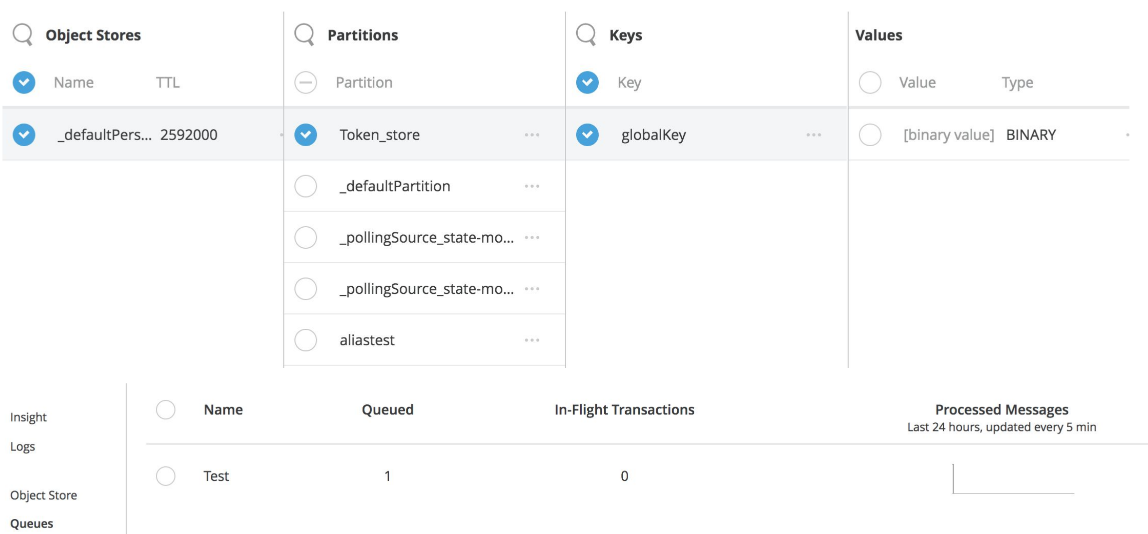
Module 8: Designing with Appropriate State Preservation and Management Options

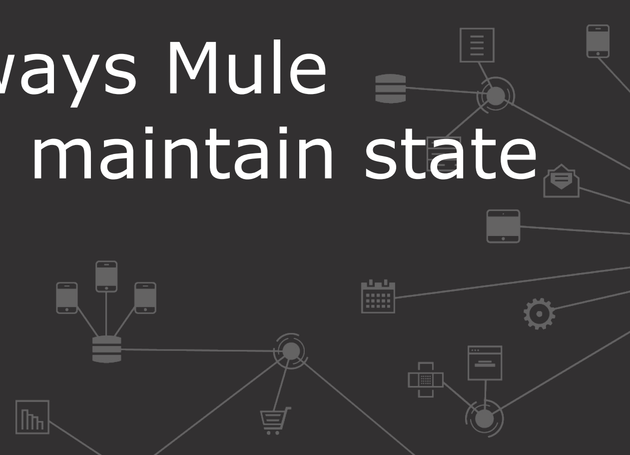
Goal



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

- Decide the best way to store Mule application state in **persistent or non-persistent storage**
- Identify how to store Mule application state using the **Object Store v2**
- Decide the best way to manage storage of Mule application state using **persistent queues**
- Configure Mule application provided **caches** to store Mule application state
- Avoid duplicate processing of previous records using Mule connector **watermarks**

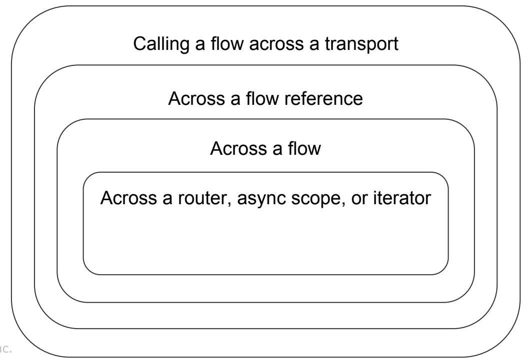
Distinguishing ways Mule applications can maintain state



The various ways the state of a Mule event may need to be available between event processors

MuleSoft

- The Mule event stores state in the payload, attributes, and variables
- These can be passed between event processors and flows within various execution contexts



The various ways state may need to be shared beyond flow invocations

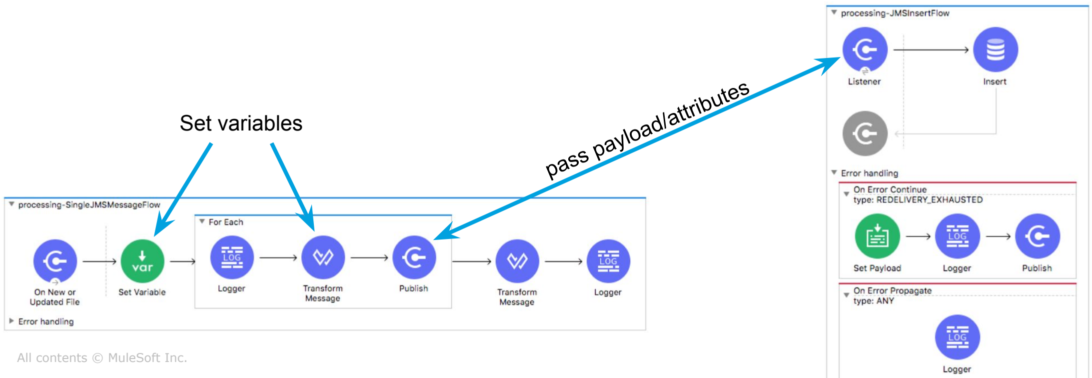
MuleSoft

- Over time, state needs to be available to a particular event processor from other previous flow invocations or even other applications
- There are various levels of guarantees that can be coded and configured beyond one Mule event
  - When all the Mule runtimes restart
  - When the Mule runtime restarts
  - The next time a flow is invoked

