writer(DataSource dataSource) {
    JdbcBatchItemWriter<Person> writer = new JdbcBatchItemWriter<>();
    writer.setItemSqlParameterSourceProvider(
        new BeanPropertySqlParameterSourceProvider<>());
    writer.setSql("INSERT INTO people (first_name, last_name) +
        " VALUES (:firstName, :lastName)");
    writer.setDataSource( dataSource );
    return writer;
}
```

Skip – XML vs Java Config

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

```
@Bean
public Step step1(ItemReader reader, ItemWriter writer) {
    return stepBuilderFactory.get("step1").chunk(10)
        .faultTolerant()
        .skip( FlatFileParseException.class )
        .skipLimit( 10 )
        .reader( reader ).writer( writer )
        .build();
}
```

Step Scope to Refer to Job Parameters

```
@Bean  
@StepScope  
public ItemReader<Person> reader(  
    @Value("#{jobParameters['input.resource.path']}") String input) {  
    FlatFileItemReader<Person> reader = new FlatFileItemReader<>();  
    reader.setResource( new UrlResource(input) );  
    ...  
    return reader;  
}
```

The bean is step-scoped

@Value annotation and SpEL expression to refer to the job parameter

Supports “file:input.csv”-like syntax

Step Scope and Proxies

- Step scope is implemented with proxies
- The proxy has the type of the method return type
- Use appropriate type if life cycle methods needed

ItemStreamReader declares life cycle methods. Spring Batch call them during chunk processing.

```
@Bean  
@StepScope  
public ItemStreamReader<Person> reader(  
    @Value("#{jobParameters['input.resource.path']}") String input) {  
    FlatFileItemReader<Person> reader = new FlatFileItemReader<>();  
    reader.setResource( new UrlResource(input) );  
    ...  
    return reader;  
}
```

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://docs.spring.io/spring-batch-admin/>

- 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

The screenshot shows a web browser window with the following details:

- Title Bar:** File Edit View History Bookmarks Tools Help
- Address Bar:** http://localhost:8080/batch-2-solution/
- Tab:** Spring Batch Admin
- Header:** Spring Batch Admin (with SpringSource logo)
- Content:** A table listing API endpoints:

Resource	Method	Description
/configuration	GET	
/files	GET	
/files/**	GET	
/files/{path}	POST	
/home	GET	
/job-configuration-files	GET	
/job-configuration-multipart	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	
- Bottom Bar:** Done, S3Fox

Registered Jobs

The screenshot shows a web browser window with the following details:

- Address Bar:** http://localhost:8080/batch-2-solution/batch/job
- Title Bar:** Spring Batch Admin: Jobs
- Header:** Spring Batch Admin (with a green gradient background) and Spring source logo.
- Navigation:** Home, Jobs, Executions, Files, SpringSource, Spring Batch.
- Content:** A table titled "Job Names Registered" showing three rows of data:

Name	Description	Execution Count	Launchable	Incrementable
infinite	No description	0	true	true
job1	No description	0	true	true
job2	No description	0	true	false
- Page Information:** Rows: 1-3 of 3, Page Size: 20.
- Footer:** © Copyright 2009-2010 SpringSource. All Rights Reserved, Contact SpringSource, Done, S3Box logo.

Launch

The screenshot shows a web browser window with the following details:

- Address Bar:** http://localhost:8080/batch-2-solution/batch/job
- Title Bar:** Spring Batch Admin: Job Summ...
- Header:** Spring Batch Admin (with a green background), SpringSource logo, and Spring Batch logo.
- Navigation:** Home, Jobs, Executions, Files, SpringSource, Spring Batch.
- Form:** Job name=job1: Job Parameters (Incrementable) (key=value pairs):
- Description:** 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.
- Text:** There are no job instances for this job.
- Footer:** © Copyright 2009-2010 SpringSource. All Rights Reserved, Contact SpringSource, Done, S3Box logo.

A blue oval highlights the "Launch" button and its associated input fields.

Launched

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 Instances for Job (job1)" section.

Job name=job1:

Job Parameters: `run.count(long)=0,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.

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](#).

Completed

The screenshot shows a web browser window displaying the Spring Batch Admin application at <http://localhost:8080/batch-2-solution/batch/>. The title bar of the browser shows "Spring Batch Admin: Job Execut...". The main page header features the SpringSource logo. Below the header, there is a navigation menu with links for Home, Jobs, Executions, Files, SpringSource, and Spring Batch. The main content area displays a table titled "Recent and Current Job Executions for Job = job1, instanceId = 0". The table has columns: ID, Instance, Name, Date, Start, Duration, Status, and ExitCode. A single row is present in the table, showing values: 0, 0, job1, 2010-09-01, 14:05:58, 00:00:00, COMPLETED, and COMPLETED. The "Status" and "ExitCode" columns are circled in blue.

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

Job Execution Details

The screenshot shows the Spring Batch Admin interface. At the top, there's a green header bar with the text "Spring Batch Admin" on the left and the "spring source" logo on the right. Below the header is a navigation bar with links for "Home", "Jobs", "Executions", and "Files". To the right of the navigation bar are two more links: "SpringSource" and "Spring Batch". The main content area is titled "Details for Job Execution". It features a "Stop" button in a red box. Below it is a table with the following data:

Property	Value
ID	0
Job Name	<u>job1</u>
Job Instance	0
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

The screenshot shows a web browser window with the following details:

- Address Bar:** http://localhost:8080/batch-2-solution/batch/jobs/exec
- Title Bar:** Spring Batch Admin: Step Exec...
- Header:** Spring Batch Admin (with SpringSource logo) and Spring source logo.
- Navigation:** Home, Jobs, Executions, Files, SpringSource, Spring Batch.
- Content:** Step Executions for Job = job1, JobExecution = 0. A table displays the following data:

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

At the bottom, there are links for Copyright 2009-2010 SpringSource, All Rights Reserved, Contact SpringSource, Done, and S3 Fox.

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 completed)

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
Done	Done

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	

Failures

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	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
Done	

Restart or Abandon

The screenshot shows the Spring Batch Admin interface. At the top, there's a green header bar with the title "Spring Batch Admin" and the SpringSource logo. Below the header, a navigation bar includes links for "HOME", "Jobs", "Executions", and "Files". On the right side of the navigation bar are links for "SpringSource" and "Spring Batch". The main content area is titled "Details for Job Execution". It features two buttons: "Abandon" (which is highlighted with a blue oval) and "Restart". Below these buttons is a table displaying job execution properties:

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 main content area, there's a "Done" button. In the bottom right corner of the entire window, there are icons for "S3Box" and "Pivotal". A cursor arrow is visible on the right side of the interface.

Stop a Running Job

Spring Batch Admin

SpringSource Spring Batch

Details for Job Execution

Stop

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	00:00:19
Status	STARTED
Exit Code	UNKNOWN
Step Executions Count	1
Step Executions	[j1.s1]

Stopped

Spring Batch Admin

Home Jobs Executions Files

Job name=job1: **Launch**

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

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

Job Instances for Job (job1)

ID	JobExecution Count	Last Job Date
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
Rollbacks	0
Exit Code	STOPPED
Exit Message	<code>org.springframework.batch.core.JobInterruptedException</code>

Done

Restart or Abandon

The screenshot shows the Spring Batch Admin interface. At the top, there's a green header bar with the title "Spring Batch Admin" and the SpringSource logo. Below the header is a navigation menu with links for Home, Jobs, Executions, Files, SpringSource, and Spring Batch. The main content area is titled "Details for Job Execution". It features two buttons: "Abandon" and "Restart", which are circled in blue. Below these buttons is a table with the following data:

Property	Value
ID	2
Job Name	job1
Job Instance	2
Job Parameters	run.count(long)=2,fail=false
Start Date	2010-09-01
Start Time	14:32:24
Duration	00:01:35
Status	STOPPED
Exit Code	STOPPED
Step Executions Count	1
Step Executions	[j1.s1]

At the bottom left of the main content area is a "Done" button. In the bottom right corner, there's a small "S3 Fox" logo.

Restarting a Stopped/Failed Job

The image shows the Spring Batch Admin interface. At the top, there's a green header bar with the 'Spring Batch Admin' logo on the left and the 'spring source' logo on the right. Below the header is a navigation menu with links for Home, Jobs, Executions, Files, and two SpringSource links. The main content area is titled 'Recent and Current Job Executions'. It features a 'Stop All' button and a table listing four job executions. The table has columns for ID, Instance, Name, Date, Start, Duration, Status, and ExitCode. The rows show the following data:

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

Below the table, it says 'Rows: 1-4 of 4 Page Size: 20'. A blue arrow points from the text 'Restart is a new Execution of the same Instance' to the 'Instance' column header in the table.

© Copyright 2009-2010 SpringSource. All Rights Reserved

Contact SpringSource

Done

S3Fox

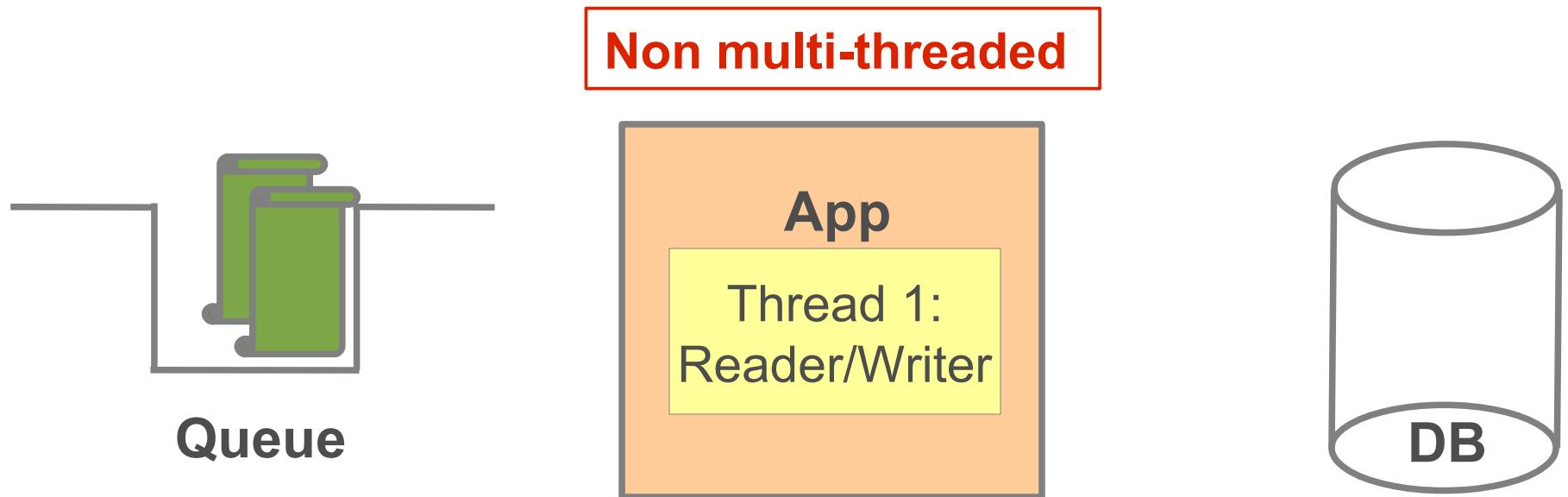
Topics in this Session

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

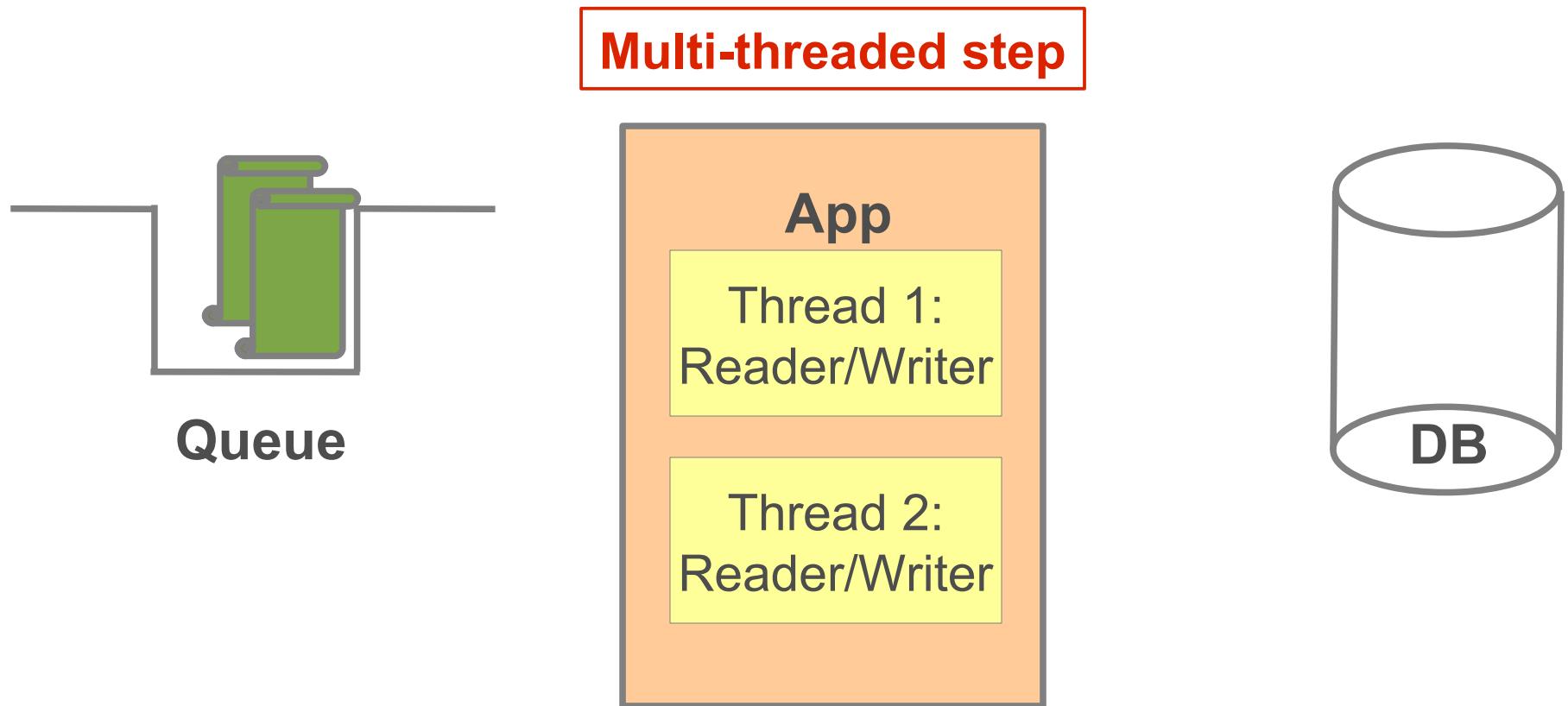
Scaling and Parallel Processing

- First Rule:
 - Use the simplest technique to get the job done in the required time
 - Do not optimize/parallelize unnecessarily
- Different options available:
 - Where is the bottleneck? Reader, Writer, both?
 - What type of Input/Output? JMS, DB, File?

Use case: JMS Input



Use case: JMS Input



Multi-threaded Step

- Just add a TaskExecutor to the tasklet

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

- 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

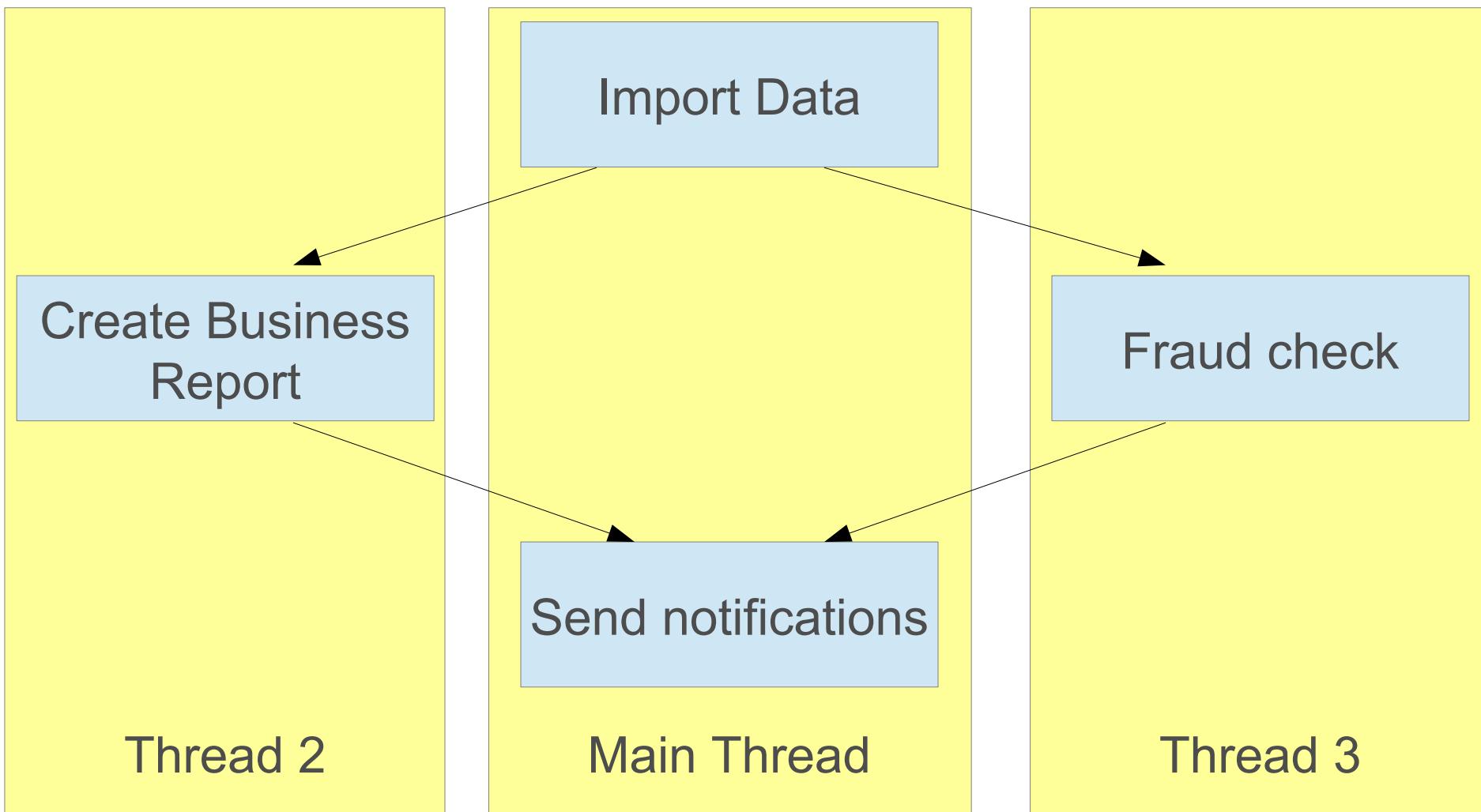
Multi-threaded Step

- When to use:
 - Source can handle the load balancing (like JMS, AMQP)
 - Own stateful readers taking care of partitioning the data
 - To speed up the step execution
- When to use NOT:
 - With off-the-shelf readers for DB, File, etc.
 - Every Thread would read the same data
 - If the reader/write is not thread safe.
 - Not all off-the-shelf readers/writers are thread safe, check the JavaDocs

Use case: Independent Steps



Use case: Independent Steps



Parallel Steps

