

TDRA

هيئة تنظيم الاتصالات والحكومة الرقمية
TELECOMMUNICATIONS AND DIGITAL
GOVERNMENT REGULATORY AUTHORITY



United Arab Emirates

UAE GOVERNMENT API FIRST GUIDELINES



دبي الذكية
SMART DUBAI

هيئة أبوظبي الرقمية
ABU DHABI DIGITAL AUTHORITY



UAE
GOVERNMENT
API FIRST
GUIDELINES

Table of contents

1	Introduction	5
1.1	Ownership of this document	5
1.2	API Definition and Overview	5
1.3	Purpose	6
1.4	Scope	6
1.5	Key Advantages	6
1.6	Key Terminologies	7
1.7	Definitions, Acronyms and Abbreviations	7
1.8	Audience	7
1.9	UAE API Guidelines' Stakeholders	8
2	API Fundamentals	8
2.1	What is an API	8
2.2	API Example Use-Cases	9
2.3	Why is API Design Important?	9
2.4	Benefits of having API Guidelines	10
2.5	API Design Principles	11
2.5.1	Consumer-centric Design	11
2.5.2	API Design	11
2.5.3	Future-Focused Design	12
2.5.4	Stability & Backwards Compatibility	12
2.5.5	Data Privacy and Sensitivity Awareness	13
3	API Management and Operations	13
3.1	API Prioritization	14
3.2	API Release Management	14
3.3	Lifecycle/Change Management	14

3.4	Access Management	15
3.5	API Catalogue	15
3.6	API Environments	15
3.7	API Development and Testing	16
3.8	API Test Data	17
3.9	Consumer Support	17
3.10	API Availability	18
3.11	Incident/Event Management	18
3.12	Analytics	18
3.13	Backup Procedures	19
3.14	API Security	19
4	Service Level Agreement Guidelines	21
4.1.1	Onboarding SLA's	21
4.1.2	Service Life Cycle State Approval SLA's	22
4.1.3	Service Availability	22
4.1.4	Service Performance SLA's	23
4.1.5	Information Exchange SLA's	24
4.1.6	Change Management SLA's	25
4.1.7	Incident Management SLA's	26
5	Technical Guidelines	27
5.1	Recommended Protocols	27
5.1.1	REST APIs	27
5.1.2	GraphQL APIs	28
5.1.3	Usage Recommendations	29
5.1.4	SOAP API	29
5.2	Supported Features	29
5.3	Error Handling	36
5.3.1	Error Handling Strategies	36
5.3.2	Types of Errors	36
5.4	Logging	38
5.4.1	Log Levels	38
5.4.2	Log Layout	39
5.5	API Documentation	39
5.5.1	Generating API Documentation	41
5.6	Other Considerations	42
6	References	43

1. Introduction

1.1. Ownership of this document

The UAE Government API First Guidelines document is the property of the United Arab Emirates Government, and is developed by the Telecommunications and Digital Government Regulatory Authority (TDRA), in collaboration with Abu Dhabi Digital Authority and Smart Dubai Government.

The guidelines herewith are generic and can be used, reproduced or printed by any government organization in the UAE, and can be availed unaltered by any non-government organization in the UAE. This is a live document and can be updated based on the new trends and updates in the field. Any unauthorized use, reproduction or printing of this document is strictly prohibited without a written approval from the TDRA.

1.2. API Definition and Overview

Application Programming Interfaces (APIs), previously called web services, are standards-based interfaces that allow applications to expose their functionality to external systems. APIs define how one application interacts with one another in a structured way to facilitate application integration and information exchange. Software developers create APIs to share functionality or data from an application they've developed with anyone who might want to leverage that application's functionality. APIs are now a common standard across applications and lead to a service-oriented approach which is an architectural best practice.

APIs help UAE government entities increase their efficiency and effectiveness by providing higher public value in less time, cost, and effort while ensuring real-time processing, increased data connectivity, and flexibility. Using APIs, government entities can expand old systems or applications and create new ones with minimal overhead and investment, which, in turn, enables innovation and enhances the overall public value.

The growth in API usage is driven by the need to deliver more customer-oriented functionality and a faster time-to-market. An API-oriented architectural approach facilitates industry-wide innovation and increases business agility.

APIs enable software applications to communicate and interact with each other and exchange data directly without the need for human intervention. For any given software or application, an API specifies the following:

- A mechanism for connecting to the software or application.
- The data and functionality that is made available for this software.
- Specification and standards that need to be followed by other applications to interact with the application's data and functionality.

APIs expedite the realization of an interconnected and interdependent ecosystem that promotes (intra and cross-sector) partnerships, stimulates co-operation and increases information/resource sharing between organizations. APIs allow you to build reusable components and develop a platform so that entities don't have to reinvent the wheel every time. APIs are, in essence, the building blocks of a digital ecosystem. Understanding the value of APIs will help the UAE government entities shift to a digital first approach.

1.3. Purpose

The purpose of this document is to provide API guidelines for UAE government entities.

The UAE government aims to adopt an API-first approach for the digital transformation initiatives. Government entities and vendors can follow the API guidelines outlined in this document for guidance on API implementation to accelerate the development of government APIs based on best-practices. This document acts as a comprehensive guide in designing and developing the APIs in government organizations. This approach will accelerate innovation, ensure responsive and customer-focused initiatives and capitalize on the power of the community at large for public value creation. This document is a working document that will be amended over time. The guidelines in this document contain no explicit technology or protocol restrictions; rather, the document offers best practices-based guidelines that ensure that UAE digital government APIs are effective, designed correctly, secure and provide value.

1.4. Scope

This document aims to be a useful tool in planning and implementing digital transformation in UAE government entities and organizations. It contains a set of high-level guidelines with design and implementation guidance, along with low-level API best practices to guide government entities in their development of APIs.

This document will help ensure that government services are:

- Interoperable with other platforms and services.
- Less likely to be locked to a particular vendor or technology.
- More future-proof and easier to update and test.

1.5. Key Advantages

Through this document, the UAE government aims to encourage an API ecosystem to expedite digital transformation and facilitate innovation through:

- Faster Time to Market – allowing faster response to market opportunities.
- Collaborative Model of Service Delivery – enabling other organizations to participate in external business processes.
- Meeting the Goals of Open Data – using APIs to open public data to public access, enabling private sector to create their own solutions based on public data.
- Leveraging Emerging Technologies – encouraging uptake of new and innovative technologies to keep pace with customer demands and expectations.
- Enabling Channel Shift – supporting government entities in their desire to make APIs their main web channel in preference to developing entity-specific web user interfaces.
- Contributing to the UAE Economy – spurring economic growth by helping grow competitive businesses within UAE in the information-based economy.
- Allowing development of innovative products through partnerships within and across sectors
- Improving Customer Interactions with Government – enable integration between organizations and allowing for streamlined government processes.
- Moving from the model of government entities as service providers to become an enabler for the private sector while ensuring governance of the same.
- Using Cloud Services – ensuring reliable and secure information exchange with Cloud-based solutions.
- Improving quality and access to information by providing a robust interaction model.



1.6 • Key Terminologies

The following are the terminologies used in the standard with its description.

