MuleSoft Design Best Practices

DRAFT

Table of Contents

1. Overview

This document addresses the overall approach to be followed while architecting and designing MuleSoft projects. It is recommended to create a project specific document that references this document and to override/add any additional project specific details.

This document has the following sections, which are critical while designing a MuleSoft project.

1.1. Design Considerations

For any successful project implementation, architecture and design is the critical phase. Ensuring that all important aspects are covered while designing a project is the key for success. Mule projects design is based out of following keys aspects:-

- Use cases
- API Design Considerations
- Requirements: Both functional and non-functional
- Application type
- Infrastructure

1.2. Integration Style

MuleSoft project implementations would mostly fall under the below high level integration styles. The first 4 are traditional integration approaches, while the last 4 are mostly for micro-service based implementations.

- Migration
- Broadcast
- Aggregation
- Bi-directional synchronization
- Proxy
- Chained services
- Branch
- Asynchronous messaging

1.3. Demand and Capacity Planning

Integration demand and capacity planning is critical due to continuous demand for new integrations from various business units, along incremental increase in number of transactions processed every year. Hence setting the right estimation of what is required today and in the future in terms of platform capabilities and capacity is important.

1.4. High Availability and Reliability

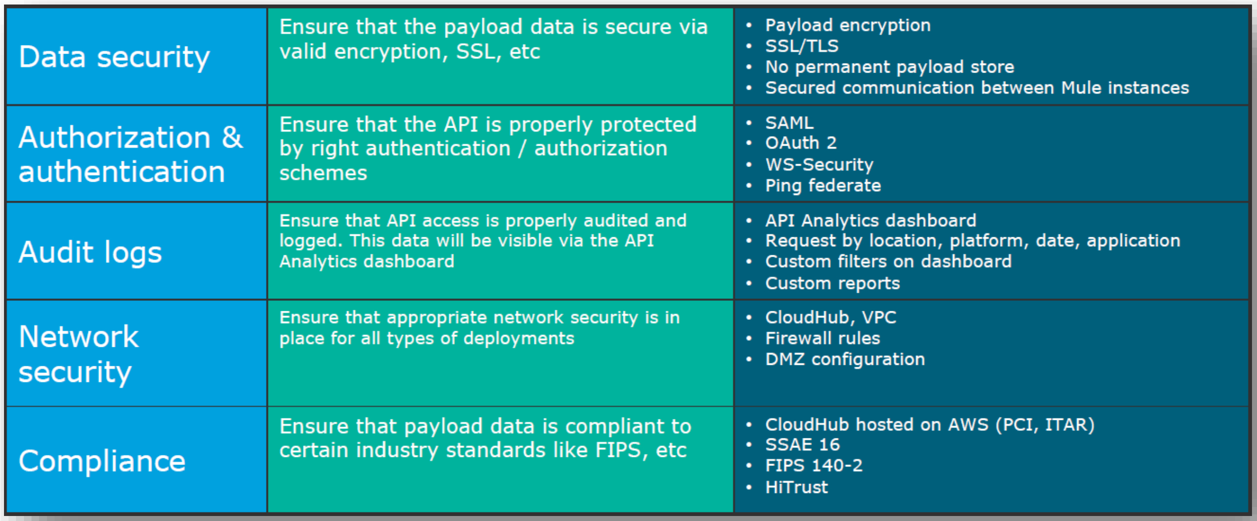
High availability and reliability options need to be considered while providing architecture for a MuleSoft project. The options vary depending on the type of implementation – on premise or Cloud.

1.5. Performance Optimization

In order to implement a performance optimum MuleSoft project, certain design guidelines have to be followed while designing the project.

1.6. Security

The integrations have to be secured. The security approach usually needs to be discussed with client before coming up with the overall design. The different security approach principles in Mule projects will fall under the below 5 categories.



Data security	Ensure that the payload data is secure via valid encryption, SSL, etc	• Payload encryption • SSL/TLS • No permanent payload store • Secured communication between Mule instances
Authorization & authentication	Ensure that the API is properly protected by right authentication / authorization schemes	• SAML • OAuth 2 • WS-Security • Ping federate
Audit logs	Ensure that API access is properly audited and logged. This data will be visible via the API Analytics dashboard	• API Analytics dashboard • Request by location, platform, date, application • Custom filters on dashboard • Custom reports
Network security	Ensure that appropriate network security is in place for all types of deployments	• CloudHub, VPC • Firewall rules • DMZ configuration
Compliance	Ensure that payload data is compliant to certain industry standards like FIPS, etc	• CloudHub hosted on AWS (PCI, ITAR) • SSAE 16 • FIPS 140-2 • HiTrust

1.7. MuleSoft Preferred SDLC Approach

For successful end to end project execution, MuleSoft preferred usage of components is recommended at every phase of the SDLC – Design, Development, Testing, Deployment and Maintenance. This methodology is called as API first approach / Top Down implementation.

