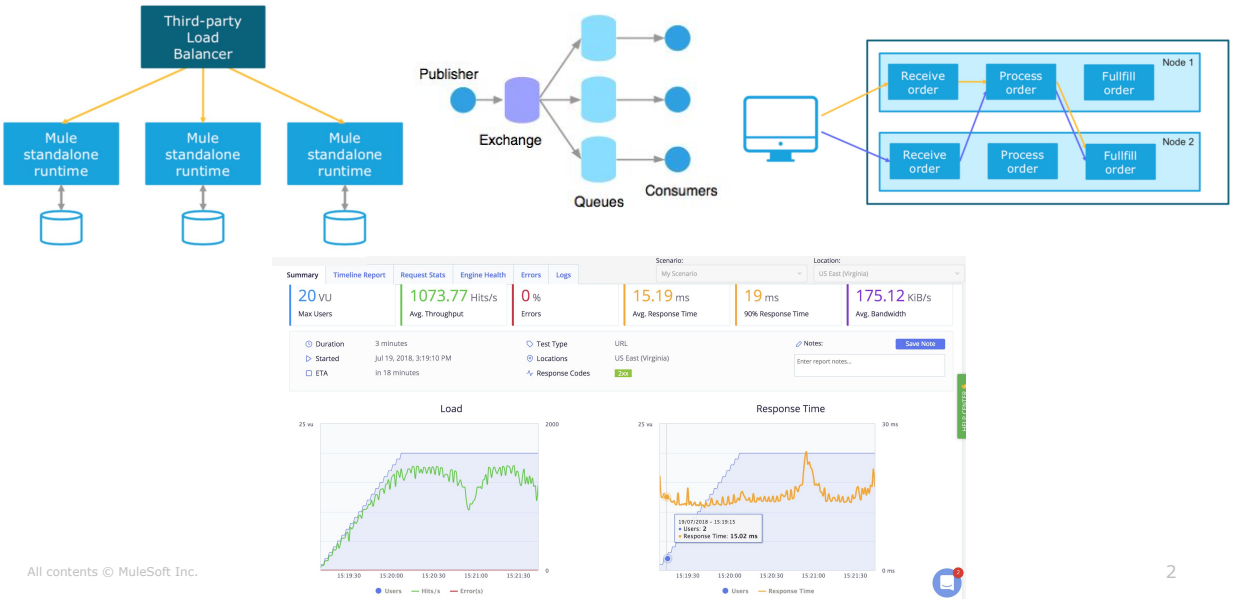
Part 3: Strategies to Meet Non-Functional Requirements

Goal

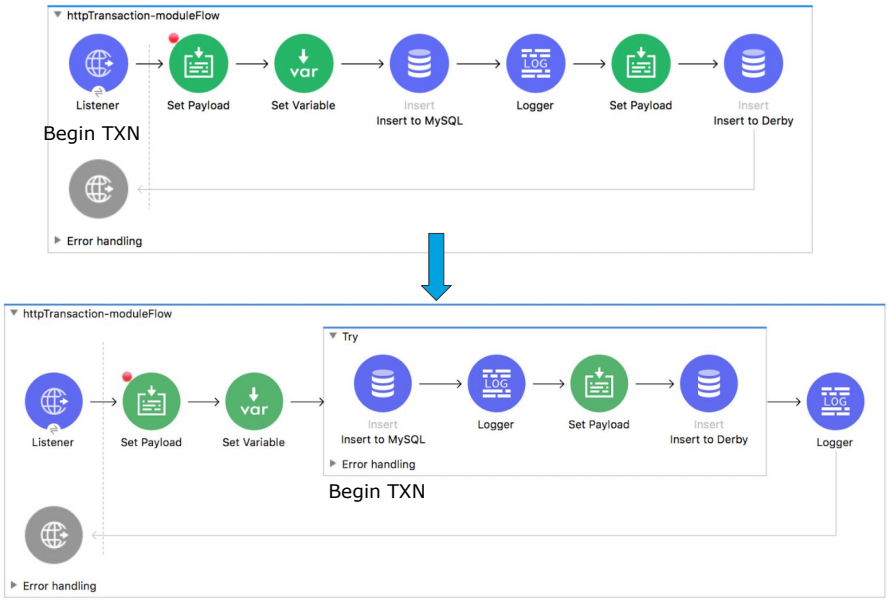


At the end of this part, you should be able to

- Decide when and how to design for transactional requirements
- Clarify and balance reliability, high availability, and performance goals
- Design to meet agreed reliability, high availability, and performance goals
- Design secure Mule applications, their network communications, and their deployments
- Summarize the course with a completed architecture