Terminology	Definitions
API Service Provider	An organization that exposes data through APIs
API Consumer	Any organization or person who uses the provider's API to access or send data



1.7. Definitions, Acronyms and Abbreviations

Acronyms	Definitions
NGO	Non-Government Organization
REST	Representational State Transfer
RPC	Remote Procedure Call
HTTP	Hypertext Transfer Protocol
SOAP	Simple Object Access Protocol
XML	Extended Markup Language
JSON	Java Script Object Notation
LDAP	Lightweight Directory Access Protocol
SLA	Service-level Agreement
WSDL	Web Services Description Language

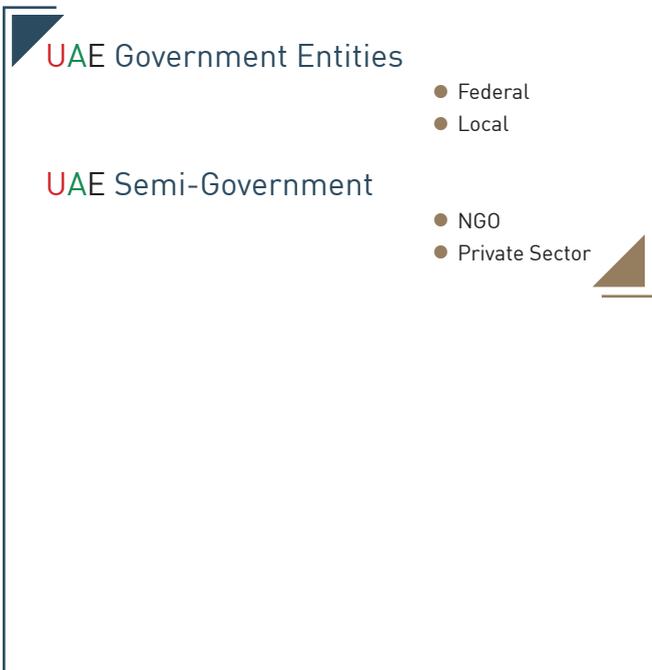
1.8. Audience

The target audience of this document is API providers and consumers who are developing or planning to develop applications which use government service APIs such as:

- Executives, Directors and Managers looking to deliver interoperable digital services.
- Software Developer or Systems Architect looking for technical guidance.
- Product Manager on any government service for design and implementation guidance.
- Business Analyst considering how to best digitize and design services.
- Policy Advisor on a digital service looking to better understand how APIs can help fulfill policy goals.
- Content Designer, User Experience Designer, or a practitioner of any other relevant discipline who wants to familiarize themselves with the principles outlined here so they can add their perspective.

1.9. UAE API Guidelines' Stakeholders

The key stakeholders for the UAE API Guidelines are the API Providers/Consumers from:



2. API Fundamentals

This section of the document considers the business and operational context for APIs within the UAE government and articulates the principles and considerations that may impact a government entity when creating APIs. The section also looks at APIs in the context of their impact on the entity as well as across government and public services.

2.1. What is an API?

An API is an application layer built on top of existing applications, systems, or software which allow other software applications or systems to use the existing application. APIs are:

- A service contract between an API consumer and an API provider describing what information and/or functionality the API consumer can access and leverage.
- A means of requesting and retrieving information from an API provider's application or system. The information retrieved can then be presented to end users in a context/process-specific way, e.g., through a mobile app or web application.
- A way of decoupling system to system interactions through well-designed service contracts.
- Not tied to any specific programming language or software product.

APIs are commonly categorized as shown in the table below:

Acronyms	Definitions
Internal API	An API that is used solely within one entity by known internal applications.
External API	An API that is provided by one entity and consumed by an external party.
Public or Open API	An API that is provided to the public and is able to be used by any party who wants to use it. Public APIs still can have access control requirements or be for open access by any party.

2.2. API Example Use-Cases

APIs are used in all industries. When you look at the weather app on your mobile phone, the information you see is coming from an API built by your local weather service. Their weather API takes weather prediction data and converts it into a format that app developers can use. With an API providing data, developers focus on building a user-friendly interface (in this case, your phone's weather app) without having to worry about the science behind the data.

APIs can do more than share data; they can be the connective "glue" between multiple independent systems. An entity— such as a traffic department, for example— that issues driver's licenses could build an API that takes a license number and checks whether it has expired. This API would be very useful for a car rental company that wants to verify that a customer has a valid license.

2.3. Why is API Design Important?

Recently, with the rise of the digital government and the digital transformation initiatives, many products and services within the government have created APIs for consumption both internally and externally, especially to launch mobile applications.

These APIs have been created based on customized data models and customized technology/platform requirements; in fact, there is significant duplication in terms of APIs built for similar functionality. The development of these 'closed' internal and redundant API ecosystems, while deemed necessary, have led to increased development and maintenance cost.

Although some entities have been producing APIs for some time, others are taking their first steps in offering APIs for public use. API standardization can provide significant benefits to support this API development across UAE government entities, in order to:

- Encourage consistency across government API offerings and thus making it easier for third parties to build solutions using government APIs.

- Avoid different government entities developing or delivering redundant APIs.

- Generate awareness of what is needed to support APIs as a delivered product to private sector customers e.g. levels of service, support capabilities, etc.

- Capture API best practice for reference by the API development community, indicating where standard adherence is required, and which areas allow for more flexibility.

- Reduce disruption to application developers as government APIs evolve.

- Reduce effort for entities by simplifying the process of developing and delivering on their API strategies.

UAE Government API First Guidelines

To ensure standardized APIs, the following standards must be present and aligned:

API Design Guidelines – address a broad range of design considerations leading to a uniform **API** design language across the UAE government and enable developers to create APIs which are easy to consume and well documented.

These are covered in detail in the coming sections of this document.

Data Standards – define the semantics, schemas, and syntax of the data (messages) being delivered through **APIs**. Data standards provide a common language of communication across the government.

Standards help define a frame of reference that two parties can use for data exchange and dictate the format and structure of data exchange. Standards define entity names, definitions, data element names, formatting rules, implementation guidelines, procedures, etc.

Introducing data standards for **APIs** help UAE government entities:

- a. Improve data quality and interoperability.
- b. Enable reuse of data elements and metadata, thereby improving reliability and reducing cost.
- c. Ensure consistency in code sets and standard lookups.
- d. Improve efficiency of mapping by providing common sets of core data elements.

Information Security Standards – are defined to secure the information being transmitted through the **APIs** and ensure the privacy of services data.

Information security standards and guidelines for designing and developing secure **APIs** are detailed as part of this document.

2.4.

Benefits of having **API** Guidelines

The benefits of providing guidelines to APIs design across the UAE government can be summarized as follows:

- **API Economy** – Embracing standardization in an API economy enables the UAE Government Entities to be prepared for and take advantage of the next technology disruptive trend.
- **Innovative new products** – Standardization will promote the creation of innovative products. This will be by exploiting the providers' unique capability or through partnerships with third parties such as the private sector. Additionally, the capabilities of different providers can be merged to form new product lines and bundle existing products.
- **Enhanced customer experience** – well-designed APIs could potentially improve end user experience through timely, efficient, accurate and contextually relevant data for customer reference and decision making.
- **Uniform integration** – Standardized APIs across entities would enable easier integration of systems across the government with additional benefits of lowering the associated costs and promoting interoperability.
- **Cost efficiencies** – Standardization would reduce development cost and time to market for new applications and/or services. It will also ensure functionalities are available to any developer independent of platform.
- **Promotes reuse and simplicity** – APIs promote the reuse of existing applications and services through the simplification of interfaces. APIs will expose existing information assets independent of their underlying access mechanisms/technologies.
- **Creating service bundles** – Standardized APIs within the government services and across sectors will allow government entities to set up partnerships (inter and intra-sectors) that could potentially create new user journeys and service bundles.

2.5. API Design Principles

2.5.1 Consumer-centric Design

In alignment with the customer-centric service design principle, APIs need to be developed with the API consumer in mind. The consumer application developers also need to be considered since the APIs must fulfill their application requirements. APIs should be developed as generic as possible in order to meet the basic needs of all potential consumers. API developers should not try to predict how the customers will interact with consuming applications, but should allow the application developer to innovate and best use the API to suit the needs of their application. By designing APIs for the consumer, agencies are more likely to build APIs which are intuitive and easy to use thus ensuring uptake of their APIs and encouraging access to public information.

The development and delivery of APIs should be geared around making it as easy as possible for developers to discover, understand and develop against those APIs. So APIs, along with the associated on boarding and support processes, should be simple to understand and well described.

Some examples of this could be:

- Ensure a low barrier to entry so it is easy to start using the API.
- Provide sandbox APIs so application developers can try out APIs and develop in parallel.
- Be responsive to feedback and bug reports.
- Provide automated on-boarding processes, as manual processes can limit take-up.
- Provide prototyping tools and support for development.
- Create an SDK to support an entity's APIs, including examples.

2.5.2 API Design

APIs should be designed in alignment with service-oriented architecture principles. The Service Provider API developers should take great care in designing their application's API. Poorly designed APIs will significantly reduce or take away the advantages of an API-based architecture. Working with potential Service Consumers, the business requirements should be gathered. Based

on the consumer business requirements, APIs should be designed in fulfillment of the consumer business requirements. The interface, operations, and fields can be defined and agreed between the provider and consumer before development starts. These business and technical details should then be formalized through a design document or specification.

