

API Implementation

Recommendations

<customer\_name>

v1.0, 20xx-xx-xx

# Overview

API Implementation Recommendations covers recommended practices for implementing the API contract using flows, components and connectors provided by the MuleSoft Anypoint Platform.

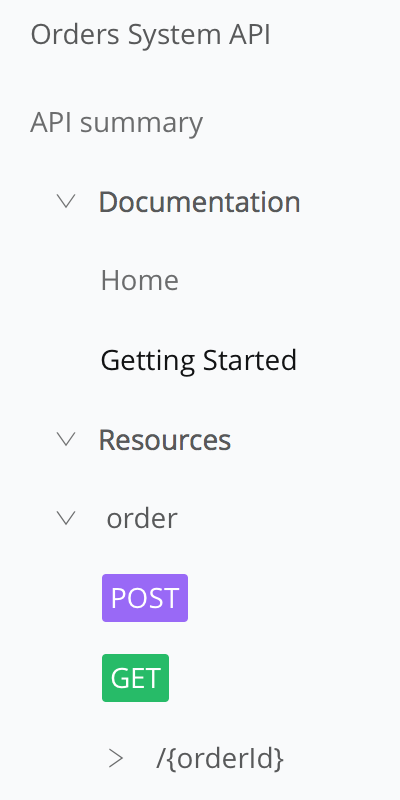
# API Contract (RAML)

## API Identification

API identification describes the identity of the API contract and how it can be understood by a prospective consumer. Take the following example (orders-process-api.raml):

|  |
| --- |
| #%RAML 1.0 --- #=============================================================== # Products API - RAML example # References: # - RAML Specification - http://raml.org/spec.html # - RAML Projects - http://raml.org/projects.html # - RAML Tools - http://www.apihub.com/raml-tools #=============================================================== title: Orders System API version: v1.0 #List of media type to support mediaType: application/json #List of protocols to support for baseUri protocols: [ HTTP, HTTPS ]  #=============================================================== # API documentation #=============================================================== documentation:  - title: Home  content: This is a prototype order API.  - title: Getting Started  content: TODO |

* The name of the RAML file should be something other than the default api.raml (e.g. orders-process-api.raml).
* The title of the API expressed in the RAML should be human readable (e.g. Orders System API).
* The version of the API contract (RAML) should *always* be defined and the versioning standard should be consistent across all APIs.
* While optional, specify the mediaType that the request/response bodies of the API will accept and return.
* Ensure that the protocols used by the API are defined. HTTPS is standard but can also include HTTP (recommended only for internal calls).
* Including documentation elements in the API contract is valuable when publishing to the API portal within Exchange. Adding the documentation tag in the RAML file automatically generates pages within Exchange for view.



## Resource Definitions

* Following ReST best practices, resources should be defined as nouns and should be a single word in plural form.

|  |
| --- |
| /orders:  displayName: An order placed by a customer.  description: One or more orders placed by a customer. |

* If the resource needs to be defined as multiple words, specify the resource using kebab-case.

|  |
| --- |
| /archived-orders:  displayName: An order placed by a customer that is no longer active.  description: Orders that have either been cancelled, delivered or resolved. |

* Every resource should also be defined as a type. Ensure that all types are defined either within the API contract or (*preferred*) referenced as an external file.

|  |
| --- |
| OrderItem:  type: object  properties:  id: string  name: string  quantity: number  description: string   Order:  type: object  properties:  id: string  name: string  items: OrderItem[] |

## Action Definitions

* Following ReST best practices, HTTP methods should be considered as action verbs that are being applied for each resource. For clarity, provide documentation to explicitly define each operation.

|  |
| --- |
| get:  displayName: Retrieve all orders for last month (or input time range).  description: Retrieves all orders currently in the ‘Active’ state. |

* Non-CRUD operations (or operations that don’t follow ReST standards) should *always* be extensively documented for prospective consumers.
* Resources that allow for only certain fields to be returned in the response should define the multi-word fields either in camelCase or snake\_case.