Reviewing state management options

MuleSoft

- You have already seen how to pass state
  - **Between** event processor **components** in a flow using variables
  - **Across transports** using attributes, payloads, or attachments
- This state is lost between subsequent executions of the flows



Use cases for saving state in a Mule application

MuleSoft

- Between **flow executions**
  - To remember state from previous outbound HTTP requests in the flow
  - To filter out already processed messages for exactly once processing (idempotency)
  - To **cache** unchanged data or responses to speed up response time
- Between **iterations** (or **steps**) when processing a collection of elements
  - Including **batch processing**
  - Including iterations driven by a **scheduler** or **cron job**
  - For example, to aggregate results, such as a total price or total count
  - Each iteration (or batch step) may occur in parallel, perhaps in separate threads

Persistence guarantees that might be required by particular Mule applications

- Store state **non-persistently** in a single Mule runtime's memory
  - Data can be reused while the application is still running
    - For example, to retry an operation after a transient network glitch
  - Data is lost after application restarts or server crashes
- Store state **replicated** in a **distributed memory data grid cluster** between several Mule runtimes
  - Data survives application restarts or server crashes as long as one server with a replica survives
- Store state **persistently** to disk in a **single** Mule runtime
  - Data survives application restarts or server crashes, but not disappearance of the disk (such as when a CloudHub worker is stopped)
- Back a Mule runtime **cluster** with **persistent database storage**
  - Data survives if every server crashes

Persistence guarantees vs. performance

- Store state **non-persistently** in a single Mule runtime's memory
  - Fastest
- Store state **persistently** to disk in a **single** non-clustered Mule runtime
  - Added latency to write data to the local file
- Store state **replicated** in a **distributed memory data grid or** between several Mule runtimes
  - Added latency to replicate data across the network
- Back a Mule runtime **cluster** with **persistent database storage**
  - Added latency to replicate data across the network to the database
  - This may be the slowest option

When to maintain state in an external system or store

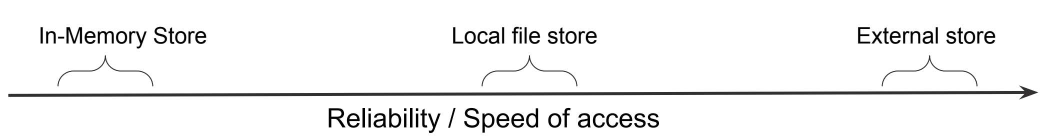
MuleSoft

- An external system or store is required
  - When the Mule application requires additional guarantees, such as **transactional** guarantees
  - To share data among distributed applications, perhaps including non-Mule applications
- Examples include a **database**, an **object store**, or an **external cache**

Tradeoffs between different types of state storage

MuleSoft

- Non-persistent in memory storage is the **fastest** way to access data, but is the least reliable/durable
- Local persistent storage, such as to a file system, is **slower** but more **reliable**, and not shared
- Replicated in-memory data grid storage lies somewhere between in-memory and on-disk
- An external storage system may be the most **reliable** and can be **shared**, but also may be much **slower**



Identifying MuleSoft object store behavior in various runtime planes

What is a MuleSoft Object Store?

MuleSoft

- A MuleSoft **Object Store** is a key-value store implemented using Java
  - The values can be any serializable Java object
  - There is **no query mechanism**, objects are only retrievable by the key
- The interface definition of the MuleSoft Object Store is defined at
  - https://github.com/mulesoft/mule-api/blob/master/src/main/java/org/mule/runtime/api/store/ObjectStore.java
- From Mule application code, an Object Store is typically accessed via the Object Store connector
  - But an Object Store implementation might provide other communication mechanisms, such as a secure REST connection

The design intent of object stores

- The Object Store component was designed to store state information between flow invocations
  - Synchronization information like watermarks
  - Temporal information like access tokens
  - User information
- Several factors change the runtime **behavior** or object stores
  - Whether an object store is configured as **persistent** or **non-persistent**
  - The runtime plane to which the Mule application is deployed
    - CloudHub worker(s)
    - A single customer-hosted Mule runtime
    - A cluster of customer-hosted Mule runtimes

What MuleSoft means by persistence

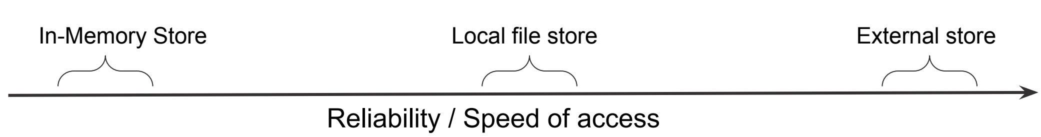
- **Persistence** usually refers to storage that is copied to disk or some externals storage, or replicated across several nodes
- **Non-persistent** usually refers to data that is only stored in the JVM memory of one Mule runtime
- Each object store can be configured as persistent or transient (non-persistent)