Good API design includes the following principles:

1. Usability – ensure high quality user experience for consumers.
2. Interoperability – enable exchange of data across organizations without any dependencies on underlying technologies.
3. Reuse – leverage existing standards and taxonomies to avoid duplication of efforts.
4. Independence – avoid dependency on any vendors or technologies to provide options in delivery models and implementation technologies.
5. Extensibility – establish flexibility to extend APIs to new stakeholders and business channels.
6. Stability – ensure consistency and transparency of changes through communications and governance.
7. Transparency – provide clarity on environments and standards supported.
8. Loosely coupled – provide flexibility and minimize impact of changes to operations of other APIs.
9. Granularity – provide the appropriate level of functionality and not be too monolithic or too specific.

Additionally, it is strongly encouraged that application developers start to produce API consuming applications based on the interface specification as early as possible. This agile or iterative approach helps ensure real-world feedback is incorporated into the API design as early as possible. Completeness is not necessarily the goal, especially in initial APIs. The goal should be to get early partially complete releases out, defining the limited capability they offer, to enable consuming application uptake. Development needs to be flexible and agile to adapt to early adopters' feedback in identification of pitfalls and issues. However early releases should be tested and stable so as not to impede uptake. The aim is to try, then adapt, rather than waiting to release a fully functional API.

2.5.3 Future-Focused Design

Most entities will have a variety of legacy systems that they need to continue to support and service. It is important to remember that business, technology and application architects should be designing for the future of their organization, and not "hamstring" their APIs under development by designing them to work in the way the legacy system currently works, or to tailor APIs so that they work perfectly with all legacy systems. The aim is to be future-focused (whilst still pragmatic) and develop APIs to meet future needs.

2.5.4 Stability & Backwards Compatibility

It is important that APIs have stability (are available and work consistently) and support a velocity of change which is acceptable to the application developers. Early versions of APIs should be available via pilots or on developer portals so application developers can work on them and identify areas of enhancements and improvements before an API goes into production.

Application developers will not always be able to adapt to new capabilities or changes to existing interfaces as quickly as the API providers might wish, due to organization priorities and funding. Hence minor changes to APIs must always be deployed as fully backwards-compatible upgrades. For major changes, which are not backwards compatible, the old API version should be maintained alongside the new version for an appropriate period to allow all-consuming applications to transition. By monitoring usage, it should be possible to assess when an API version can be deprecated, either because it is no longer being used or because the usage pattern does not warrant maintaining that particular version. In any case, it is important to clearly communicate with your developer community and manage expectations as to longevity of a particular version of an API.

2.5.5 Data Privacy and Sensitivity Awareness

APIs are used extensively for passing information, so it is important to consider the information privacy and sensitivity aspects of data being passed to and from APIs to ensure that data is protected adequately. Consideration should be given as to whether a privacy impact assessment and/or a security risk assessment is appropriate for the API during each stage of development, from concept through to implementation. For example, if the API is providing programmatic access to publicly available information, the privacy considerations are likely to be minimal and the security considerations limited to the usual suspects. However, privacy and security considerations become hugely important if the API is providing programmatic access to private personal information. In this situation, it may be appropriate to do regular assessments, especially early in the concept phase to ensure any privacy or security constraints are understood before design.

There is also the issue that with ease of data consumption comes increased ability to combine data from different sources, which increases privacy risks and the potential for unintended information leakage. Hence, an API provider should assess the published API to be compliant with organization/country wide privacy policy and regulation.

The graphic features a dark blue vertical rectangle on the left containing a large white number '3' followed by a small brown square. To the right of this rectangle is a thin vertical brown line, followed by the text 'API Management and Operations' in a dark blue, sans-serif font.

3. API Management and Operations

APIs need to be managed as products similar to software products that commercial entities release. API management needs to be consistent across all the APIs the provider is publishing.

Full lifecycle management of APIs includes development, deployment, releases, and access management. API management also manages the availability of the API. This can involve leveraging features like throttling to make sure all consumers can get access to the API within the bounds of the SLA. It can also include quota management, whereby consuming applications are given limited access (e.g. a set number of calls per hour) to protect the API from abuse or overuse. It should be possible to use analytics to assess whether throttling or quotas are needed.

Service operations for APIs covers the actual delivery of the services to the service levels advertised. It handles management of, and access to, the API and any underlying applications, and looks after the infrastructure which underpins it. It also includes consumer support and incident management.

3.1. API Prioritization

As a first step, the provider should identify and prioritize the APIs for publishing. The drivers for the APIs should be considered and the internal infrastructure required to develop these APIs assessed.

Some of the key considerations include:

- Assess cost and benefits of publishing APIs.
- Alignment with Business and IT strategy.
- Current IT maturity to support the design and development.
- Degree of digitization in the business.
- Undertake business impact analysis.
- Identify and segment core APIs for publishing.

3.2. API Release Management

Release management is an important aspect of API transition. The aim with API development is to make small changes and release often.

The release management aspects include:

- Versioning - Informing application developers:
 - Where the interface specification has been changed, a major version release is required, with appropriate warning to application developers, scheduled deprecation of the previous major version and support for migration to the new version
 - A minor version change release is appropriate for backend changes that have little or no impact on the interface specification, and should have minimal impact on consumersHence it is important to know who your application developers are. When the API changes, the interface specification must also be changed to reflect the changes.
 - Planning API Rollout: Ensuring all the API artifacts are rolled out effectively and on time to the correct platforms
 - Emergency Patches: Informing application developers as to the need, and schedule, for emergency patch rollout, and ensuring emergency patching does not impede consuming applications
- Releases should be made to a test/development environment first.

3.3. Lifecycle/Change Management

An API has its own lifecycle, and it needs to be managed through the whole of its lifecycle, from creation to deprecation. This involves:

- Monitoring usage to make sure an API, or API version, is only retired when the majority of consuming applications have migrated off it

- Trimming APIs – removing features and functionality which are unused, have never been used or are not likely to

- Ensuring the API roadmap is up to date and gives a good indication of when major changes are scheduled to be made to each AP

3.4. Access Management

The first interaction Service Operations is likely to have with APIs is the initial act of on-boarding application developers. On-boarding should provide everything the application developer needs in order to interact with the API, including access to a test environment running a representative copy of the API underpinned by test data, documentation and preferably examples to work with. Consideration may be needed as to whether all application developers can work with shared test data or if they need individual test data specific to their purposes. In their day-to-day development activities application developers will want to be able to test the API without making (computationally or potentially financially) costly calls out to third party services, so it may be beneficial to provide mock versions of those APIs specifically for testing purposes. For full system tests, however, application developers will want their applications to test the full flow including any third-party service, so an automated mechanism for that may need to be built. Access management includes providing mechanisms through which application developers can apply for, and receive, permission and associated details to use the API i.e. external developers who want to build applications which make use of the API and applications which will ultimately be integrated with the API. It also covers managing access to the API, including specific, fine grained control for consuming applications. This allows operations to deploy access policies to ensure a consuming application's access to the API is in line with agreed constraints.

3.5. API Catalogue

Before an API is released for application developers to access, the API's description and interface specification should be published to an API catalogue. The API catalogue will contain a list of all the APIs offered, along with their interface specifications and guidance on how to gain access, and use the APIs, including the granularity of access control. This information needs to be up to date and accurate in order to be relied upon by API application developers. Some API hosting technologies can offer an automatic cataloguing capability.

An API catalog can be exposed to Service Consumers to allow developers to peruse available APIs and request access to those APIs.

3.6. API Environments

API Provider should provide the following environments for any of their published APIs:

Environment Type	Environment	Purpose
Non Production	Development	Build APIs and configure assets.
	Test	Used for Integration Testing
Production	Staging	Used for Performance Testing, Training and User Acceptance Testing
	Production	Production
	Disaster Recovery (DR)	Disaster Recovery in case of any failure
	Sandbox	Sandbox Environment for Public/Open APIs

Please note that the staging (or pre-production) environment is considered a production environment. The staging environment should adhere to the same SLA levels as the production environment and should mirror the specifications of production. The staging environment is essential for final verification of functionality before moving to production for all Service Consumers.

The test environment should also be stable so that Service Consumer developers can use them to develop their applications.