|  |
| --- |
| GET /deliveries?fields=id,customerName,dateDelivered GET /deliveries?fields=id,customer\_name,date\_delivered |

* Ensure that the proper HTTP Response Status codes are defined for each operation within the resource:

|  |
| --- |
| responses:  400:  body:  application/json:  type: APIStatus-Response  example: !include examples/API-Status-Response-400.raml  401:  body:  application/json:  type: APIStatus-Response  example: !include examples/API-Status-Response-401.raml  403:  body:  application/json:  type: APIStatus-Response  example: !include examples/API-Status-Response-403.raml  404:  body:  application/json:  type: APIStatus-Response  example: !include examples/API-Status-Response-404.raml  500:  body:  application/json:  type: APIStatus-Response  example: !include examples/API-Status-Response-500.raml  502:  body:  application/json:  type: APIStatus-Response  example: !include examples/API-Status-Response-502.raml  504:  body:  application/json:  type: APIStatus-Response  example: !include examples/API-Status-Response-504.raml |

## Reusability

* To encourage reuse during design, RAML data types should be externalized. For example, the Order data type can be externalized into its own RAML DataTypes file (Order.raml):

|  |
| --- |
| #%RAML 1.0 DataType  type: object  properties:  id: string  name: string  items:  description: The items contained within the order.  type: array  items:  type: !include OrderItem.raml |

* Similarly, all externalized data types should have an associated externalized NamedExample representation (e.g. Order-Response-Example-200.raml):

|  |
| --- |
| #%RAML 1.0 NamedExample value:  apiStatus:  name: 'Orders Data API'  code: '200'  link: 'http://example.org/..../Orders.html#200'  description: 'OK'  transactionId: '1234567890'  title: 'Order retrieved successfully.'  order:  id: '1234567890'  name: 'Initial order.'  items:  -  id: '1234567890'  name: 'Widget'  description: 'It is a widget.'  quantity: 1 |

* Reusable traits (e.g. Client-ID Enforcement) should also be externalized whenever possible. For example (Client-ID-Required.raml):

|  |
| --- |
| #%RAML 1.0 Trait queryParameters:  client\_id:  type: string  client\_secret:  type: string |

## Testability

* When possible, explicitly define all of the potential status codes that the API may return when processing a request. This will significantly help the testing team plan their test cases beyond the simple “happy-path”.

|  |
| --- |
| Responses:  200:  body:  application/json:  type: Order-Response-Example-200  example: !include examples/Order-Response-Example-200.raml  400:  body:  application/json:  type: APIStatus-Response  example: !include examples/Order-Response-Example-400.raml  401:  body:  application/json:  type: APIStatus-Response  example: !include examples/Order-Response-Example-401.raml  403:  body:  application/json:  type: APIStatus-Response  example: !include examples/Order-Response-Example-403.raml  404:  body:  application/json:  type: APIStatus-Response  example: !include examples/Order-Response-Example-404.raml  500:  body:  application/json:  type: APIStatus-Response  example: !include examples/Order-Response-Example-500.raml  502:  body:  application/json:  type: APIStatus-Response  example: !include examples/Order-Response-Example-502.raml  504:  body:  application/json:  type: APIStatus-Response  example: !include examples/Order-Response-Example-504.raml |
|  |

* Providing a consistent API response can really help when parsing the results of a given API. Not only does the additional data provide human-readable output for easier resolution, but the consistent format allows automated tools (e.g. Log File Analyzers) to easily parse and categorize the error.

|  |
| --- |
| #%RAML 1.0 NamedExample value:  status: '400'  errorCode: '1'  developerMessage: 'Technical description of error.'   userMessage: 'User friendly description of error.'  moreInfo: 'https://api-customer.com/resetSponsorLoad/v1'  transactionId: '92b83605-8828-4e26-ad65-88e7efa1c5c5' |

* Ensure that all externalized data types and examples are valid RAML prior to implementation in Anypoint Studio.