2. Design Considerations

The design considerations aspects are detailed as below:-

2.1. Use cases

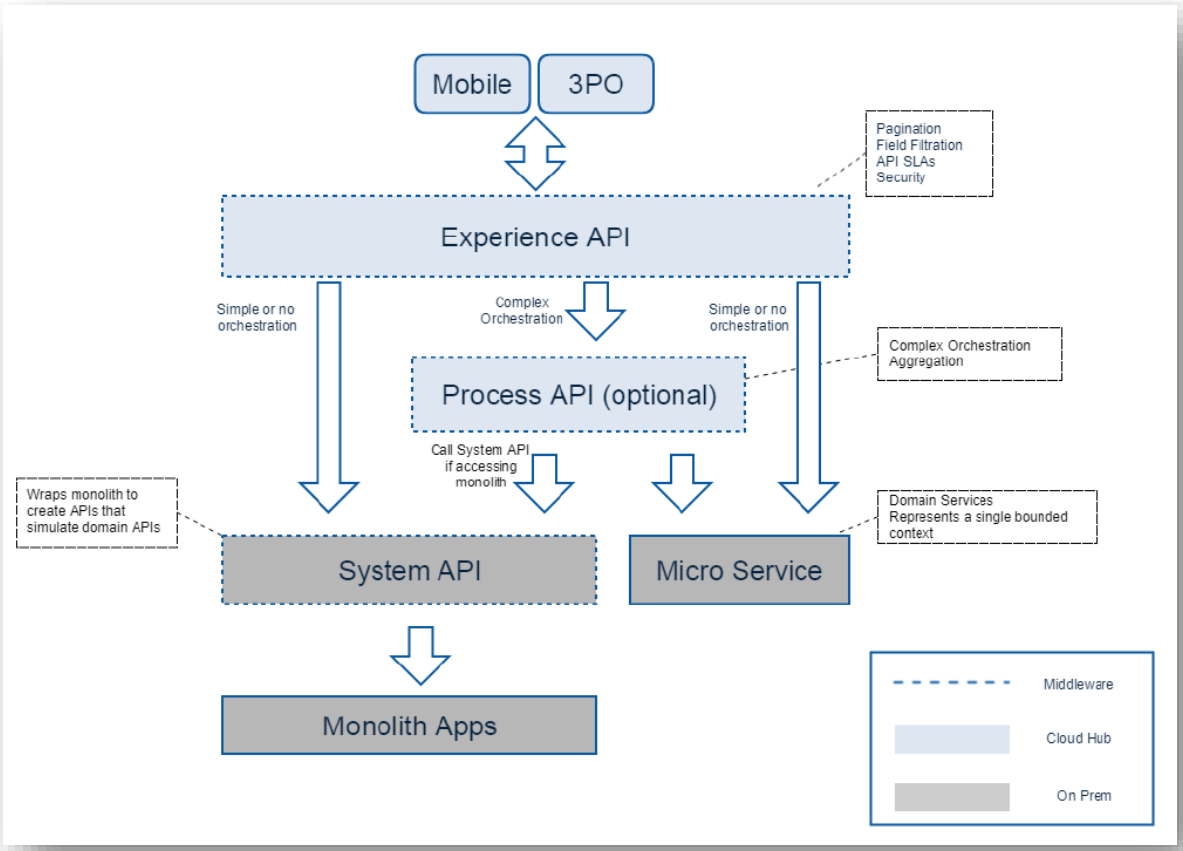
The use cases for Cloud solutions are classified as below:-

- Hosting cross-cloud integration applications
- Hosting hybrid applications
- Hosting public APIs/REST services/web services
- Hosting Software as a Service (SaaS) applications
- Hosting public web applications
- B2B scenarios

2.2. API Design Considerations

A layered approach to APIs is recommended, to clearly segregate responsibilities on each layer. The enterprise architecture to be comprised of the following layers for API.

- **Experience API**: These should be the only public APIs in the entire layer of APIs, which all channels would directly access. They would largely be channel agnostic but will build flexibility to allow channel developers to make them channel specific. This layer exists to provide
    - Abstraction of different functions from an API client perspective, allowing for different configurable attributes for pagination, field filtration
    - Since this is the only layer the API client interacts with, in the API layered architecture, all responsibilities that relate to a gate-keeper function will be done at this layer - such as SLA checks (rate limiting or throttling), security checks, monetization and the like.
- **Process API**: This is an optional layer. Where complex orchestration/aggregation is required resulting in managing multiple calls to different domain APIs (System or Microservices), a Process API would be needed. If the case for a complex orchestration or aggregation doesn't exist, Experience APIs will communicate directly with System APIs. This layer is needed to perform complex facade functions of orchestration and the associated operations leading to errors, failovers, alternate paths and the like including transaction management where applicable.
- **System API**: This layer only exists to provide a micro service-like-facade on a monolith application not yet broken into micro services. The system APIs provide a micro service abstraction and help create bounded contexts in order to access functions within a monolith application, using modern architectural paradigms of single bounded context and self-contained APIs.