```
<job id="job1">
  <step id="importData" />
  <split id="split1" task-executor="taskExecutor" next="sendNotifications">
    <flow>
      <step id="createBusinessReport" />
    </flow>
    <flow>
      <step id="fraudCheck" />
    </flow>
  </split>
  <step id="sendNotifications" />
</job>
```

Parallel Steps

- When to use:
 - For independent steps.
 - To speed up the whole JOB execution.
- When to use NOT:
 - For dependent steps.
 - To speed up the STEP execution itself.

Use case: DB Input

ID	NAME
1	Noelle
2	Paul
3	Mike
4	Michael
5	Mark
6	Ken
7	Kevin
8	Wes
9	Bijoy
10	Clarence

No optimization

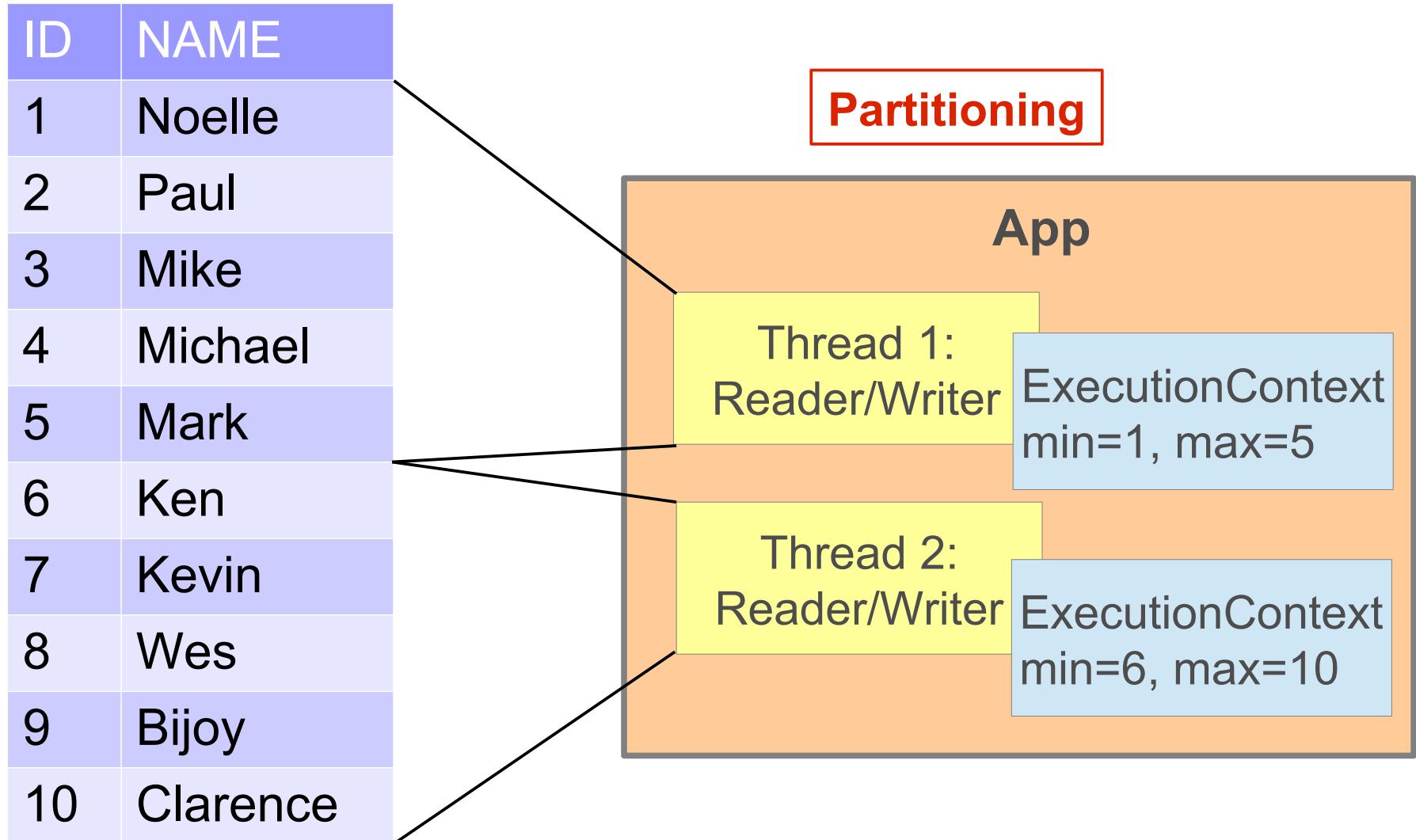
App

Thread 1:
Reader/Writer

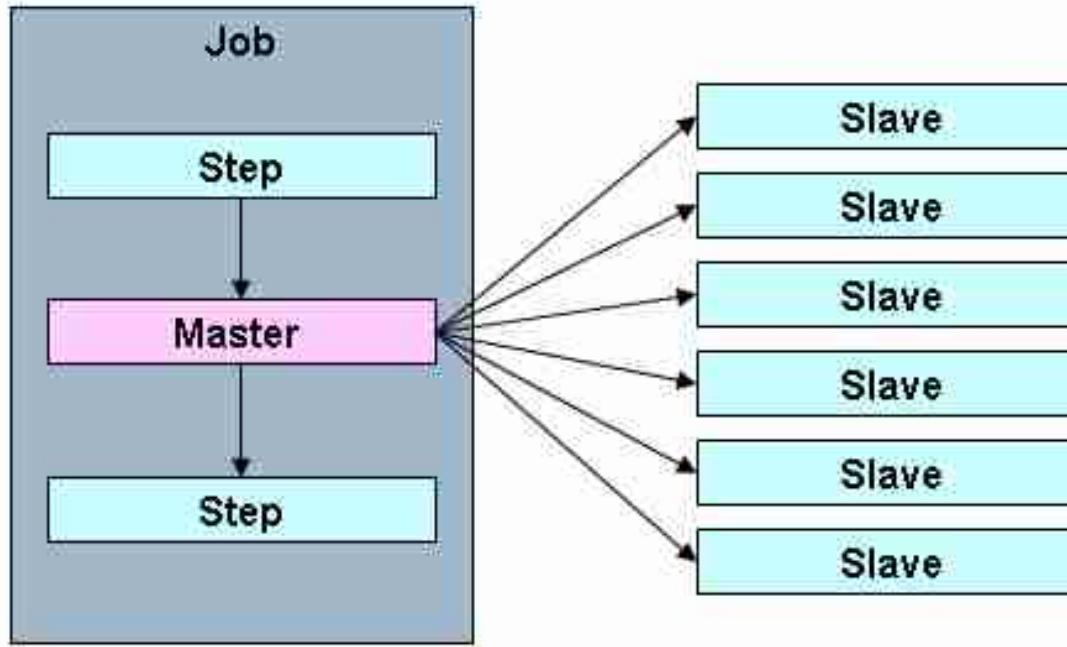
ExecutionContext

Cannot use multi-threaded step,
as both steps would access same data.

Use case: DB Input



Partitioning



- Similar to multi-threaded step, but with a “Partitioner” who splits up the input data.
- Master step is just the container.
- Slaves are all the same steps, but with different Execution Contexts, running in different threads.

Partitioning

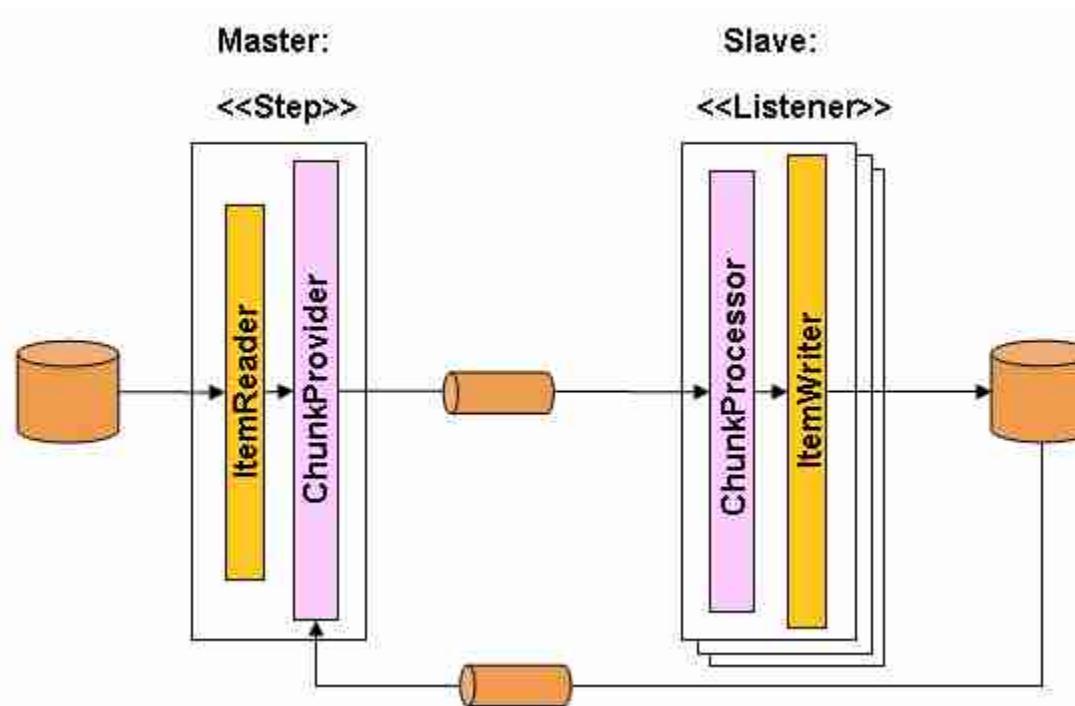
```
<job id="job1">
  <step id="master">
    <partition step="processData" partitioner="partitioner">
      <handler grid-size="2" task-executor="taskExecutor" />
      <!-- Creates processData.partition0 and processData.partition1 -->
    </partition>
  </step>
</job>
<step id="processData">
  <tasklet>
    <chunk reader="jdbcReader" writer="..." />
  </tasklet>
</step>
```

- See batch-admin-solution for a complete example.

Partitioning

- When to use:
 - For non-messaging sources, like DBs
 - To speed up the step execution
- When to use NOT:
 - For single threaded sources, like a File
 - For data, which cannot be partitioned (very rarely)

Remote Chunking



- Reader reads chunks; they are sent for remote processing
- Requires durable middleware (e.g. JMS) for communication between master and slaves

Remote Chunking

- When to use:
 - To speed up processing and/or writing
 - Can use off-the-shelf readers, writers and processors
 - Slaves are multiple processes
- When to use NOT:
 - To speed up reading (still single threaded)

Lab

Using Spring Batch Admin to Manage
Jobs

Spring XD

Introduction

Overview of Spring XD

Topics in this session

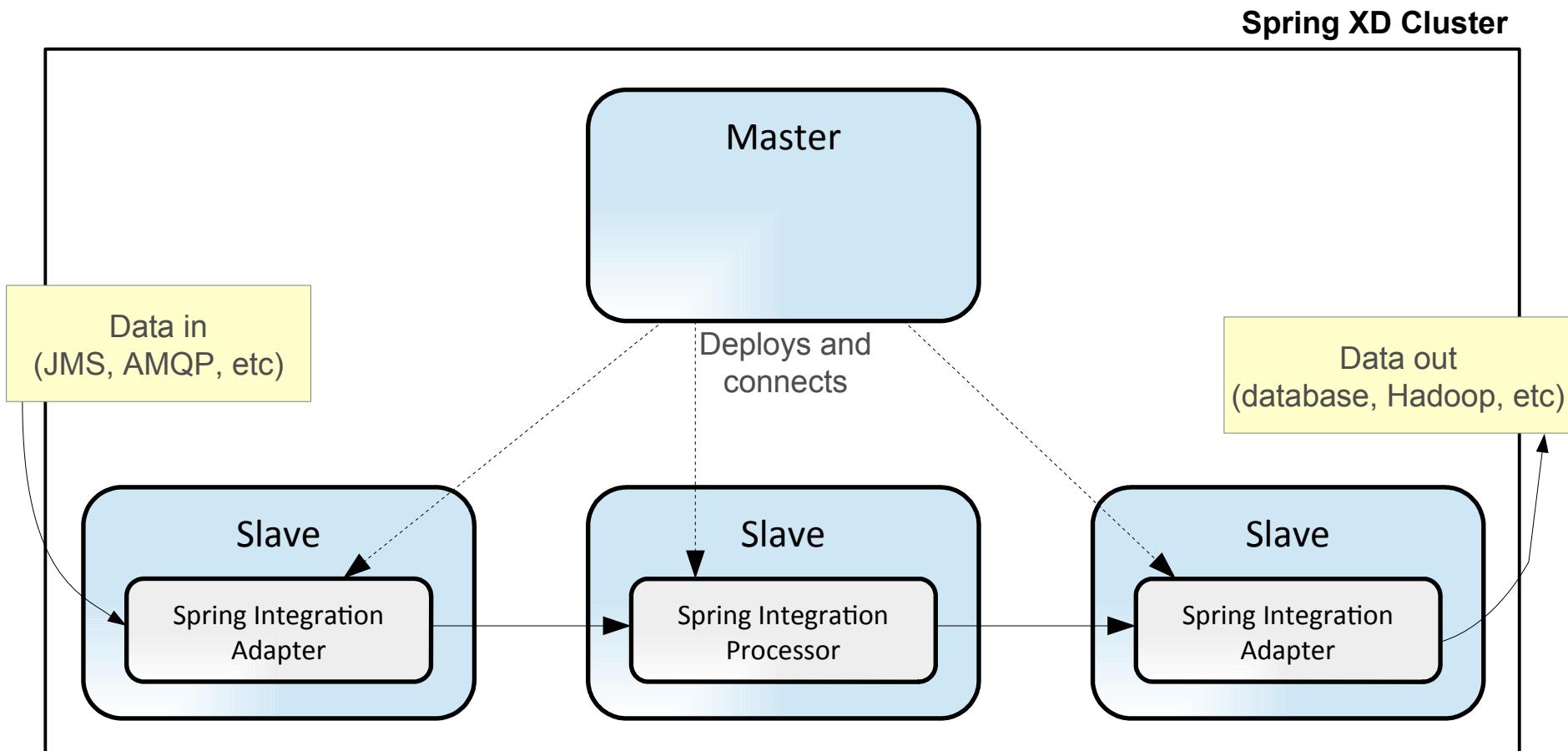
- **Spring XD overview**
- Lab, part 1, installation

What is Spring XD?

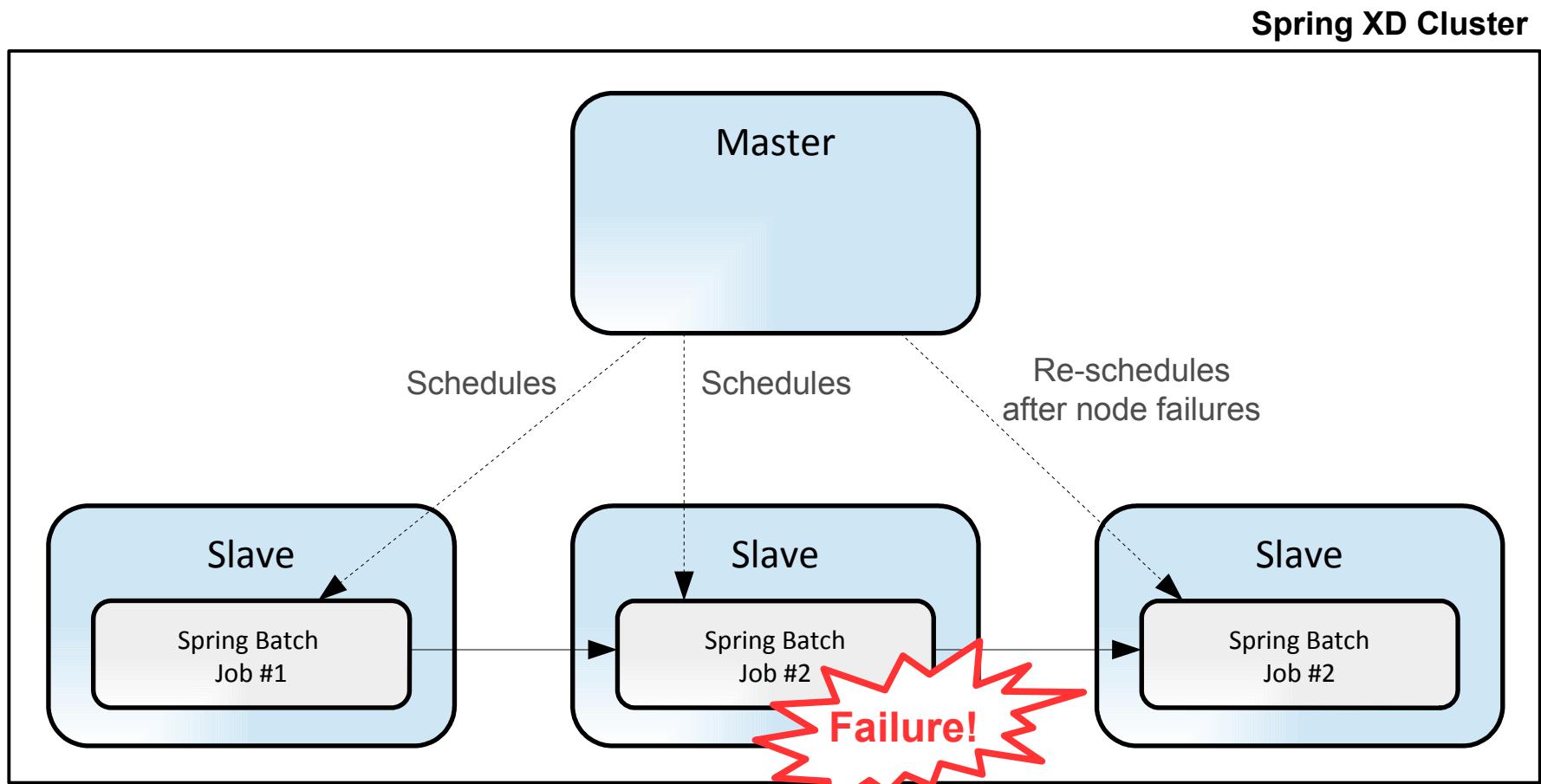
- It is a runtime for Spring Batch and Spring Integration
- It can run flows and jobs in a cluster for
 - Scalability
 - Fault-tolerance
- It brings also
 - Ease of deployment (through a REST API)
 - Monitoring
 - High throughput



Spring Integration flow in Spring XD



Spring Batch jobs in Spring XD



Data Ingestion with Spring XD

- Spring XD is for data ingestion
- Data ingestion is the process of:
 - reading, cleaning, transforming and potentially enriching data from one or more sources;
 - storing the resulting processed data into a consolidated repository



Why Do We Need Data Ingestion?

- Modern applications generate massive amounts of data
- Modern data repositories are optimized for the task at hand
 - But they are not optimized for data analysis
- Data stored in diverse repositories is difficult to analyze
- Ingesting data into a consolidated repository permits practical data analysis



Data Ingestion Mechanisms

- Broken down into streams or batch jobs
- Streams: Real time event flows
 - Example: Sensor to measure speed of a vehicle, generates readings every 5 seconds
- Batch jobs: Data collected over a period of time, and processed “offline” in bulk
 - Example: Stock trades executed over a day, batch job is run nightly to process the trading information and generate daily reports

Built-in adapters & processors

- The same as in Spring Integration
 - As Spring XD builds on top of Spring Integration
- Among others:
 - Adapters: HTTP, email, Twitter, TCP, JMS, RabbitMQ...
 - Processors: splitter, filter, json mapper, ...



See: [Spring XD Reference - Sources / Sinks / Processors](#)
<http://docs.spring.io/spring-xd/docs/current/reference/html/#sources>

When To Use Spring XD

- High-throughput distributed data ingestion from various input sources into big data stores
- Real-time analytics at ingestion time
- Workflow orchestration and lifecycle management for batch jobs
- Distributed high-throughput data export
- Can you think of any specific use cases?

Spring XD architecture overview

- User interacts with the XD Admin server using the XD Shell
- Submits tasks
- XD Admin server is responsible for executing tasks to manage streams and jobs
 - Creating, deploying, destroying, etc.
- Streams and jobs consist of modules
- Modules run in XD Containers
 - Each XD Container can run multiple modules

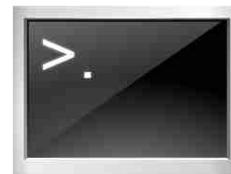
Spring XD architecture

Developer



create stream
create job
etc...

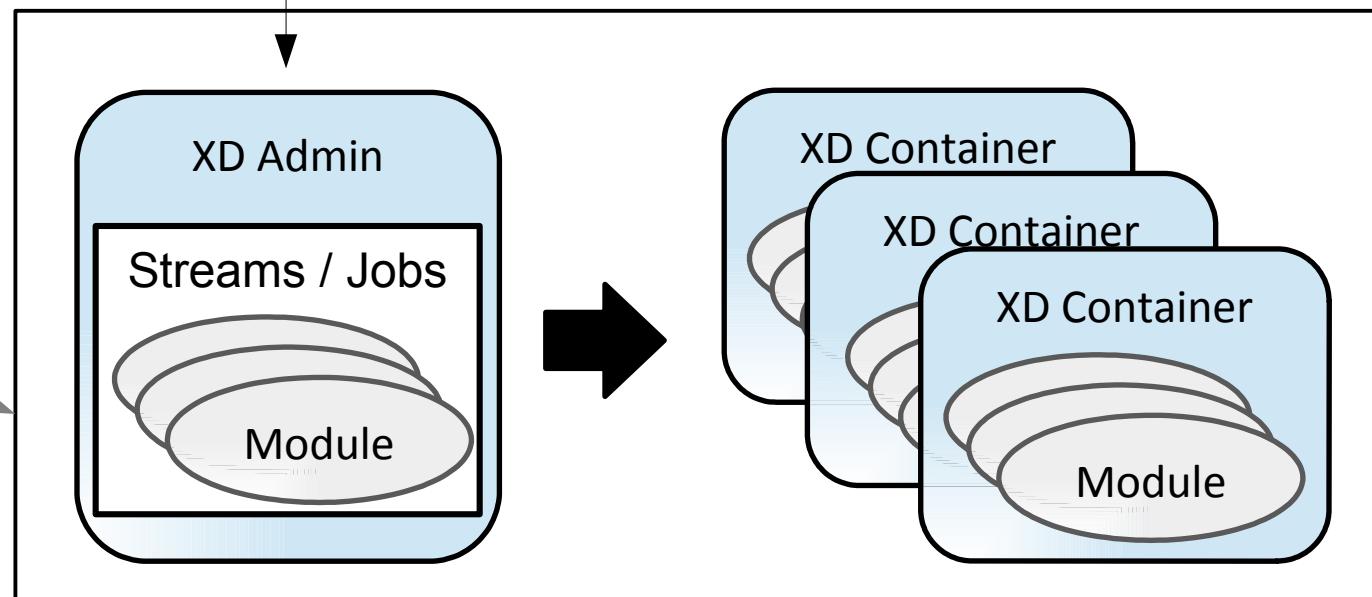
XD Shell



REST

Spring XD Cluster

Deploys on
On premise cluster
YARN (Hadoop)
EC2
Pivotal CF



XD Admin Server

- Embedded servlet container
- Exposes REST endpoints for
 - Creating, deploying, undeploying, and destroying streams and jobs
 - Querying runtime state, analytics, and the like
- Monitors runtime state using Apache ZooKeeper

Modules in Spring XD

- The unit of deployment
- Streams
 - Continuous "feed" of data
 - Often real time
 - Eg: Tweets, Log Messages, Stock tickers
- Jobs
 - Processing on large amounts of data
 - Actual Spring Batch jobs
 - Eg: Daily analysis of credit card transactions

Modules

- Definitions stored in Module Registry
 - In `<xd-install-directory>/modules`
- Consist of
 - Spring XML configuration file
 - Classes for validation and options
 - Dependent JAR's
- Options
 - Placeholders that correspond to Domain Specific Language (DSL) parameters
 - Facilitates module customization eg. "--port=8100" would change listening port of HTTP source

Quiz

- What are the 2 kinds of modules in Spring XD?
 - *Answer:* streams and batch jobs
- What is a stream?
 - *Answer:* a continuous feed of data
- What is a batch job?
 - *Answer:* some processing on large amounts of data
- How are streams and batch jobs implemented?
 - *Answer:* as Spring Integration flows and Spring Batch jobs

Spring XD installation

- Distributed mode on Linux is recommended for production
- Single-node runtime available
 - Good for tests and development
 - Do not use in production!
 - ZIP archive to decompress
 - Needs Java 7

Lab

XD Lab part 1 : installation

Spring Cloud Data Flow

- The cloud native redesign of Spring XD
- Still under active development
 - As of December 2015
- More flexible in terms of deployment
 - Local, Cloud Foundry, Lattice, and Hadoop YARN
- Same programming model as Spring XD

The future of Spring XD

- Spring Cloud Data Flow will replace Spring XD
- Beta version already available
- Stable version planned for 2016
 - XD will be then EOL'd
- Spring XD modules will be Spring Cloud Data Flow-compatible
 - To ensure seamless migration

Spring XD

Streams

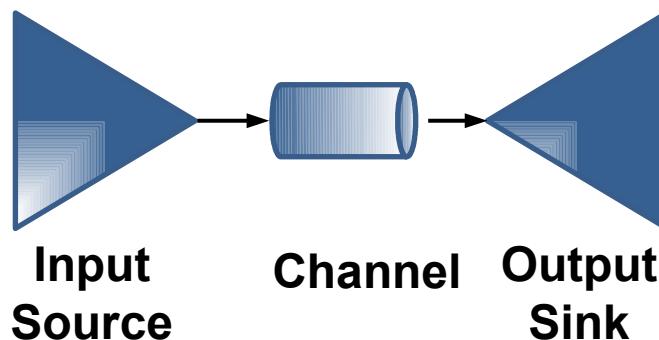
Pipes and filters with Spring XD

Topics in this session

- **Streams**
- Lab, part 2

Overview

- A stream is composed of modules and channels
- Deployed to containers by the XD Admin server
- The simplest stream has a source and a sink



Source and sink modules

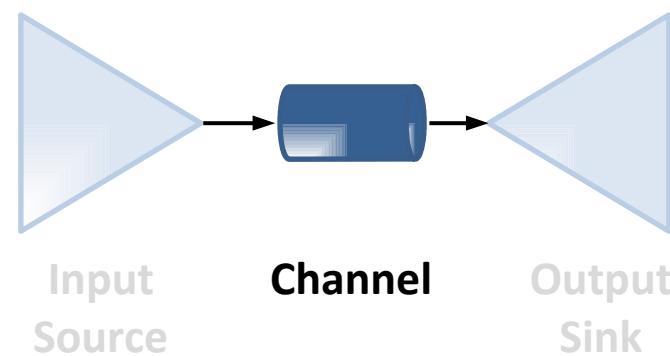
- Two types of modules in a simple stream:
 - **Source:** digests some form of input and emits a "message"
 - **Sink:** receives a message, writes to some kind of output
- Under the covers, Sources and Sinks are actually stand-alone Spring Integration contexts with a specific purpose



The third type of stream module is the processor. It allows to embed logic between the source and the sink. It is covered later.

Channels

- Channel implementation is pluggable
- Current options:
 - Local (in-memory), RabbitMQ , Kafka, and Redis
 - Future releases may support other implementations
- Single node deployment uses local (in-memory) channels by default

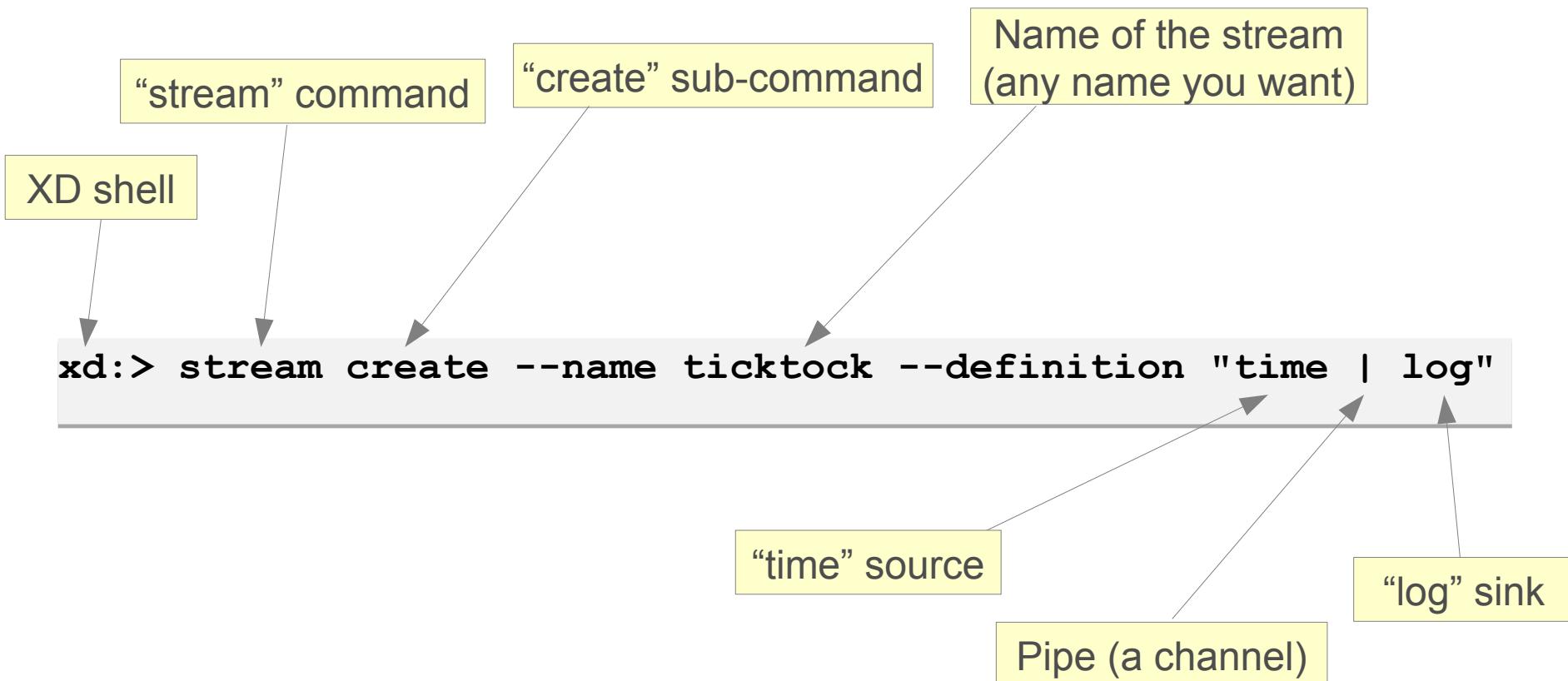


Defining Streams

- XD Domain Specific Language (DSL) mimics Unix pipes and filters syntax
- New streams created by posting stream definitions from Spring XD shell
- Defined with the “stream” command
- Syntax:

```
stream create --name <streamname> --definition <streamdef>
```

Stream creation example



Deploying Streams

- Streams must be explicitly *deployed* to start
- Use **stream deploy**

Deploy a stream previously created

```
xd:> stream deploy --name ticktock
```

- Or **stream create** with **--deploy** option

Create and deploy a stream

```
xd:> stream create --name ticktock  
      --definition "time | log" --deploy
```

Stream : Checking Status

- Use `stream list` command
- Displays definition and status

```
xd:>stream list
```

Stream Name	Stream Definition	Status
ticktock	time log	undeployed

Undeploying Streams

- To stop a stream from running

```
stream undeploy --name <streamName>
```

- Keeps stream definition intact, but stops and removes running streams from containers

```
xd:> stream undeploy --name ticktock
```

Stream name.

Deleting Streams

- Remove stream from containers

```
stream destroy --name <streamName>
```

- Destroying a deployed (i.e. running) stream will also undeploy it

```
xd:> stream destroy --name ticktock
```

Stream name.

Simple HTTP Source Example

- Instantiates an HTTP listener
- Options:
 - Flag for HTTPS (**--https**)
 - Port to listen on (**--port**). Default 9000
 - Location of SSL properties containing pkcs12 keyStore and pass phrase (**--sslPropertiesLocation**)



See: **Spring XD Reference - HTTP**

<http://docs.spring.io/spring-xd/docs/current/reference/html/#http>

Simple HTTP Source Example

```
xd:>stream create --name httpdemo --definition  
"http --port=9090 | file" --deploy  
Created and deployed new stream 'httpdemo'
```

```
xd:>http post  
    --target http://localhost:9090  
    --data "Hello Spring XD!"  
> POST (text/plain; Charset=UTF-8) http://localhost:9090  
Hello Spring XD!  
> 200 OK
```

Create HTTP POST manually

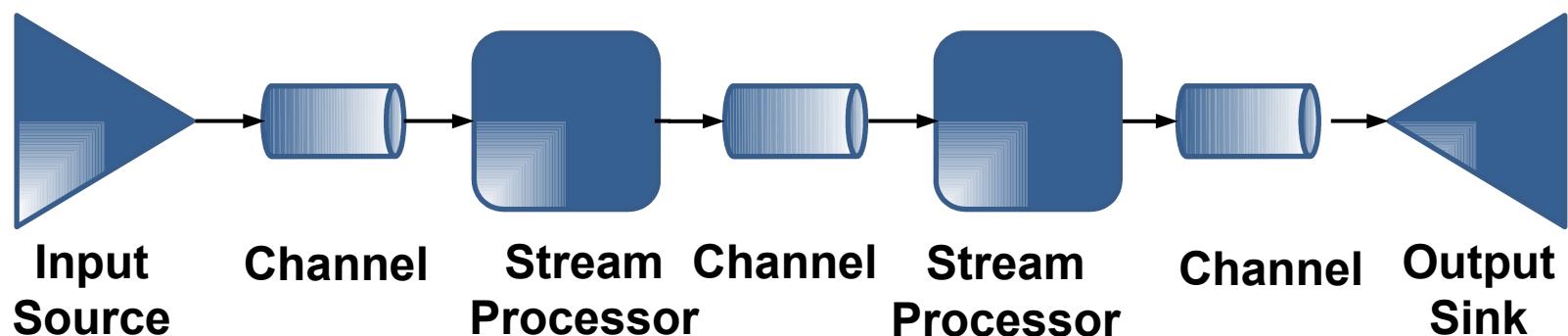
```
xd:>! more /tmp/xd/output/httpdemo.out  
command is:more /tmp/xd/output/httpdemo.out  
Hello Spring XD!
```

Use ! to run an OS command



Processors in a stream

- A stream can have processors between the source and the sink
- A processor receives a message, performs some kind of processing, and emits another message
- A processor is also a standalone Spring Integration context



Processor example: JSON File Filtering

- Write only files with country code of “CA”

```
xd:>stream create --name poller --definition  
"file --dir=/tmp --pattern=*.json --outputType=text/plain  
| filter --expression=  
#jsonPath(payload,'$.origin.country').equals('CA')  
| outfile:file --dir=/tmp --mode=APPEND"  
--deploy
```

“file” source to pick up
JSON files

“filter” processor
to filter
with JSON Path

“file” sink



See: Spring XD Reference - Filter

<http://docs.spring.io/spring-xd/docs/current/reference/html/#filter>

JSON File Filtering Example – Input Files

```
{  
  "origin": {  
    "id": 3372,  
    "country": "CA"  
  }  
}
```

countries.json

```
{  
  "origin": {  
    "id": 3373,  
    "country": "US"  
  }  
}
```

countries-2.json

```
{  
  "origin": {  
    "id": 3374,  
    "country": "CA"  
  }  
}
```

countries-1.json

JSON File Filtering Example – Output File

```
{  
    "origin": {  
        "id": 3372,  
        "country": "CA"  
    }  
}  
{  
    "origin": {  
        "id": 3374,  
        "country": "CA"  
    }  
}
```

/tmp/poller.out

Quiz: Vocabulary Test

- Is a sink for data in or data out?
 - *Answer:* Sink is for data out
- Is a source for data in or data out?
 - *Answer:* Source is for data in
- What is the character used to connect sources, processors, and sinks?
 - *Answer:* “|” (pipe)
- What is the syntax of a stream definition?
 - *Answer:*
`source | processor1 | processor2 | sink`

Quiz: Stream Creation

- Does the stream run after the following command?

```
stream create --name demo --definition "http | file"
```

- No, it needs the **--deploy** parameter:

```
stream create --name demo --definition "http | file" --deploy
```

- How to specify the port to listen on?

```
stream create --name demo --definition "http | file"
```

- Use the **--port** parameter in the definition

```
stream create --name demo --definition "http --port=8080 | file"
```

Quiz: What do these Streams do?

```
http --port=9090 | log
```

- Listens on port 9090 and logs HTTP requests

```
http --port=9090 | transform --expression='*** '+payload+' ***' | log
```

- Listens on port 9090, decorates the message content with ***, and logs the decorated content

```
tcp --port=1234 --outputType=text/plain | filter  
--expression=payload.toString().contains('Hello') | file
```

- Listens on port 1234, let pass messages containing “Hello”, and write messages in a file

Summary

- Streams are composed of source, processor, and sink modules
- Spring XD comes with several pre-defined modules
- Each module in a stream can be configured by specifying options at the time of creation
- The XD shell is used to manage the stream lifecycle

Lab

XD Lab part 2 : streams

Spring XD

Batch jobs

Deploying batch jobs on Spring XD

Topics in this session

- **Batch jobs**
- Lab, part 3
- Advanced Topics

Overview

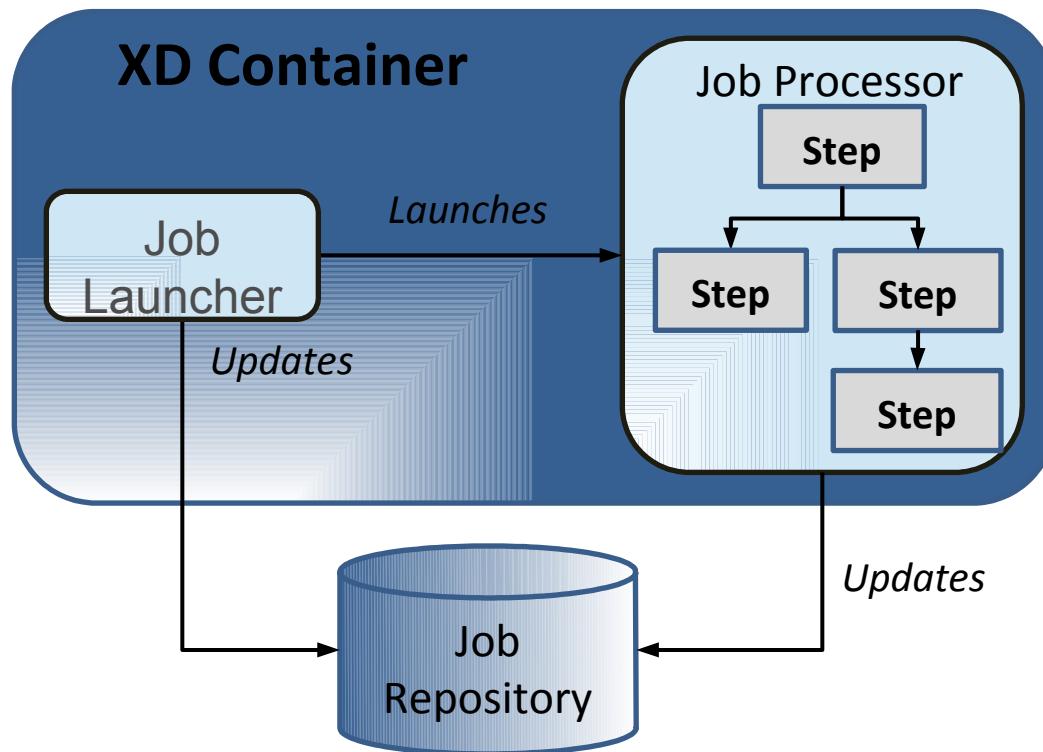
- Spring XD offers the ability to launch and monitor batch jobs based on Spring Batch
- Spring XD builds upon Spring Batch to simplify creating batch workflow solutions



See: **Spring Batch Reference**

<http://docs.spring.io/spring-batch/reference/htmlsingle/>

Basic Components of a Workflow



Batch Job Features

- Spring XD allows you to create and launch jobs
- Launching can be triggered with a scheduler (fixed delay / date / cron expression) or in reaction to data on a stream
- Spring XD provides simple, pre-defined Jobs:
 - Poll a Directory and import CSV files to HDFS
 - Import CSV files to JDBC
 - HDFS to JDBC Export
 - JDBC to HDFS Import
 - HDFS to MongoDB Export
 - ...

The Lifecycle of a Job in Spring XD

1. Register a Job Module
2. Create a Job Definition
3. Deploy a Job
4. Launch a Job
5. Job Execution
6. Un-deploy a Job
7. Destroy a Job Definition

1. Register a Job Module

- Necessary when defining custom job
- Spring XD Module Registry is configured to search for modules in the following locations, in order:
 - File path specified in `servers.yml` by `xd.module.home`
 - Default is `${xd.home}/modules`
 - `classpath:/modules/`
 - Empty by default
 - File path specified in `servers.yml` by `xd.customModule.home`
 - Default is `${xd.home}/custom-modules`

1. Register a Job Module

- Register a Job Module **from jar** with the Module Registry by using the **module upload** command.

```
xd:>module upload --type job --file myjob.jar --name xd-job-basic-2  
Successfully uploaded module 'job:xd-job-basic-2'
```

- Register a Job Module **manually**
place your xml configuration file to directory
 `${xd.customModule.home}/job/module_name/config/`

your xml must be in config dir (in jar too)

2. Create a Job Definition

- Create a Job Definition from a Job Module
- Provide definition name and properties that apply to all Job Instances

```
xd:>job create --name myBatchJob --definition "simple_example"
Successfully created job 'myBatchJob'
```

Job Name

Definition Name
(name of module folder)

- At this point the job is not deployed, yet

3. Deploy a Job

- Deploy the Job Definition to one or more Spring XD containers
- This will initialize the Job Definitions on those containers
- The jobs are now "live" (not launched)
 - Job can be launched by sending a message to a job queue that contains optional runtime Job Parameters

Deploy the Job

Job Name

```
xd:>job deploy myBatchJob  
Deployed job 'myBatchJob'
```

4. Launch a job

- There are 3 ways to launch a batch job in Spring XD:
 - Ad-hoc
 - Launch a job via command: `job launch`.
 - Job will run to completion and finish.

