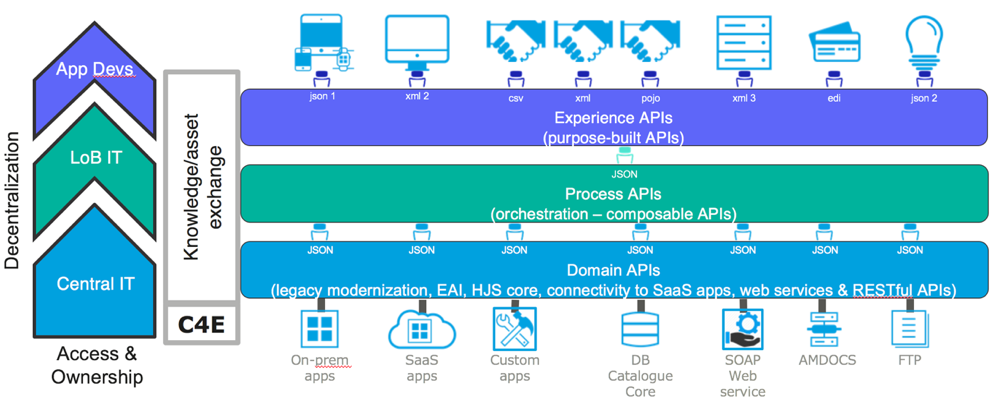
# Concept

Microservices architecture forms the core of this model in that the services deployed in each of these layers are autonomous, loosely coupled, scalable and adaptable for use in multiple contexts. This is particularly true for the System APIs that expose the underlying systems as a set of granular services that can be mashed up to service various use case scenarios. These services communicate with each other using either REST-style synchronous protocol such as HTTP(S) or asynchronous protocol such as JMS or AMQP. There could be file based protocols such as FTP should a use case demand.

An API-led connectivity approach provides a framework for ordering and structuring the building blocks of a composable enterprise to optimize the agility and flexibility benefits of using an API Platform.  This comes in a form a multi-tier architecture with three distinct API layers.

Figure 1: API Layers Concept



## **System APIs**

System APIs are used to manage the interaction with application layer code via a secure, testable endpoint. They can be utilized as the back-end for a website or mobile application, or to expose existing closed application functions to other consumers.

Some System APIs will be built in native application technology. A good example of this would be an ASP.NET site that utilizes domain logic and Entity Framework to provide secure management of data stored in a database. In this case, it would not make sense to build this API in the API platform, as a significant amount of application functionality would be given up.

Other System APIs will expose existing application functionality as a REST API. In these cases, the API platform with connectors or other features would likely be used as the implementation technology to build this integration.

The list of bulleted items below are characteristics of System APIs and should be followed as a rule of when to leverage a System API (as opposed to an Experience or Process API) in the integration architecture.

* Underlying core systems of record
* APIs for doing Create, Read, Update, Delete (CRUD) operations on a system
* Fine Grained with very specific actions associated with a specific system.
* Typically, private to the specific line of business, team, or system the action is taking on, unless the system API itself can be exposed for reusability.

If needed by the functional or technical requirement, System APIs can perform filtering if needed to limit the payload size on a large data. For example, if a System API call resulted in a payload of 200,000 records per call, then filtering can be applied or even chunking of the payload sizes into multiple API payloads through node iteration techniques to maintain a cursor on the array of records for the payload.

## Process APIs

A Process API is a microservice that pulls together many other microservice APIs into a coherent piece of business functionality. Process APIs are built to manage the meshing of microservices. The key value add of a Process API is to prevent business-level functions from becoming large monolithic applications. While System APIs can be implemented in either an API platform or native technologies, in most cases Process APIs will be built as an API in an API platform that has orchestration, workflow, or instrumentation features, and has the ability that API’s can consumes other API (i.e. “Composite API’s).

An encapsulation of a business entity capability, like case, customer, or equipment is a digital asset for any organization. The process APIs or process-level microservices are in line with the concept of an autonomous service which has been designed with enough abstraction to hide the underlying systems of record. None of the system details are leaked through the API. The responsibility of these APIs is discrete and agnostic to any business process.

Aggregate Process APIs are formed by composable System APIs and other APIs. The composition of system APIs can take the form of explicit API orchestration (direct calls) or through the more reliable API choreography by which they are driven by business events relevant to the context of the composition (order fulfillment, for example).