Experience layer guidelines

- API Identification
  - APIs should be designed to be channel agnostic, and should be designed to support multiple channels. The design should optimize data to conform to a specific user experience across channels, and should be able to support dynamic inbound and outbound payloads per channel.
  - APIs should be modeled by not aligning to screen design, but should focus on details needed to complete actions pertaining to use cases across screens/channels. This enables screens to be changed with minimal changes to the API interface.
- API Design
  - APIs should be designed to be non-blocking, with a decision made on sync or non-sync flows to be implemented.
  - Data considerations needed to support the business needs from an experience point of view should include – inbound and outbound data models, required and non-required fields.
  - Process or System API responses should be mapped Experience API responses (payload reduction) in order to segregate clearly the responsibilities per layer.
  - APIs should be designed to support Pagination and navigation (we recommend Hateoas)
  - API payloads should use Compression (gzip used by default at ios and android dev kits)

    - APIs should offer dynamic Data Filtration and transformations where applicable.
    - This is also the layer where non-functional aspects such as the following should be handled – rate limiting/throttling, Analytics, Security / Authentication and Authorization (JWT/ OAuth2 security schemas in RAML).
    - APIs should be designed to support multiple versions.
- API Implementation
    - Experience APIs should be documented to make it easier for API developers to consume and contribute to them.
    - Mocks for APIs should be created along with MUnits in a TDD paradigm, in order to ensure clients to the APIs can develop to the specification, while the API is being developed.
    - Experience APIs should externalize configuration where applicable for different markets to customize them without needing to perform changes to API implementations.
- API Operations
    - Experience APIs should be monitored and controlled for use per channel.
    - Experience APIs usage should directly correlate to monetization decisions.

Process layer guidelines

- API Identification – Process API identification should include orchestration needs that support aggregation/orchestration, conditional routing and filtering of information for functional and non-functional requirements.
- API Design
    - APIs should be designed to be non-blocking, with a decision made on sync or non-sync flows to be implemented.
    - APIs in this layer will need to understand the domains exposed by system/domain layer before aggregating/orchestrating.
    - While accessing system layer APIs, this layer will perform tasks related with data aggregation (split-join), conditional routing (based on parameters from experience layer/global parameters) and filtering.
    - APIs will need to be designed to handle non-functional aspects such as transaction management (where needed), conditional routing, business events.
    - A process API may call other process APIs if needed or multiple System API/micro services.
    - Process APIs should represent re-usable aggregations/facade for APIs within the eco-system and they should help define common processes to be consumed by Experience layer.
    - Interaction with Business Analysis teams will be needed in order to understand which business processes/rules are need to be modeled in this layer
- API Implementation
    - Process APIs should be documented to make it easier for Experience API developers to consume and contribute to them.
    - Mocks for APIs should be created alongwith MUnits in a TDD paradigm, in order to ensure clients to the APIs can develop to the specification, while the API is being developed.
- API Operations
    - Process APIs should be monitored to ensure low level issues such as the below are highlighted and corresponding actions initiated – circuit open and circuit close, number

    - of requests flowing through individual flow paths to understand patterns of usage of API, errors/exceptions with highlights to occurrences at specific as well as trends on the different types of exceptions/errors.
    - Should be auto scaled and support other hooks to enable scaling the APIs up or down based on usage.

<u>System API layer guidelines</u>

- <u>API Identification</u> – The micro services identification should be based on MCD Capabilities and Sub-capabilities.
- <u>API Design</u>
    - The micro service should expose APIs that pertain to the unique bounded context as represented by the core business entity that the micro service represents.
    - The APIs should support functional, as well as operational requirements such as Bulk Data needs to feed the data warehouse and the like.
    - The APIs should support multiple protocols such as REST, Streaming etc where applicable for consuming different information from third parties.
    - The API should abstract the complexity of interacting with legacy / core back end services, creating a coherent API context.
    - The path to getting to a micro service for an existing monolithic application involves the following steps – System APIs should abstract the monolithic application and provide a facade following domain driven design principles, Monolith application should be iteratively broken into micro service APIs, and the corresponding System APIs for them should be deprecated and retired, Over time, the APIs from micro services should completely replace the System APIs.
    - A System API should be called from either a Process API or an Experience API. A System API interacting with another System API should be an exception and should be closely analyzed to see – if the principles applied w.r.t domain driven design, can be re-looked at, to avoid the interaction, if business events (choreography) or orchestration at the Process API level can help avoid the interaction.
    - Design for APIs should allow for time based access where needed.
- <u>API Implementation</u>
    - The APIs abstract access to third parties, data sources that pertain to the bounded context The APIs may interact with underlying IT architectures and core systems, normally not easily accessible due to connectivity or security concerns
    - The APIs will need to account for functional as well as non-functional aspects such as transaction management, business events, and failovers.
    - The APIs should expose a Bulk Data /Streaming / Events access to the other APIs for accessing the data for operational analytics/personalization and the like.
- <u>API Operations</u>
    - Systems APIs/Micro services should be monitored for availability, errors, and resilience.
    - Handoffs to other systems and subsequent follow through should be monitored to ensure complete business orchestration/choreography has been achieved
    - Close attention should be paid to business rule execution and failures and the tweaking needed, data integrity and the reasons for failure, in order to quickly address the root cause of failures where needed.

2.3. Requirements

The requirements will be ranging from Functional and Non-Functional. From functional requirements perspective, it is the FSPEC which details out the functionality required to achieve as part of the Integration. In case of Non-functional Requirements (NFRs), the following aspects need to be understood well.

- SLAs – Service Level Agreements
- TPS – Transactions per second to be processed by the interface / API
- Response Time – The expected response times for real-time APIs
- Size of Payload – The expected size of payload to be processed
- HA – High Availability need to be considered
- DA – Any Disaster Recovery methods to be in place
- Environments – How many environments need to be accounted (DEV, QA, PROD … etc.)

