MS3

www.ms3-inc.com

MAR.
2019

Deep Down into API-Led Architecture

Bahman Kalali

MOUNTAIN STATE SOFTWARE SOLUTIONS

WHO ARE WE?

MS^3 is a Global IT consulting firm focused on business acceleration and engineering future proof software solutions. We are a leading provider for enterprise-ready mission-critical solutions for commercial and government customers. With our team of experienced engineers, we provide globally distributed organizations the ability to meet today's most complex integration challenges. Learn more about us at **https://www.ms3-inc.com**

WHAT DO WE DO?

- **API INTEGRATION**
- **DEVOPS & CICD**
- **RPA**
- **BIG DATA**
- **IoT**
- **MULESOFT**
- **24x7 SUPPORT**
- **ANGULAR**

Deep Down into API-Led Architecture

THE AUTHOR
BAHMAN KALALI

Bahman Kalali is a Principal Mulesoft Architect with Mountain State Software Solution (MS^3). He is a seasoned professional consultant with 18 years of experience in Information Technology, specifically software development and integration. He has contributed to the success of many software development and integration projects as a team lead, senior architect and engineer. He started his professional software development and integration career in 2000. After 11 years of development and gaining experience in Java, JEE, Spring, API, SOA, distributed computing, application and integration, he began focusing on using the MuleSoft platform. Kalali graduated from the University of Waterloo with a master's degree in computer science specializing in software engineering and Web services. He also earned a bachelor's degree in computer science from Concordia University. His education has given him a broad base allowing him to implement reliable, scalable, maintainable, sustainable and cost-effective solutions for many customers. Kalali has provided value added consultancy services to many organizations in the U.S.A and Canada.

ABSTRACT

API-Led connectivity is an architectural style to connect data to applications through reusable and purposeful APIs. These are developed to play a specific role in a multi-layer environment. However, since API-Led is an architectural style, it can be designed and implemented in a variety of ways. This document will address some of the questions which arise when adopting API-Led connectivity. In specific, the following questions will be addressed.

> What is the responsibility of the APIs in each layer?
> What is the core functionality of the APIs in each layer?
> Do I need APIs on all three layers?
> What naming convention should I use for APIs?
> Should APIs follow microservice design paradigms?
> What is the right strategy in designing System APIs?
> Where should I use Proxy?
> Do I need the Process API?
> Do I need the User Experience API?
> Do I need the System API?

This document will provide high level roles and responsibilities for APIs in each layer and will describe the core functionalities that have to be designed. Along with those roles and responsibilities, the document will highlight some industry best practices for designing API specifications and implementations. Next, this document will provide some variants of API-Led patterns, which can be applied when adopting API-Led connectivity.

This document does not cover discussions around system architecture or physical architecture such as load balancer, runtimes, cloudhub vs on premise solutions, API manager or applying policies to APIs. This paper can be used as a decision making tool where the standard three layer API-Led approach is not applicable. There are use cases where creating unnecessary APIs can bring maintainability, sustainability, scalability, reliability and overall performance issues. This document is written solely based on my personal experiences reviewing the design and architecture of clients who wanted to change their original architecture, or who applied one of the patterns from this paper, to address those non-functional requirement concerns.

System APIs

System APIs are responsible for exposing data from their underlying system of records such as database, COBOL, RPG functions from AS400, Salesforce, Workday, etc., to upper layer APIs in a secure and reliable way. A System APIs main core functionality can be outlined as:

Expose Rest API endpoints to upper layer consumers by implementing well-defined API specification.

Transport connectors such as database, JMS, SFDC, Workday, Web Service Consumer and many other similar connectors are used to perform CRUD operations on system of records.

Transform data from target system of records are transformed into an acceptable model and bounded context model (e.g. JSON).

Data clean and convert raw data from system of records are cleaned (e.g. extra space from a string value being removed) and converted (e.g. dates are represented in the backend system as unix epoch and are converted to the local date format).

Error handle mapping errors received by underlying system of records.

Handle transaction issues where underlying systems are transactional such as databases.

Secure and encrypt PII data if System API is responsible for transfering data outside of an organization. As a general guideline, System API should follow non-functional requirements defined by the organization's security department.

Strategy in Developing Business Logic in a System API

In most cases, it is not the responsibility of System APIs to handle heavy business logic implementations. Typically such responsibility is off loaded to other downstream systems (e.g. rule engines, store procedures, Spring boot applications, etc.), which are better suited for developing business logic, or Process APIs. In most cases, MuleSoft is used as an integration and API platform where underlying downstream subsystems are responsible for exposing business logic. However, MuleSoft has all the necessary tools and functionality to implement heavy business logic as well. The decision to implement business logic in a System API is mostly logistics and the following questions should be answered first.

Do MuleSoft developers have enough knowledge of business requirements to develop business logic?

Can maintainability, sustainability and extendability of code, which are written in a System API, be preserved when business requirements change?

Is there any plan to retire other deployment platforms (e.g. Tomcat, WebSphere) and only use MuleSoft platform in the future?

Does the developer have an architectural strategy, provided to development teams, for writing reusable business components?

Does the developer have a proper SDLC which has been adopted for developing business logic on the MuleSoft platform?

Strategy in Designing a System API Implementation

System APIs can be designed based on these principles:

- Domain Driven Design (DDD)
- Business Capability
- System of Records (SR)

Domain Driven Design (DDD) and Business Capability are strategies which use the microservices design approach, whereas the SR approach is adopted and advocated by MuleSoft. These approaches are not mutually exclusive and they need to be augmented together to produce a solid architecture. The microservice approach requires one to have DDD techniques or to have a knowledge of business capabilities. System APIs should follow the Single Responsibility Principle (SRP) and the Common Closure Principle (CCP), should be cohesive, and should not be coupled to other System APIs.

From the system of records perspective, a System API should be designed to wrap around one subsystem (e.g. one database, one Salesforce instance, one legacy API). One way of checking if a System API is designed in the right direction is to verify whether the System API has used one or more connectors. For example, a System API should not use more than one JDBC connector, one SFDC connector, one WorkDay connector, or any other connectors that require data to be transformed from a system of record to a System API domain context and vice versa.

DDD calls for the scope of a domain model, or a “bounded context”. When a System API is designed based on a microservice approach, then each System API should have its own domain model.

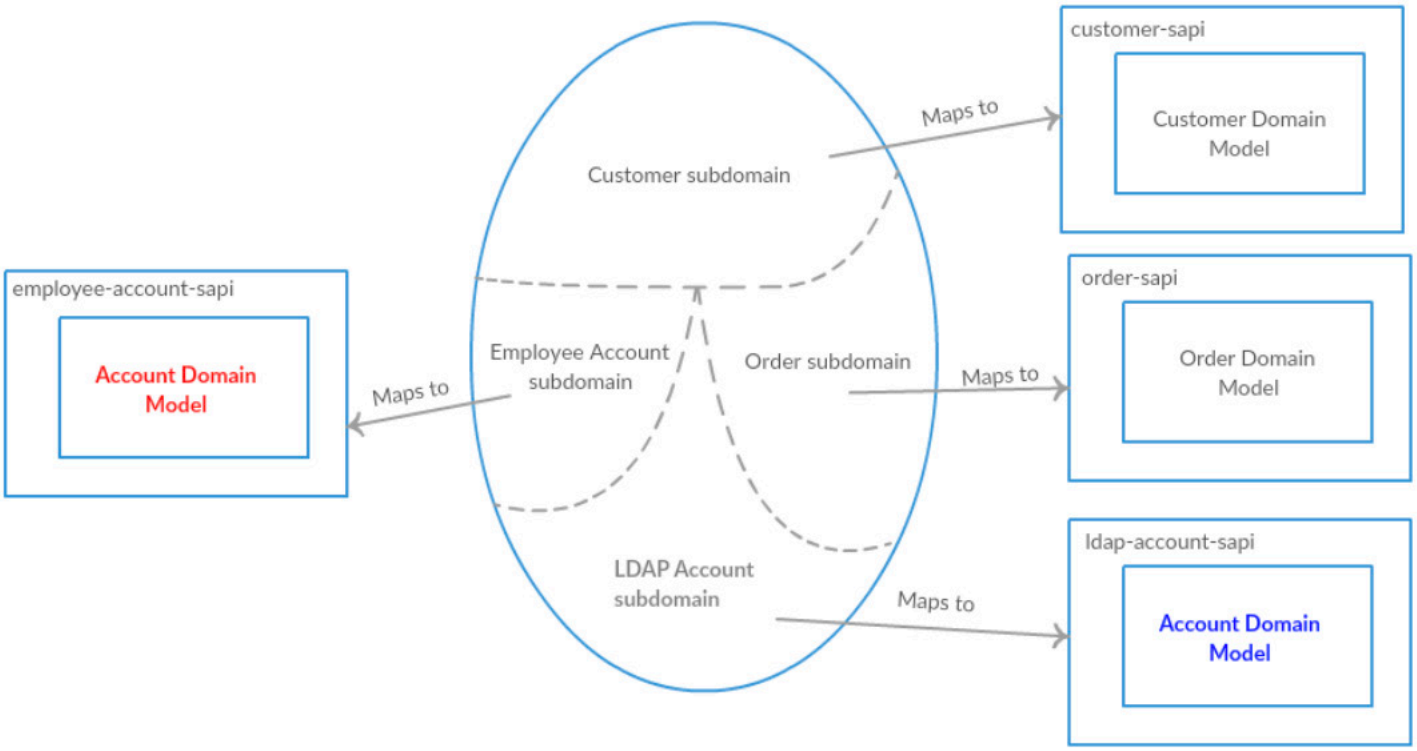


Diagram 1.0

Diagram 1.0 shows how the DDD is applied in designing system APIs. Here the account domain model is different in ldap-account-sapi (assume system of records in LDAP) than in the account model for employee-account-sapi (assume system of records are on salesforce).