```
xd:> job launch helloSpringXD
```

- Use a Trigger (fixed delay, date or cron expression - see advanced section)
- As a sink from a stream (see advanced section)

5. Monitoring a Job (Job Execution)

- A discrete attempt at executing a Job Instance is a ***Job Execution***
- Job Execution object captures success or failure of the Job Instance
- You can query for Job Executions associated with a given job name

6. Undeploy Job

- Removes job from Spring XD containers
- Prevents launching of new job instances
- Keeps job definition available for future use

Undeploy
the Job

```
xd:>job list
  Job Name      Job Definition      Status
  -----
  myBatchJob    simple_example      deployed
Job Name

xd:>job undeploy myBatchJob
Un-deployed Job 'myBatchJob'
xd:>job list
  Job Name      Job Definition      Status
  -----
  myBatchJob    simple_example      undeployed
```

7. Destroy a Job Definition

- Destroying a Job Definition will
 - Un-deploy the Job (as needed)
 - Remove the Job Definition itself
- Note: The Job XML still exists in the modules/job directory.

Destroy the Job

Job Name

```
xd:>job destroy myBatchJob
Destroyed job 'myBatchJob'
xd:>job list
  Job Name  Job Definition  Status
  -----  -----  -----
```

Job : Pre-Packaged Batch

- Spring XD comes with several batch import and export modules
- Run them as-is or use them as a basis for building your own custom modules:
 - CSV Files to HDFS Import (filepollhdfs)
 - CSV Files to JDBC Import (filejdbc)
 - HDFS to JDBC Export (hdfsjdbc)
 - JDBC to HDFS Import (jdbchdfs)
 - HDFS to MongoDB Export (hdfsmongodb)
 - FTP to HDFS Export (ftphdfs)



See: **Spring XD Reference - Jobs**

<http://docs.spring.io/spring-xd/docs/current/reference/html/#jobs>

Summary

- Spring XD simplifies the creation of batch workflow jobs
- Manages job lifecycle
- 3 ways to launch a batch job in Spring XD
 - What are they?

Lab

XD Lab part 3 : jobs

Topics in this session

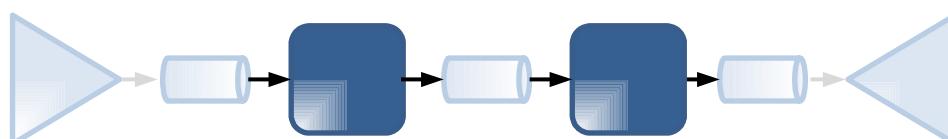
- Batch jobs
- Lab, part 3
- **Advanced Topics**

Stream : Custom Processing Modules

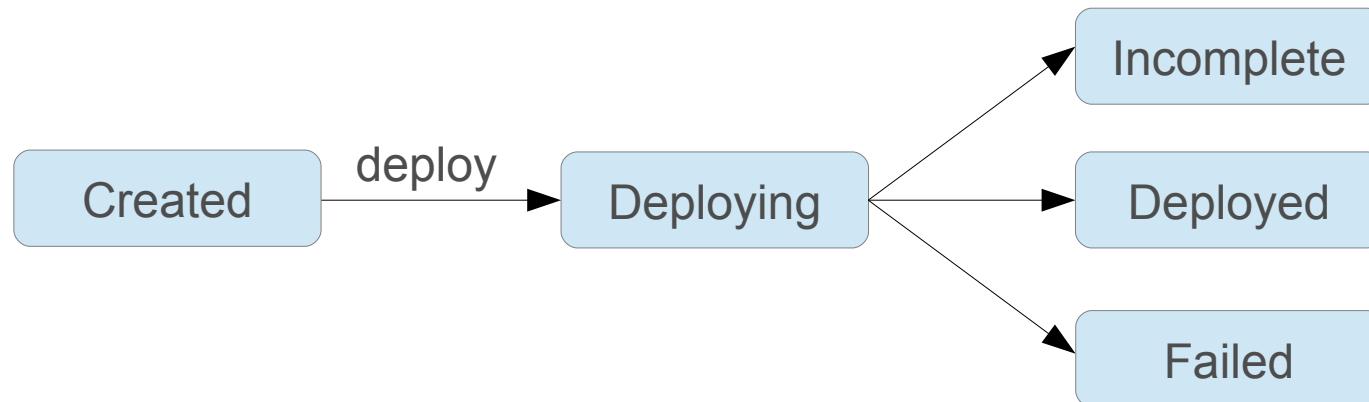
- If not using SpEL or Groovy, then can write custom processors in Java
- Central concept is Message Handler class
 - Relies on coding conventions to map inbound messages to processing methods

```
public class SimpleProcessor {  
    public String process(String payload) {  
        return payload.toUpperCase();  
    }  
}
```

Input **Output**



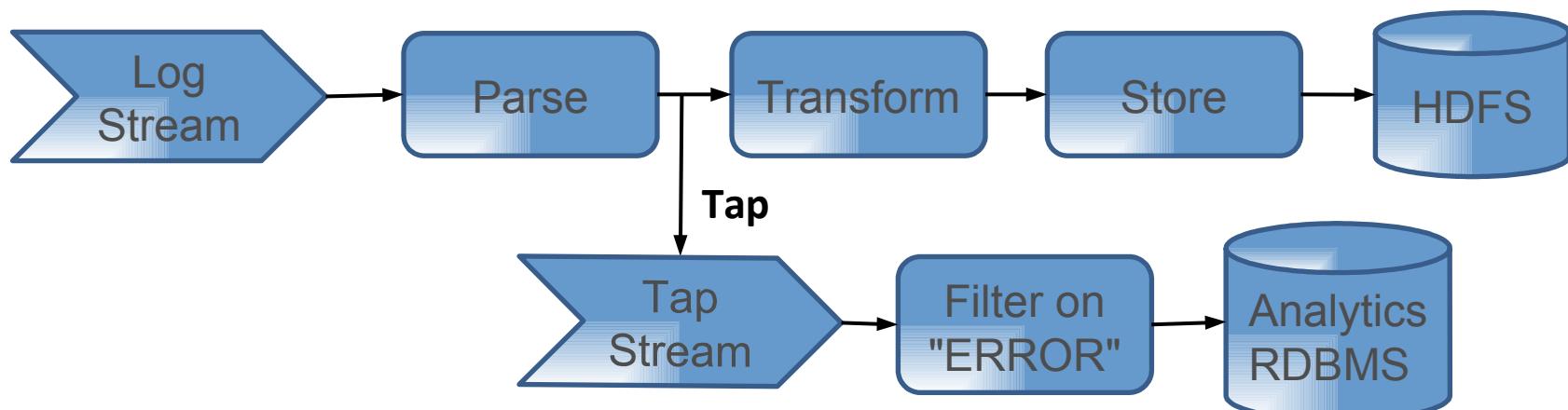
Stream : Deployment States



- **Created:** Stream created but not deployed
- **Deploying:** Deployment in progress
- **Incomplete:** At least one instance of each module was deployed successfully
- **Deployed:** All modules deployed successfully
- **Failed:** At least one module was not deployed

Taps

- Taps are used to intercept data within a stream or job
 - And pass to another stream
- “Listen”, without modifying
 - Analogous to a phone wiretap
- Can use any point in another stream as a source



Taps

- To specify point in stream to tap, use
<stream_name>.<module_name>
 - Tapped directly *after* the module specified
 - If <module_name> is omitted then stream is tapped at source
 - Can also use an alias for <module_name>
- Tap is not deleted when stream is destroyed
 - Tap is actually a separate stream
- Can also tap other XD targets such as jobs



See: **Spring XD Reference - Taps**

<http://docs.spring.io/spring-xd/docs/current/reference/html/#taps>

Tap Example

- First, create a stream
 - Example: **timelogger** logs every 10 seconds.

```
xd:>stream create --name timeLogger --definition  
"time | filter --expression=payload.endsWith('0') | log"  
--deploy  
Created and deployed new stream 'timeLogger'
```

```
23:30:50 1.2.1.RELEASE INFO...sink.timeLogger - 2015-08-17 23:30:50  
23:31:00 1.2.1.RELEASE INFO...sink.timeLogger - 2015-08-17 23:31:00  
23:31:10 1.2.1.RELEASE INFO...sink.timeLogger - 2015-08-17 23:31:10  
23:31:20 1.2.1.RELEASE INFO...sink.timeLogger - 2015-08-17 23:31:20
```

Job : Launch the Batch using Cron-Trigger

- Launch a Job Instance based on a schedule
 - Create a stream with **trigger --cron** source:

```
stream create --name cronStream --definition "trigger  
--cron='0/5 * * * *'" > queue:job:myCronJob"
```

- Job Instance can receive parameters from a source (in this case a trigger) or process
- A trigger uses **--payload** option to declare its payload

```
stream create --name cronStream --definition "trigger  
--cron='0/5 * * * *'  
--payload={"param1":"Clarence"}  
> queue:job:myCronJob"
```

Job : Using Fixed-Delay-Trigger

- A fixed-delay-trigger launches a Job on a regular interval
- **--fixedDelay:** seconds between executions
- Example: launch myXDJob instance every 5 seconds and pass a payload with a single parameter -

```
stream create
--name fdStream
--definition "trigger
  --fixedDelay=5
  --payload={"param1":"holiday"}
> queue:job:myXDJob"
```

Job : Retrieve Notifications

- Spring XD captures notifications sent from executing jobs.
- When a batch job is deployed, by default it registers the following listeners along with pub/sub channels that these listeners send messages to:
 - Job Execution Listener
 - Step Execution Listener
 - Chunk Listener
 - Item Listener
 - Skip Listener

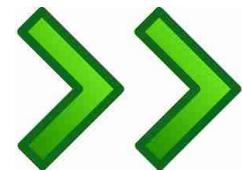
Finishing Up

Course Completed

What's Next?

What's Next

- Congratulations, we've finished the course
- What to do next?
 - Certification
 - Other courses
 - Resources
 - Evaluation
- Check-out optional sections on Remoting, SOAP Web Services, HATEOAS, Spring Data Rest and AMQP



Certification



- Computer-based exam
 - 50 multiple-choice questions
 - 90 minutes
 - Passing score: 76% (38 questions answered successfully)
- Preparation
 - See Enterprise Spring certification guide
 - <http://pivotal.io/training>
 - Review all the slides
 - Redo the labs

Certification: Questions

Typical question:

- Which statement best describes REST features in Spring?
(select one)
 - Spring MVC provides its own JAX-RS implementation
 - Spring MVC's REST features may rely on any JAX-RS implementation
 - Spring MVC does not provide REST features
 - Spring MVC's REST features do not rely on the JAX-RS standard

Certification: Logistics

- Where?
 - At any Pearson VUE Test Center
 - Most large or medium-sized cities
 - See <http://www.pearsonvue.com/vtclocator>
- How?
 - At the end of the class, you will receive a certification voucher by email
 - Make an appointment
 - Give them the voucher when you take the test
- For any further inquiry, you can write to
 - education@pivotal.io

Other courses



- Many courses available
 - Core Spring
 - Spring Web
 - JPA with Spring
 - What's New In Spring?
 - Cloud Foundry
 - Gemfire
 - Processing Big Data with Hadoop and Pivotal HD
 - Data Analytics
- See <http://www.pivotal.io/training>

What's New in Spring?

- Three day course covering new features of Spring 3.x and 4.x
 - Annotation and Java based configuration
 - Servlet 3.x
 - Spring Security
 - Spring Data
 - Spring 4 and Java 8
 - Spring Boot
- Ensure your Spring skills are up-to-date

Spring Web

- Four-day workshop
- Making the most of Spring in the web layer
 - Spring MVC
 - REST using MVC and AJAX
 - Security of Web applications
 - Mock MVC testing framework
 - Spring Boot, Web Sockets
- Spring Web Application Developer certification

JPA with Spring

- Three day course covering
 - Using JPA with Spring and Spring Transactions
 - Implement inheritance and relationships with JPA
 - Discover how JPA manages objects
 - Go more in depth on locking with JPA
 - Advanced features such as interceptors, caching and batch updates
 - Using Hibernate as the JPA provider

Processing Big Data with Hadoop and Pivotal HD

- Four day course
 - Big/Fast data, NoSQL, and their role in modern Business applications
 - Background on Hadoop and the significance of the Pivotal HD distribution
 - Introduction to GemFire and SQLFire, distributed cache technologies

Developing Applications with Cloud Foundry

- Three day course covering
 - Application deployment to Cloud Foundry
 - Cloud Foundry Concepts
 - Deployment using cf tool or an IDE
 - Accessing and defining Services
 - Using and customizing Buildpacks
 - Design considerations: “12 Factor”
 - JVM Application specifics, using Spring Cloud
 - Microservices



Pivotal Support Offerings

- Global organization provides 24x7 support
 - How to Register: <http://tinyurl.com/piv-support>
- Premium and Developer support offerings:
 - <http://www.pivotal.io/support/offering>
 - <http://www.pivotal.io/support/oss>
 - Both Pivotal App Suite *and* Open Source products
- Support Portal: <https://support.pivotal.io>
 - Community forums, Knowledge Base, Product documents



Pivotal Consulting

- Custom consulting engagement?
 - Contact us to arrange it
 - <http://www.pivotal.io/contact/spring-support>
 - Even if you don't have a support contract!
- Pivotal Labs
 - Agile development experts
 - Assist with design, development and product management
 - <http://www.pivotal.io/agile>
 - <http://pivotallabs.com>



Resources

- The Spring reference documentation
 - <http://spring.io/docs>
 - <http://spring.io/projects>
- The official technical blog
 - <http://spring.io/blog>
- Stack Overflow – Active Spring Forums
 - <http://stackoverflow.com>



SPRING INTEGRATION



SPRING BATCH

Thank You!

We hope you enjoyed the course

Please fill out the evaluation form

Asia-Pac: <http://tinyurl.com/pivotalAPACeval>

MyLearn: <http://tinyurl.com/mylearneval>



Spring AMQP

Simplifying Messaging Applications

Topics in this Session

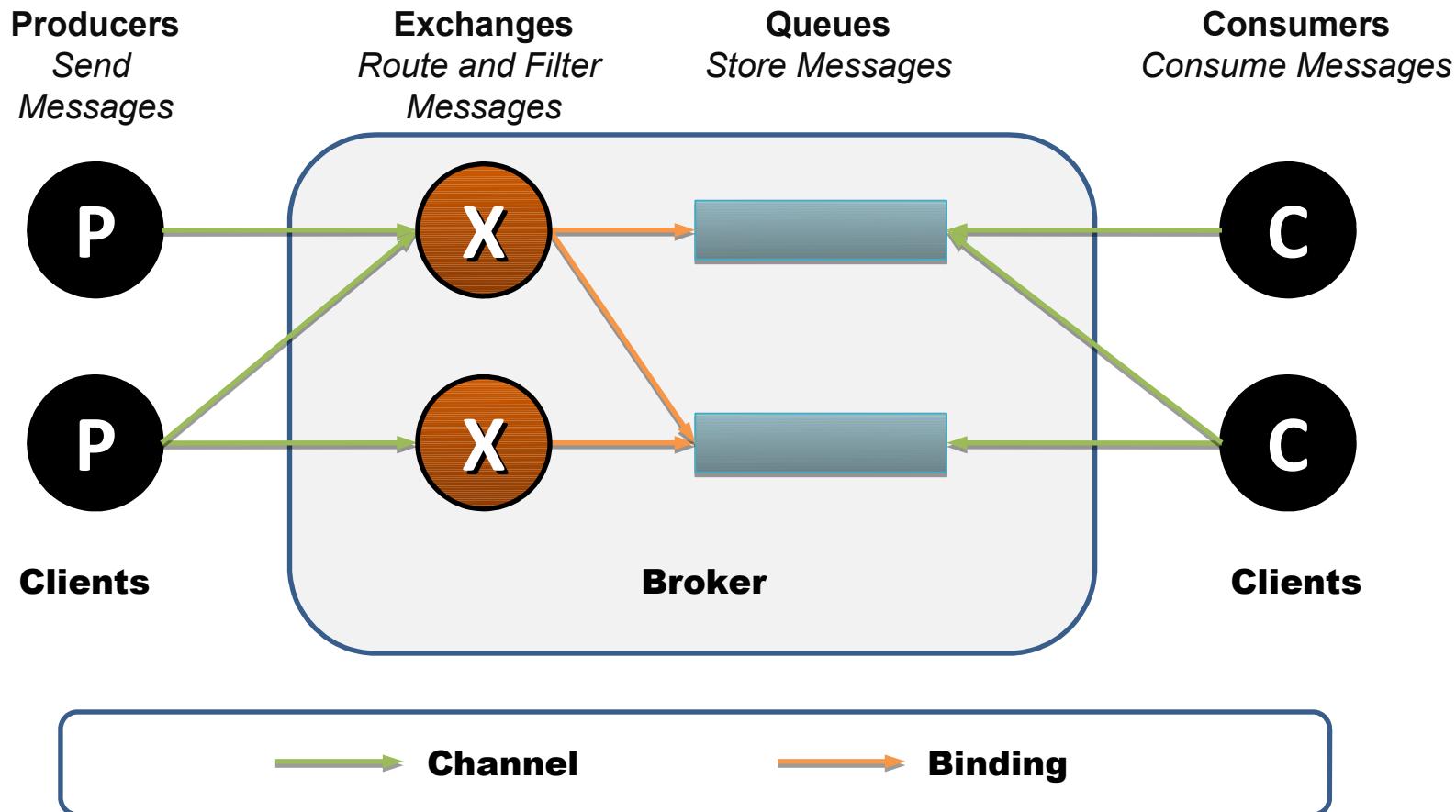
- **Introduction to AMQP**
- RabbitMQ
- Configuring AMQP Resources with Spring
- Spring's AmqpTemplate
- Sending Messages
- Receiving Messages

Advanced Message Queuing Protocol

AMQP

- AMQP is a messaging protocol that is compatible at the binary (or wire) level
 - Avoid vendor lock-in
 - Increase portability
- AMQP enables different Message-Oriented Middleware (MOM) vendors to communicate
 - Bridge is not required

The AMQP Model



AMQP Terminology

- **Queue**

queue

- A container for holding messages
- The first message placed into the container is the first message that can be removed (FIFO).

- **Exchange**



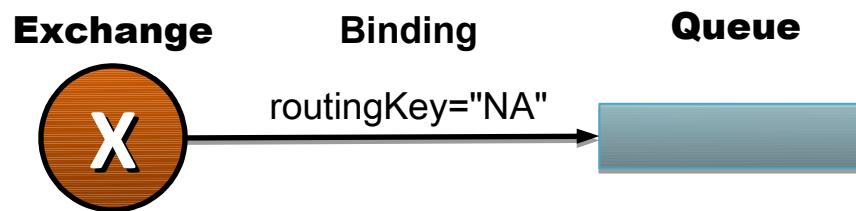
- Exchanges are responsible for applying routing rules to messages in order to deliver them to the correct queue(s) for consumption by consumers.
 - 4 types: direct, fanout, topic, and headers
 - The type of exchange, in combination with message headers, determine routing

AMQP Terminology

- **Producer (or Publisher)**
 - Client that creates and sends messages. Producers always publish messages to exchanges.
- **Consumer (or Listener, Subscriber)**
 - Client that consumes and processes messages
 - Consumers always consume messages from queues.
 - Can poll queue synchronously (blocking), or
 - Can register an asynchronous callback (non-blocking, preferred)

AMQP Terminology

- **Binding**
 - Bindings are rules that exchanges use to route messages to queues, or to other exchanges
 - The type of exchange will dictate which message header properties are considered for the binding
 - For example, direct exchanges have bindings that route messages based on the *routingKey* message header property



AMQP Terminology

- **Broker**
 - Brokers are AMQP servers that manage all AMQP resources
 - *Exchanges, Queues, Bindings, etc.*
 - *RabbitMQ* is a an example of a Broker.

AMQP Terminology

- **Connection**

- A connection is a TCP socket between a client and a broker.
- An AMQP Connection is obtained from a factory

```
ConnectionFactory factory = new ConnectionFactory();
factory.setUsername("guest");
factory.setPassword("guest");
factory.setVirtualHost("/");
factory.setHost("localhost");

int serverPort = 5672;
factory.setPort(serverPort);
Connection connection = factory.newConnection();
```

AMQP Terminology

- **Channel**
 - The means through which multiple parallel operations can be performed over a single connection.
 - Multiple channels are multiplexed over a single connection.
- Channels can be shared by multiple threads
 - Care must be taken that only a *single* thread executes a command at a given point in time

```
Channel channel = connection.createChannel();
```

AMQP Messages

- **AMQP Messages** consist of:
 - Header properties (i.e. key-value pairs)
 - Body
- Some pre-defined header properties provide instructions to the broker on how to handle the message.
 - For example, "*routingKey*" is typically used for routing.
 - "*deliveryMode*" tells the broker whether to persist the message to disk.

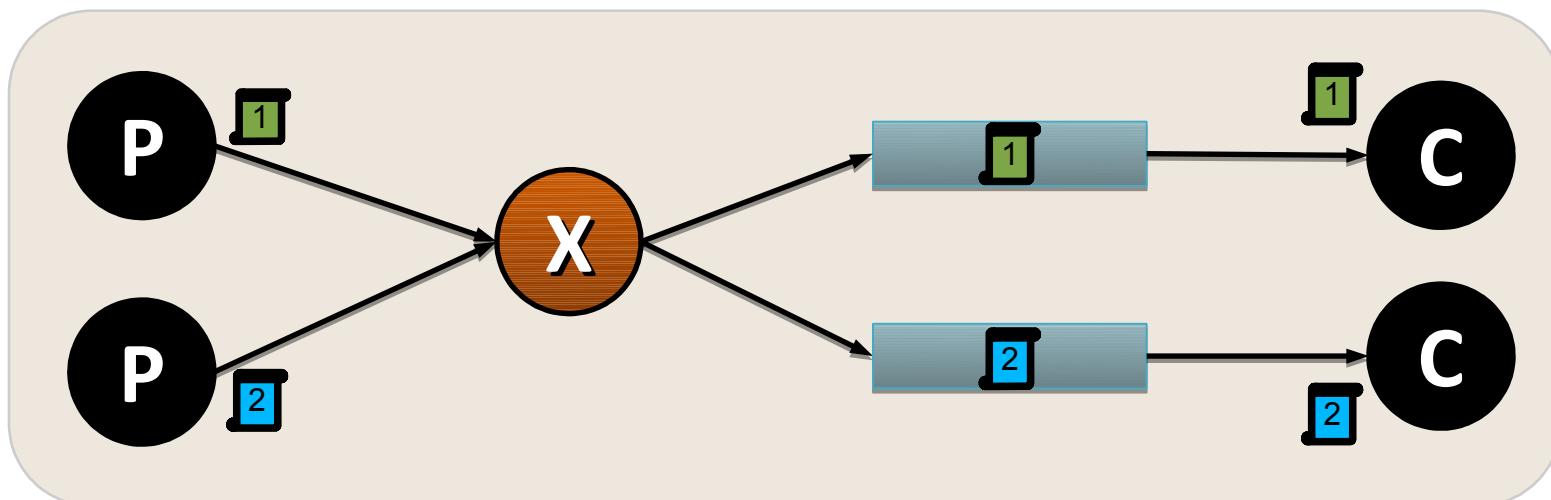
AMQP Messages

- Message body is binary – a simple byte array
- Separate parameters for the header properties and the message payload are provided in the Java client API

