**SOA 2.0**

**API Design Guidelines**

Prepared by : Cognizant

Last Review : October, 2017

Version : 0.2

| **CHANGE HISTORY** | | | | |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Description of Changes** | **Remarks** |
| 0.1 | 23rd Oct, 2017 | SOA COE | Initial Draft of document |  |
| 0.2 | 11th Dec 2017 | SOA COE | Updated Design Overview section |  |

**Table of Contents**

[1. Introduction 3](#_Toc501013066)

[2. Acronyms & Definitions 3](#_Toc501013067)

[3. API Design Overview 4](#_Toc501013068)

[4. API Design Principles 5](#_Toc501013069)

[5. API Identification / Discovery Approach 6](#_Toc501013070)

[6. API Design Specification 7](#_Toc501013071)

[7. API Management using APIGEE platform 8](#_Toc501013072)

[8. API Implementation Best Practices & Guidelines 9](#_Toc501013073)

# Introduction

Purpose of this document is to establish and record the key characteristics, design principles, guidelines and identification strategy for API discovery, design specification and implementation. This document will help SOA 2.0 factory teams to enforce best design practices while working on different APIs.

The intended audiences for this document are –

* Solution Architects
* API Designers
* DevOps Leads
* Developers
* Testers

**NOTE – This is a living / evolving document. It will be continuously updated with more contents as and when we learn more and make progress into the API design activities.**

# Acronyms & Definitions

|  |  |
| --- | --- |
| **Acronym** | **Definitions** |
| API | Application Programming Interface |
| REST | Representational State Transfer |
| URI | Uniform Resource Identifier |
| SWAGGER / Open API | API specification standard |
| MSA | Microservice Architecture |
| SOA | Service Oriented Architecture |
| PCF | Pivotal Cloud Foundry |
| APIGEE | API Management & Gateway Platform |
| HATEOAS | Hypermedia as the Engine of Application State |

# API Design Overview

API stands for Application Programming Interface. There are many different types of APIs like Program-centric APIs (JARs / DLLs etc.), Web APIs (RPC style, SOAP based), and also the popularly used REST APIs. In Microservices context, we only deal with REST APIs.

REST is an architectural style which is primarily designed to work with HTTP protocol. Its core principle is to define named resources that can be manipulated using a small number of methods. The resources and methods are known as nouns and verbs of APIs. With the HTTP protocol, the resource names naturally map to URLs, and methods naturally map to HTTP methods GET, POST, PUT, DELETE etc.

Microservices design goal is to create a set of decoupled, self-contained, independent and small systems working together to realize a business process. In order to implement business functions / processes, it is important that Microservices are exposed as REST APIs so that they be interfaced in a loosely coupled manner.

REST APIs are in use even before Microservices concept is introduced to us. REST API design goal is to ensure that **every object (resource) has a uniform interface which is exposed as HTTP endpoints.** All requests for that object should always be made using the exposed API interface.

To summarize – Microservices and REST APIs works together but solve different problems and hence should be exclusively designed. Microservices design is usually driven by domain model (bounded context) whereas the API design considers many aspects including – user / customer experience, granularity and aggregation requirements, re-usability, functional coverage etc.

Designing an API means providing an effective (HTTP) interface that helps the API consumers to better understand, use and integrate with them while helping the administrators to maintain them effectively by following open standards and generating explicit documentation / user manual.

An API design presents a solid blueprint of what your API wants to achieve and provides a comprehensive overview of all the endpoints and CRUD operations associated with each of them. Clearly defined API designs help the team to know about the role of each resource / sub-resource across the enterprise thus preventing implementation errors.