What MuleSoft means by durability and reliability

- **Durability** and **reliability** are more general ideas that refer to
  - The runtime plane to which the Mule application is deployed
    - One vs. multiple CloudHub workers
    - One vs. multiple vs. clustered customer-hosted Mule runtimes
  - Other tuning parameters like TTLs, storage limits, etc.
- All these interdependent factors affect the overall performance and SLAs of an objects store, so they must all be considered together to make good decisions

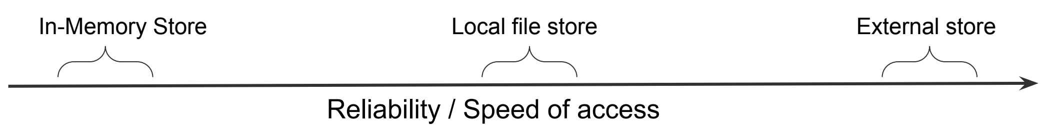
Behavior of various MuleSoft object store implementations based on storage type

- Object store instances can be configured to be **persistent** or **non-persistent** in the Mule application
  - This parameters changes the runtime behavior, speed of access, and reliability of the object store



Behavior of various MuleSoft object store implementations based on storage type

- An object store's runtime implementation and behavior also varies based on the **runtime plane** and **version** and its **infrastructure**
  - Deployments vary between a single customer-hosted Mule runtime, single CloudHub worker, customer-hosted cluster, or multiple CloudHub workers
  - Customer-hosted clusters have additional configuration options that also affect the implementation and behavior



Choosing an object store instance to use in a Mule application

- Every Mule application is provided a **default** object store instance
  - This is automatically selected as the default object store for Mule components that use an object store
  - This object store is automatically configured as a **persistent** object store
- **Additional** object store **instances** (also called **partitions**) can be added as **global elements** to the Mule application

How CloudHub deployments implement a Mule application's object store

MuleSoft

- A persistent object store uses the **Anypoint Object Store** service (OSv2)
  - A **cloud native implementation** of the Object Store interface
  - Data is shared between and accessible by all CloudHub workers
- A non-persistent object store does not use the OSv2 service
  - The object store is implemented locally in each CloudHub worker
  - Data is isolated within each CloudHub worker

Behavior or Mule 4 apps deployed via Anypoint Runtime Manager (RM) to various runtime planes

MuleSoft

Runtime plane and its configuration	Object Store type defined in Mule app	Contents survives server restart (stop/start)?	Contents survives update to new app version (jar with different name)?	Contents shared between all nodes/workers?	Contents visible in RM?
Standalone runtime (previously configured via RM)	Transient Object Store	N	N	N/A	N/A
	Persistent Object Store	Y	N	N/A	N/A
Mule runtime cluster (previously configured via RM)	Transient Object Store	Y (1+ must remain)	Y	Y	N/A
	Persistent Object Store	Y (1+ must remain)	Y	Y	N/A
Multiple CloudHub workers, no OSv2, (persistent queues NOT ticked)	Transient Object Store	N	N	N	N
	Persistent Object Store	N	N	N	N
Multiple CloudHub workers, OSv2, (persistent queues NOT ticked)	Transient Object Store	N	N	N	N
	Persistent Object Store	Y	Y	Y	Y

Note: Application upgrade implies deploying a Mule application deployment package (jar) of a different name

When to code custom object store implementations

- In rare cases, custom object stores can be coded in Java
- This allows the Object Store connector to connect to a custom object store, such as an external database or data grid
  - https://dzone.com/articles/custom-object-store-in-mule
- An architect should evaluate the need to develop and maintain custom Java code, vs. reusing or coding a Connector to the external service, or using an external API directly

Configuring the storage behavior of object stores and VM queues in a cluster of Mule runtimes

- The object store is implemented in a Hazelcast data-grid
- The Hazelcast data-grid (the cluster) itself can be configured with different **store profiles**
- With the default **reliable** store profile
  - VM queues are distributed across the data grid, providing HA and reliability behaviors
  - Object store are replicated across the data grid
- With a **performance** store profile
  - Distributed queues are disabled, using local queues instead to prevent data serialization/deserialization and distribution in the shared data grid
  - The object store is implemented without replication to other nodes

Configuring the performance profile of a Mule runtime cluster

MuleSoft

- A Mule runtime can set a performance or reliability profile
  - The default is reliable

```
mule.cluster.storeprofile=performance
```

- Each Mule application can override this setting in its configuration global element

```
<configuration>
  <cluster:cluster-config>
    <cluster:performance-store-profile/>
  </cluster:cluster-config>
</configuration>
```

```
<configuration>
    <cluster:cluster-config>
      <cluster:reliable-store-profile/>
    </cluster:cluster-config>
  </configuration>
```