Module 11: Designing Transaction Management in Mule Applications

Goal



At the end of this module, you should be able to

- Identify why and when transactions are supported in Mule applications
- Understand resources that participate in transactions in Mule applications
- Demarcate transaction boundaries in Mule applications
- Choose the transaction type based on the participating resources
- Manage a transaction using the Saga pattern

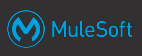
Identifying why and when transactions should be used in Mule applications

What is a transaction?

MuleSoft

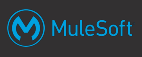
- In its narrow technical definition, a **transaction** is a grouping of operations that are guaranteed to all complete, or none at all
  - This is the definition often used in computer science
  - Traditionally were implemented to meet ACID guarantees
    - Atomicity
    - Consistency
    - Isolation
    - Durability
- A **business transaction** is a much more general term that comprises potentially any kind of finite activity with business meaning
- A **financial transaction** is a transfer of funds, either in a short time or over an extended period of time
  - For example, paying a mortgage over 30 years is a financial transaction

How traditional two-phase commit protocols are applied to distributed systems



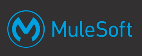
- Databases and related systems **traditionally** use two-phase commit protocols to manage transactions
  - Transactions progress through a prepare and a commit phase across all participants of the two-phase commit transaction
  - The hope is to guarantee all the separate operations complete all at once, or none of them complete and they are all rolled back
  - They maintain the ACID properties of a transaction
  - They require heavy-weight **transaction manager** services
- **Global transactions** are transactions that involve multiple separate transactional resources (such as databases)
  - A **transaction manager** is used to **atomically** coordinate and rollup the individual transaction of each resource

How XA-capable transaction resources participate in global/distributed transactions



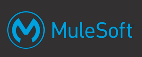
- Resource managers participating in global transactions, using two-phase commit, often implement the **XA interface**, so they can be managed by a transaction manager
  - Allows separate protocols and systems to be combined into one global transaction
- The XA interface specifies communication between a **transaction manager** (**TM**) and a **resource manager** (**RM**)
  - Each RM must respond to requests from the TM to **prepare**, **rollback**, or **commit** an operation in its managed resource

Problems with two-phase commit protocols applied to modern large distributed systems



- Requires remote communication to each participating system
- Requires each participating system to support two-phase commit
- Locks (parts of) each participating system for the duration of the transaction
- Communication protocol grows linearly (O(N)) with the number of nodes

Alternatives to two-phase commit protocols for distributed transactions



- Large **distributed transactions systems** are available to improve upon two-phase commit protocols
  - Such as Google Spanner
- **Eventual consistency** is often used in large distributed systems, but this compromises the **consistency** goals of 2-phase commit transactions
- **Sagas** are another alternative to global distributed transactions
  - Requires explicitly coding and managing compensating transactions
- All these alternatives can be characterized through the **CAP theorem**
  - Can only have two of Consistency, Availability, and Partition Tolerance
  - https://en.wikipedia.org/wiki/CAP_theorem