3.7. API Development and Testing

APIs need to be developed in a collaborative, flexible and adaptive way. Once the API interface specification is agreed, iterative releases with limited functionality can be deployed quickly by the Service Provider instead of waiting for the entire functionality of the API to be completed. This approach allows Service Consumers to start using the API immediately without having to wait for the final product. Initial versions of APIs can have functionality stubbed out (resources which can be called but return sample responses) or only offer partial functionality as long as the documentation indicates this. Support for this form of iterative development can be enabled through:

- Agile Software Development - Using Agile practices ideally suit this form of iterative development as they focus on developing small, incremental releases, 'failing fast' (finding out what's wrong early rather than too late) and frequent delivery of products.
- Configuration Management - All the components which make up an instance of the API should be held within version control, so that it is possible to rebuild a previous version if necessary. This involves:
 - Version controlling API interface specifications
 - Version controlling the associated API code
 - Keeping track of dependencies (e.g. external libraries being used within the API code)
 - Making sure access policies for individual consumers are version controlled
 - Being able to reliably recover all the elements of previous iterations of APIs and rebuild/redeploy if required
- DevOps - Following a DevOps approach enables Service Providers to automate and streamline all development and deployment activities. DevOps allows automated build, testing and deployment of APIs and the associated software whenever an update of the code is placed into configuration management.
- Automated testing - To make the incremental release pattern efficient, it is advisable to develop automated tests in conjunction with the API code so that testing becomes an intrinsic part of the build process.

3.8. API Test Data

Service Providers must provide adequate test data for their APIs. This test data is mandatory for Service Consumers to test APIs and develop their applications.

Test data must have the following characteristics:

- **Comprehensive** - Test data must allow Service Consumers to test all business flows that are covered by the APIs. Often, test data is provided to cover “happy flows”, and edge cases are neglected. Without proper test data coverage, a Service Consumer cannot have any assurance that their developed application will work in all cases.
- **Reusable** - As Service Consumer application developers develop their application, they will need to test APIs over and over again. After development is complete, the QA process of the application will also require full-cycle, full-coverage testing of the application that uses the APIs. This testing can only be performed with a reliable and reusable set of test data. Some Service Providers provide specialized APIs that reset the test data of their main APIs.

API test data must be provided for all environments:

- **Test/Integration Environment** - The test data for the test environment will be the most comprehensive. Having this test data will allow developers and QA specialists to test and retest APIs as their development progresses.
- **Staging/Pre-production** - The staging environment should have adequate test data to cover all scenarios. The staging environment will be the final step for application developer and QA specialists to ensure that their application is working as expected. This test data will often be used for automated testing on deployments. If the staging environment contains production data, any sensitive data should be masked or scrambled to avoid confidentiality breaches.
- **Production Environment** - Some test data must be provided for the production environment as a final assurance that the API and application using it are behaving as expected. Sensitivity must be used with production testing since some test scenarios, if not executed with caution, may affect real users and processes and potentially have legal ramifications.

3.9. Consumer Support

Application developers will need a variety of support mechanisms to aid their use of an API, including:

- Getting set up to use the API
- Understanding what to use during development
- Support for testing of their application in use of the API
- Reporting issues with the API

These mechanisms should include:

- Telephone support
- Support desk email address
- Online forum/support community

It could also include:

- Interactive real time support

It is useful to include indications of the level of support so that application developers know which form of support will most rapidly address their issue e.g. 5-9 telephone support, 24x7 community forum, response times to API failure reports.

There should also be support for handling requests for change, modifications or additions needed to the API. It should also be possible to capture and handle requests for new APIs.

3.10. API Availability

It is important not just to monitor and gather operational data about running APIs, but also to use that information to improve API offerings. API availability is of utmost importance to API consumers, so the API providers need to monitor availability, usage and respond dynamically to increases in demand. This requires comprehensive monitoring to identify potential issues, outages, bottlenecks or overloaded APIs and to ramp up availability to meet demand.

Proactive monitoring of API by the Service Provider is mandatory. Any issues or outages should be proactively identified and rectified immediately. The Service Provider should not rely on the Service Consumer to notify them of outages or issues.

3.11. Incident/Event Management

Incident and event management is geared around events picked up through monitoring, and unplanned incidents, and involves substantial amounts of communication. This includes:

- Monitoring the system as a whole to identify potential issues and pre-emptively apply mitigation e.g. throttling to counter potential DoS attacks.
- Capturing reports of, then informing application developers of, unforeseen incidents which are currently causing a disruption of service.
- Informing application developers of remediation of incidents, including resolution plans and predicted completion times.
- Deploying temporary fixes, if necessary.

3.12. Analytics

Capturing and analyzing data about an API in operation will pull out information useful to adapting to changes in demand. It is therefore useful to gather analytical data around:

- Take-up metrics, end user analytics such as location
- Tracking API consumers, their registrations and API usage
- API performance – identifying most commonly used APIs calls so they can be made efficient
- Event behavior (e.g. common patterns of behavior)
- Trace and diagnostics data

From an API take-up/consumption perspective it is useful to capture who, where, when, how, how often, and what device type is being used. Analysis of this data can then be used to demonstrate ROI.

Performance metrics are also useful, such as error rate, throughput, response time, transaction speed, backend performance, cache performance. These values can help identify trends and bottlenecks.

3.13. Backup Procedures

The data should be backed up at the application level and operating system level at regular intervals. The archived data will be retained for a specified period of time so that it can be recovered in case of any unforeseen issues. Data backups are taken without bringing the applications down, hence without impacting the running applications. The backup jobs are run at off peak hours so that the backup process will not impact the performance of the running applications.

The host-based backup at the operating system level is the full backup of the server and should be done weekly on the development and test servers. The file system-based backup should be done daily on the development servers and weekly on the test servers.

The backup strategies for application and database components in Production environment is explained below:

Application backups are taken at file system level daily and twice a week at the operating system level and would be retained for a week so that in case of issues, data backup would be available for the previous 7 days. The backup should include the application configuration changes, log files, messages, deployment changes, etc. These backups will be stored in the storage appliance

Database backups The backups are scheduled to run every day and the backups are retained for one week. These backups should also be stored in the storage appliance
Note: The backup strategy for Staging and DR environments is defined to be the same as Production environment

3.14. API Security

Services shall be classified based on the level of security as given in the table below. This shall help to protect the service (based on its classification) from unauthorized access and to address the integrity and confidential requirements.

Service Security Categorization	Classification	Type of information
High	Confidential	Military/police services, national security related services, legal services and sensitive entities' services
Medium	Official Use	Confidential citizen information, financial information and corporate details
Low	Public	Non sensitive private information such as names, addresses, email addresses

Note:

1. Services by default will be “Public” classified, which implies that the published services are viewable to all and API key is required for service consumption. In cases where the services are classified as “Confidential” or “Official Use”, Service Providers shall decide the service Consumers who are eligible to view their services.

Run time rules and actions need to be implemented to assure the confidentiality using web service level security. The actions to be applied depend on the sensitivity of the data transferred and transactions performed in the services. The following table shows the runtime actions based on the above categorization.

Security categorization	Run time action/rule
High	<ul style="list-style-type: none"> ● Authenticate consumers against a user directory ● Message-level security <ul style="list-style-type: none"> » Username/password credentials » Digital certificate authentication » Data encryption ● API identification ● Source IP ● HTTP Basic authentication. ● Transport-Level Security (https using SSL/TLS with minimum 256 bit encryption) ● Geo Location/Geo Velocity/IP Fencing/Error rates
Medium	<ul style="list-style-type: none"> ● Authenticate consumers against Service Provider user directory ● Transport-Level Security (https using SSL/TLS with minimum -256bit encryption) ● GEO Location / IP Fencing ● API identification ● HTTP Basic authentication. ● Transport-Level Security (https using SSL/TLS with minimum 256 bit encryption)
Low	<ul style="list-style-type: none"> ● Authenticate consumers against Service Provider user directory ● API identification ● Transport-Level Security (https using SSL/TLS with minimum 256 bit encryption)

4. Service Level Agreement Guidelines

Without robust service level management, it will be hard to engender trust in government APIs, which will negatively impact uptake. Application developers will need to know how long the API will exist, what commitment there is to its availability and performance, and what support is offered to those who consume the API. Without this, API usage will be based on an untrusted model, where application developers prepare for the API being unavailable. This results in consuming applications using APIs to top up local caches of data, or to support existing batch processes, missing out on the real time benefits of APIs.

There are several parameters that need to be agreed between the stakeholders for the easy usage of the published APIs.