Configuring the durability of a Mule runtime cluster

MuleSoft

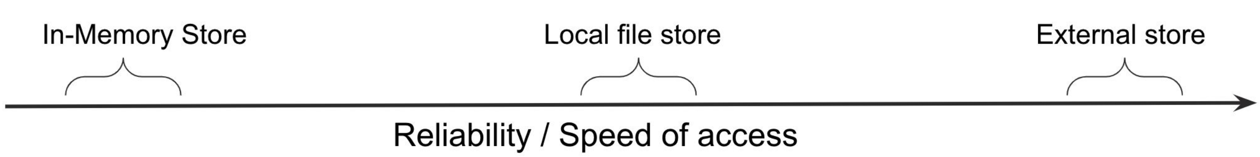
- A Mule runtime cluster can be manually configured in each Mule runtime's configuration
- A **quorum** size sets the minimum number of Mule runtimes that must be in the cluster in order Mule applications to run and accept inbound requests
  - This also sets the number of nodes to which data is replicated
- A JDBC store can also be configured to store cluster data from VM queues and object stores
  - So data survives restarting the cluster after all the nodes stop or crash

Reflection questions

- How is an object store implemented for a single Mule runtime?
  - What are the persistence options and how does this change the implementation?
- How is an object store implemented for a server group of Mule runtime?
  - What are the persistence options and how does this change the implementation?
- How is an object store implemented for a cluster of Mule runtime?
  - What are the persistence options and how does this change the implementation?
- How is an object store implemented for a one or more CloudHub workers?
  - What are the persistence options and how does this change the implementation?
- What are the limitations to exchange data from each of these object stores?

Exercise 8-1: Identify why and when Mule applications need to maintain state

- Identify some scenarios where a Mule application needs to maintain state



Exercise steps

- Provide some example scenarios where state need to be maintained, and how is it maintained
  - Between applications
  - Between flows
  - Between HTTP requests
  - When schedulers are used
  - When a collection of records is processed
  - When a collection processes records out of order in batch processing
- What are the benefits and tradeoffs of storing state in these use cases
- How does the choice or runtime plane affect these decisions?

Exercise solution: When and why should Mule application state be maintained?

- To maintain the state of the Mule application between flow executions
  - To store a watermark (like the last file modified data) to avoid duplicate consumption of triggering data that has not changed
    - For example, to avoid duplicate synchronization between a source connector and target connector(s)
  - Multiple flow executions, triggered by different Mule events, perhaps asynchronously, but need to contribute to a common result
    - Such as aggregating (sum, average, …) or accumulating (adding to list) values
  - Distinct incoming messages (such as HTTP requests) may contain duplicates, while avoiding duplicate processing

Exercise solution: When and why should Mule application state be maintained? (Cont.)

- To share variables between flows in a Mule application that are called by connectors rather than by flow references
  - However, it's more likely variables will be passed to the connector in the event payload or attributes
- To maintain authentication, session, or any other data which refreshes periodically or rarely
  - For example, to maintain a local lookup table of data or attributes that is populated from external sources that does not change often

Differentiating between the two Anypoint Object Store service versions

There are two versions of the Anypoint Object Store service

- Available in the MuleSoft-hosted runtime plane (CloudHub)
- Both versions store key/value pairs in an external online object store
- Both versions are secure, highly available online services
- Both versions can be used in a Mule application using the Object Store connector, when the Mule application is deployed to CloudHub
  - Mule 3 applications can also be configured to use OSv2
  - Mule 4 applications can only use the Anypoint Object Store v2 (OSv2)
- OSv2 also has a REST API for external applications to share object store data

Identifying the behavior of the Anypoint Object Store v2

- Simple key value store
- Synchronized access at key level
- Uses TLS for transport and FIPS 140-2 compliant encryption standard for data at rest
- Designed mainly for
  - Storing synchronization information
  - Storing short-lived information like access tokens
  - Storing user information
- Supports unlimited keys and 10 MB max value size
- Is preserved upon application redeployment and restarts
- Is deleted upon application deletion

Limitation of The Anypoint Object Store

- Not a universal data storage solution
- Does not support transactions
- Does not replace an actual DB and not suitable for searches, queries, etc.
- Data is automatically removed after a set amount of time
  - 30 days for OSv2

Creating and accessing MuleSoft object stores

How a Mule application creates and uses MuleSoft object stores

- The **Object Store module** can be added to a Mule application or Mule domain
- This module is used to operate on object stores from the Mule application
  - Create new object stores as global elements
  - Use Object Store operations to store and retrieve data from an object store
- Depending where an object store is defined, it may be accessible with other Mule applications
  - An object store defined as a **Mule domain's** global element **can** be shared between Mule applications in that Mule domain
  - An object store defined as a **Mule application's** global element **cannot** be shared with other Mule applications

Deciding between object store types

Object store instances available to Mule applications

- Every Mule application is automatically provides a **default object store** that is **persistent**
- Other **global object stores** can be defined as global elements and used by any components in the Mule application or Mule domain
  - Used to isolate data access to a subset of components in the Mule application
  - Each global object store can have a different **partition name** to separate its stored data from other object stores used in the Mule application
  - Any additional persistent object stores use the same type of runtime storage as the default object store
  - Non-persistent object stores are always stored locally in the Mule runtime's memory

https://blogs.mulesoft.com/dev/anypoint-platform-dev/new-mule-4-objectstore-connector/

