**API Management Best Practices**

* [Objective](#APIManagementBestPractices-Objective)
* [Audience](#APIManagementBestPractices-Audience)
* [Best Practices](#APIManagementBestPractices-BestPractice)
  + [SLA Tiers](#APIManagementBestPractices-SLATiers)
  + [API Policies](#APIManagementBestPractices-APIPolicies)
  + [API Alerts](#APIManagementBestPractices-APIAlerts)
  + [Analytics](#APIManagementBestPractices-Analytics)

**Objective**

The following guide is intended to show and describe the best practices around API Management**.**

**Audience**

* Application/Operator Admin
* C4E Architect / Architects
* C4E Core Developers

**Best Practices**

**SLA Tiers**

* Despite you can define SLA tiers per API, as a best practice the SLA definitions should be homogeneous across all APIs or across defined groups of APIs.
* Define SLA tiers for each API to enforce the approval workflow and access limits (limit the number of requests an application can make to the API).
* The Names and values of the tiers should be standard across all APIs following a naming-convention

| **Tier Name** | **Approval** | **Limits (example)** |
| --- | --- | --- |
| **Basic** | Auto | 100 requests / hour |
| **Gold** | Manual | 100 requests / minute |
| **Platinum** | Manual | 100 requests / second |

* The limits are purely descriptive and should be enforced by using policies. Enforce the SLA tiers with SLA-based policies such as**rate-limiting** and **throttling**
* For more information: <https://docs.mulesoft.com/api-manager/defining-sla-tiers>

**API Policies**

* The policies should be applied homogeneous across all environments
* Apply at least one of the following security-related policies
  + Client ID enforcement
  + Open ID token enforcement (OAuth2)
* Apply an SLA rate-limiting policy defining the limits based on the Performance Testing results for each API implementation

**API Alerts**

* Despite you can define Alerts per API, as a best practice the Alert definitions should be homogeneous across all APIs or across defined groups of APIs.
* See: Notifications and Alerts (TBD)

**Analytics**

* Use the default dashboard to see API consumption parameters
* Create a custom dashboard with custom charts if needed

**API Resource design guidelines**

**DEPRECATED - Please use:**API Design Recommendations

**Resource Naming Conventions**

The key principles of REST support separating an API into logical resource and manipulating these using HTTP methods, where each method has a specific meaning. A resource represents object types within your domain.  For example, /employees would present employees with your organization.

**Use Nouns, Not Verbs**

A URI should refer to a resource that is a “thing” instead of referring to an action. Therefore, avoid using “actions” within your resource name. For example, getOrders or deleteOrder should be avoided. The action prefix should be implied by the HTTP method i.e. GET or DELETE.

Examples of resources would be:

/orders

/payments

/shipments

**Use Plural Nouns**

It is common practice to standardise on using plural nouns over a mixture of both singular and plural nouns in URIs. The makes it consistent and predictable for developers.

**Use Concrete Nouns**

Avoid making a resource name too abstract. Tunneling a number of objects through an abstract resource name makes it difficult to understand what the resource actually represents or how the API resource should be used. For example, consider the different types of services – order, payment and shipment. Representing all these services as a /services resource is too abstract.

**Identify Actions**

The next step in the design process is to identify the actions that can be performed on the resource model. When designing RESTful API the API designer should try to use as much as possible CRUD actions, that is, Create, Retrieve, Update and Delete. Similar to the “nouns” rule, the API designer should consider the verbs and try to identify candidate actions.  For example, the customer should be able to create a purchase order, to add products to that order (that is, to create Order lines that refer to a specific product and belongs to a Purchase order), and to retrieve his/her orders. Further, the customer wants to be able to change (that, is to update) the shipping address of an order. It could be the case that the customer might want to completely delete a purchase order that has not been shipped yet.

**Map Actions to HTTP Verbs**

On the first view REST using standard HTTP verbs are a matching implementation of the CRUD pattern. The main difference is that CRUD is a pattern to be applied to entities in systems and data stores while REST using HTTP commands implements system behaviour. Nevertheless, in many cases the CRUD-REST mapping can be used to explain the behaviour of the RESTful API. Closer inspection shows that there are differences which need to be taken into account. In the majority of the cases REST services can be mapped as shown in the following table:

|  |  |  |
| --- | --- | --- |
| **CRUD** | **Description** | **HTTP mapping in REST services** |
| Create | Create an Entity on the service side | POST |
| Read | Retrieve an Entity from the service | GET |
| Update | Implements a persistent change of the Entity on the service | PUT (replace) / PATCH (partial update) |
| Delete | Remove an Entity from the service | DELETE |

Problem in the mapping is the use of the POST and PUT verbs. While PUT is commonly used to update an entity on the service side (the entity does exist and some of the attributes are modified), the POST verb is commonly used to create a new entity (as a subordinate of a special resource). Some implementations are using the PUT verb for “create” and “update”, with the difference that the implementation creates an entity on the service side if it does not exist during an “update” function. Important for the particular implementation is the requirement that the behaviour of PUT and POST are different: the PUT verb is idempotent (i.e. if the verb is repeated, the outcome on the service side is identical) while POST is not.

In some cases, the action do not map to CRUD actions. There are several approaches to deal with this:

* Restructure the action to operate on a field of a resource. This works if the action does not use parameters. For example, ship order could be mapped to PATCH that updates only the filed “shipped”.
* Use a sub-resource to capture the state that results from the execution of the action. For example, the state of an offer could be changed by creating a sub-resource Shipped Order. Then, an order state could be changed by *POST /orders/{id}/shipped-order*.
* In some cases, it is impossible to map an action to a reasonable resource structure. For example, a search action over a collection of multiple, different resources. In such cases, it makes sense to define a “fake” resource “Search” but this has to be documented clearly to avoid confusion.

