**Spring Integration, Batch, WebLogic and MQ Notes**

## A. Questions

1. [JpaHibernateStatelessSession](#JpaHibernateStatelessSession)
2. Git appears to allow development in isolated branches and isn’t as collaborative? What happens if somebody checks in some code affecting your, do you need to merge their code into your branch and test? Even when there are no conflicts as such but affects testing.
3. Interaction between WebLogic Topics/Queues (and ConnectionFactories) and the MQ topics and queues. I think MQ is installed as a ‘Foreign JMS Module’ in weblogic.

## WebLogic and MQ

**WebLogic JMS Modules** are basically like containers that hold the JMS System Resources which include queues, topics, connection factories, templates, destination keys, quota, distributed queues, distributed topics.

**Foreign JMS** Server provides some additional benefits:

Since the Foreign Destinations are mapped to WebLogic JNDI tree, any **message-driven bean (MDB)** that you deploy to the server can simply reference the remote destination using its local JNDI name.

Websphere MQ and ActiveMQ are JMS Brokers (or mom).

Publishers and Consumers (listeners).

JMS require **Connection Factories** to establish connection for pros and cons to the jms infrastructure.

## Spring Integration

Message-Driven Beans (ejb 3) can be can be configured as Consumer(MessageListener) sets destination as Queue (note how listener onMessage() destination is the Q that it is getting msgs from).

A message **endpoint** is the abstraction layer between the application code and the messaging framework.

Data can come into the framework from **external systems** using a specific type of endpoint called an **adapter**.

**Event Driven Architecture**: What is an event? In formal terms, an event is a significant **change in state** In an event-driven architecture, this event get **sent or published** to all interested parties. The subscribing parties can look at the event and choose to respond or ignore it. The response may include invoking a service, running a business process, or publishing another event.

**Eg of How a msg is transferred**:

P157 Pro SI: Note the addition of the **jms namespace**. By default, the message-driven and outbound JMS **channel adapters** use the connection factory with the default name connectionFactory, so it does not need to be declared in the adapter configuration. The **message-driven** channel-adapter element is used instead of the inbound-channel-adapter element since it **does not require polling**, and will forward the JMS message as soon as it is received. Based on the configuration file, when a message is send to the input channel, it will be converted to a JMS message and sent to the **requestQueue** destination. By convention, a JMS text message is used since the input Message payload is a String. The JMS message is then received by the **message-driven adapter** and forwarded to the **output Message channel**. The text message is converted to a String and sent as the Message payload.

**Point-to-Point**: many types: **QueueChannel**, PriorityChannel, RendezvousChannel, DirectChannel, ExecutorChannel, and NullChannel.

**Pub-Sub**: allows one-to-many relationship between the producer and consumer, such that multiple consumers may receive the same message. The message is marked as “received” and **removed** from the channel when **all the subscribed receivers** have consumed the message. So msgs retained on channel .

**Data-Types Channel**: CDW uses channels with diff event types on same channel? Investigate further.

**Channel Adapter**: Allows an application to connect to the messaging system.

**This is probably a definition of the Events-Bus**

**Message Bus**: a message bus is a combination of **a canonical data model**, a common **command set**, and a **messaging infrastructure** that allows different systems to communicate through a **shared set of interfaces**. By leveraging Spring Integration adapters and transformers, message channels can be connected and used to create a message bus allowing communication between different systems.

**Event-Driven Consumers**

The org.springframework.integration.endpoint.EventDrivenConsumer only supports channels that implement the org.springframework.integration.core.SubscribableChannel interface, such as DirectChannel, ExecutorChannel, **PublishSubscribeChannel**, and **SubscribableJmsChannel**.

**ConnectionFactories**: Connect SI to JMS broker.

**Control Bus:** same messaging system used for managing and monitoring can be used for application-level control. Ie @ManagedOperation is a bean method which can be invoked by a message (containing SpEl). Used to give application-level control.

**JMS Integration**: Before Spring Framework 1.1, applications needed to handle JMS connection management, session management, message creation, synchronous and asynchronous message delivery, and error handling manually.