## Security

* If the API requires a specific security scheme, ensure that the scheme is externalized to its own file. For example (OAuth2-Security.raml):

|  |
| --- |
| #RAML 1.0 SecurityScheme type: OAuth 2.0 description: |  API supports OAuth 2.0 for authenticating all API requests. describedBy:  headers:  Authorization:  description: |  Used to send a valid OAuth 2 access token. Do not use  with the "access\_token" query string parameter.  type: string  queryParameters:  access\_token:  description: |  Used to send a valid OAuth 2 access token. Do not use with  the "Authorization" header.  type: string  responses:  401:  description: |  Bad or expired token. This can happen if the user or Dropbox  revoked or expired an access token. To fix, re-authenticate  the user.  403:  description: |  Bad OAuth request (wrong consumer key, bad nonce, expired  timestamp...). Re-authenticating the user won't help here. |

* Any confidential data present in examples should either be removed or obfuscated.

## Traceability

* In order to trace a message across multiple API calls, each method call should support the concept of a Correlation ID that can be passed in the header of the request. For more information, please consult:

<https://blogs.mulesoft.com/dev/anypoint-platform-dev/total-traceability/>

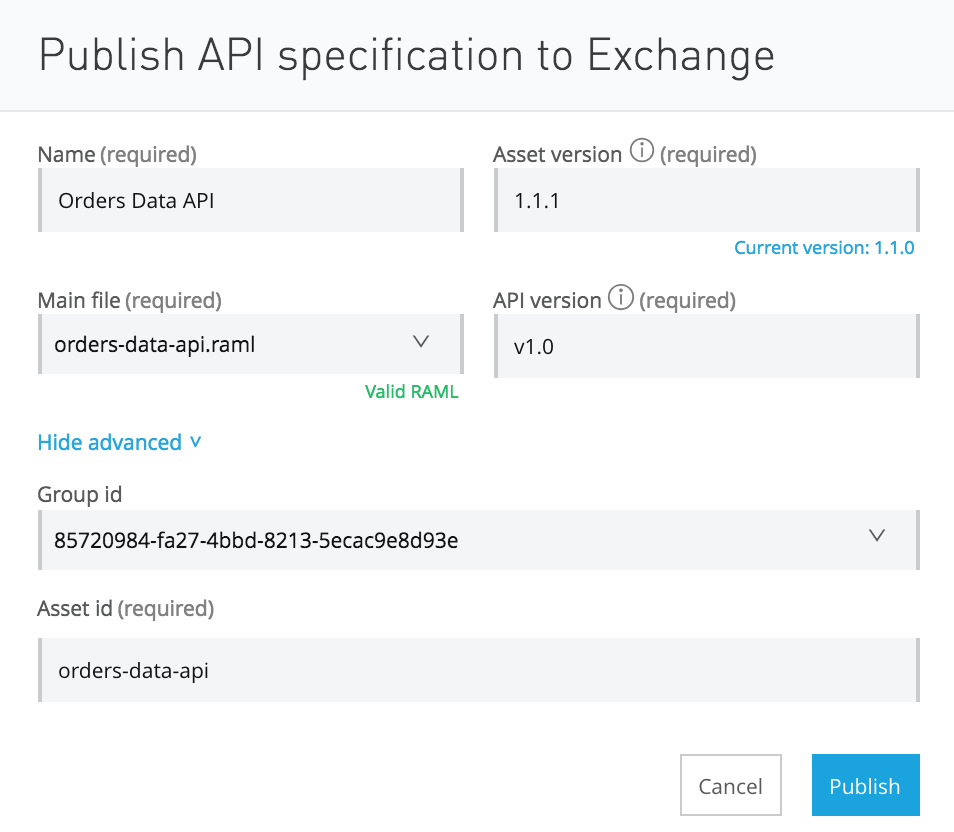
For example, this Correlation ID (expressed as the HTTP standard header request field X-CORRELATION-ID) can be passed via a request header. The header can be encapsulated within a RAML trait (Traceable.raml):

|  |
| --- |
| #%RAML 1.0 Trait headers:  X-CORRELATION-ID:  description: Identifier for tracking the message across multiple APIs.  type: string  example: 2c1fe860-ab8f-11e8-98d0-529269fb1459 |

## Availability

* Once the API has been designed, always make sure to publish the design to Exchange. Ensure that the name of the Exchange asset is human-readable and that the API version is correct.