* They orchestrate and choreograph the invocation across multiple System APIs.
* The orchestrations may perform data aggregation (e.g. split-join), conditional routing (e.g. if Region=APAC then do this…), looping, or filtering (e.g. Only show available orders), and more. Filtering could also be performed at the other API layers, such as Experience and System API’s if needed.
* Business Rules are applied at the Process APIs

## **Experience APIs**

Both Process and System APIs should be tailored and exposed to suit the needs of each business channel and digital touch point. The adaptation is shaped by the desired digital experience and is what is called the Experience API. Sometimes the adaptation of the API is technically motivated: a security mechanism might be needed on one channel; the types of channel may differ greatly as do mobile, web and devices; a composition of multiple APIs might be needed according to the backend for frontend pattern. In other contexts, business differences must be catered to handle such adaptations that consider the specific requirements of groups of users like employees, customers, and partners.

Where a Process API is built to maximize business process reuse across the enterprise, an Experience API is purpose-built to manage the connectivity needed by an application. These Experience APIs will mesh together Process APIs (and in some cases, System APIs) to provide a consistent application experience despite changes in the overall microservice makeup of the enterprise. Like with Process APIs, almost all Experience APIs will be built in an API platform that can publish APIs for consumers.

In all these cases the API gateway pattern is a good approach because it is where API compositions and proxies are deployed. API Management facilitates the administrative application of recurring logic, like security, rate limiting, auditing and data filtering to the experience APIs on the gateway. The use of API management to apply policies that encapsulate the tailored logic makes the adaptation of system and process APIs relatively quick and easy. Experience APIs will be public and accessible only to VA developers and selected partners to streamline integration with VA applications.

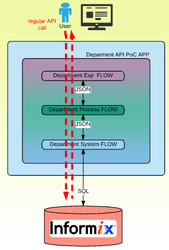
The list of bulleted items below are characteristics of Experience APIs and should be followed as a rule forwhen to leverage an Experience API (as opposed to System or Process API) in the integration architecture.

* How data can be reconfigured so that it is most easily consumed by its intended audience.
* APIs that are exposed to a User Interface or Application layer
* Typically, APIs exposed to Trading Partners or across organization lines is through Experience APIs.
* In a mature and consolidated architecture, separation between Experience, Process and System layers might be physical. This means we will find different deployment units (a different app, or zip) dedicated to each one of these layers.

## API Flow

Typically, the runtime flow of APIs is Experience to Process to System (Top-Down), as depicted in the diagram below. However, if the consumer of an API is another System instead of an End User for a User Interface application, then often the Process or System API are exposed directly to the Consumer

Figure 2: API Layers Invocation



## API Layer Decision Tree

When determining whether to categorize an API and which corresponding layer it should be aligned with, the following decision tree should be applied.

(Editors Note: This will become a table to resolve Mediawiki issues).

1. Does the system being integrated already provide native APIs that can be invoked to perform system interoperability?
2. Sequence Flow Diagram.
3. RAML or Swagger API Specification for the API.
4. Key Use Cases to support the requirements and user stories
5. Does the system being integrated already provide native APIs that can be invoked to perform system interoperability?
6. Sequence Flow Diagram.
7. RAML or Swagger API Specification for the API.
8. Key Use Cases to support the requirements and user stories
9. **Mock Phase** includes creating the Mock service to provide sample request and response in the service endpoint. The service at this point should be deployed in its initial iteration with mock inputs and outputs, for target users to be able to consume that service to integrate with their end user platforms. The mock APIs are only available in Dev and Integration and QA environments, and should never be promoted to Production.

**Deliverables** produced by each developer in this phase:

1. Mock service operations returning sample output for successful service requests, following the RAML or Swagger API contract defined in the Design phase.
2. Mock service operations returning sample error output for negative service requests.
3. Deployment of the mock service.
4. Does the system being integrated already provide native APIs that can be invoked to perform system interoperability?
   1. If yes…. Use the native System API and don’t develop a new one in the API Platform
   2. If no…. Proceed to 2.
5. Is the API performing an Insert, Update, Read, or Delete on a specific system (EX: Oracle, SFDC, Workday) without applying business rules?
   1. If yes… System API.
   2. If No…. Proceed to 3.
6. Is the API orchestrating a process or require invoking other API(s)?
   1. If yes… Process API.
   2. If No… Proceed to 4.
7. Is the API performing transformation or business rules to prepare for system interoperability?
   1. If yes… Process API.
   2. If No…Proceed to 5.
8. Is the API being consumed or invoked by a User Interface?
   1. If yes… Experience API
   2. If No…. Proceed to #1 and check through the rules

# Loose Coupling

Loose coupling is an Archicture design technique that allows a solution to reduce dependencies between components in a system so that the impact of change is isolated. In turn, this reduces the total cost of ownership and impacts of changes, while enabling solution teams to engineer and maintain systematic solutions rapidly.