 We gave here some examples on potential SLA's that need to be considered to ensure successful API governance

The SLA's can be classified as follows:

- 1) Onboarding SLA's
- 2) Service Provider Availability SLA's
- 3) Service Performance SLA's
- 4) Information Exchange SLA's
- 5) Change Management SLA's
- 6) Incident Management SLA's



4.4.1 Onboarding SLA

The entities will be on boarded to consume the published service as one of the following:

- Service Consumer
- Service Provider

The onboarding process is clearly defined for both publishing a service and consuming a service. Some of the steps involved in the process should require action from one or more of the stakeholders involved as well as provide approvals on certain states of a service.

4.4.2 Service Life Cycle State Approval SLA's

The published services will be governed via Service Lifecycle. This model defines set of states that make up the lifecycle of a service and the transition between the states are governed by predefined approver groups from Service Provider and Service Consumer

The below is a sample of the Service Life Cycle State Approval SLA

Table 4 Service Life Cycle States

Parameter	Value	Responsible
Approval of Service Life Cycle States	2 day	Service Provider
	2 day	Service Consumer

4.4.3 Service Availability

The following environments together decides the availability of the platform.

- Service Provider Platform
- Service Consumer Platform
- Service Provider Environment

UAE Government API First Guidelines

All Services, Platforms, and Infrastructure should adhere to the SLA's that must be defined by the organization. This excludes the regular maintenance windows.

In case of planned maintenance window or any unplanned outage, the Service Provider will be responsible to notify the service consumers.

The Service Provider needs to ensure that the service is highly available and this availability may vary depending on the service and the Service Provider entity that is described in the Service Performance SLA's section.

The below is a sample of the Service Availability SLA

Parameter	Value	Environment	Responsible
Service Provider Availability %	99.9 %	Staging & Production	Service Provider
Service Provider Availability Hours	7*24	Staging & Production	Service Provider

Clear guidelines and agreed SLA levels should be set for each environment, including test environments.

4.4.4: Service Life Cycle State Approval SLA's

The SLA's specific to a service performance can be:

- Response Time
- Throughput
- Service Availability
- Secured Message Transmission

These SLA's shall be defined in the service requirements document and can be agreed upon and signed off with the stakeholders depending on the nature of the engagement.

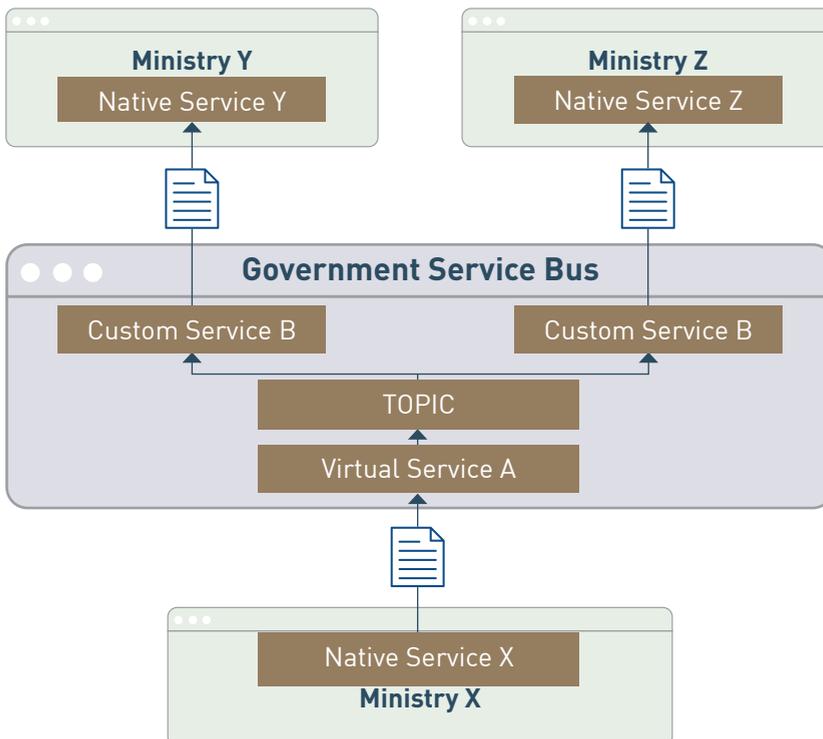
As a general guideline, APIs should adhere to the following performance metrics:

Parameter	Value	Responsible
API Response Time (Simple API)	← 200 ms	Service Provider
API Response Time (Complex API with intensive processing)	← 750 ms	Service Provider
API Availability	%99.9	Service Provider

4. Service Level Agreement Guidelines

If adherence to the aforementioned API performance guidelines cannot be achieved due to application or software constraints, the Service Provider can consider the following options:

- **API Service Caching** – If an API returns a set of data that does not change much or has a limited set of possible returned datasets, then the Service Provider application can cache the API data and return it from the cache instead of burdening the application on each call. Each API application will have data or response caching mechanisms that can be used. Any caching must carefully consider caching TTL (“time-to-live”) values in order to ensure that stale data is not shared with consumers.
- **API Publish-Subscribe Model** – If an API executes a batch or business process that takes minutes or even hours, then the publish-subscribe model can be used. The same API will be redesigned to have 2 APIs. The first API will be offered by the Service Provider and, when called, will initiate the request in the back-end application. A second API will be hosted on the Service Provider or Service Consumer. If the second API is hosted by the Service Provider, the Service Consumer will invoke the second API after a certain time after the first API invocation when the API results will be ready. If the second API is hosted by the Service Consumer, then the Service Provider will simply invoke the second API once the processing of the first request is complete. The following diagram illustrates how the Publish-Subscribe model works.



4.4.5 Information Exchange SLA's

Quality of Data:

The Service Providers will be responsible for the quality of data (correctness and coverage) that is shared through their services exposed. The decision and implementation of sharing complete information or partial information will be the responsibility of the Service Provider.

Confidentiality of Data:

Service Consumer will be responsible for maintaining the confidentiality of the data consumed from the services Provider. Service Consumer should use this data only for the intended purpose approved by their organization and compliant to any applicable laws and regulations set by the government of UAE

As part of the onboarding process to consume a service, the Service Provider and Service Consumer are expected to have a Memorandum of Understanding (MoU) or any other binding agreement that is signed between themselves or follow the protocol or process specific to entities on the quality and confidentiality of data.

The information passing through service Provider and consumer will be maintained by the following mechanisms:

- 1} The information is exchanged only within a secure network.
- 2} Services can be consumed only after approvals from both Service Provider and consumer
- 3} Appropriate Authentication and Authorization mechanisms are implemented.

4.4.6 Change Management SLA's

Any ad hoc changes made to the service or the service will impact the ability of the Service Consumer to consume the service. It is important that any changes to the service should follow change management process.

This is also applicable for any planned down times of published service.

Following is the summary of potential changes and the stakeholder responsible to comply with the SLA's defined in this section.

Parameter	Responsible	Impacted Stakeholders
Any changes to published Service	Service Provider	Service Consumer
Any downtimes to published Service	Service Provider	Service Consumer
Any Changes to Consuming Application	Service Consumer	Service Provider

Following SLA's needs to be adhered by the stakeholders:

Service Providers

The Service Provider may have a need to make one or more of the following changes to the service hosted in their environment:

- Change the service's address (end point, bindings)
- Change the service's business logic that may or may not impact the behavior of the service
- Change the service' contract or structure.
- Changing the IP address of the consuming application
- Changing the domain name

The above is only a list of the examples and not exhaustive.

In such scenarios, the following approach needs to be adhered to:

- The Service Provider needs to inform the Service Consumer about the change in advance.
- The Service Provider must assess the impact to the service and the Service Consumer and plan the change in a coordinated manner.
- The changes to the service need to be tested along with service and the Service consumer in test environments before the Service Provider promotes the changes to the production environment.

4.4.7 Incident Management SLA's

An Incident is an unplanned interruption to the Service Provider or degraded performance of the platform.

A Service Provider should provide the first line of support for incidents and can be contacted through suitable channels. Incidents have to be logged in an Incident Management system and the support team from Service Provider will evaluate the incident and involve one or more of the following teams to get the incident resolved.

