Best Practices & Guidelines for Mule ESB

- Introduction
- Context
    - On-Cloud
    - On-Premise
- Operational View
    - Application Logging & Exception Handling
        - Logging
        - Exception Handling
    - Default System Exceptions
    - Flow Exceptions
    - Custom Exception
    - High Availability
    - Scalability
        - Horizontal Scaling
        - Vertical Scaling
    - Reusability & Extensibility
    - Messaging
    - Security
    - Monitoring
        - Mule Agent
        - Custom JMX Monitoring
        - CloudHub
        - Custom Business Events
    - Caching Considerations
    - Capacity Planning & Performance
    - Service SLA
    - Performance Tuning
    - Deployment
        - On-Cloud
        - On-Premise
    - Unit Testing
    - Functional Testing
    - Continuous Integration
    - Debugging
- Best Practices, Conventions and Standards
    - Naming Conventions
    - Project Layout
    - Properties Layout
    - Flow Design and Variables
    - Testing Strategy
    - Mule Credential Vault
- Integration Patterns
    - Synchronous Interactions
    - Synchronous Interaction with Asynchronous Processing
    - Asynchronous Interaction with No Reliability
    - Asynchronous Interaction with Reliability
    - Batch Data Exchange
    - Mediation Pattern
    - Service Orchestration
- API Design Strategy
    - RESTful API Modeling Language (RAML):
    - From design to implementation
    - Documentation
- Validation Strategies
- Versioning Strategies
    - Schema Versioning
    - Application Versioning
    - API Catalog
    - Transactions
- Connectors
    - Reconnection Strategy
- API Platform
    - Environment Configuration
    - API Life Cycle
    - Enhancements based on roadmap
- Security - Confidentiality

        - API Keys
        - API Authentication and Authorization
        - Encryption
        - Security - Integrity
        - Mule Credential Vault
- Logging & Auditing
- Appendix
        - Common Resources

Table of Figures

Figure 1: High Level Architecture - On-Cloud
Figure 2: High Level Architecture - On-Premise
Figure 3: Exception Handling Strategies
Figure 4: Flow-Choice Exception Strategies
Figure 5: Custom Exception Flow
Figure 6: JMeter Initial Setup
Figure 7: Project Layout
Figure 8: Dissection of the Mule Message
Figure 9: Synchronous pattern with Async Scope
Figure 10: Sync-Asynch Processing
Figure 11: Asynchronous interaction with Reliability
Figure 12: Batch Module
Figure 13: Example Batch Processing
Figure 14: Mediation Pattern
Figure 15: Service Orchestration

Introduction

The purpose of this document is to capture Mule ESB based Integration flow development best practices and guidelines based upon our vast industry experience in implementations spanning from simple to mission critical solutions. This document is a collection of standards, conventions and guidelines for writing any Mule flows that is easy to understand, maintain, and enhance. Most of the standards are based on MuleSoft's best practices.
The purpose of this document can be summarized as per the points below:

- Document a set of best practices both at the implementation level and at the architectural level to assist the projects that are adopting the MuleSoft technology in their technical decision making process. The implementation best practices are at a more detailed technical level and their primary purpose is to assist the developers, whilst the architectural best practices are more geared towards the technical and solution architects.
- The document aims to cover common use-cases and provide a quick reference for each approach from an 'integration-pattern' perspective and describe how developers can leverage MuleSoft platform capabilities to implement those patterns.

This document will cover for both On-premise Mule ESB and On-Cloud Mule ESB best practices.

Context

On-Cloud

Below diagram captures all the required infrastructure required for building an on-cloud based Mule ESB environment.

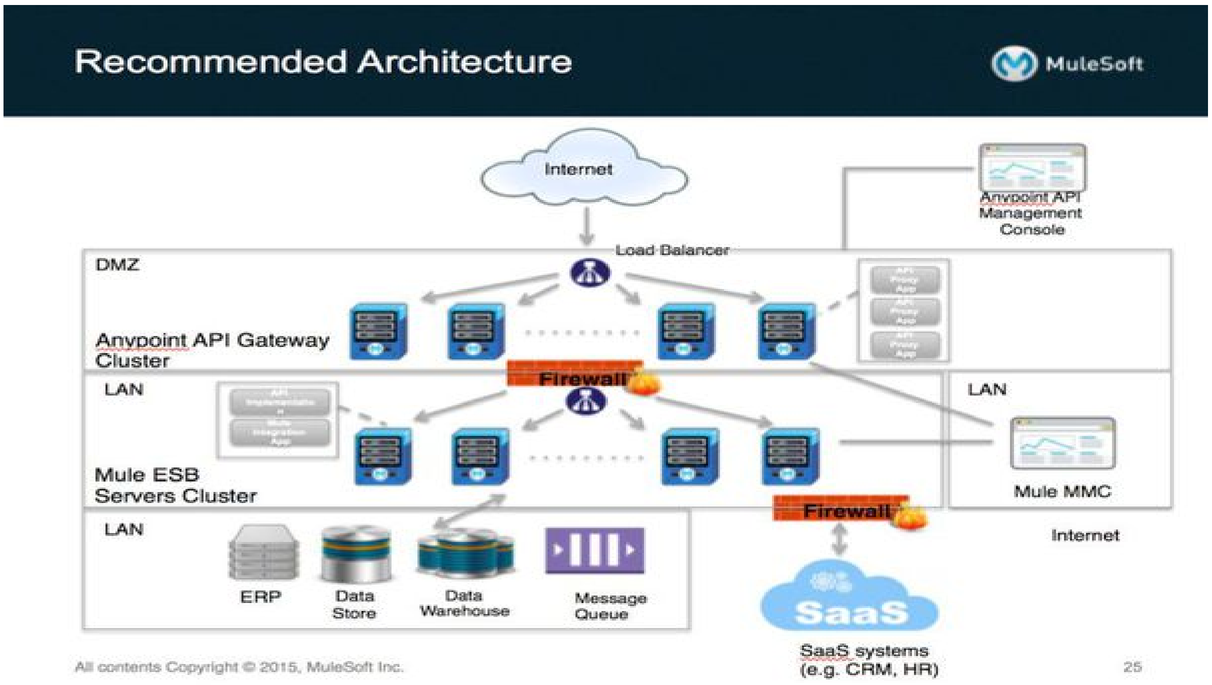


Figure 1: High Level Architecture - On-Cloud

On-Premise

Below diagram captures all the required infrastructure required for building an on premise based Mule ESB environment.

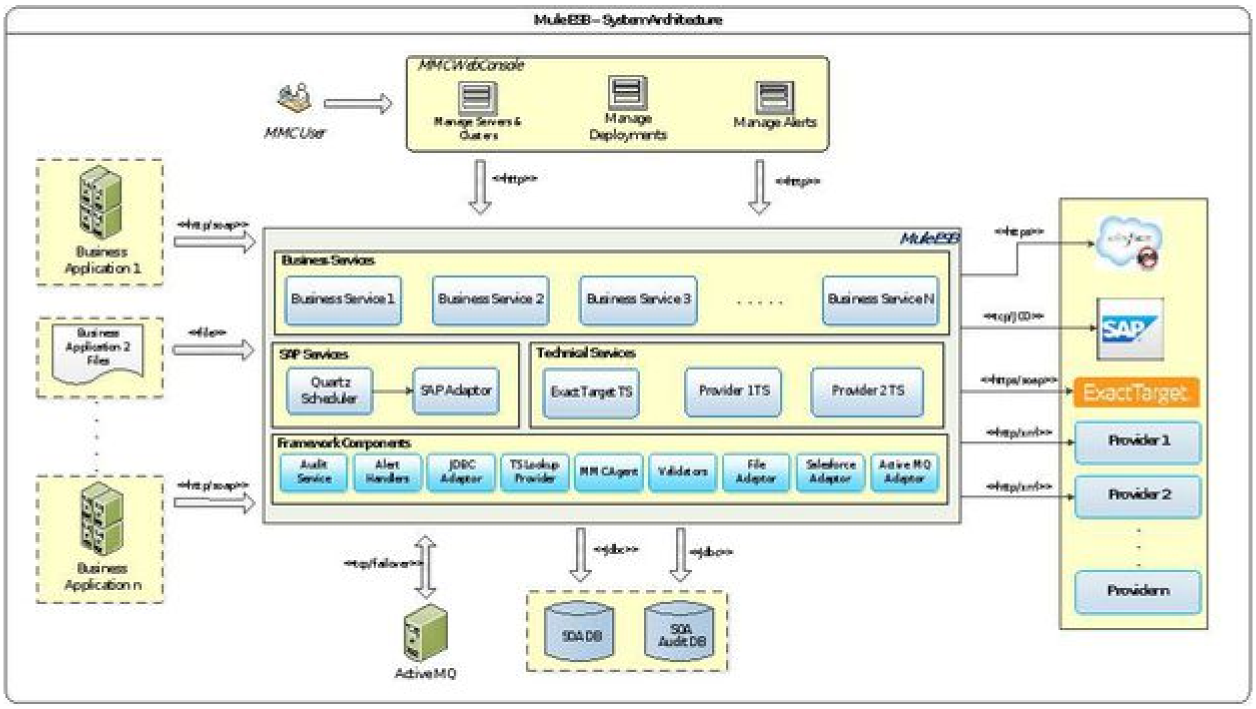


Figure 2: High Level Architecture - On-Premise

Operational View

Application Logging & Exception Handling

Logging

The key best practice around the application logging and exception handling routines are to ensure that the various log entries can be correlated via a correlation ID. This is even more crucial in asynchronous architecture where the end to end flow is being handled by distributed fragments.

The source of the correlations ID is very much tied to the business data associated with a flow. And more importantly, it is crucial that this chosen value is available to all fragments within the end to end flow so that it can be logged. The next question is how should these entries be logged and how can the visualization of these entries be centralized. The table below offers three options that can be implemented on the Mule Platform
Table 1: Options for Common Logging and Exception Handling Routines

Option	Description
Use of MULE Business Events	The MULE Business Events are a component of the Insight dashboard in CloudHub. The Business Events component can be configured at each flow level or connector level. The configuration is simply a checkbox to enable or disable event logging and an MEL to determine the value to be used for the transaction ID as oppose to allowing MULE to generate its own. All events are stored in a database that has been configured against the MMC. 'Custom Business Events' can also be scattered across the flow to persist additional event details. The advantages and disadvantages of this approach are highlighted below **Advantages:** •Easily Configurable •Allow for correlation via the MEL against the transaction ID •Part of the Cloudhub so a single dashboard •Centralized MULE Logging **Disadvantages:** •Specific to MULE so cannot be an enterprise solution •No flexibility over the structure of the event data stored •Cannot modify choice of persistence
Log4j Extensions	This is a standard approach that many organizations are quite familiar with and as a result it is the easiest one to implement at this stage. The advantages and disadvantages of this approach are highlighted below: **Advantages:** •Complete control over log formats and log points •Complete control over the choice of the appender •Open source log4j extension **Disadvantages:** •No centralized approach for file appenders •Difficulty in correlation of log files are scattered
Common Logging Service	This is the optimal approach as it provides an enterprise solution. At a high level this solution provides a service that exposes multiple log channels (SOAP, JMS, REST) ties to a logging schema model that can be used not only by MULE components but all components within the enterprise. The solution offers log persistence and a visualization layer on top of the persistent layer.

It is a best practice to ensure that exception handling leverages the common logging routines. It is always recommended that exception handling routine extend the logging routine and to then provide the required additional logic to handle the exception itself.
Key points to be noted while designing & developing logging component are listed below

- Log class name, method name and appropriate information (like input parameters, return values) at begin of a method and before exit of the method. This helps in analysing logs in case of any issues.
- Set the appropriate logging level when logging messages.
- While logging in catch blocks of exceptions, make sure that context information is logged along with exception.
- When communicating with external systems, log all the information that is sent to external system and returned from it. For example, logging full messages contents, including SOAP and HTTP headers in web services is extremely useful during integration and system testing.
- Logging should be easy to read and easy to parse.

Exception Handling

The key point in relation to exception strategies is that these strategies can be applied at the flow level but not at the sub-flow level. This feature should be noted in the design of the overall implementation and in the definition of locations where the exceptions are to be handled. The figure below highlights exception handling within the Mule Context. The key point to take out of this figure is that certain exceptions occur outside of the message flow and as a result will NOT be caught by the flow exception strategies. Exception handling is usually handled within a common routine. Flow exceptions can adopt this common routine and now the question is how to ensure that exceptions NOT caught by the flow exception strategy can also adopt this same common routine. Section Default System Exceptions highlights the best practice in ensuring the capture of system exceptions.