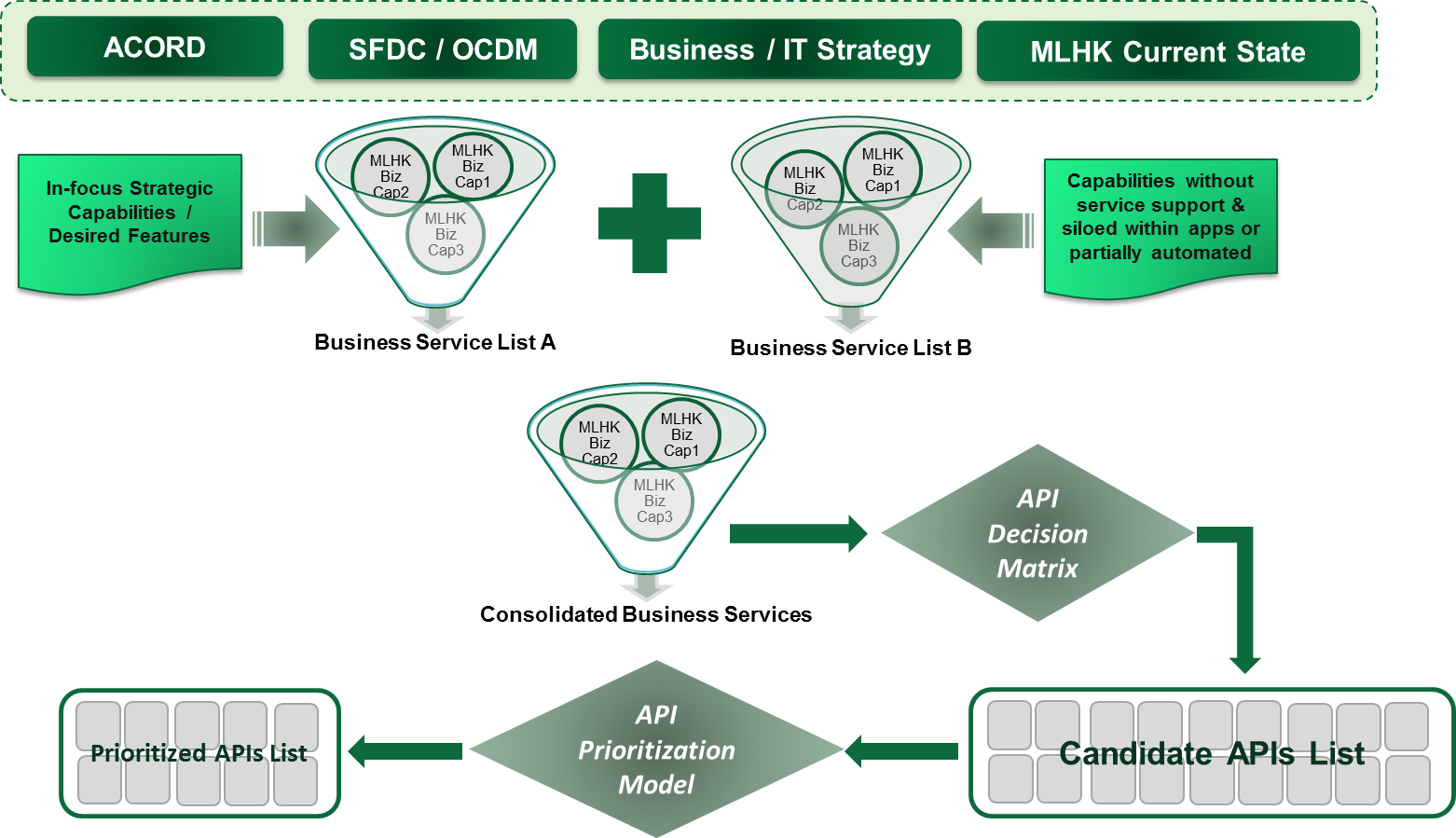
API documentation is very crucial aspect of API design. By following Open API / Swagger like standards greatly simplifies the API documentation process as it can be automatically generated from API design definition.