- Service Provider Team – in case of issues in the published service
- Service Consumer IT Team – in case of issues in the consuming application

The priority of incidents will be classified as per the below guideline:

Priority	Definition
Priority Level 1 — Critical Critical Business Impact	The incident has caused the platform to stop immediately and completely, which is affecting primary business processes of Service Consumers. There is no available way to work around this problem.
Priority Level 2 — High Major Business Impact	The incident affects a business process so severely that the business function of Service Consumer is severely degraded. There may be a way to work around this problem, but the solution is not easily sustainable.
Priority Level 3 — Medium Moderate Business Impact	The incident affects a business process so that certain functions are unavailable to Service Consumers, or the platform is degraded. There may be a way around this problem.
Priority Level 4 – Low Minimal/No Business Impact	The incident has very minimal or no impact to the business function of the Service Consumers.

4. Service Level Agreement Guidelines

The below is a sample of the Service Consumers Incident SLA

Priority	Resolution Time
Priority 1 (P1)	8 hours
Priority 2 (P2)	16 hours
Priority 3 (P3)	4 days
Priority 4 (P4)	8 days

5. Technical Guidelines

5.1. Recommended Protocols

The following protocols are recommended whenever implementing an API

- 1) REST
- 2) GraphQL

5.1.1 REST APIs

REST (REpresentational State Transfer) API are the most commonly used standard for APIs today. REST is an architectural style introduced in 2000, based on a set of principles that describe how networked resources are defined and addressed. Data and Functionality are considered resources and are accessed using Uniform Resource Identifiers (URI) REST have six guiding principles which are as follows: _____

- 1) **Client-server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- 2) **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
- 3) **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- 4) **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- 5) **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
- 6) **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

The data should preferably be exchanged in JSON (JavaScript Object Notation) format which is a lightweight human readable data-interchange format. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers. JSON stores data in an array/ordered list of name-value pairs.

5.1.2 GraphQL APIs

GraphQL is a language for querying databases from client-side applications. It provides a set of tools that operated over a single endpoint using HTTP. On the backend, GraphQL specifies to the API how to present the data to the client. GraphQL redefines developers' work with APIs offering more flexibility and speed to market; it improves client-server interactions by enabling the former to make precise data requests and obtain no more and no less, but exactly what they need. GraphQL offer Strongly Typed Scheme for API Queries. It ensures there is no Over-Fetching on Under-Fetching of data saving time and bandwidth. Unlike REST it requires no versioning and it elegantly handles mutations, filtering, sorting and pagination etc. It transforms the shape of the response using the GraphQL execution library to match the shape of the query

	REST	GraphQL
Architecture	Server Driven	Client Driven
Organized in terms of:	Endpoints	Scheme and Type System
Operations	Create Read Update Delete	Query Mutation Subscription
Data Fetching	Fixed data with Multiple API Calls	Specific data with single API call
Performance	Multiple network calls take up more time	Fast
Development Speed	Slower	Rapid
Self Documenting	No	Yes
Web Caching	Yes	via libraries built on top
File Uploading	Yes	No
Recommended Use Cases	Simple Apps Resource-Driven APps	Multiple Microservices Mobile Apps

5.1.3 Usage Recommendations

REST API is the commonly used API protocol today and supported by a large developer community versus GraphQL which has an evolving community. While we recommend REST APIs to be used, GraphQL should be explored where the use-cases justify the use of GraphQL. GraphQL should be considered for Apps for Smart Devices where Bandwidth usage matters as well as applications where nested data needs to be fetched in a single call or in case of a composite pattern, where application retrieves data from multiple different storage APIs. GraphQL should be avoided in case of OpenAPIs for external entities.

5.1.4 SOAP API

SOAP (Simple Object Access Protocol) is a messaging protocol specification for exchanging structure information in the implementation of web services. It allows web service based applications to communicate between entities. SOAP predates REST API has been a very popular API mechanism. Still many systems only support SOAP APIs.

It is our recommendation to use SOAP only where legacy systems cannot support REST API or GraphQL APIs.

5.2. Supported Features

It is recommended that the Service Providers/Consumers incorporate the below features while developing their services.

Table 12 Recommended Features

Attributes/Feature	Guidelines/Recommendations
<p>Logging</p>	<ul style="list-style-type: none"> ● Service Providers/consumer should have appropriate logging mechanisms which will help trace a request/response end to end and provides information during a root cause analysis ● The following log levels can be set: <ul style="list-style-type: none"> » Info » Warning » Error » Debug <p>Note: Debug should be turned off in Production by default and can be turned on a need basis.</p> <p style="text-align: center;">The following details are logged:</p> <ul style="list-style-type: none"> » Log Level » Log Message » Transaction ID » Service Provider/Consumer » Timestamp <ul style="list-style-type: none"> ● In case of an API with PII (Personal Identifiable Information), all API transactions must be logged as to when data was provided and to whom. It should be compliant with all applicable data protection regulations (Example ISR) and should be capable of detecting and protecting fraud and misuse.

Attributes/Feature	Guidelines/Recommendations
<p>Error Handling</p>	<ul style="list-style-type: none"> ● Service Providers are recommended to handle all the exceptions related to the service and send appropriate response codes and descriptions in the response. The error codes and descriptions need to be shared with Service Consumers ● Service Providers are recommended to handle all the exceptions that could occur when invoking the published service ● API Consumers should implement appropriate retry mechanism when invoking a service API ● Appropriate alerts/notifications should be configured ● Service Provider should implement an error handling framework defined for handling error/exceptions ● The following details are logged in a service specific log file in case of an error: <ul style="list-style-type: none"> » Error Code » Error Message (stack trace) » Transaction ID » Timestamp » Consumer application ● For Publish-Subscribe scenarios, a retry mechanism is implemented in case the message delivery to the subscribing service fails ● Appropriate notifications should be sent on encountering an error
<p>Auditing</p>	<ul style="list-style-type: none"> ● Service Providers should ensure key information is captured in the request and response header and logged, so that the information is available to be audited and used for root cause analysis and non-repudiation ● Every service will log key information in the request/response header when it is invoked and this information is available for auditing ● Historical data (error logs, audit logs) is stored in database and can be used for root cause analysis, debugging and non-repudiation

Attributes/Feature	Guidelines/Recommendations
<p>Security</p>	<ul style="list-style-type: none"> ● Service Providers should ensure appropriate authentication and authorization mechanisms are implemented for the Service API. ● Service Consumers should ensure appropriate use of the information shared by the Service Providers ● To invoke a published service in, a service consumer needs a minimum of HTTP Basic Authentication over SSL. Additional security mechanisms will be implemented within service Provider based on the confidential nature of the service ● The security mechanism required for invoking the service is implemented in Service Provider environment. This security mechanism is defined and owned by the service Provider ● Service Provider can support the following authentication mechanisms: <ul style="list-style-type: none"> » Username, password » Certificates » Keys
<p>Dependability\</p>	<ul style="list-style-type: none"> ● Service Provider should take measures for API Dependability including prevention for Denial of Service as well as monitoring of APIs for unusual activity

Attributes/Feature	Guidelines/Recommendations
<p>Performance</p>	<ul style="list-style-type: none"> ● Service Provider should be responsible for the performance of the service and hence should fine-tune the performance of their services ● The Service Provider should adhere with the base lined SLAs like response time, throughput, etc. when the service is invoked by consumer ● Key performance metrics (average response time, throughput, etc.) are captured to assess the performance of the services ● The key performance improvement considerations with most Service Providers are: <ul style="list-style-type: none"> » In-memory caching » Multithreading » Memory management » Compression of any attachments
<p>Monitoring</p>	<ul style="list-style-type: none"> ● Service Providers should monitor their applications for performance, availability, etc. ● Service Provider implements appropriate mechanisms to monitor the services (for performance, availability, SLA, failure, success, response time, etc.) hosted in the integration platform ● Monitoring is implemented for the infrastructure (CPU usage, memory usage, availability, etc.) in which Service Provider is hosted
<p>Usage of Unique IDs</p>	<ul style="list-style-type: none"> ● It is recommended that the Service Provider stores data with a unique identifiers and invoke the service with this unique ID ● A unique identifier should uniquely identify the records retrieved by a service ● Unique identifier can be a single ID like Emirates ID ● Unique identifier can be a composite key like combination of Passport Number, Name, Nationality and Date of birth