Let's look at other scenarios: First, let's assume the schema of a database is very monolithic (e.g. the schema has DDL for both accounting and employee subdomains). There should be multiple System APIs, even though there is only one database schema and one physical database. Breaking down to multiple System APIs, where each one wraps a subdomain, is a good design decision when dealing with legacy systems.

Another scenario: let's say we are going to design System API(s) over a very coarse grained SOAP web service. In this case, a proper design decision is first to logically group SOAP operations together and design a system API over each logically grouped SOAP operation. Each group of operations should represent a business capability or subdomain. Each group should be designed to be highly cohesive with low coupling with any other System APIs. Therefore, a very coarse grained SOAP web service can be broken down to multiple independent, atomic and cohesive System APIs.

Strategy in Adopting a Naming Convention

As a general guideline, a naming convention should follow the strategy of designing the System API and the underlying backend system, subdomain or both. If only DDD is used, then for a subdomain of an employee acount, the name employee-account-sapi is appropriate. However, if SR is the only design principle more coarse grained naming can be adopted. For example, if a System API is going to expose data from the specific schema of a database (e.g. ART), then art-sapi is a proper name. If there are multiple similar subdomain names, then it is better to pick up a naming scheme which presents the backend subsystem name plus a subdomain name (e.g. **ldap-account-sapi**, **slack-account-sapi**). As a general principle, names such as db2-sapi, as400-sapi, or db2-req-res-sapi should not be used.

Strategy in Designing a System API Specification

When designing API specifications for a System API, it is important to consider that System APIs are the most reusable assets. Usability and reusability are two important factors when designing API specifications for a System API. First a System API has to be usable; that is, it has to be driven by some kind of business requirement to bring values for developing it. Second, that System API should be reusable in other contexts, not just for the current requirement. For those reasons, API specifications should be backward compatible as much as possible. The general rule of thumb is to follow open/close design principles.

For example, after a few iterations of RAML design, where API specifications reach a stability phase, additions of new datatypes to the RAML are ok, but removing or renaming any elements of the RAML should NOT be allowed. This is an example of the open/close principle and, if development teams properly follow this principle, a System API is always backward compatible. This principle also reduces the complexity of dealing with versioning hell. One way to govern this open/close design principle is to follow Agile practices during initial API specification design. However, as soon as API specifications start to be implemented, and have a consumer, they must follow a rigid waterfall model/guideline for any change requests to the API specification.

When designing an API specification for a System API, we have to consider that underlying API implementations might change (e.g. moving from a legacy CRM to Salesforce), but API specifications which expose a System API should NOT change. Any changes in underlying System API implementations should not break either Process APIs or User Experience APIs. A System API specification should be seen as a long term asset for an organization.

Process APIs

Process APIs are responsible for orchestrating and choreographing various data exposed by underlying System APIs. The orchestration involves the splitting, routing and aggregating of data. The main purpose of Process APIs is to strictly encapsulate the business process, independent of the System APIs from which the data originates.

For example, when a travel agency which has multi-channel devices, such as mobile and web portals, that travel agency can have a line of business responsible for developing a Process API. That Process API is responsible for collecting itinerary from underlying System APIs (each system API is responsible for getting the itinerary from one specific airline), aggregating them and exposing them to either portal or mobile devices via two User Experience APIs. A Process API's main core functionality can be defined as:

Expose Rest API endpoints to upper layer consumers by implementing well-defined API specifications.

Transportation in which the main connector used are HTTP Request Connector, Rest Connect Connector or Messaging Connectors such Amazon SQS, JMS, Anypoint MQ, or AMQP

Routing request to each required System API. For example, using a choice component along with a Scatter-Gather component.

Transformation in which data from target System API is transformed into a canonical domain model for that specific Process API.

Implement SAGA pattern if transaction spans across multiple System APIs.

> Note: This pattern is only required if the Process API is responsible for use cases which span transactions across multiple System APIs. In the case, when a Process API is only aggregating data (read only from System APIs), then there is no need for a SAGA implementation.

Error handling by mapping errors received from underlying System APIs.

Strategy in Developing Business Logic in a Process API

Process APIs, in theory, should be designed by the line of a business unit and should be responsible for either orchestrating or choreographing data. However, business logic can also be implemented in Process APIs. The same logistic concerns about System APIs are applicable here as well. If the logistic questions are addressed, then there are some technical challenges that have to be considered before making the decision to implement business logic in a Process API. In the design strategy, these challenges are discussed. Also, when we cover an example of a SAGA implementation later in this document, you will notice that implementing business logic in a Process API requires a lot of back and forth (a.k.a chatty APIs) with backend systems the HTTP protocol. When APIs are used for mission critical integration, implementing business logic in Process APIs might not be the best option.

Strategy in Designing a Process API Implementation

Process APIs, which are responsible for orchestration and choreography, can be designed based on the Principle of Business Capability and DDD. The order-processing-papi, loan-calculator-papi and sales-report-papi are examples of such Process APIs. In here, order, loan and sales are domains and their functions are processing an order, calculating a loan or reporting sales per business capability.

One of the challenges of designing Process APIs is understanding how to deal with transaction attributes (atomicity, consistency, isolation, and durability), which do not exist in a Process API. To manifest the issue better, the following use case is provided.

Use Case:

As an e-commerce application, I want my customers to have a credit limit. My application must ensure that a new order will not exceed the customer's credit limit. The requirement is to either approve or reject an order if a customer's credit limit has been exceeded.

Order information is kept in orderDB and customer information is kept in customerDB. Both of these databases are wrapped by two System APIs as *Diagram 2.0* illustrates. The **order-fulfillment-papi** is responsible for receiving the customer order and coordinating tasks (choreography approach) between **order-sapi** and **customer-sapi**.

The following is a very high level API Specification for these three APIs

customer-sapi: responsible for reserve credit in customerDB:

```
Http method1: POST
ResourcePath1: /reserve-credit
```

order-sapi: responsible for creating order in orderDB and change the state of order:

```
Http method1: POST
ResourcePath1: /order
Http method2: PUT
ResourcePath2: /state
```

order-fulfillment-papi: responsible for receiving a customer order

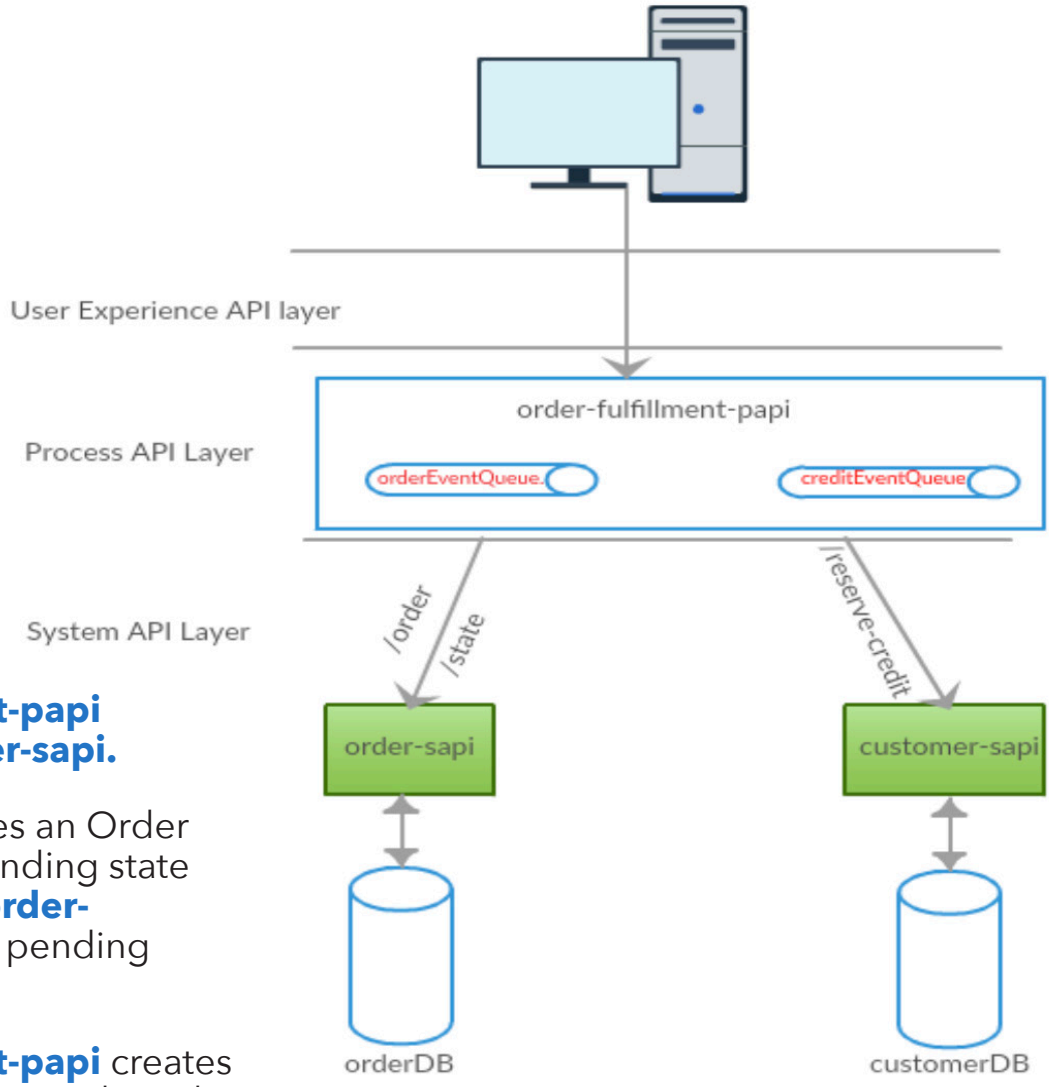
```
Http method1: POST
ResourcePath1: /customer-order
```

