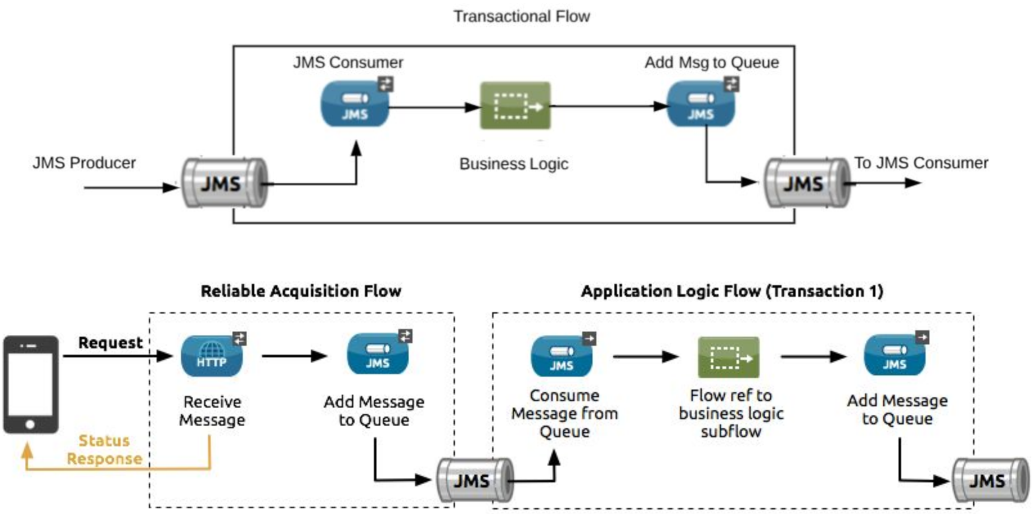
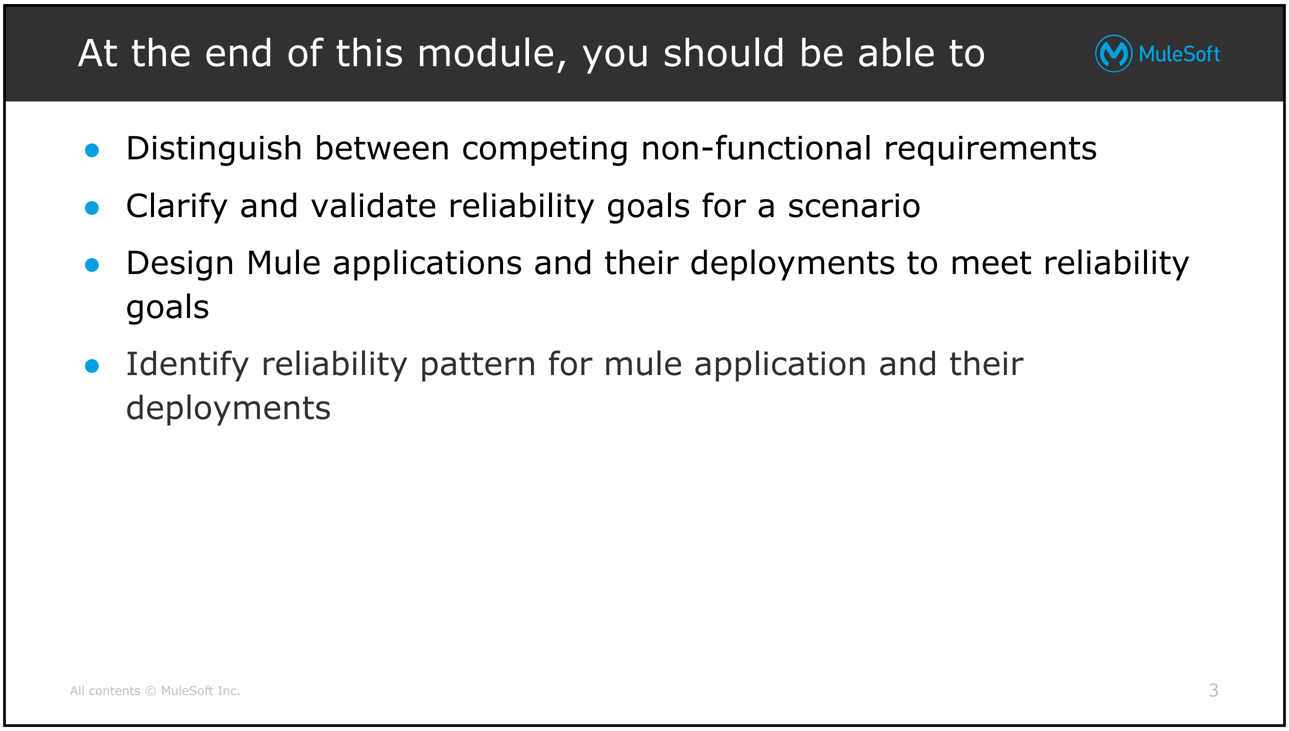
MuleSoft