# API Design Principles

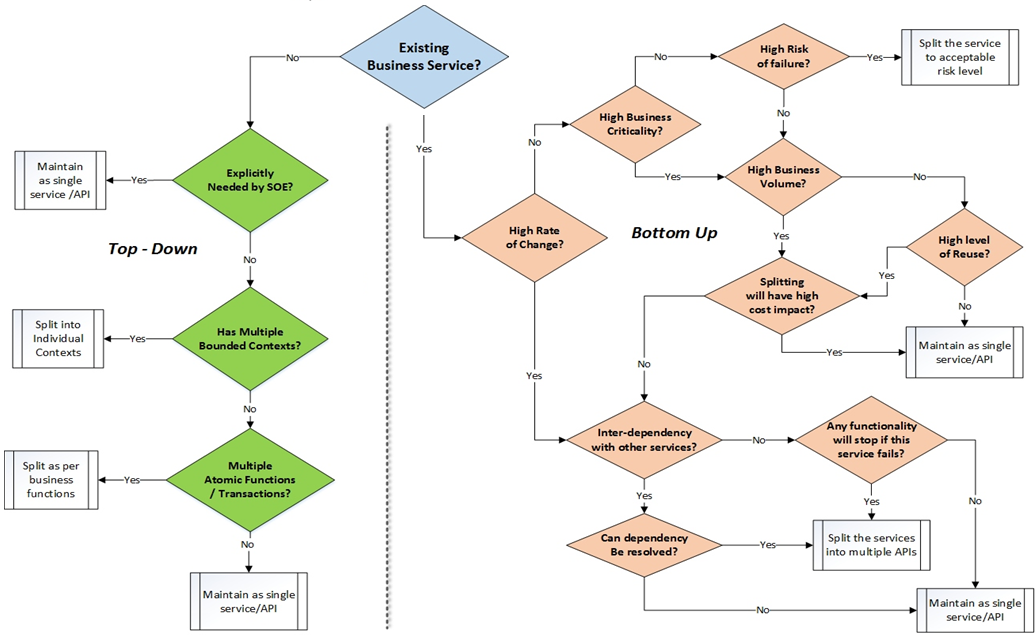
|  |  |
| --- | --- |
| **S No** | **API Design Principles** |
| 1 | API Design is an architectural concern that spans across business functions, product designs and service operations. |
| 2 | Existing systems design needs to be re-looked through domain driven design lens in order to identify API boundaries and resources. |
| 3 | While designing APIs, we first need to determine what resources it will offer and the bounded context it will serve to. Domain driven design concepts can be looked into to find out relevant business entities, entity relationships, possible state transitions, events and callbacks |
| 4 | For API design - UI / App Developers are the real users. Principles of user-centric design also applies to APIs (simplicity, obviousness, fit-for-purpose etc.) |
| 5 | APIs should be designed at the lowest practical level of granularity because it allows them to be re-used and combined in different ways to address consumer needs. |
| 6 | API granularity can be determined by identifying key entities and then attempt to model their life cycle. One API operation for each entity life cycle state transition is recommended. |
| 7 | There should be only one API function for one business outcome (e.g. change of address). |
| 8 | Microservices APIs can be combined / aggregated for creating customer facing experience APIs. |
| 9 | When a service API is being integrated (aggregated) with another service then it’s important to ensure that the aggregated API covers full life cycle of a business process. Failure to do so would seriously impact the user experience, because users would need to make multiple calls (which may result in managing lifecycle states) to complete the business process. |
| 10 | Consumer centric APIs should offer complete business transaction as offered by websites / UI Apps |
| 11 | API design should be technology agnostic and must adhere to Industry standards like Open API |
| 12 | Graceful handling of error conditions is an essential element of API Design. Error messages should be delivered in human-readable format that can be easily understood by the end user. |
| 13 | Error messages should also include runtime diagnostics and other technical details for UI / App Developers and Administrators reference. |
| 14 | API availability must be aligned and equivalent to the availability of the websites / UI Apps. |
| 15 | Changes to APIs must always be deployed as fully backwards compatible upgrades. If they are not backward compatible, the old API version must be maintained alongside the new version for an appropriate period to allow all consumers to transition. |
| 16 | The level of security required for a specific API depends on a risk assessment. Every API should specify its own risk assurance level and then enforce the needful security practices. |
| 17 | Every API must be discoverable and documented. Good API documentation is the single most important quality that UI / App Developers look forward to. |
| 18 | The API documentation should be published and a link to the documentation should be provided from the API endpoint. |