Attributes/Feature	Guidelines/Recommendations
<p>Scope of service</p>	<ul style="list-style-type: none"> ● The scope of a service should be clearly defined ● A service should map to a single business function ● Technical and non-functional details of a service are clearly defined while designing a service in Service Provider Environment
<p>Segregation of operations</p>	<p>Service Consumer should segregate operations where required ● (within a service (E.g., summary vs detailed information</p> <ul style="list-style-type: none"> ● Operations of the same service should represent the same business function ● A service can be created corresponding to multiple other service operations or can be segregated for every service operation based on the need
<p>Namespace</p>	<ul style="list-style-type: none"> ● It is recommended that service Providers should define and use their own namespaces ● Appropriate namespaces for element types and attribute names will be defined to avoid element name conflicts
<p>Hosting your API</p>	<p>All API naming in URLs (including the name of your API, namespaces and resources) should:</p> <ul style="list-style-type: none"> ● use nouns rather than verbs ● be short, simple and clearly understandable ● be human-guessable, avoiding technical or specialist terms where possible ● use hyphens rather than underscores as word separators for multiword names <p>For example: [api-name].apis.government.ae</p>

Attributes/Feature	Guidelines/Recommendations
<p>Grouping of attributes</p>	<ul style="list-style-type: none"> ● It is recommended that Service Providers should group related attributes while defining the service request and response (E.g., Nationality Code, Nationality Description in Arabic and Nationality Description in English can be grouped to a Nationality node)
<p>Validation</p>	<ul style="list-style-type: none"> ● It is recommended that the Service Providers perform appropriate validation of the request (E.g., mandatory fields should not be empty, enumeration of fields where possible, etc.) ● Appropriate technical and business validations should be executed by the entities and proper error or response messages (code and description) should be sent ● Input Validation - An API must check that data is both syntactically and semantically valid (in that order) before using it in any way. <ul style="list-style-type: none"> » Syntax validity means that the data is in the form that is expected. For example, an if an API expects a four-digit "account ID", then the API should check that the data provided is exactly four digits in length, and consists only of numbers. As a general principle special character are not allowed as a part of any input unless there is a very specific business requirement » Semantic validity includes only accepting input that is within an acceptable range for the given application functionality and context. For example, a start date must be before an end date when choosing date ranges ● Output Validation – An API must ensure that the produced output must be filtered and sanitized to prevent unintended and unexpected interpretation and operation of data at the receiving end.
<p>Reusability</p>	<ul style="list-style-type: none"> ● It is recommended that Service Providers design and build services which can be consumed and used by multiple entities ● Service Provider should reuse common frameworks like Logging framework and Error handling framework

Attributes/Feature	Guidelines/Recommendations
<p>Lookups</p>	<ul style="list-style-type: none"> ● It is recommended that Service Providers maintain codes which are industry standards (like ISO codes) and descriptions in English and Arabic for the lookups (country code, document type code, etc.) provided to the service Providers ● It is also recommended to use enumeration whenever possible. For E.g., Gender can have enumeration values of Male or Female
<p>Content-type</p>	<ul style="list-style-type: none"> ● The content-type (XML, JSON) shall be defined by the Service Provider to be uniform for request and response and should be followed by the service consumer
<p>Naming Standards</p>	<ul style="list-style-type: none"> ● Service Providers should use unique and standard application names ● Service Providers can follow camel casing (E.g., sampleServiceName) for their services ● All services and elements must follow proper naming standards
<p>Addressing Scheme</p>	<p>URI Addressing Scheme in lines with RFC 3986 should be followed</p>
<p>Session Tracking</p>	<ul style="list-style-type: none"> ● Service Providers should have placeholders in the request and response header to capture a unique Transaction ID to track the message. ● Service Providers to generate a unique identifier and pass it in the request header of each message ● Service Provider to map the Transaction ID in the response ● Service Provider should track and log the unique Transaction ID passed in the request and response header ● This unique Transaction ID will be used for root cause analysis and for end to end tracking of the request/response

Attributes/Feature	Guidelines/Recommendations
Consumer Identification	<ul style="list-style-type: none"> ● Service Providers should have placeholder in the request header to identify the end consumer of the service ● Service Provider to pass the entity name in the request header and the API Key in the authorization block of HTTP header while invoking the service. It is recommended to pass the same standard entity name in the request header for all consuming services by the service consumer ● Service Provider logs the service consumer name passed in the request header to track the consumer of the service ● Service Provider should generate API keys that shall be used to identify the service consumer
Authentication	<ul style="list-style-type: none"> ● Service Providers should be able to authenticate Service Consumers when service is invoked ● Service Provider to pass the authentication details for Service Consumer validate it (E.g., Basic Authentication details in the HTTPS header) ● Service Provider shall authenticate the service consumer who invoke the services
Inter environment connectivity	<ul style="list-style-type: none"> ● Service Providers should open firewall ports to allow the Service Consumer servers connect to the services hosted in appropriate environments in the entity premises ● Service Providers production environment should be configured to accept connections from service consumer Production environment ● Service Provider Test and Staging environment should be used for Integration testing and User Acceptance Testing respectively
Date and time	<ul style="list-style-type: none"> ● It is recommended using the ISO 8601 standard to represent date and time in the response. This helps people read the time correctly. It is recommended to use a consistent date format. For dates, this looks like 2020-08-09. For dates and times, use the form 2020-08-09T13:58:07Z.

Attributes/Feature	Guidelines/Recommendations
Physical location	<ul style="list-style-type: none"> ● It is recommended to use the World Geodetic System 1984 (WGS 84) standard. ● It is recommended to use GeoJSON for the exchange of location information.
Encoding	<ul style="list-style-type: none"> ● It is recommended to use the Unicode Transformation Format (UTF-8) standard when encoding text or other textual representations of data.
Data requests	<ul style="list-style-type: none"> ● It is recommended to configure APIs to respond to 'requests' for data rather than 'sending' or 'pushing' data. This makes sure the API user only receives the information they require. ● It is recommended that an API must answer the request fully and specifically. For example, an API should respond to the request "is this user married?" with a Boolean. The answer should not return any more detail than is required and should rely on the client application to correctly interpret it.
Data Fields design	<p>Service Providers should consider how the fields will meet user needs. They can also regularly test the documentation. For example, to collect personal information as part of the dataset, before deciding on the response, it is recommended to consider whether:</p> <ul style="list-style-type: none"> ● the design can cope with names from cultures which don't have first and last names ● the abbreviation DOB makes sense or whether it's better to spell out the field to date of birth ● DOB makes sense when combined with DOD (date of death) or DOJ (date of joining)
Publishing bulk data	<p>Make data available in CSV formats as well as JSON when you want to publish bulk data. This makes sure users can use a wide range of tools, including off-the-shelf software, to import and analyze this data</p>

5. Technical Guidelines

Attributes/Feature	Guidelines/Recommendations
Rate Limiting/Throttling	Service Provider should implement API to have control on Rate Limiting/Throttling the APIs
Metering and Billing	Service Provider should implement Open APIs that should support Metering and Billing
Privacy by Design	Privacy by Design approach should be followed when developing APIs

5.3. Error Handling

A robust error handling strategy is required to handle the various errors and exceptions encountered while processing the service requests and responses. Good error handling implementation will increase the service reliability and provide the end user or application higher visibility and confidence about the service. An error in a service can occur due to a multitude of factors, some may be system related and some business data and process flow related (assuming all logical and programming errors are eliminated in the testing phase and the deployed application in production is already a robust one). As long as all these errors are trapped and reported accurately in a user friendly manner, with sufficient information to track the error and take any required subsequent actions on the error (to mitigate its cause), the service reliability and its robustness stands intact.

5.3.1 Usage Recommendations

The basic rationale behind exception handling is to catch errors and report them. The common error handling framework is built to address the below mentioned points.

- A detailed and user friendly explanation of the error
- Methods to notify the user
- Guidelines for the developer to handle the exceptions
- What types of exceptions should be handled?

The basic rationale behind exception handling is to catch errors and report them. A framework for error handling should address the following questions:

- 1) What is the level of detail that is needed in reporting the exception?
- 2) How should the user be notified of the exception?
- 3) As a developer where do you handle these exceptions?
- 4) Should we handle every exception?

5.3.2 Types of Errors

Errors and exceptions could occur while processing the request/response within the Service Provider. As a principal, it is recommended to handle the exceptions wherever possible and to pass appropriate response messages to the invoking services. This helps in understanding the error scenarios better and resolving the issues without delays.

The exceptions can occur within the service hosted by the Service Provider. These exceptions/errors should be handled as detailed out in the section below and appropriate error codes and descriptions should be sent to the invoking service. Service Providers should handle these errors appropriately.