Module 12: Designing for Reliability Goals

Goal

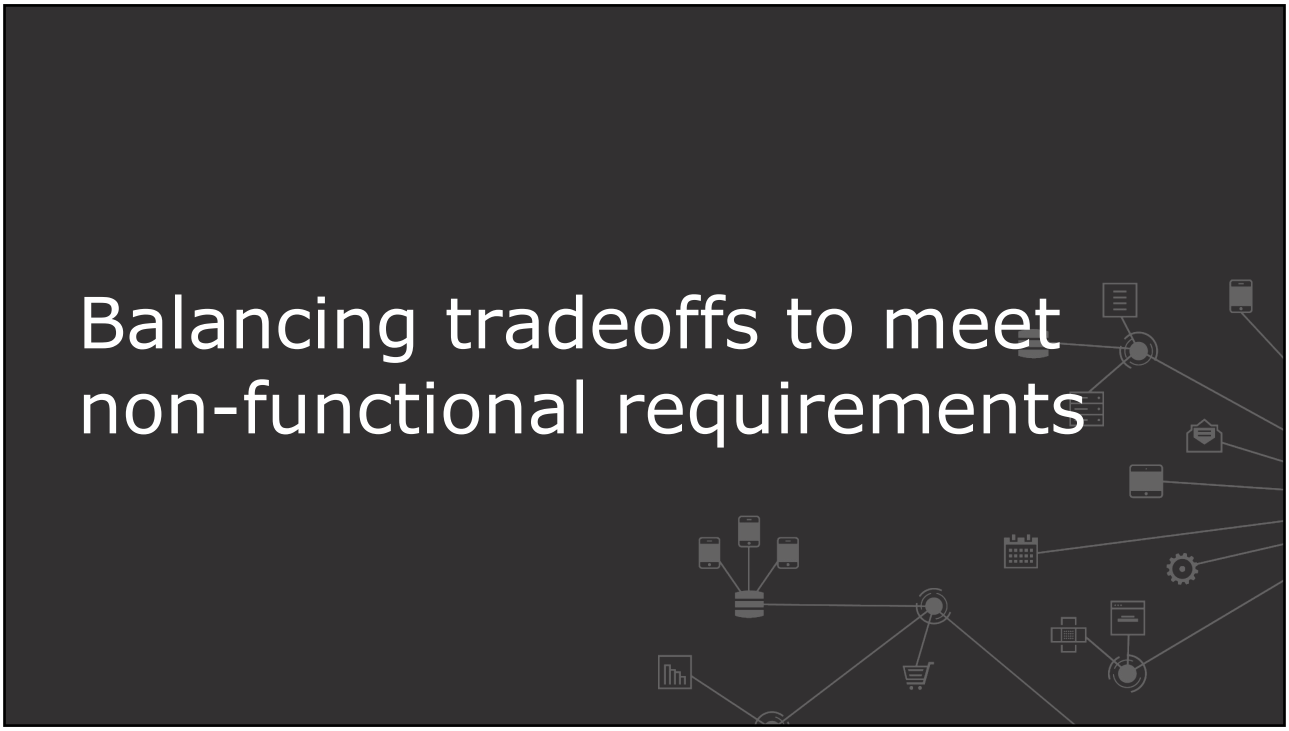
MuleSoft



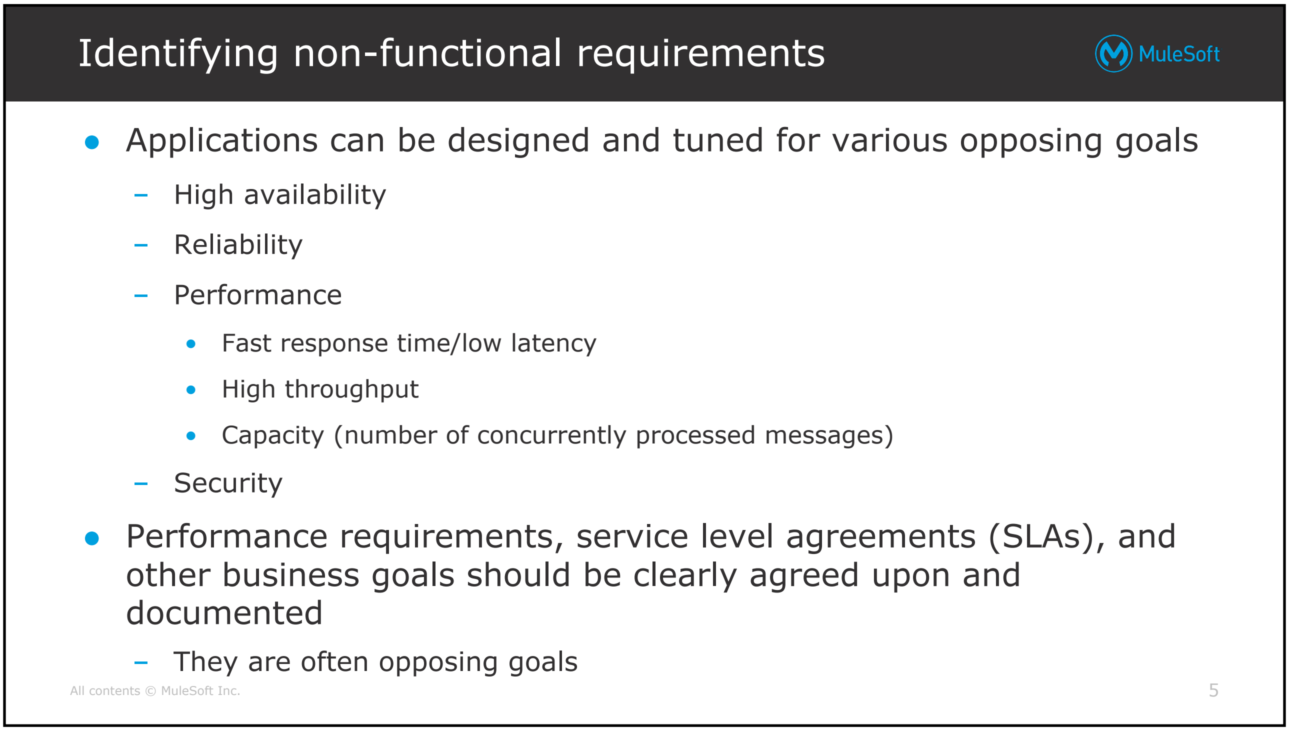


At the end of this module, you should be able to

- Distinguish between competing non-functional requirements
- Clarify and validate reliability goals for a scenario
- Design Mule applications and their deployments to meet reliability goals
- Identify reliability pattern for mule application and their deployments