How data is stored in the default object store

- You already saw that the persistence option is implemented differently depending on the runtime plane type
  - The runtime behavior also depends on the number of Mule runtime's to which the Mule application is deployed
- In the default object store, data is saved
  - To an online object store service when the Mule application is deployed to CloudHub
  - To shared distributed memory when the Mule app is deployed to a cluster of Mule runtimes
  - To a local file system when the Mule app is deployed to a standalone Mule runtime
- Mule applications deployed to customer-hosted infrastructure can use the OSv2 REST API to store data in OSv2 and retrieve it

Blocking access to an object store from other Mule application components

- A **private object store** can be configured by a particular component, to securely hide its object store data from any other component in the Mule application
- Instead of configuring a global object store reference, certain components provide additional child elements to configure the object store
  - The object store configuration options (partition name, persistence, storage limits, etc.) are the same as for a global object store configuration

https://blogs.mulesoft.com/dev/anypoint-platform-dev/new-mule-4-objectstore-connector/

Ways to use various types of object stores to store state

- A global object store can share state
  - Between Mule components in a Mule application
  - Among the replicas of a Mule application executing on different nodes of a customer-hosted Mule runtime cluster
  - For OSv2, between the workers of a multi-worker CloudHub deployment of a Mule application
- Use a private object store when sharing data between components in the flow is deemed to be a security risk
- The Anypoint Platform Object Store v2 can also be used to share state between different Mule applications and distributed Mule runtimes using the OSv2 REST API

Reflection questions

- What is the difference between a global object store and a private objects store?
- Which of these object store types can be persistent?
- Which of these object store types can be transient?
- What type of persistence is used by the default object store?
- How is the default object store implemented in each of the possible runtime planes?

Reflection questions

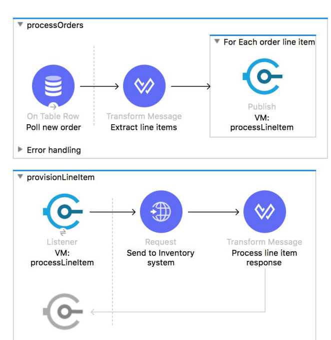
- What use cases are a good fit for using an object store?
- What use cases should not use an object store?

Storing Mule application state using persistent VM queues

How VM queues are used in Mule applications

MuleSoft

- Mule applications can use VM queues
  - Each VM queue persists state in the Mule runtime's Java virtual machine in a first in first out (FIFO) order
  - Depending on the runtime plane, each VM queue may be shared
    - When a Mule application is deployed to a cluster of multiple Mule runtimes, the VM queue may be shared between all replicas of of the Mule application
    - If the VM queue is in a Mule domain, several Mule applications on that Mule runtime can share the VM queue
- VM queues can be configured to persist after all Mule runtimes restart
- Other messaging systems can provide other SLAs and guarantees
  - Including sharing messages outside Mule apps



Types of VM queues supported by runtime planes

- Queues can be **transient** or **persistent**
- Transient queues are **faster** than persistent queues, but less reliable in case of a system crash
- Conversely, persistent queues are **slower** but more reliable
- For the customer-hosted runtime plane, queue persistence behavior also depends on the number of Mule runtimes and their dependencies
  - For standalone Mule runtimes, persistent queues are stored to the **local disk**
  - For a cluster of Mule runtimes, persistent queues are backed by the **cluster's distributed data grid**

VM queues behave differently based on the cluster's performance profile

- A cluster can be configured with a **reliable** or **performance** profile
- With a **performance** profile, VM queues and object stores do not use the data grid
  - Data is stored locally in the memory of the node in which the Mule application connects to the VM queue or object store
  - With multiple nodes, VM queue and object store data is split up and isolated

How persistent VM queues work in the MuleSoft-hosted runtime plane

MuleSoft

- In CloudHub, persistent queues can retain certain Mule application data such as messages in VM queues after service outages
  - Slower, but more durable, than in-memory storage
  - Currently implemented using Amazon SQS storage
- The queue storage is automatically provided by the Anypoint Fabric service
  - Persistent queuing has no message limit
  - maxOutstandingMessages attribute is set to limit the number of messages saved in each VM queue

When and how to use persistent VM queues to share state

MuleSoft

- Share messages (events) between the flows in a single Mule application
- Share messages (events) between Mule applications running in the same Mule domain(on-prem only)
- Share state (events) between Mule application instances deployed to multiple customer-hosted Mule runtimes (nodes) in a cluster or CloudHub workers

Limitation of persistent VM queues

- All objects persisted must be serializable
- Overly complex structure may cause serialization error or performance issue
- Persistence queues may not guarantee that a message is delivered only once

Deciding to use persistent VM queues instead of another 3rd party messaging solution

- Use persistent queues when the use case does not require
  - Another more durable or more reliable state management option such as JMS or DB
  - Or to share state between multiple Mule applications or with non-Mule applications
- Persistent queues are not available
  - Between different Mule applications, especially deployed to different Mule runtimes
  - To non-Mule applications
- Persistent VM queues do **not** provide operation teams any advanced queue management features
  - Especially not in customer-hosted deployments

Managing state with file-based persistence

How file-based persistence works in customer-hosted runtime planes

- In **file-based persistence**, the data for state management is stored in files on the machine hosting the Mule runtime
  - Depending on the runtime plane and the Mule application's configuration, VM queues can also be persisted to files
- When a Mule application is deployed to **Customer-hosted** Mule runtimes
  - The account running Mule must have read and/or write permissions on the specified directories
  - VM queues can persist data in a file-based store
- File persistence is **less reliable, durable,** and **persistent** in Runtime Fabric compared to in a standalone Mule runtime