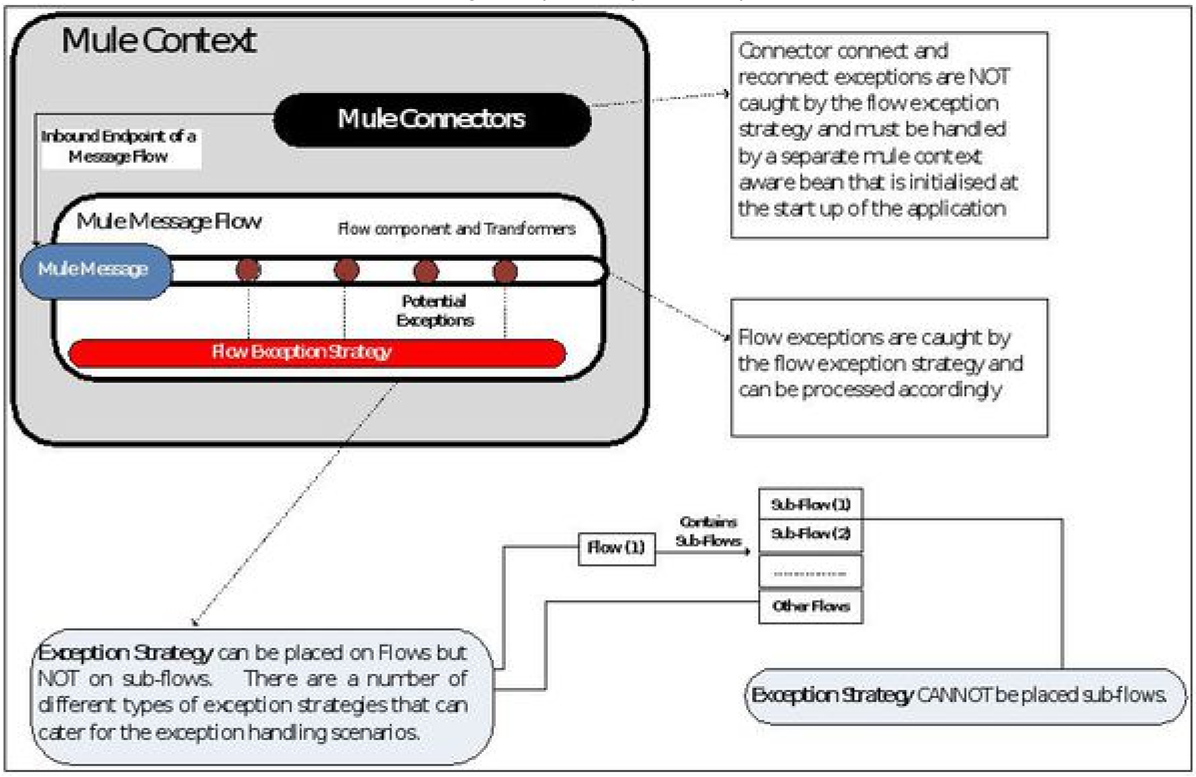

Mule Context
Mule Connectors
Inbound Endpoint of a Message Flow
Mule Message Flow
Flow component and Transformers
Mule Message
Potential Exceptions
Flow Exception Strategy
Connector connect and reconnect exceptions are NOT caught by the flow exception strategy and must be handled by a separate mule context aware bean that is initialised at the start up of the application
Flow exceptions are caught by the flow exception strategy and can be processed accordingly
Flow (1)
Contains Sub-Flows
Sub-Flow (1)
Sub-Flow (2)
....
Other Flows
Exception Strategy can be placed on Flows but NOT on sub-flows. There are a number of different types of exception strategies that can cater for the exception handling scenarios.
Exception Strategy CANNOT be placed sub-flows.

Figure 3: Exception Handling Strategies

Some key coding standards to be followed while implementing Exception handling in the flows are listed below

1. Never swallow the exception in catch block

```
catch (NoSuchMethodException e) {

        return null;

}
```

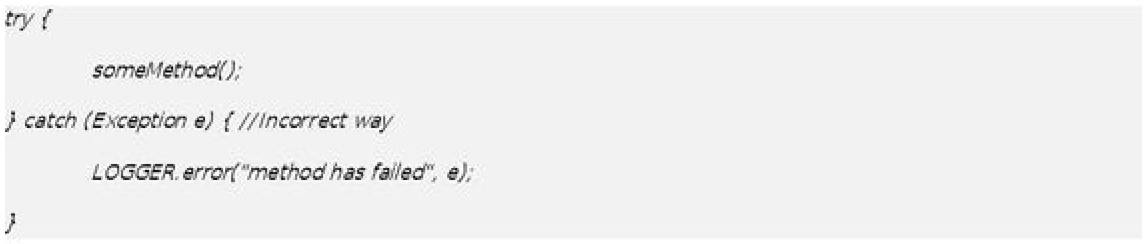
Doing this not only return "null" instead of handling or re-throwing the exception, it totally swallows the exception, losing the cause of error forever.

1. Declare the specific checked exceptions that your method can throw

```
public void foo() throws Exception { //Incorrect way

}

public void foo() throws SpecificExp1, SpecificExp2{ //Correct way

 }
```

Always avoid doing this as in above code sample. It simply defeats the whole purpose of having checked exception. Declare the specific checked exceptions that your method can throw. If there are just too many such checked exceptions, you should probably wrap them in your own exception and add information to in exception message. You can also consider code refactoring also if possible.

1. Do not catch the Exception class rather catch specific sub classes



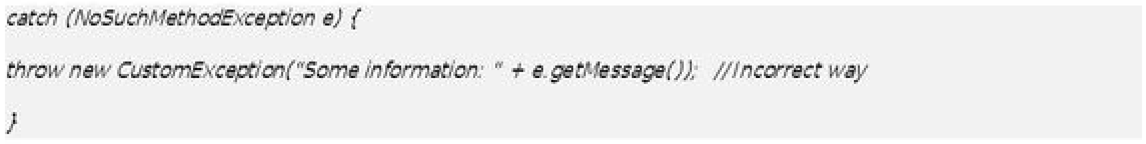
```
try {

        someMethod();

} catch (Exception e) { //Incorrect way

        LOGGER.error("method has failed", e);

}
```

The problem with catching Exception is that if the method you are calling later adds a new checked exception to its method signature, the developer's intent is that you should handle the specific new exception. If your code just catches Exception (or Throwable), you'll never know about the change and the fact that your code is now wrong and might break at any point of time in runtime.

1. Never catch Throwable class

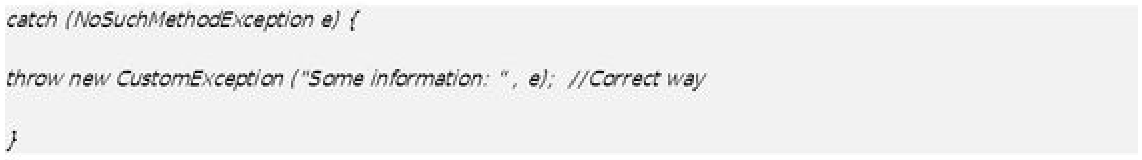
Never catch Throwable class because java errors are also subclasses of the Throwable. Errors are irreversible conditions that cannot be handled by JVM itself and for some JVM implementations, JVM might not actually even invoke your catch clause on an Error.

1. Always correctly wrap the exceptions in custom exceptions so that stack trace is not lost



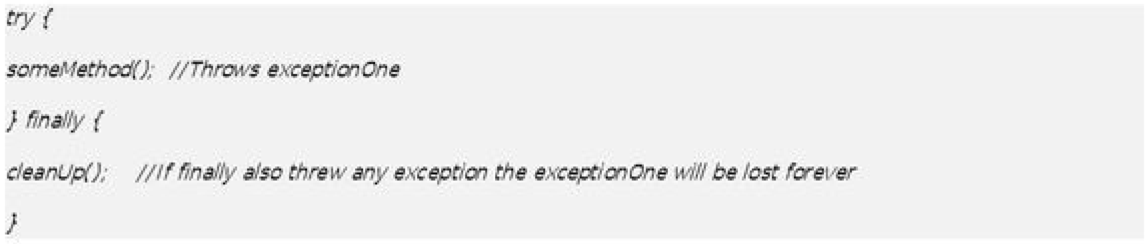
```
catch (NoSuchMethodException e) {

throw new CustomException("Some information: " + e.getMessage());  //Incorrect way

}
```

When we do this, we lose the information of the stack trace from the original exception. The right way of wrapping exception is:



```
catch (NoSuchMethodException e) {

throw new CustomException ("Some information: " , e);  //Correct way

}
```

1. Never throw any exception from finally block



```
try {

someMethod();  //Throws exceptionOne

} finally {

cleanUp();    //If finally also threw any exception the exceptionOne will be lost forever

}
```

This is fine, as long as cleanUp() can never throw any exception. In the above example, if someMethod() throws an exception, and in the finally block also, cleanUp() throws an exception, that second exception will come out of method and the original first exception (correct reason) will be lost forever. If the code that you call in a finally block can possibly throw an exception, make sure that you either handle it, or log it. Never let it come out of the finally block.

1. Always catch only those exceptions that you can actually handle

```
catch (NoSuchMethodException e) {

throw e; //Avoid this as it doesn't help anything

}
```

Do not catch any exception just for the sake of catching it. Catch any exception only if you want to handle it or, you want to provide additional contextual information in that exception. If you can't handle it in catch block, then best advice is just don't catch it only to re-throw it.

1. Don't use printStackTrace() statement or similar methods

Never leave printStackTrace() after finishing your code. Chances are one of your fellow colleague will get one of those stack traces eventually, and have exactly zero knowledge as to what to do with it because it will not have any contextual information appended to it.

1. Use finally blocks instead of catch blocks if you are not going to handle exception

```
try {

someMethod();  //Method 2

} finally {

cleanUp();    //do cleanup here

}
```

This is also a good practice. If inside your method you are accessing some method 2, and method 2 throw some exception which you do not want to handle in method 1, but still want some cleanup in case exception occur, then do this cleanup in finally block. Do not use catch block.

1. Remember "Throw early catch late" principle

This is probably the most famous principle about Exception handling. It basically says that you should throw an exception as soon as you can, and catch it late as much as possible. You should wait until you have all the information to handle it properly.
This principle implicitly says that you will be more likely to throw it in the low-level methods, where you will be checking if single values are null or not appropriate. And you will be making the exception climb the stack trace for quite several levels until you reach a sufficient level of abstraction to be able to handle the problem.

1. Always clean up after handling the exception

If you are using resources like database connections or network connections, make sure you clean them up. If the API you are invoking uses only unchecked exceptions, you should still clean up resources after use, with try – finally blocks. Inside try block access the resource and inside finally close the resource. Even if any exception occur in accessing the resource, then also resource will be closed gracefully.

1. Throw only relevant exception from a method

Relevancy is important to keep application clean. A method which tries to read a file; if *throws NullPointerException* then it will not give any relevant information to user. Instead it will be better if such exception is wrapped inside custom exception e.g. *NoSuchFileFoundException* th en it will be more useful for users of that method.

1. Never use exceptions for flow control in your program

We have read it many times but sometimes we keep seeing code in our project where developer tries to use exceptions for application logic. Never do that. It makes code hard to read, understand and ugly.

1. Validate user input to catch adverse conditions very early in request processing

Always validate user input in very early stage, even before it reached to actual controller. It will help you to minimize the exception handling code in your core application logic. It also helps you in making application consistent if there is some error in user input.
For example: If in user registration application, you are following below logic:

1. Validate User
2. Insert User
3. Validate address
4. Insert address
5. If problem the Rollback everything

This is very incorrect approach. It can leave you database in inconsistent state in various scenarios. Rather validate everything in first place and then take the user data in Dao layer and make DB updates. Correct approach is:

1. Validate User
2. Validate address
3. Insert User
4. Insert address
5. If problem the Rollback everything

1. Always include all information about an exception in single log message

```
LOGGER.debug("Using cache sector A");

LOGGER.debug("Using retry sector B"); //Incorrect way
```

Using a multi-line log message with multiple calls to LOGGER.debug() may look fine in your test case, but when it shows up in the log file of an app server with 400 threads running in parallel, all dumping information to the same log file, your two log messages may end up spaced out 1000 lines apart in the log file, even though they occur on subsequent lines in your code.

```
LOGGER.debug("Using cache sector A, using retry sector B"); //Correct way
```

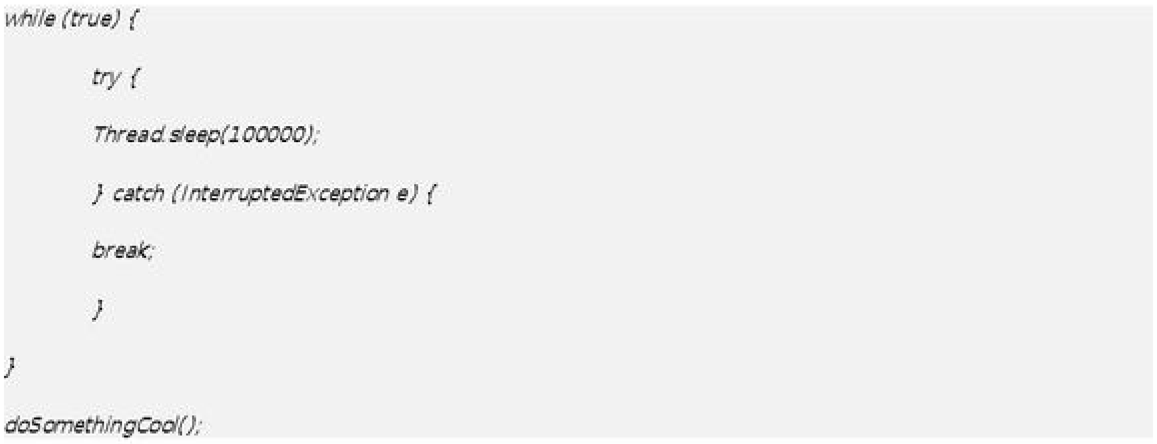
1. Pass all relevant information to exceptions to make them informative as much as possible

This is also very important to make exception messages and stack traces useful and informative. What is the use of a log, if you are not able to determine anything out of it? These type of logs just exist in your code for decoration purpose.