**Resource States**

Analysing the user stories, the API designer finds out that the user should be able to perform some of the action only if certain preconditions are met. For example, it should be possible to change the shipping address or cancel an order only if the order has not been shipped yet. This implies that the API resources could have different states, e.g., order could be created, shipped or cancelled. It is a good practice to capture the states of the resources and the respective transitions that change the state of a resource in a state diagram as shown below:

**API URLs**

The structure of a URI is central to how APIs are organised and categorised within your enterprise domain. A good URI taxonomy helps to categorise your APIs across functional domains, regions, access (public or private) and helps define relationships (hierarchical). A good URI also helps to govern the lifecycle of your API through versioning practices.

Recommended URI Structure:

|  |  |  |
| --- | --- | --- |
| **Part** | **Description** | **Example** |
| {env} | **Optional**. The API environment. An API could be available in a sandbox environment to enable developers to test that API. The {env} part is excluded for production APIs. | sandbox |
| {access} | **Optional**. The access level of the API. This could be public or private. By default the {access} part is excluded for public APIs or simply set to "api". | api |
| {company} | Required. The name of the company or business division for private services. | mytaxis |
| {region} | Required. The region of the API | .[co.uk](http://co.uk) |
| {context} | Required. The name of the API as defined in the API Manager. This typically presents the business service and should be a short but descriptive name. | quickbooker |
| {version} | Required. The version of the API. Depending on requirements, the version can reflect only major versions or include a more hierarchical convention to identify minor versions. | v1 |
| {resource} | Required. The name of the resource that represents the actual object. An API may contain multiple resources. The resource can also be referred to as the API endpoint. | bookings |
| {resource-id} | Optional. The id of the resource to be fetched/updates. The resource id is optional. | 1981927 |
| {queryparams} | Optional. The query string can define state transition parameters. | page=1&sort=+<field> |

**Setting the Base URI**

Once the APIs URLs are established, the baseURI defined in the RAML should reflect the DNS entires: http://[env].[access].[company].[region]/[context]/[version] as described in the table above.

**Filtering**

In some cases, the API consumer might only need a subset of a collection of resources. This could be accomplished by using query parameters. For example, to get the list of all shipped orders, the API consumer could use:

GET /orders?state=shipped

Here, the "state" query parameter is used to filter the response.

**Sorting**

Similar to filtering, a generic query parameter sort could be used to describe sorting rules. To allow sorting on multiple fields, the query parameter could be designed to take a list of fields instead of a single value.

Next, to allow for ascending and descending sort order, the query parameter could take minus (“-“) as a prefix of each field.

For example, the following request will return all purchase orders sorted by date (descending) and then by product (ascending):

GET /orders?sort=-date,product

**Partial Resources**

In some cases, the consumer might not need all the fields of a resource. To allow for obtaining only a partial resource the API URL could be designed to take a list of fields as a query parameter, and return only the fields that are included in that list.

For example, the following request will return only the date and the total of the purchase order:

GET /orders/1?fields=date,total

**Aliases**

To make the experience of using an API more pleasant for the application developers, the API could package a set of conditions into an easily accessible URL. For example, to return the recently shipped orders, the API could provide the following endpoint: *GET /orders/most-recent*

A resource name should remain short in order to avoid any size limitations.  The base URL should also contain no more than 2-3 resources if possible. URIs can be limited in some HTTP stacks.

**Asynchronous Processing**

**Problem**

RESTful APIs use HTTP, which is a synchronous protocol.  An application sends requests, blocks, and waits for a response. However,  in some cases when the process has the potential to take a significantly long time (due to a slow backend system), this is not the desired behavior. Typical examples are mobile or modern Web application that provides reactive UI. Such an application can submit a request to create a new object, continue with some other task, and only show a notification when the customer object is created in the backend system and ready to consume.

**Solution**

Design the API so that an application can trigger asynchronous work. And only track the processing status from time to time. When the result is ready, the application should be able to retrieve it, or if the processing is not required any more, the application should be able to cancel it.

As in the previous case, this can be done using standard HTTP headers and status codes. For that reason, we use HTTP code 202 Accepted to inform the application that its request has been accepted for processing.

The W3C’s HTTP standard (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>) defines 202 code as follows:

**202 Accepted**

The request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place. There is no facility for re-sending a status code from an asynchronous operation such as this.

The 202 response is intentionally non-committal. Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed. The entity returned with this response SHOULD include an indication of the request's current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

The W3C’s HTTP standard (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>) defines 303 code as follows: The Location header is used to provide a link to such a status monitor. The application can use the URI provided in the Location header to poll the status of the processing. Once the processing is finished, the API provides the location of the actual result again using Location header but with a different HTTP status code, namely 303.

**303 See Other**

The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The new URI is not a substitute reference for the originally requested resource.

The 303 response MUST NOT be cached but the response to the second request might be cacheable.  
The different URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI.

Note: Many pre-HTTP/1.1 user agents do not understand the 303 status. When interoperability with such clients is a concern the 302 status code may be used instead since most user agents react to a 302 response as described here for 303.