Do not use file-based persistence works for Mule applications deployed to CloudHub

- Mule applications deployed to CloudHub have **only limited** and **ephemeral** file system access
  - EC2 disk storage is removed when the CloudHub worker (EC2 instance) is removed
  - The File connector can only access specific folders such as /tmp or /opt/storage, depending on the CloudHub worker size

Limitation of file-based persistence

- File storage is typically less performant than in-memory storage
- File persistence does not work across nodes/workers/servers in replicated deployments of Mule applications
  - The file-system is not shared between CloudHub workers and also typically not within a Mule runtime cluster
- File storage is not transactional
  - Does not participate in transactions demarcated by a Mule application

Managing state with external storage systems

How external state management works

- Mule applications can also use connectors to external state management systems
  - Data is stored in a database, cache, or a cache backed by a database or API
- Different external stores provide various **quality of service** levels
  - Peristsence
  - Transactional
  - Replication
  - Various eviction policies (Least frequently used)
  - High availability through cluster
  - Fast data retrieval through partitioning
  - Automatic failover
- Various options support a wide range of data structures
  - String, Maps, List, Set, Blob dependent on data stores
- MuleSoft has connectors for Redis and MongoDB

When to use an external store to store and manage Mule application state

MuleSoft

- Pros
  - May provide more exacting performance, reliability, and durability guarantees
  - May be a preferred option to achieve certain high availability or failover goals
  - Use when backup and replication is required for cache objects
- Cons
  - Additional cost, management, and staffing requirements
  - Added layers and complexity

Limitation of external store

MuleSoft

- Need to maintain one more system for state management
- High network latency as a result of network call to external system

Behavior or Mule 4 apps deployed via Anypoint Runtime Manager (RM) to various runtime planes

MuleSoft

Runtime plane and its configuration	Queue type defined in Mule app	Messages survive server restart (stop/start)?	Messages survive update to new app version (jar with different name)?	Messages shared between all nodes/workers?	Messages visible in RM?
Standalone runtime (previously configured via RM)	**Transient queue**	N	N	N/A	N/A
	Persistent queue	Y	N	N/A	N/A
Mule runtime cluster (previously configured via RM)	**Transient queue**	Y (1+ must remain)	Y	Y	N/A
	Persistent queue	Y (1+ must remain)	Y	Y	N/A
Multiple CloudHub workers, no persistent queues, (OSv2)	**Transient queue**	N	N	N	N
	Persistent queue	N	N	N	N
Multiple CloudHub workers, persistent queues, no queue encryption, (OSv2)	**Transient queue**	Y	Y	Y	Y
	Persistent queue	Y	Y	Y	Y

Note: Application upgrade implies deploying a Mule application deployment package (jar) of a different name

Designing Mule applications that use the Cache scope

State management using caching

- The Cache scope maintains the state of **multiple requests** in a Mule application over a particular time frame
- Avoid **duplicate** processing
- Maintains the state of requests in and **in-memory or persistent object store, or in an external third party store**

Limitation of caching

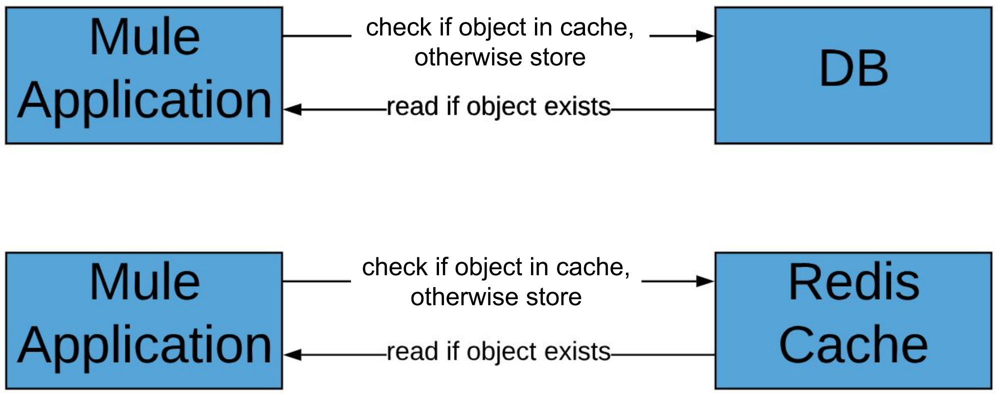
- Does not cache consumable payloads such as a stream
- An in-memory cache is fastest but least persistent
- In Mule 4, the cache scope does not directly support external stores such as DB, Redis
  - Can be done using the Mule SDK