Publishing to Exchange makes the API design available for review by potential consumers and allows others to provide feedback.



## 

# API Development

## Readability

* In order to simplify development scope, there should *only be a single API developed per application*. There can be multiple resources offered per API, but the scope of the API should be constrained based on:
  + data provided by the system (System API),
  + the orchestration required to fulfill a business function (Process API)
  + or the channel in which the API is used (Experience API).

Constraining the scope also helps ensure that the API is recognizable by potential consumers in Exchange and allows the operations teams to scale the API more effectively.

* Every API should be developed against an API-kit generated configuration file. The RAML specification is the connection between what the potential consumer reviews in Exchange and the actual API capabilities being developed. It is critical to the concept of API-led Connectivity that the API contract remain consistent between design and development.
* Each flow should be labeled to provide insight into the function/output of the flow. For example:
  + retrieve-current-orders-flow
  + update-all-orders-batch
  + remove-item-from-order-flow

Avoid default flow names (e.g. orders-impl-flow) or single-word flow names (e.g. orders-flow) when possible. For flow naming conventions, refer to the MuleSoft Code Style Guide.

* Each component in the flow should be self-documenting. Each component should describe the capability it provides within the flow. For example, change the DataWeave default name (Transform Message) to something more descriptive:

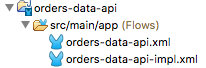


* If common flows/subflows are used within the API, either:
  + separate them within the same configuration file and provide an alternative naming structure
    - authorize-token-common-flow
  + or externalize them into a separate configuration file.
    - Orders-system-api-common.xml

Common flows (e.g. JSON logging and exception handling) can be provided as dependencies available within the customer’s artifact repository. New APIs/applications can easily include them as dependencies using Maven’s excellent dependency management system.

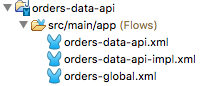
## Maintainability

* Separate implementation from interface. Implementation code should not be written in the configuration file generated by APIkit. Create a separate configuration for implementation flows. For example:



The only changes that should be made to the generated APIkit configuration is to provide Flow Reference components to the flows defined in the implementation.

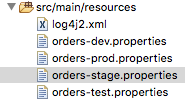
* For any global elements that defined, consider separating these into a special ‘globals’ configuration. A global element is configured once and then referenced many times from elements within multiple flows. For example:



* All applications using Mule 3.9.x or below, should be ‘Mavenized’. For more information on project ‘Mavenization’, please consult:

<https://docs.mulesoft.com/anypoint-studio/v/6.5/enabling-maven-support-for-a-studio-project>

* Throughout the different phases in the development lifecycle (i.e. Dev, QA, Staging or Production), the API may need to be configured differently per environment (e.g. server names or URLs, login credentials, etc.). Externalizing all of the environment-specific configuration parameters into separate properties files is the key to developing a single application across multiple environments. For example:



For more information, please consult: <https://docs.mulesoft.com/mule-user-guide/v/3.9/deploying-to-multiple-environments>

* When externalizing any properties to separate files, ensure that any confidential/sensitive properties are stored securely.

For Mule 3.9.x and below, use the Mule Credentials Vault.

* + <https://docs.mulesoft.com/mule-user-guide/v/3.9/mule-credentials-vault>

For Mule 4 and above, use Secure Configuration Properties.

* + <https://docs.mulesoft.com/mule4-user-guide/v/4.1/secure-configuration-properties>
* Check that all components (especially third-party system connectors) are current and up-to-date.

## Performance

* DataWeave is MuleSoft’s powerful data transformation engine and has been extensively optimized to provide the best performance for any data transformation need. Ensure that any flow requiring data transformation is using DataWeave.
* Define reconnection strategies for any component that manages connectivity to an external server (e.g. HTTP and FTP connectors). Reconnection strategies specify how a connector behaves when its connection fails.
* Calculate the initial performance baseline for each API. A performance baseline is the expected performance of an application under certain conditions. For an API, one way to measure a performance baseline is by calculating the average response time at the maximum throughput that it can consistently sustain.