2.4. Application Type

The application types which can be implemented would be:-

- <u>Cloud only</u>: In this type of applications, use only Cloudhub
- <u>On-Premise</u>: Traditional enterprise integration applications
- <u>Hybrid</u>: This is a common approach we see for most of the implementations, where we do both Cloud and On-Premise implementations.

2.5. Infrastructure

The infrastructure needs to be considered depending on the application types.

- <u>Cloudhub</u>: The infrastructure will be managed by MuleSoft and the back end is on AWS.
- <u>On-Premise</u>: The customer has to manage the servers, load balancers, clusters, firewalls … etc.
- <u>Hybrid</u>: Along with the above Cloudhub and On-Premise infrastructure elements, MuleSoft managed VPC tunnel where MuleSoft configures the VPC between Cloudhub and Customer on-premise network.

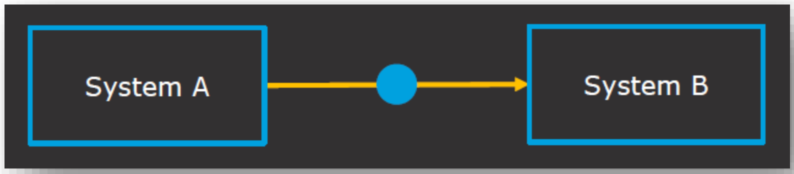
3. Integration Style

The Integration styles are explained in detailed below for some examples from traditional integrations and micro-service based approaches.

3.1. Migration

The most common style of Integration we see is Migration, as most of the Integrations are moving Cloud based and we look at migrating on-premise integrations.

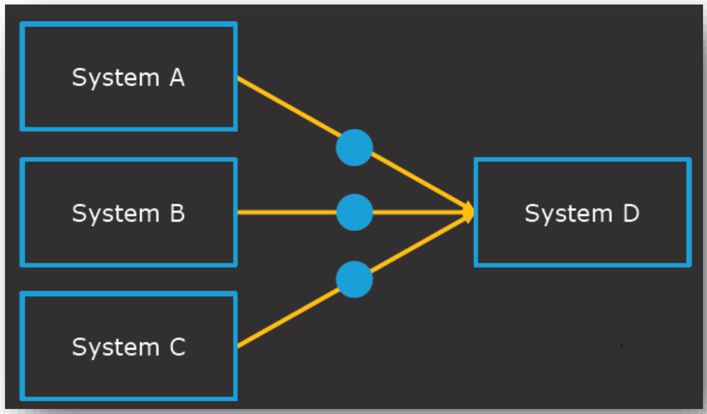
- Data migration is moving a specific set of data at a particular point in time from one system to another.
- Migration pattern allows developers to build automated migration services that create functionality to be shared across numerous teams in an organization.
- Implementation of the Migration can be record-by-record or in batch depending on volume.



3.2. Aggregation

The interface usually takes or receives data from multiple systems and copies or moves it into just one system. This kind of approach enables the extraction and processing of data from multiple systems into one application.

- Ensures that data is always up to date, does not get replicated
- Can be processed or merged to produce any desired dataset or report

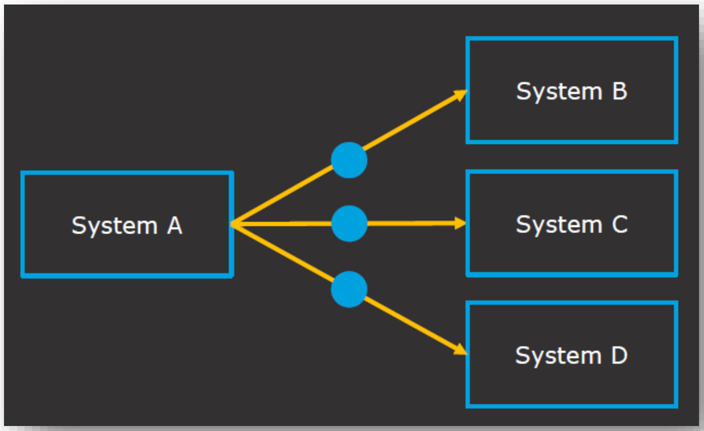


3.3. Broadcast

We can say this is the opposite style of Integration when compared to Aggregation pattern, as the data transferred from a single source to multiple destination systems.

This approach is most common use case is keeping data up-to-date between multiple systems, which usually implemented as one-way synchronization from one to many systems.

- Ensures that data is always up to date, does not get replicated
- Can be processed or merged to produce any desired dataset or report

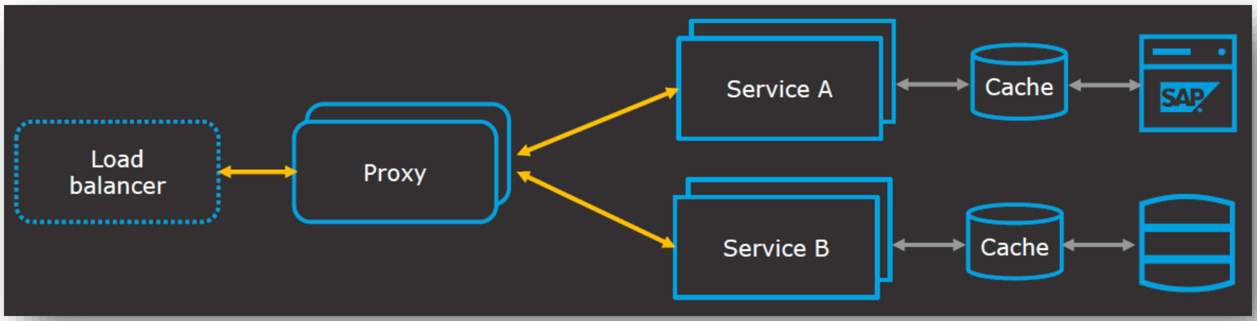


We look at Microservices style of integrations usually to achieve Functional decomposition; Domain based segregation of services which allows low coupling and high cohesion. The following are the same:-