Reflection questions

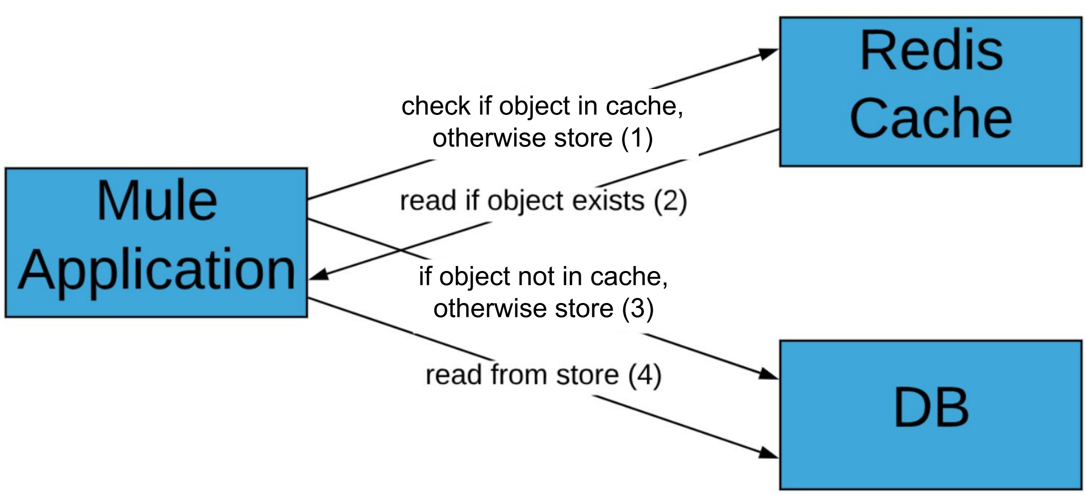
- What use cases should use a local cache?
- What use cases should use a cache backed by an external store?
- What types of external stores are possible, and how do you decide the best choice?
- What use case would benefit from an intermediary API between caching operations in the Mule application and an external store?

Designing with an external cache

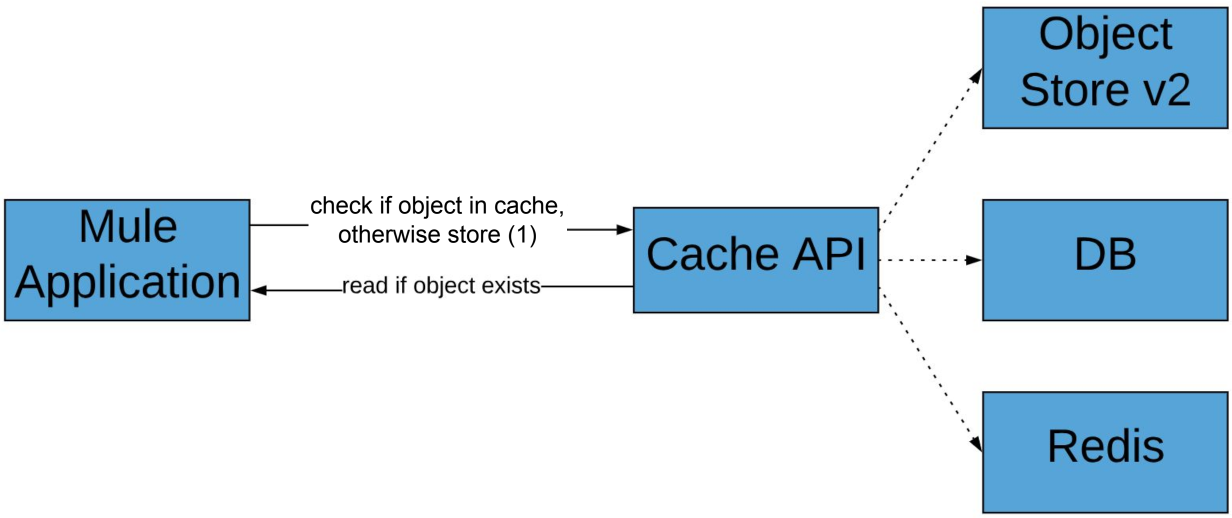
Configuring Mule applications to use a local cache plus external storage



State management using cache backed by store



State management using custom API



Avoiding duplicate processing using watermarks

What is a watermark?

- A **watermark** is a form of state (in other words, a value) that is stored during a recurring processing cycle, such as to process a collection of records
  - During the next processing cycle, the watermark is retrieved and compared against the new corresponding values in the next processing cycle
  - Only records with a new value **bigger** or **higher** than the previous watermark are processed
- Examples
  - The timestamp of a previously processed files
  - The last account id processed in a group of records, where the account id is **always increasing**

Types of watermarks typically used in applications

- **Automatic**
  - The saving, retrieving, and comparing is automatically handled for you through an object store
  - Available for several connector listeners
    - On New or Updated File
    - On Table Row
  - Restricted in how you can specify what items/records are retrieved
- **Manual**
  - You handle saving, retrieving, and comparing the watermark
  - More flexible in that you specify exactly what records you want retrieved

State management using watermarking

- There is a watermarking option for the **On New and Updated File** operation for the family of file connectors
  - There are two watermarking modes
    - CREATION_TIMESTAMP
    - MODIFIED_TIMESTAMP
- The Database connector has watermarking option an **On Table Row** operation that is triggered for every row in a table
  - On each poll, the component will go through all the retrieved rows and store the maximum value obtained

Limitation of watermarking

- Not all connectors supports watermarking
- Does not supports **multiple columns** in DB connectors
- Watermark attributes are limited for each connector
- Asynchronous processing may not deliver the watermarked value in the correct increasing order, which might cause new records to be skipped

Reflection questions

- How does an On Table Row operation's watermark work?
- How does this feature compare with using a Scheduler and any other database operation?
- How does an On New or Updated File operation's watermark work?
- How does this feature compare with using a Scheduler and any other database operation?