Managing transactions using the Saga pattern

Managing transactions using the Saga pattern

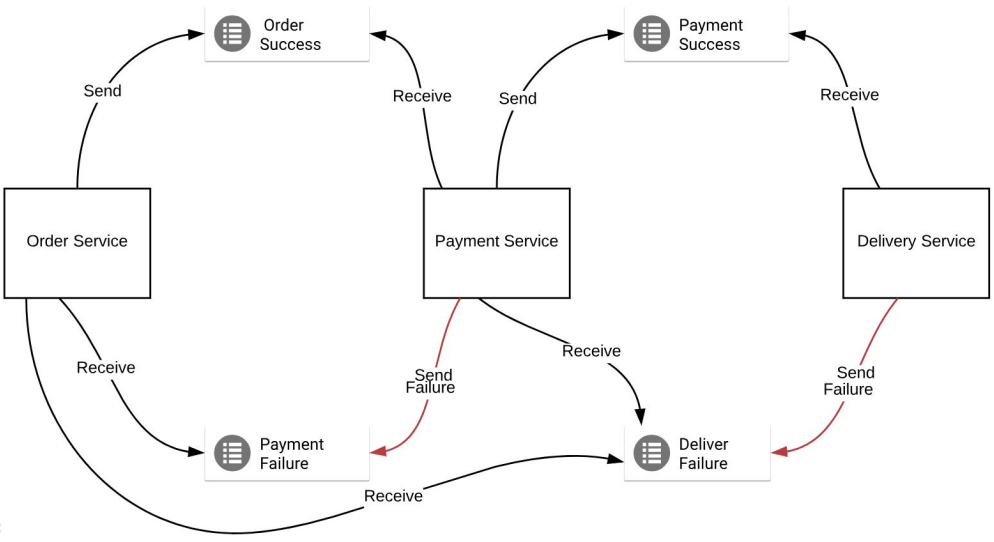
MuleSoft

- The Saga pattern is an alternative to XA when participating resources are not XA capable or XA is undesired
  - For instance, API invocations
  - A way to achieve reliability that does not have ACID guarantees
- Features of the Saga pattern include
  - Maintains consistency (but not atomicity) across multiple services without requiring a **transaction manager system**
  - Uses a **sequence** of local TXs
  - Failure of a local TX executes a **series of compensating transactions** to undo the previous successful TXs
  - The leads to a more complex programming model compared with XA resources

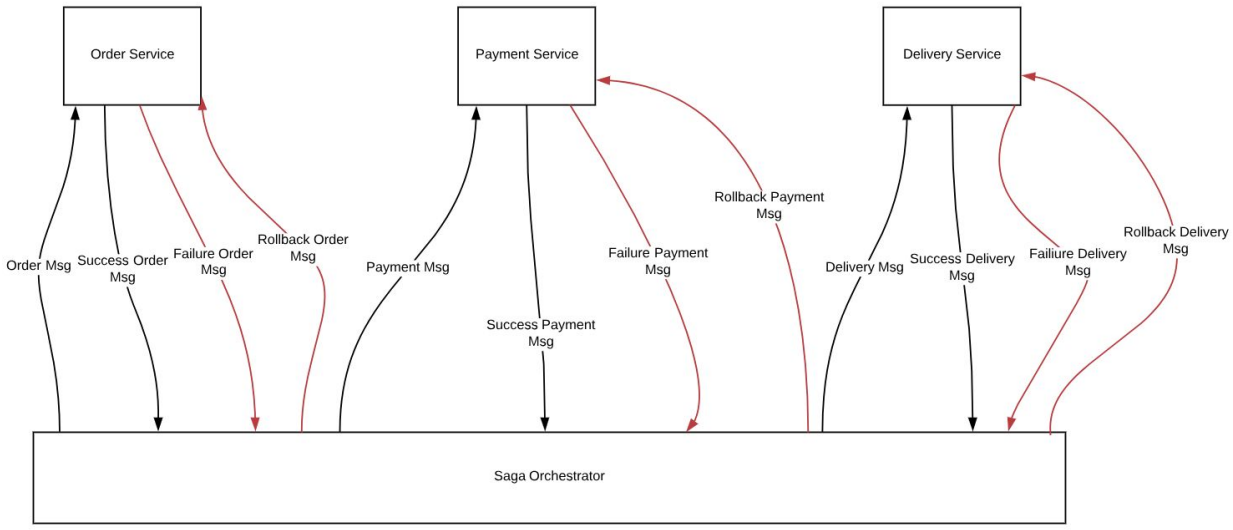
Some ways to implement Saga patterns

- Event/Choreography pattern
  - The service executes the transaction (**TX**) and publishes an event to be used by the next service
- Command/Orchestration pattern
  - A **Saga orchestrator** communicates with each service using a command/reply style
  - The orchestrator logic can be modeled as a state machine
  - Similar to an XA pattern, the orchestrator coordinates transactions
    - But does not require an XA-capable transaction manager nor resource managers
    - So this works with API invocations, messages, etc.