## Testability

* MUnit tests should be developed and executed periodically to ensure no regressions have been identified. Ideally, the execution of the MUnit tests should be performed by an external Continuous Delivery server, but test execution should always occur prior to committing to the Version Control System (VCS).
  + MUnit tests should be developed for all flows within the API.
  + At least one error path should be tested per API method.
  + A test should be written for every single HTTP status code identified in the API design(e.g. 200, 400, 404, 500, etc.).
  + Overall MUnit code coverage > 60%.
    - For more about MUnit Code Coverage, refer to https://docs.mulesoft.com/munit/v/2.1/munit-coverage-report

## Security

* Determine if appropriate security mechanisms been defined and configured for the API. Consider the following:
  + API Authentication/Authorization
  + Message Confidentiality & Integrity

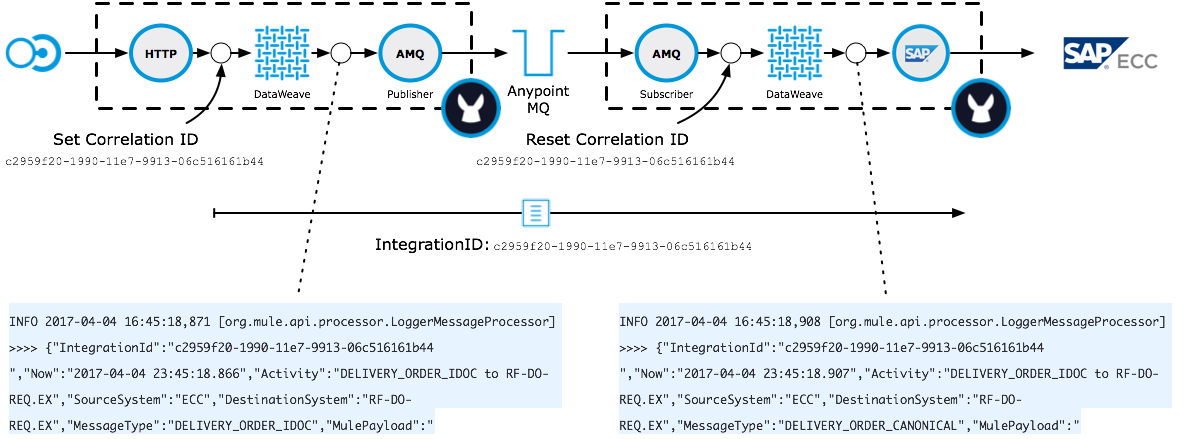
The most common means for supporting authentication and message

confidentiality is by using TLS. TLS supports both through the use of public-key cryptography which requires the exchange of keys between the API consumers and provider.

* + When using One-Way TLS, ensure that the keystore has been created with its associated keys.
  + When using Two-Way TLS, ensure that the keystore and truststore have been created with their associated keys and certificates.

## Traceability

* As mentioned in the traceability design considerations, Correlation IDs should be used to track messages as they flow through each Mule application. The input connector for each flow should be maintaining the Correlation ID as it is transmitted.



## Reusability

* When possible, adopt a common exception/error handling strategy across all APIs and applications. This strategy should include processes for handling synchronous and asynchronous Mule applications. For more information on building a global strategy for handling exceptions/errors in Mule, please visit:

<https://blogs.mulesoft.com/dev/howto/howto-exceptions-and-error-handling-in-mule/>

A standard exception strategy should be adopted for both synchronous and asynchronous flows. The implementation of the strategy should be available within the customer’s artifact repository (Nexus/Artifactory). New APIs/applications can easily include the strategy using Maven’s dependency management system.

* Follow a common logging strategy for all exceptions and APIs. The strategy should include a standard logging layout (configurable in the log4j2.xml file) and formats for the logged messages.
* When required, follow standard auditing practices as defined by the organization. Each organization may have unique auditing requirements and thus an auditing strategy should be defined for each requirement identified.