How can **order-fulfillment-papi** be designed so orders are either rejected or approved when there is no ACID transaction?

The solution to this problem is to design a SAGA pattern in the **order-fulfillment-papi**, which uses a queuing mechanism. A saga is a sequence of local transactions to each System API. Each local transaction updates the database and sends a response (event Id) via its System API to the Process API to publish an event, triggering the next local transaction in the saga. *Diagram 2* illustrates the interaction between System APIs and Process APIs when a SAGA pattern is implemented.

Deep Down into API-Led Architecture

Diagram 2.0



1. The **order-fulfillment-papi** calls `/order` on **order-sapi.**

2. The **order-sapi** creates an Order in OrderDB with a pending state and returns back to **order-fulfillment-papi** that pending state.

3. The **order-fulfillment-papi** creates an OrderCreated event and sends it to orderEventQueue.

4. A corresponding flow in **order-fulfillment-papi** picks up the orderEventQueue and calls `/reserve-credit` on customer-sapi to allocate credit.

5. The **customer-sapi** tries to reserve credit in the customer database.

6. The **customer-sapi** returns a CreditReserved state or a CreditLimitExceeded state to **order-fulfillment-papi**.

7. The **order-fulfillment-papi** sends either Reserved or Exceeded to creditEventQueue.

8. A corresponding flow in **order-fulfillment-papi** picks up the event from the creditEventQueue and calls `/state` on **order-sapi** to either change the state to approved or cancelled.

9. The **order-sapi** will return back to **order-fulfillment-papi** with transactional success or failure.

10. The **order-fulfillment-papi** will return the result of order creation to user experience API.

Strategy in Adopting a Naming Convention for a Process API

Process APIs that follow the Principle of Business Capability and DDD should have a naming convention similar to "<domain>-<business capability>-<papi>". For example, order-processing-papi, loan-calculator-papi, sales-report-papi are proper naming conventions. Here, order, loan and sales are our domain and processing an order, calculating loan or reporting sales are the business capabilities.

Strategy in Designing a Process API Specification

Process APIs are typically used by User Experience APIs (or proxies) and aggregate models derived from underlying backend System APIs. Typically, the API specification at the Process API layer is much more coarse grained than System APIs. For example, for a customer order API, a resource path such as /customer-orders/{customerId} with a GET operation can be defined to return an aggregate response model from both an order-sapi context and a customer context. When designing RAML specifications at process layer, you can use System API data types (e.g. customer and order data types) for designing the Process API data types. Similar to System APIs, Process APIs should also be designed with usability in mind. A Process API should be developed for the specific requirements of a business. However, the degree of reusability for a Process API is not as high as with a System API.

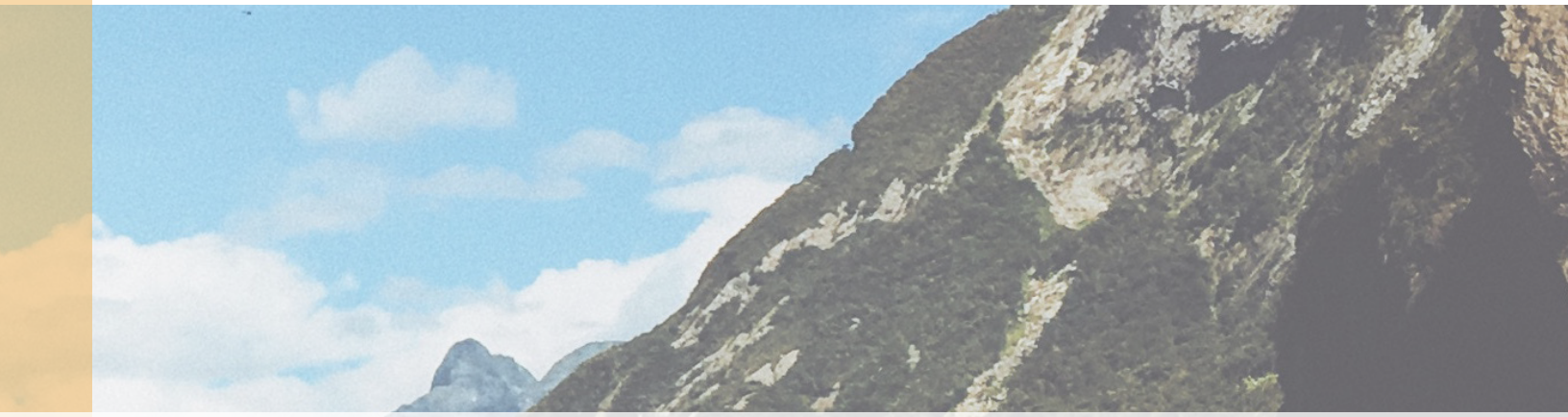
When you want to reuse a Process API for another Process API (horizontal dependency), take into consideration that Process APIs are typically developed by different teams for different business needs. Therefore, there will be high coupling for Process APIs with horizontal dependencies. These high couplings might impact timely delivery, maintainability and overall sustainability of a project.

User Experience APIs

User experience APIs are targeted to be used and reused by different channels such as native mobile applications, web portal applications and portlets, smart assistant applications, etc. Typically, there is an end user behind these devices who indirectly calls User Experience APIs. However, there might be a situation where there is not any user behind the User Experience devices. For example, an external application requires an internal User Experience API to be created with specific API specifications provided by that external application. In that case, the User Experience API must comply with those external API specifications and must transform the external format to internal formats, and vice versa.

A User Experience API's main core functionality can be defined as:

> Expose Rest API endpoints to upper layer consumers by implementing well-defined API specifications, or adapting to an external API specification.
>
> Transportation in which the main connectors used are HTTP Request Connectors, or Rest Connect connectors for calling downstream APIs.
>
> Transformation in which data from downstream APIs are transformed into a format suitable for the end API consumer device.
>
> Error handling by mapping errors received from underlying downstream APIs.
>
> From a non-functional perspective, User Experience APIs have different security requirements based on their target devices.