Example: Saga Event/Choreography pattern



Example: Saga Command/Orchestration pattern



Tradeoffs of Event/Choreography vs Command/Orchestration patterns

	Event/Choreography	Command/Orchestration
Characteristics	• Listens for successful event from previous step and failure events from subsequent steps	• Listens for success and failure messages from the Saga orchestrator
Pros	• Use when coordination across domains of control/visibility is required	• Use to manage business processes that are inside one organisation that you have control over • Centralizes orchestration of TX • Complexity grows linearly when steps are added
Cons	• Adding and deleting steps in a TX is more difficult	• Single point of failure

Exercise 11-1: Identify when and why a Mule application should support transactions

- A Mule application makes multiple related modifications, and at least one fails
- Answer these questions
  - How can the consistency of the entire system be guaranteed?
  - What does consistency mean?
  - Discuss the different aspects of this scenario, including edge cases and performance considerations

Exercise solution

- Maintain data integrity with transactions or Sagas
- Characteristics
  - Data integrity
    - Atomicity
    - Consistency
    - Isolation
    - Durability
  - Performance
    - Can decide the size of the unit of work that is committed in one transaction
  - Distributed transactions are slow because of logging and network communication
  - Performance overhead of locking multiple resources in a transaction

Exercise solution

- Sagas enable an application to maintain data consistency across multiple services without using distributed transactions:
  - Performance
  - Complex programming model
  - Must design compensating transactions that explicitly undo changes made earlier in a Saga

Managing transactions in Mule applications

Transactions (TXs) in Mule

- Mule supports
  - Single-resource (local) TX
    - For example, a series of operations to one database connection
  - Global (XA) TX
    - A series of operations between one or more (typically two) different (XA compatible) systems, such as two different databases, or a database and a JMS server
- Supported connectors must be configured to use transactions
- All message processing done on a single thread
  - Bypasses typical reactive thread usage
- XA transactions need an XA-capable transaction manager

Connector operations that support transactions

- JMS
  - Publish
  - Consume
- VM
  - Publish
  - Consume
- Database
  - All operations

Demarcating the beginning of a transaction

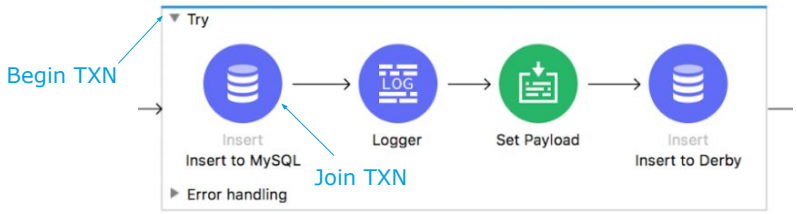
- Begin a transaction through either
  - A Try scope
  - A transactional connector acting as an event source
    - Database, JMS, or VM Listeners

Demarcating the end of a transaction

- Committed automatically at end of a
  - Flow
  - Try scope
  - On Error Continue scope
- Rollback transaction
  - After a failure occurs in a transaction scope, but only if the error is not handled in an On Error Continue scope
  - On Error Propagate scope
  - By throwing an error in a flow or in a Try scope using <raise-error>
    - Note: This component is not currently available in an Anypoint Studio palette, it must be typed into the Mule XML configuration file

Demarcate transaction boundaries

```
<!-- Local Transaction Begins -->
    <try transactionalAction="ALWAYS_BEGIN">
        <db:insert config-ref="Derby_Database_Config"
    transactionalAction="ALWAYS_JOIN"/>
        <!-- Rollback Transaction in case of error -->
        <error-handler>
            <on-error-propagate>
                <raise-error type="ANY" description="#[Rollback]"/>
            </on-error-propagate>
        </error-handler>
    </try>
<!-- Transaction Ends -->
```