DELETE /resources/123/status request: In some cases, the application may want to cancel the processing. This could be achieved by sending

The following RAML snippet demonstrates the pattern:

#%RAML 1.0

title: Example Asynchronous API

version: v1

mediaType: application/json

types:

Data: object

Status:

properties:

status:

enum: [working, canceled, failed, success]

/resources:

post:

body:

type: Data

responses:

202:

description: |

The request has been accepted for processing. Use the URI provided in the Location: header to monitor the status of the processing

headers:

Location:

example: /resources/123/status

/{id}:

get:

responses:

200:

body: Data

/status:

get:

responses:

200:

description: |

The API is still processing the request.

body:

type: Status

example: {"status": "working"}

303:

headers:

Location:

example: /resources/123

description: |

The processing has finished successfully.Use the URI provided in the Location: to obtain the actual result

body:

type: Status

example: {"status": "success"}

500:

description: |

The processing has failed

body:

type: Status

example: {"status": "failed"}

delete:

responses:

200:

description: |

The processing has been canceled

body:

type: Status

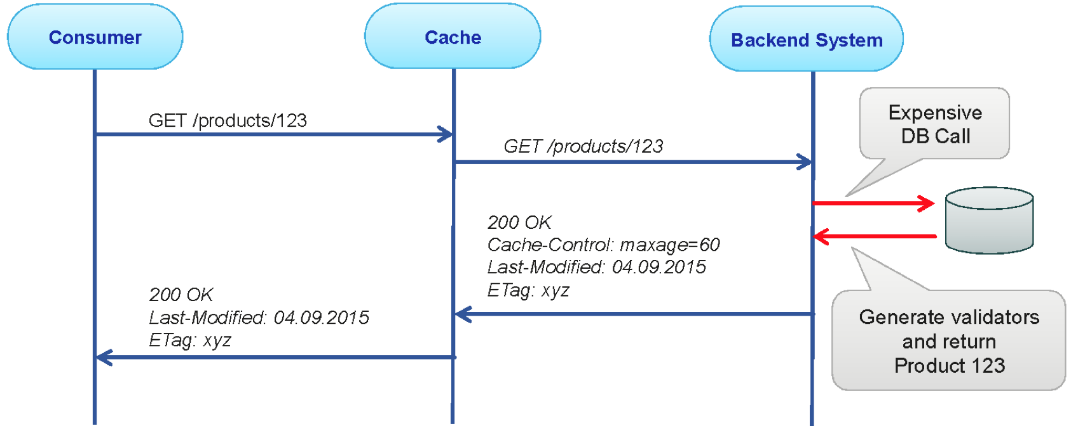
example: {"status": "canceled"}

# Caching

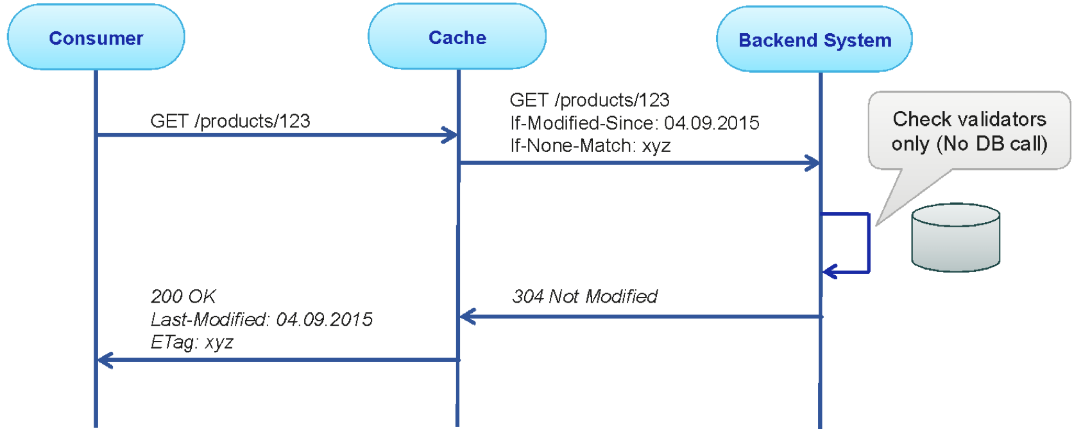
The Caching Pattern has been introduced to improve the performance and load management of the API. In many cases the information requested from the API is fairly static, with limited change over time (typical examples: address information on locations, price information, …). In these cases the API does not necessarily have to retrieve the information from back-end systems, but keep it in memory close to the API, called a Cache. Only if the information is not available in the Cache, the integration retrieves the information from the back-end. Based on the frequency of the retrieval and the size of the Cache, this information from the back-end could be added. The following charts describe an example of the caching pattern:

The Consumer requests information on the product 123 from the product API. The Cache does not have valid information on the product in the store and sends the request to the back-end system which retrieves the information.

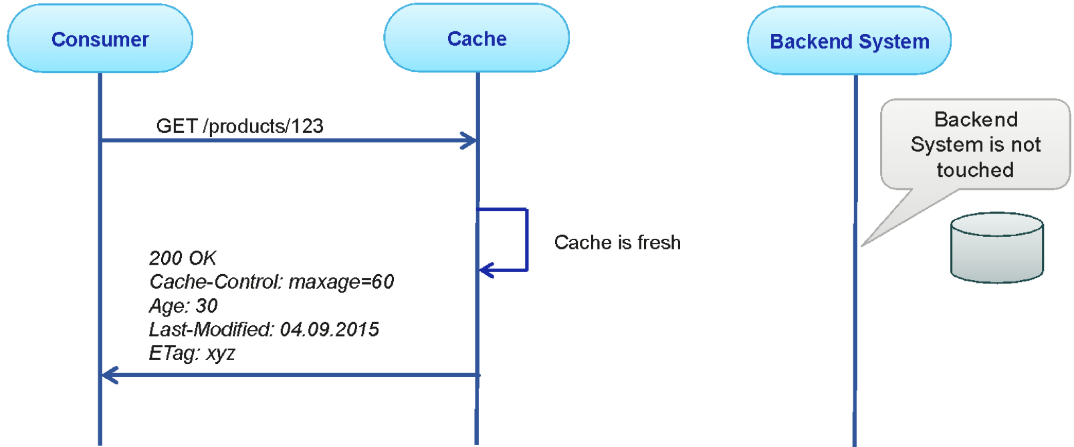
As this is the first request of the information the Cache does not keep the information as frequently used in local memory, but returns it back to the consumer.