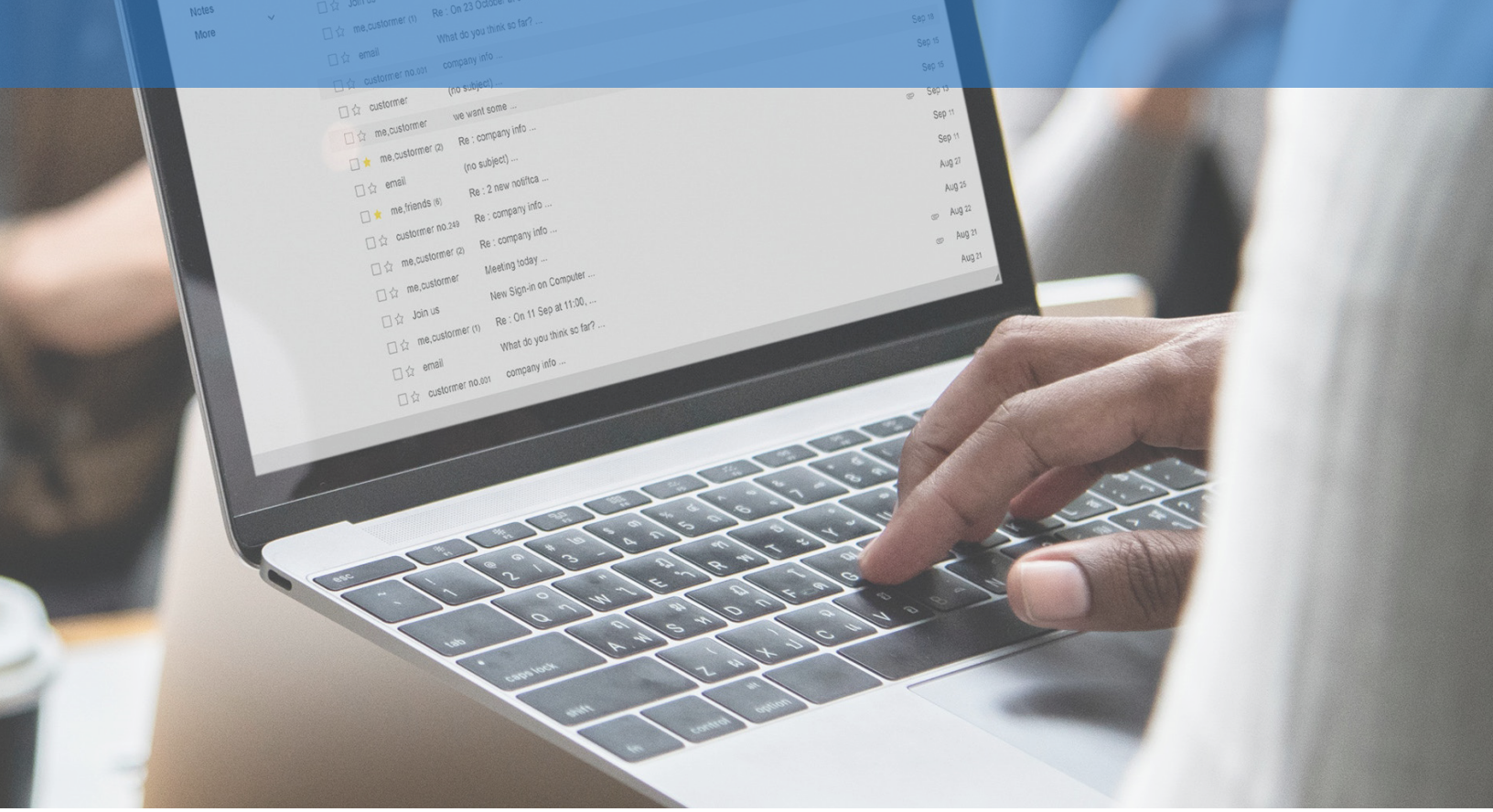


Strategy in Developing Business Logic in a User Experience API

User Experience APIs are not responsible for developing any business, orchestration or choreography logic.

Strategy in Adopting a Naming Convention for a User Experience API

The naming convention is typically based on the target API consumers. For example, for an order domain which has two User Experience APIs, one for mobile and one for portal, then **mobile-order-xapi** and **portal-order-xapi** are good naming conventions. In the situations when a User Experience API is designed for a specific target of groups, then a naming scheme prefixing a group name to a User Experience API should be acceptable. For example, if a User Experience API is designed for a claims department at an insurance company, you might use **claim-users-order-xapi** (claim is another department which needs a subset of data from order-fulfillment-papi) is an appropriate name.



Strategy in Designing API Specification for a User Experience API

Since a User Experience API is targeting an end user's device, the specification of these types of APIs should follow a top-down design approach. That is, the requirements for writing entire an API specification should come from API consumers. The degree of reusability at a User Experience API is not as high as a Process API or a System API. Therefore, we should NOT try to design API specifications to be reused by multiple devices or target groups. However, there are situations where there is no need for developing extra User Experience APIs. Those situations are discussed under the provided design patterns.

API-Led Connectivity Patterns

Next I will outline some variants of API-Led patterns. These patterns can be adopted based on applicable use cases. The format of patterns is as follows:

Pattern Name: Name of the Pattern based on layer.

Motivation (Forces)**:** A brief description of the motivation behind this pattern.

Applicability: Situations and context where this pattern is used.

Participants: Only API types which are implemented on a MuleSoft Platform.

Structure: A graphical representation of the API-Led pattern in context.

Since System APIs do not have to be implemented in MuleSoft, the following legend is used to identify whether a System API is designed based on MuleSoft practice and guidelines, or other guidelines.

System API followed MuleSoft guideline

System API followed based on other industry best practices

Adhoc System APIs including SOAP based APIs

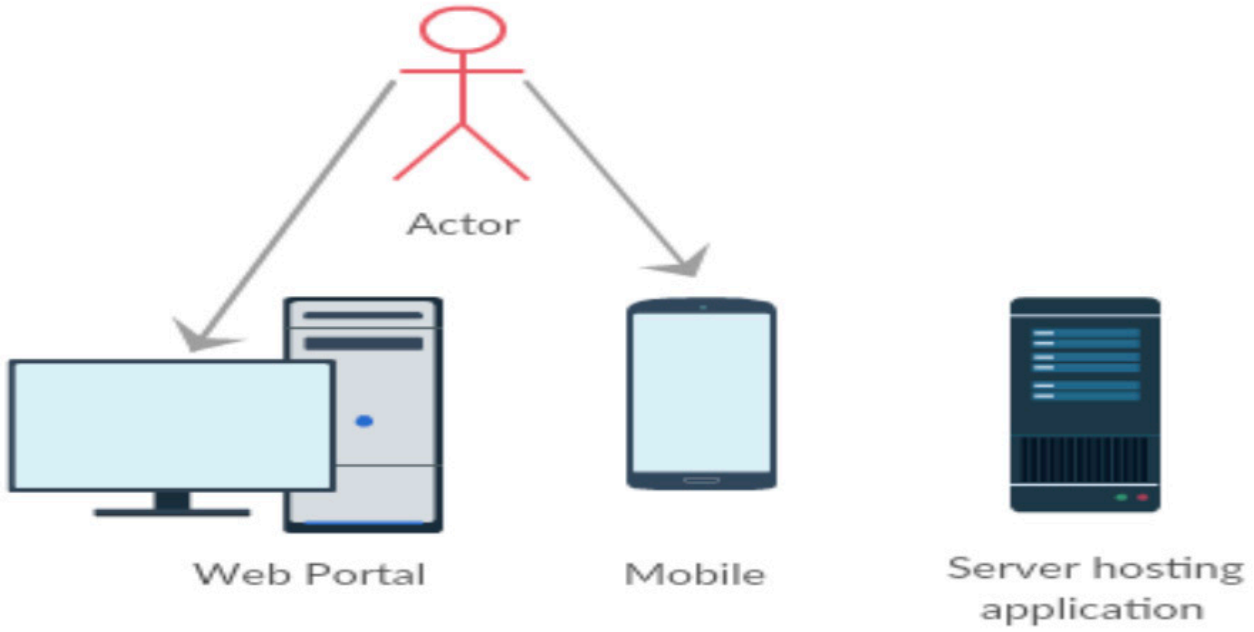
Also, in the diagrams provided for pattern structures, the following letters are used to indicate the type of API used in each layer.

S: System API

P: Process API

X: User Experience API

Further, the following devices are used to indicate the channels where the API will be consumed.



1. Pattern I

a. Pattern: Three Layers Greenfield

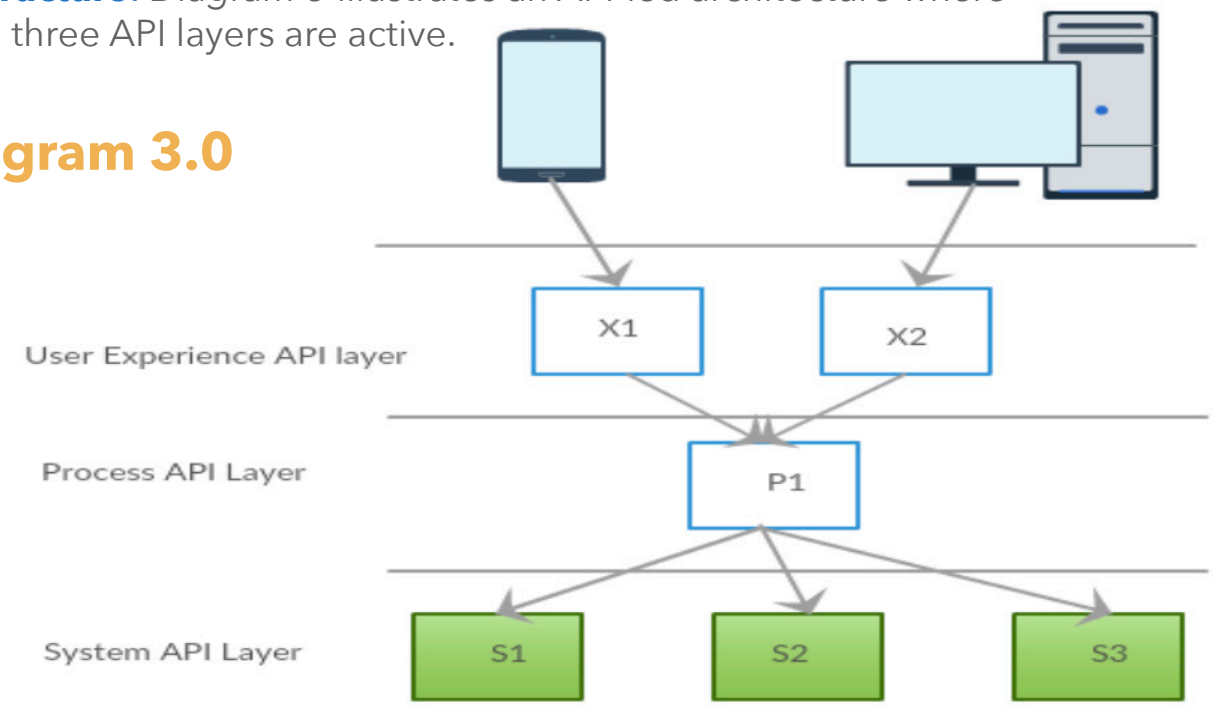
b. Motivation: When all three layers of API are required to be designed on the MuleSoft platform.