1. Always terminate the thread which it is interrupted

```
while (true) {

        try {

        Thread.sleep(100000);

        } catch (InterruptedException e) {} //Don't do this

        doSomethingCool();

        }

}
```

InterruptedException is a clue to your code that it should stop whatever its doing. Some common use cases for a thread getting interrupted are the active transaction timing out, or a thread pool getting shut down. Instead of ignoring the InterruptedException, your code should do its best to finish up what it's doing, and finish the current thread of execution. So to correct the example above:



```
while (true) {

          try {

          Thread.sleep(100000);

          } catch (InterruptedException e) {

          break;

          }

}

doSomethingCool();
```

1. Document all exceptions in your application in javadoc

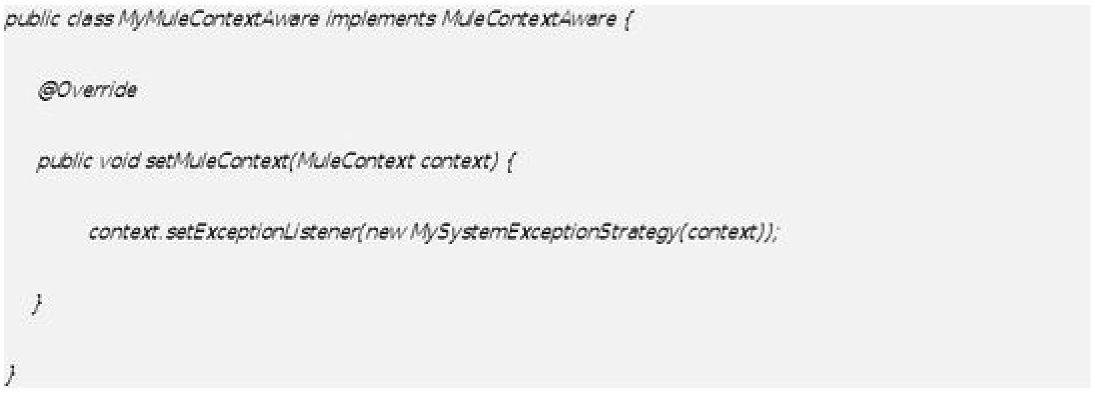
Make it a practice to javadoc all exceptions which a piece of code may throw at runtime. Also try to include possible course of action, user should follow in case these exception occur.

Default System Exceptions

The requirements to capture system exceptions are defined below:

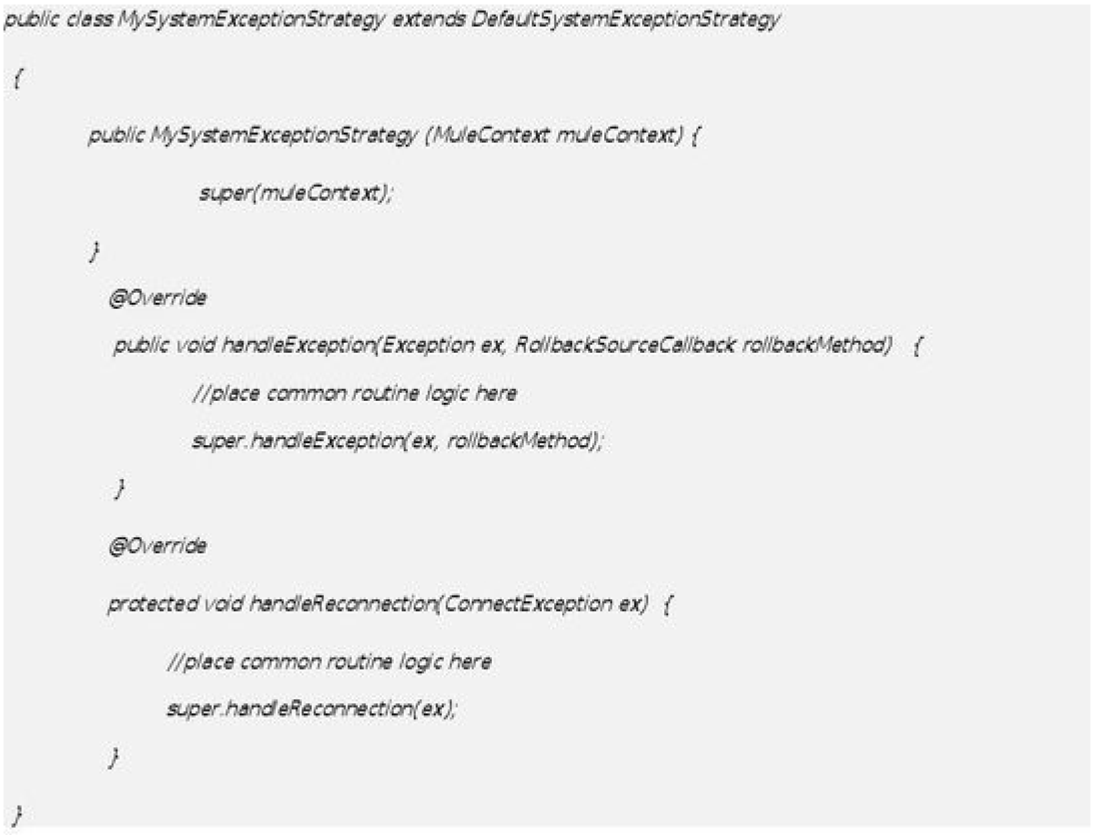
1. The following JAVA classes must be created:
2. A JAVA class that implements the Mule Context and sets an Exception Listener
3. The exception listener in (a) is the next JAVA class that is created.

Table 2: Java Class for Handling Default Exceptions – Mule Context



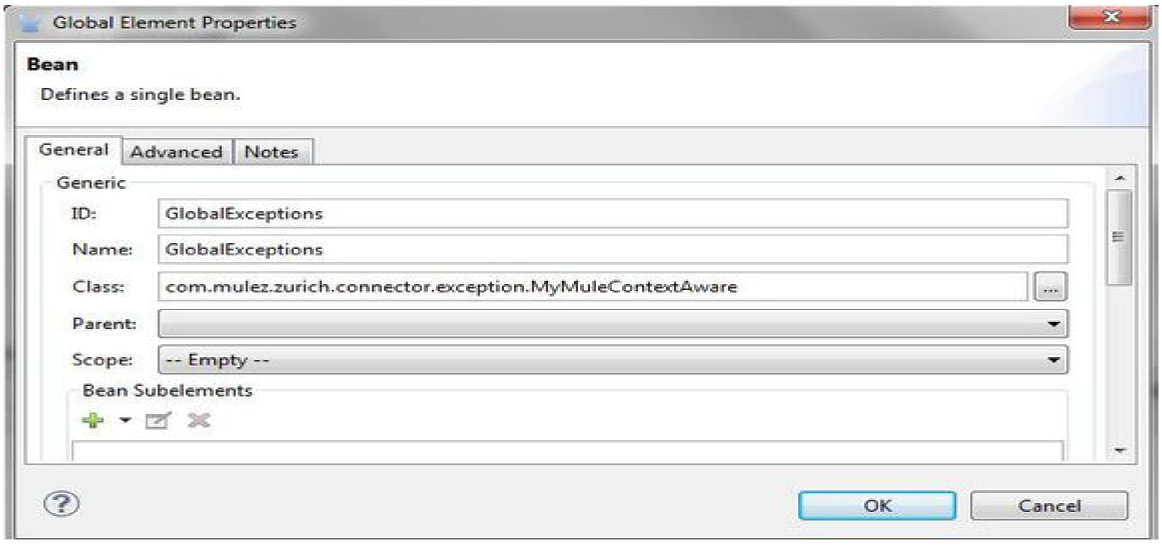
```
public class MyMuleContextAware implements MuleContextAware {

    @Override

    public void setMuleContext(MuleContext context) {

          context.setExceptionListener(new MySystemExceptionStrategy(context));

    }

}
```

Table 3: Java Class for Handling Default Exceptions – Exception Strategy



```
public class MySystemExceptionStrategy extends DefaultSystemExceptionStrategy

{

        public MySystemExceptionStrategy (MuleContext muleContext) {

                    super(muleContext);

        }
          @Override
          public void handleException(Exception ex, RollbackSourceCallback rollbackMethod)   {
                    //place common routine logic here
                    super.handleException(ex, rollbackMethod);
          }
          @Override

          protected void handleReconnection(ConnectException ex)  {

                  //place common routine logic here
                  super.handleReconnection(ex);
          }

}
```

1. Create a bean as part of the global elements that references the implementation of the Mule Context as per the snapshot below:



Global Element Properties

Bean
Defines a single bean.

General | Advanced | Notes

Generic

ID:	GlobalExceptions
Name:	GlobalExceptions
Class:	com.mulez.zurich.connector.exception.MyMuleContextAware
Parent:	
Scope:	-- Empty --

Bean Subelements

Flow Exceptions

As mentioned before, exception strategies can be applied to flows only. The Exception strategy can be either done within a flow to handle certain flow specific exception or can use a generic global exception strategy. In the event that all flows adopt the same exception strategy it is a best practice to define the exception flow at the global level and ensure that all other flows references the global strategy via the use of a reference exception strategy.
The use of a choice exception strategy as highlighted in the figure below is recommended where the different exception types need to be handled differently. The figure below is an example of a choice strategy faults are handled differently for each case. As part of the choice strategy other exception types can be added to this strategy as well.

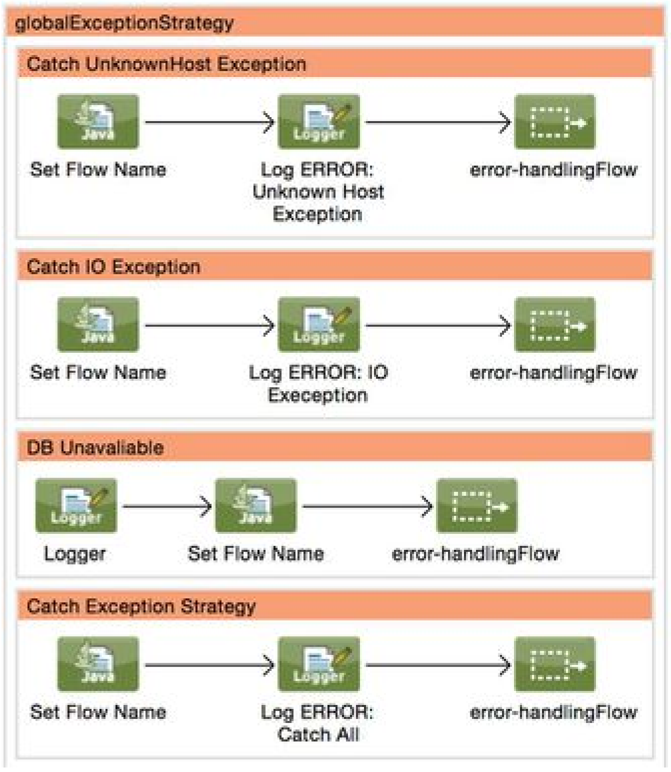


Figure 4: Flow-Choice Exception Strategies

Please refer to the sample exception strategy provided with the Mule documentation.

Custom Exception

Custom exceptions can be thrown in a flow using groovy expressions. For example :

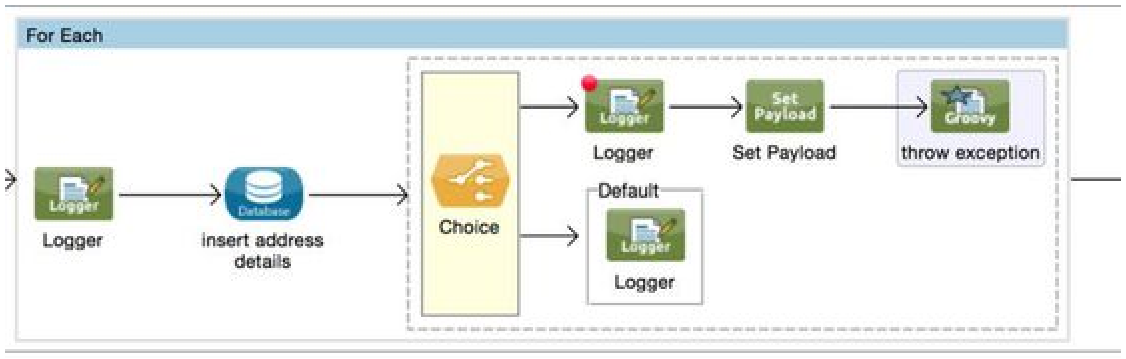


Figure 5: Custom Exception Flow

Set the payload to whatever you want the consumer to receive as exception message. The groovy expression then should look something like this –

```
throw new java.lang.RuntimeException(payload);
```

Excepti
on Notifications
Notification listeners can be registered to listen for exceptions if there is a requirement to do some custom action when an exception occurs. To enable exception message notifications, you set the event you are interested to listen and reference to a notification listener in your flow configuration file.

```
<spring:beans>
<spring:bean name="myExceptionListener" class="com.file.sample.MyExceptionListener"/>


</spring:beans>
<notifications>
<notification event="EXCEPTION"/>
<notification event="EXCEPTION-STRATEGY"/>
<notification-listener ref="myExceptionListener"/>
</notifications>
```

Table 4: Flow Configuration

```
public class MyExceptionListener implements ExceptionNotificationListener{


static Logger rootLogger = null;
static {
rootLogger = Logger.getRootLogger();
}


public void onNotification(ServerNotification notification) {
// TODO Auto-generated method stub
if ( notification.getAction()== ExceptionNotification.EXCEPTION_ACTION){
rootLogger.info("Inside my custom notification listener");
}
System.out.println("Inside Server notification");
//notification.

}

}
```

Table 5: Exception Listener Sample Code

High Availability

HA in Mule instance can be achieved either by using HA clustering or CloudFabric. HA clustering is only for On-Premise Mule deployments. CloudFabric allows reliable message delivery across availability zones in the same region. In order to achieve HA, you can deploy additional workers for the same application. This will allow for horizontal scaling and dynamic load balancing to optimize performance. However the following points are important to consider

- Watchout for your vcore usage as every worker will be allocated a separate vcore. Ensure that you don't exceed your license model with the additional workers.
- It is important for the downstream applications to also be resilient and highly available using the replication features. For example your target database or file system should be replicated across data centres without failures.
- Static IP Configuration for worker will not work for this model. If you have a requirement for Mule to call an external WebService, it is a normal practice to use static IP address for whitelisting the source IP.

