

Best Practices & Guidelines for API development

- Introduction
- API Definition
 - How API works ?
 - Definition and Design
 - Development
 - Deployment
- API led connectivity
 - Overview
 - API Connectivity Layer
 - System layer
 - Process layer
 - Experience layer
 - Benefits
 - Business benefits
 - Technical benefits
 - Characteristics
 - Web Services
 - RESTful Services
- API life cycle
 - Overview
 - Architecture
 - Introduction
 - Architecture Cycle
 - Architecture Context
 - Architecture Delivery
 - Transition to Development
 - Implementation Governance
 - Change Management
 - Requirements Management
 - DevOps Cycle
 - Design
 - High Level API need
 - Identification of APIs and Resources
 - Specification of the API using RAML
 - Documentation of the API
 - Build
 - Using RAML to generate the development framework
 - Implementing the functionality using AnyPoint Studio
 - Version Control
 - Automation of the build process / Continuous Integration
 - Test
 - Unit Test
 - Integration Test
 - Regression Test
 - Build / Test Iteration
 - Publish / Implement
 - Deployment of APIs
 - Policy Configuration
 - Publication of an API to Exchange
 - Operate
 - Monitor and Analyze
 - Notification and Alarming
 - Insights provided by iPaaS
- API Guidelines
 - API URI / URL
 - Design API URLs
 - Setting the Base URI
 - Design API Request and Response representation(s)
 - Design API Request and Response headers
 - DevOps
- API Patterns
 - Endpoint Redirection
 - Problem
 - Solution
 - Entity endpoints
 - Problem

- Solution
- Content negotiation
 - Problem
 - Solution
- Idempotent Capability
 - Problem
 - Solution
- Idempotent
- Safe
- Concurrency
 - Problem
 - Solution
- Asynchronous processing
 - Problem
 - Solution
- Caching
- API Management
 - API Implementation
- API Versioning
 - Major and Minor versions
- API Security
 - Supporting OAuth2
 - Supporting OpenId Connect
 - Provide machine-readable JSON schema
 - Provide human-readable documents
 - Using modeling languages
- Useful Guidelines
 - Meaningful Error Message
- Glossary
 - Headers
 - Status Codes

Introduction

This section of the document provides a short but complete overview on the design and development of APIs. Throughout the section are references for more detailed information on the different aspects of the API Lifecycle and the technical topics discussed in this section.

API Definition

How API works ?

APIs are providing a very structured interface to integrations or applications. Having incorporated the experience of integration and development projects from the last twenty years, APIs are an approach to avoid cost- and productivity issues of Enterprise Application Integration (EAI) and Service Oriented Architecture (SOA) by - at the same time - incorporating the benefits from these integration technologies. APIs have a number of characteristics which make them distinct from other integration approaches

- On the technology side, APIs use a limited technology stack of RESTful services and JSON for the implementation.
- On the methodology side, the MuleSoft approach to the development of APIs uses a design driven approach, using RAML as a lightweight, human readable definition language. APIs do expose application entities and attributes of those entities. This is a different of a service architecture which exposes business processes and the entities as attributes to the services. APIs are therefore assuming the business service implicit to the environment they are serving.
- On the business side, the definition of APIs are driven by the producer / developer, not by consensus between a limited number of parties.

Definition and Design

Like previous service based integration approaches, APIs are using a formal interface definition to allow developers on consumer and

producer side to understand the characteristics of the interface. The approach endorsed by MuleSoft is the use of RAML documents as a human readable lightweight definition. It is important to follow a design first approach, where the definition of the API is developed before any development of the code is performed. By providing the RAML definition before the development starts, it is possible to develop producer and consumers in parallel.

Development

Using the MuleSoft tooling the development of the API is supported by an end-to-end process: starting with the generation of the RAML specification, the environment provides a mock-up environment for the developer of the API consumer to use for the development. For the developer of the producing side, the RAML can be imported into the AnyPoint Studio environment, which also creates a API application scaffolding. This implies that the interfaces for each function of the API and also the error returns are generated. These scaffolds are the base for the development and testing for the user. A later section of the document describes the development process in deeper detail.

Deployment

After completion of the development and internal developer testing, the API and the application related to the API can be deployed to either an online hosting environment or into a local testing and production environment. This deployment process is supported by local tooling, but also (like the development process) can be supported by tools for Continuous Integration / Continuous Deployment.

API led connectivity

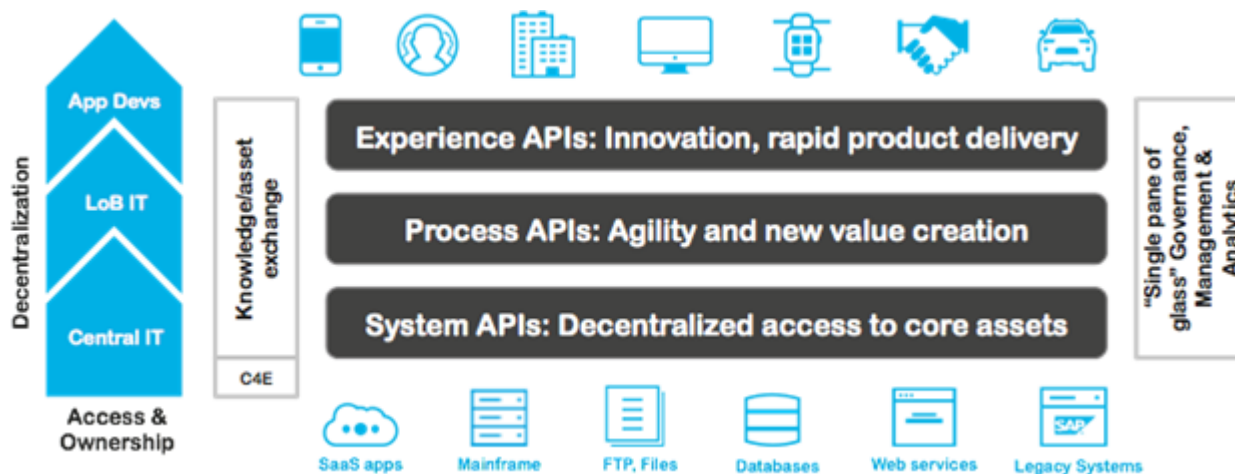
Overview

API-led connectivity is an approach that defines methods for connecting and exposing digital assets. The approach shifts the way IT operates and promotes decentralized access to data and capabilities while not compromising on governance.

The realization of API-led connectivity is a journey that changes the IT operating model and which enables the establishment of the “composable enterprise”, an enterprise in which its assets and services can be leveraged independent of geographic or technical boundaries.

However, the integration application must be more than just an API; the API can only serve as a presentation layer on top of a set of orchestration and connectivity flows. This orchestration and connectivity is critical: without it, API to API connectivity is simply another means of building out point-to-point integration.

Large enterprises have complex, interwoven connectivity needs that require multiple API-led connectivity building blocks. In this context, putting in a framework for ordering and structuring these building blocks is crucial. Agility and flexibility can only come from a multi-tier architecture containing three distinct layers:



API Connectivity Layer

System layer

Underlying all IT architectures are core systems of record (SoR, e.g. one's ERP, key customer and billing systems, proprietary databases, etc.). Often these systems are not easily accessible due to connectivity concerns and APIs provide a means of hiding that complexity from the user. System APIs provide a means of accessing underlying SoR and exposing that data, often in a canonical format, while providing downstream insulation from any interface changes or rationalization of those systems. These APIs will also change more infrequently and will be governed by Central IT given the importance of the underlying systems.

Process layer

The underlying business processes that interact and shape this data should be strictly encapsulated independent of the source systems from which that data originates, as well as the target channels through which that data is to be delivered. For example, in a purchase order process, there is some logic that is common across products, geographies and retail channels that can and should be distilled into a single service that can then be called by product, geography- or channel-specific parent services. These APIs perform specific functions and provide access to non-central data and may be built by either Central IT or Line of Business IT.

Experience layer

Data is now consumed across a broad set of channels, each of which wants access to the same data but in a variety of different forms. For example, a retail branch POS system, ecommerce site and mobile shopping application may all want to access the same customer information fields, but each will require that information in very different formats. Experience APIs are the means by which data can be reconfigured so that it is most easily consumed by its intended audience, all from a common data source, rather than setting up separate point-to-point integrations for each channel.

Benefits

API-led connectivity offers both business and technical benefits.

Business benefits

- IT as a platform for the business: By exposing data assets as services to a broader audience, IT can start to become a platform that allows lines of business to self-serve.
- Increase developer productivity through reuse: Realizing an API-led connectivity approach is consistent with a service oriented approach whereby logic is distilled to its constituent parts and re-used across different applications. This prevents duplication of effort and allows developers to build on each other's efforts.
- More predictable change: By ensuring a modularization of integration logic, and by ensuring a logical separation between modules, IT leaders are able to better estimate and ensure delivery against changes to code. This architecture negates the nightmare scenario of a small database field change having significant downstream impact, and requiring extensive regression testing.

Technical benefits

- Distributed and tailored approach: An API-led connectivity approach recognizes that there is not a one-size-fits-all architecture. This allows connectivity to be addressed in small pieces and for that capability to be exposed through the API or Microservice.
- Greater agility through loose coupling of systems: Within an organization's IT architecture, there are different levels of governance that are appropriate. The so-called bi-modal integration or two-speed IT approach makes this dichotomy explicit: the need to carefully manage and gate changes to core systems of record (e.g. annual schema changes to core ERP systems) whilst retaining the flexibility to iterate quickly for user facing edge systems such as web and mobile applications where continuous innovation and rapid time to market are critical. Separate API tiers allow a different level of governance and control to exist at each layer, making possible simultaneous loose-tight coupling.
- Deeper operational visibility: Approaching connectivity holistically in this way allows greater operational insight, that goes beyond whether an API or a particular interface is working or not, but provides end to end insight from receipt of the initial API request call to fulfilment of that request based on an underlying database query. At each step, fine-grained analysis is possible, that cannot be easily realized when considering connectivity in a piecemeal fashion.

Characteristics

Web Services

The concept of services is an evolution of initial concepts in the integration realm. It defines the attributes of an interface to a system in an integration flow. By providing a number of rules and best practices, services allow a controlled access to well defined functionality and information.

In common definition, services are “unassociated, loosely coupled units of functionality that are self-contained”. From the outside, a service is seen as a capability that provides one or more business functions. In most cases, the service hides a more complex system and exposes a subset of the business functionality of that system.

The system or application providing the functionality is called a service provider, the system or entity using the functionality is called a service consumer. A service can be implemented in many different ways, most commonly used are Web Services, but other means are possible. To describe the interface structure of the Web Service commonly an interface definition is provided. A common standard language for this definition is called WSDL. This language allows the automated generation of producer and consumer components for the service.