Deciding the best state storage and state management options

Deciding state management options for a specific use case

- Some factors involved when deciding the best state storage and management options for a use case
  - What are the performance goals and how can state storage options help meet those goals?
  - Can caching avoid duplicate processing?
  - What part of requests or response events should be stored or cached?
  - Does stored data need to be encrypted, and if so what are the tradeoffs?

Exercise 8-2: Design state management for a polling use case

- Identify ways to improve performance by storing and managing the state of Mule events in a flow
- Identify ways to avoid duplicate data processing by storing and managing the state of Mule events in a flow
- Decide when the state of response data should be stored
- Decide when stored state data should be encrypted
- Identify the best integration style for a scenario

Exercise context

MuleSoft

- A scenario that may require state management
  - The mule flow is **polling** a **backend** system **API multiple times**
  - The **backend** system API call is takes **over 30 seconds** to respond
  - The **response** from the **backend** system contains **PII (Personally Identifiable Information) data**
  - Sensitive data should be encrypted and secured within the Mule application
  - What is best solution to **improve performance** by managing state from the flow
  - Architect your integration solution to best fix this particular use case

Exercise step: Compare state management options in the context of the deployment model

MuleSoft

- Compare state management options based on performance and reliability goals in context of the deployment model

Runtime Plane	Object store v1	Object store v2	Persistent queue	File store	Cache	Watermark
Access Control						
Persistence lifespan						
Cluster Support						
Has REST API						

Exercise step: Decide the best state management option based on performance and reliability goals

Runtime Plane	Object Store v1	Object Store v2	Persistent Queue	File Store	Catch	Watermark
Automated failover						
HA (multiple workers)						
Transactional						
Encryption						
Performance						
Replication						

Exercise steps

- Decide the following questions
  - How can you **improve** the **delayed response time** cause by the API taking more than 30 seconds to respond?
  - Is there a way to **avoid calling the same API repeatedly**, and what are the benefits of avoiding these repeated API calls?
  - Is **state management needed**, and if so, why?
  - How can state management help this use case?
  - If state management is needed, **what** Mule application **state** or **objects should be stored** (Mule event, request and response)
  - If you design to avoid repetitive API calls, then how should the Mule application **send back its response**?

Exercise steps

MuleSoft

- Decide these follow up questions
  - Should responses that were sent back to clients be stored?
  - When should state management objects be stored?
  - What kind of state management option should be used?
    - File, Persistent Queues, Object Store v2, Caching, Watermark
  - Can PII data be stored in the Mule application's store?
  - Must data be encrypted in the store, and if so, how?
  - Should the component using the store restrict access from other components in the Mule application?
- Agree with the rest of the class on the requirements for this use case

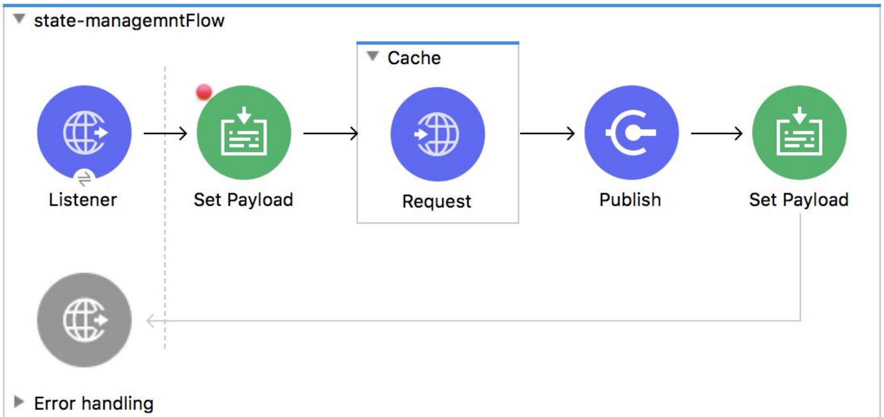
Exercise steps

MuleSoft

- Use Anypoint Studio to mock a flow to meet the agreed requirements
- After everyone completes a solution, discuss and compare all the solutions

Exercise solution

MuleSoft




```
<ee:object-store-caching-strategy name="Caching_Strategy" >
        <os:private-object-store alias="aliastest" />
</ee:object-store-caching-strategy>
```


Summary

Summary

MuleSoft

- Applications often require various persistence guarantees to store state
- Mule applications can leverage various features to more easily meet persistence guarantees
- **Object Stores** persist and share a watermark (or other data) across flow executions
- Use persistent queues for managing state of application in case of failure of application or Mule runtime
- Use Caching to avoid intensive processing for repetitive payload
- Persists cache objects in object store to share across requestsUse a watermark to keep a persistent variable between scheduling events

Summary

MuleSoft

- Applications often require various persistence guarantees to store state
- Mule applications can leverage various features to more easily meet persistence guarantees
- Use the **Object Store** connector to persist and share a watermark (or other data) across flow executions
- Use persistent queues for managing state of application in case of failure of application or Mule runtime
- Use Caching to avoid intensive processing for repetitive payload
- Persists cache objects in object store to share across requestsUse a watermark to keep a persistent variable between scheduling events