Scalability

Mule Applications can be scaled both horizontally and vertically.

Horizontal Scaling

This can be achieved by adding additional workers/instances for your deployed application.

Vertical Scaling

This can be done by adding more vCores/memory to the Mule server instance.

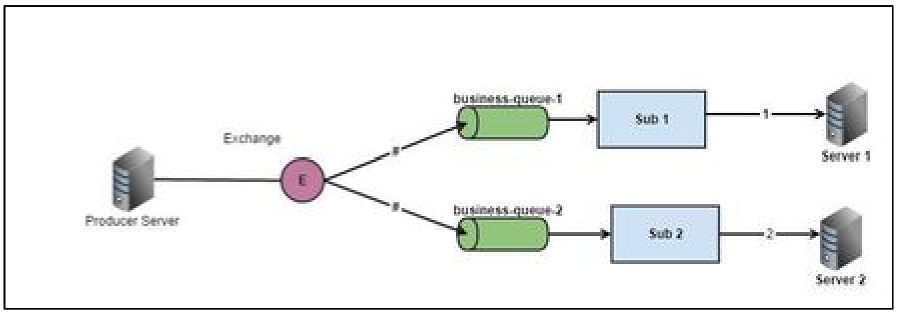
Reusability & Extensibility

Reusable applications can be developed and deployed in your CloudHub instance. These reusable applications can either expose HTTP or JMS endpoints. For HTTP endpoint, although the application to application communication will go via public DNS, the traffic remains within AWS. Slight latency penalty outweighs the benefits of creating granular services via correct DNS names. MuleSoft doesn't provide a messaging server, however it supports various JMS providers through the use generic JMSConnector.

Messaging

Mule supports quick and easy integration with many popular messaging services over JMS, AMQP, MSMQ and WebSphere MQ Series. One of the critical aspects to success in SOA is to integrate mule services with a messaging server, there by defining an effective and scalable Message Oriented Middleware (MOM). Such an architecture helps decoupling systems, build reliability and scale for performance.

By extending the messaging server with the integration architecture, a new service design can be achieved. As demonstrated in the diagram below, the producer server pushes the message and returns. The consuming servers will process the message defined by the routing logic and message headers. The consuming servers are free to pick up the message at will (as per configuration) there by relieving the producer from memory, processing and persistence requirements.



Traffic from the server can be off loaded to the message brokers – these messages can later be processed as per the business rules. This again saves on server memory and increases throughput.

Apart from enabling de-coupled systems, a re-usable fault handling mechanism can also be devised through a MOM server

Security

There are various levels of security that can be incorporated within the Mule Platform and these are very dependent on the organizational requirements. These levels include:

1. Transport Level Security: The incorporation of SSL in its different formats (e.g. one-way or Mutual SSL) to secure the channel between the client and the server.
2. Protocol Level Security: These include the introduction security components such as WS-Security for Web Services, OAUTH for Restful Services, Basic/Digest Authentication and other mechanisms that are protocol related. Currently Mule API platform supports out of the box support for OpenAM and Ping as the external identify provider. Support for SAML2 is expected to be released in Q4 2015.
3. Data Level Security: This is the encryption/signing of data in flight or in rest that would very likely incorporate custom algorithms or standard algorithms with the requirement of external resources such as certificates and trust stores.

When the runtime is hosted in the CloudHub, make sure that the region is changed to your default target region for your organization Change the region to Sydney in your CloudHub Admin console. This will allow for all availability zones to belong to the same region for your HA configuration. If there is a requirement for regional data constraints where data is not allowed to go across regions, make sure you deploy a specific instance of you application to the target region in CloudHub.
Any connection from CloudHub to internal network should use VPC peering, VPN or IPSec. Please refer the CloudHub guide for configuration - https://developer.mulesoft.com/docs/display/current/Virtual+Private+Cloud

Monitoring

Monitoring of Mule Servers/Applications can be done using the following options –

Mule Agent

All Mule instances can be monitored using the Mule Agent. Mule Agent is a plugin that exposes the Mule ESB Java API as a service allowing users to manipulate and monitor Mule ESB instances from external systems. The agents can send Notifications about events that occur in Mule instance in JSON format so that the user can implement their own system for listening and handling these notifications. Currently Mule Agent is only available on premise but is expected to be fully supported in CloudHub by Q3 2015. You should use the Mule agent to publish notifications to Splunk, ElasticSearch etc.

Custom JMX Monitoring

Applications can enable JMX monitoring by writing a custom JMX agent. A sample JMX monitoring application has been created which will be uploaded in confluence.

CloudHub

CloudHub Admin console provides monitoring information in the form of dashboard. The dashboard provides a graph representation of CPU, Memory and Mule Message throughput for each application. Apart from this, the following alerts can be configured per application

- Deployment Failure
- Deployment Success
- Exceeds event traffic threshold
- Custom Application Notification
- Secure Data Gateway disconnected
- Secure Data Gateway connected
- Worker not responding

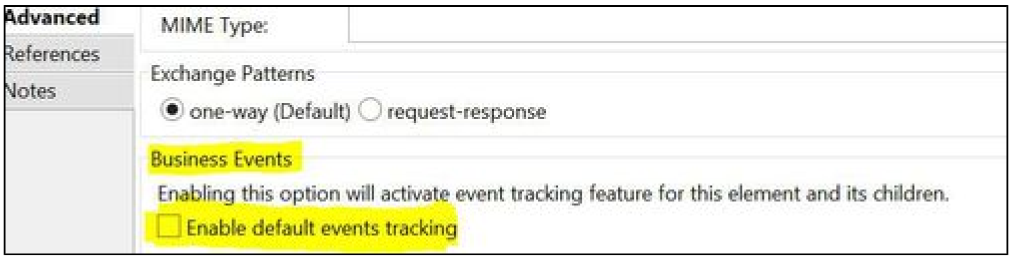
Custom Business Events

Custom Business Event can be designed and implemented to provide insights into runtime transactions. This enables administrators, support personnel and business users to identify points of failures and address them either reactively of proactively. This feature, however, is the least used across many clients.

Apart from Mule’s default server events, the custom events are easy to define, develop and monitor.

```
 <tracking:custom-event event-name="Order Status" doc:name="Order
Status">
    <tracking:meta-data key="Order Id" value="#[payload.ID]"/>
</tracking:custom-event>
```

Some of the components such as File, AMQP, Email etc, provide an option to enable default events tracking. By using this feature, events can be tracked without a line of code.



MIME Type:

Exchange Patterns

[✓ one-way (Default)] request-response

Business Events

Enabling this option will activate event tracking feature for this element and its children.

☐ Enable default events tracking

Caching Considerations

It is discouraged to add caching in an Enterprise Service Bus such as Mule. This is to keep ESB free from any persistent data, which in turn enables scalability of integration layer. ESB should ideally hold only business and proxy services. As a design approach, caching should be owned by the User Experience Layer as this is where most caching requirements arise (due to storing user account or device session or transaction references).

Though Mule provides an In Memory Caching component, it is recommended to first define the amount of data the architect plans to cache. Only store transaction references that would be required during business processing between disparate systems. In cases where higher loads of data is cached ESB should delegate the caching responsibility to a different tier/product (such as an in-memory data grid).

Capacity Planning & Performance

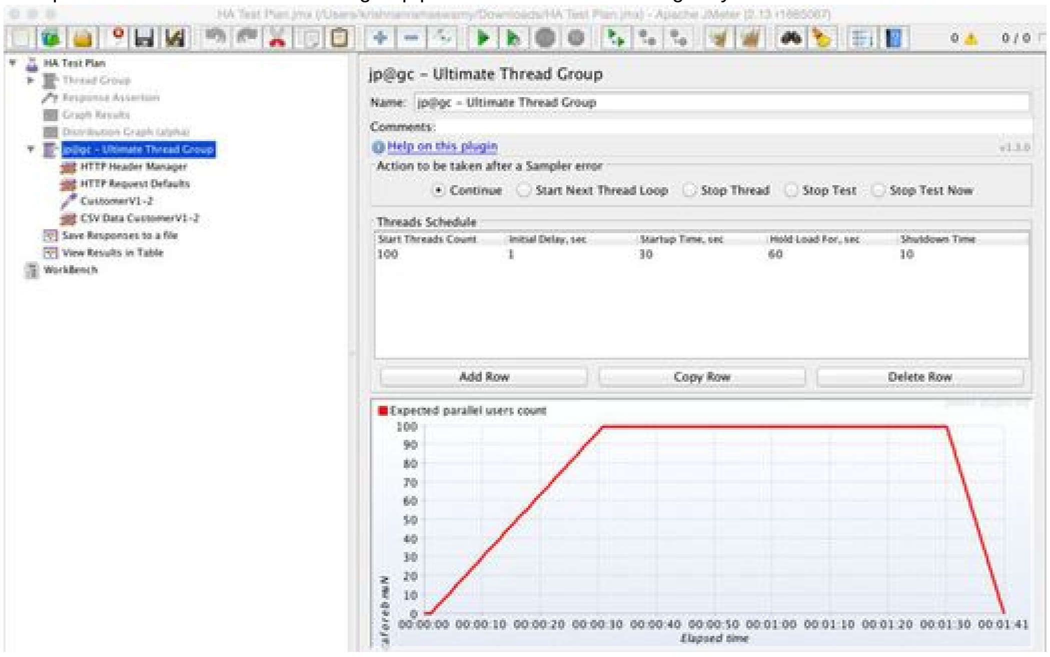
A Mule Application consists of flows which in turn contain one or many endpoints. To estimate the memory requirements for a particular Mule Application, the following steps need to be carried out:

1. Determine all the endpoints (Receivers/Triggers) that are embedded within the application.
2. Estimate the memory required for each of these endpoints. The memory required for each endpoint can be estimated using three parameters:
3. A – The throughput on this endpoint (Number of triggers over time)
4. B – The size of data for each trigger = (Incoming Message Size + Flow Data Size)
5. C – The interval time = The estimated time to execute each request (trigger)

Using the above, the memory for each endpoint = (A x B x C)

1. The total memory required for the application is the sum of the above plus a 256M requirement for the JVM initialization. The JVM initialization should also include any data that is loaded into memory on startup.
2. A Mule ESB consists of one or many Mule Applications and hence to estimate the size of memory required for a particular ESB node, it is merely a matter of aggregating the memory requirements for the various applications that will be part of the ESB node.

A good starting point for any sizing calculation is to begin with a performance testing tool like JMeter or LoadRunner. JMeter is free and very easy to setup. Download the ultimate thread group profile for JMeter and configure your tests as shown in the below diagram.


jp@gc - Ultimate Thread Group

Name: jp@gc - Ultimate Thread Group

Comments:

Action to be taken after a Sampler error: [✓ Continue] Start Next Thread Loop / Stop Thread / Stop Test / Stop Test Now

Threads Schedule

Start Threads Count	Initial Delay, sec	Startup Time, sec	Hold Load For, sec	Shutdown Time
100	1	30	60	10

Figure 6: JMeter Initial Setup

It is a good practice to run the scripts from command line. The syntax for running the scripts command line is as shown below

```
./jmeter -n -t /Users/krishnanramaswamy/Downloads/example.proxy.jmx -l test.jtl
```

where
-n = non-gui
-t = the test file. This will be the jmx file that you will create that will have configurations for your test data.
-l = log file.
Capture your test results using the tables below.

1. Worker Configuration

Server Name	Number of Cores	Memory

2. Performance Testing – XYZ Module

The following observation were made for this module

URI	ThreadCount	Initial Delay (In Seconds)	StartupTime (Seconds)	Hold Load for (In Seconds)	ShutdownTime (seconds)	Median Response Time (ms)	Throughput (msgs/sec)

Your load profile for CPU, Memory should be similar to the graph shown in the JMeter screen shot. If the CPU is well below 50% of the utilized core, then we should be able to deploy multiple applications with fewer cores. In CloudHub with licensing constraints it is very important that we divide the applications based on their usage and allocate cores accordingly.