c. Applicability: This pattern is applicable when there are multiple channels of business, orchestrating and choreographing need to happen in MuleSoft, and backend systems all need to be exposed through MuleSoft System APIs.This is the standard and most general use case where all three API layers are required. This pattern can be used for both greenfield or legacy migration projects. All APIs are deployed onto MuleSoft platforms, where there are multiple channels of business (e.g. mobile app and web portal). There is a line of business responsible for developing the Process APIs. In the best case scenario, User Experience APIs should be running on a different environment (e.g. In an on-premise solution, User Experience APIs run close to the edge-- in the DMZ) than that of the Process and System APIs. This pattern allows for a new channel of business (e.g. a new portal to be developed for sales department, which is different from other portals) to be added to the overall system architecture without impacting existing APIs. (It is assumed here that an organization does not have any existing System APIs.)

d. Participants: User Experience APIs, Process APIs and System APIs

e. Structure: *Diagram 3* illustrates an API-led architecture where all three API layers are active.

Diagram 3.0



2. Pattern II

a. Pattern: Two Layers no Process API

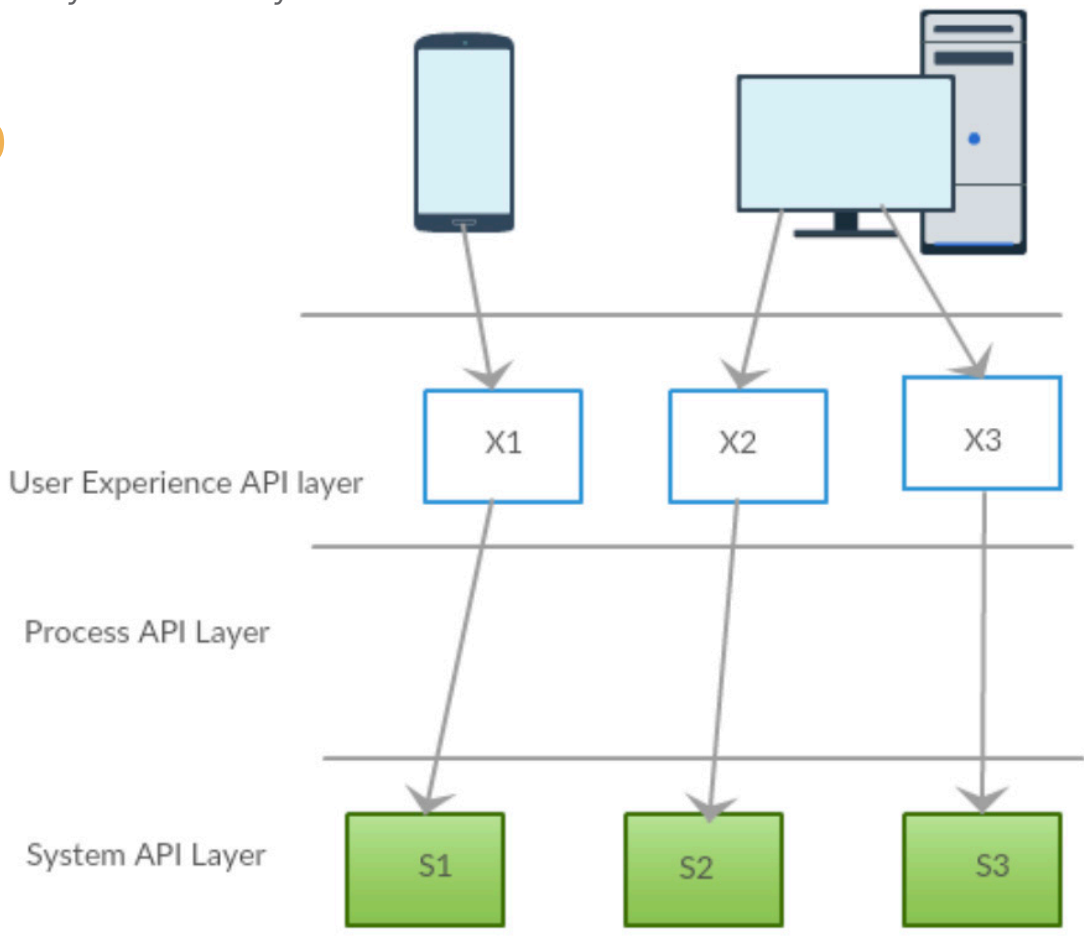
b. Motivation: When there are no requirements for orchestration or choreography, and when the backend systems are exposed through System APIs.

c. Applicability: This pattern is applicable for reusing existing System APIs, or when the User Experience API needs to be in a different environment than the System APIs. When non-functional requirements (e.g. security) on User Experience API are different from the System API layer and data formats are not the same. User experience APIs act simply as a transformation of backend data using System APIs to target audience or devices. It is assumed that downstream systems are responsible for developing business logics orchestration and choreography. For example, when business logic implemented is a stored procedure, AS400 RPG code or COBOL.

d. Participants: User Experience APIs and System APIs.

e. Structure: Diagram 4 illustrates an API-led architecture where only User Experience and System API layers are active.

Diagram 4.0



3. Pattern III

a. **Pattern: Two Layers no MuleSoft System API**

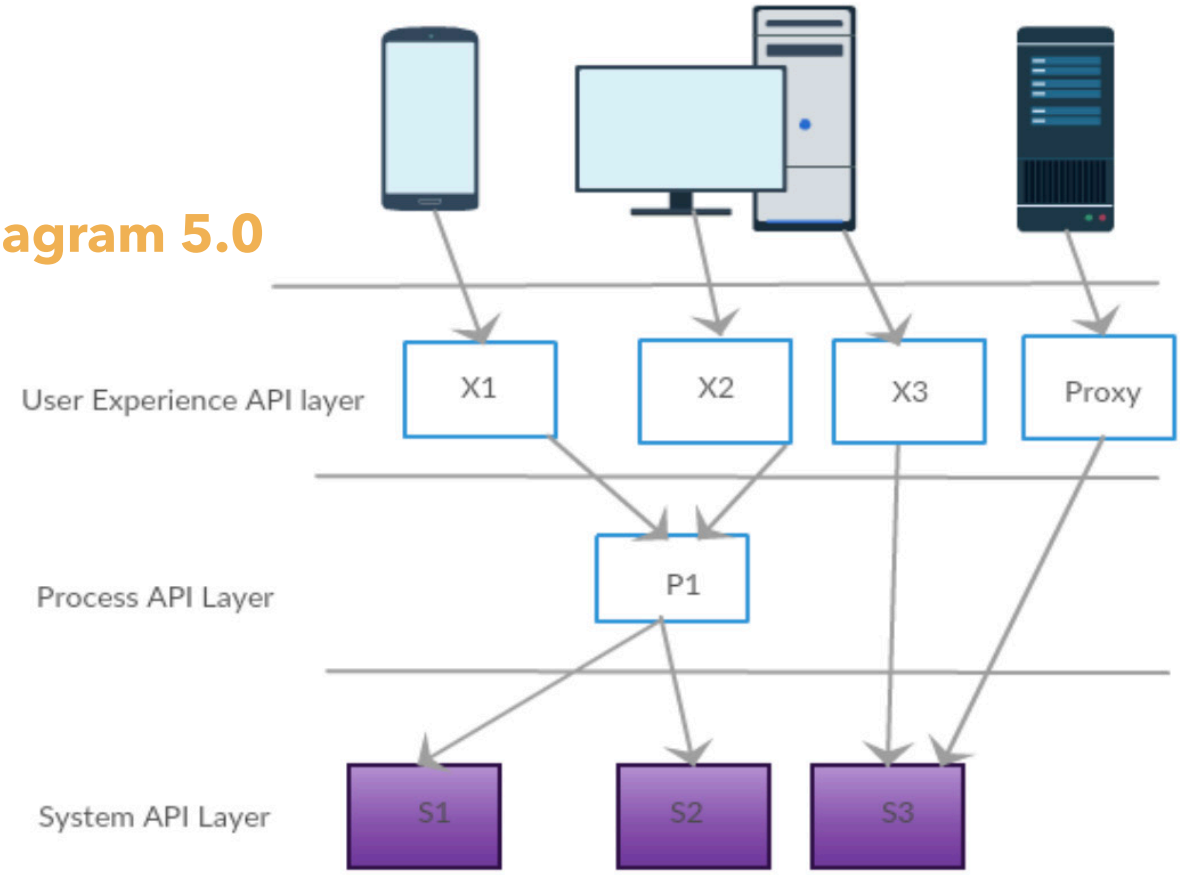
b. **Motivation:** When System APIs are designed outside of the MuleSoft platform.