```
channel.basicPublish(exchange,  
    routingKey,  
    new AMQP.BasicProperties.Builder()  
        .deliveryMode(2).build()),  
    msgBodyBytes);
```

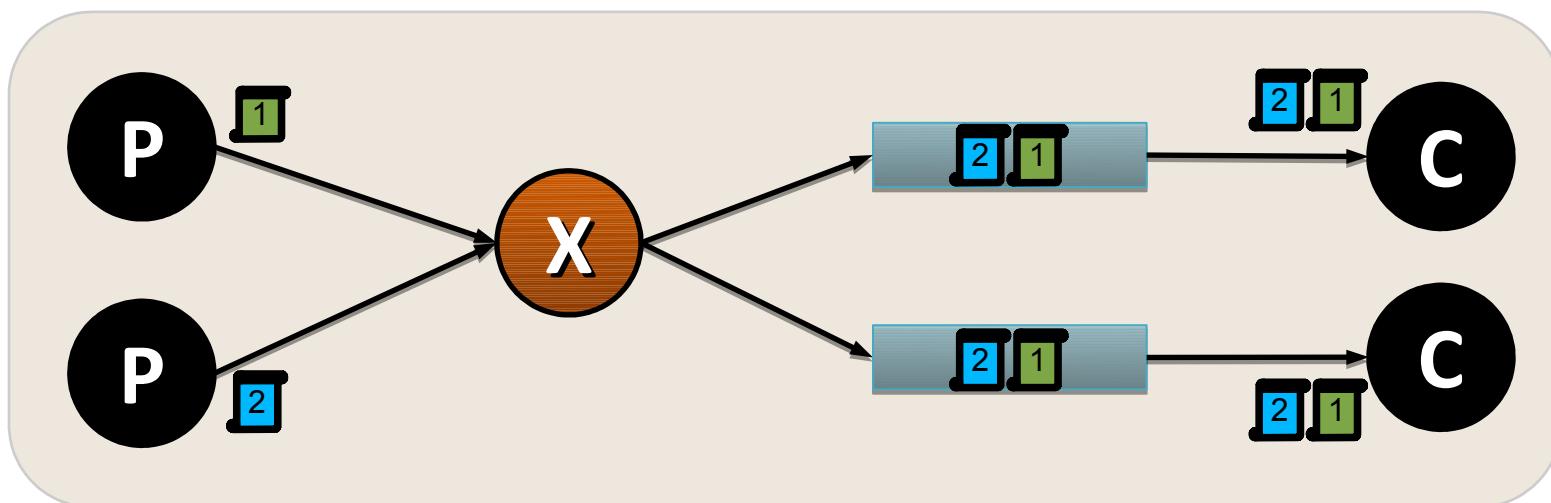
Point-to-Point Pattern

1. Message sent to exchange (eg. "direct", not a fanout exchange)
2. Routed to queue based on binding
3. Message queued
4. Message consumed by one and only one consumer



Publish-Subscribe Pattern

1. Message sent to fanout exchange
2. Routed to all queues bound to the exchange
3. Message queued
4. Message consumed by multiple consumers – each gets a copy of the message.



Topics in this Session

- Introduction to AMQP
- **RabbitMQ**
- Configuring AMQP Resources with Spring
- Spring's AmqpTemplate
- Sending Messages
- Receiving Messages

AMQP Compliant Brokers

- Most common ones are:
 - Pivotal's RabbitMQ
 - Apache Qpid
- We will use RabbitMQ!

RabbitMQ

- Open source AMQP compliant message broker written in Erlang
- Officially supported clients:
 - Java, .NET, Erlang, JMS (commercial only)
- Many third party OSS clients:
 - C, Python, Ruby, PHP, Perl, etc.

RabbitMQ Features

- Clustering for High Availability
- Mirrored queues for guaranteed messaging
- Shovel and Federation plugins for WAN replication
- LDAP Authentication plugin
- Web-based management console plugin
- Management REST API for monitoring
- Extensible through custom plugins
- Comprehensive list at
<http://www.rabbitmq.com/features.html>

Topics in this Session

- Introduction to AMQP
- RabbitMQ
- **Configuring AMQP Resources with Spring**
- Spring's AmqpTemplate
- Sending Messages
- Receiving Messages

Configuring AMQP Resources with Spring

- Spring enables decoupling of your application code from the underlying infrastructure
 - Container provides the resources
 - Application is simply coded using POJO's
- Provides and abstracts from the developer much of the tedious boilerplate code for things such as connection retries, transactions, etc.

Configuring a ConnectionFactory

- Typically use the *CachingConnectionFactory*
 - Caches channels and connections for reuse
 - Use carefully, see documentation:
 - <http://docs.spring.io/spring-amqp/reference/html/amqp.html#connections>

```
<bean id="connectionFactory"
      class="org.springframework.amqp.
          rabbit.connection.CachingConnectionFactory">
    <constructor-arg value="somehost"/>
    <property name="username" value="guest"/>
    <property name="password" value="guest"/>
</bean>
```

Topics in this Session

- Introduction to AMQP
- RabbitMQ
- Configuring AMQP Resources with Spring
- **Spring's AmqpTemplate**
- Sending Messages
- Receiving Messages

Spring's AmqpTemplate

- *AmqpTemplate*
 - interface that defines the main operations, such as sending and receiving messages.
- *RabbitTemplate*
 - currently the only implementation of the *AmqpTemplate*
 - Specifically depends on the RabbitMQ Java-client.

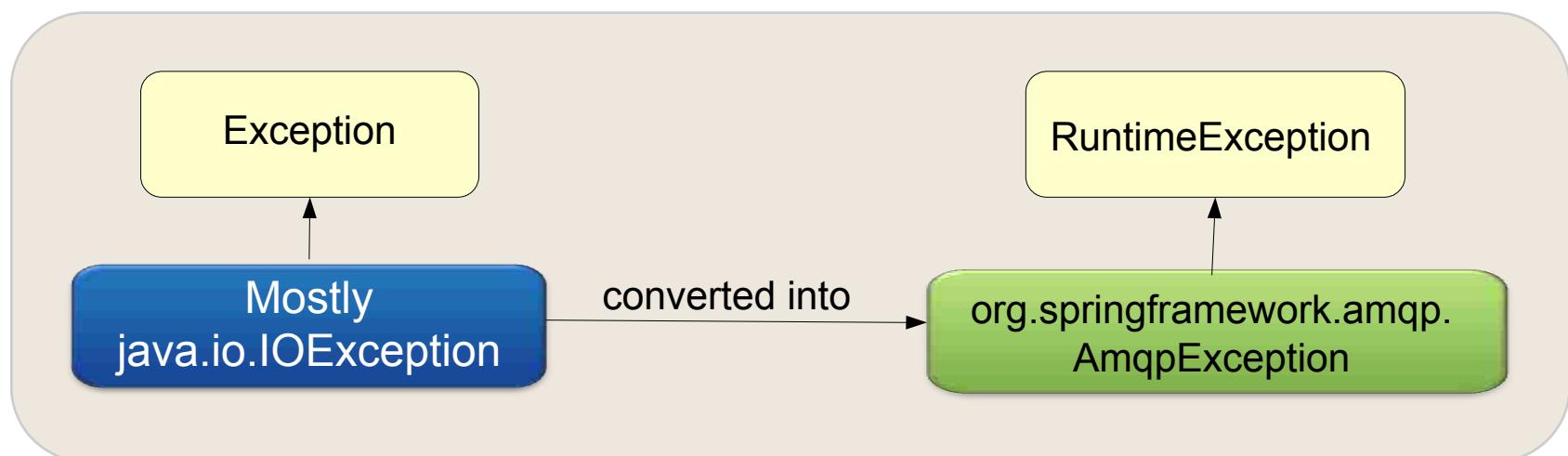
Spring's AmqpTemplate

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

NOTE: The *JmsTemplate* (used with JMS) has an almost identical API to the *AmqpTemplate* – they offer similar abstractions over different products

Exception Handling

- Many exceptions in AMQP are checked by default
- *AmqpTemplate* converts checked exceptions to runtime equivalents



RabbitTemplate Configuration

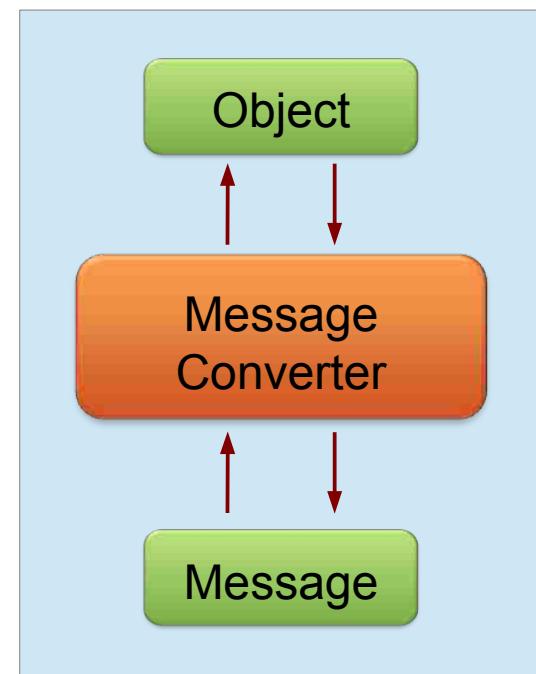
- Must provide reference to *ConnectionFactory*
 - via either constructor or setter injection
- Optionally provide delegates to handle some of the work
 - *MessageConverter* (next slide)

```
<bean id="rabbitTemplate"
      class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <property name="connectionFactory" ref="rabbitConnectionFactory"/>
    <property name="messageConverter" ref="jsonConverter" />
</bean>
```

MessageConverter

- The `AmqpTemplate` uses a `MessageConverter` to convert between objects and messages
 - You only send and receive objects
- The default `SimpleMessageConverter` handles basic types
 - `String` to `TextMessage`
 - `byte[]` to `BytesMessage`
 - `Serializable` to `ObjectMessage`

NOTE: It is possible to implement custom converters by implementing the `MessageConverter` interface



JSON MessageConverter

- JSON is a common message payload
- *JsonMessageConverter* is one implementation that is available

```
<bean class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <property name="connectionFactory" ref="rabbitConnectionFactory"/>
    <property name="messageConverter" ref="jsonConverter" />
</bean>

<bean id="jsonConverter"
      class="org.springframework.
              amqp.support.converter.JsonMessageConverter">
</bean>
```

Marshalling MessageConverter

```
<bean id="rabbitTemplate"
      class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <property name="connectionFactory" ref="rabbitConnectionFactory"/>
    <property name="messageConverter" ref="messageConverter"/>
</bean>

<oxm:jaxb2-marshaller id="marshaller" >
    <oxm:class-to-be-bound name="com.mypackage.Order"/>
</oxm:jaxb2-marshaller>

<bean id="messageConverter"
      class="org.springframework.amqp
              support.converter.MarshallingMessageConverter" >
    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
</bean>
```

Send and receive XML messages

Class annotated with
`@XmlRootElement`

CachingConnectionFactory

- For convenience, a *CachingConnectionFactory* can also be created using the rabbit namespace
 - Note that the default cache size is 1, but can easily be overridden using the *channel-cache-size* attribute

```
<rabbit:connection-factory id="rabbitConnectionFactory"
    channel-cache-size="10"
    addresses="localhost:5672"
    virtual-host="/"
    username="guest"
    password="guest" />
```

Topics in this Session

- Introduction to AMQP
- RabbitMQ
- Configuring AMQP Resources with Spring
- Spring's AmqpTemplate
- **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 AMQP Java-client API
- Use the simplest option for the task at hand

Sending POJO

- A message can be sent in one single line

```
public class OrderManagerImpl implements OrderManager {  
    @Autowired  
    private RabbitTemplate rabbitTemplate;  
    private final String ORDERS_EXCHANGE = "Orders";  
  
    public void placeOrder(Order order) {  
        String routingKey = order.getSourceRegion();  
  
        // Use message converter  
        rabbitTemplate.convertAndSend  
            (ORDERS_EXCHANGE, routingKey, order);  
    }  
}
```

Synchronous Request-Reply

- RPC pattern using messaging
 - Convenience *convertSendAndReceive()* method deserializes reply message payload into object

```
...
String orderId = "12345";
OrderDetailsRequest req = new OrderDetailsRequest(orderId);

String exchange = "order-services";
String routingKey = "get-order-details-req.1_0";
Order order = (Order)template.convertSendAndReceive
    (exchange, routingKey, req);
...
```

Topics in this Session

- Introduction to AMQP
- RabbitMQ
- Configuring AMQP Resources with Spring
- Spring's AmqpTemplate
- Sending Messages
- **Receiving Messages**

Synchronous Message Reception

- *AmqpTemplate* can also receive messages, but methods are blocking (with optional timeout)
 - *receive()*
 - *receive(String queueName)*
- The *MessageConverter* can be leveraged for message reception as well

```
Object someSerializable1 =  
    template.receiveAndConvert();
```

```
Object someSerializable2 =  
    template.receiveAndConvert(someQueue);
```

The AMQP MessageListener

- AMQP Java Client API defines this interface for *asynchronous* reception of messages
 - Or use *ChannelAwareMessageListener*

```
import org.springframework.amqp.core.Message;
import org.springframework.amqp.
    rabbit.core.ChannelAwareMessageListener;
import com.rabbitmq.client.Channel;

public class GetOrderDetails
    implements ChannelAwareMessageListener {

    public void onMessage(Message msg, Channel channel) {
        // Process message...
    }
}
```

Spring's MessageListener Containers

- JEE Containers support *message-driven* beans
 - Implement an identical JMS *MessageListener* interface
- Spring provides a lightweight listener container
 - *SimpleMessageListenerContainer*
 - Uses AMQP Java client API
 - Defaults to one concurrent consumer
 - Supports transactions if input channel is transactional
- Advanced options to control consumers, scheduling, timeouts ...

Defining a plain AMQP Message Listener

- Define listeners using *rabbit:listener* elements