Separate APIs into the 1 of the 3 API layers described.

Each API is categorized into Experience, Process, or System API and managed accordingly.

Separate Provider API Logic from Consumer API Logic

Sometimes referred to as “Separation of Concerns,” separate API’s and flows logically. Separate provider and consumer logic. Separation of Concerns provides a logical architecture whereby IT assets (such as APIs, queues, or other architecture components) are separated and categorized for providers and consumers, and are connected through an independent mediation component (API, queue, or other). It creates an architecture whereby provider and consumer never directly communicate

Figure 3: Separation of Concerns



Leverage the Reliability Pattern as a mediation layer between API Process flows that invoke in each other (i.e. Composite Processes).

The Reliability Pattern The [Reliability Pattern](https://docs.mulesoft.com/mule-user-guide/v/3.9/reliability-patterns) documented by MuleSoft, Inc, allows APIs to communicate with each other through a mediation layer—the queue—so that APIs are not invoking each other. The queue provides a mediation layer to “loosely couple” the API flows together. Non- persistent queues can be used so not to impact runtime performance latency of the message flows.

Figure 4: Reliability Pattern for Loose Coupling

A screenshot of a cell phone

Description generated with very high confidence

Process #1)

Process #2

Source (Provider) and Target (Consumer) data never map data fields directly to each other

Instead, source data maps to an enterprise entity data model object (“Canonical”) and target data also maps to the enterprise entity data model (“Canonical”). The Canonical provides the “Broker” mechanism for Providers and Consumers to share data. Universal data model that represents the business process, through a data object acts as a mediation point between Provider and Consumer data mappings. All Provider systems map to the Canonical, all Target systems map from the Canonical, and Provider and Consumer never map directly to each other. A loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. A design should identify and separate components based on volatility and capability, minimizing dependencies

Figure 5: Provider/Consumer Separation



A well-designed Canonical has the following characteristics:

* At a technical definition: it’s an object that is solely comprised of constrained attributes, data structures, and their hierarchical relationships..
* At a business definition: It’s a super-set of all attributes required for a specific business process to be defined and executed across a range of technologies, processes, and external and internal entities.
* Process and result of pre-defining an enterprise data structure and model that will encompass all information required for a business process.
* Enterprise representation of a Business Object.
* Also referred to as: Universal Data Model (UDM), Common Object Model (COM), Common Information Model (CIM)
* Virtual Meta Data

Figure 6: Canonical Example



# Licenses Considerations

## API License Awareness

Because each API has licensing impacts on its associated platform, the team must be judicious on the number of APIs created, to reduce the necessity of procuring more licensing just to comply with the 3-layer design approach described above. The goal is to minimize or keep the number of APIs lower while still maintaining architecture integrity, to keep costs down. Typically, this means only creating “Process” and “Experience” where truly needed. If the design for a specific use case has “Process” and “Experience” APIs that are just wrappers of a “System” API and not providing any additional functionality or logic, it might be best to eliminate these additional layers for the use case, because they will have impacts on the licensing. If building additional APIs has overtly negative impacts on the team’s licensing and causing project cost concerns and the additional APIs aren’t performing any processing outside of wrapping for layer architecture described above, that wouldn’t align with licensing best practices. Therefore, the design for a use case could only call for 1 API layer, the “System API”.

## Native System API

The other licensing consideration is if the system being integrated already provides a “System API.” For example, Salesforce.com provides out-of-the-box system APIs. Other systems, also provide out-of-the-box system APIs. Therefore, in the API platform, you would only design “Process API” to invoke, orchestrate, and map to the already provided native System APIs provided by the system itself. Creating another System API layer on the already provided System API by the system vendor would increase API Management and impact license cost too (i.e. “API Frugality” must be observed).

# API Layer Best Practices

* Always ensure there is clear separation between experience, process and system APIs.
* There has been appropriate decomposition of functionality into deployable services - not just one large service that does everything.
* Always ensure there is appropriate separation of API interface (RAML) from API function (concrete services). RAML can be created at any layer of the API.
* Service/micro-service decomposition is logical and will not impede redeployments unduly.