# API Identification / Discovery Approach

API Identification Model

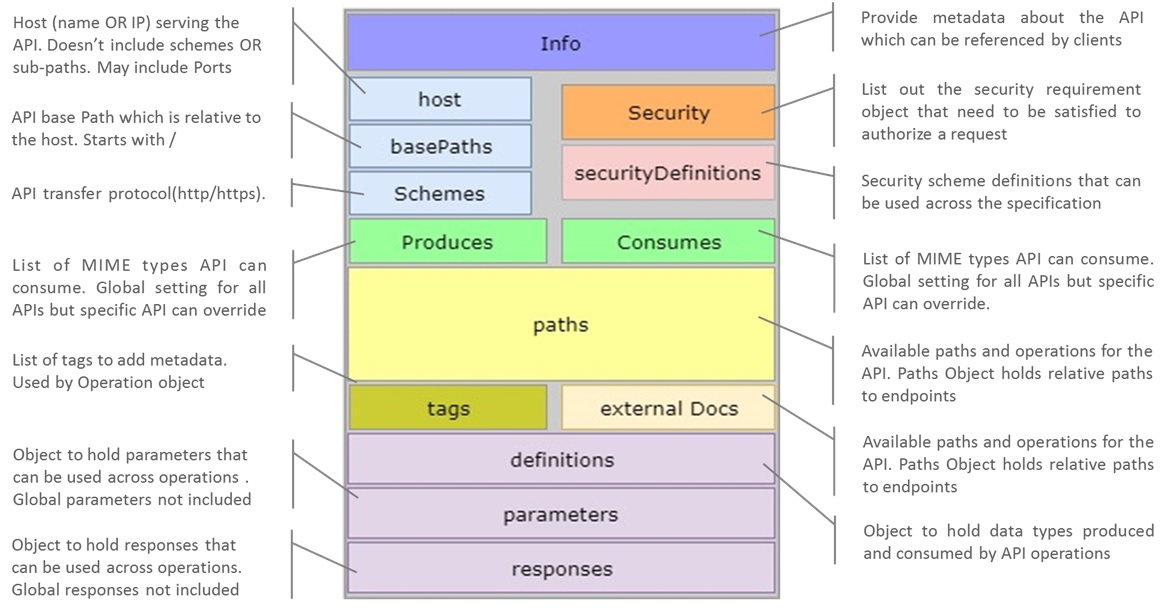
****

API Decision Matrix

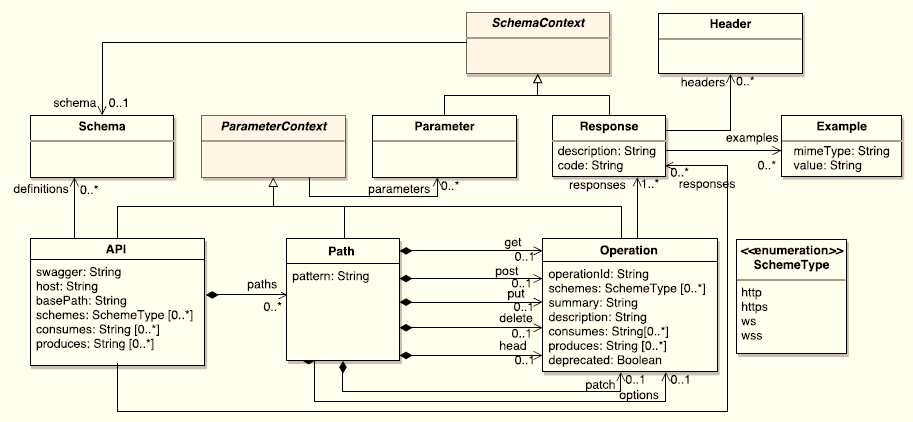


# API Design Specification

Open API / Swagger 2.0 Design Specification

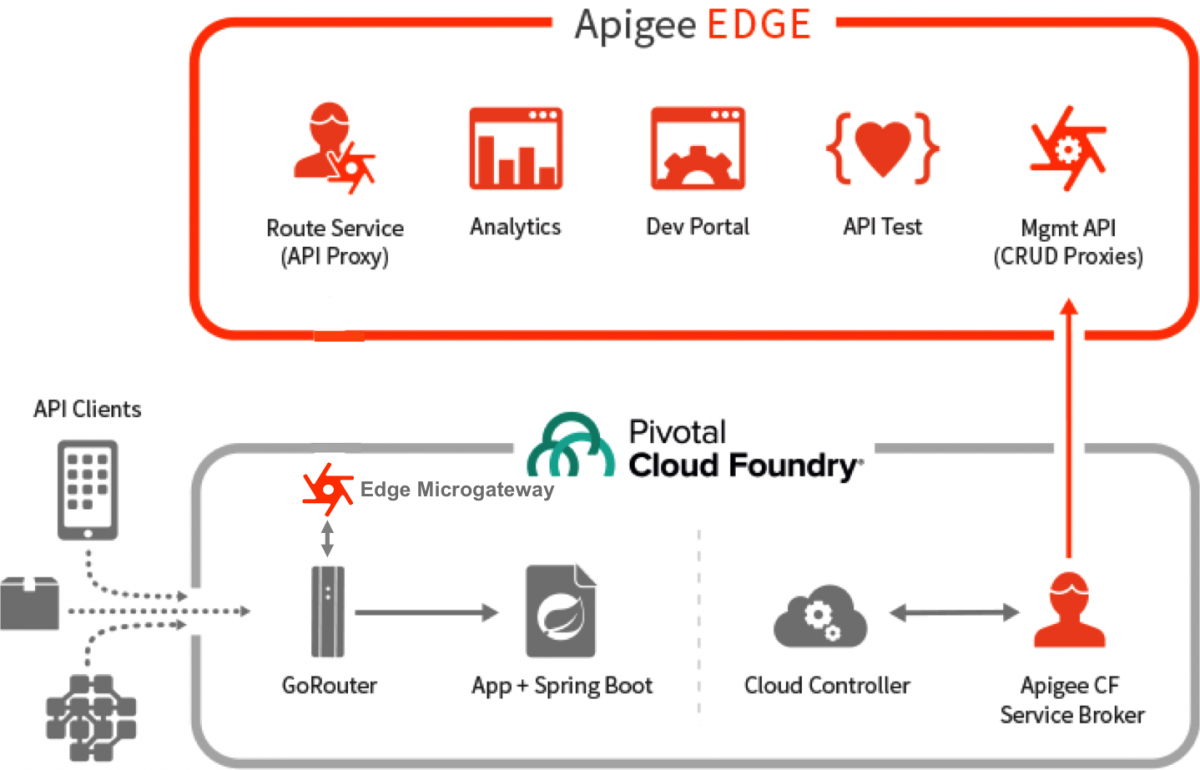


**API Design Specification – Meta Model**



# API Management using APIGEE platform

APIGEE Edge Microgateway Architecture



**APIGEE Platform – Capabilities Overview**



# API Implementation Best Practices & Guidelines

API Design Guidelines

|  |  |
| --- | --- |
| **No** | **API Design Guidelines** |
| 1 | Follow Swagger / Open API standards for defining API design specifications |
| 2 | Resources should be named using nouns, actions as HTTP verbs |
| 3 | The API URL paths should be as short as possible |
| 4 | Always version the API's as per the defined API versioning strategy |
| 5 | Ensure the API returns proper HTTP status codes and error messages |
| 6 | Ensure that the API always returns the latest version of data representation |
| 7 | Make use of different set of query parameters for different service consumers |
| 8 | Provide pretty print by default along with gzip or via query parameter |
| 9 | Implement Filter, Sort as query parameters based on predefined conventions |
| 10 | Make use of camel Case as naming convention for field names |

API Development Guidelines

|  |  |
| --- | --- |
| **No** | **Service Design Guidelines** |
| 1 | Provide API User manual with detailed description of scope, purpose, API usage examples, error details etc. |
| 2 | Make sure that each and every request is stateless and atomic in nature |
| 3 | Always validate the input fields and return dedicated error information in case of any violations |
| 4 | Any unknown input fields should be handled as Bad Request |
| 5 | Always return JSON object as response instead of Array |
| 6 | Always prefer usage of conventional query strings (search, q, limit, fields, sort, offset etc.) |
| 7 | Use consistent and structured error bodies with an error code and error message |
| 8 | Use content headers correctly (Content-Type, Content-Length, Content-Encoding, Content-Disposition etc.) |
| 9 | Long running requests should be executed in asynchronous mode with response status as 202 |
| 10 | Provide support for pagination using standards parameters like offset, limit etc. |