```
<rabbit:listener-container  
    connection-factory="rabbitConnectionFactory" />  
    <rabbit:listener queue-names="quote"  
        ref="quoteRequestAmqpEndpoint" />  
</rabbit:listener-container>
```

- Listener needs to implement *MessageListener*
 - or *ChannelAwareMessageListener*
- rabbit:listener-container* is configurable
 - concurrency, acknowledgement mode, transaction manager and more

Spring's Message-Driven POJO

- Spring also allows you to specify a plain Java object that can serve as a listener
 - *MessageConverter* provides parameter
 - Any return value sent to response-exchange after conversion

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

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

Messaging: Pros and Cons

- Advantages
 - Resilience, guaranteed delivery
 - Asynchronous support
 - Application freed from messaging concerns
 - Interoperable – languages, environments
- Disadvantages
 - Requires additional third-party software
 - More complex to use – but not with AmqpTemplate!

Lab

Sending and Receiving Messages in
a Spring Environment

Introduction to Spring Remoting

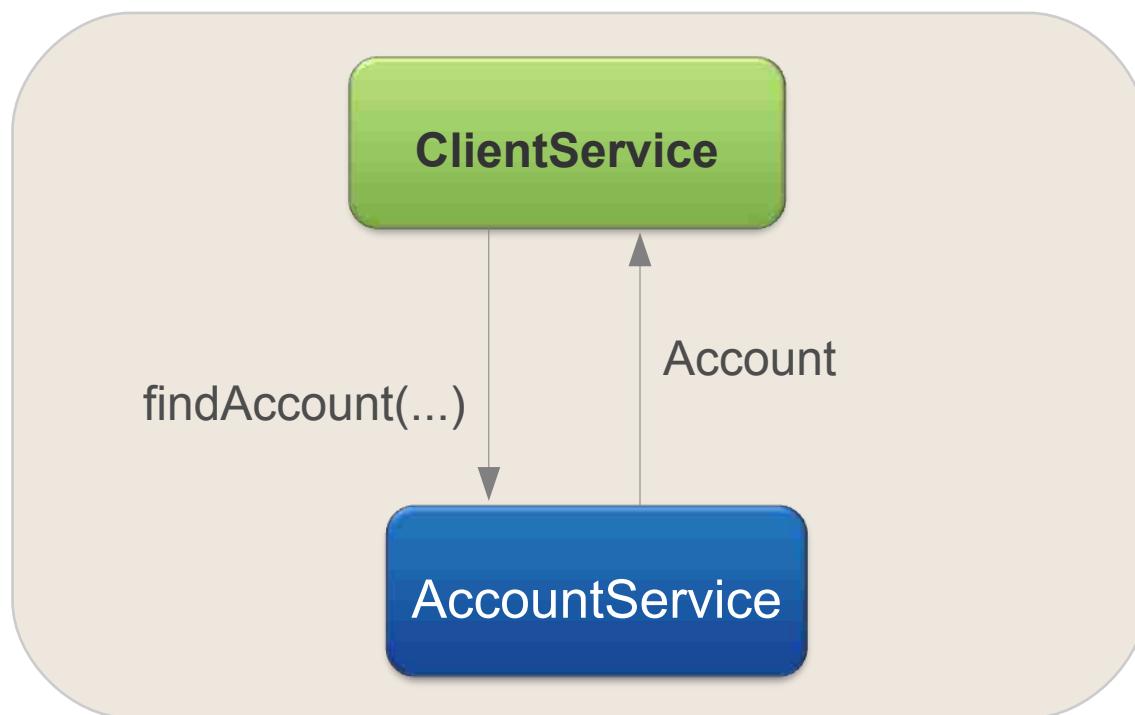
Simplifying Distributed Applications

Topics in this Session

- **Introduction to Remoting**
- Spring Remoting Overview
- Spring Remoting and RMI
- HttpInvoker
- Additional supported Protocols

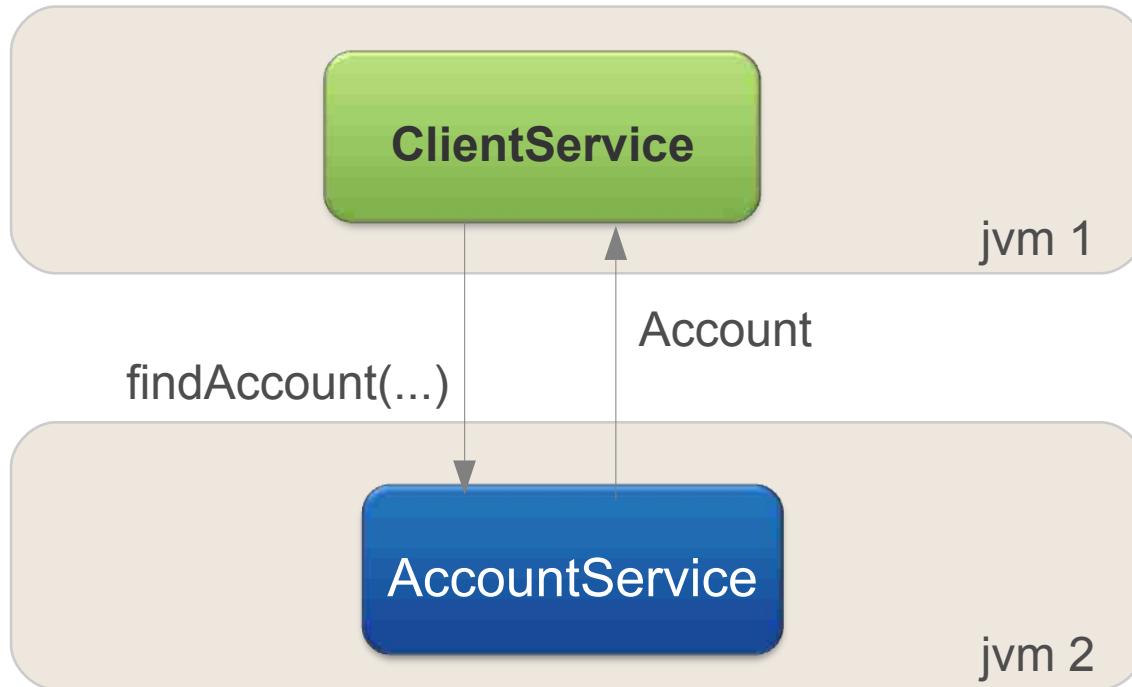
Local access

- So far, you have seen how to access objects locally



Remote access

- What if those 2 objects run in some separate JVMs?



In this module, only Java-to-Java communication is addressed
(as opposed to remote access using Web Services or JMS)

The RMI Protocol

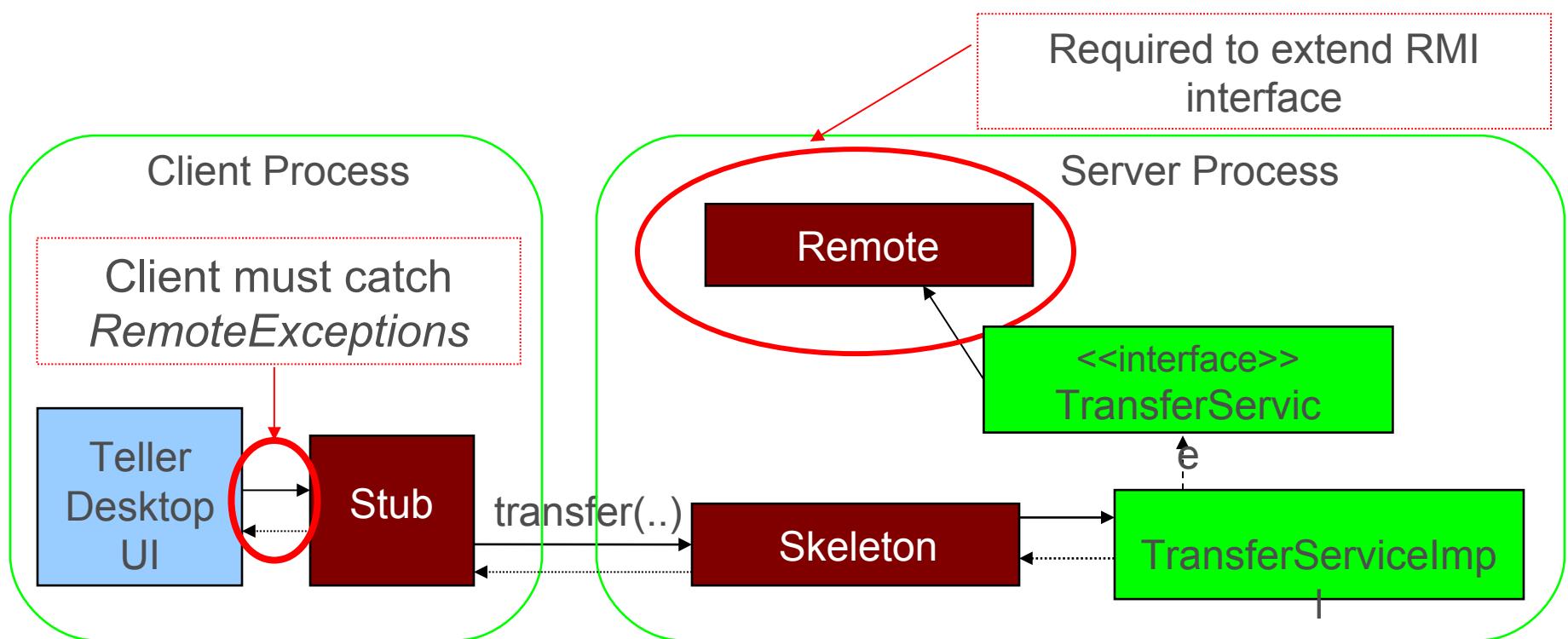
- Standard Java remoting protocol
 - Remote Method Invocation
 - Java's version of RPC
- Server-side exposes a *skeleton*
- Client-side invokes methods on a *stub* (proxy)
- Java serialization is used for marshalling

Working with plain RMI

- RMI violates separation of concerns
 - Couples business logic to remoting infrastructure
- For example, with RMI:
 - Service interface extends **Remote**
 - Client must catch **RemoteExceptions**
- Technical Java code needed for binding and retrieving objects on the RMI server

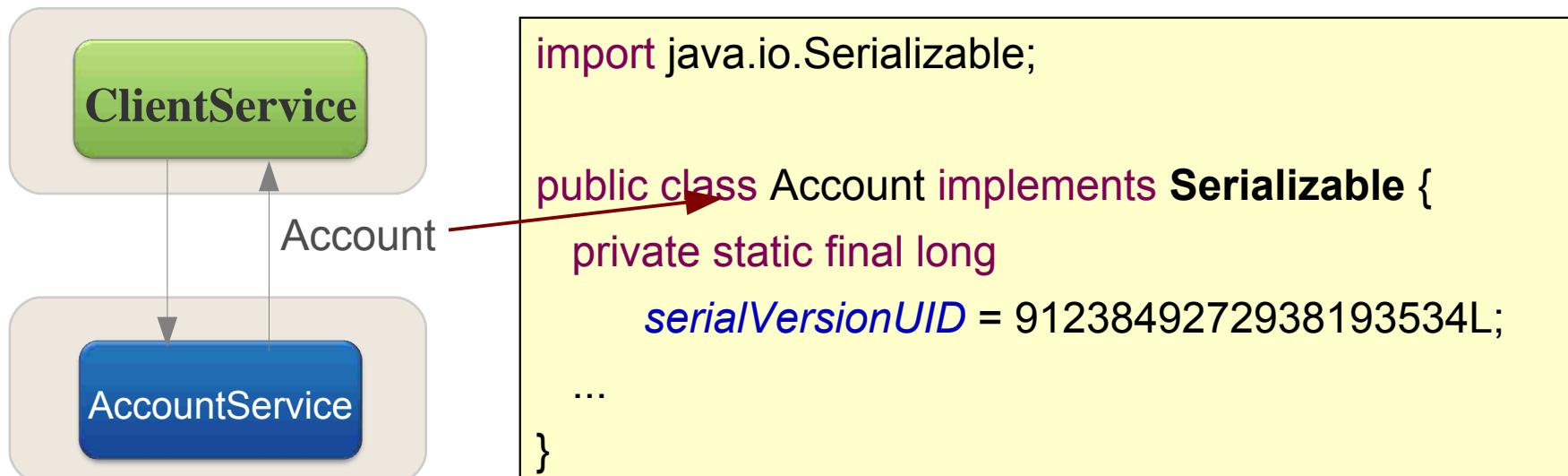
Traditional RMI

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



RMI and Serialization

- RMI relies on Object Serialization
 - Objects transferred using RMI should implement interface *Serializable*
 - Marker interface, no method to be implemented



Topics in this Session

- Introduction to Remoting
- **Spring Remoting Overview**
- Spring Remoting and RMI
- HttpInvoker
- Additional supported Protocols

Goals of Spring Remoting

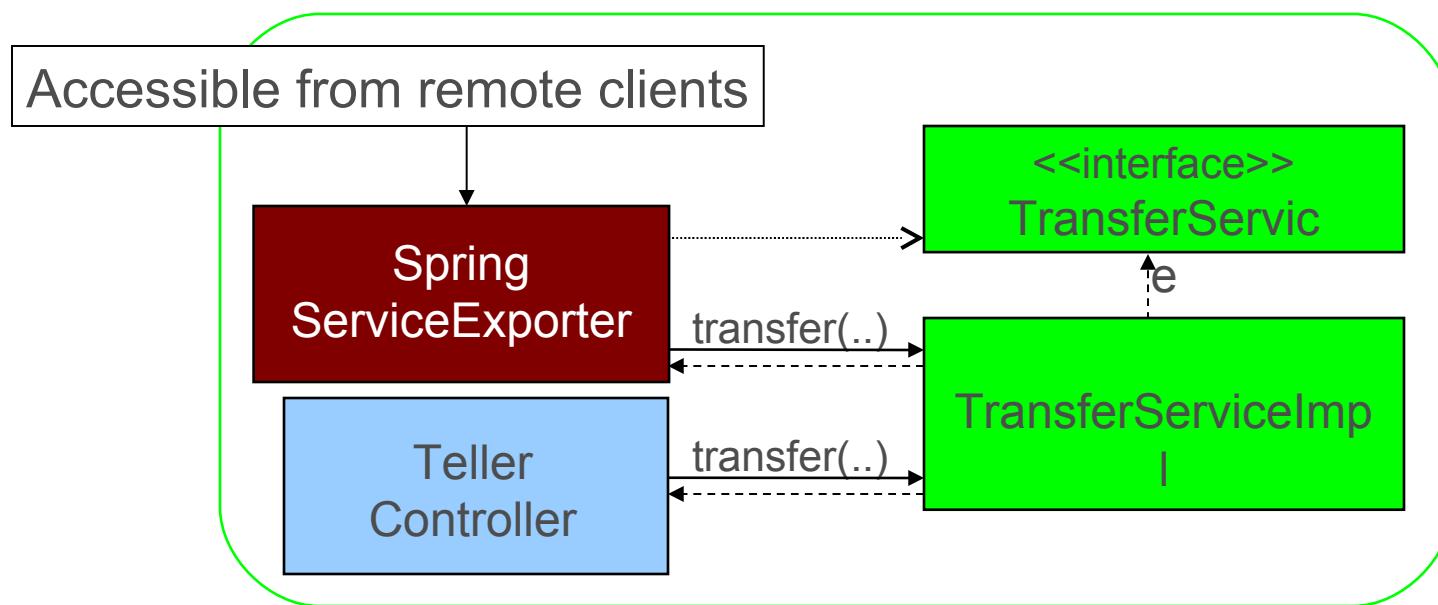
- Hide “plumbing” code
- Configure and expose services declaratively
- Support multiple protocols in a consistent way

Hide the Plumbing

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

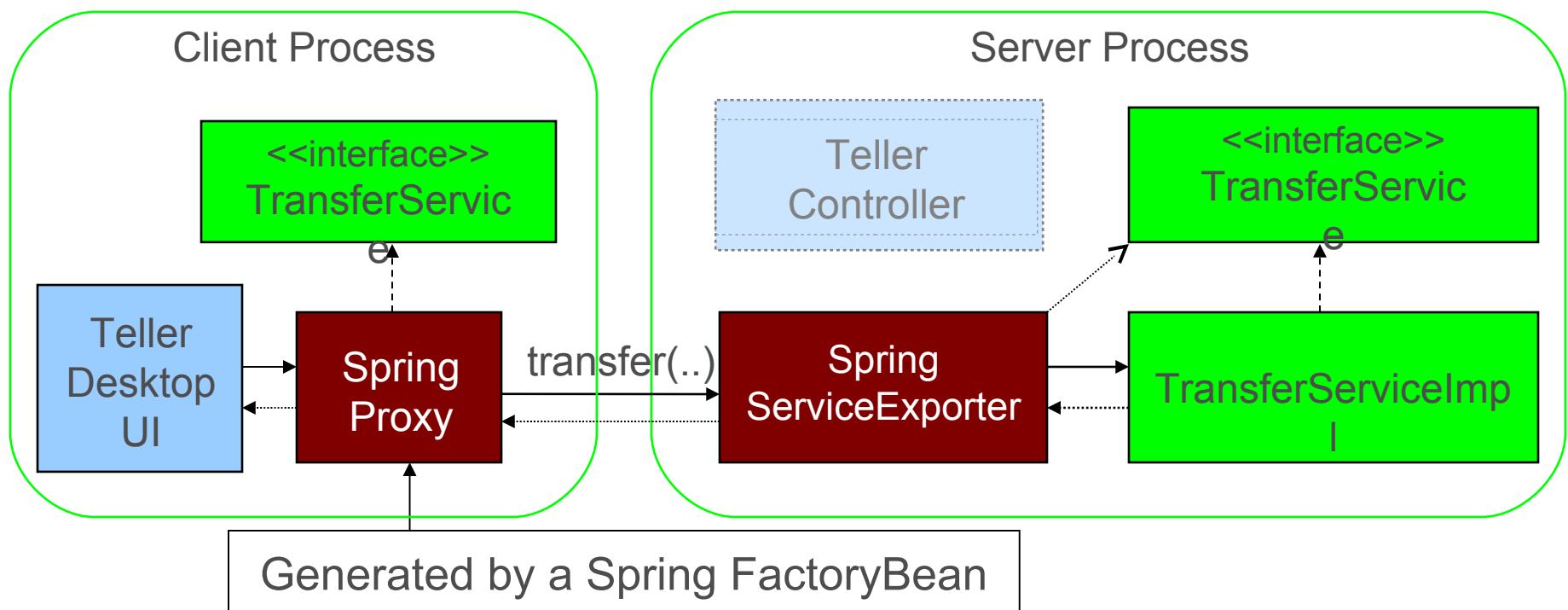
Service Exporters

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



Client Proxies

- Dynamic proxies generated by Spring communicate with the service exporter



A Declarative Approach

- Uses a configuration-based approach
 - No code to write
- On the server side
 - Expose existing services with NO code changes
- On the client side
 - Invoke remote methods from existing code
 - Take advantage of polymorphism by using dependency injection
 - Migrate between remote vs. local deployments

Consistency across Protocols

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

Topics in this Session

- Introduction to Remoting
- Spring Remoting Overview
- **Spring Remoting and RMI**
- HttpInvoker
- Additional supported Protocols

Spring's RMI Service Exporter

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



Transferred objects still need to implement the interface
java.io.Serializable

Configuring the RMI Service Exporter

Server

- Start with an existing POJO service

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

- Define a bean to export it

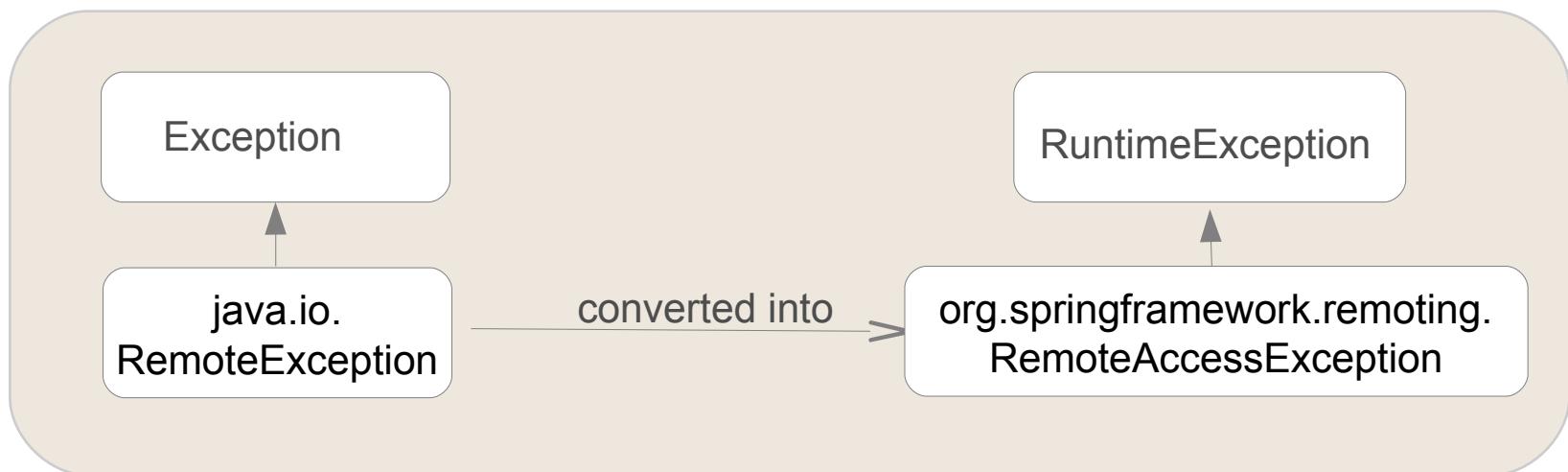
```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName" value="transferService"/>
    <property name="serviceInterface" value="app.TransferService"/>
    <property name="service" ref="transferService"/>
    <property name="registryPort" value="1096"/>
</bean>
```

“*registryPort*” defaults to “1099”

Binds to rmiRegistry as “transferService”

Spring's RMI Proxy Generator

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



Configuring the RMI Proxy

Client

- Define a factory bean to generate the proxy

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

- Inject it into the client

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

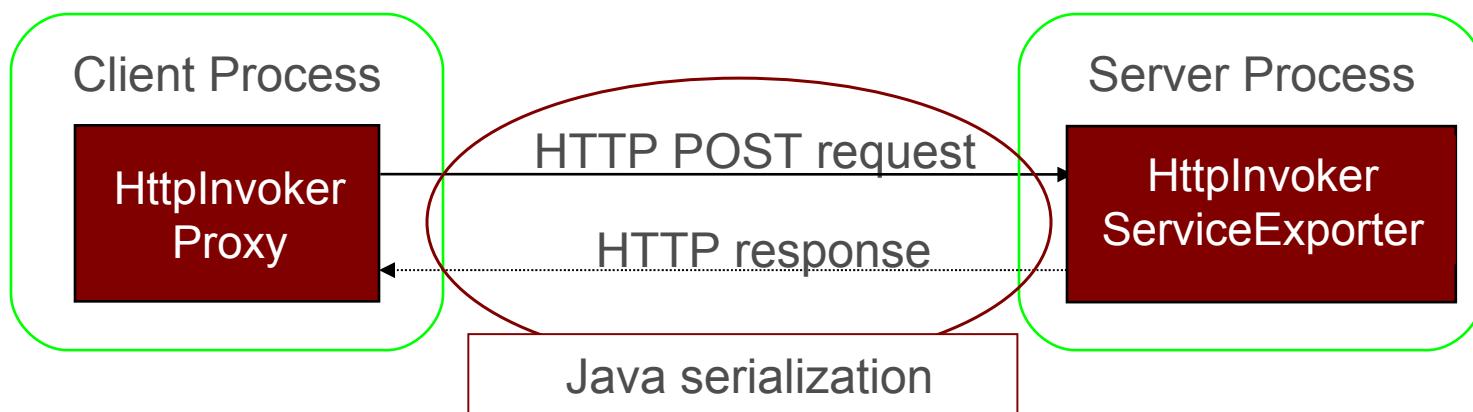
TellerDesktopUI only depends on the TransferService interface

Topics in this Session

- Introduction to Remoting
- Spring Remoting Overview
- Spring Remoting and RMI
- **HttpInvoker**
- Additional supported Protocols

Spring's HttpInvoker

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



HttpInvoker is using serialization → transferred objects need to implement the interface `java.io.Serializable`

Configuring the HttpInvoker Service Exporter (1)

Server

- Start with an existing POJO service

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

- Define a bean to export it

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



Pre Spring 3.1: Spring did not allow characters such as '/' in an *id* attribute. If special characters are needed, the *name* attribute could be used instead.

Configuring the HttpInvoker Service Exporter (2a)

- Expose the HTTP service via DispatcherServlet
 - Uses BeanNameUrlHandlerMapping to map requests to service
 - Can expose *multiple* exporters



Configuring the HttpInvoker Service Exporter (2b)

- Alternatively define HttpServletRequestHandlerServlet
 - Doesn't require Spring-MVC
 - Service exporter bean is defined in root context
 - No handler mapping – servlet can only be used by one HttpInvoker exporter
 - Servlet name *must* match `bean name`

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

`http://foo:8080/services/transfer`

service exporter bean name

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

web.xml

Configuring the HttpInvoker Proxy

Client

- Define a factory bean to generate the proxy

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

HTTP POST requests will be sent to this URL

- Inject it into the client

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

Remoting: Pros and Cons

- Advantages
 - (Too) Simple to setup and use
 - Abstracts away all messaging concerns
- Disadvantages
 - Client-server tightly coupled by interface
 - hard to maintain, especially with lots of clients
 - Can't control underlying messaging (hidden)
 - Not interoperable, Java only

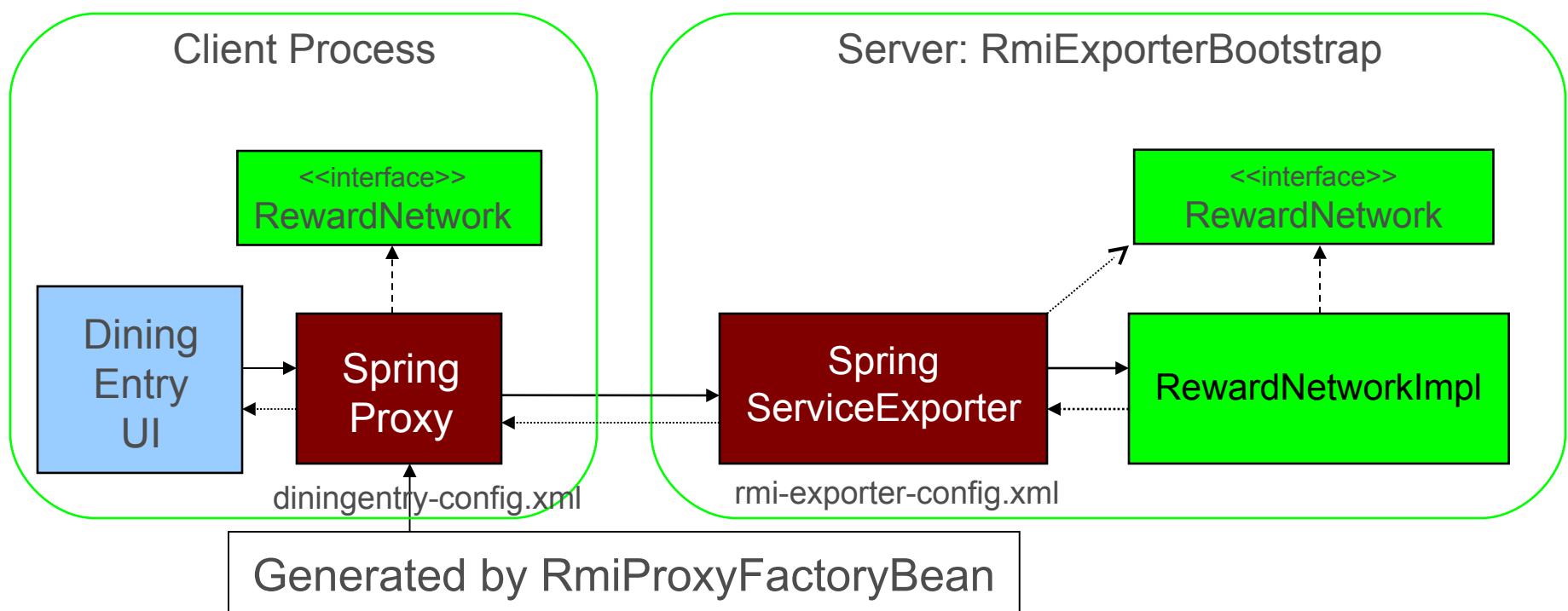
Additional supported Protocols

- Hessian/Burlap (Cross Platform support)
- JAX-RPC (J2EE 1.4 standard for Web Services)
- JAX-WS (Java EE 5 standard for Web Services)
- JMS (basic JMS support)

Lab

Simplifying Distributed Applications
with Spring Remoting

Remoting Lab



Spring Web Services

Implementing Loosely Coupled
Communication with Spring Web Services

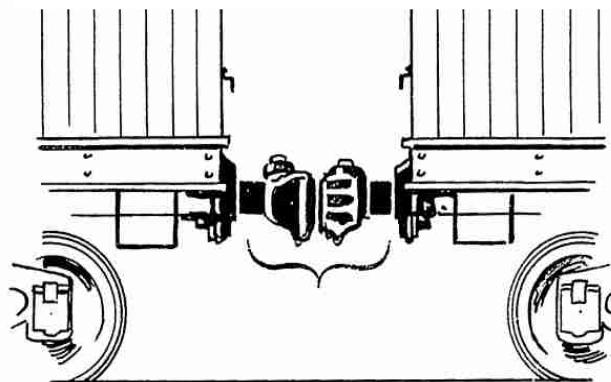
Topics in this Session

- **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**"*

Karl Weikek (attrib)



Loose coupling
increases tolerance ...
Changes should not
cause incompatibility

Web Services enable *Interoperability*

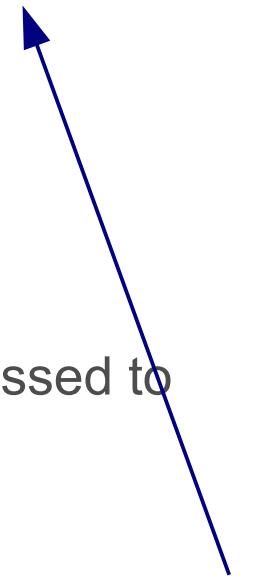
- XML is the lingua franca in the world of interoperability
- XML is understood by all major platforms
 - SAX, StAX or DOM in Java
 - *System.XML* or *.NET XML Parser* in .NET
 - *REXML* or *XmlSimple* in Ruby
 - *Perl-XML* or *XML::Simple* in Perl

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
- Easier to successfully process different requests
 - Looser coupling
 - Fewer 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

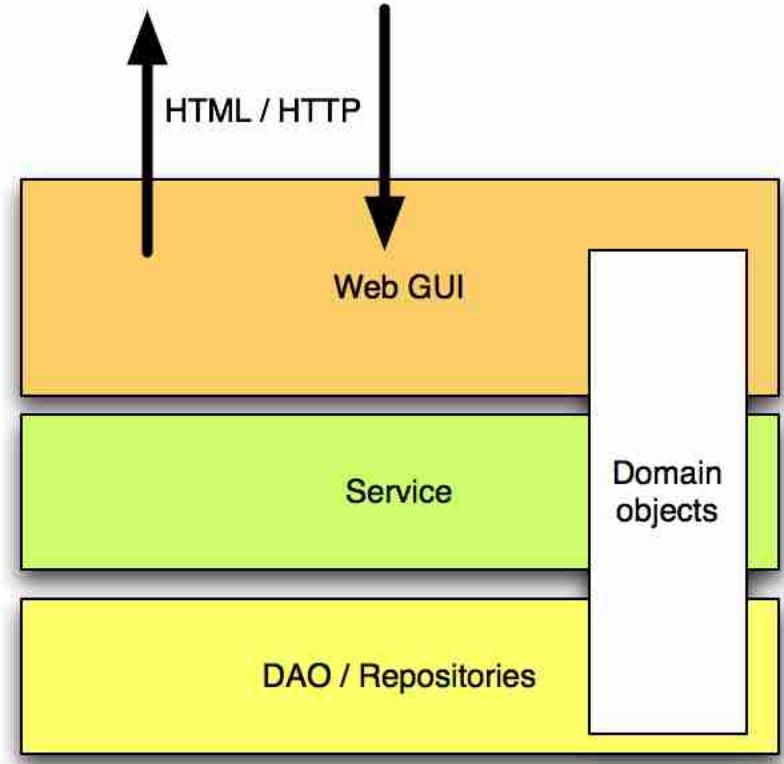


Postel's Law

*"Be conservative in what you do;
be liberal in what you accept from others"*

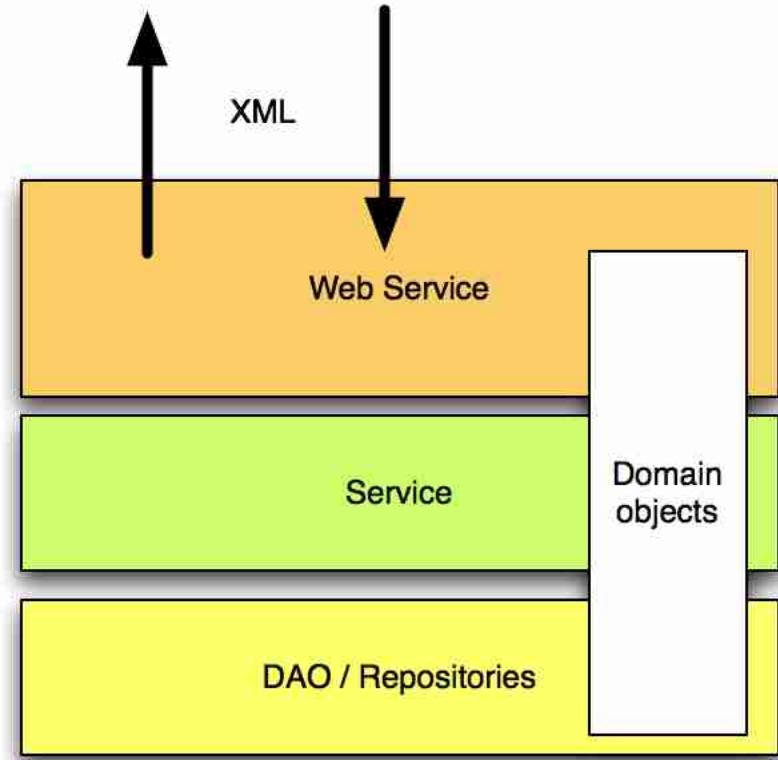
Web GUI on top of your services

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



Web Service on top of your services

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



Topics in this Session

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

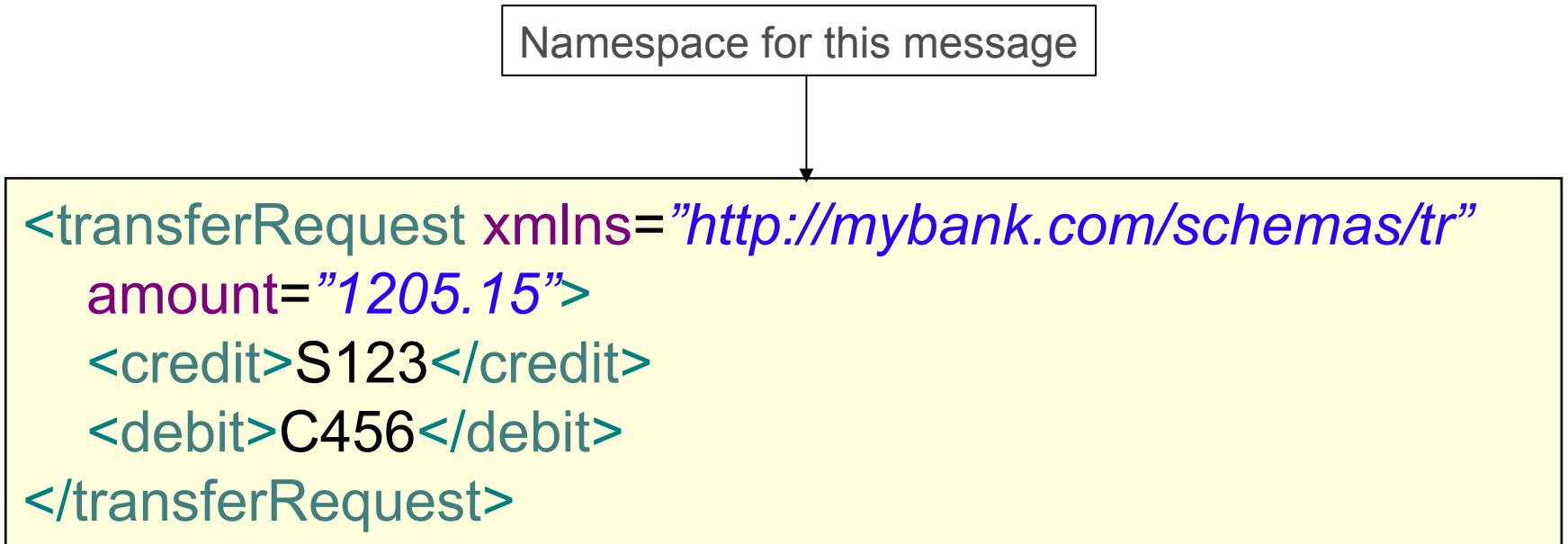
Define the contract

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

Contract-first in 3 simple steps

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

Sample Message



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>
```