RESTful Services

The REST architectural style describes six constraints. These constraints, applied to the architecture, were originally communicated by Roy Fielding in his doctoral dissertation and define the basis of

RESTful-style:

Client-server: A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

Stateless: The client-server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client.

Cacheable: Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.

Uniform interface: The uniform interface constraint is fundamental to the design of any REST service. It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

- **Identification of resources:** individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as HTML, XML or JSON, none of which are the server's internal representation. Manipulation of resources through these representations: when the client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
- **Self-descriptive messages:** each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type. Responses also explicitly indicate their cacheability.
- **Hypermedia as the engine of application state (HATEOAS):** clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by hyperlinks within hypertext). Except for simple fixed entry points to the application, a client does not assume that any particular action is available for any particular resources beyond those described in representations previously received from the server.

Layered system: a client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing caches. They may also enforce security policies.

Code on demand (optional): Servers can temporarily extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client-side scripts as JavaScript. “Code on demand” is the only optional constraint of the REST architecture.

Applications conforming to these REST constraints are named “RESTful”. So, if it violated any of the required constraints, it cannot be considered RESTful.

API life cycle

Overview

Like any other artefact in the IT world, APIs do have a lifecycle that cover from the inception of the API over the design and development to a

productive and maintained stage leading to retirement. As APIs are long lasting components in the architecture (used by a multitude of consumers) an API needs to be maintained over time. This section describes on a high level the concepts of the API lifecycle and compares the different views (Delivery Work and Architecture Work) on the APIs. A detailed view on the lifecycle can be found in the appendix of this document.

Architecture

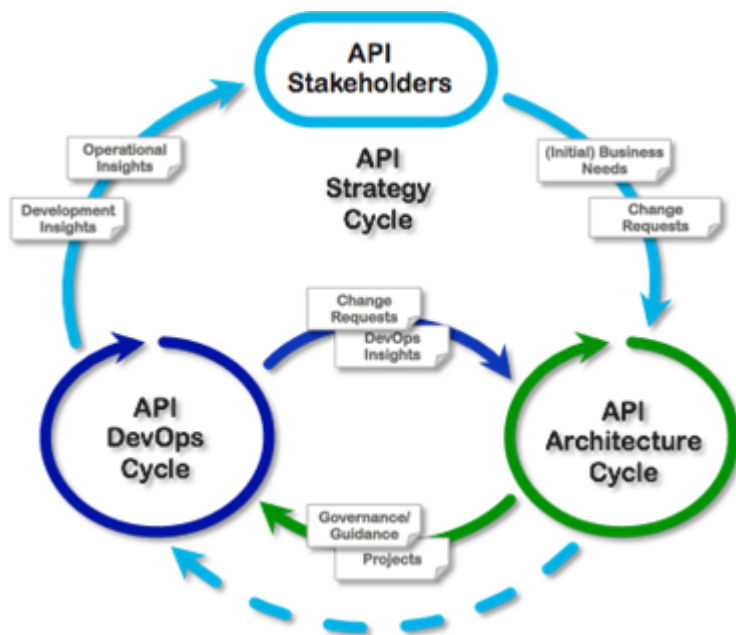
Introduction

APIs and Services always have to be seen as part of the environment they are implemented to serve. All components and systems in these ecosystems are not static architecture and development artefacts. They develop, grow, adopt, and adjust their design and implementation over their lifetime. As the need for functionality and information in the business changes, so have the APIs to adapt to the changed needs and requirements.

For this reason, the different stages in the architecture and development on APIs are displayed as cycles, moving from planning over design over build over productivity over improvements and adjustment back to planning.

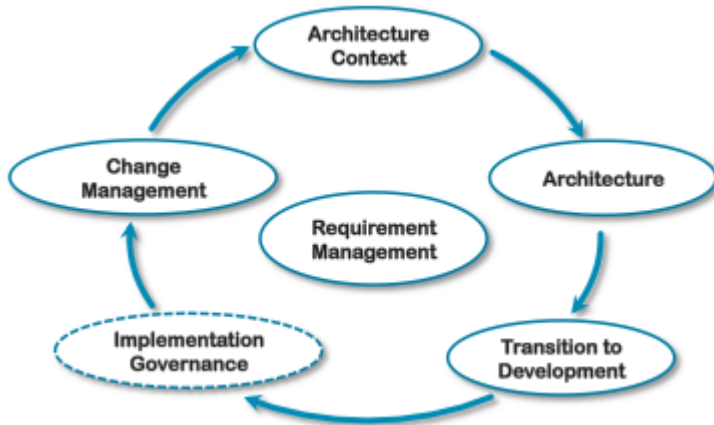
MuleSoft distinguishes 3 different API life-cycles, namely Strategy, Architecture and Development.

A typical starting point for the Strategy cycle is the identification of (business) needs to build fast interfaces exposed to organization internal or external systems. These business needs, e.g. to provide access to functionality for third parties or mobile devices, or alternatively to improve and standardize the integration structure internally, triggers the design process for the APIs, delivering the different artifacts for the implementation. As described in a later part of this section, the architecture cycle is an iterative process, providing quality assurance and fast adaptation on the needs and requirements. Following is the DevOps cycle, previously known as implementation cycle. The result of the implementation cycle is the executable API and integration platform. The cycle closes by identifying additional interfaces or interface requirements by the API stakeholders. It is important to recognize the different group of stakeholders: the API owners, which sponsor and manage the APIs, the development community which adopts and uses the APIs to implement their functionality, and finally the users, which requires availability, speed, and scalability for using the external application.



Architecture Cycle

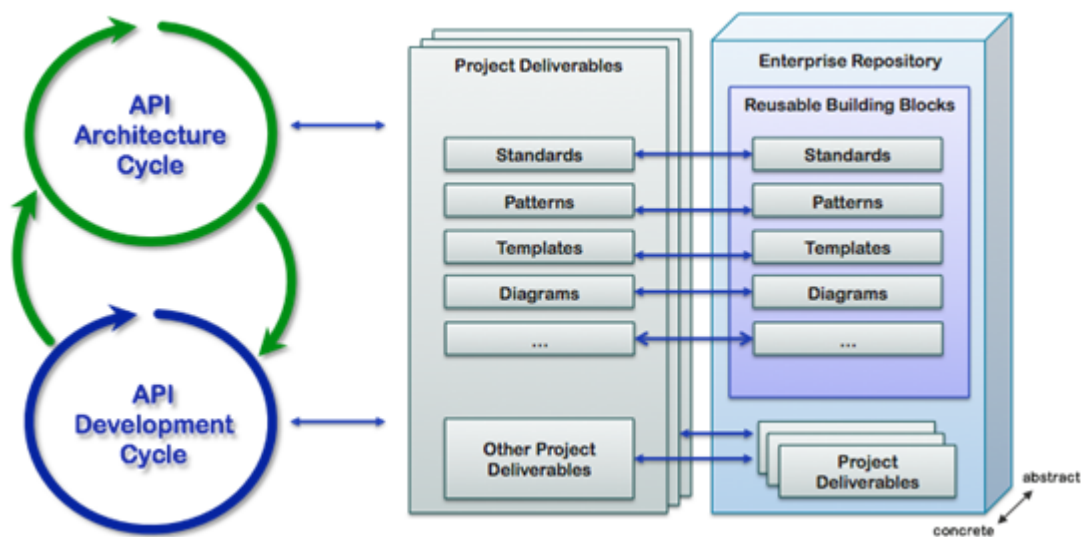
The architecture cycle consists of a number of phases which allow the controlled design of the API architecture. In the display, the center of the architecture cycle is the area of requirement management. The different requirements trigger the context of the architecture. The actual architecture cycle shows the full circle from the architecture context over delivery to the transition to the development cycle. From there it closes via the governance functions and the change management. All steps are detailed in this section.



Architecture Context

The purpose this first phase of the Architecture Cycle is the review of the business and technology context in which the API work will take place, to define the scope of the API program, to establish governance processes and models and to define the key API architecture principles. This phase is also concerned with validating and capturing the business goals and the strategic business drivers and identification of the relevant API users and stakeholders, their objectives and concerns. The result of this phase is the definition of key functional and non-functional requirements for the overall API work and the identification of relevant patterns and building blocks to address these requirements.

The Architecture Context phase (as well as the Architecture Delivery) could greatly benefit from the establishment and utilization of a repository of reusable. Such a repository serves as a central place to create and discover reusable organizational models, business processes, data models, application components, API definitions, integration flows, and infrastructure/technology models. Typically, the repository is initially populated with patterns, best practices, templates, standards, etc., that are needed to start the API work and updated during the Architecture Delivery phase with potentially reusable building blocks.

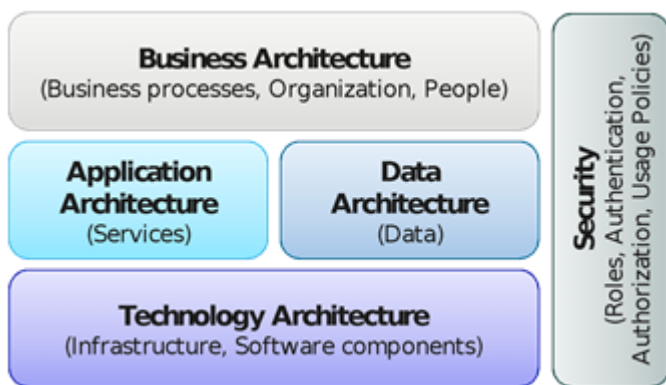


As part of the Anypoint platform, *Anypoint Exchange* provides this repository, it captures, discovers and fosters the reuse of integration best practices, connectors, templates, examples, and APIs, as well as internally developed best practices that are private to the organization. This results in increased productivity for architects and developers, avoids redundant work and ensures adoption of best practices.



Architecture Delivery

The goal of the Architecture Delivery phase is to define the baseline and target API architectures and identify gaps between them. This is done at 3 different levels each focusing on business, information systems (application and data) and technology aspects as shown below



Business Architecture captures the existing and defines the target business processes, organizational structure and governance model that will be supported or impacted by the API work. It also identifies the gaps between the existing and desired (business) states.

Application and Data Architecture has two distinct outcomes: First it captures the existing services and APIs, defines the functionality of the APIs of the target applications which are needed for the Business Architecture, and identifies the gaps between them. It also defines the integration components and interfaces required to bridge these gaps. The second objective is the capture of the data models of the existing application services, the definition of the data models of the new application services, and –as before - the identification of the gaps among them. Finally, in this step the high-level mappings and data transformations required to bridge the gaps are defined.

Technology Architecture defines the required infrastructure and software components, and by comparing them with the existing infrastructure and software components identifies the gaps between them. It also describes non-functional requirements such as availability, performance, scalability, operability and maintainability.