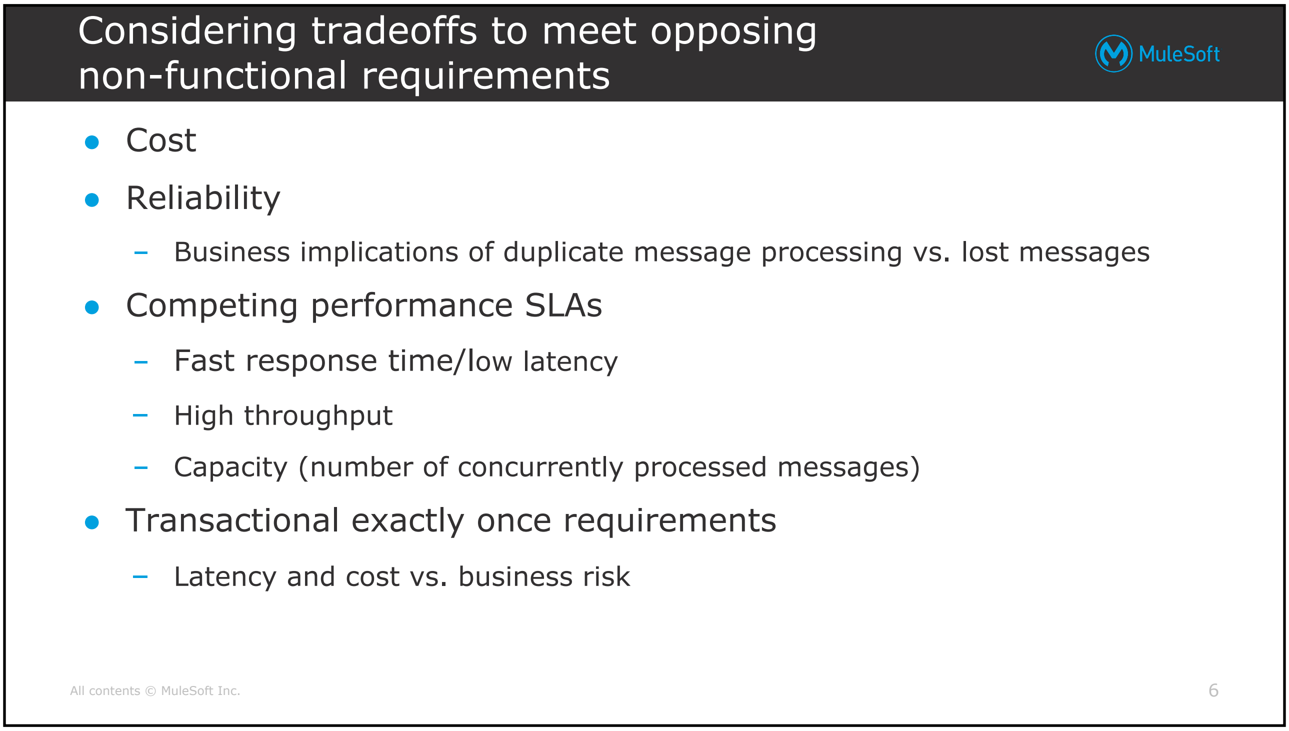


Balancing tradeoffs to meet non-functional requirements



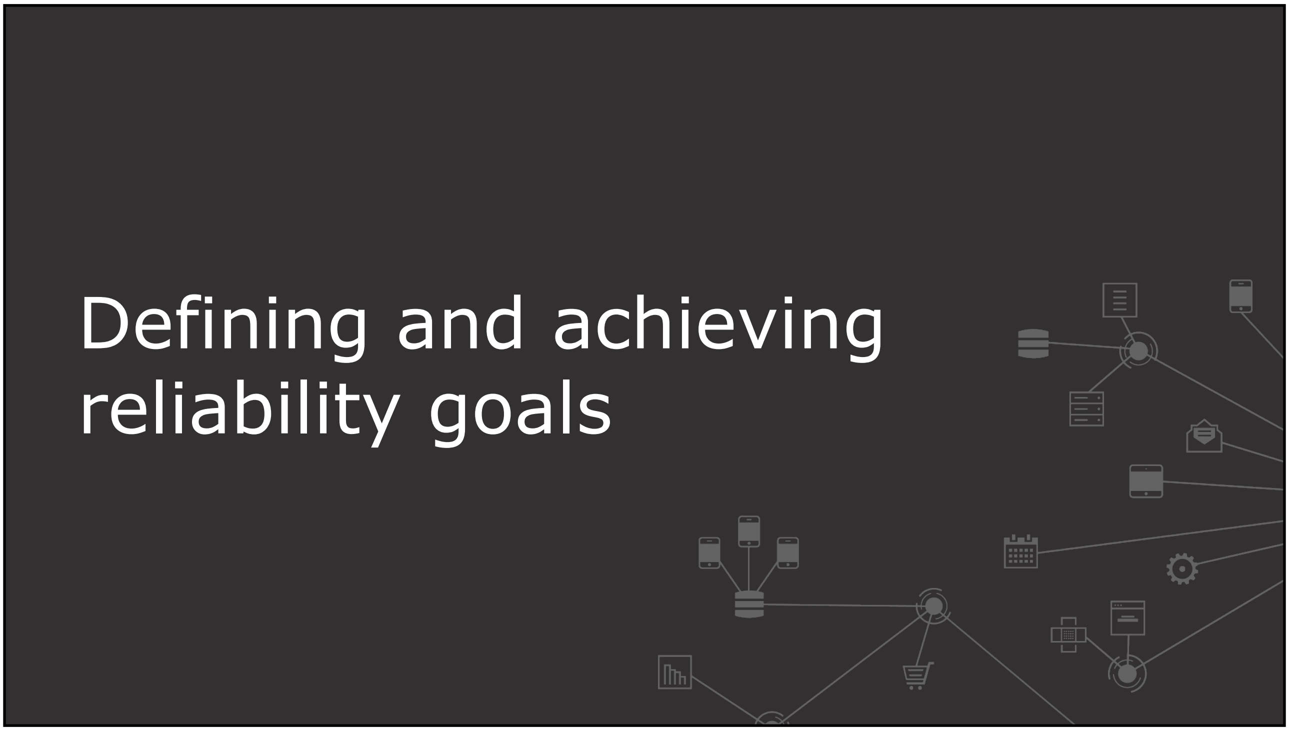
Identifying non-functional requirements

- Applications can be designed and tuned for various opposing goals
  - High availability
  - Reliability
  - Performance
    - Fast response time/low latency
    - High throughput
    - Capacity (number of concurrently processed messages)
  - Security
- Performance requirements, service level agreements (SLAs), and other business goals should be clearly agreed upon and documented
  - They are often opposing goals