The second request for the product now triggers the cache to inspect the information retrieved. As the information has not changed it is likely to be static. Also the information has an expiration time added, so the Cache can keep the information local for any following request.



Any following request will now be handled by the Cache unless the information is expired in the Cache or (because of changes in the back-end system) removed from the Cache.



It is important that the Cache is refreshed (i.e. deleted) if the back-end is changed, or the maximal Cache storage period (age of the information) exceeds the allowed age of the information.

**Performance Test Report Example**

* [Scope](#PerformanceTestReportExample-Scope)
* [Monitoring Points](#PerformanceTestReportExample-Monitoring)
* [Test Plan](#PerformanceTestReportExample-TestPlan)
* [Load Tests to Find the Baseline](#PerformanceTestReportExample-LoadTestst)
* [Soak Tests](#PerformanceTestReportExample-SoakTests)
* [Results](#PerformanceTestReportExample-Results)

**Scope**

|  |  |
| --- | --- |
| **Key** | **Value** |
| Goal | Verify if the system remains stable during 4hs of work-load |
| Environment | Test |
| Application | sys-customers |
| API resource | /customers |
| Data | internalId=123456  client\_id (specific for performance testing)  client\_secret (specific for performance testing) |
| Nodes to be monitored | * Cloudhub workers * API Manager - Analytics |
| Nodes excluded | Backend system nodes: PeopleSoft server |
| Baseline redline | Do not allow CPU% to exceed 70%.  Do not allow Memory% to exceed 70%. |
| Testing tool | JMeter: <http://jmeter.apache.org/> |

**Monitoring Points**

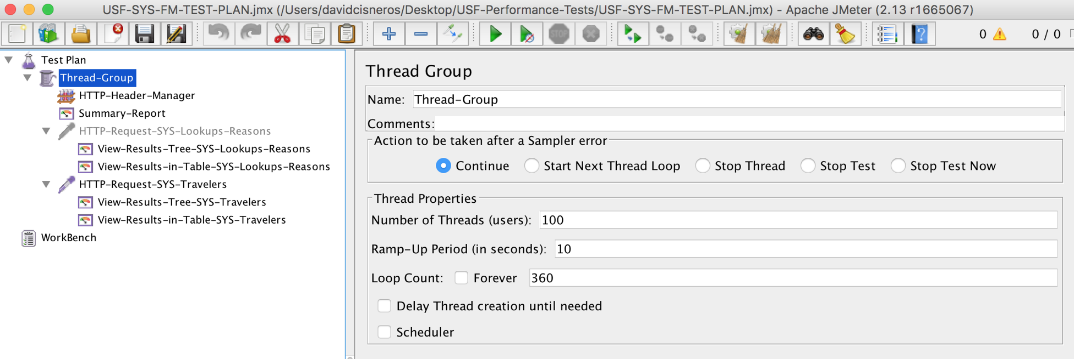
|  |  |
| --- | --- |
| **Node** | **Metrics** |
| Workers | CPU  Memory  Mule Messages |
| API Manager | TPS  HTTP Error Codes |
| PeopleSoft | Excluded |
| Local machine | CPU  Memory |

**Test Plan**

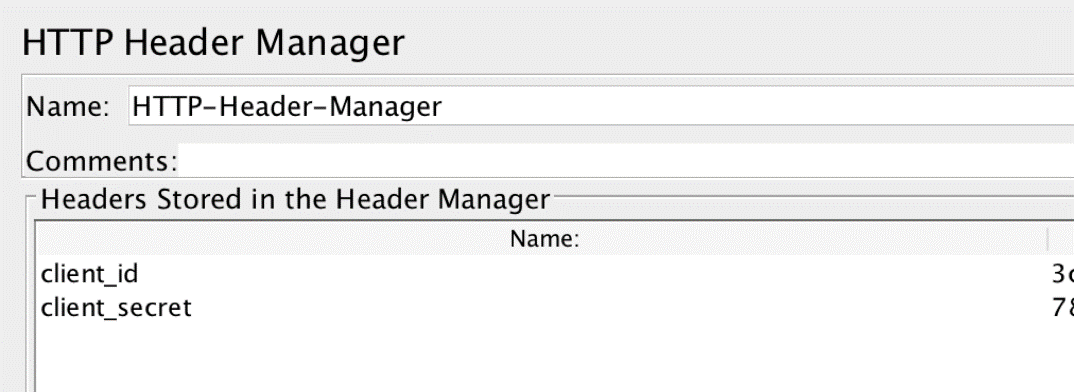
Configuring the Thread Group.