Transaction types in Mule applications

Transaction Type	Characteristics	Number of resources	Available resources	Performance
Single Resource/ Local	• Receive and/or send message to only one resource • Perform database operation to only one database resource	1	JMS VM JDBC	relative to XA performs better
XA	• Receive and/or send message to one or more resources • Perform database operation to one or more transactional resources • Connectors must be XA enabled • Uses two phase commit	>=1	JMS VM JDBC	relative to local slower, but ACID across all resources

Transaction log

- The TX log stored in $MULE_HOME/.mule/TX-log
- By default, this file will contain up to 50000 records
- For performance, the TX log size can be optimized in MB using

```
<configuration maxQueueTransactionFilesSize="150" />
```

- Logs are not human readable and are available during the lifetime of the TX
- The TX log contains begin, commit, rollback, and isolation details for each TX
- A TX manager has its own TX log file - BTM, XA, and Local
- The TX log enables the Mule runtime to rollback or restore a TX in case of a hardware failure or Mule application failure.

Using the default Bitronix XA transaction manager in Mule applications

- **Bitronix** is available as the XA transaction manager for Mule applications
- To use Bitronix, declare it as a global configuration element in the Mule application

```
<bti:transaction-manager />
```

- Each **Mule runtime** can have only one instance of a Bitronix transaction manager, which is shared by all Mule applications
- For customer-hosted deployments, define the XA transaction manager in a Mule domain
  - Then share this global element among all Mule applications in the Mule runtime

Setting a transaction timeout for the Bitronix transaction manager

- Set the transaction timeout either
  - In wrapper.conf
  - In CloudHub in the Properties tab of the Mule application deployment
- The default is 60 secs

```
mule.bitronix.transactiontimeout = 120
```

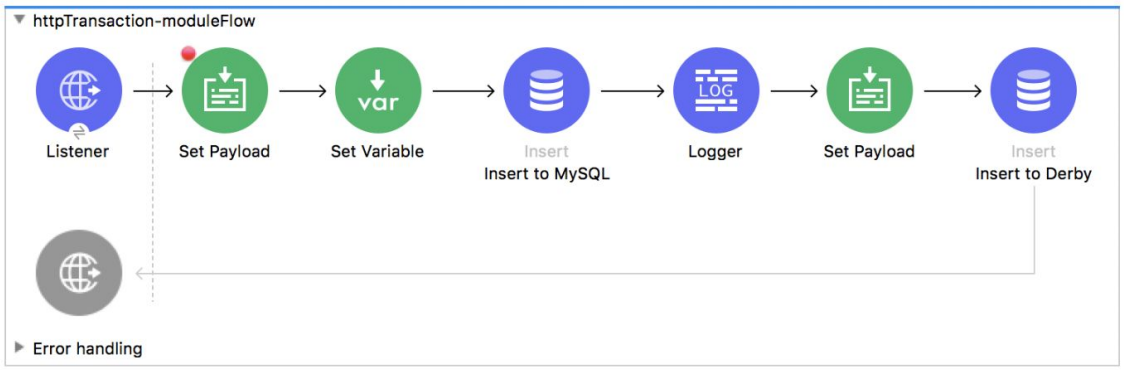

Participating in transactions (TX)

- Defining transactions behaviour on the connector operation

Action	Processor Supports	Description
None	JMS, VM and DB Connectors (listeners only)	Never participate in TX Ignore any ongoing TX
INDIFFERENT	Try scope	Never participate in TX Ignore any ongoing TX
ALWAYS_BEGIN	JMS, VM and DB Connectors (listeners only) and Try scope	Always start new TX Throw exception if TX ongoing