Covering all of the three architecture layers, the Security architecture defines the security requirements, which includes i.e. identity management, authentication, authorization, encryption, non-repudiation, etc. at all levels (business, application, data and technology).

Transition to Development

This phase is concerned with the planning the API implementation projects and the establishment of an effective DevOps organization to execute and operate them. The objective of the planning is to provide a consolidated approach, distributing the requirements on projects and development iterations based on business and technical priorities, benefits and dependencies.

Next, the phase is concerned assigning resources with the right skills and experience to the identified projects. The table below only provides a description of key roles and responsibilities that are needed to support API lifecycle. A definitive team structure should be decided together with customer.

Role	Description / Skill	Typical Responsibilities
------	---------------------	--------------------------

API Developers	Deep experience of integration development, API and Agile methodology.	Develop API assets.
API Analysts	Integration and business analysts who are able to understand project -level integration requirements and translate these to C4E assets (and vice versa)	Generate appropriate demand for the API by triaging project requirements into priority self-serve candidates
API Coaches & Evangelists	Experienced in evangelising, provoking and coaching teams	Evangelise and coach teams across customer to think differently and adopt the API-led connectivity approach
API Product Owners / Asset Owners	From Central IT, ETS or the rest of customer-stakeholders the API / Asset Product Owner should understand and apply product management fundamentals to each API or asset	Champion the API, engage the rest of customer stakeholders to reach mass adoption. Keep API operational and optimise through API lifecycle (inception through to deprecation)
API Architect(s)	Deep experience of integration and API architecture - and a thorough understanding of industry trends, vendors, frameworks and practices.	Provide 'enough' governance over the design and operation of the actual API assets. These architects could be part of a wider community, and not just sit within the API initiative.
API Lead / Sponsor(s)	Owner of the API within customer stakeholders. Strategic and operational management and leadership experience.	Manage the overall success of the API initiative, manage operation on a daily basis, measure ROI and performance, manage senior stakeholder and management perception, manage budget and funding.

Implementation Governance

As part of the development effort, the architecture needs to provide governance to the implementation of the API. This includes the necessary quality assurance to the implementation (ensuring that requirements are reflected in the implementation design and actual delivery, ensuring re-use on front-end and (very important) back-end functionality, incorporation of best practices and implementation pattern, ...

Change Management

The implementation of APIs requires an adjustment in the work processes, both internal to the IT and business organization and also to the external communication to the consumers of the API.

The internal change management includes the definition of changed deliverables, adjustment of the development process, and the management of the different development speeds within the IT department. As APIs are following a different integration paradigm, the involved teams need to adjust their deliverables to this change. A main change to the development process is the increased frequency in the delivery of the APIs to the back-end systems.

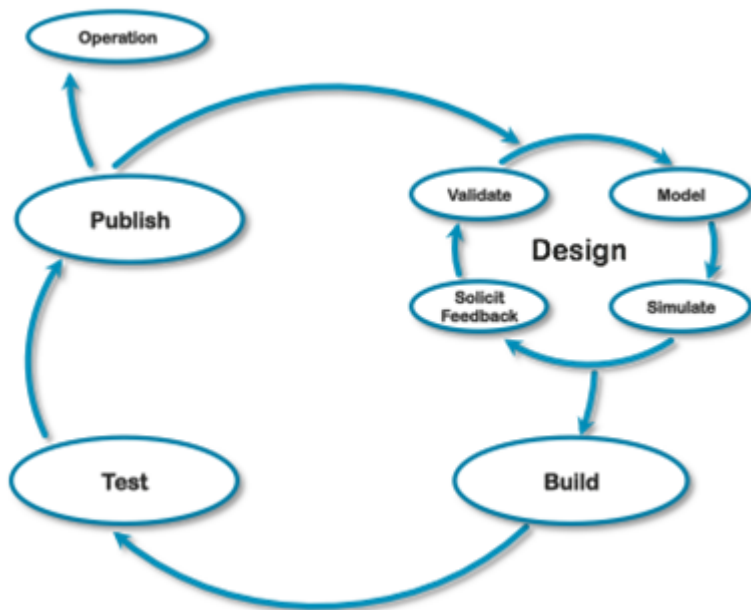
Also important is the management of the outwards looking side of the APIs. Within the change management process the API provider needs to inform the consumer community about new API functionality, changes to the existing functionality with new releases, and the testing functionality (e.g. within the development portal).

Requirements Management

Requirements, such as all artifacts of the design and development process, needs to be managed and controlled. As a best practice, requirements are usually separated into functional and non-functional (technical) requirements. After the compilation and acceptance of the requirements they need to be referenced in the design and also in the definition of the different test cases. The use of a central requirement management system (with an included issue management system) such as Jira is highly recommended.

DevOps Cycle

The development process is best described by a circular process. The circle starts with the design of the APIs, followed by build, test, publishing (deployment), and operations - until the analysis of the APIs acceptance, behaviour, and functional or technical changes restart the circle. This iterative approach invites for the use of circular, agile development approaches.



Design

In the Design Phase the architect defines and specifies the different APIs, the behaviour of those (successful and error completion), and provides examples for the interfaces and the responses. As a result of this phase the API consumer developer of the APIs receive the specification and mock-ups to develop against, the API provider developer receive the framework to develop the backend functionality. APIs are living software artefacts, they exist over time and will get adjusted and modified during this time. The Design Phase within the iterative process needs to take account of this. The deliverables of the phase (RAML and supplementary documentation) needs to be adjusted as the API is modified and adjusted.

More information on this topic can be found in the appendix of this document.

High Level API need

The very first step is the identification of the high level business functionality which needs to be exposed by the different applications. This identification can be done e.g. by discussing given use stories or use cases and the mapping the data and functional ownership from these documents to the applications in the ecosystem.

Identification of APIs and Resources

Following the process flow in the initial use cases it is possible to define the interfaces needed for the different applications. These interfaces and their functionality need to be characterized their resources and their relationships, i.e. by using the business entities and processes owned or managed by the application (example: "add shipping address to customer" in a CRM system identifies the customer as primary entity and the shipping address as a secondary entity related to a particular customer). These identification process might need a number of iterations before reaching the level intended. Within this process, the identified resources and their relations get detailed and revised. At this stage it is also good practice to formalize the identification documentation by building initial specifications of the API using the RAML language.

Specification of the API using RAML

With the identified entities and relations it is now possible to document the API in a RAML document using the AnyPoint Platform. The RAML contains the structure of the API call, the specification of the different return messages, and examples for the consumer developer. The RAML document as a core deliverable of the phase can be used directly in the development and test phase.

Documentation of the API

Additional to the syntactic documentation of the API using RAML, the developer needs supportive documentation. This documentation should contain a description of the different resources, their relationships, implemented functionality, possible API return messages, code samples,

and information for testing and verification implementations. Some of the documentation is inherent part of the RAML document. Additional location for the documentation is the AnyPoint API portal, which provides space for the semantic information mentioned and the location to test the API.

Build

Following the design phase, the build and test phase translates the defined API architecture of the design phase into tested development artefacts.

Using RAML to generate the development framework

After reaching a stable point in the design process, the RAML description of the API is used to provide the development framework. After testing the mock-up implementation of the RAML in the API development environment, the API developer packs the RAML project and downloads it to the AnyPoint Studio. This generates a API implementation framework which is the base for the development of the backend development.

To speed the development of the API consumer and the API producer up, the RAML documentation and mock-up functionality can be made available to the consumer development team. The development of the consumer and the producer are then executed in parallel.

Implementing the functionality using AnyPoint Studio

The recommended tooling for the implementation of the backend functionality is AnyPoint Studio. The RAML download did provide the framework for the implementation of the backend functionality: Access points for the API, functionality to catch errors and exceptions are put in place. The development processes at this stage is focussed on the implementation of the functionality for the API.

Version Control

Like any other development process, the use of a version control system such as GIT is highly recommended. As a best practice, the generated artifacts of the development process, the projects generated and managed by AnyPoint Studio, can easily be moved as projects under version control. As an initial part of the implementation process of the development environment, the existing version control rules (e.g. the rules related to version numbering scheme, the version numbering scheme) can be adopted to also support APIs. For a detailed view on version control also refer to the related section in the document.

Automation of the build process / Continuous Integration

The automation of the build / test / publishing process reduces the manual effort and potential issues and problem related with the process. By using the concepts of Continuous Integration and the related tools (such as Jenkins / Hudson) it is possible to reduce the migration, regression testing, and publication effort and errors related to manual processes.

Test

The testing phase is the second part of the building process of the functionality. It takes the previous generated designs and implements the functionality, resulting in artifacts ready for deployment / publishing.

Unit Test

A successful Unit Test completes the building effort of the developer. The Unit Test should be a documented artifact within the development line, based on the design of the API. A recommended way to execute the Unit Test is to use tools for the design and execution of the test: JUnit for the testing of potential Java components and MUnit for the implementation of Unit Tests in the AnyPoint Studio environment.

MUnit allows the implementation of test flows as part of the AnyPoint Studio implementation of application flows. Each test flow provides reference to the implemented flow, additional assertions, pre-loading of values, and evaluation of return payloads and messages are supported.

As with all good unit tests, the Unit Test is a “white box” test (the test uses the knowledge on the implementation of the functionality). It is important that the test executes all lines of process in the application (full coverage), provides testing with valid and invalid input data, and ensures that all possible endpoints of the flow are reached correctly.

Since the Test Flow is part of the AnyPoint project and is implemented in a separate folder in the development environment, the implementation of the Unit Test is automatically documented and can be repeated (e.g. by the test team) as required.

Integration Test

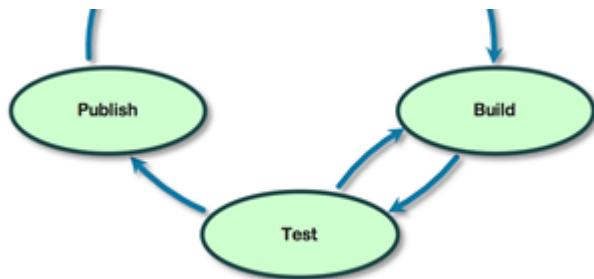
The integration test is a functional and technical black-box end-to-end test. It shows the correct implementation of the different processes and flows. As part of the integration test, the implemented functionality is triggered and calling either the concrete target systems or test environments that are dedicated to perform integration tests.

Regression Test

The regression test is an assurance test. It is a re-test of previous implemented and tested functionality. A regression test should be executed whenever new functionality is implemented and tested. It assures that the existing functionality is not changed as part of new developments or changes. Very often, regression tests are automated tests, to be executed as part of every test cycle.