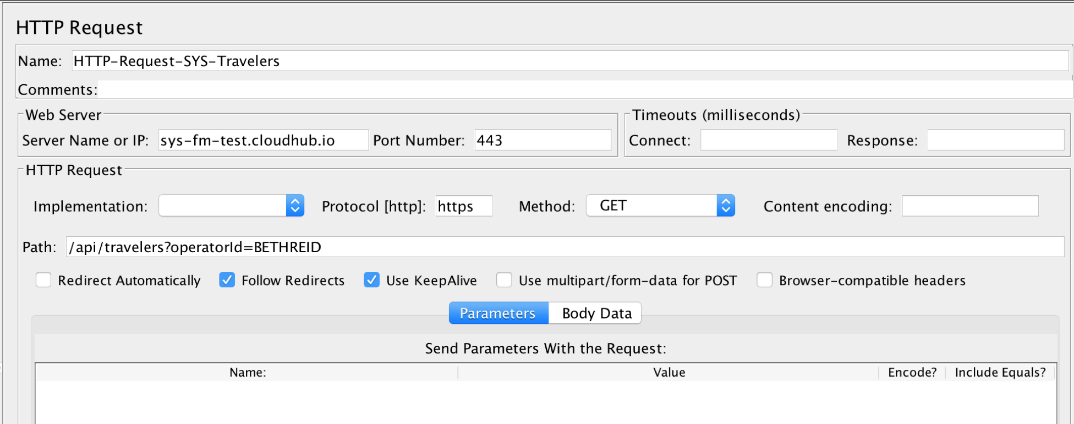
|  |  |
| --- | --- |
| **Key** | **Value** |
| Threads | 100 |
| RampUp | 10 |
| ramp-up/threads relation | 0.1 (every 0.1 secs a new thread is created) |

Define the Thread Group.

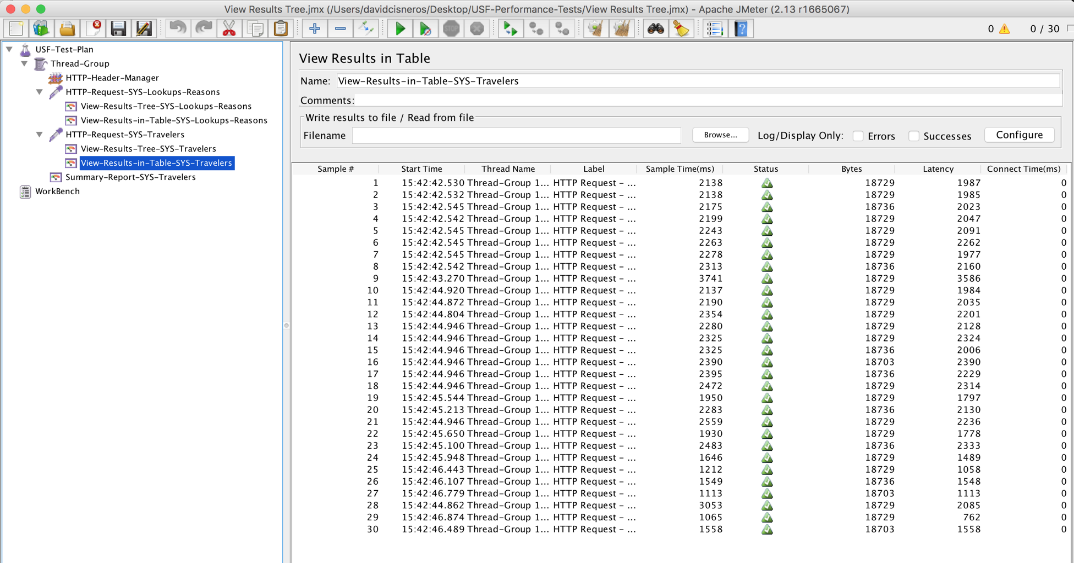


Define the headers (client\_id/client\_secret) in a header manager.





**Load Tests to Find the Baseline**



**Soak Tests**

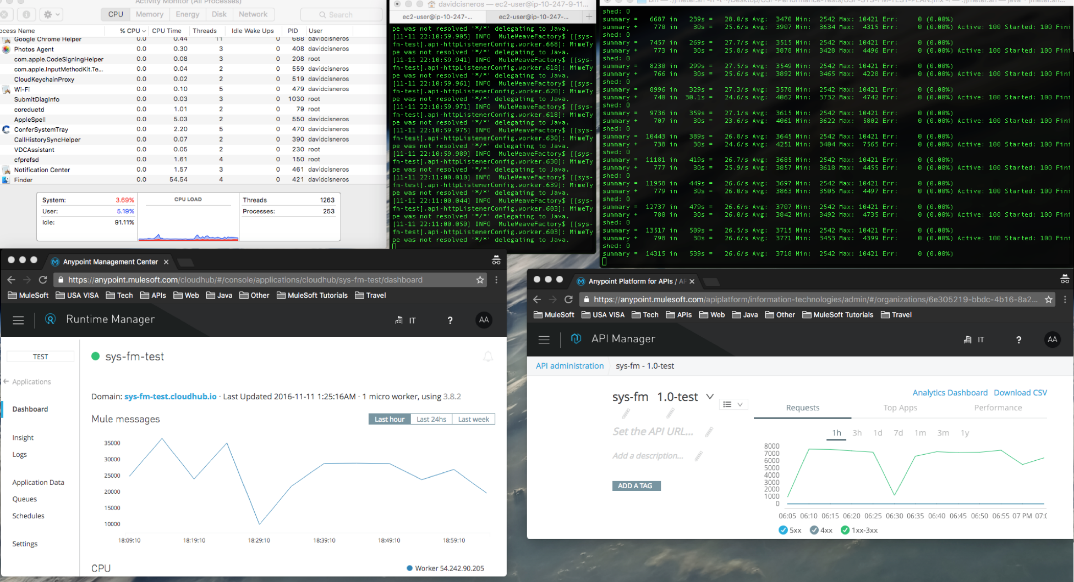
Base Line + loop count increased to reach the desired time.