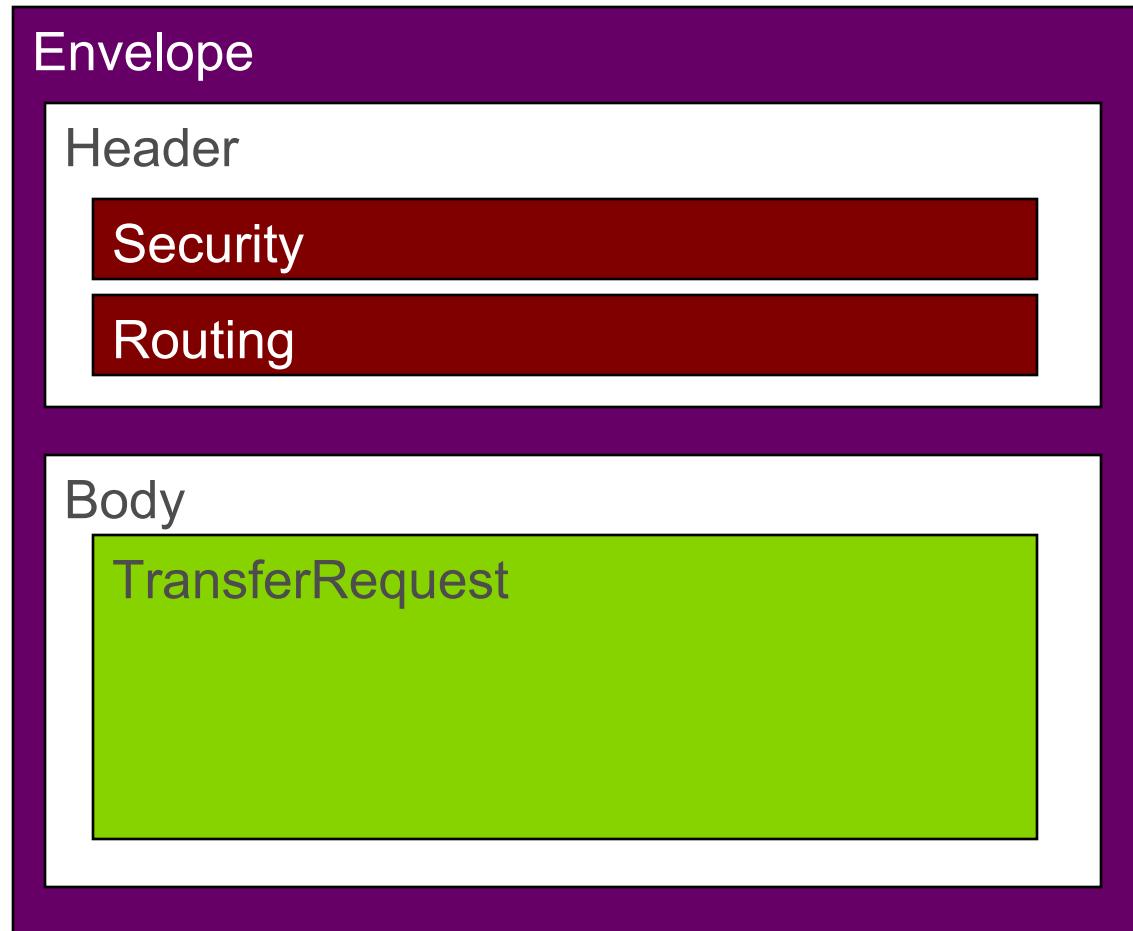
```
<transferRequest
  amount="1205.15">
  <credit>S123</credit>
  <debit>C456</debit>
</transferRequest>
```

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

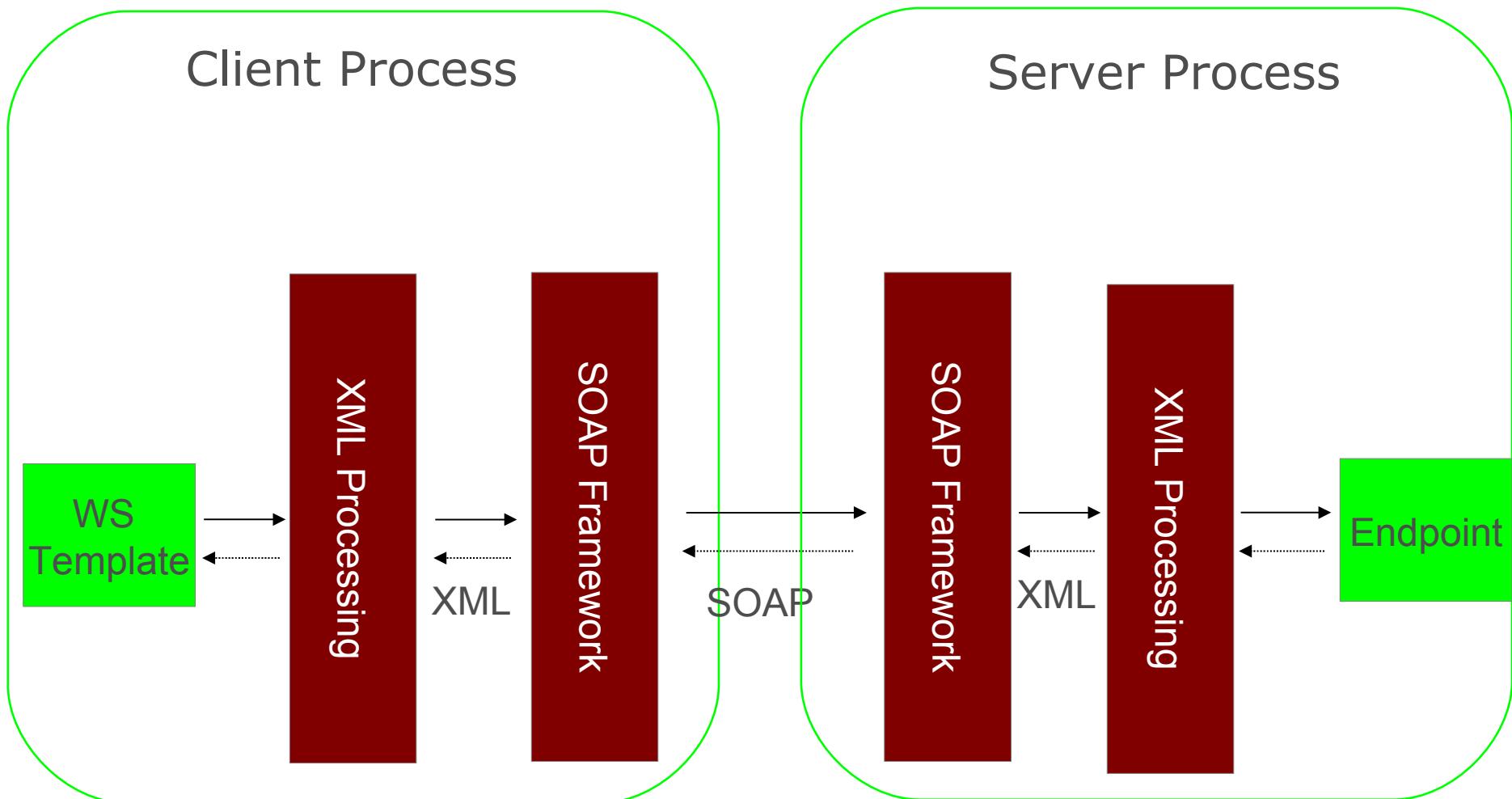
SOAP Message



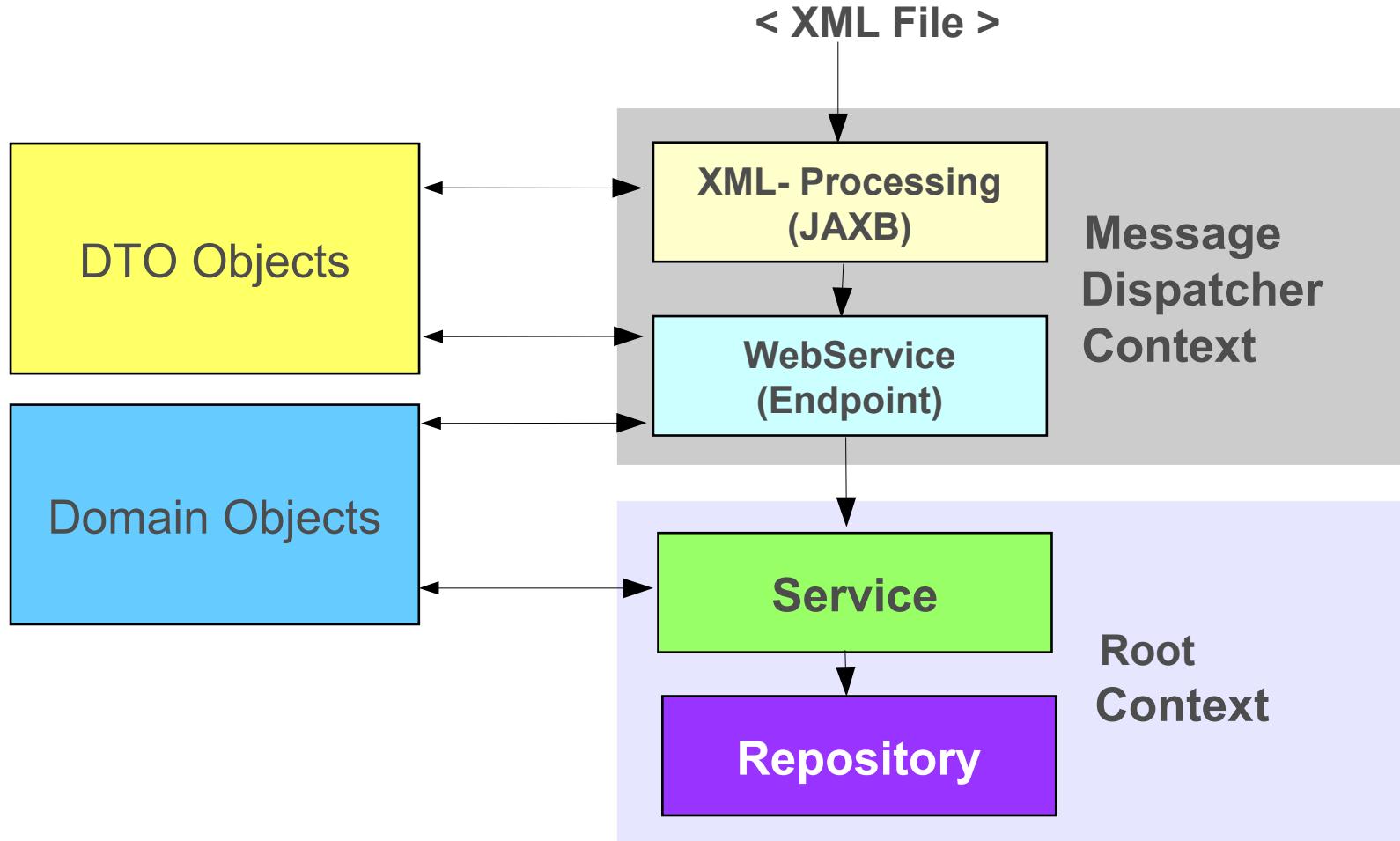
Simple SOAP 1.1 Message Example

```
<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



Spring Web Services Configuration



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

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

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

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