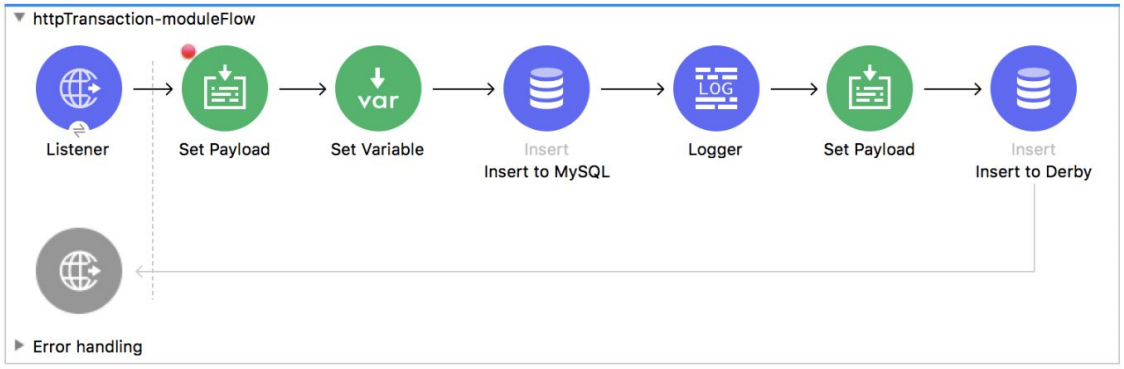
Exercise 11-2: Define a transaction

- For a specific flow, identify transactional resource(s), transactional types, transactional boundaries and their demarcation



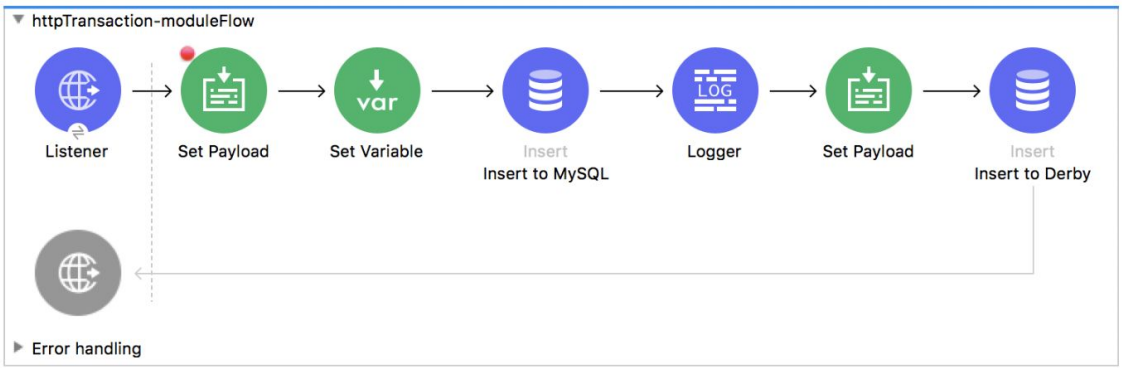
Exercise steps

- Identify transaction resources
- Transactional resource(s) are DB, VM and JMS, identify in flow TX resources



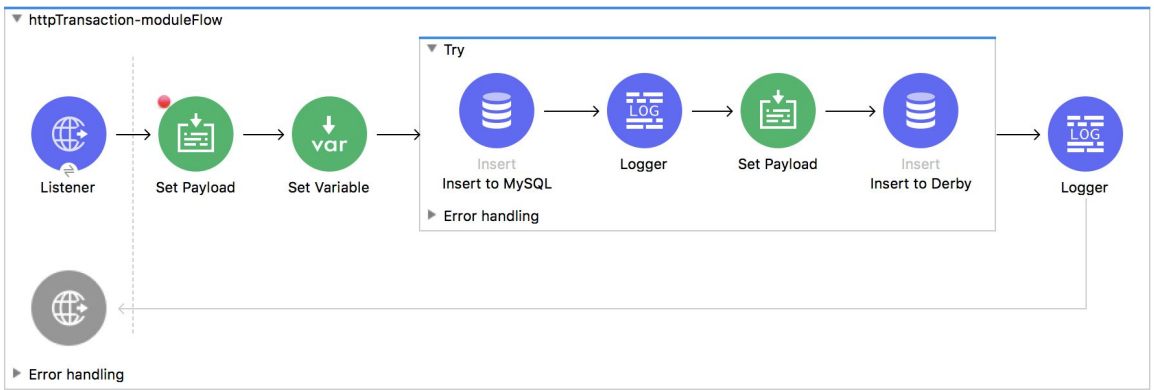
Exercise steps

- Define TX boundary and TX type
  - Should we use local/single resource or XA TX
  - Should TX start with listener or have transaction in Catch scope