Service SLA

Service Level SLAs can be defined for your API using the AnyPoint API manager in CloudHub. The platform offers rate-limiting or throttling policy. These policies require all applications that consume your API to register for access to a specific tier, then pass their client credentials in calls to your API, so that the AnyPoint Platform can identify them, associate them with their contracted tier, and enforce the throughput limitation.
Refer https://developer.mulesoft.com/docs/display/current/Defining+SLA+Tiers

Performance Tuning

In CloudHub, JVM tuning is not possible. It is important to remember that if performance is one of the key measurements for your application, then the following points are important –

- Use of in-memory data storage wherever possible to avoid unnecessary database lookups
- Use of Cache with a time-to live property. The time-to-live property should be set according to your data. If the data is volatile and constantly changing then this should be kept to a small time.
- If the application uses JMS with ActiveMQ as the message broker, consider disabling message persistence. Doing so may alleviate IO bound concerns and increase ActiveMQ performance. Similarly messages are consumed by subscribing to a queue or topic. When a message arrives to a broker, it will notify the topics consumer that they have a message to consume. Defining the number of consumers will impact application performance. That is because the application will be able to consume more than one message at time per subscriber, thus processing more messages at the same time (if needed).
- Flow References are a direct way to enable flow communication within an application. Prefer flow references to VM endpoints to communicate between flows. Flow references inject messages into target flow without intermediate steps. Although the VM connector is an in-memory protocol, it emulates transport semantics that serialize and deserialize parts of messages. The phenomenon is especially notable in the Session scope. As such, flow references are superior to VM endpoints for the purpose of inter-flow communication because the former avoid unnecessary overhead generated by serialization and deserialization.

Although log4j2 is asynchronous by nature, high disk IO may impact your performance. If you observe disk IO as your bottleneck, change the log level to WARN.

Deployment

On-Cloud

Deployment can be automated using the CloudHub public APIS. Please refer to this link https://anypoint.mulesoft.com/apiplatform/anypoint-platform/#/portals. Select the CloudHub API from the list and this should provide you all the publicly available API's for Application Management.
CloudHub allows an organization administrator to create multiple environments for your application deployment. This allows for environment segregation. Any new or existing application can be deployed in a new environment with minimal changes. The only change required will be to add in the appropriate user role to the target environment for a particular user.

On-Premise

- Keep all environment dependent properties (e.g.: IP address of SAP system, user name and password for database etc.) in configuration files so that only one build is made for all environments (DEV, INT, UAT , PROD) and it requires only to change configuration files.
- Manage all the deployments from MMC (in case of Mule Enterprise edition) which gives good control on deployments of the applications and monitoring them.
- It is always advisable to have composite inbound end points, one for production and second for debugging, so that in case of production issues, second inbound end point can be used to analyse the flow using flow analyser in MMC.
- As soon as the deployment is done, configure all the alerts.
- Make sure that the size of the log file and maximum number of log files properties to be set properly according to the needs of the application. If the size of the log file is too small and maximum number of files are less, then all the log files will be rotated (overwritten) and there won't be any logs to analyse in case of any issues.

For application specific logs, configure these settings in logger properties. (e.g. log4j.xml)
For mule logs, configure these settings in <MULE_HOME>/conf/wrapper.conf. For example, the following properties set the maximum size of mule_ee.log to 1 MB and maximum number of files to 10.

```
wrapper.logfile.maxsize=1m

wrapper.logfile.maxfiles=10
```

Unit Testing

- Use JUnit framework to unit test Java components.
- Use Mule ESB's Test Compatibility Kit (TCK) of unit tests that can be used to your simple extensions as well as custom modules and transports.
- Use in memory database like HSQLDB/Apache Derby to unit DAO classes.
- Unit test cases code coverage should be greater than 80%.

Functional Testing

As Mule ESB is light-weight and embeddable, it is easy to run a Mule Server inside a test case. Mule provides an abstract JUnittest case called org.mule.tck.junit4.FunctionalTestCase that runs Mule inside a test case and manages the lifecycle of the server. The org.mule.tck.functional package contains a number of supporting classes for functionally testing Mule code.
The flows In Mule application should be designed in such a way that they can be functional tested automatically.
For inbound end points where there is a dependency on external system (for e.g.Http end point), we can consider the option of using composite end point (e.g. Http end point + VM end point) so that we can use other end point (in this example: VM end point) for automated functional test.
For out bound end points, we can mock the out bound end points and inject them during functional testing. This can be achieved in one of the two ways below

- By having sub flows for the out bound points and mock out bound end point, call the sub flow with original out bound end point in real scenarios and call sub flow with mock out bound end point in functional test scenario
- Have a content based router before out bound end point and route the request to original out bound end point in real scenarios and to mock out bound end point in functional test scenarios

Continuous Integration

Continuous integration helps in finding out any issues/bugs and fixing them as we develop the code, instead of waiting until integration testing. This can be achieved by following the below steps.

- Write unit test cases for all the classes.

- Write automated functional test cases.
- Write automated integration test cases wherever possible.
- Set up code quality management tool like Sonar Qube and configure the project in Sonar Qube.
- Create a Jenkins job to compile, build the project and run the analysis on Sonar Cube server.

Debugging

- During development phase of the project, use debugger option in IDE (e.g. in eclipse) instead of printing information to output stream (for e.g. using system. out) for debugging.
- Whenever there is a problem, first analyse the logs for the expected logs for the scenario under test which helps in narrowing down to the issue. Check whether there are any exceptions in the logs.
- Sometimes the thread under execution might get blocked. To check the status of the threads in Java, it is helpful to take thread dump using 'jstack -l <pid>', where <pid> is the process id of java which can be obtained using 'jps' command.
- Try to reproduce the problem in local environment, by writing unit test case/functional test case of the use case. This will not only help in fixing the issue quickly, but also adds the test case to the regression test suite.

Best Practices, Conventions and Standards

The Mule development environment is the Mule Studio which is an eclipse based IDE. This studio provides a hybrid visual and programming style for implementation. All the implementation is captured in a mule configuration file (xml file) that embeds the runtime logic. The purpose of this section is to provide best practices on how these XML files should be named and structured within the studio project.

Naming Conventions

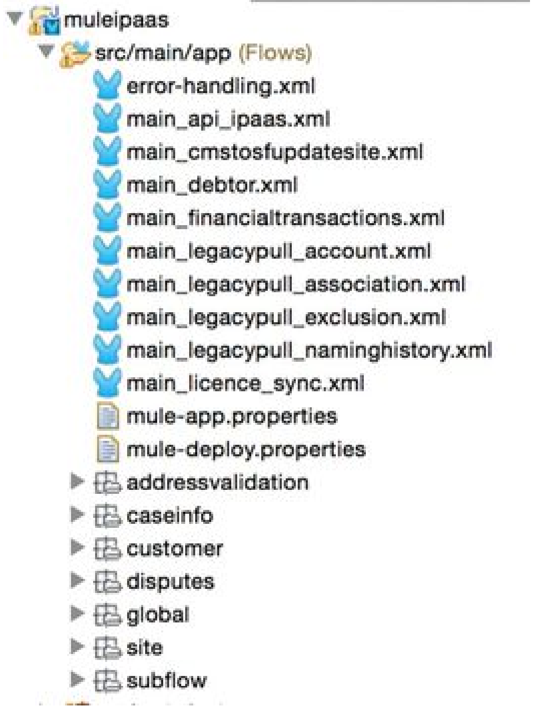
The purpose of the table below is to provide a naming standard and associated description for all flows that may be contained within a project.

XML file Prefix	Flow Descriptions
main_ • main_service • main_queue • main_rest • main_scheduler • main_poller • main_file • main_api	All entry level flows should be prefixed with ***main{_}***. Entry level flows are all flows that expose one or multiple endpoints. Examples of these endpoints include web services, queue receivers, restful services and etc. It is a good practice to dedicate each flow to a particular endpoint type. For example if a project exposes both queue receivers and web services then place the queue receivers in their own dedicated XML and the web services in their own XML as well. The text that follows the prefix is a short description of the endpoint type and some examples have been provided on the left hand column and include: • main_service: Web Service entry points • main_queue: Queue Receivers • main_rest: Restful Services • main_scheduler: Scheduled flows • main_poller: Polled Data Sources • main_file: File Pollers • main_api: APIKit exposed services These main level flows can contain sub-flows but it must be ensured that sub-flows associated with these main flows are in fact only dedicated to these flows. If they are sub-flows that can be used by other flows then it would be a better idea to place them in the shared_ section. Sub-flows ensure that the provided visual implementation is more readable and modularized to facilitate reuse and maintenance. The remainder of the flow name should be based on the business context that the flow is addressing.
subflow_ • sf_ops_ • sf_api_ • sf_q_ • sf_f_	All sub-flows should be prefixed with ***sf{_}***. Each sub-flow provides a specific type of processing and as such the short description that follows the initial prefix must address this type of processing. For example 'ops' to highlight the implementation of a web service operation, 'q' to address the processing of a JMS queue and others. As per the flow names, the text provided after these prefixes must be related to the business context that the sub-flow is addressing.

shared_ • globals_ • flows_ • sf_	All components that are common across the project should be placed in an XML file that is prefixed by shared_. In the event of XML files dedicated to shared resources such as Beans, Connectors, other, it is perfectly OK for the XML to NOT contain any message flows. The short description following the initial prefix is related to the type of common resources e.g. • globals_: connectors, beans, global strategies • flows_: common flows across project • sf_: common sub-flows across the project The remainder of the file name is an optional text that should once again be related to the business context.

Project Layout

As explained in the previous section, all Mule Configuration Files are encapsulated in a Studio project. Each project is translated into a Mule Application at runtime. It is thus important to ensure that if logic is to be separated at runtime to allow for downtime, maintenance and isolation amongst other points, it is done so at the project level.



```
muleipaas
  src/main/app (Flows)
    error-handling.xml
    main_api_ipaas.xml
    main_cmstosfupdatesite.xml
    main_debtor.xml
    main_financialtransactions.xml
    main_legacypull_account.xml
    main_legacypull_association.xml
    main_legacypull_exclusion.xml
    main_legacypull_naminghistory.xml
    main_licence_sync.xml
    mule-app.properties
    mule-deploy.properties
  addressvalidation
  caseinfo
  customer
  disputes
  global
  site
  subflow
```

Figure 7: Project Layout

A typical project layout is as shown in the diagram above.

- ***global folder***: This folder contains the definition of all the components that are common to the project. These include connectors, beans, common sub-flows and all other components that can be used across all other flows and sub-flows within the project
- ***queue.receivers folder***: These are the flows whose endpoint is a JMS queue receiver
- ***services folder***: This is the location of the flows that provide a WebService or a Restful Service. These flows will typically have an HTTP end point.
- ***services.ops folder***: This is the folder that would contain the sub-flows which correspond to the actual implementation of the operations and APIs that have been exposed in the services folder.
- ***java folder***: This is the location of all the java packages and associated classes that were used in the flows. If these classes are global classes then it is best to create a JAR which is imported into the project as opposed to having the same java classes scattered across all projects
- ***resources folder***: This folder should contain all the schema artifacts such as WSDL definitions, XSD and JSON schemas. Additionally it should contain any scripts that were used in the flow such as external groovy scripts.
- ***maven-shared-archive-resources folder***: This folder should contain all schema artifacts that are shared across multiple projects.

Properties Layout

Properties files are a key component of any project. These files should hold all the properties that are environment specific for example:

- All connectivity parameters for inbound and outbound endpoints
- Static values that may be used within the flow to avoid hardcoding
- Configurable processing properties such as time outs, Number of thread, etc.

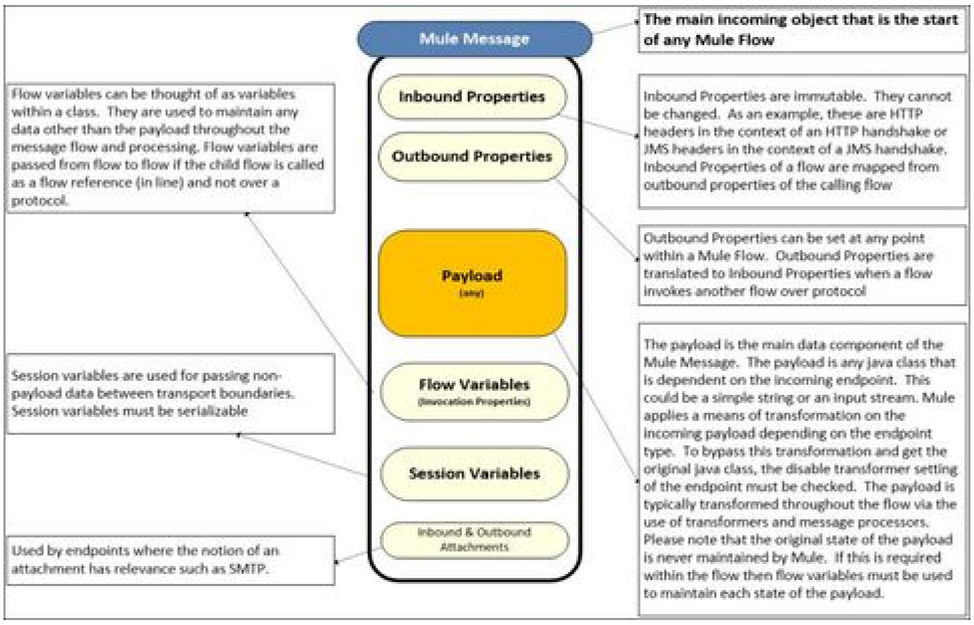
It is typically sufficient to have a single properties file for any project (application). The file is placed in a Property Place Holder which should be defined within the Global section. It is also a best practice to include the environment name in the properties file name and to ensure that the correct file is loaded at runtime based on the environment e.g. DEV, UAT, SIT and PROD. The best practice to ensure that the correct properties file is loaded in relation to the environment is to use a maven profile that uses the correct properties file as per the maven build –P option. For example a Maven profile can be created in the POM using the following code snippet –

```
<profiles>
<profile>
<id>dev</id>
<activation>
<activeByDefault>true</activeByDefault>
</activation>
<build>
<plugins>
<plugin>
<artifactId>maven~~antrun~~plugin</artifactId>
<executions>
<execution>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<copy file="src/test/resources/dev.properties" overwrite="true" tofile="${project.build.outputDirectory}/api.properties"/>

</tasks>
</configuration>
</execution>
</executions>
</plugin>
<plugin>
<groupId>org.mule.munit.tools</groupId>
<artifactId>munit~~maven~~plugin</artifactId>
<version>3.6.0-BETA</version>
<executions>
<execution>
<id>test</id>
<phase>test</phase>
<goals>
<goal>test</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
</profiles>
```

Further instructions on how to do a maven build is described in section Common Resources

Flow Design and Variables

The figure below highlights the various components of a Mule Message. The Mule Message object is available at the start of any Message Flow regardless of what endpoint type has been incorporated in the flow. The figure below also provides a description of the type of data that each portion of the Mule Message contains. The payload class type and the various flow and session variables that are provided in the Mule Message are naturally dependent on the endpoint type that has been incorporated. For example the flow variables associated with HTTP endpoints include all HTTP headers whereas JMS headers constitute the flow variables in the event of JMS endpoints.