Build / Test Iteration

Based on the used implementation methodology (e.g. using agile or scrum methodologies), the development and test phases are iterating in faster cycles than the main development process.



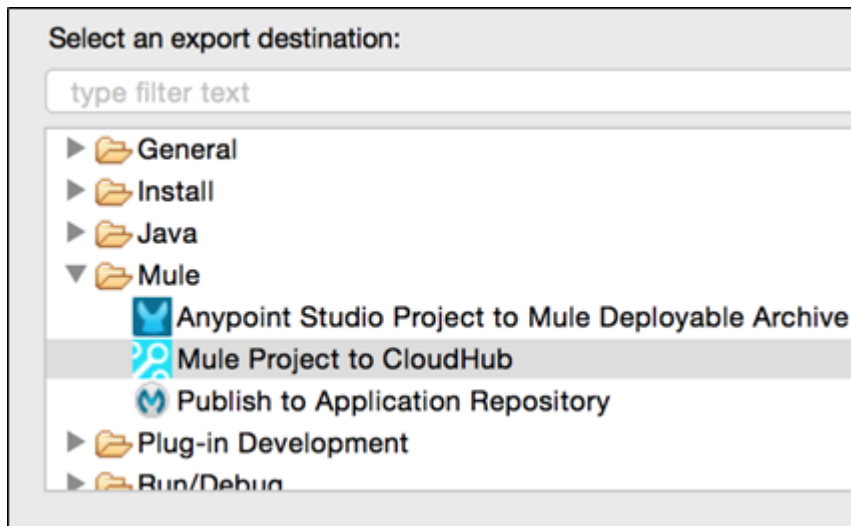
These shorter sprints are selected for in case of fast adoption of changing requirements or a need for early testing of functionality.

Publish / Implement

The tested and accepted artifacts of the development and testing phase are implemented in the pre-production and production environments. The Publication of the APIs in the Pre-Production environment makes them available for the development community outside, in the Production environment allows the APIs to be used as part of business processes.

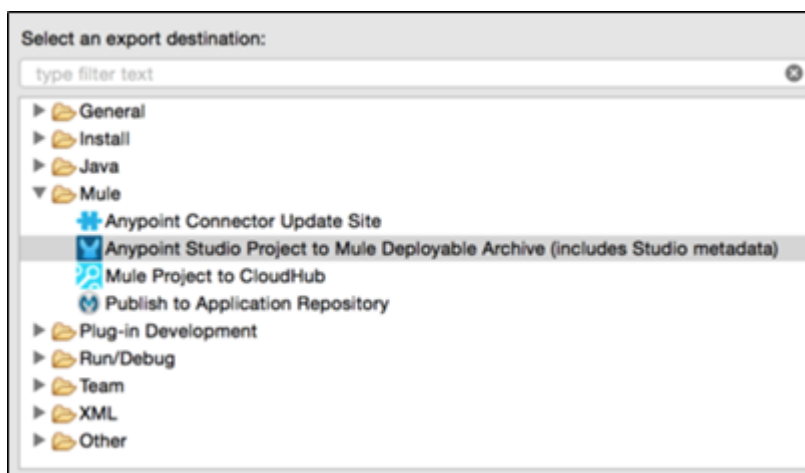
Deployment of APIs

Deployment is possible in two different modes. The developed API can be deployed to CloudHub, the cloud based implementation platform of Mulesoft. Alternatively the artifacts can be deployed to a On Premise implementation of MuleSoft. The selection of the deployment target is part of the architecture design and needs to be determined based on the requirements and organizational policies of the producer. The deployment to the CloudHub location is triggered out of the development stream. When deployed to this location, the deployment environment provides a number of technical services to the API.



The implementation on CloudHub provides a number of advantages towards a local, on-premises implementation. It is important for the stability of the integration and the production environment that the implementation environment is providing a number of features, such as being high-available, sized and being scalable, securely hosted, monitored, and managed. These requirements were observed for the implementation of the service. This reduces the effort of implementing and running additional environments locally.

Alternatively the deployment can be made to an implementation On-Premises.

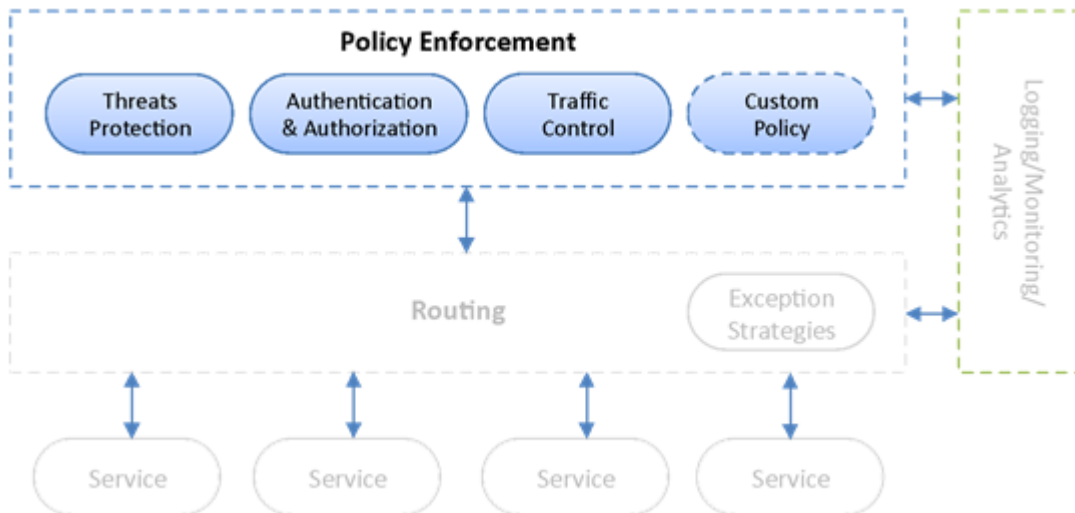


This deployment option provides the installation of the API and the integration on a local installation. Here all technical requirements, such as server shadowing and hot-standby, load balancing, integration to monitoring and alarming, etc. needs to be designed, developed, and implemented locally. These non-functional requirements need to be identified and included in the architecture context at design time.

Since the publication to a pre-production or production environment is usually done past the acceptance of the delivered artifacts, the publication of the API and the related materials are usually in the responsibility of operations or production maintenance. Here the tooling of Continuous Integration (CI) or dedicated publication scripts can automate the publication process and reduce issues caused by manual intervention.

Policy Configuration

Before an API is deployed, policies that govern how it can be used must be defined and implemented. These policies will dictate who can access the API, how users will be authenticated and authorized, and how much traffic they may consume.



Applying Policies to an API

Implementing policies to control access (authentication, authorization, but also SLA and response time) is critical to keeping APIs and the underlying services they leverage managed. Policies e.g. like a limitation on consumption are important to keeping the API performing at peak levels and with the set expectations of the API consumers. Most solutions provide a number of pre-built policies out of the box in a policy library to manage common tasks like rate limiting, throttling, and security enforcement. In addition, they will allow for custom policies to be created.

Applications	Policies	SLA tiers	Permissions			
Name			Category	Fulfills	Requires	
▶ Client ID enforcement	RAML snippet		Compliance	Client ID required		Apply
▶ Cross-Origin Resource Sharing			Compliance	CORS enabled		Apply
▶ Rate limiting			Quality of service	Baseline Rate Limiting		Apply
▶ Rate limiting - SLA based	RAML snippet		Quality of service	SLA Rate Limiting,Client ID required		Apply
▶ Throttling			Quality of service	Baseline Rate Limiting		Apply
▶ Throttling - SLA based	RAML snippet		Quality of service	SLA Rate Limiting,Client ID required		Apply
▶ HTTP basic authentication			Security	Requires authentication	Security manager	Apply
▶ IP blacklist			Security	IP filtered		Apply
▶ IP whitelist			Security	IP filtered		Apply
▶ JSON threat protection			Security	JSON threat protected		Apply
▶ LDAP security manager			Security	Security manager		Apply
▶ OAuth 2.0 access token enforcement	Deprecated	RAML snippet	Security	OAuth 2.0 protected	OAuth 2.0 provider	Apply
▶ OAuth 2.0 access token enforcement using external provider		RAML snippet	Security	OAuth 2.0 protected		Apply
▶ OAuth 2.0 provider	Deprecated		Security	OAuth 2.0 provider	Security manager	Apply
▶ Simple security manager			Security	Security manager		Apply
▶ XML threat protection			Security	XML threat protected		Apply

The Cloudhub platform additionally allows the generation of SLA levels which allows flexible consumption rules by consumer type, and contract.

Add SLA tier


Name *

Description

Approval *

Limits

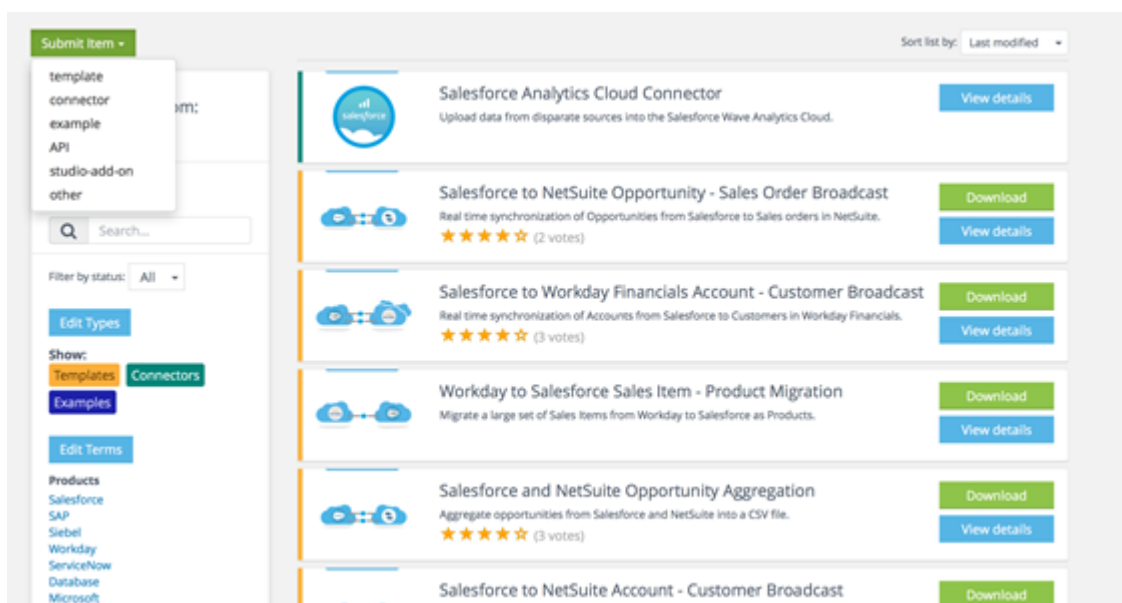
of Reqs * **Per ***

☒ visible 

Publication of an API to Exchange

Exchange is a community environment for the exchange of information, best practices, patterns, and connectors. The entries in Exchange can be made readable to public audiences, but also the user can also use Exchange as a private tenant. In this case the information is only available to an internal team.