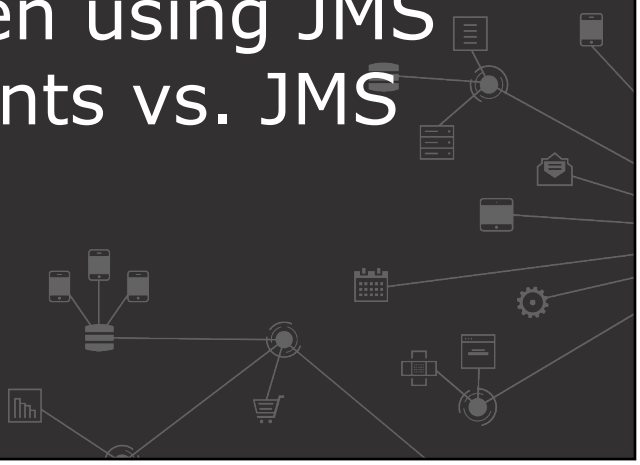


Exercise solution

- Define transactions in a Mule application
- Use a Try scope to handle rollback exceptions
- Define



Deciding between using JMS acknowledgements vs. JMS transactions



JMS acknowledgement in Mule applications

MuleSoft

- Receipt of JMS messages must be acknowledged
  - Assuming JMS transactions are not used
- Unacknowledged messages are **redelivered**
- JMS Recover Session **redelivers** all the consumed messages that had not being acknowledged before this recover
- **JMS Recover session** is used inside Error Handler to redeliver messages
- Acknowledgement mode set on JMS listener
  - auto, manual, immediate, dups_ok

JMS acknowledgement modes

- AUTO (default)
  - Automatic acknowledgement at end of flow or On Error Continue scope
- IMMEDIATE
  - Same as NONE in Mule 3
  - Automatic acknowledgement upon receipt
  - Not redelivered if processing fails afterwards
- MANUAL
  - Use jms:ack operation
- DUPS_OK
  - Similar to AUTO
  - Optimization: acknowledges messages lazily
    - Typically after delay or in bulk
  - Application must handle duplicate message delivery

JMS acknowledgement vs JMS transactions

- **JMS transaction (TX)** means a local transaction involving a JMS broker
- **JMS acknowledgments** are non-transactional messages defined by the JMS protocol for a client to acknowledge receipt of a message to the JMS broker
- JMS **acknowledgements** and JMS **TXs** are mutually exclusive
- JMS TX
  - Is analogous to other transactions
  - Applies to sent and/or received messages
  - Can bundle one or several sends/receives in a TX
- JMS acknowledgments (ACK)
  - Are specific to JMS brokers
  - Only apply to confirming receipt of received messages
  - Performance and granularity can be tuned

Deciding when to use JMS transactions or JMS acks

- Use JMS ack if
  - Acknowledgment should occur eventually, perhaps asynchronously
  - The performance of the message receipt is paramount
  - The message processing is idempotent
  - For the choreography portion of the SAGA pattern
- Use JMS transactions
  - For all other times in the integration you want to perform an atomic unit of work
  - When the unit of work comprises more than the receipt of a single message
  - To simply and unify the programming model (begin/commit/rollback)

Summary

Summary

MuleSoft

- Mule applications can manage TXs
- Demarcate a TX with connectors supporting TX, or in a Try scope
- Can be local or XA based on the transactional connectors involved
- Begin a TX by selecting an appropriate action (ALWAYS_JOIN etc)
- A TX is committed automatically in the absence of an exception
- Rollback is possible in case of failure within TX boundary
- JMS acknowledgement is an alternative to TX and provides various service levels for message acknowledgement
- The saga pattern provides a pseudo-distributed TX for resources such as REST services