c. **Applicability:** This pattern is applicable when System APIs are not developed on the MuleSoft platform but are designed using other best practices such as OpenAPI specification. There is no need to unnecessarily wrap and produce a new set of System APIs. This pattern can be used in many scenarios. For example, when the User Experience 3 wants to expose a subset of data from the System API 3, or an internal application wants to get exact data from the System API 3, but we do not want (e.g. use proxy to throttle or make S3 secure) to expose the System API 3 directly to the internal application. Another example is when the Process API 1 has to orchestrate and choreograph functionality provided by the S1 and S2 system APIs.

d. **Participants:** User Experience APIs or Process APIs and/or Proxy

e. **Structure:** *Diagram 5* illustrates an API-led architecture with only two active User Experience and a Process API layers.

Diagram 5.0



4. Pattern IV

a. Pattern: Two layers no User Experience

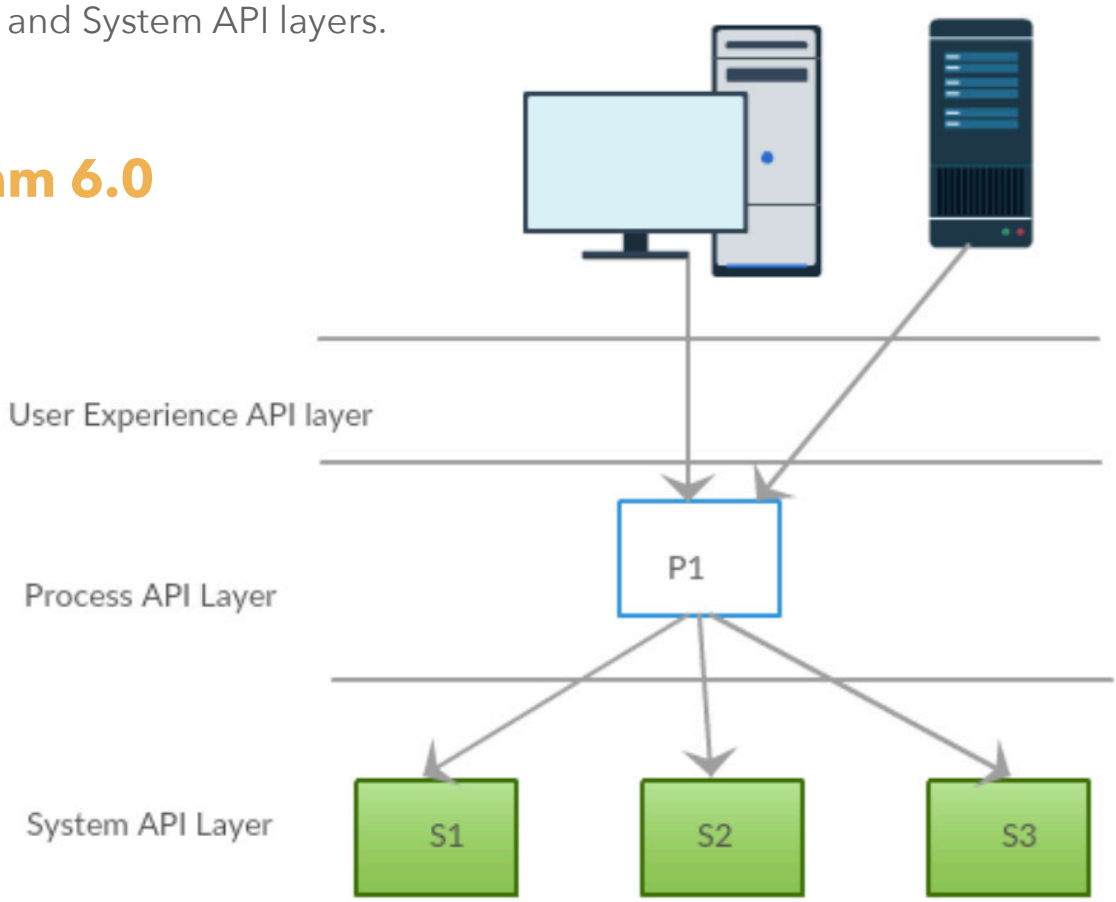
b. Motivation: When there is no need for a User Experience API and the Process API can be exposed to API consumers.

c. Applicability: This pattern is applicable when the process API has the entire API specification and payload to meet the requirements of the API consumer. This pattern is applicable when there is only one User Experience channel and there is no special security requirement for accessing the Process API. This pattern can be introduced in the beginning of a project and when there are not any specific requirements or needs for a User Experience API. As the project evolves, a need for a User Experience API may arise. In that instance, then a User Experience API can be introduced at that time. This pattern is also applicable in situations where APIs are used for integrations between subsystems, where there is no actual user interface to consume the Process API through a User Experience API.

d. Participants: Process APIs and System APIs.

e. Structure: *Diagram 6* illustrates an API-led architecture with two active Process and System API layers.

Diagram 6.0



5. Pattern V

a. **Pattern: Four Layers with Legacy System of Records**

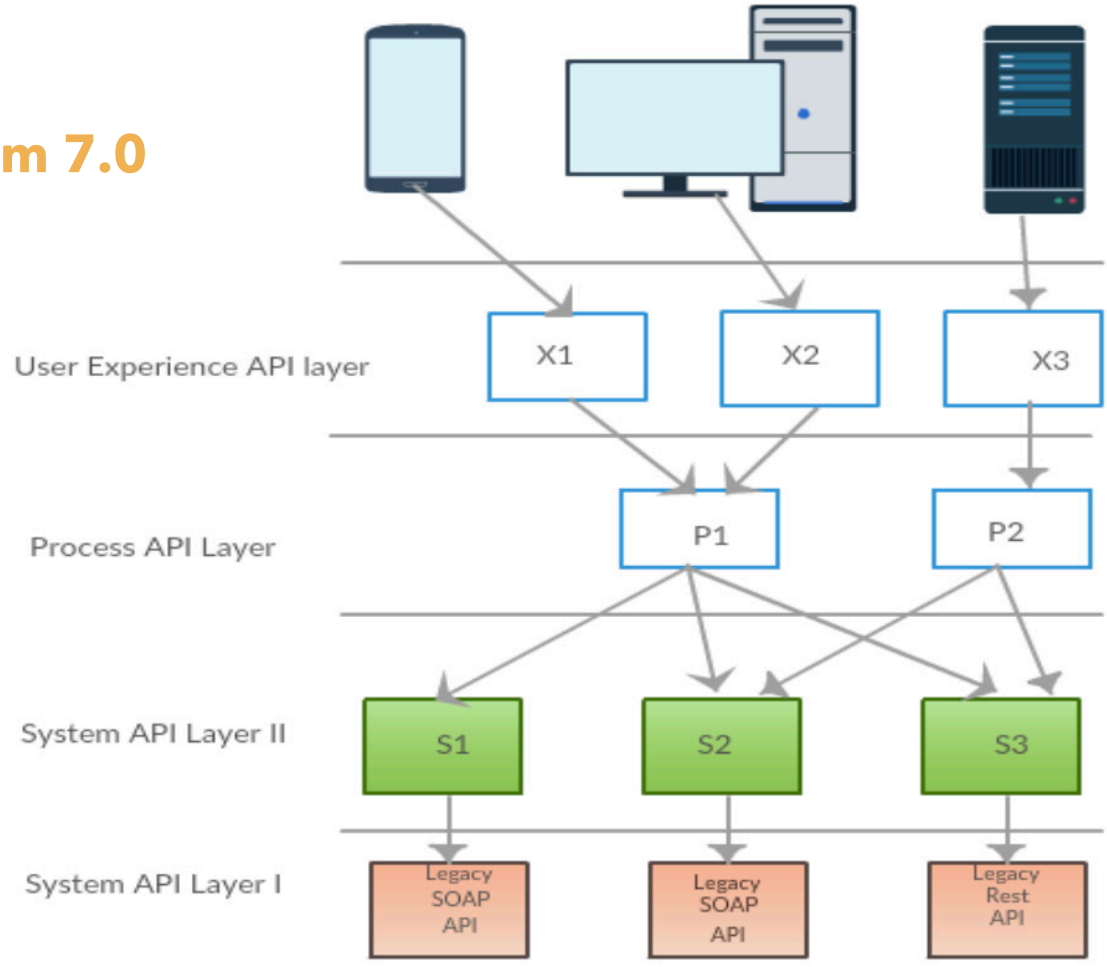
b. **Motivation:** When the underlying system of records originated from legacy System APIs or products.