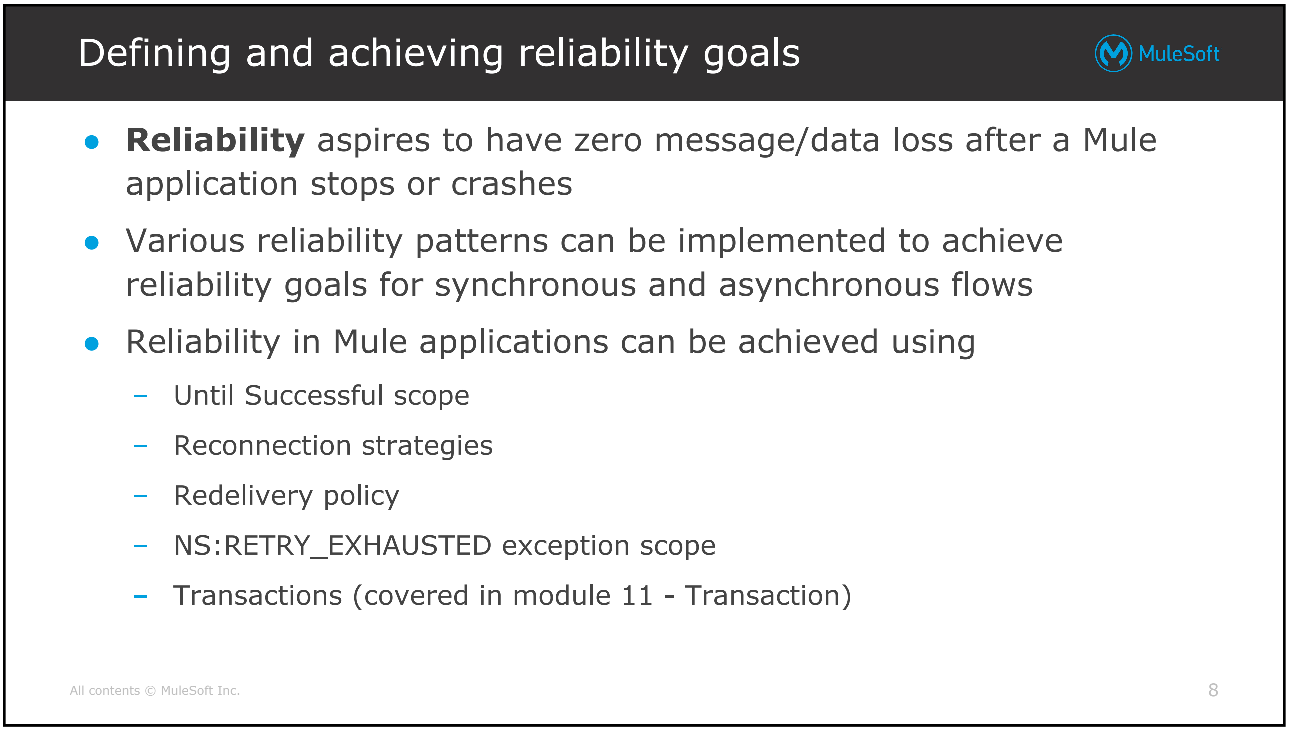


Considering tradeoffs to meet opposing non-functional requirements

- Cost
- Reliability
  - Business implications of duplicate message processing vs. lost messages
- Competing performance SLAs
  - Fast response time/low latency
  - High throughput
  - Capacity (number of concurrently processed messages)
- Transactional exactly once requirements
  - Latency and cost vs. business risk



Defining and achieving reliability goals



Defining and achieving reliability goals

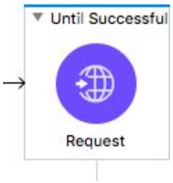
- **Reliability** aspires to have zero message/data loss after a Mule application stops or crashes
- Various reliability patterns can be implemented to achieve reliability goals for synchronous and asynchronous flows
- Reliability in Mule applications can be achieved using
  - Until Successful scope
  - Reconnection strategies
  - Redelivery policy
  - NS:RETRY_EXHAUSTED exception scope
  - Transactions (covered in module 11 - Transaction)

Achieving reliability goals with Mule components

Achieving reliability using an Until Successful scope

MuleSoft

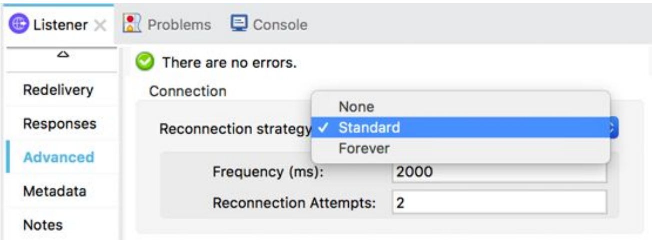
- The **Until Successful** scope repeatedly triggers the scope's components (including flow references) **until they all succeed or until a maximum number of retries is exceeded**
- The scope provides option to control the max number of retries and the interval between retries
- The scope can execute any sequence of processors that may fail for whatever reason and may succeed upon retry



```
<until-successful maxRetries="5"
  millisBetweenRetries="1000">
    <!-- One or more processors here -->
</until-successful>
```

Achieving reliability reconnection strategies

- **System errors** are thrown when a connection to an external system (DB, message broker, etc) fails
- To retry after connection failures, Mule connectors can set a **reconnection strategy**
  - Set for a connector (in the Global Elements Properties) or for a specific connector operation (in the Properties view)



There are no errors.

Connection

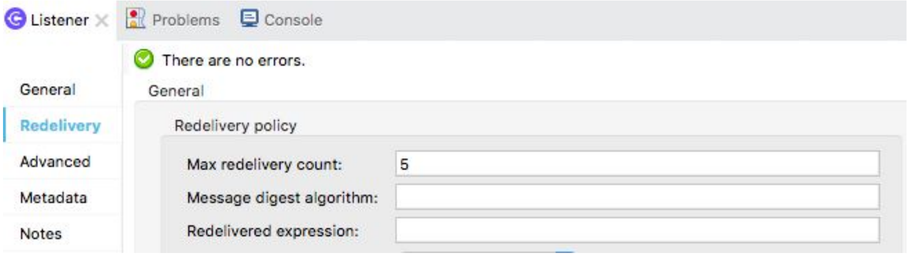
Reconnection strategy: ✓ Standard

Frequency (ms): 2000

Reconnection Attempts: 2

Achieving reliability using a redelivery policy

- A **redelivery policy** is configured on inbound connectors, such as the JMS connector, to specify the number of redeliveries before discarding the message
  - 0 means no redelivery
  - -1 means infinite redeliveries



There are no errors.