By making the API or other artifacts available on Exchange, the environment serves as a central knowledge hub. Additionally to storing the information, the different entries can be tagged and categorized for easy retrieval and reuse.



The screenshot displays the Salesforce Exchange interface. On the left, there is a sidebar with a 'Submit Item' dropdown menu containing options like 'template', 'connector', 'example', 'API', 'studio-add-on', and 'other'. Below this is a search bar and a 'Filter by status' dropdown set to 'All'. The main content area shows a list of connectors, each with a Salesforce logo icon, a title, a brief description, a star rating, and buttons for 'Download' and 'View details'. The connectors listed are:

- Salesforce Analytics Cloud Connector**: Upload data from disparate sources into the Salesforce Wave Analytics Cloud. (2 votes)
- Salesforce to NetSuite Opportunity - Sales Order Broadcast**: Real time synchronization of Opportunities from Salesforce to Sales orders in NetSuite. (3 votes)
- Salesforce to Workday Financials Account - Customer Broadcast**: Real time synchronization of Accounts from Salesforce to Customers in Workday Financials. (3 votes)
- Workday to Salesforce Sales Item - Product Migration**: Migrate a large set of Sales Items from Workday to Salesforce as Products.
- Salesforce and NetSuite Opportunity Aggregation**: Aggregate opportunities from Salesforce and NetSuite into a CSV file. (3 votes)
- Salesforce to NetSuite Account - Customer Broadcast**

Operate

This section of the document describes the operational side of the development cycle. It focusses on the core part of the phase: Monitoring

and Analysis of the behaviour of the APIs and the connected applications.

Monitor and Analyze

As the APIs are productive it is important to ensure that their performance and usage is within the realm of the expectations. Monitoring the running APIs provides visibility of the availability and performance of the APIs, and to ensure that they fulfill the expectations of the consumer community. The analysis (and visualizing the of the results of the analysis) provides important information for the continuous improvement of the installed system and gives input for the design process.

Typical outcome of the Analytics are:

- Detailed metrics on the usage and performance of the APIs, including custom reports on the user community (the actual user of the API) and the performance of the interfaces
- Information regarding consumers (developer and applications) - which can be aggregated to a consolidated view on the usage of the API
- Forecasting on business and technical level to provide input for future investments.

It is best practice to plan the different level of reporting and monitoring during the design phase. This allows to implement the necessary reports and functions from an early stage of the use of the APIs.

Notification and Alarming

To ensure the performance of the APIs and related applications in a productive environment it is important that exceptional situations or occurrences of predefined situations are brought to the attention of the responsible team. iPaas provides two distinct mechanisms used by deployed applications to report on their status and situation: Notification and Alarming.

- Notification are standard messages, which are generated during the execution of the application and appear in iPaas.
- Alerts report on abnormal situations. Based on the configuration they are targeted to provide information on the situation, i.e. by email.

iPaas provides several alert types out of the box:

- Performance: a defined number of events exceeded the time period for processing
- Deployment: a new deployment in the monitored environment as been completed successful or in error
- Connectivity: a secure data gateway has been connected or disconnected
- Problem: iPaas encounters a problem either with a worker or with an application which is monitored by the worker monitoring system.

A standard alert creates notification on console and an alert action (an email or a message into a monitoring system). Additionally Custom Alerts can be generated by applications. These are triggered by notifications send to the iPaas console by the application.

Insights provided by iPaas

To allow more detailed information on the behaviour of transactions and to analyse situations that caused alarms or notifications, iPaas provides inside information. These help to identify the behaviour of a particular transaction, it's occurrence, result, and (in cause of a non-planned situation) the location of the issue.

API Guidelines

API URI / URL

Design 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 life cycle 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.	mytaxi.
{region}	Required. The region of the API	.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

The Base URI is defined within the RAML and takes the form:

`https://{env}.[access].[company].[region]/[context]/[version]`

The API resources are defined relative to the base URI.

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 is query parameter that is used to implement filtering.

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 design 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 prefix of each field. For example, the flowing request will return all purchase orders sorted by data (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 design to take a list of fields as a query parameter, and return only the fields that are includes 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.

Design API Request and Response representation(s)

This step involves specifying the format of the API's request and response messages. A good starting point would be to investigate whether there is an existing, standard format and media type that matches the API use cases and requirements. For that purpose use IANA .

If there is no standard media type and format, try to use as much as possible extensible formats such as JSON (application/json) and XML (application/xml), preferably JSON.

Further, to increase the interoperability, use standards such as W3C XML Schema, ISO 3166, ISO 4217, RFC 3339 to represent dates, times, numbers, currencies, countries.

Last but not least, use multipart media types such as multipart/mixed, multipart/related, or multipart/alternative to encode request or response that contain mix of textual and binary data, that is, avoid encoding binary data within textual formats using Base64 encoding.

Design API Request and Response headers

- Use Content-Type, Content-Length, Content-Encoding and Content-Language to provide additional metadata that helps to interpret the content of the API request and response messages.
- Use Location header to include a link to the current resource or a collection or to provide a link to a queue (in the case of asynchronous scenario), for example, for example

Location: <http://www.example.org/users/1>

- Use Last-Modified header to indicates the last time resource has been modified
- Use ETag to represent the a specific "version" of the resource. Clients may choose to save an ETag header's and use it in future GET requests, as the value of the conditional If-None-Match request header. If the REST API concludes that the entity tag hasn't changed, then it can save time and bandwidth by not sending the representation again.
- Use Cache-Control, Expires, and Date response headers to encourage caching.
- Use Authorization to implement authentication and authorization

Design Response Codes

HTTP status codes are grouped into five numeric categories:

- 1xx – informational
- 2xx – successful
- 3xx – redirection
- 4xx – client error
- 5xx – server error

The appendix of this document contains a complete list of the return codes defined by the W3C consortium. These response codes should be used as standard, the use of not defined return codes is discouraged and should only be done in exceptional circumstances.

The following table shows the most commonly used response codes and a short description of the meaning.

Response Code	Definition of the Response Code
100	CONTINUE: Indicates that the HTTP Server expects the client to continue with the request. It is used to inform the client that a part of the request has been received by the server, but the transaction is not complete yet.

200	OK
201	Created: The request has been fulfilled and the resource has been created. The newly created resource can be retrieved by the returned URI in the response message. If the request from the client cannot be completed immediately and completely then the return code 202 (Accepted) should be used.
202	Accepted: The request has been received, but the processing cannot be immediately processed. If the request is asynchronous, no additional return code is send.
300	Multiple Choices
302	Found
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error: The request generated an internal error on the server side. This does not imply that the request is erroneous – the error shows a problem on the server itself.

DevOps

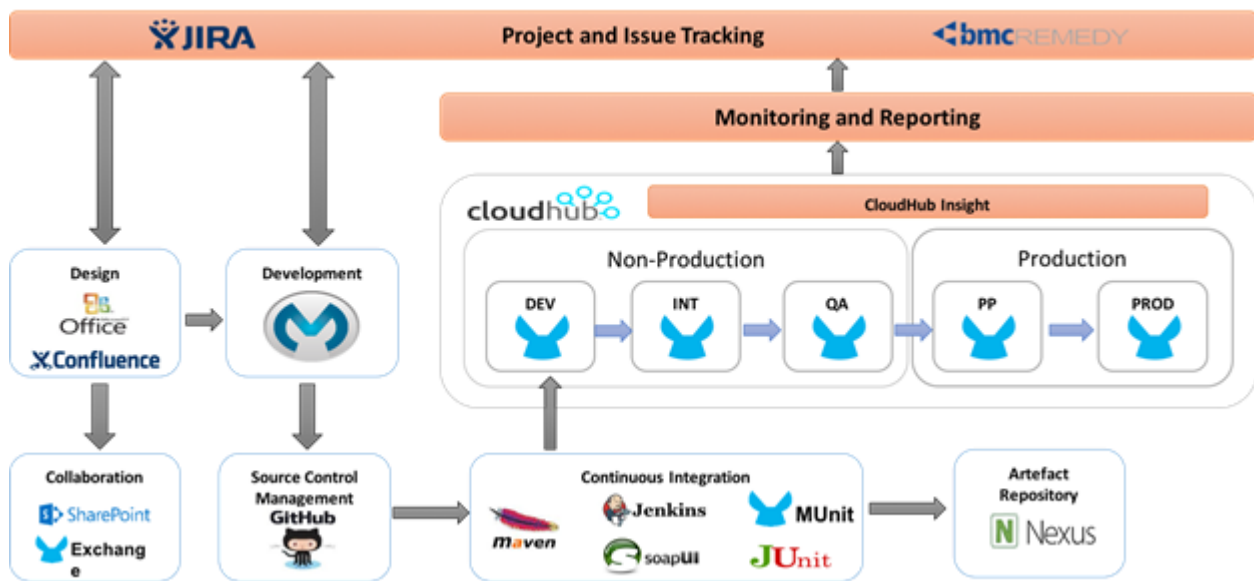
DevOps is the latest trend in software development. It focuses on partnerships between developers and operations staff supported by automation techniques and tools to achieve a quick feedback and enable an end-to-end streamlined, rapid, and repeatable release cycle.

There are multiple benefits from **DevOps**

- Reduction of the operational costs. For example, using DevOps and automation, Facebook achieves ratios of one administrator supporting 20.000 servers
- Increased business agility. According to Puppet Labs, high-performing organizations are deploying code 30 times more frequently, with 50% less failures than their lower-performing competitors.