c. **Applicability:** This pattern is applicable when the question arises: why do we have MuleSoft System APIs when there is already a legacy System API (Layer I) in an organization. Although the Pattern I seems similar to this one, in Pattern I the development of System APIs is justifiable as there have not been any System APIs in the organization, whereas here there are already some legacy System APIs. The question is whether the existing legacy APIs should be reused or wrapped by new System APIs. The MuleSoft System APIs (Layer II) here are introduced with the decision in mind that the legacy system APIs or products will change in the near future or legacy SOAP Web services will need to be broken down into multiple cohesive micro APIs.

d. **Participants:** User Experience APIs, Process APIs and System APIs.

e. **Structure:** *Diagram 7* illustrates an API-led architecture where all three API layers are active.

Diagram 7.0



6. Pattern VI

a. Pattern: Four Layer Proxy

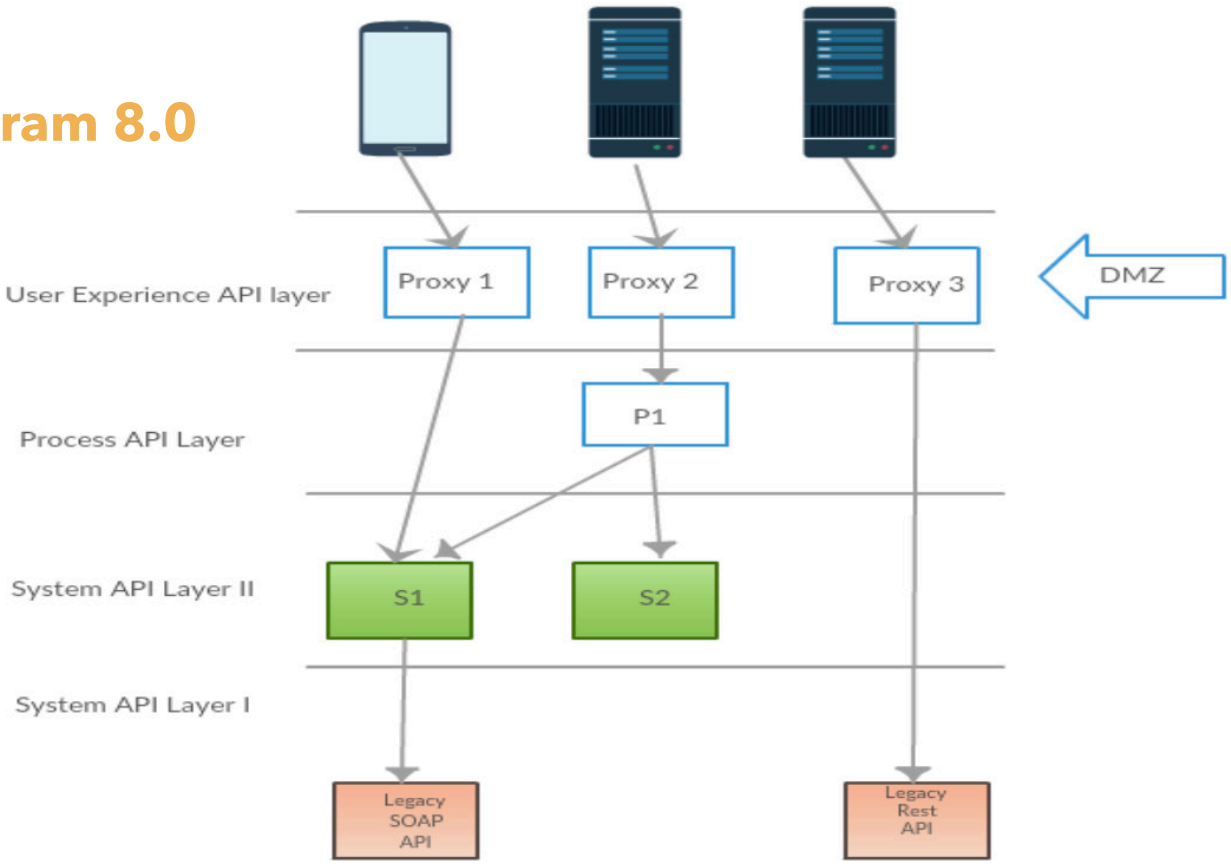
b. Motivation: When a Proxy API can act as a User Experience API.

c. Applicability: This pattern is applicable when there is no need to develop an User Experience API to transform underlying API specifications to the format an API consumer needs. Also, when API consumers should not directly communicate with either Process APIs or System APIs. The common use case here is when an existing legacy API needs to be exposed to other applications, but we would like to use MuleSoft's API manager to add policies around this integration (e.g. legacy API is not secure, or request has to be throttled). Another scenario is when the client application of an API-Led wants access to Process API, but the Process API endpoints cannot be exposed, or the proxy of Process API needs to be on a different environment (e.g. in DMZ). It is also possible to design a RAML specification for a legacy Rest API and deploy it as part of the proxy to exchange. (e.g. Proxy 3).

d. Participants: Proxy, Process APIs and System APIs.

e. Structure: *Diagram 8* illustrates an API-led architecture where proxy pattern is used in three different scenarios.

Diagram 8.0



7. Pattern VII

a. **Pattern: Two Layer Adapter**

b. **Motivation:** When there is no User Experience API and the APIs are used for integrating one system of records to another. System APIs are engaged here for developing an adapter pattern.

c. **Applicability:** This pattern is applicable when there is no user interaction involved in API-led. For example, the system of records from Application A needs to be read via System API S3 periodically, enriched from System API 2 and written into Application B via System API 1. These System APIs are designed so they can be reused by other integrations as well. ***Note, Diagram 9 only illustrates the structure of this pattern in the context of API-Led. It does not include technical specification for such an integration.*** For example, if data loss, duplication or volume of data are considerations, then those should be addressed in a technical design specification. This pattern is a variant of the source adapter (S3) and the target adapter (S1) integration pattern. If there is a need, the Process API can also implement other reliability patterns such as SAGA.

d. **Participants:** Process APIs and System APIs.

e. **Structure:** *Diagram 9 illustrates an API-led architecture with two active Process and System API layers.*