The following type of exceptions can happen at service level:

- 1) Business Validation Failures
- 2) Technical Validation Failures
- 3) System Failures

The following table lists down the common exceptions and the expected response from Service Provider for the above type of exceptions.

Table 13 Error Types

5. Technical Guidelines

Exception Type	Exception	Response – REST services	Corrective action
Business Validation Failure	Request not meeting business criteria such as age, financial status, etc.	Response with appropriate response code and description in the body	Service Consumer should resend the request after populating the mandatory fields
Technical Validation Failure	Invalid request	Response with appropriate response code and description in the body	Service Consumer should resend a valid request
Technical Validation Failure	Internal database connectivity issues in Service Provider System or any other internal errors	Response with appropriate response code and description in the body	Service Provider should fix the issue with their internal systems and service Consumer should retry sending the request
Technical Validation Failure	Incorrect/Invalid authentication credentials	Appropriate HTTP status codes along with response code and description in the body	Service Consumer should update the authentication credentials and Service Consumer should retry sending the message
System Failure	Service Down	Appropriate HTTP status codes	Service Provider should ensure the service is available and service Consumer should retry sending the request
System Failure	Service Provider servers not reachable	Appropriate HTTP status codes	Service Provider should ensure the server is available and Service Consumer should retry sending the request
Technical Validation Failure	Request schema validation failure	Appropriate HTTP status codes along with response code and description in the body	

5.4. Logging

5.4.1 Log Levels

The log levels within the Service Provider is a common logging framework is used to set the type of messages that shall be logged in the service specific log file. The various log levels are given below.

Table 13 Log levels

Log Level	Description
ALL	All levels including custom levels
TRACE	Designates finer-grained informational events than the DEBUG
DEBUG	Designates fine-grained informational events that are most useful to debug an application
ERROR	Designates error events that might still allow the application to continue running
FATAL	Designates very severe error events that will presumably lead the application to abort
WARN	Designates potentially harmful situations
INFO	Designates informational messages that highlight the progress of the application at coarse-grained level
OFF	The highest possible rank and is intended to turn off logging

5.4.2 Log Layout

Services Provider should use the layout and the conversion pattern to format the logging information. The layout is set to `%d{yyyy-MM-dd HH:mm:ss:SSS zzz -}6%p - %m%n`

Table 14 Log Layout Mapping

Conversion Character	Description
d	Used to output the date of the logging event
p	Used to output the priority of the logging event
m	Used to output the supplied message associated with the logging event
n	Used to output the platform dependent line separator character or characters

An example for a debug message written using the common logging framework would be as follows:
 15:05:40:162 29-08-2016 GST - DEBUG - Message sent using logDebug service

5.5. API Documentation

API documentation helps your users integrate with Providers' API by explaining what it is and how to use it. API documentation should at least contain the below items with sufficient description. It is strongly recommended to use Open API Specification (OAS) or similar prevalent API standards

Table 15 API Documentation

Documentation Requirements	Description
API Introduction	A short introduction describing the API
Resources	A resource is a piece of data or a collection of data provided by an API in response to a request similar to a row in a database. Provide a short description of each of your API resources so users are clear what they're for and why they should call them. You can read more about what resources are in Roy Fielding's dissertation.
Endpoints and methods	Within the resource description, list all of the endpoints related to that resource. The endpoints are the paths a user will use (or call) to access or manipulate the resource. The API you're documenting is likely developed to accept standard methods in resource calls, for example GET, PUT, POST or DELETE. Even though these methods are standard, you should still list the methods applicable to the API you're documenting and describe their functionality.
Parameters	Parameters are optional filters that affect what the API will return. Next to the documented endpoints, you should list any parameters for that endpoint with a short description. If you have different types of parameters, for example, header and path parameters, you may find it useful to group them by type.

Documentation Requirements	Description
Example requests and responses	<p>One of the most useful things in an API reference is example code. For each endpoint, you should provide an example request, with example parameters (if they are available for that endpoint).</p> <p>The request is usually published as a code snippet using curl, but it can be useful to show the request in different programming languages if available.</p> <p>Service Provider should also publish an example response as a code snippet, in the same programming language as the example request. It should show the exact response a user can expect from the example request.</p> <p>Be aware that your examples will not tell service consumers what the fields mean, for example if the fields are optional or mandatory, or if they carry any constraints. If this information is not clear from the example, consider including explanations alongside your example.</p>
Error codes	<p>The Service Provider should include specific error responses associated with an endpoint close to the endpoint documentation or publish them together at the end of your API reference. Even if you only expect standard responses such as 400 or 200, you should interpret their specific meaning to your API.</p>
Authentication	<p>If the Service Consumer needs to authenticate before they use an API, Service Providers should include a section explaining the API authentication. For example, the API documentation includes a separate section on how to get and send an API key.</p>
Authorization	<p>If access to parts of your API requires authorization, have a section in your documentation explaining how a user can gain access. You can then link to this section from the endpoints or resources that require authorization.</p>
Rate limits	<p>If Service Provider API uses rate (or record) limiting, Service Provider should explain how many requests users can make within a set period. Even if it's unlikely a user will meet the maximum number of requests, you must still explain what will happen if users exceed that limit, including the type of error message they can expect and how to correct that error.</p>
Versioning information	<p>Service Provider should tell users how versioning works for API and version the documentation alongside the API.</p> <p>Each version of your documentation should include a clear introduction explaining what makes it different from the version before. You should include all revision history for your API documentation and make it easy for users to switch between documentation for different versions.</p>

Documentation Requirements	Description
Information Handling, Incident Management and Risk Management	
Availability, Ownership and Depreciation Policies	
Governance Frameworks	e.g PCI compliance for Payment API
API Lifecycle Management	The guidelines should define policies for API Lifecycle management PLANNED --> BETA --> LIVE --> DEPRECATED --> RETIRED and also suggest naming conventions addressing backward incompatible changes.

5.5.1 Generating API Documentation

You can write an API reference by hand or auto generate a reference from comments in the code of the API itself.

There are many tools that let you auto generate an HTML file from developer code comments to display to your users. The benefit of this approach is that when developers update comments in their code, your docs will be updated too.

You'll still need to tidy up the reference information after it's been generated and make sure it fits with any accompanying guidance. Ideally you will have a technical writer to help you do this.

You can use a number of criteria to choose between different auto generation tools. As well as the standard considerations when choosing new technology, you should also consider if the tool:

- supports the programming language you need
- outputs in a suitable format, for example HTML
- supports the format options you might need, for example code samples, tables, images

OpenAPI (formerly known as Swagger) is commonly used in government for RESTful API reference documentation. Alternative tools include:

- WADL
- RAML
- I/O Docs
- API Blueprint
- JSON Schema

5.6. API Other Considerations

- 1) It will be the responsibility of the API Consumers and API Providers to utilize the credentials or system details shared between each other only for the intended purpose by authorized users and systems only.
- 2) It is recommended to get the customer consensus to use his/her information. This can be added to the UI used to provide the service to the user. Moreover, the front desk employees should be educated to inform the user if their information will be accessed on his behalf.
- 3) It is also recommended to have data confidentiality agreement between the entity and the employees involved in processing the information shared through the API Consumer and API Provider
- 4) The maintenance of the API and the consuming application will be the responsibility of respective entities and the maintenance of published services will be the responsibility of API Provider Team.
- 5) Point of Contacts should be made available by entities for coordination of any activities.
- 6) Entities will be responsible for providing resources during the development and testing of services before go-live as well as post go-live.
- 7) API Provider owns and is responsible for the business process/logic and transformations at service level.
- 8) Reports of the service usage will be made available to entities by API Provider.

6. References

We've adapted these guidelines capitalizing on the work done by other digital governments, including:

- Government of Canada (<https://www.canada.ca/en/government/system/digital-government/modern-emerging-technologies/government-canada-standards-apis.html>)
- UK Digital Service (<https://www.gov.uk/guidance/gds-api-technical-and-data-standards>)
- 18F (the United States General Services Administration) (<https://github.com/18F/api-standards>)
- Government of New Zealand (<https://www.digital.govt.nz/standards-and-guidance/technology-and-architecture/application-programming-interfaces-apis/api-implementation-guidance/>)
- Government of the State of Victoria, Australia (<https://github.com/VictorianGovernment/api-design-standards>)
- Guiding Principles of REST (<https://restfulapi.net/>)
- Architectural Styles and the Design of Network-based Software Architectures (<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>)