3.4. Proxy

Proxy patterns usually come in implementations, where client will not invoke the actual service directly. The most common use-case would be each individual service does not need be exposed to consumer and should instead go through an interface.

In the below depicted pattern, the request are coming from load balancer to the proxy service and the proxy service is fetching data from target services A and B. The proxy service is just acting as routing service for requests to Service A or Service B in the backend.

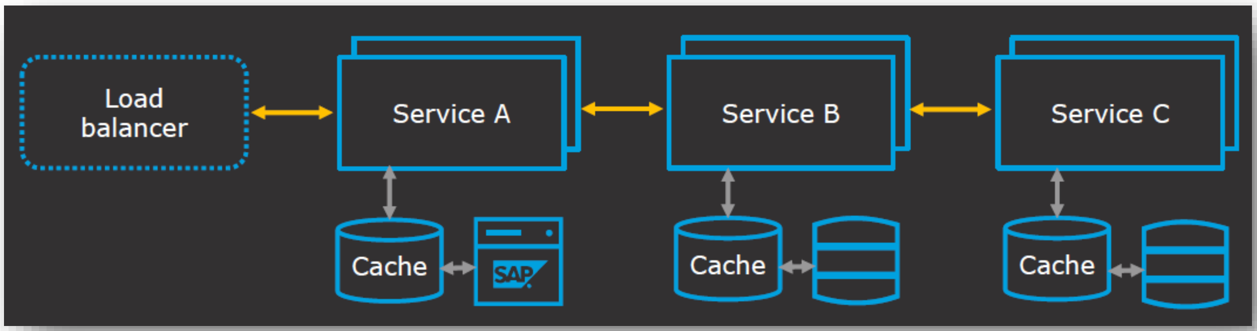


3.5. Chained Services

This type of integration is used when it is required to produce a single consolidated response to the request. The services will follow the below characteristics.

- All services must use synchronous request/response messages, such as HTTP

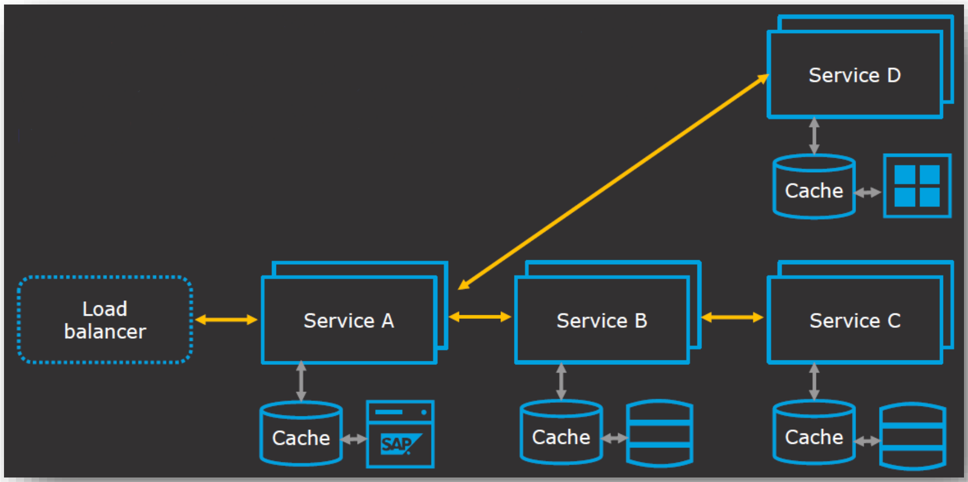
- Client is blocked until the complete chain of request/response is completed



3.6. Branch

This type of Integration style usually extends the Aggregator type, and is used in situations:-

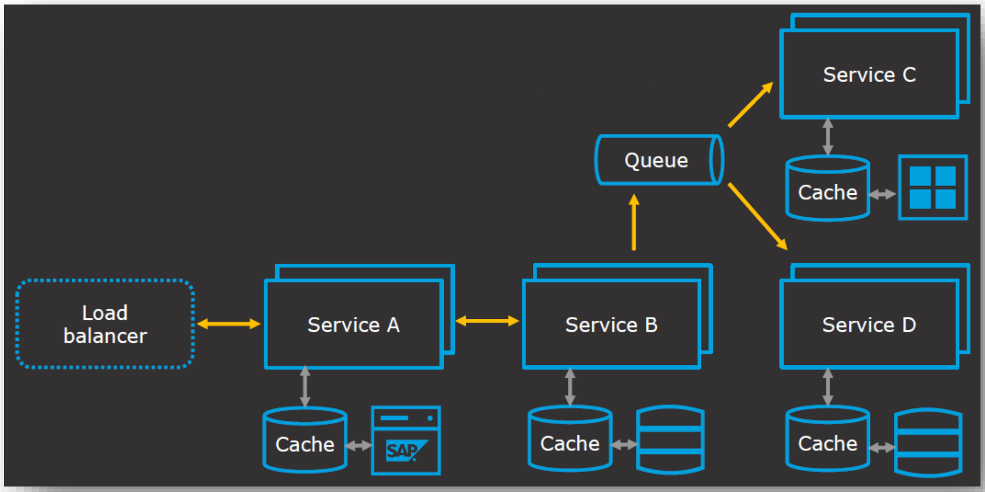
- To allow simultaneous response processing from two, likely mutually exclusive, chains of Microservices
- Can also be used to call different chains, or a single chain, based upon the business needs