The main incoming object that is the start of any Mule Flow

Flow variables can be thought of as variables within a class. They are used to maintain any data other than the payload throughout the message flow and processing. Flow variables are passed from flow to flow if the child flow is called as a flow reference (in line) and not over a protocol.

Inbound Properties are immutable. They cannot be changed. As an example, these are HTTP headers in the context of an HTTP handshake or JMS headers in the context of a JMS handshake. Inbound Properties of a flow are mapped from outbound properties of the calling flow

Outbound Properties can be set at any point within a Mule Flow. Outbound Properties are translated to Inbound Properties when a flow invokes another flow over protocol

The payload is the main data component of the Mule Message. The payload is any java class that is dependent on the incoming endpoint. This could be a simple string or an input stream. Mule applies a means of transformation on the incoming payload depending on the endpoint type. To bypass this transformation and get the original java class, the disable transformer setting of the endpoint must be checked. The payload is typically transformed throughout the flow via the use of transformers and message processors. Please note that the original state of the payload is never maintained by Mule. If this is required within the flow then flow variables must be used to maintain each state of the payload.

Session variables are used for passing non-payload data between transport boundaries. Session variables must be serializable

Used by endpoints where the notion of an attachment has relevance such as SMTP.

Figure 8: Dissection of the Mule Message

- The payload component of the message is transformed from its current form to its new form depending on the transformer being used in the flow. The original payload is no longer available in the flow. Flow variables must be used to capture any state of the payload if it is required at later stages of the flow. The life of flow variables is only within the flow and sub-flows or flows that are called regardless of the scope (async, etc.) that is applied to the child flows.
- Session variables on the other hand are accessible across transport boundaries. Session variables are serialized on flow end and de-serialized on flow entry and as a result incur a performance impact. As a result of this the use of session variables must be reduced if performance is a huge priority within the implementation. It is a better practice to design the payload such that it includes the variables that would have been provided within the session variables.
- Inbound Properties at flow start become Outbound Properties on Outbound Transports.
- Conversely, outbound properties become inbound properties on flow entry
- The above two rules are applicable over transports and where the properties are applicable to the transport.
- In the event of the receiver not wanting certain out of the box properties, it is a good practice to remove these properties within the flow

Testing Strategy

Refer https://developer.mulesoft.com/docs/display/current/testing+strategies

Mule Credential Vault

Use the Mule Credentials Vault to encrypt data in a .properties file. (In the context of this document, we refer to the .properties file simply as the properties file.)
The properties file in Mule stores data as key-value pairs which may contain information such as usernames, first and last names, and credit card numbers. A Mule application may access this data as it processes messages, for example, to acquire login credentials for an external Web service. However, though this sensitive, private data must be stored in a properties file for Mule to access, it must also be protected against unauthorized – and potentially malicious – use by anyone with access to the Mule application. How do you protect the data in a properties file while still making it available to Mule? Use three ingredients:

1. a Mule Credentials Vault
2. a global Secure Property Placeholder element
3. a key to unlock the vault

Refer https://docs.mulesoft.com/mule-user-guide/v/3.6/mule-credentials-vault

Integration Patterns

Synchronous Interactions

These interactions are the simplest type of interactions where the entire request is fulfilled within a single execution thread and the outcome, success or failure is returned to the caller. If there is a requirement to provide replay capacity without involving the original caller, then the best practice is to persist the incoming request via an async scope within the flow. The persistence is done in an async manner to minimize the impact on the response time to the original caller. The actual implementation of the request should be done in a sub-flow so as to allow the processing of the request either via the original caller or the replay logic. A typical example flow with async persistence will look like this -

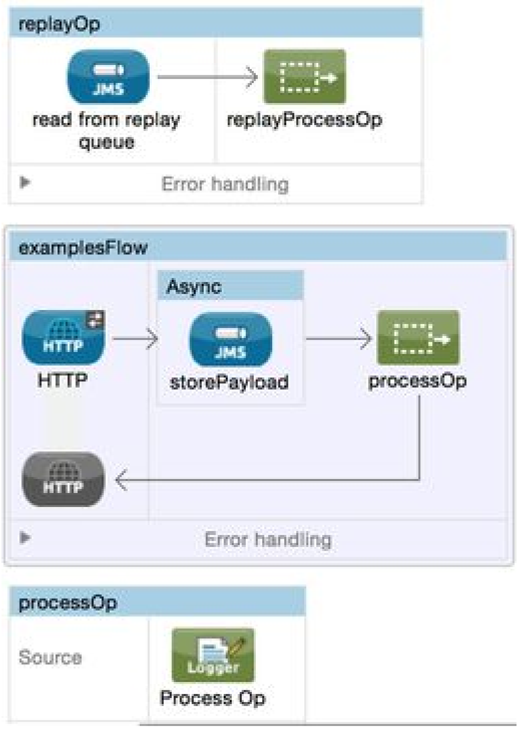


Figure 9: Synchronous pattern with Async Scope

Synchronous Interaction with Asynchronous Processing

It is a best practice to adopt this sort of pattern where the request cannot be fulfilled in a timely fashion (i.e. within a reasonable synchronous timeout) but the caller needs to be made aware of a notification that the request has actually been received. This is somewhat an extension of the above pattern in the sense that the async scope will not only persist the message to allow for replay capacity but also process the request. The response to the caller is correlated to the asynchronous processing. This pattern could be further extended to actually provide a response to the original caller on a separate channel at the end of the asynchronous processing. A good example of this kind of interaction is when doing bi-directional data sync integration with CRM and legacy systems.

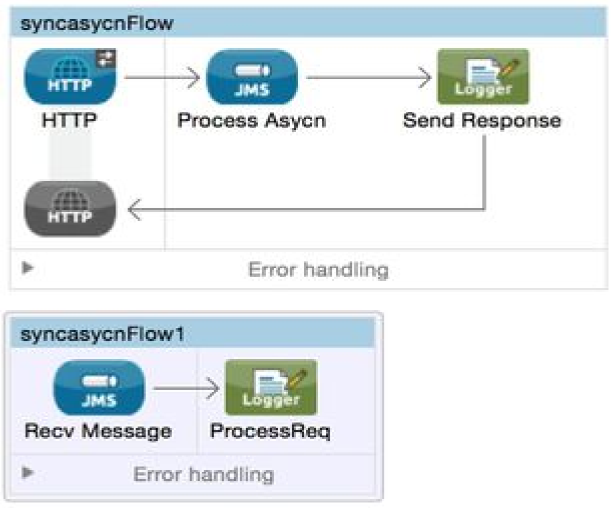


Figure 10: Sync-Asynch Processing

Asynchronous Interaction with No Reliability

This is a 'fire and forget' interaction. In these interactions, a JMS provider is typically involved and the acknowledgment of the message by the receiver is AUTO ACKNOLEDGE. This pattern is utilized where throughputs are high but guaranteed delivery is NOT a requirement and message losses are acceptable.

Asynchronous Interaction with Reliability

This integration pattern is an extension of the above pattern. It is a 'fire and forget' interaction where guaranteed delivery is a requirement. Message losses are not acceptable as part of this requirement. The requirements to achieve this pattern are summarized below:

1. The publication of the original message must be done with the PERSISTENT delivery mode so that when the subscriber is down or there is a restart of the JMS broker, the message is NOT lost.
2. On the subscriber (receiver) side, the acknowledgement mode must be set to CLIENT_ACKNOWLEDGE and the actual confirmation must occur at the end of the flow. Within the Mule Platform there are a number of configuration steps that are required to ensure that this receiver side pattern can be achieved. These are provided in the figure below and are also summarized;
3. The acknowledgement of the JMS connector must be set to CLIENT.
4. The 'Disable Transformer' of the endpoint must be checked to ensure the java.jms.Message object is available to the message flow.
5. This JMS object must be stored within a flow variable at the very start of the flow to allow for the acknowledge () method to be called on this object at any point within the flow.
6. A JMS to object transformer is then called to ensure the content of the JMS message is then placed in the payload.
7. Confirmations should occur at the end of the flow in the sunny day path.
8. In the event of an exception, confirmations should occur in one of two ways:
    a. No confirmation in the event of a technical exception to allow for the JMS message to be redelivered by the JMS Broker
    b. Confirmation in the event of an application (data) exception only after the message is placed on an associated Dead Letter Queue (DLQ)
    c. In the event of a technical exception and exhaustion of the redelivery logic the message should be confirmed only after it has been placed on an associated Dead Letter Queue.fs

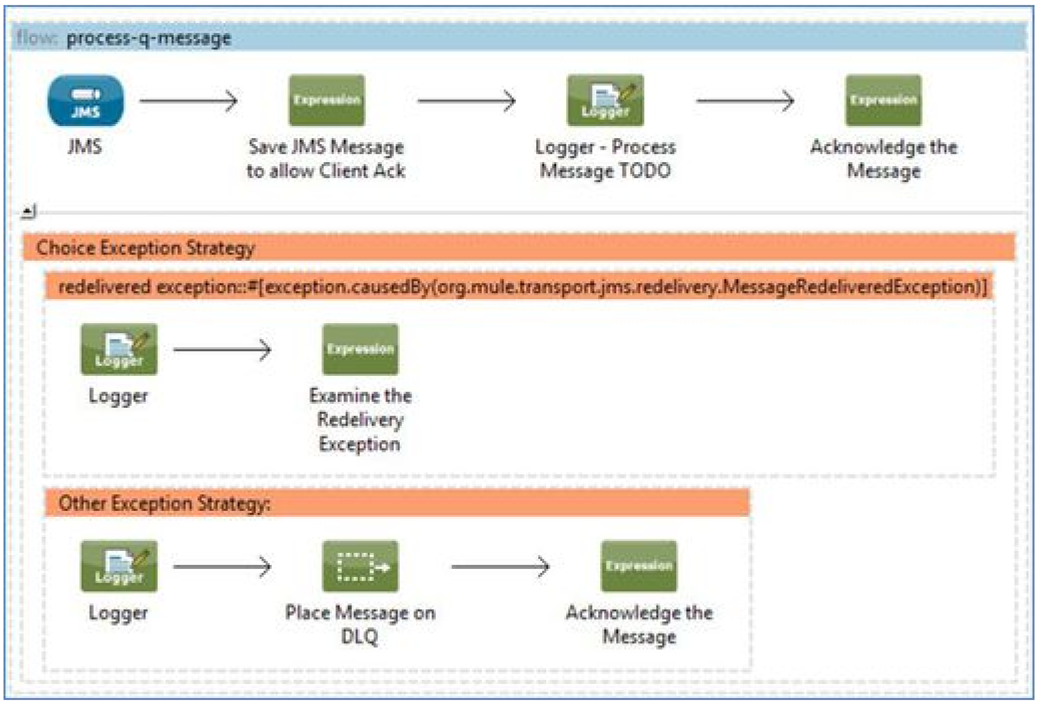

flow: process-q-message
JMS
Save JMS Message to allow Client Ack
Logger - Process Message TODO
Acknowledge the Message
Choice Exception Strategy
redelivered exception::#[exception.causedBy(org.mule.transport.jms.redelivery.MessageRedeliveredException)]
Logger
Examine the Redelivery Exception
Other Exception Strategy:
Logger
Place Message on DLQ
Acknowledge the Message

Figure 11: Asynchronous interaction with Reliability

Batch Data Exchange

This pattern is typically used for ETL kind of jobs where you want to load bulk data from one system to another. A typical example will be a process to load contacts or users into Salesforce or any other CRM system. Mule ESB supports this pattern using the batch module. The batch module is as shown in the below diagram –

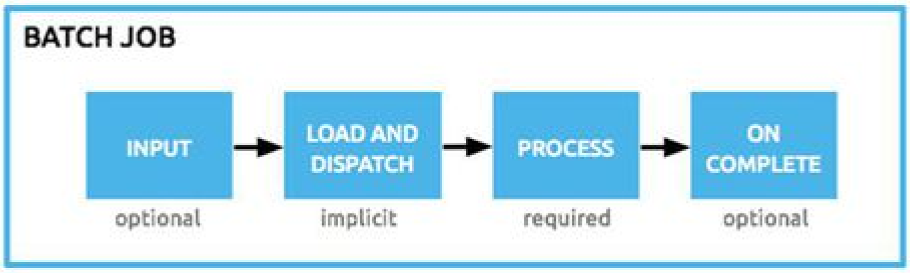


Figure 12: Batch Module

The input phase is an optional phase where you can use a connector to load the data or do some basic transformation to the payload. The load & dispatch phase converts a serialized payload into a collection of records for processing into batch. The third step "Process" can be used to perform various steps for each collection. Each step can be configured to use a filter expression so that it can accept records with an "Accept Expression" expressed in MEL format and an "Accept Policy". The Accept Policy can be ALL, NO_FAILURES or only FAILURES.