There are number of prerequisites for DevOps to succeed in practice, such as

- Change of Organization Culture to promote closer and better relationship between the IT development team (concerned with delivery or enhancing of APIs and applications) and the IT operation teams (accountable for API and application availability and operation)
- Use of Continuous Integration and Continuous Delivery practices to implement responsive, reliable and efficient software configuration, change and release management;
- Use of Continuous Testing and Test Automation to allow comprehensive testing to be performed on every change made;
- Use of Agile delivery techniques to support frequent software releases;
- Use of Version control for all software artefacts, including code, data and configuration
- Use of Application architecture that supports automated releasing, scalability, feature control, automated testing, instrumentation and operability;
- Use of Infrastructure



Activity	Description / Skill
Development	Developer uses Anypoint platform tools to develop the API
Unit testing	Developer uses testing frameworks such as SOAP-UI, MUnit , JUnit, JMeter to test the functionality od the API that is been developed.
Quality control	Developer reviews the configurations and code with peers and incorporate the reviewer comments
Local Build	Developer makes sure that all the latest code from repository is synced to local system. Next, using the Maven POM file the developer builds the full project to verify whether the local build is successful
Check In	If the build and the unit testing is successful, developer checks in the configurations and the code to GitHub
Poll SCM	Jenkins will be configured to listen to GitHub. Any time when a configuration or code is checked in or a Maven dependency is changed, a new build is triggered and the respective automated test are executed.
Publish	If the build and the tests are successful, the application is published to Nexus Repository
Deploy to Sandbox	CloudHub Command Lines Tools are used to deploy the applicaiton to the DEV environment. The deployment will be automated using Jenkins. Automated Smoke Test are then performed once the application is deployed to all pre-production sandboxes.
Promote to SIT	Once the testing is successful and there are no errors, the application is published to SIT using CloudHub Console. It is recommended to manually do this step as it gives control of when the application is to be published to SIT. Smoke test are performed to validate the SIT build and build notification are sent to the Testing Team (successor failure).
Release to PORD	Release Management decides whether the build should be promoted to Production for Go Live.
Auditing and Reporting	CloudHub Insights is used to track and monitor the transactions. Log Analyser can be used to analyse the logs. This process involves offloading the logs from cloud to on premise on a regular basis.
PGLS	Post Go Live Support is done by the Development team, typically for a period of one month.
AMS	The application is handed over the Maintenance team

API Patterns

Endpoint Redirection

Problem

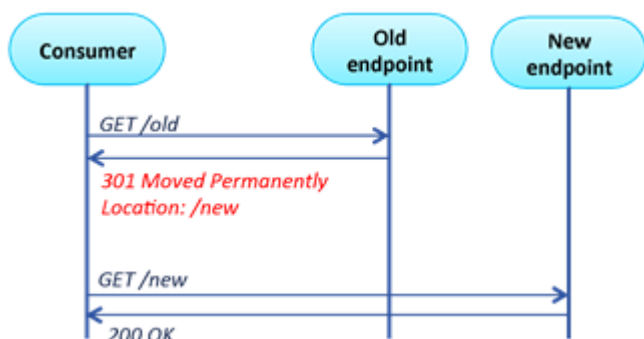
A service provider may change the endpoint of its service over time for business or technical reasons. It may not be possible to replace all references to the old endpoints simultaneously.

Solution

Automatically refer service consumers that access the old endpoint to the new one. HTTP natively supports this pattern by using a combination of 3xx status codes and standard headers:

- Code 301 Moved Permanently
- Code 307 Temporary Redirect
- Header Location: /new

Redirection responses can be chained, however, be careful not to create redirection loops



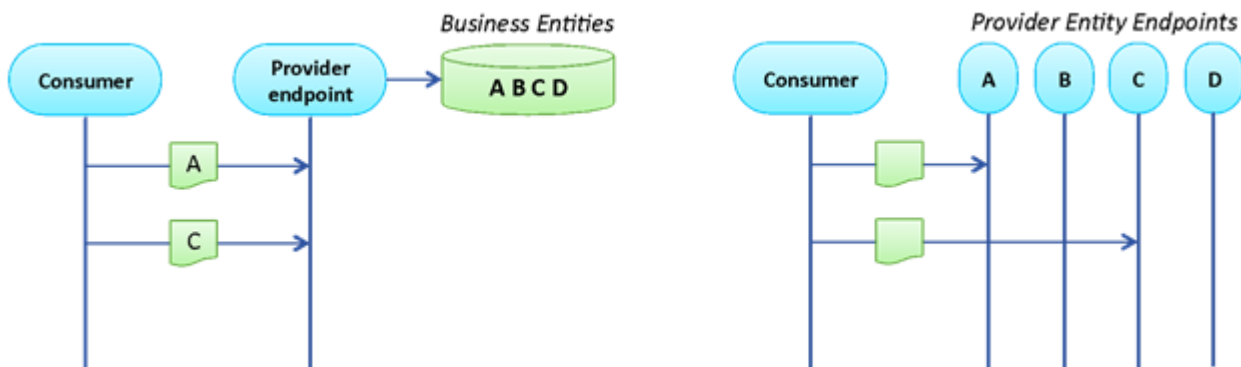
Entity endpoints

Problem

A service with a single endpoint is too coarse-grained when it provides capabilities to manage multiple data entities. A consumer needs to work with two identifiers: a global one for the service and a local one for each business entity. Entity identifiers cannot be reused and shared among multiple services.

Solution

Expose each entity as individual lightweight endpoints of the service they reside in. The benefit of this approach is global addressability of business entities.



Content negotiation

Problem

Service consumers may change their requirements in a non-backwards compatible way. A service may need to support both old and new consumers without having to introduce a specific capability for each kind of consumer.

Solution

A service capability could be negotiated at runtime and based on the outcome, a specific content and data representation formats are returned by a service. The service contract refers to multiple standardized “media types”. The benefits of this solution are loose coupling and increased interoperability

GET /resource

Accept: text/html, application/xml, application/json

The client lists the set of understood formats (MIME types)

200 OK

Content-Type: application/json

The server chooses the most appropriate one for the reply (status 406 if none can be found)

Quality factors allow the client to indicate the relative degree of preference for each representation (or media-range) using

Media/Type; q=X

If a media type has a quality value q=0, then content with this parameter is not acceptable for the client.

Accept: text/html; q=0.1 application/xml; q=0.9

The client prefers to receive xml, and html as fall back.

Request Header	Example Values	Response Header
Accept:	application/xml, application/json	Content-Type:
Accept-Language:	en, fr, de, es	Content-Language:
Accept-Charset:	iso-8859-5, unicode-1-1	Charset parameter for the Content-Type header
Accept-Encoding:	compress, gzip	Content-Encoding:

Idempotent Capability

Problem

Service oriented architectures are distributed systems. Failures (such as the loss of messages) may occur during service capability invocation. A lost request should be retried, but a lost response may cause unintended side-effects if retried automatically.

Solution

Use idempotent service capabilities, whereby services provide a guarantee that capability invocations are safe to repeat in the case of failures that could lead to a response message being lost.

Idempotent

Idempotent requests can be processed multiple times without side-effects

GET /book

PUT /order/x
DELETE /order/y

If something goes wrong (server down, server internal error), the request can be simply repeated until the server is back up again.

Safe

Safe requests are idempotent requests that do not modify the state of the server (can be cached). In contrast, unsafe requests modify the state of the server and cannot be repeated without additional (unwanted) effects. Unsafe requests require special handling in case of exceptional situations (e.g., state reconciliation).

POST /order/x/payment

In some cases the API can be redesigned to use idempotent operations:

GET /account/1/balance //safe
New Balance = Balance + 200\$ //calculate on the customer side
PUT /account/1/balance (New Balance) //idempotent

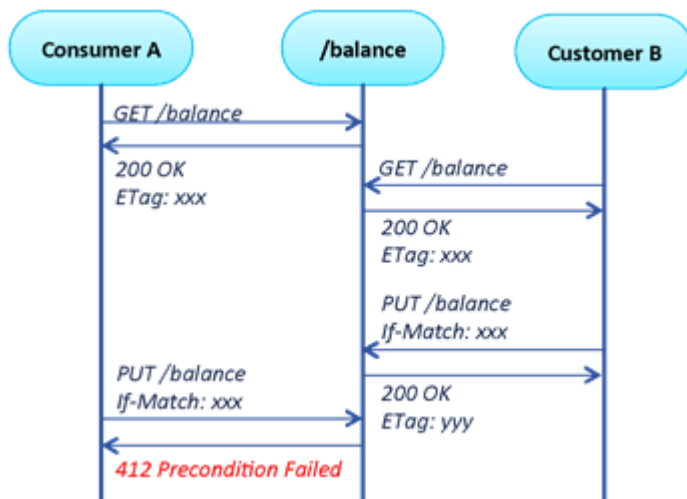
Concurrency

Problem

Breaking down the API into a set of idempotent requests helps to deal with temporary failures. Problem arises when another client concurrently modifies the state of the resource that is to be updated.

Solution

ETag and If-Match headers are used to implement optimistic locking. Code 412 Precondition Failed can be used to inform the customer that she works with out-of-date resource.



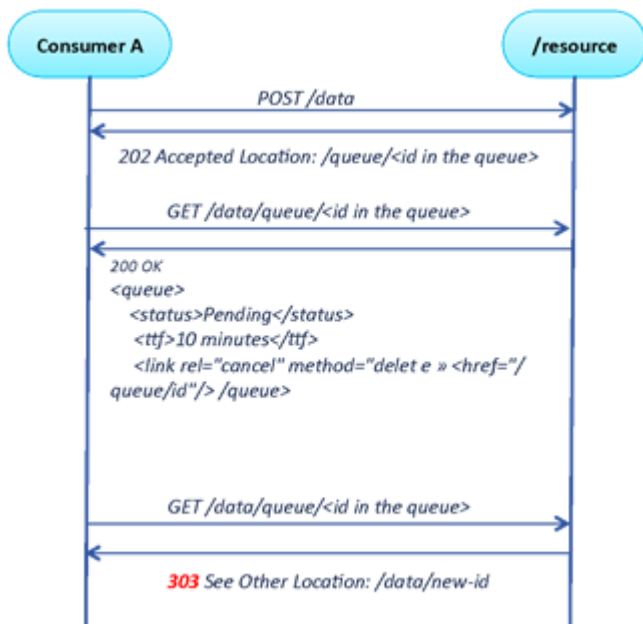
Asynchronous processing

Problem

Rest is based on synchronous protocol (HTTP). Through API, we may trigger asynchronous work or process that we want to follow state and track termination to better chain API or service call.

Solution

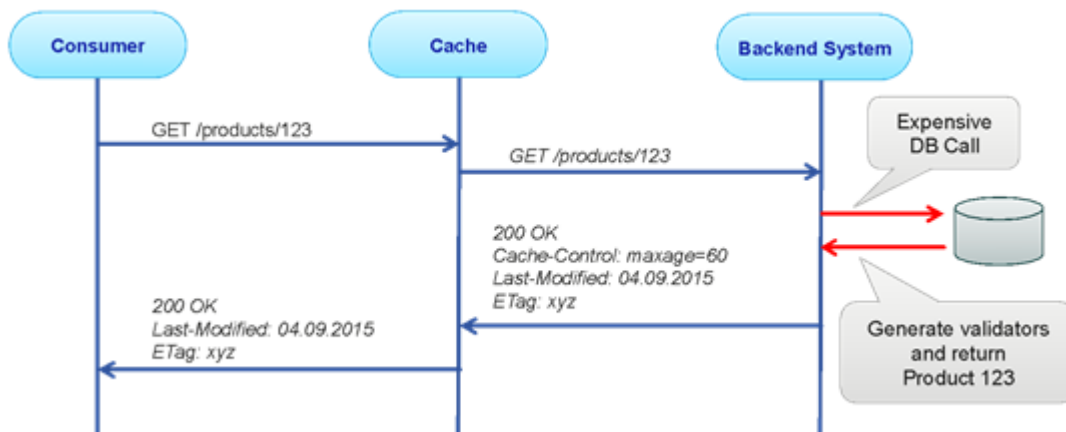
Asynchronous processing is based on HTTP code 202 Accepted, HATEOS for process management and status and client async checking (for instance, AJAX method in JavaScript). The asynchronous process could be managed by checking regularly the process URL given as a response to the (async) service request. The process service can give time to finish indication, status and link through HATEOS on cancel or other subsequent actions available on the process.