```
<!-- registers all infrastructure beans needed for annotation-based  
     endpoints, incl. JABX2 (un)marshalling: -->  
<ws:annotation-driven/>
```

Implement the Endpoint

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

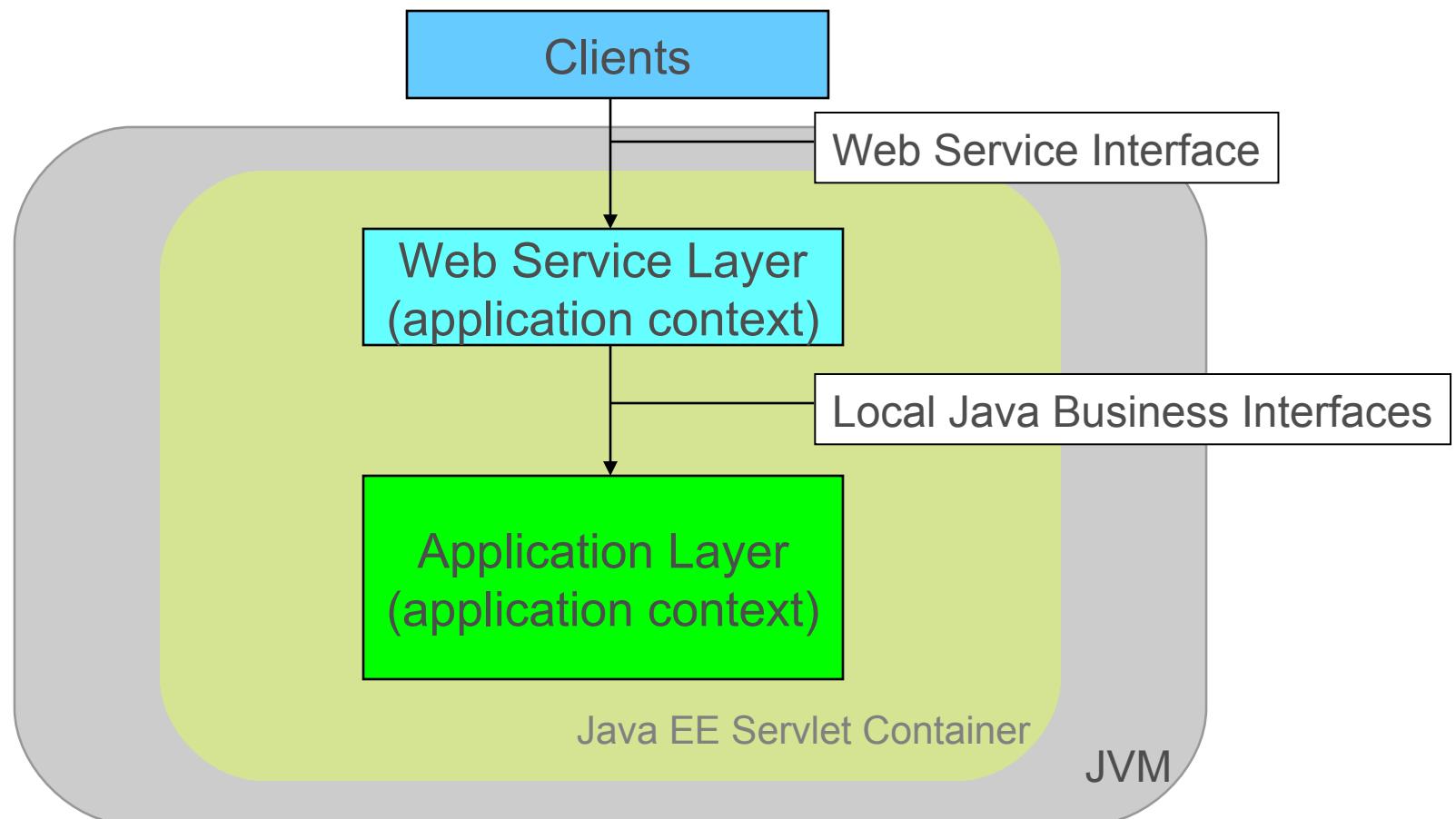
The diagram illustrates the flow of data through a Spring WS Endpoint. A box labeled "Spring WS Endpoint" contains the annotation "@Endpoint". An arrow points from this box to the word "@Endpoint" in the code. Another arrow points from the code's "Mapping" annotation to a box labeled "Mapping". A third arrow points from the "Mapping" box to a box labeled "Converted with JAXB2".

Configure Spring-WS and Endpoint beans

- @Endpoint annotated classes can be component-scanned
 - Saves writing lots of trivial XML bean definitions
 - But 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

See: Spring Web Services Reference, 5.5 Endpoint Mappings

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.marshallSendAndReceive(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

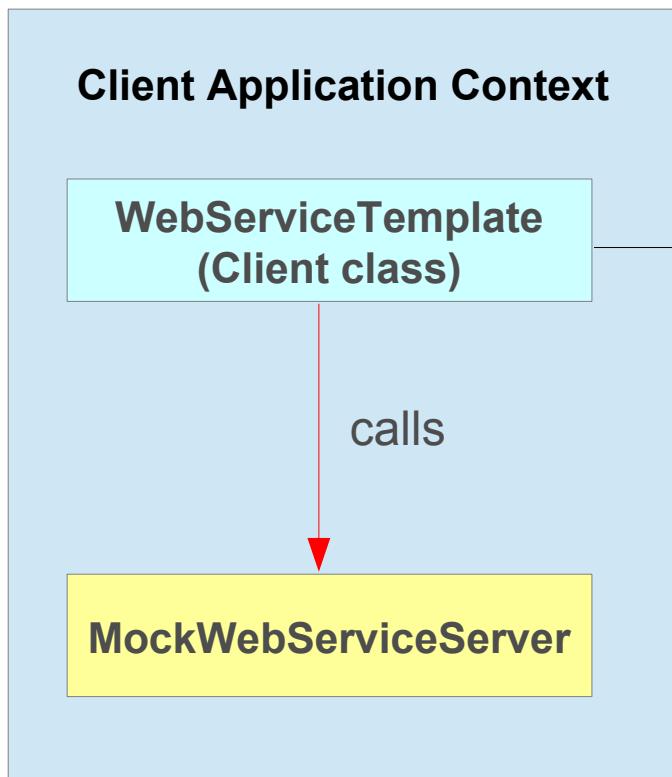
- Interceptors
- Error handling
- **Out-of-container testing**

Out-of-container Testing

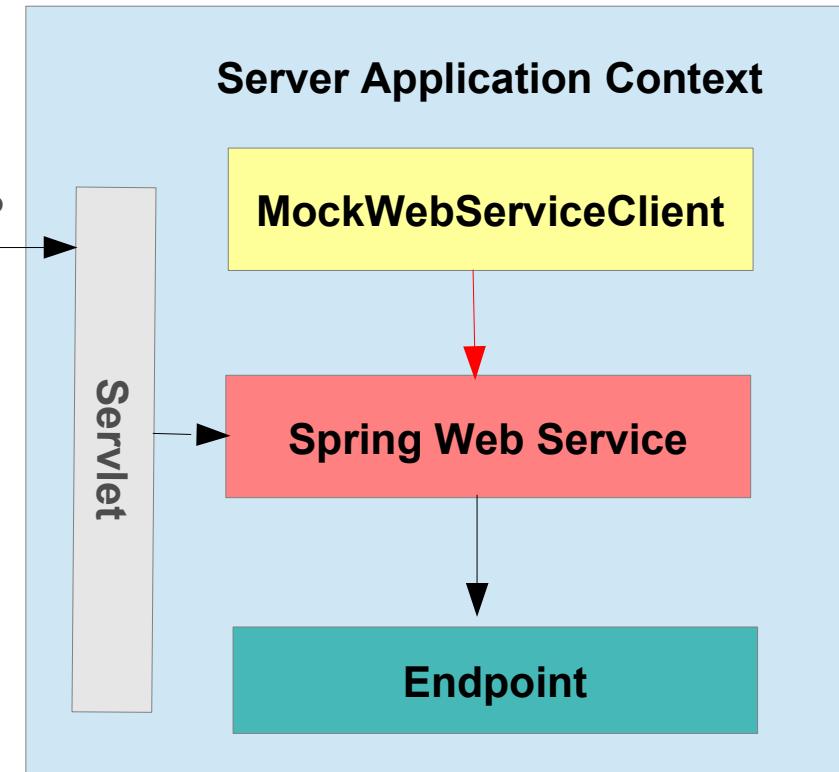
- Spring-WS also supports out-of-container integration testing
 - No stubbing/mocking of endpoint parameters
 - No deployments to local server
- Test endpoints with MockWebServiceClient
 - Fake client that sends request and validates response
 - Tests full MessageDispatcher configuration
- Test clients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

Integration Testing

Client-Side Integration Testing



Server-Side Integration Testing



Using MockWebServiceClient

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
 - 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

Using MockWebServiceServer

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

Error handling and testing

Advanced REST

Spring HATEOAS, Spring Data Rest

Topics in this Session

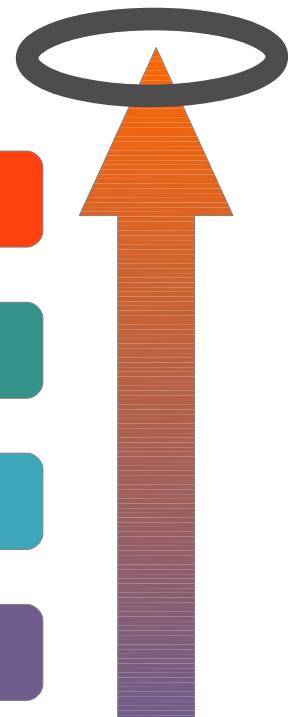
- Richardson Maturity Model
- Spring HATEOAS
- Spring Data REST

Richardson Maturity Model

- REST is more than GETting objects converted to JSON/XML
- There are several levels of REST adoption
- Expressed by *Richardson Maturity Model*

Richardson Maturity Model

The Glory of REST!



Level 3: Hypermedia Controls

Level 2: HTTP Verbs

Level 1: Resources

“The Swamp” – POX or HTTP Tunneling

See <http://martinfowler.com/articles/richardsonMaturityModel.html>

“Correct” REST

- Every accessible resource has a unique URL
- Access via HTTP methods: GET, POST ..
 - *Remember:* not all methods (verbs) make sense for a given resource URL
- **Hypermedia As The Engine of Application State**
 - Resources should contain links to other resources.
 - Clients decide whether to follow the links
 - If system is upgraded, returned links can change to match

Topics in this Session

- Richardson Maturity Model
- HATEOAS
- **Spring HATEOAS**
- Spring Data REST

HATEOAS

- Hypermedia As The Engine of Application State
 - Probably the world's worst acronym!
 - RESTful responses contain the links you need
 - Just like HTML pages do
 - Warning: no standard for this yet
 - Least understood part of Roy Fielding's dissertation

[http://roy.gbiv.com/untangled/2008/
rest-apis-must-be-hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven)

- For example: GET /orders/123
 - Response should include possible URLs from here
 - Such as: how to get order item details, add new item ..

HATEOAS - Concepts

- REST clients need *no* prior knowledge about how to interact with a particular application/server
 - SOAP web-services need a WSDL
 - SOA processes require a fixed interface defined using interface description language (IDL).
- Clients interact entirely through hypermedia
 - Provided dynamically by application servers
- Serves to *decouple* client and server
 - Allows the server to evolve functionality independently
 - Unique compared to other architectures

HATEOAS Account Example

```
<account>
  <account-number>12345</account-number>
  <balance currency="usd">100.00</balance>
  <link rel="self" href="/account/12345" />
  <link rel="deposit" href="/account/12345/deposit" />
  <link rel="withdraw" href="/account/12345/withdraw" />
  <link rel="transfer" href="/account/12345/transfer" />
  <link rel="close" href="/account/12345/close" />
</account>
```

Spring Hateoas provides an API for generating these links in MVC Controller responses

```
<account>
  <account-number>12345</account-number>
  <balance currency="usd">-25.00</balance>
  <link rel="self" href="/account/12345" />
  <link rel="deposit" href="/account/12345/deposit" />
</account>
```

Note: links change as state changes



There is no standard for links yet. This example uses the link style from the *Hypertext Application Language (HAL)*, one possible representation

Implementing HATEOAS

- Need to return data with links in
 - GET requests
 - OPTIONS requests (just return the links, no data)
- Two problems
 - How do we represent links in JSON, XML, ... ?
 - How do we create them in our Java code?
- Spring HATEOAS provides support for links
 - Resource class wraps return data
 - Methods to add links
 - Supports HAL (Hypertext Application Language)

Spring HATEOAS



- Spring Data sub-project for REST
 - For generating links in RESTful responses
 - Supports ATOM (newsfeed XML) and HAL (Hypertext Application Language) links
- For more information see
 - <http://projects.spring.io/spring-hateoas/>
 - <http://spring.io/guides/gs/rest-hateoas/>

How it works?

- Links can be defined
 - Using the **Link** class
 - Don't need to know the URLs directly
 - Deduced at run time from the Controller annotations
- Then responses are extended to include the links
 - Using a **Resource** and a **ResourceAssembler**
- *Let's look at explicit Link creation first ...*

Creating Links from Controller – 1

```
@Controller  
@ExposesResourceFor(Order.class)  
@RequestMapping("/orders/{orderNumber}")  
public class OrderController { ... }
```

```
List<Link> links = new ArrayList<Link>();  
  
// Link to current order  
links.add(ControllerLinkBuilder.linkTo  
    (OrderController.class, orderNumber).withSelfRel());  
  
// ProductController annotated: @RequestMapping("/products/")  
links.add(ControllerLinkBuilder.linkTo(  
    ProductController.class).withRel("products"));
```

Uses reflection and the annotations to determine the correct URLs

Creating Links from Controller – 2

```
@Controller  
@ExposesResourceFor(Order.class)  
@RequestMapping("/orders/{orderNumber}")  
public class OrderController {  
    @RequestMapping("/items")  
    public @ResponseBody Item[]  
        getItems(@PathVariable int orderNumber) { ... }  
    ...  
}
```

```
List<Link> links = new ArrayList<Link>();  
  
// Use getItems() method to derive link to get items. This code  
// doesn't actually call getItems() - works like a mock object.  
links.add(ControllerLinkBuilder.linkTo(  
    ControllerLinkBuilder.methodOn(OrderController.class)  
        .getItems(orderNumber)).withRel("items"));
```

Works out URL needed to invoke getItems():
Creates: <link rel="items" href=".../orders/{orderNumber}/items"/>

Adding Links to RESTful Responses

- Next we need to add these links to RESTful responses
 - Requires slight modification to Controllers
- Key Concepts:
 - Special **Resource** class.
 - **ResourceAssembler** interface.
 - **HttpEntity<T>**

Resource & ResourceAssembler

- Spring HATEOAS Resource class:
 - Wraps a domain object
 - Adds Links to it

```
public class Resource<T> extends ResourceSupport {  
    ...  
}
```

- Spring HATEOAS ResourceAssembler interface:
 - Describes domain to Resource conversion logic

```
public interface ResourceAssembler<T, D extends ResourceSupport> {  
    /** Converts the given entity into an {@link ResourceSupport} */  
    D toResource(T entity);  
}
```

HttpEntity<T>

- Spring Web **HttpEntity<T>**
 - General purpose object in Spring Web
 - Refines definition of HttpServletRequest / Response
 - Consists of headers and body

```
public class HttpEntity<T> {  
    public HttpEntity(T body) { this(body, null); }  
    ...  
    public HttpHeaders getHeaders() { return headers; }  
    public T getBody() { return body; }  
}
```

- Use as return value from MVC @Controller method

Spring HATEOAS Applied

- Before:

```
@RequestMapping (method=RequestMethod.GET, value="/users/{user}")  
@ResponseBody  
User loadUser(@PathVariable Long user) {  
    return crmService.findById(user);  
}
```

- After:

```
@RequestMapping (method=RequestMethod.GET, value="/users/{user}")  
HttpEntity<Resource<User>> loadUser(@PathVariable Long user) {  
    User u = service.findById(user)  
    Resource<User> resource = userResourceAssembler.toResource(u);  
    return new ResponseEntity<Resource<User>>(resource, HttpStatus.OK);  
}
```

ResourceAssembler

```
public class UserResourceAssembler  
    implements ResourceAssembler<User, Resource<User>> {  
  
    public Resource<User> toResource(User user) {  
        Resource<User> userResource = new Resource<User>(user);  
        Collection<Link> links = new ArrayList<Link>();  
  
        links.add( linkTo(methodOn(UserController.class).loadUser(userId))  
                  .withSelfRel() );  
        links.add( linkTo(methodOn(UserController.class).loadUserCustomers(userId))  
                  .withRel("customers"));  
  
        for (Link l : links) { userResource.add(l); }  
        return userResource;  
    }  
}
```

Create Resource and Links collection

Create “self” link

Create “customers” link

Add links to Resource

... ControllerLinkBuilder *static* methods imported

Topics in this Session

- Richardson Maturity Model
- Spring HATEOAS
- **Spring Data REST**

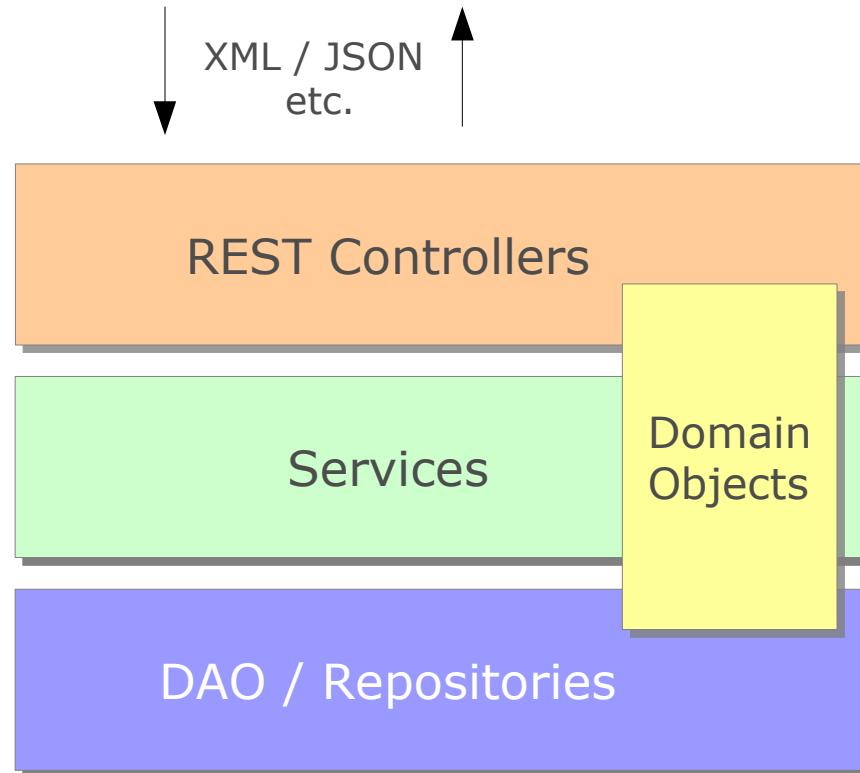
Spring Data REST

- Why write Controllers and ResourceAssemblers at all?
- Spring Data REST defines endpoints based on conventions
- A subproject within the *Spring Data* umbrella
 - <http://projects.spring.io/spring-data-rest/>



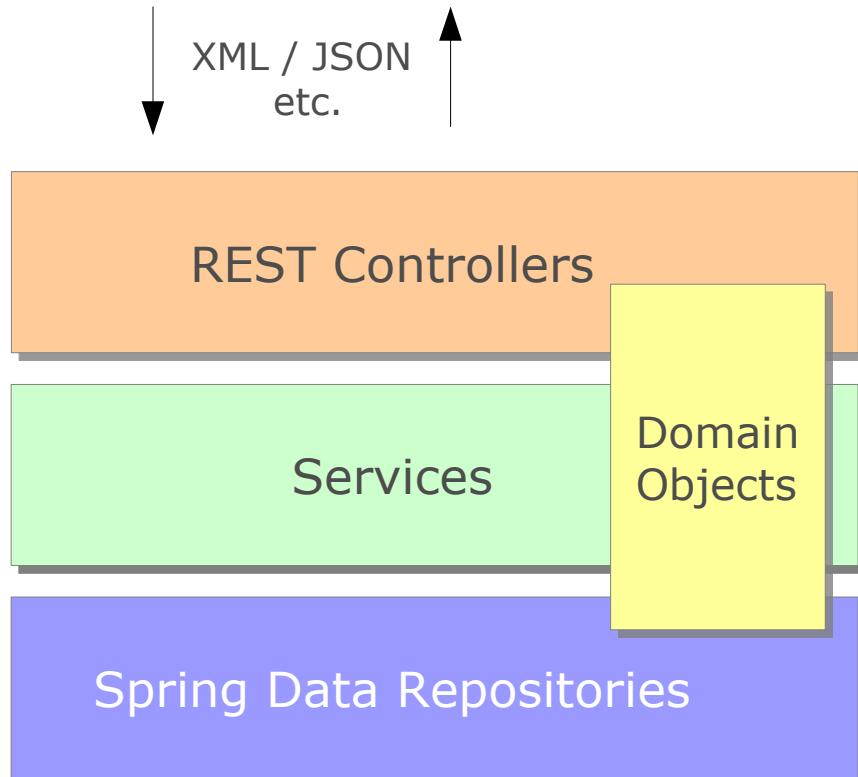
Typical REST Application

- Familiar web application architecture
- REST Controllers provide CRUD interface to clients
- DAO provides CRUD interface to DB



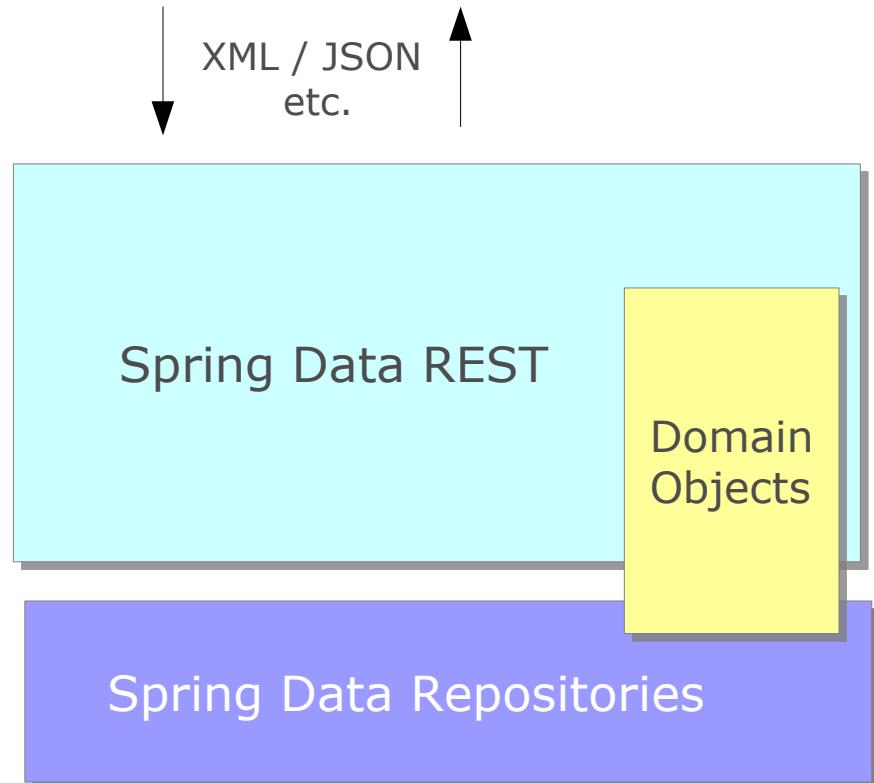
Add Spring Data

- Spring Data provides dynamic Repositories
- You provide the interface, Spring Data dynamically implements.
 - JPA, MongoDB, Neo4J, GemFire, ...
- Controllers & Service Layer have minimal logic



Spring Data REST

- Plugs into dynamic repositories
- Generates RESTful interface
 - *GET, PUT, POST, DELETE*
- Code needed only to override defaults.



Quick Start

- Annotate Domain objects
 - i.e. use JPA annotations if using JPA
- Define interface for dynamic repository
 - Implement CrudRepository at minimum
 - Add @RestResource to describe how to expose via REST

```
@RestResource(path="users", rel="users")
public interface UserRepository extends CrudRepository<User, Long> {}
```

- Configure Spring Data REST
 - `@Import(RepositoryRestMvcConfiguration.class)`
 - Or, extend this class to override defaults

HTTP Verb to Method Mapping

Verb	Repository Method
GET	<code>CrudRepository<ID, T>.findOne(ID id)</code>
POST	<code>CrudRepository<ID, T>.save(T entity)</code>
PUT	<code>CrudRepository<ID, T>.save(T entity)</code>
DELETE	<code>CrudRepository<ID, T>.delete(ID id)</code>

- All methods available by default
 - Disable via override / configuration

```
// Restrict DELETE:  
@RestResource(exported=false)  
void delete(Long id);
```

Links

- Links are inferred based on relationships
 - JPA - @ManyToOne, @OneToOne, etc.
 - Mongo - @DBRef
- Specific links can be added manually

```
@Component public class ExtraLinks {  
    @Inject  
    EntityLinks(entityLinks) {  
        this.entityLinks = entityLinks;  
    }  
  
    Link getSelfLink(User user) {  
        return entityLinks.linkForSingleResource  
            (User.class, user.getId()).withSelfRel();  
    }  
}
```

Summary

- Well implemented RESTful applications utilize hypermedia/links
- Spring HATEOAS adds links to existing Spring MVC RESTful resources
- Spring Data REST provides automatic REST implementations for CRUD repositories