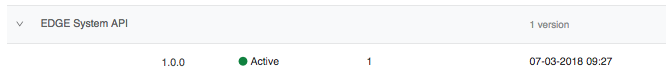
# 

# API Platform Onboarding

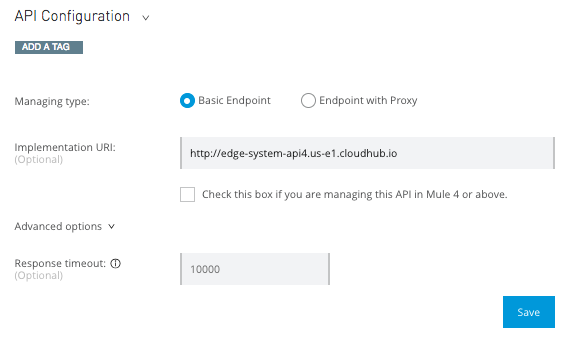
Onboarding refers to the process of deploying the API into Runtime Manager (either via CloudHub or to an on-premise runtime).

## Discoverability

* Ensure that the API Contract (RAML) has been published to the API Manager, autodiscovery has been implemented in the API, and that the status of the API is Active.

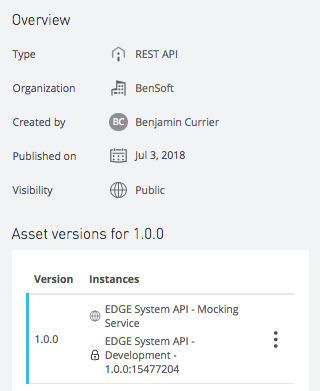


* The API endpoint should also be registered in the API Manager as either a Basic Endpoint or an Endpoint with Proxy.



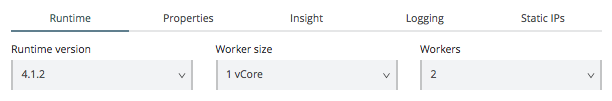
Check that the the Implementation URI is correct for the current environment.

* Once published, the Visibility of the API Portal should be made Public in Exchange.



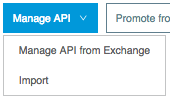
## Performance

* Based on the initial performance baselines, the application should be “sized” to its performance requirements:
  + Worker Size (Scaling Up) - If the complexity of the API is high (e.g. requiring a significant amount of computation/memory), then the vCore size should be adjusted to accommodate the requirements.
  + Number of Workers (Scaling Out) - If the expected usage of the API is high (e.g. multiple consumers) or the API supports a critical process (i.e. requires multiple instances in case of error), then the number of workers should be adjusted to accommodate the requirements.



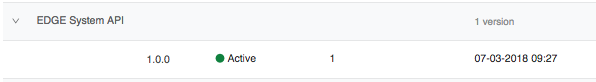
## Availability

* Check that the API is being managed via API Manager. The API contract should be loaded into API Manager via Exchange.



Also check that the autodiscovery elements have been added to the implementation so that API Manager is aware of the status of the API.

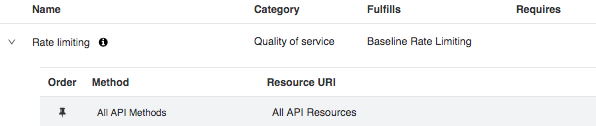
Once the API is deployed into Runtime Manager, check the status in API Manager. If the status of the API is Active, then the API is being successfully managed.



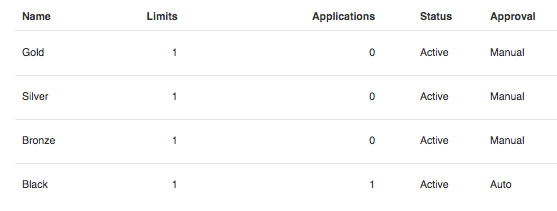
* To support the overall quality of service of the API, apply appropriate throttling/rate limiting policies to the API.