3.7. Asynchronous Messaging

Usually REST design patterns are synchronous, and hence the request thread is blocked until the response is received from the service. In situations where synchronous behavior is not required and near real-time responses are accepted, we can look into Asynchronous messaging style.

The services will be de-coupled using message queues or web sockets, instead of REST.



4. Demand and Capacity Planning

Demand planning is an important aspect while designing MuleSoft projects / solutions. This activity is usually done during sales cycle or any new opportunities. This activity can also be done while client is evaluating any changes in existing projects too.

This metric would give estimation of what is needed today and in the future in terms of platform capabilities and capacity. The below are the high level capabilities considered during this phase.

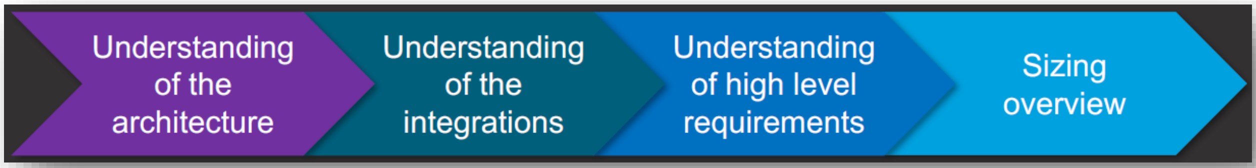
- Cloud / on-premises / hybrid integrations
- API Management
- Cloud messaging service
- B2B Partner Management
- Platinum support
- Type of connectors required. For example SAP, Salesforce … etc.

When we are taking about capacity planning, the following aspects are what we need to account for.

- Number of Cores
- Number of B2B partners
- Number of MQ messages
- Number of Environments (Production and Sandboxes)

It is very important to consider different phases or project rollouts, while doing capacity planning. The reason being, while phase 1 is being rolled out, phase 2 development activities might start and need to account for additional environments.

The outcome of capacity planning would be after having clear understanding of the following sequence of steps. The recommendation will be given along with discussion with MuleSoft Engagement Manager.


Understanding of the architecture
Understanding of the integrations
Understanding of high level requirements
Sizing overview

- Understanding of Architecture: The details around network setup, reference integrating applications and format of messages need to be understood.
- Understanding of Integrations: Each type of integration can have different use-cases, and each use-case will have different requirements. Need to understand various aspects of integrations, like real-time vs batch, complexity, peak-volume … etc.
- Understanding Requirements: This is as mentioned in requirements section.
- Sizing Overview: The sizing overview step will give a picture about required capacity and the list of recommendations. For example for Cloudhub based development, the number of cores will be provided. This will be output of discussion along with MuleSoft Engagement Manager. While the requirements change, the sizing needs to be re-assessed again.

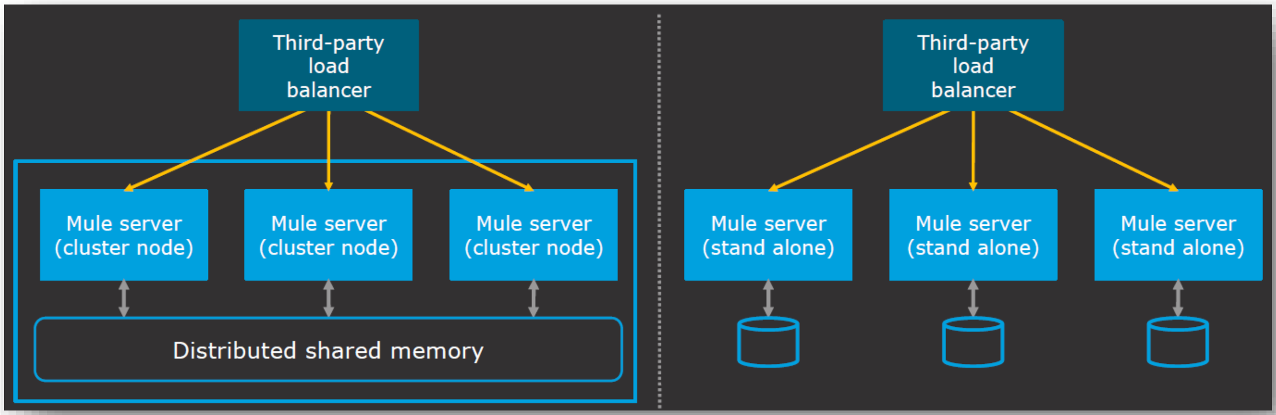
5. High Availability and Reliability

The options will vary depending on the application type Cloud or On-premise. One important point to understand is that High Availability and Reliability are two different concepts. High availability is about up-time of your application, whereas reliability is about zero message / data loss. The details are captured below for both Cloud and On-premise.