|  |  |
| --- | --- |
| **Key** | **Value** |
| Threads | 100 |
| RampUp | 10 |
| ramp-up/threads relation | 0.1 (every 0.1 secs a new thread is created) |
| loop count | 360 (x 4) |

Execute from Command Line: from {JMETER\_HOME}/bin

./jmeter.sh -n -t {JMETER\_PLAN\_PATH}/JMETER\_PLAN\_NAME.jmx -l {LOGS\_OUTPUTH\_PATH}/LOG\_ANY\_NAME.csv

Execution and monitoring (Cloudhub, API Manager, Local/Test-exec Machine monitoring)



**Results**

* Baseline: 100 threads, ramp-up 10
* Stability time-unit: 4 hs
* Stability: Ok



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.

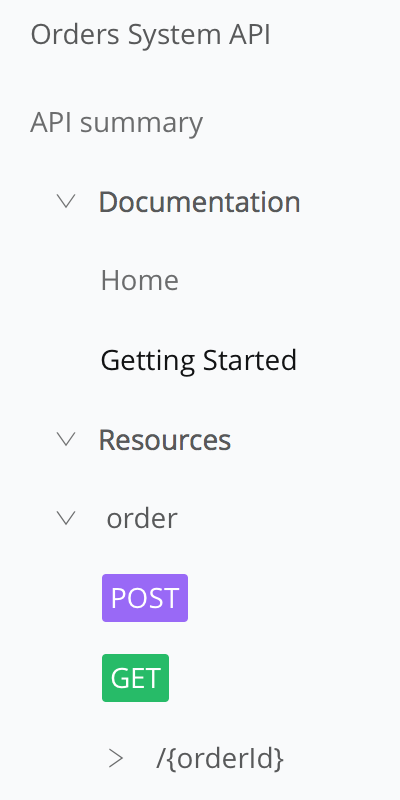
API Contract (RAML)

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/>

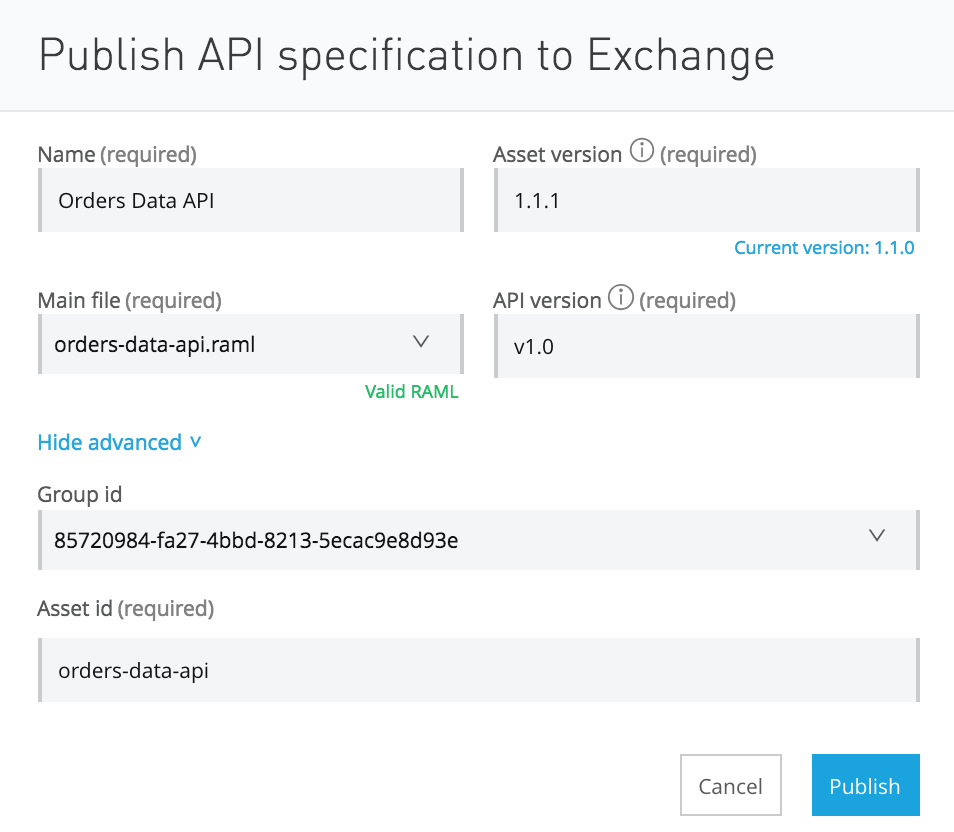
For example, this Correlation ID (expressed as the HTTP standard header request field X-CORRELATION-ID) can be passed via a request header. The header can be encapsulated within a RAML trait (Traceable.raml):

|  |
| --- |
| #%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:

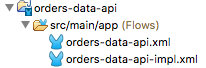


* If common flows/subflows are used within the API, either:
  + separate them within the same configuration file and provide an alternative naming structure
    - authorize-token-common-flow
  + or externalize them into a separate configuration file.
    - Orders-system-api-common.xml

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.

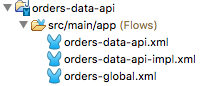
Maintainability

* 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.