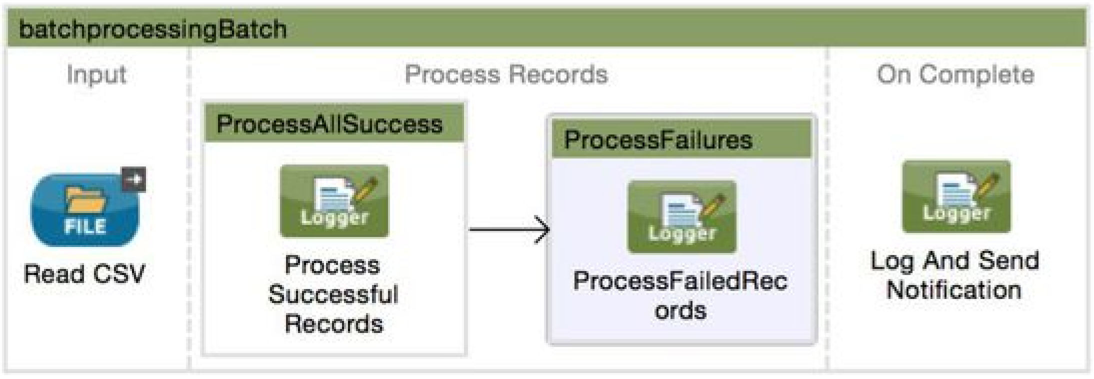


Figure 13: Example Batch Processing

Mediation Pattern

Mediation is an abstraction layer in between the service consumer and the service provider. The mediation layer can have hooks in place to perform authentication, encryption, logging and metrics across all service requests. It decouples the actual implementation of the service so that there is flexibility for the service provider to change their backend logic. It can also be used to translate physical transports to fulfill a service request. For example, service provider may have a capability to expose via JMS and the consumer might need that capability available over HTTP. An example mediation flow is shown below:

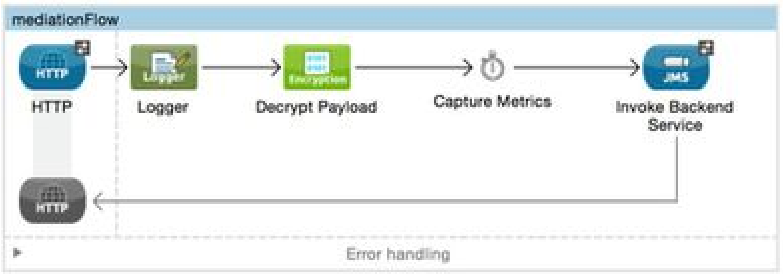


Figure 14: Mediation Pattern

Service Orchestration

Service Orchestration is the integration of several backend services and exposing them as a single service to satisfy a business process use-case. This can also be referred to as a composite service. An example orchestration flow in Mule is shown below:

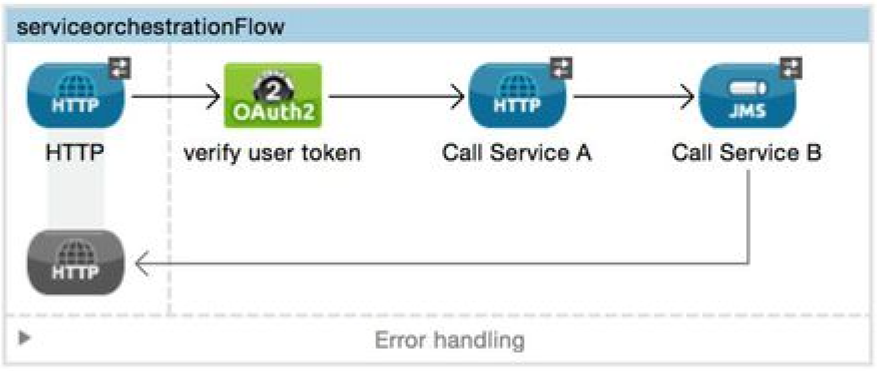


Figure 15: Service Orchestration

API Design Strategy

Despite the number of benefits of REST API's, a common shortcoming for real enterprise adoption is around design. Given there is no contract (such as WSDL for SOAP) it takes a lot of effort for distributed development teams to agree on a particular standard. Documentation is one way to mitigate such problem but history and experience dictates that such approach is not bullet proof. Thus, the REST world, especially inside the enterprise, could really benefit of a design-first approach.

RESTful API Modeling Language (RAML):

"RAML is a simple and succinct way of describing practically-RESTful API's. It encourages reuse, enables discovery and pattern-sharing, and aims for merit-based emergence of best practices. The goal is to help our current API ecosystem by solving immediate problems and then encourage ever-better API patterns. RAML is built on broadly-used standards such as YAML and JSON and is a non-proprietary, vendor-neutral open spec."
RAML website (http://raml.org/)

From design to implementation

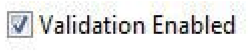
The next step for any API lifecycle is the implementation phase. Given that an API is just an interface, it would not be of much value just having a nice design. Thus, MuleSoft provides unique support for scaffolding the defined resources as Mule flows with the use of APIKit.
"The first of its kind, APIkit is an open-source, declarative toolkit specially created to facilitate REST API implementation. As a simple framework that caters to API-first development, it enforces good API implementation practices. Rather than spending weeks or months building an API, you can use this toolkit to develop, document, and test it within a few days or even hours"
MuleSoft website (http://www.mulesoft.org/)
As stated above, APIKit is an open-source project available through Mule Studio IDE.

Documentation

- http://www.mulesoft.org/documentation/display/current/APIkit

Validation Strategies

The main recommendation around validation is to ensure that validations are carried out as close to the source point as possible. In the context of Mule, this would be at the start of every flow. Where possible the validation component of the CXF component must be enabled as per the snapshot below. This is located under the 'General' section of the CXF component.



☑ Validation Enabled

For endpoints with XML payloads, the use of the 'Schema Validation' is recommended to ensure that the received payload conforms to the required schema. For endpoints that incorporate JSON payloads, use the 'Validate JSON Schema' to validate the request against your schema.
In the event that validation is bypassed, it is likely to produce cryptic error messages further down the flow.

Versioning Strategies

Versioning usually applies to cases where there is a schema change associated with the data that is coming into the flow via the endpoint. The format of the schema i.e. whether XML, JSON or a custom format is not really applicable but the required change in schema in a flow that has already been deployed. The manner in which a service handles multiple versions of this schema is a discussion that is platform agnostic. The following sub-sections address the different means of deployment to handle different versions of data coming into the flow.

Schema Versioning

Schema versioning are normally carried out using namespaces. Use a version attribute in the XML instance document. That will enable a processing application or a version adapter service to figure out what it is processing.
In this scenario, the same runtime service is modified to handle multiple versions. The key aspects of this scenario are summarized below:

- There is only one runtime engine that is capable of handling multiple schema versions
- The choice logic at the start of the flow is responsible for identifying the schema version
- Based on the schema version, different logic paths are taken within the flow
- New elements within the schema must be introduced such that the schema validation at the start of the flow is able to handle the multiple versions, e.g. for XML schemas the new elements must be optional.
- Alternatively to above, the schema validation at the start of the flow must be turned OFF and the sub-flow logic must then carry out the schema validation in accordance with different schema formats i.e. the different versions.

The advantages of this approach include:

- No impact on the end consumers as the same endpoint URL is used
- Lower memory footprint as the single service is handling multiple versions

The disadvantages of this approach include:

- The service becoming version aware introduces a maintenance overhead
- End consumers are less likely to migrate to the new version.

Application Versioning

In this scenario, an additional runtime service is created to handle the new version of the schema. The key aspects of this scenario are summarized below:

- There are as many runtime services as there are schema versions.
- The implementation of the service need NOT be schema aware. Each service is tied to a single version of the schema and as a result the implementation becomes simpler and easier to maintain.

The advantages of this approach include:

- Ease of Maintenance.
- Logical separation of end consumers as there is a one to one mapping between consumers and the actual service they are consuming,
- Provides a better path from migration of end consumers to the new service

The disadvantages of this approach include:

- Minor impact on the end consumers as the new service will adopt a new endpoint URL
- Higher memory footprint as there are now additional services deployed within the ESB

API Catalog

Applications can be created with new versions of API and managed in the API portal. This is available in the CloudHub Anypoint API platform. The platform allows you to manage the lifecycle of your API.
Refer https://developer.mulesoft.com/docs/display/current/Managing+API+Versions

Transactions

It is recommended to place atomic units of work within a message flow in a transactional scope. Not all components offer a compensating transaction i.e. a rollback. The Mule activities that offer transactional support include JMS, JDBC and VM components. As a best practice, the involvement of other activities that do not have a rollback capacity within a transactional scope is to place the activities with the most likely potential for an error at the end of the scope. For example placing an SMTP and JDBC activity within a transactional scope will cause the JDBC component to rollback if there is a failure in the SMTP component.

Connectors

Connectors are reusable components that interact with Mule ESB and the target system. A Connector enables Mule flows to communicate with the target resource. The connector conveys data between the resource and a Mule flow, and transforms data into a Mule message. A well-developed connector must support the following features –
Connection Management – This ensures that the connector handles all connections behind the scenes by creating a pool of connection resources for clients to use.
Pagination – Allowing caching and indexing of data so that large result sets can be handled gracefully.
Session Management – Every connection is associated with a session. Session management ensures that each connection has a

time-to-live attribute defined so that connections can be terminated and not held forever.
Input/output Meta Data – Connectors expose the input/output fields/attributes required for the target system.

Reconnection Strategy

The majority of connectors in MULE (such as Salesforce, Database, RabbitMQ, FTP, etc.) offer a reconnection strategy. This strategy allows for the configuration of additional connections over a specified time interval in the event that the original connection has been deactivated. As explained before in section Flow Exceptions, connector exceptions are handled outside of the message flow. There are three key points that need to be highlighted here:

- Connector exceptions can be captured via the extension of the Mule Context. This is useful for integrating connector exceptions with whichever logging strategy that has been adopted to ensure connector exceptions are stored in the same manner as log events.
- Reconnection strategies are useful in ASYNC interactions where the original caller is NOT waiting for a response. Reconnection strategies should not be used in synchronous interactions. This is because every reconnection strategy is associated with following two attributes: reconnection interval and the number of times the connector can try to reconnect. If the time interval * reconnection attempts exceed the original caller timeout, then the client will timeout no matter if the reconnection was successful after x number of attempts.
- All reconnection parameters should be configurable.

A detailed configuration guide for each of the connector is given in the Mule public documentation site. An example URL for FTPConnector can be found here
https://developer.mulesoft.com/docs/display/current/FTP+Connector

API Platform

The API platform consists of three core sections -

1.
   a. Environment configuration: The configuration of the Anypoint Platform organizations, their associated roles, and gateway mappings.
   b. API lifecycle: The processes to be followed to manage APIs across environments and tenants including a description of the what/how of automation.
   c. Enhancements based on roadmap: Future enhancements feasible based on the Anypoint Platform for APIs 2015 roadmap.

Environment Configuration

This section describes the steps required for the configuration of Anypoint Platform organizations and their association with gateways. This configuration is a prerequisite for the API lifecycle operations to take place.

Create organizations -
It is generally recommended that two separate Anypoint Platform organizations be created for a single shared API management infrastructure: One organization to serve to serve the needs of APIs across all preproduction environments, and another organization to address the needs of APIs purely for the production environment. This model allows for a clear separation between production and preproduction and facilitates the configuration of separate identity management infrastructures between these environments. If a federated identity model is being used, each organization needs to be federated with its target SAML 2.0 identity provider system in preproduction and production.
Configure roles -
Organization roles allow for the management of different levels of access to different APIs for different users.
Note that the creation of new users and their mapping to the appropriate roles can be scripted by using the Anypoint Platform core services API.
Add users –
As mentioned in the previous section, all users of the Anypoint Platform need to be created within the Anypoint Platform user repository itself through invitation by organization administrators. Users are added through invitations, and the invitation process allows for users to be added to the appropriate roles.
Note that the creation of new users and their addition to the appropriate roles can be
scripted by using the Anypoint Platform core services API.
Configure gateways –
IT administrators need to configure an API gateway instance (or cluster) per department/initiative and per environment. The configuration of each gateway instance/cluster requires two important steps:
1. The configuration of the gateway's shared SSL port with appropriate certs so that it can be used to expose APIs using HTTPS and
2. The configuration of the gateway with the appropriate Anypoint Platform organization (i.e preproduction or production).