Quality of service policies define the *quota per time window* configuration for the rate limiting/throttling algorithm. The algorithm is executed when the first request is received. This event fixes the time window. Each request consumes quota from the current window until the time expires. When quota is exhausted, the resulting action depends on the policy:

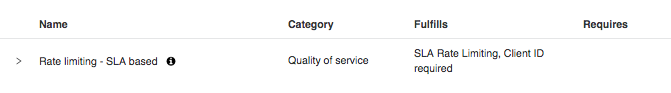
* + Rate limiting rejects the request.
  + Throttling queues the request for retry.



* Ensure that the Service Level Agreement tiers have been defined for the API.



Also ensure that the *SLA based* Throttling/Rate Limiting policies have been applied for the API in API Manager.



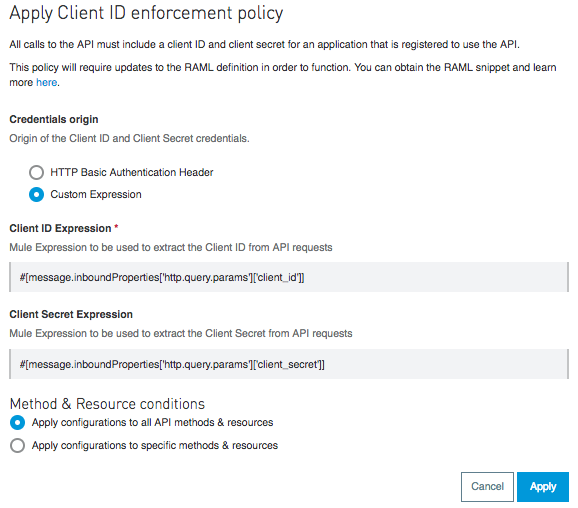
All SLA based policies mandate the usage of Client ID enforcement and thus the API contract should already have client-id-required RAML trait defined.

Also, if the Client ID enforcement policy has already been applied, it will need to be removed before any SLA based policies can be applied.

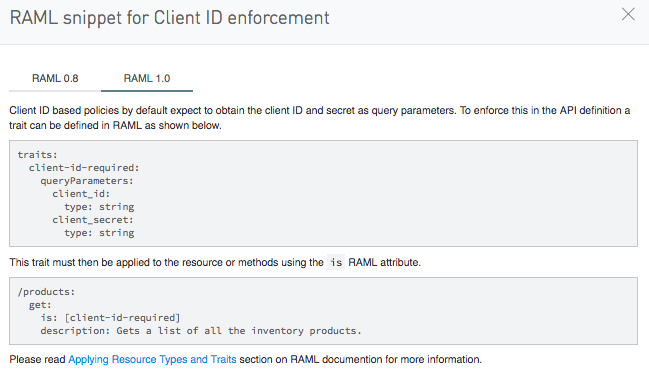
## Security

* If no SLA based policies have been applied (see above), ensure that the Client ID enforcement policy been applied. The Client ID enforcement policy allows *only* those applications that have a registered client\_id and client\_secret access to the API.

The way in which the client\_id and client\_secret are passed via the request can also be configured. The default means is to specify the client\_id and client\_secret in the query parameters of the URL request. But the values can be passed as unique request header entries or they can be passed using a HTTP Basic Authentication Header.



All Client ID enforcement policy also requires that the API contract have the client-id-required RAML trait defined.



* When using OAuth 2.0, the appropriate OAuth 2.0 policy should also be applied. For more information on OAuth 2.0 policies and setup for using an external OAuth 2.0 provider, review the following:

<https://docs.mulesoft.com/api-manager/v/2.x/oauth2-policies-new>

<https://docs.mulesoft.com/access-management/managing-api-clients>

## Traceability

* After a successful deployment into Runtime Manager and registration into API Manager, check to see if analytics for the API are being collected. For more information:

<https://docs.mulesoft.com/api-manager/v/2.x/viewing-api-analytics>

* If additional information is required, enabling API Insight can provide more granular data around transaction execution. For more information:

<https://docs.mulesoft.com/runtime-manager/insight>

**Note:** *Enabling Insight may cause a performance impact when processing application data. Enabling Insight in production environments is strongly discouraged for long periods of time.*