HTTP Method Usage Guidelines

|  |  |
| --- | --- |
| **No** | **HTTP Method Usage Guidelines** |
| 1 | Always use appropriate HTTP methods as per the action being performed on the resource |
| 2 | Use GET to read a single or multiple set of resources |
| 3 | Try to avoid request body payload in GET requests and instead use POST with body content |
| 4 | If resource(s) does not exist in a GET request return 404 as response |
| 5 | Use PUT requests only on single resource |
| 6 | PUT requests to return 200 or 204 if the resource is updated and 201 in case the resource is created |
| 7 | Use POST request to create one or more new resources |
| 8 | POST request to respond with 200 on successful resource creation and 202 for asynchronous requests |
| 9 | Use DELETE method to delete a particular resource |
| 10 | DELETE request should respond with 200 if the deleted resource is returned as part of response, 204 if no content is returned and 404 if the resource to delete is not found |

HTTP Response Status Details

|  |  |
| --- | --- |
| **No** | **HTTP Response Status Details** |
| 1 | Use 200 as standard success response (GET, PUT, POST, DELETE) |
| 2 | Use 201 for successful resource creation (PUT, POST) |
| 3 | Use 202 for request which would be processed asynchronously (POST, PUT & DELETE) |
| 4 | Use 204 if the there is no content in the response (PUT, DELETE) |
| 5 | Use 400 for Bad Request (Unknown fields, Validation fails etc.) |
| 6 | Use 401 if user is not logged in |
| 7 | Use 403 if user does not have proper access permission |
| 8 | Use 404, if the resource is not found |
| 9 | Use 500 in case of any server side exception |
| 10 | Use 503 in case the service is not available |

API Testing Guidelines

|  |  |
| --- | --- |
| **No** | **API Testing Guidelines** |
| 1 | Ensure all routes to the API is tested to ensure correct operations are executed |
| 2 | Make sure all routes are tested for appropriate access checks |
| 3 | Ensure all inputs to the API are validated appropriately |
| 4 | Make sure the test cases are automated with separate test bed setup |
| 5 | Test all the exception scenario’s and ensure meaningful HTTP response is sent back to client |
| 6 | Ensure all request and response messages are well formed |
| 7 | Ensure all request headers are tested properly and service responds with appropriate messages |
| 8 | Test all different permutation and combinations of query parameters provided by the API |
| 9 | Ensure all asynchronous operations are completed successfully |
| 10 | Ensure performance and stress tests are run for every major release |

API NFR Guidelines

|  |  |
| --- | --- |
| **No** | **API NFR Guidelines** |
| 1 | Always secure the API access with TLS (HTTPS) |
| 2 | Secure the API endpoints with OAuth 2 |
| 3 | Ensure API is protected with appropriate access rights using Scopes and Resources |
| 4 | Do not expose stack traces to client which could provide several details about the inner working of the services |
| 5 | Enable GZIP compression to reduce bandwidth and improve responsiveness |
| 6 | Provide filtering of resource fields based on the standard field parameter which helps to reduce payload size and also useful in case of different types of client needs |
| 7 | Enable monitoring on the API’s to ensure the API’s are always available, analyze API usage pattern and identify problems which may become an issue in future |
| 8 | Enable tracing of request to understand the success and failure rates of request processing and analyzing the performance of the request |
| 9 | Enable monitoring of system resources including CPU usage, Memory usage, Request queue length, Thread count, Network IO, Request and Response size etc. to understand and analyze the overall resource usage trend |
| 10 | Ensure all request and response are logged for auditing and analysis purpose |
| 11 | Track the number of requests from same client and if crosses the threshold within a specific timeframe, decline the request for that client and avoid Denial of Service attacks |