API Life Cycle

Once the gateways are configured, APIs can be created within the system and moved through their lifecycle.

Once a proxy is deployed to a gateway the API is ready to be managed. The sections below detail the steps associated with each step.
Create -
Once an API owner has created a new API, a user belonging to the organizational administrator role needs to add the API to the appropriate role depending on the API's department/initiative and environment.
It is recommended that APIs be created with a convention whereas their version string contains the name of their associated environment, such as for example:
API Name: Quote
API Version: v1_qa
Note : API creation and its association with the appropriate roles can be automated through the usage of the API platform's own published APIs

Design and publish -
Once an API has been created, API owners can define it using RAML and publish it using an API portal. Both of these steps are optional. The benefit of having a RAML definition on a RESTful API is that it leads to an automatically generated API console that can then be used by application developers to play around with the API to understand it better. The RAML definition can also be used by API owners to define API notebooks, which show how the API's different pieces can be used with other APIs to accomplish useful tasks. Finally, an API with a RAML definition also allows for the creation of RAML based proxies that honor the definition of the RAML (in terms of ensuring that incoming calls adhere to the API contract defined by the RAML).
If a portal is created for an API, it will automatically include the API's console (if a RAML has been defined for it). Portals also allow for the definition of arbitrary content that allow for the better documentation of an API. Finally, API portals can be themed so that application developers have an experience that reflects the brand associated with that API.
Note that the association of an API version with a portal is necessary for an API to be available for application registration by application developers (i.e. API consumers).
Deploy proxy -
For an API to be manageable (that is for policies to be applicable to it and for analytics data to be collected) it needs to have an associated proxy application deployed to an API gateway. A proxy application for an API is configured by specifying the APIs implementation URL and the URL port and path of the proxy frontend. The generated proxy application as-is simply forwards requests from the frontend path to the implementation URL.
A proxy application is a Mule application and, if needed, can be extended with additional capabilities such as for example ensuring that it's outbound connection uses HTTPS or that all calls are forwarded to two backend endpoints instead of one. Typically, such post proxy generation modifications are performed for functions that can **not** be applied as custom policies using the Anypoint Platform's API management capabilities.
The target API gateway for the deployment of the proxy depends on its tenant, and environment.
Note : the recommendation is to create post generation scripts that enable proxies to call their backend APIs through an HTTPS outbound endpoint instead of plain text outbound call. The deployment of the proxy itself can also be automated through integration with the Mule Management Console.

Manage -
Once an API's proxy is deployed it is ready to be managed through the application of policies. The following policies are recommended:
1) SLA based throttling policy: To ensure that
a) Application registration is required and enforced at the gateway layer, and that
b) Each application's access is throttled at the rate for which it has been approved.
2) OAuth token validation policy: A policy that validates access to the API by checking requests against an external OAuth server with which the organizations are configured.
Promote -
An API can be moved from one environment to the next through a sequence of export and import operations. An export operation on an API version exports the following pieces of information into a single "API bundle file":

- The API's name, version name, tags, and endpoint metadata.
- The API's RAML definition (if defined).
- The API's portal (if published).
- The API's proxy configuration (if configured).
- The API's associated policies.

An import operation can be performed on an exported bundle into a target organization. The target organization can be the same as the one from which the API version was exported.

Export and import operations can be used in combination with the API creation, proxy deployment, and management phases described so far in order to promote an API from one environment to the next. The following is a description of each illustrated step.
1. Export API version from source environment.
2. Import API version into target environment using its naming convention.
a. Ensure that newly imported API version (from now on referred to as the target API) belongs to appropriate roles.
3. Reconfigure the proxy of the target API to point to the implementation API if different in the new environment.
a. Generate the proxy and apply post proxy generation logic to proxy as per the new API creation process described before.

b. Deploy the proxy to the gateway belonging to the new target environment.

API lifecycle automation -
The lifecycle of an API can be automated through the creation of the following scripts:
API creation script: Every time a new API is required, an administrator runs this script which accepts as input the following information:

- API's name
- API version
- API description
- Target tenant
- Target environment

The script then uses the API platform's own published APIs in order to automate the creation, role mapping, portal publishing (optionally), and proxy generation for the API.
API deployment script: This script accepts as input the following information:

- Generated proxy
- A target gateway environment
- Any post generation processing dependent information needed (e.g. SSL certs for backend API call)

The script modifies the generated proxy according to its post generation template and deploys it to the gateway in the target environment. Deployment would be done through integration with MMC.
API promotion script: This script would accept as input the following information:

- The source organization
- The destination organization (if different)
- The API name and version
- Target tenant
- Target environment

The script begins by exporting the API from the source organization and imports it into the destination organization. The script then proceeds to call the API deployment script with a proxy generated from the destination organization and using the target tenant and environment as a gateway destination.

Enhancements based on roadmap

Anypoint Platform roadmap capabilities that are planned for delivery in 2015. These enhancements are summarized in the table below and expanded on in the following sections –

Aspect	**Today**	**Aspect Today With 2015 roadmap enhancements**
Roles	Custom roles that are closely managed	Sub organizations with default permissions
Deployment	Through MMC as automated today for ESB applications	Through unified Anypoint Platform management API

Security - Confidentiality

The APThe Anypoint Platform is in compliance with some of the highest security standards in the industry. CloudHub has been certified for Level-1 PCI-DSS, Health Information Trust Alliance (HITRUST) Common Security Framework (CSF) certification and SSAE 16 SOC 2.
The HITRUST Common Security Framework (CSF) is a framework that seeks to normalize security control implementations of healthcare organizations including federal (e.g., ARRA and HIPAA), state (Mass.), third party (e.g., PCI and COBIT) and government (e.g., NIST, FTC and CMS). The CSF is not a new standard; rather, it attempts to unify the control requirements of many disparate standards such as PCI, HIPAA, etc. The CSF supplements the existing controls with the industry knowledge and leading practices of HITRUST's community and provides the clarity and consistency lacking in many standards and regulations.

API Keys

The MuleSoft's Anypoint Platform automatically generates an Application Client ID and Client Secret key when developers register an application on the platform. Developers can access application information from the My Applications tab in the Developer Portal. The My Applications display provides a unique client ID and client secret for the application, which needs to be passed with the API calls for APIs that are protected with policies.
And with the API Contract Manager, NYUAD can then approve or revoke the API keys that have been generated for the developers.

API Authentication and Authorization

The following security policies are supported and available on the MuleSoft's Anypoint Platform.

Policy Template Name	Description
HTTP Basic Authentication	Enforces authentication per the details configured in a Security Manager policy.
LDAP Security Manager	Injects an LDAP-based security manager into the target API.
Simple Security Manager	A placeholder security manager that can be configured with a hard-coded username and password for testing purposes.
OAuth 2.0 Provider	Configures an OAuth 2.0 authorization server at the target API.
AES OAuth 2.0 Access Token Enforcement	Configures the API so that its endpoints require a mandatory and valid OAuth 2.0 token.
PingFederate Access Token Enforcement	Configures the API so that its endpoints require a mandatory and valid PingFederate token. Note that this policy is only available to organizations using a PingFederate Federated Identity Management system.
OpenAM Access Token Enforcement	Configures the API so that its endpoints require a mandatory and valid OpenAM token. Note that this policy is only available to organizations using an OpenAM Federated Identity Management system.

The above policies allows to enforce Basic Authentication, OAuth and SAML 2.0 based authentication and identity federation. Authorized users are have been authenticated would then be allowed to access the APIs managed on the Anypoint Platform.

Encryption

The Anypoint Platform includes the Anypoint Enterprise Security components which includes the Message Encryption Processor. This allows NYUAD to support message-level encryption of all the message payloads across the platform beyond just relying on the transport-level encryption that is available, such as TLS/SSL.
Mule ESB's Anypoint Enterprise Security supports encryption and decryption of message content using JCE, XML, or PGP. It also facilitates fine-grained encryption or decryption for fields within the message payload. Any properties exposed in a configuration properties file (known as the Mule Credentials Vault) can be encrypted and decrypted.

Security - Integrity

As stated in the previous section, the Anypoint Platform includes the Anypoint Enterprise Security. This suite of security features provides various methods for applying security to Mule Service-Oriented Architecture (SOA) implementations and Web services. The following security features bridge gaps between trust boundaries in applications:

- Mule Secure Token Service (STS) OAuth 2.0a Provider
- Mule Credentials Vault
- Mule Message Encryption Processor
- Mule Digital Signature Processor
- Mule Filter Processor
- Mule CRC32 Processor

Mule Credential Vault

Use the Mule Credentials Vault to encrypt data in a .properties file. (In the context of this document, we refer to the .properties file simply as the properties file.)
The properties file in Mule stores data as key-value pairs which may contain information such as usernames, first and last names, and credit card numbers. A Mule application may access this data as it processes messages, for example, to acquire login credentials for an external Web service. However, though this sensitive, private data must be stored in a properties file for Mule to access, it must also be protected against unauthorized – and potentially malicious – use by anyone with access to the Mule application. How do you protect the data in a properties file while still making it available to Mule? Use three ingredients:

1. a Mule Credentials Vault
2. a global Secure Property Placeholder element
3. a key to unlock the vault

Refer https://docs.mulesoft.com/mule-user-guide/v/3.6/mule-credentials-vault

Logging & Auditing

An interesting challenge that's likewise sure to emerge is how to handle logging in distributed environments such as those of Web services. Unlike operating system logs, which are physically located on a single machine or a single network device, Web services are by their nature distributed across multiple systems, disparate technologies and policies, and even organizational domains. This creates many constraints on audit log design in terms of both scope and strength. Given these constraints, logging might reside in different parts of Web services architectures:

1. In the simplest case (two machines talking to each other via a Web service), we could probably log events occurring on both machines, but we won't have a complete view of the overall architecture. If we put a log on each machine, we'll need additional technologies for consistent aggregation and reconciliation.
2. At each endpoint, Web services interaction involves multiple software components. At the very least, this will involve a Web server, an application server, and most likely a database server on each side. Moreover, the application server might be self-distributed, thus presenting additional challenges to determine where to put the log.

Name	Attachment
Best practices for the web service logging	Logging and Auditing.docx

Logging:

1. RECIEVE Task (onComplete): ENTRY >> {BP_NAME} : {BPID} : {ANY UID FROM REQUEST} : {message}

Description: To denote start of a bpel process

1. REPLY Task (onStart): REPLY << {BP_NAME} : {BPID} : {ANY UID FROM REQUEST} : {SUCCESS/EXCEPTION/FAULT}

Description: To denote end of a bpel process

1. EXIT Task (onStart): EXIT << {BP_NAME} : {BPID} : {ANY UID FROM REQUEST} : {SUCCESS/EXCEPTION/FAULT}

Description: To confirm the complete exit from the current bpel process instance

1. INVOKE Task (onStart): INVOKE >> {BP_NAME} : {BPID} : {ANY UID FROM REQUEST} : {INVOKED SERVICE NAME} : {INVOKED SERVICE OPERATION NAME} : {message}

Description: To denote invocation of a partner like CRM/LDAP/Integration-DB technical services

1. INVOKE Task (onComplete): RESPONSE << {BP_NAME} : {BPID} : {ANY UID FROM REQUEST} : {INVOKED SERVICE NAME} : {INVOKED SERVICE OPERATION} : {message}

Description: To denote end of invoking a partner and response from partner like CRM/LDAP/Integration-DB technical services

1. Any other Task (onStart/onComplete): <TASKNAME> >> {BP_NAME} : {BPID} : {ANY UID FROM REQUEST} : {message}

Description: To denote a simple static message or text for any other BPEL tasks like Assign, Empty, Condition, For Each, Throws, Catch, Catch All, etc

Appendix

Common Resources

Maven remote-resources-plugin can be used to create common resources bundle. The following steps explain what is required to create and use a common resources project.

- In the CommonResources POM.xml make sure the packaging is changed from mule to jar.
- Execute the following command

  ```
  mvn clean install
  ```

- Install the CommonResources binary (jar) in your local maven repository by running the following command :

  ```
  mvn install:install-file -DgroupId=com.mycompany -DartifactId=commonresources -Dversion=1.0.0-SNAPSHOT -Dpackaging=jar
  -Dfile="/Users/krishnanramaswamy/AnypointStudio/mule/commonresources/target/commonresources-1.0.0-SNAPSHOT.jar"
  ```

  Make sure you change the file path to reflect your local file system.
- In the target project POM.xml where you want to make use of these resources add the following to your POM

  ```
  <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-remote-resources-plugin</artifactId>
  <version>1.5</version>

  <executions>
  <execution>

  <goals>
  <goal>process</goal>
  </goals>
  <configuration>
  <resourceBundles>
  <resourceBundle>com.mycompany:commonresources:${project.version}</resourceBundle>


  </resourceBundles>

  <attachToMain>true</attachToMain>
  </configuration>
  </execution>
  </executions>
  </plugin>
  ```

- Execute the following command

  ```
  mvn install package –P <profile>
  ```

  Where <profile> is your target maven profile as defined in your POM

  Base Document

WW_MuleE...1.2.docx