General

Redelivery policy

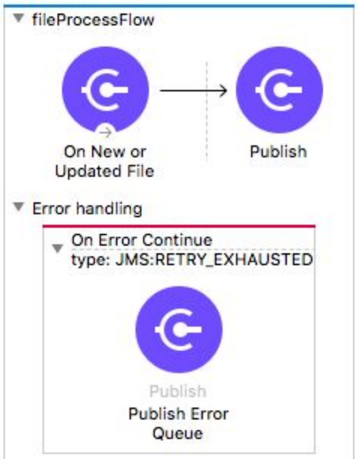
Max redelivery count: 5

Message digest algorithm:

Redelivered expression:

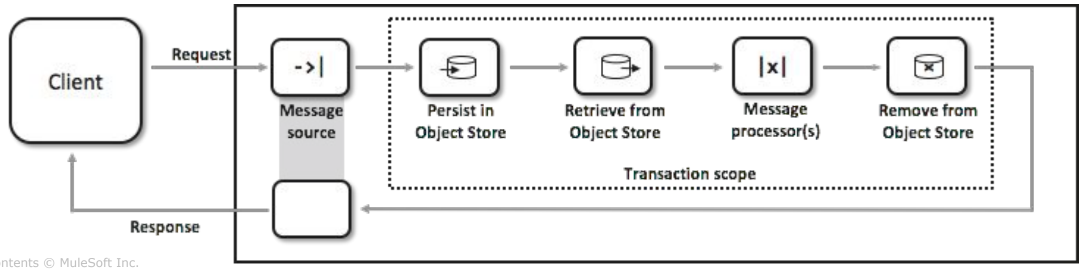
Achieving reliability using MuleSoft components - RETRY_EXHAUSTED exception scope

- Before discarding the message after the number of redeliveries attempted, the connectors raises an exception of type RETRY_EXHAUSTED
- An error scope can handle the RETRY_EXHAUSTED error with logic required to handle the error, so the event is not lost



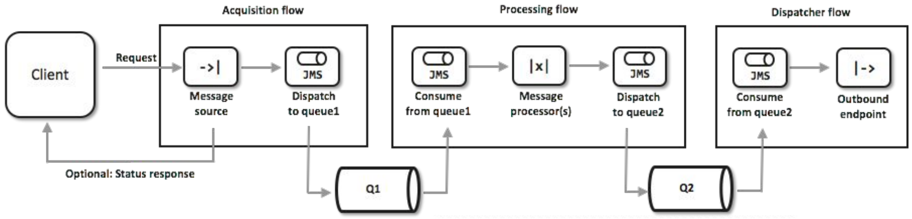
Achieving reliability for transactional systems

- Zero message/data loss for transactional systems is achieved using a **transaction**
- Message Ack is another way to achieve zero message loss
- Message persistence for application downtime/crash is required
- More details are provided in the persistence module



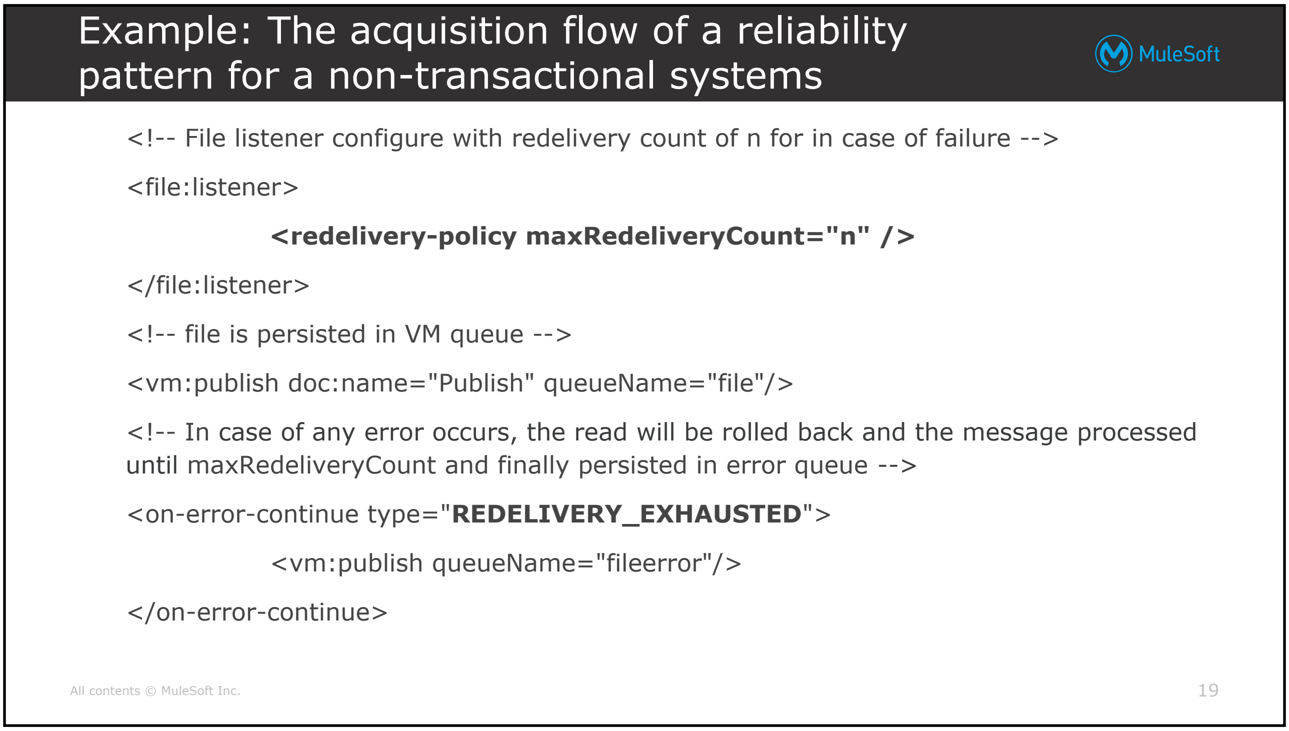
Achieving reliability for non-transactional systems