5.1. On-premise

High availability can be achieved by below two models for on-premise MuleSoft implementations.

- Mule Clustering – Where multiple Mule servers are available within the same cluster environment and the routing of requests will be done by the load balancer.
- Load balanced standalone Mule instances – The high availability can be achieved even without cluster, with the usage of only load balancer pointing requests to different Mule servers.


Third-party load balancer
Mule server (cluster node)
Mule server (cluster node)
Mule server (cluster node)
Distributed shared memory
Third-party load balancer
Mule server (stand alone)
Mule server (stand alone)
Mule server (stand alone)

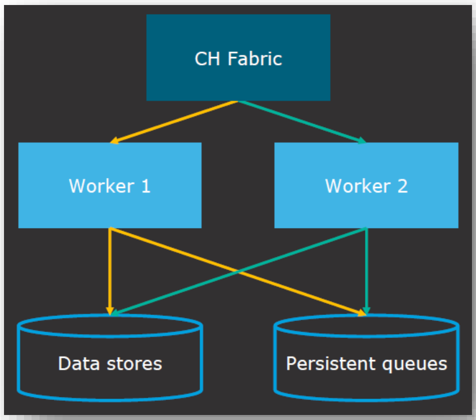
Reliability will depend on the components used for On-premise implementations. This is relevant to the availability of data and messages, specifically in case of errors states, system failures etc. When you consider introducing messaging layer for reliability, you need to choose from below:-

- JMS transport will provide optimal reliability
- VM transport will provide optimal performance

5.2. Cloudhub

High availability can be achieved for Cloudhub applications by utilizing Cloudhub Fabric, where same application can be deployed to multiple workers which is usually called as horizontal scaling.

Cloudhub Fabric will have an inbuilt load balancer provided by MuleSoft, which routes the requests to horizontally scaled applications.



In order to achieve reliability on Cloudhub, the options available are as below:-

- Object Store – For persistent key value pair based data storage, which can be accessed by multiple workers.
- Persistent Queues – To avoid any message loss and achieve reliability.

6. Performance Optimization

When performance optimization is being done for an application, the following goals have to be considered. Usually these goals will be opposing each other.

- High throughput
- Low latency
- Reliability
- Security

The design recommendations from MuleSoft for optimal performance are listed below.

- Minimal or no usage of session variables need to be considered. The larger or more session variables will impact the performance of the application.
- Java payloads usually give best possible performance. Try to avoid large XML payloads
- Preferably use MEL over scripting languages like Groovy
- Dataweave need to be used over one to one transformers. However Dataweave need to be avoided if possible in Batch jobs.

6.1. High Throughput

Throughput is usually measured in TPS (Transactions per second). In order to maximize the throughput, several aspects like thread pools, cache, connection pooling … etc. can be used. Sometimes while achieving high-throughput, latency or response times will be a trade off.

While configuring thread pools, selecting the right processing strategy of the flow is also important. While calculating the number of threads, considering a 1 vCore we need to keep 2000 as the maximum allowed threads.

6.2. Low Latency

For achieving low latency or minimal response times, the following aspects can be considered.

- Processing in parallel, with usage of Scatter-Gather component as an example
- Try to avoid transaction scopes, persistent queues, object stores …etc.

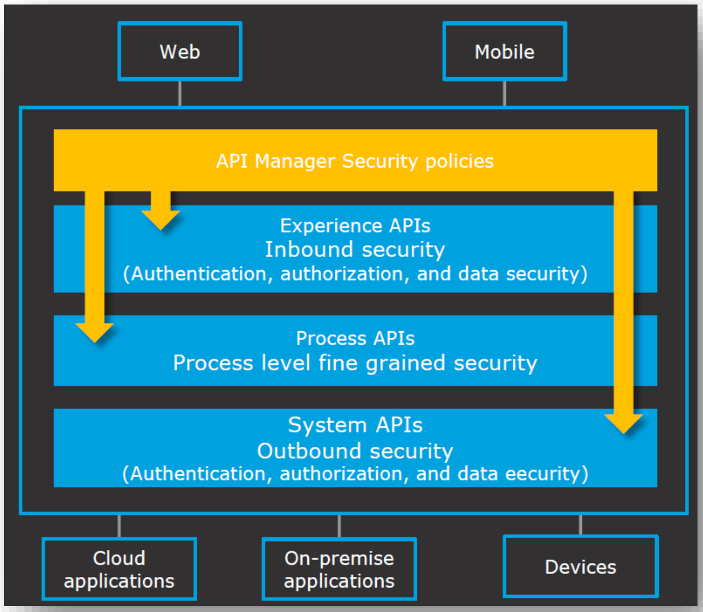
The other aspects of Reliability and Security are covered in the other individual sections.

7. Security

There are several aspects of security need to be considered while designing Mule applications.

7.1. API Security

The 3 layered API approach gives what level of security need to be incorporated as shown below.