In JMS, there are two types of destinations: **queue** and **topic**. A JMS queue (javax.jms.Queue) is a point-to-point destination that allows one consumer to receive a message at any given time. A JMS topic (javax.jms.Topic) is a publish/subscribe-style destination that allows messages to be delivered to multiple consumers. Both classes extend **javax.jms.Destination**.

**JMS Brokers**: eg ApacheMQ and JBoss HornetQ.

**Sending JMS Messages**. The JMS adapters require either a reference to a Spring JmsTemplate instance, or both a ConnectionFactory and java.jms.Destination reference. Directly specifying a destination name works as well: <int-jms:outbound-channel-adapter id="ticketOutbound" destination-name="ticket.queue".

**Receiving JMS Messages**: There are two types of inbound channel adapters: **polling** and **message-driven**. The polling channel adapter uses **JmsTemplate** internally while the **message-driven inbound channel adapter uses the Spring MessageListener container abstraction.**

## M. Spring Batch

**Job-step**: ItemReader-Itemprocessor-ItemWriter

**JobRepository**: configured to use HSQLDB by default, but can use any db to persist to disk.

**jobListeners**: Events at given points in the job. JobExecutionListener beforeJob and afterJob.

**ExecutionContext**: Essentially the state/session of a job(s). Whole job or single step.

**Hibernate (not really approp for batch processing)**: Using Hibernate in batch processing is not as straightforward as it is for web applications. For batch processing, if you use Hibernate naively, you would use the normal stateful session implementation. The standard session within Hibernate is stateful. If you are reading a million items, processing them, then writing those same million items, the Hibernate session will cache the items as they are read and an OutOfMemoryException will occur. Hibernate incurs larger overhead than straight JDBC does.

**JPA**: Provides standardised api for impls such as hibernate – must be used in conjunction with provider. JPA supports paging but not cursor driven access. Uses a *EntityManager*with Hibernater injected as DataSource.

**ItemReaders (Cursor based):** 'streaming' relational data. The Java ResultSet class is essentially an object orientated mechanism for manipulating a cursor calling next. The Spring core JdbcTemplate gets around this problem by using the callback pattern to completely map all rows in a ResultSet and close before returning control back to the method caller.ie you get a list of objects.

ItemReaders allow chunks to be read and written (GC’ed) before the end of the resultSet (streamed):

<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="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>

</property>

</bean>

See [SpringBatchandHibernateCursor](#SpringBatchandHibernateCursor) for explanation of how Hibernates implements cursoring and avoids issues with stateless session.

**ItemReaders (Paging based):**

An alternative to using a database cursor is *executing multiple queries* where each query is bringing back a *portion of the results*. We refer to this portion as a page. Each query that is executed must specify the starting row number and the number of rows that we want returned for the page.

JpaPagingItemReader: JPA doesn't have a concept similar to the Hibernate StatelessSession so we have to use other features provided by the JPA specification. Since JPA supports paging, this is a natural choice when it comes to using JPA for batch processing. After each page is read, the entities will become detached and *the persistence context will be cleared* in order to allow the *entities to be garbage collected* once the page is processed.

Q. JPA relies on a vendor such as hibernate? As we are using jpa does this mean we can’t use Hibernate stateless session?

**Spring Batch and Integration**:

SI is included in SB as it provides support for messaging/file Inbound channel triggers to start jobs. Additionally you can create Spring Integration flows that use multiple different adapters to easily ingest data for your Batch Jobs from multiple sources simultaneously using configuration only. Allows for an decoupled ***event-driven execution*** of the JobLauncher.

## N. Spring Data

**Reasons for**: Consists of dedicated modules for NoSQL stores as well as JPA and JDBC modules with additional support for relational databases.

**JPA**: Does not provide annotations for NoSQL. Spring Data aims to do this.

**Repository**: Repository abstraction sits on top of template impl that will reduce the effort to implement data access objects to a plain interface definition for the most common scenarios like performing standard CRUD operations as well as executing queries in case the store supports that.

**Native No-SQL**: Main actions Implemented via MongoTemplate, but native api support via callbacks.

**Repository Interface**: extend Repository<Type, ID>. Auto-discovery will now cause the Spring Data repositories to be found, and Spring beans will be created that actually *consist of proxies* that will implement the discovered interface. Ie a client could now autowire.