- Zero message/data loss for non-transactional systems is achieved using a **reliability pattern**
- Splits processing between an **acquisition** flow and a **processing** flow
- The flows do not call each other directly, but use **persisted queues**
- In case of application failure, messages/data are still available in those queues



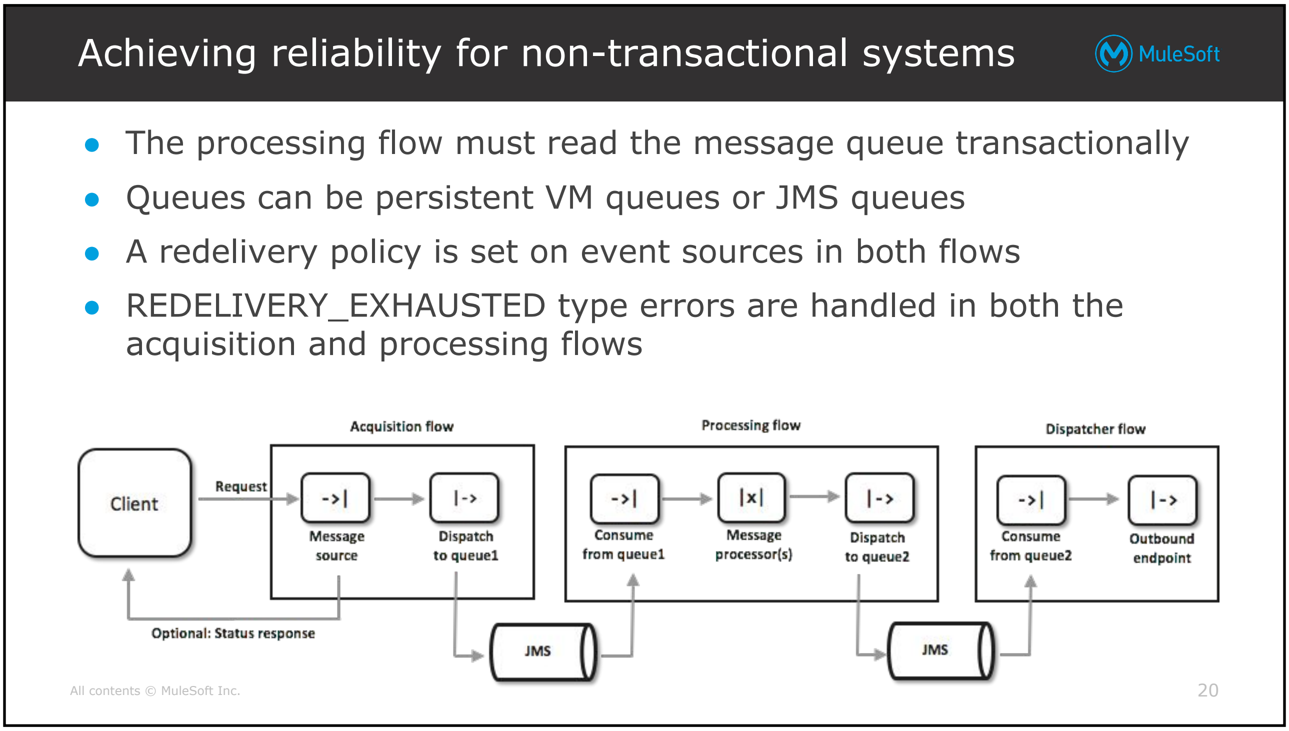
Understanding the two flows in this reliability pattern for non-transactional systems