Web
Mobile
API Manager Security policies
Experience APIs
Inbound security
(Authentication, authorization, and data security)
Process APIs
Process level fine grained security
System APIs
Outbound security
(Authentication, authorization, and data eecurity)
Cloud applications
On-premise applications
Devices

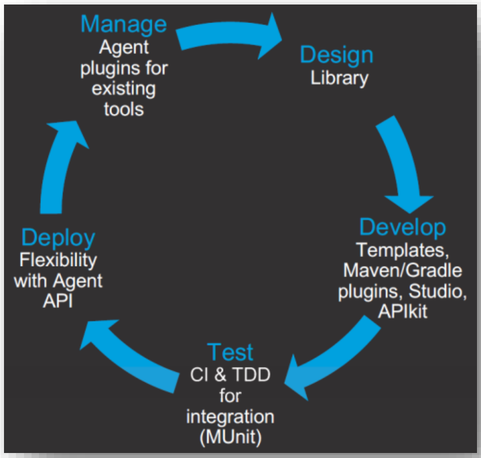
7.2. Anypoint Platform Security Components

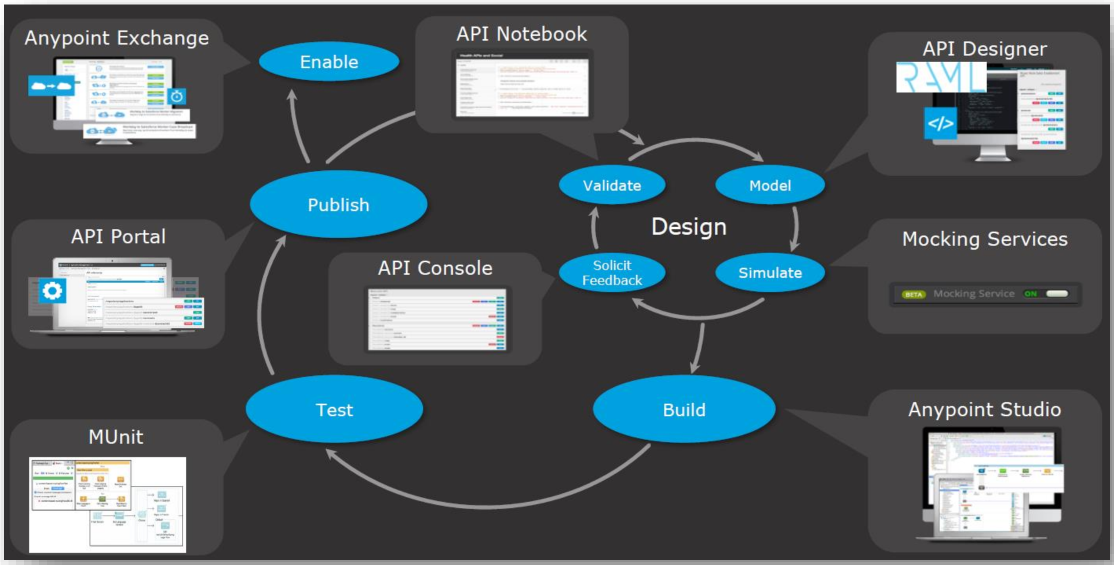
Anypoint platform provided security components consists of below 3 aspects

- Anypoint Enterprise Security – This module can be incorporated in both Cloud and On-premise based applications. It has capabilities around Filtering IPs, Securing passwords in property files, Message level encryption such as PGP … etc. It also provides OAUTH 2.0 capability and processors to ensure message integrity.
- API Security Manager – Securing the APIs using various API manager policies and custom policies
- Virtual Private Cloud (VPC) – Securing the Cloud / On-premise network, using options such as VPC peering, VPN tunnel and Cloudhub direct connect.

8. MuleSoft Preferred SDLC Approach

MuleSoft recommended tools and framework need to be used at each phase of SDLC as listed below, along with the recommended API first approach.


Manage
Agent plugins for existing tools
Design
Library
Develop
Templates, Maven/Gradle plugins, Studio, APIkit
Test
CI & TDD for integration (MUnit)
Deploy
Flexibility with Agent API



8.1. Design

For designing MuleSoft APIs / Interfaces, the below MuleSoft components can be used.

- API Designer
- Anypoint Exchange

8.2. Development

For developing MuleSoft projects, utilities required are as below:-

- Anypoint Studio
- Maven / Gradle plugins

- DevKit and APIkit for developing custom connectors / APIs respectively.

8.3. Testing

For unit testing Mule applications the following components are required.

- MUnit, Maven plugin

8.4. Deployment

After developing and unit testing, the applications can be deployed as Mule runtime agent application or as an API.

- Agent, APIs, CLI

8.5. Maintenance

In order to manage / maintain the deployed Mule applications, we can use MMC for on-premise and API Manager / Runtime manager from Cloudhub perspective.

- Agent, APIs, MMC

9. References

The following references are considered while writing this document.

#	Reference Link
1	https://training.mulesoft.com/instructor-led-training/aparch-solution-design
2	https://www.MuleSoftsoft.com/resources/esb/cloud-integration-patterns
3	https://docs.MuleSoftsoft.com/MuleSoft-user-guide/v/3.9/understanding-enterprise-integration-patterns-using-MuleSoft
4	https://www.mulesoft.com/ty/wp/secrets-great-api