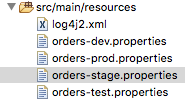
* For any global elements that defined, consider separating these into a special ‘globals’ configuration. A global element is configured once and then referenced many times from elements within multiple flows. For example:



* 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

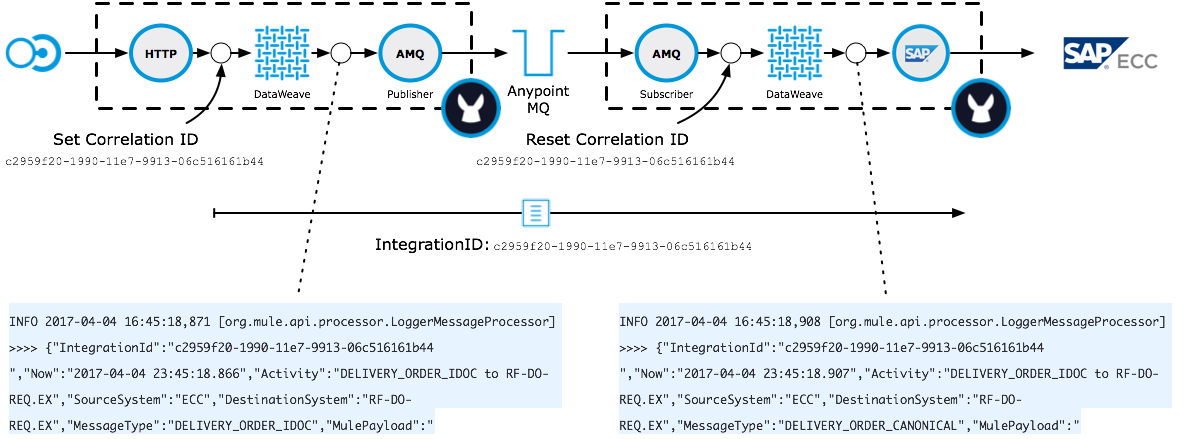
The most common means for supporting authentication and message

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.

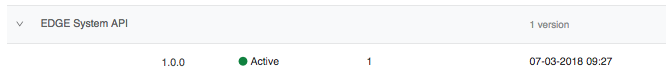
* Follow a common logging strategy for all exceptions and APIs. The strategy should include a standard logging layout (configurable in the log4j2.xml file) and formats for the logged messages.
* When required, follow standard auditing practices as defined by the organization. Each organization may have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

API Platform Onboarding

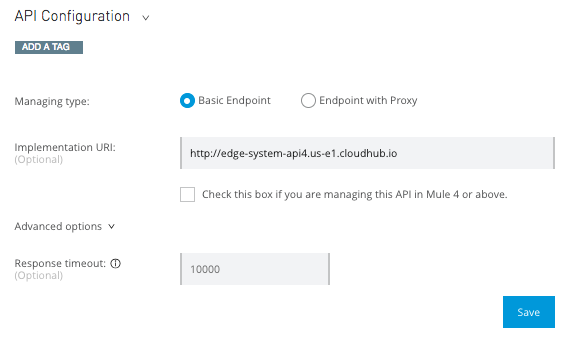
Onboarding refers to the process of deploying the API into Runtime Manager (either via CloudHub or to an on-premise runtime).

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.

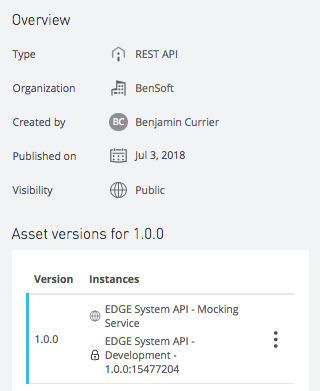


* The API endpoint should also be registered in the API Manager as either a Basic Endpoint or an Endpoint with Proxy.



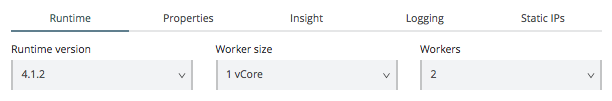
Check that the the Implementation URI is correct for the current environment.

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



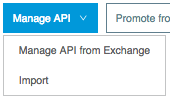
Performance

* 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.



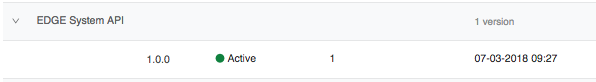
Availability

* 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.

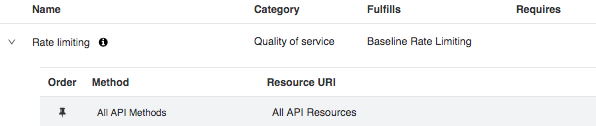
Once the API is deployed into Runtime Manager, check the status in API Manager. If the status of the API is Active, then the API is being successfully managed.



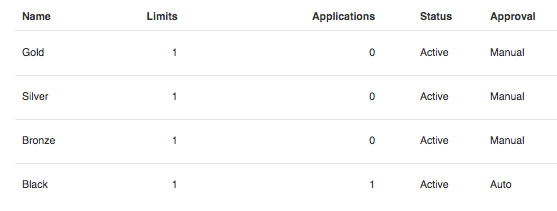
* To support the overall quality of service of the API, apply appropriate throttling/rate limiting policies to the API.

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:

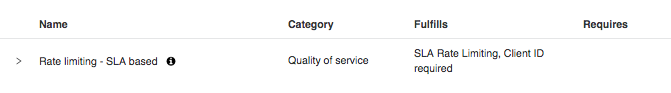
* + Rate limiting rejects the request.
  + Throttling queues the request for retry.



* 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.