- The reliability pattern consists of flows with specific responsibilities
  - The **Acquisition flow**
    - Receives incoming messages then dispatches them to a persisted processing queue
    - In case of failure, processes the message until redelivery exhausted, and if that fails, then dispatches the message to a persistent error queue
  - The **Processing flow**
    - Process messages from the processing queue then dispatches them to another persisted queue
    - In case of failure, processes the message until redelivery exhausted, and if that fails, then dispatches the message to a persisted error queue



Example: The acquisition flow of a reliability pattern for a non-transactional systems

```
<!-- File listener configure with redelivery count of n for in case of failure -->

<file:listener>

              <redelivery-policy maxRedeliveryCount="n" />

</file:listener>

<!-- file is persisted in VM queue -->

<vm:publish doc:name="Publish" queueName="file"/>

<!-- In case of any error occurs, the read will be rolled back and the message processed
until maxRedeliveryCount and finally persisted in error queue -->

<on-error-continue type="REDELIVERY_EXHAUSTED">

              <vm:publish queueName="fileerror"/>

</on-error-continue>
```



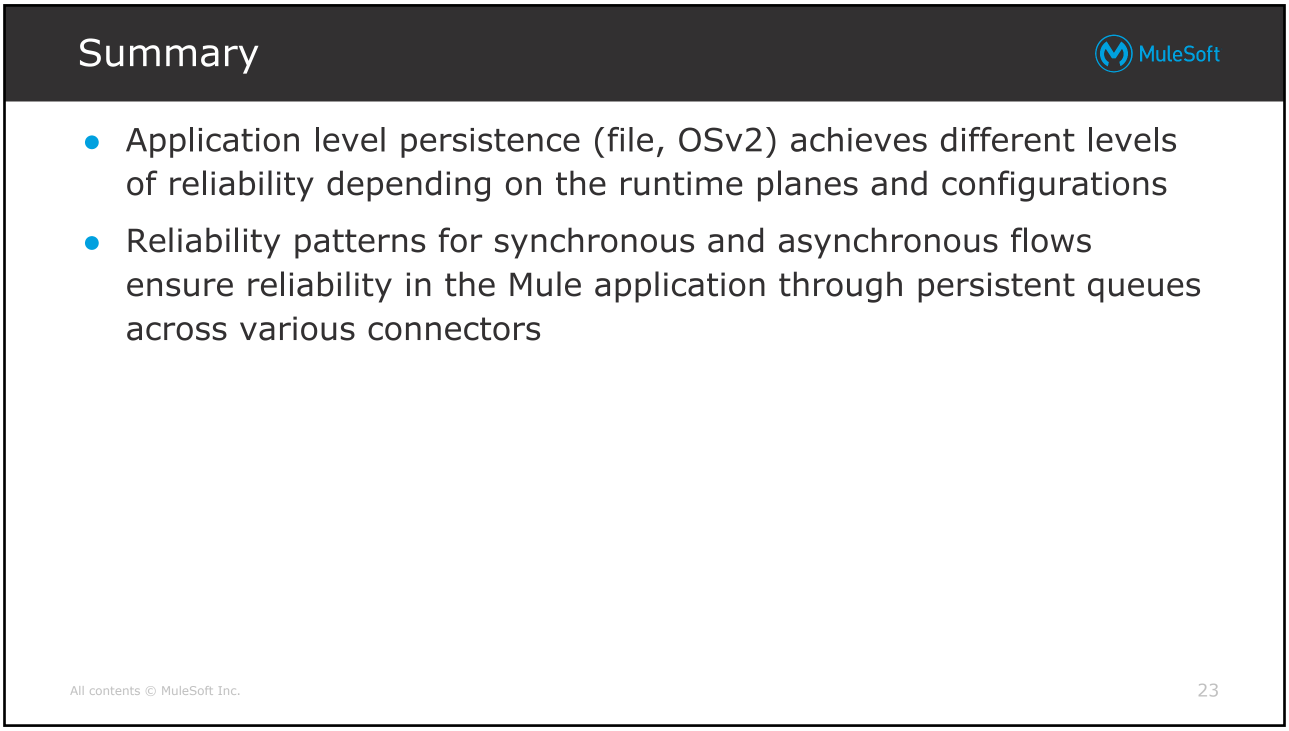
Achieving reliability for non-transactional systems

- The processing flow must read the message queue transactionally
- Queues can be persistent VM queues or JMS queues
- A redelivery policy is set on event sources in both flows
- REDELIVERY_EXHAUSTED type errors are handled in both the acquisition and processing flows

Other ways to achieve reliability with Mule applications

- In an earlier module you already learned about how to store state in a Mule application
  - Object Store
  - Persistent queues
  - File persistence
  - Database connector
  - Caches
  - Other external systems
- Each option has tradeoffs between reliability goals vs. performance vs. cost

Summary



Summary

- Application level persistence (file, OSv2) achieves different levels of reliability depending on the runtime planes and configurations
- Reliability patterns for synchronous and asynchronous flows ensure reliability in the Mule application through persistent queues across various connectors