**Default Query Methods**: Following convention for method names. Eg findByEmailAddress() the infrastructure will inspect the methods declared inside the interface and try to determine a query to be executed on method invocation.

**Manual JPA Query in Interface**: Can use @Query(“select”) rather than auto-gen. Queries can be externalised into props files.

**Query Paging**: For large return sets. Pageable.

**CrudRepository and PagingAndSortingRepositories**: Extend Repository and add additional functionality. *NB. Spring Data provides implementation classes for these interfaces.*

**AbstractEntity**: Define a parent domain class with id, equals and hashcode.

**QueryDSL**: TBC if necessary. This *allows Type-safe* querying.

**REST Repository Explorer**: Will expose CRUD ops on repositories to http REST posts. Run using Jetty: command line using the Maven Jetty plug-in. Jetty is a tiny servlet container capable of running Servlet 3.0 web applications. Start jetty cmd line: mvn jetty:run -Dspring.profiles.active=with-data. P158. Use *Dev HTTP Client* for Google Chrome as a simple http client. Eg using curl curl -v http://localhost:8080.

## O. Spring Integration

Ref: http://docs.spring.io/spring-integration/docs/4.0.3.RELEASE/reference/html/

Does use adapters

## Y. Terminology

**Topics** = pub-sub

**Queue** = point-to-point

**Durable subscription** = When a durable subscription is closed, however it continues to exist in the MQ broker, accumulating messages. A closed durable subscription is described as being inactive. Allows msgs sent when sub closed to be picked up when active again. applies to topics but not to queues. Queue has a single built-in sub shared by all consumers.

See <http://openmessaging.blogspot.co.uk/2009/04/durable-messages-and-persistent.html>

**Events** = logically *independent* msgs. Eg trades

**State** = info that is regularly updated/replaced. Eg stock prices

**STP** = Straight Trade Processing. Separate to the BDSL (which is batch)

**BDSL** = Integration Infrastructure

**Magix** = xml schema/db (canonical format)

**FODB** = Persistent layer after confirmation received on trade.

**GoldenSource** = 3rd party product for reference data.

# Appendix

**1. Spring Batch and Hibernate Excerpt**

http://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html

**HibernateCursorItemReader**

Just as normal Spring users make important decisions about whether or not to use ORM solutions, which affect whether or not they use a JdbcTemplate or a HibernateTemplate, Spring Batch users have the same options. **HibernateCursorItemReader** is the Hibernate implementation of the cursor technique. **Hibernate's usage in batch has been fairly controversial.** This has largely been because Hibernate was originally developed to support online application styles. However, that doesn't mean it can't be used for batch processing. The easiest approach for solving this problem is to use a **StatelessSession** rather than a standard session. **This removes all of the caching and dirty checking hibernate employs that can cause issues in a batch scenario**. For more information on the differences between stateless and normal hibernate sessions, refer to the documentation of your specific hibernate release. The HibernateCursorItemReader allows you to declare an HQL statement and pass in a SessionFactory, which will pass back one item per call to read in the same basic fashion as the JdbcCursorItemReader. Below is an example configuration using the same 'customer credit' example as the JDBC reader:

HibernateCursorItemReader itemReader = new HibernateCursorItemReader();

itemReader.setQueryString("from CustomerCredit");

//For simplicity sake, assume sessionFactory already obtained.

itemReader.setSessionFactory(sessionFactory);

itemReader.**setUseStatelessSession**(true);

int counter = 0;

ExecutionContext executionContext = new ExecutionContext();

itemReader.open(executionContext);

Object customerCredit = new Object();

while(customerCredit != null){

customerCredit = itemReader.read();

counter++;

}

itemReader.close(executionContext);

This configured ItemReader will return CustomerCredit objects in the exact same manner as described by the JdbcCursorItemReader, assuming hibernate mapping files have been created correctly for the Customer table. The 'useStatelessSession' property defaults to true, but has been added here to draw attention to the ability to switch it on or off. It is also worth noting that the fetchSize of the underlying cursor can be set via the setFetchSize property. As with JdbcCursorItemReader, configuration is straightforward:

<bean id="itemReader"

class="org.springframework.batch.item.database.HibernateCursorItemReader">

<property name="sessionFactory" ref="sessionFactory" />

<property name="queryString" value="from CustomerCredit" />

</bean>