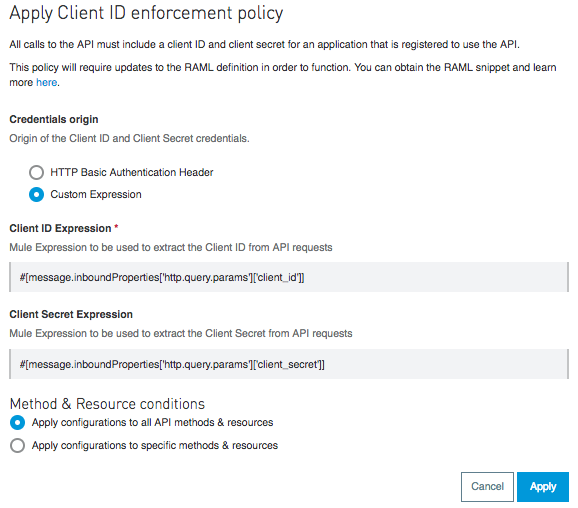
All SLA based policies mandate the usage of Client ID enforcement and thus the API contract should already have client-id-required RAML trait defined.

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.

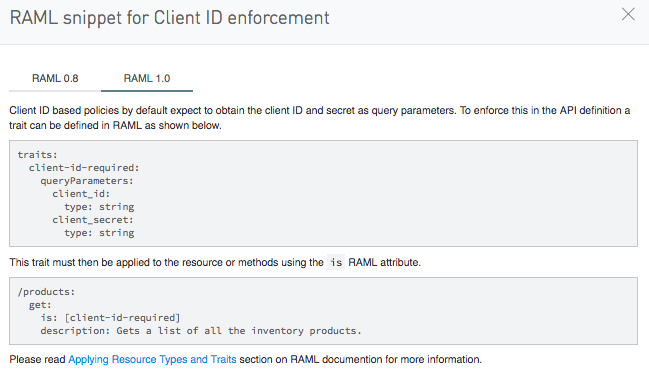
Security

* 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.*

**API Design Conventions**

* [REST Design Conventions](#APIDesignConventions-RESTDesignConventi)
* [RAML Conventions](#APIDesignConventions-RAMLConventions)

**REST Design Conventions**

|  |  |
| --- | --- |
| **Topic** | **Description** |
| Resources | * Use nouns, not verbs * Coarse grained, not fine grained * Architectural style for use-case scalability * Use lower case (example: /accounts) * For resources with more than 2 words   + use lowercase for both words (example: /lineitems) or   + use kebab-case (aka spinal-case) (example: /line-items) |
| Resource types | * Collection resource   + /users * Instance resource   + /users/{id}  (example: /users/123 ) |
| Behaviour | Create the resources with the right verb logic.   * GET: For obtaining data * PUT  (Idempotent): To update data (the entire instance) * POST (Not Idempotent): To store data * PATCH: To update partial data of an instance * DELETE: To delete an instance |
| Media Types | * For the request: use ‘Accept’ header * For the response: use ‘Content-Type’ |
| Base URL & Versioning | Include the “api” word and the version of the API in the base Url (e.g. domain**/api/v1**)  **(pre CROWD2 only)** If you have the same API in multiple environments, you should differentiate them in the version of the API Manager (e.g. v1-test, v1-dev ), pointing to a different implementation application (e.g. [mydomain-dev.com/api/v1](http://mydomain.com/api/v1), [mydomain-test.com/api/v1](http://mydomain-test.com/api/v1)) |
| CamelCase | Use camelCase for all the names (fields), preferably don’t use underscores. |
| Date/Time representation | * Use standard date formats: ISO8601 * Use UTC   + **2016-10-27T13:42:21+00:00    (+00:00 is the time zones hour offset)**   + **2016-10-27T13:42:21Z     (Z is place holder for local time zone)** |

**RAML Conventions**

|  |  |
| --- | --- |
| **Topic** | **Description** |
| Version | * Specify the RAML version:   + <https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/>   + <https://github.com/raml-org/raml-spec/blob/master/versions/raml-08/raml-08.md> |
| Schemas | * Use schemas in the specification to determine the format of the requests and responses. (RAML 0.8) * Separate the schemas from the base RAML file. |
| Examples | * Always include examples * Separate the examples from the base RAML file |
| DataTypes | * Use Data Types when possible (RAML 1.0) * Separate the dataTypes from the base RAML file |
| Traits | * Use traits to define common method properties such as query-parameters and responses. * Separate the traits from the base RAML file |

**Mulesoft Coding Best Practices**

In this tutorial we have listed down some best practices that every mule developer should follow

* Avoid hardcoding values,hosts,urls and port number inside the mule code, try to use the properties files and refer the values from properties file inside mule code
* Avoid hardcoding the passwords in code, all the passwords should be encrypted use the secure property placeholder to manage the passwords
* Avoid logging the complete payload, it impacts the performance and security by logging the sensitive information
* User loggers with proper log level inside mule flows
* While logging the payload. Log only required information and mask the sensitive data
* Naming convention should be proper for system,process and experience API’s for mule projects and for flows and subflows, variables so that everyone can differentiate between all components
* Try to use common logging and error handling framework and use it across all mule application to maintain the consistency
* All possible runtime exception should handle correctly in mule code
* Response code should be mapped correctly as per HTTP standard
* MUnit should be written for all the mule flows and test coverage should be greater than 80 %
* Try to do the validation on fields in well advance at the starting of the mule flows
* It is recommended to use the latest mulesoft connector version in pom.xml
* Avoid duplicacy inside mule code, try to wrap the reusable code in subflow/flow and use across mule application
* There should not be any unused variable/code inside mule project
* All the dependent 3rd party library should be added as dependency under pom.xml
* Should not load large file or payload in to memory
* Avoid creating copy of the payload and store in variable
* Reprocessing connection strategy should be implemented in case of technical fault