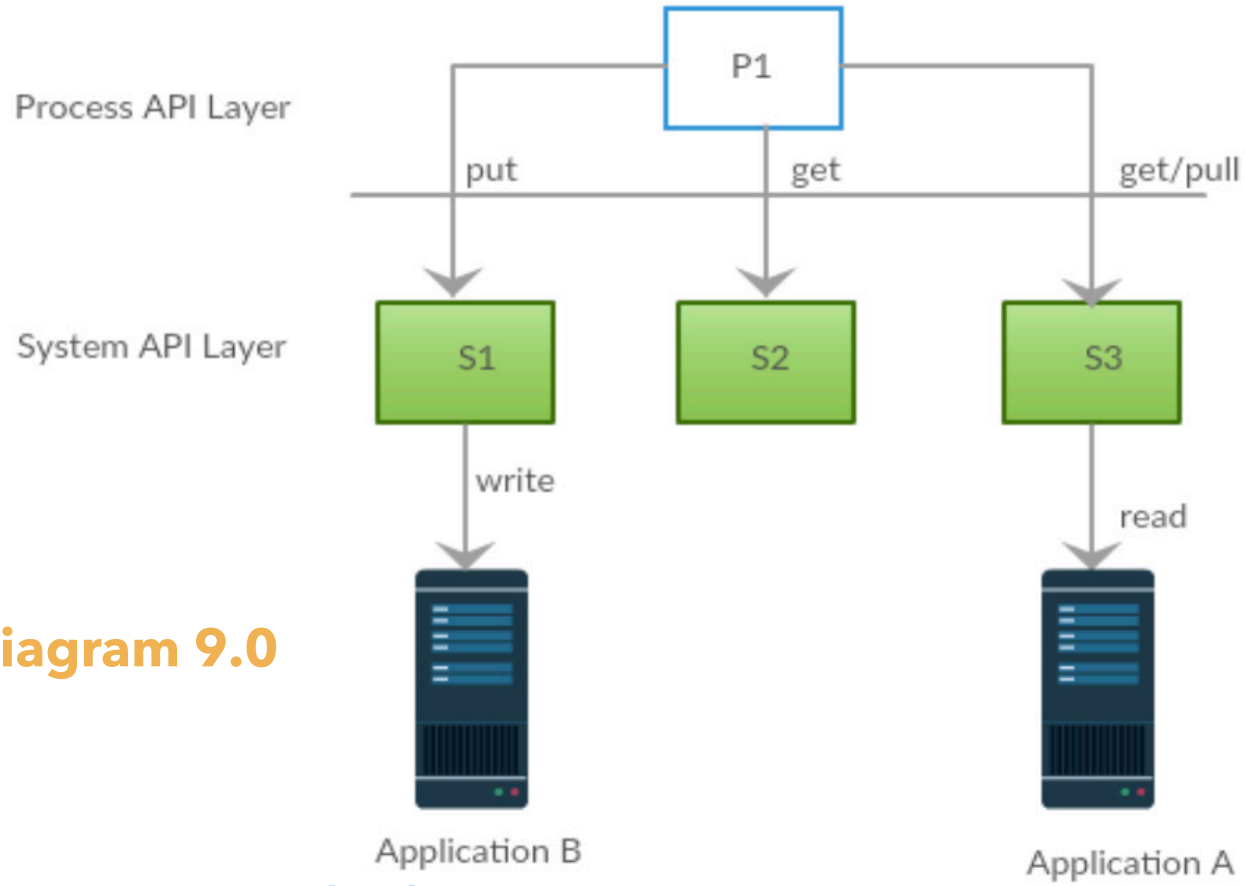


Diagram 9.0

8. Pattern VIII

a. Pattern: Three Layers Adapter

b. Motivation: When there are not any User Experience APIs, and when APIs are used for integrating one system records to another. There are legacy APIs (S4) which can be reused and will not be retired in the future. There are standard Web APIs (S3), but they are not part of the MuleSoft environments. There is a legacy product, which exposes its functionality through its legacy API (S2). This product will be retired within a year.

c. Applicability: This pattern is applicable when there is no user interaction involved in API-led. For example, a system of records from Application A needs to be read via a legacy system API S4 periodically, enriched from System API 3 and written into Application B via System API 1. The legacy System API 4 needs to be securely accessed and monitored via a MuleSoft proxy. System API 3 is well designed and exposes its API through open API specifications. The Application B, which is running on a legacy system (e.g. CRM on AS400 has legacy APIs. This legacy product will be replaced with a modern application (e.g. Salesforce) in the near future.

d. Participants: Process APIs and System APIs and Proxy.

e. Structure: *Diagram 10* illustrates an API-led architecture where only MuleSoft API Process and System Layers are active.

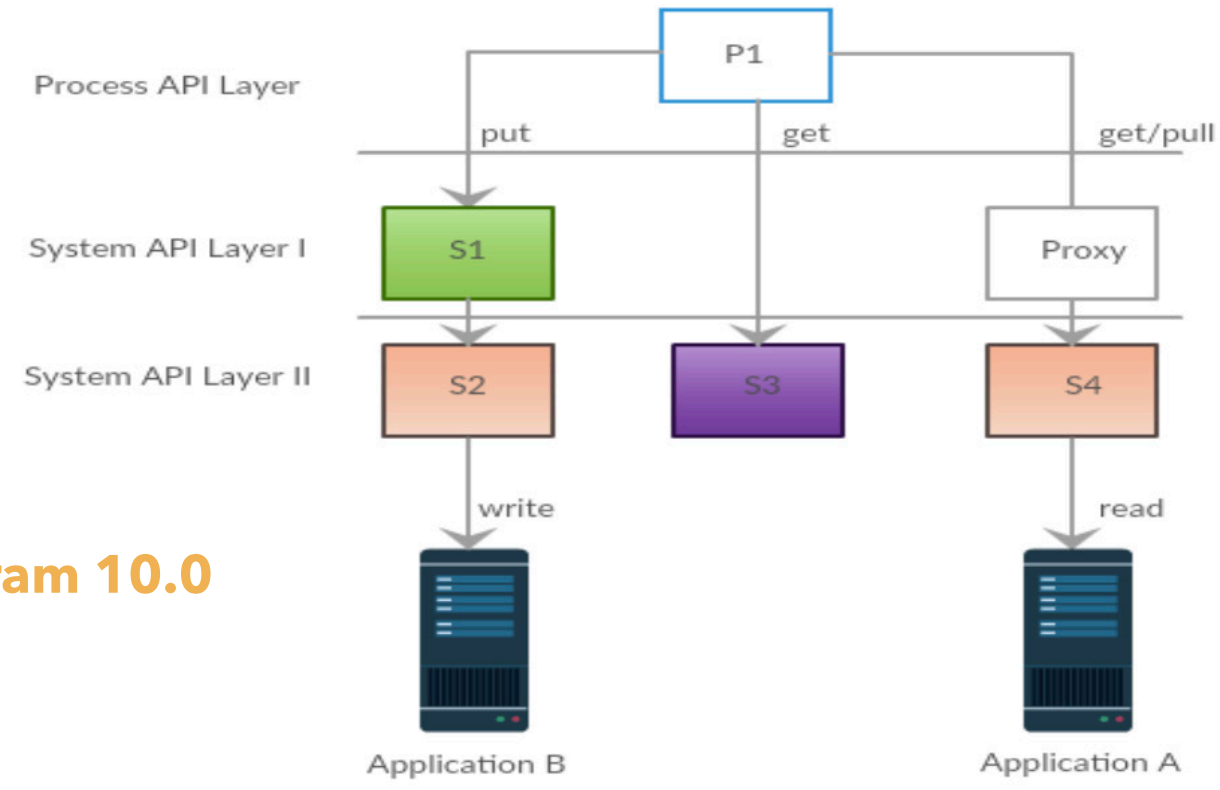
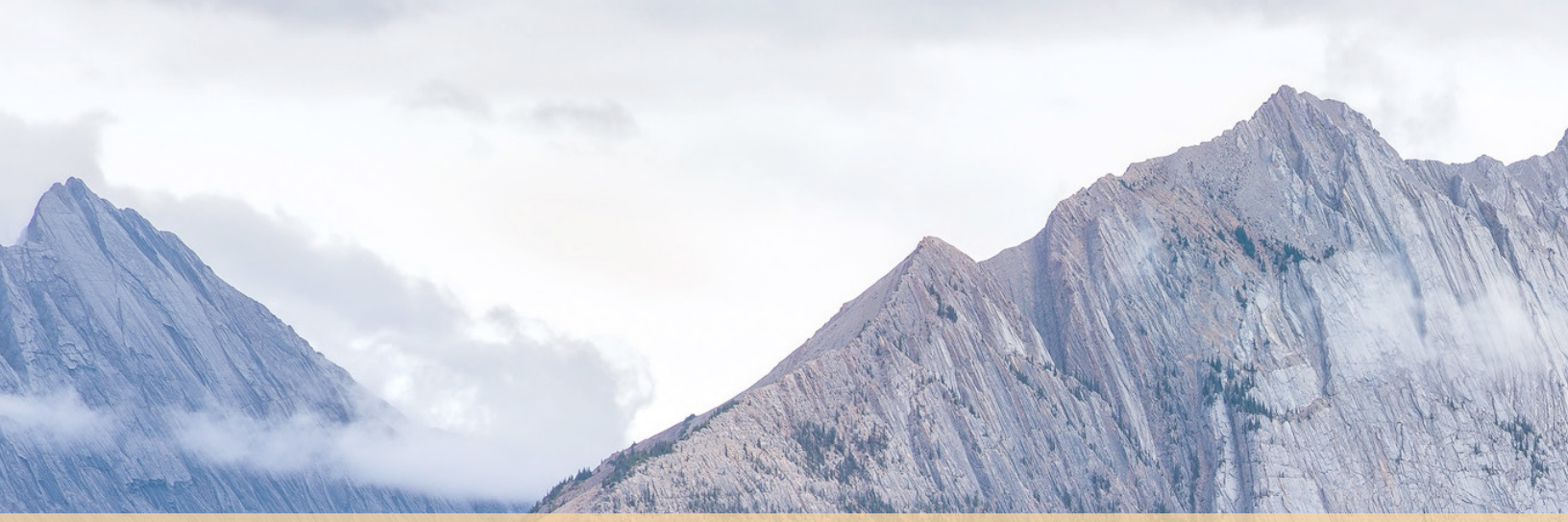


Diagram 10.0



CONCLUSION

This white paper covers a few strategic decision points for adopting API-led architecture. It briefly provides some high-level guidelines for assigning roles and responsibilities to each API layer. It also provides a few API-led patterns, which can be used in different use cases and scenarios. This document, by no means, is a complete technical representation of API-Led architecture. It can be used as a helpful tool to guide decision making during the API-Led architecture discovery and design process.

Deep Down into API-Led Architecture

MS3

FUTURE PROOF
CLOUD INTEGRATION

www.ms3-inc.com

contact@ms3-inc.com

304-579-8100

308 S Fairfax Blvd.
Ranson, WV 25438