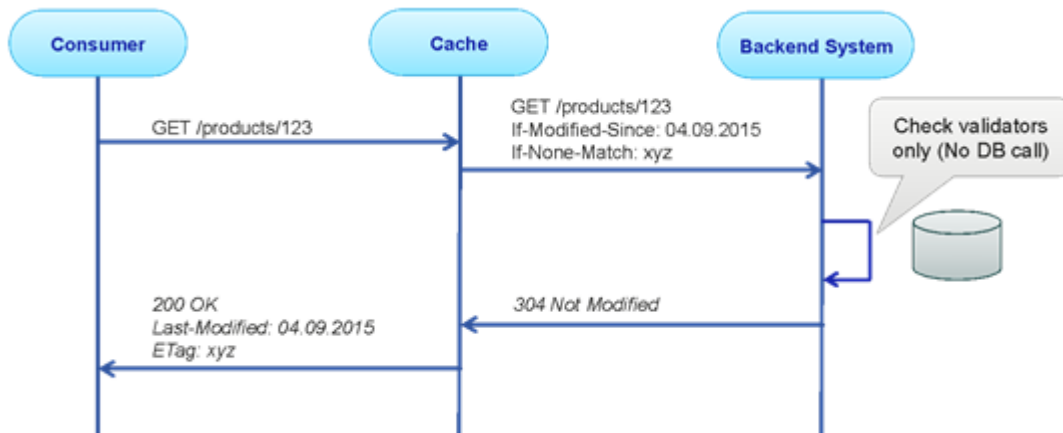
Caching

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:

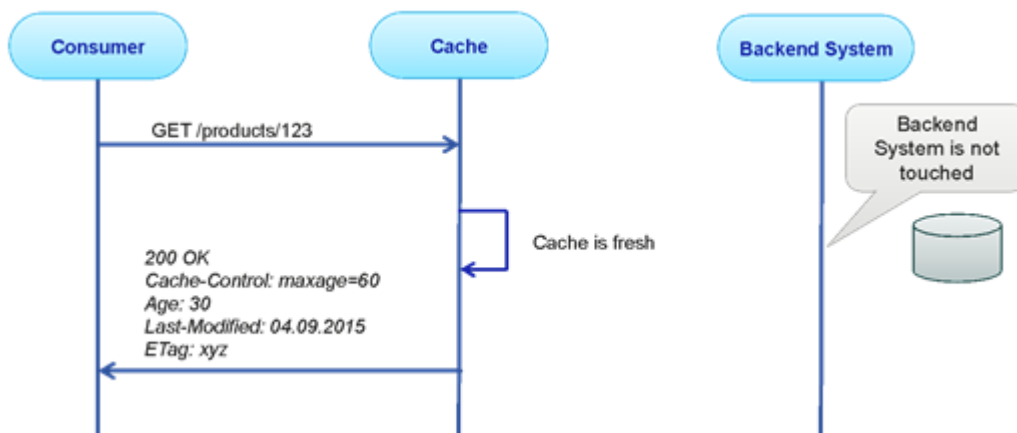
The Consumer requests information on the product 123 from the product API. The Cache does not have valid information on the product in the store and sends the request to the back-end system which retrieves the information. As this is the first request of the information the Cache does not keep the information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

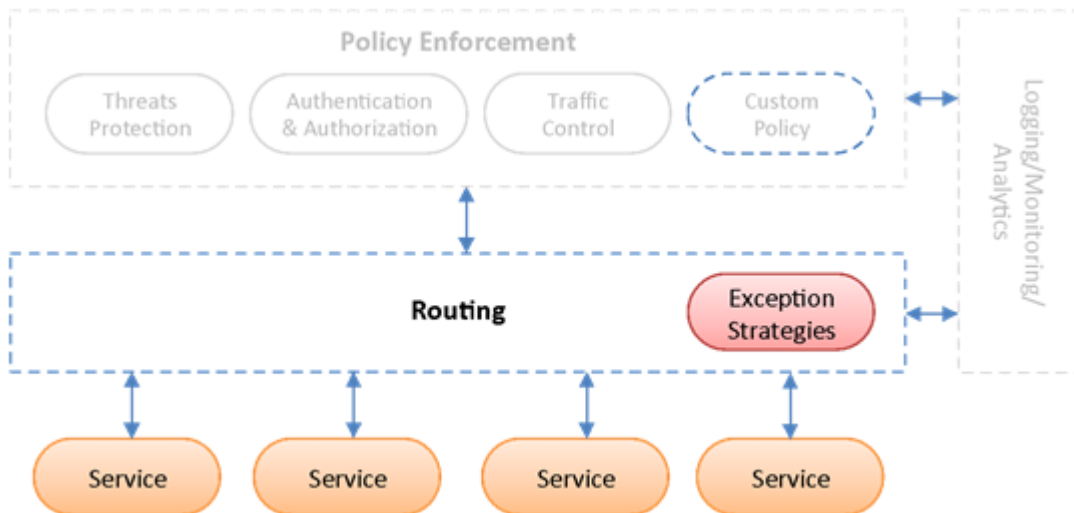
API Management

API Implementation

Once designed, the API needs to be built by connecting to the backend services or applications that will power it. If the API will be connecting to existing web services, this connection will be a simple proxy and should be easy and fast to configure.

However, if data needs to be orchestrated across multiple systems on the backend, transformed to a new format, or if the backend system is legacy, or otherwise difficult to connect to, the API build is more complex. In this case, integration and orchestration capabilities are required. Be careful not to write custom code in the API layer to perform these tasks as that is a short-sighted solution that will make your API brittle and difficult to manage and maintain over time.

Leverage a fit-for-purpose integration solution here or choose a gateway that delivers these capabilities natively.



MuleSoft provides a set of open source Maven and Mule Studio-based tools called APIKit that enable developers to be massively productive in creating well-designed REST APIs. It allows Anypoint Studio to import a RAML file and automatically generate the following items:

- A Main flow with an HTTP endpoint, an APIkit Router, and an exception strategy reference
- Skeletal Backend flows, one for each resource-action pairing in the RAML file
- Several Global Exception Strategy mappings

API Versioning

An API should be designed for long-term, but since change is inevitable and the API will never be completely stable. What is important is to manage the changes well through comprehensive versioning strategy. This includes multi-month deprecation schedule and documentation that is up to date and describes the change in sufficient level of detail. Not all changes require new version. API's designed for backwards compatibility do normally not require new versions for minor changes and is recommended practice to design for this. Nevertheless, it is not always possible the design APIs in a way. The table below outlines when to introduce a new version and when a new version is not required.

Type of Change	Requires a new version	New version NOT required
Service contract	<ul style="list-style-type: none"> • Remove API endpoint • Change effect of API operation • Change response type of API operation • Add new required operation parameter 	<ul style="list-style-type: none"> • Add new API operation
Data contract	<ul style="list-style-type: none"> • Remove existing element (or attribute) • Add new required element (or required attribute) • Change existing element (or attribute) 	<ul style="list-style-type: none"> • Add optional element (or attribute) • Add derived element type
Representation format	<ul style="list-style-type: none"> • Remove existing representation 	<ul style="list-style-type: none"> • Add new representation
Accessibility	<ul style="list-style-type: none"> • Restrict permissions 	<ul style="list-style-type: none"> • Relax permissions

Besides the types of changes listed in the table above, there are some special considerations when to introduce a new version or not:

Change in quality of service (QoS). When a change affects qualities of service, such as response time and availability, the API version might need to be change as the change in the QoS might break the applications that use the API.

Changing assumed domain of values in the output message. Applications that use the API may have an expectation about the domain of values of certain fields in the response messages. For instance, the API might only return products of a certain category, or specific payment methods. If the API starts returning values outside that assumed domain, it will likely have an adverse effect on the existing consumers. In this case, consider introducing a new API version.

Major and Minor versions

As part of the standards introduced with the versioning of the APIs, there should be well defined rules of impact of the change of version numbers:

- Major versions: introduce a change in the structure of the API that require the user of the API to adopt the interface on the consumer side
- Minor versions: introduce changes to the API that do not require the API user to change, or contain bug fixes.

Using these rules allows to define the following naming conventions:

- Specify the version with a “v” prefix.
- Move it all the way to the left in the URL so that it has the highest scope.
- Use a simple ordinal number, e.g., “v1”, “v2”, and so on.
- Avoid using dot notation, e.g., “v1.2”

Eg.

<http://www.example.org/v1/customer/1234>
<http://www.example.org/v2/product/1234>

In general, avoid more than 2 versions of the API at the same time. Declare the old version of the service deprecated, update the developer’s portal and notify the consumers. Allow time to for service customers to implement and test against the new service interface and functionality before decommissioning the old service.

<< RAML Example has to be provided.. need to look into it later : Prashanth NC >>

API Deployment

<To be documented>

API Security

Like all gateways to the outside world, APIs need to be secured and monitored. Securing the APIs includes that the user of the API can be identified, authenticated, and the authorization of the API’s use can be verified. But it is also important to understand that the success of the implemented APIs is depending on the ease of the use of the API: If the complexity of the security gateways is to high, the API will be rejected by the user community.

With the use of industry standard authentication protocols it is possible to reduce the effort of securing the API whilst allowing the use of the API with limited effort. Using an accepted mean of securing the interface allows to establish a sufficient level of security while also allowing the API to be used with limited extra effort. It is part of the design process to establish the “right” level of security for the API. The use of custom security protocols is discouraged as

- the effectiveness of the protocol has to be ensured (industrial accepted protocols have been reviewed and accepted by a wide range of security experts),
- the protocol needs to be maintained covering an evolving number of security threads, and
- the user of the API has to adopt to a special protocol.

In the following section a number of standard protocols and standards are described which could be used to secure the APIs:

Supporting OAuth2

OAuth2 is an open standard for authorization by providing client applications a secure delegated access to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their

credentials.

Designed specifically to work with HTTP, OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The client then uses the access token to access the protected resources hosted by the resource server.

OAuth2 is commonly used as a way for Internet users to log into third party websites using their Microsoft, Google, Twitter, etc. accounts without exposing their password.

A lot of libraries and implementations exist for most of the popular providers to help you or your clients to consume your secured API with OAuth2.

Supporting OpenID Connect

OpenID Connect is a simple authentication layer on top of OAuth2, an authorization framework. It allows computing clients to verify the identity of an end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user in an interoperable and REST-like manner.

OpenID Connect performs many of the same tasks as OpenID2, but does so in a way that is API-friendly, and usable by native and mobile applications. OpenID Connect defines optional mechanisms for robust signing and encryption.

API Documentation

As APIs are shared resources inside and outside the organization they belong to, it is essential to provide adequate documentation for them. In the next sections, suggestions for the documentation of the APIs are presented. The list is not complete and would need to be adjusted to the documentation standards of the environment in which the APIs are used.

Provide machine-readable JSON schema

One of the main reasons for communication issues using APIs are human errors during the development process. To improve the quality of the development process the schema specification for the exchange data should be provided in a complete and machine readable form.

Provide human-readable documents

As API is only as good as its documentation, the docs should be easy to find and publically accessible.

The document should show the following information:

- Authentication, including acquiring and using authentication tokens
- API stability and versioning, including how to select the desired API version
- Common request and response headers
- Possible returned errors
- Examples of complete request/response

Once you release a public API, you've committed to not breaking things without notice. The documentation must include any depreciation schedules and details surrounding externally visible API updates.

Using modeling languages

Like WSDL for SOAP services, it can be useful to use modeling languages to describe your RESTful API before to develop it, which provides all the information necessary to describe RESTful APIs.

It encourages reuse, enables discovery and pattern-sharing, and helps to apply best practices. It allows generating automatically documentation, mocked services and proxy client.

Useful Guidelines

Meaningful Error Message

Scenario: *When APIs implement business logic*

Many times we come across situations where developer has to write code to include certain business logic within API, an example may be a 'conditional transformations'.

They are bound to handle exception around the logic and when you do that make sure you provide a valid error message back to the caller with below mentioned error detail so that the caller will take a necessary action.

This example considers JSON as the return message.

Meaningful Error Message for APIs

```
{
  "errorMessage":{
    "apiName":"","
    "apiVersion":"","
    "path":"","
    "clientId":"","
    "errorId":"","
    "errorMessage":"","
    "suggestion":""
  }
}
```

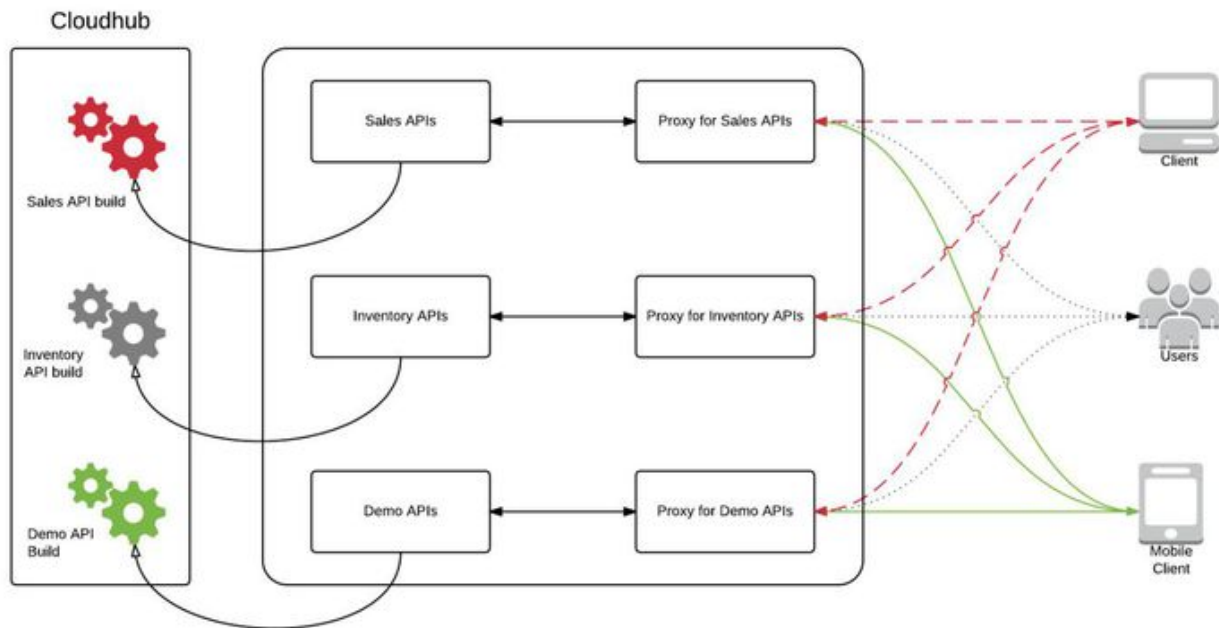
apiName:	Name of the API, this can be descriptive or unique API id
apiVersion:	Build version of API
path:	Typical http path defined for API call
clientId:	Client id as in trait 'client_id'
errorId:	Custom Error ID
errorMessage:	Appropriate message that caused exception
suggestion:	Action for the caller of this service

This can be mentioned as a part of API documentation in RAML.

Clean Proxies

Scenario: *When there is a need to create multiple proxies*

Always create proxies for API that can be logically grouped. Meaning, when creating proxies be mindful to route API requests whose business operations are similar in nature. This way the management for API and their associated proxies is more efficient and their changes are predictable. Refer to below mentioned diagram.



Grouping Similar APIs (in one project)

Scenario: When there is need to create multiple APIs

Group APIs whose functionalities are similar. In other words create API projects using Anypoint studio whose business operations are similar. This helps manage API and their deployment efficient. Refer to aforementioned diagram.

Do not clutter APIs with more methods

Scenario: When there is a need to implement more methods

When creating API be mindful to create appropriate methods for that API. Do not clutter with more methods in single API. If needed create more APIs and accommodate, logically group the methods in multiple APIs.

Glossary

Headers

Header	Description
Accept	Content-Types that are acceptable for the response, e.g., application/json
Accept-Charset	Character sets that are acceptable for the response, e.g., utf-8
Accept-Encoding	List of acceptable encodings, e.g., gzip
Accept-Language	List of acceptable human languages for response, en-US
Content-Type	The MIME type of the body of the request (used with POST and PUT requests), e.g., application/json
Date	The date and time that the message was sent, Mon, 15 Dec 2014 08:15:30 GMT
Authorization	Authentication credentials for HTTP authentication, e.g., Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
If-Match	Tells the API to only perform the action if the client-supplied entity matches the same entity on the server. This is mainly used in PUT to only update a resource if it has not been modified since the user last updated it, e.g., If-Match: v1

Pragma	Implementation-specific fields that may have various effects anywhere along the request-response chain (e.g., no-cache)
Cache-Control	Used to specify directives that must be obeyed by all caching mechanisms along the request-response chain (e.g., no-cache)
Connection	What type of connection the user-agent would prefer (e.g., keep-alive)
ETag	An identifier for a specific version of a resource (e.g., ETag: v1)
Expires	The date/time after which the response is considered stale, e.g., Mon, 15 Dec 2014 09:20:20 GMT
Last-Modified	The last modified date for the resource, e.g., Mon, 15 Dec 2014 10:10:30 GMT

Status Codes

This list is based on the W3C response code list. Some often used status have been added (and marked accordingly). As a best practice those responses should not been re-used or overwritten.

Code	HTTP Method	Response Body	Description
200 OK	GET, PUT, DELETE	Resource	There are no errors, the request has been successful
201 Created	POST	URI of the resource that has been created	The request has been fulfilled and resulted in a new resource being created
202 Accepted	POST, PUT, DELETE	An URI of a resource which represents the processing status	The request has been accepted for processing, but the processing has not been completed
204 No Content	GET, PUT, DELETE	N/A	There are no errors, the request has been processed and no content is expected in the body (by design)
304 Not Modified	conditional GET	N/A	The resource has not been modified : there is no new data to return
400 Bad request	GET, POST, PUT, DELETE	Error message	The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications
401 Unauthorized	GET, POST, PUT, DELETE	Error message	The request requires user authentication
403 Forbidden	GET, POST, PUT, DELETE	Error message	The server understood the request, but is refusing to fulfill it. For example, Authentication failure or invalid Application ID. Authorization will not help and the request SHOULD NOT be repeated
404 Not Found	GET, POST, PUT, DELETE	Error message	The server has not found anything matching the request URI
405 Method Not Allowed	GET, POST, PUT, DELETE	Error message	The method specified in the request is not allowed for the resource identified by the URI
406 Not Acceptable	GET, POST, PUT, DELETE	Error message	The request contains parameters that are not acceptable

408 Request Timeout	GET, POST, PUT, DELETE		
409 Conflict	PUT, DELETE	Error message	The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request
410 Gone	GET, POST, PUT, DELETE	Error message	Used to indicate that an API endpoint has been turned off. Could be used to deprecate API, for example, to inform the customer that the API will soon stop functioning and to migrate to new version of the API
412 Precondition Failed	GET, PUT	Error message	The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server. This response code allows the client to place preconditions on the current resource metainformation (header field data) and thus prevent the requested method from being applied to a resource other than the one intended
415 Unsupported Media Type	GET, POST, PUT	Error Message	The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method
429 Too Many Requests	GET, POST, PUT, DELETE	Error message	Indicates that the user has sent too many requests in a given amount of time ("rate limiting")
500 Internal Server Error	GET, POST, PUT, DELETE	Error message	The server encountered an unexpected condition which prevented it from fulfilling the request
502 Bad Gateway	GET, POST, PUT, DELETE	Error message	The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request
503 Service Unavailable	GET, POST, PUT, DELETE	Error message	The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay
504 Gateway Timeout	GET, POST, PUT, DELETE	Error message	The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (e.g. HTTP, FTP, LDAP) or some other auxiliary server (e.g. DNS) it needed to access in attempting to complete the request
509	GET, POST, PUT, DELETE	Error Message	Bandwidth Limit Exceeded (not included in the W3C standard, implemented as part of the Apache tooling)
510 Not Extended	GET, POST, PUT, DELETE	Error message	The policy for accessing the resource has not been met in the request. The server should send back all the information necessary for the client to issue an extended request
511	GET, POST, PUT, DELETE	Error Message	Network Authentication Required (not included in the W3C standard)
550	GET, POST, PUT, DELETE	Error message	Permission denied (not included in the W3C standard)
598	GET, POST, PUT, DELETE	Error Message	Network Read Time-Out Error
599	GET, POST, PUT, DELETE	Error Message	Network Connect

