STANDARD ST.90

RECOMMENDATION FOR PROCESSING AND COMMUNICATING INTELLECTUAL PROPERTY DATA USING WEB APIS (APPLICATION PROGRAMMING INTERFACES)

Version 2.0

*Revision approved by the Committee on WIPO Standards (CWS)
at its thirteenth session on November 14, 2025*

TABLE OF CONTENTS

ANNEXES

STANDARD ST.90

RECOMMENDATION FOR PROCESSING AND COMMUNICATING INTELLECTUAL PROPERTY DATA USING WEB APIS (APPLICATION PROGRAMMING INTERFACES)

Version 2.0

*Revision approved by the Committee on WIPO Standards (CWS)
at its thirteenth session on November 14, 2025*

INTRODUCTION

1. This Standard provides recommendations on Application Programming Interfaces (APIs) to facilitate the processing and exchange of Intellectual Property (IP) data in a harmonized way over the Web.

2. This Standard is intended to:

- ensure consistency by establishing uniform web service design principles;
- improve data interoperability among web service partners;
- encourage reusability through unified design;
- promote data naming flexibility across business units through a clearly defined namespace policy in associated XML resources;
- promote secure information exchange;
- offer appropriate internal business processes as value-added services that can be used by other organizations; and
- integrate its internal business processes and dynamically link them with business partners.

DEFINITIONS AND TERMINOLOGY

3. For the purpose of this Standard, the expressions:

- “Hyper Text Transfer Protocol (HTTP)” is intended to refer to the application protocol for distributed, collaborative, and hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. HTTP functions as a request–response protocol in the service oriented computing model;
- “Application Programming Interfaces” (API) means software components that provide a reusable interface between different applications that can easily interact to exchange data;
- “Representational State Transfer (REST)” describes a set of architectural principles by which data can be transmitted over a standardized interface, i.e., HTTP. REST does not contain an additional messaging layer and focuses on design rules for creating stateless services;
- “Simple Object Access Protocol (SOAP)” means a protocol for sending and receiving messages between applications without confronting interoperability issues. SOAP defines a standard communication protocol (set of rules) specification for XML-based message exchange. SOAP uses different transport protocols, such as HTTP and SMTP. The standard protocol HTTP makes it easier for SOAP model to tunnel across firewalls and proxies without any modifications to the SOAP protocol;
- “Web Service” means a method of communication between two applications or electronic machines over the World Wide Web (WWW) and Web Services are of two kinds: REST and SOAP;
- “RESTful Web API” means a set of Web Services based on REST architectural paradigm and typically use JSON or XML to transmit data;

- "SOAP Web API" means a set of SOAP Web Services based on SOAP and mandate the use of XML as the payload format;
- "Web Services Description Language (WSDL)" means a W3C Standard that is used with the SOAP protocol to provide a description of a Web Service. This includes the methods a Web Service uses, the parameters it takes and the means of locating Web Services etc.;
- RESTful API Modelling Language (RAML) refers to a language which allows developers to provide a specification of their API;
- Open API Specification (OAS) refers to a language which allows developers to provide a specification of their API;
- "Service Contract" (or Web Service Contract) means a document that expresses how the service exposes its capabilities as functions and resources offered as a published API by the service to other software programs; the term "REST API documentation" is interchangeably used for the Service Contract for RESTful Web APIs;
- "Service Provider" means a Web Service software exposing a Web Service;
- "Service Consumer" means the runtime role assumed by a software program when it accesses and invokes a service. More specifically, when the program sends a message to a service capability expressed in the service contract. Upon receiving the request, the service begins processing and it may or may not return a corresponding response message to the service consumer;
- "Camelcase" is either the lowerCamelCase (e.g., applicantName), or the UpperCamelCase (e.g., ApplicantName) naming convention;
- Kebab-case is one of the naming conventions where all are lowercase with hyphens "-" separating words, for example a-b-c;
- "Open Standards" means the standards that are made available to the general public and are developed (or approved) and maintained via a collaborative and consensus driven process. "Open Standards" facilitate interoperability and data exchange among different products of services and are intended for widespread adoption;
- Uniform Resource Identifier (URI) identifies a resource and Uniform Resource Locator (URL) is a subset of the URIs that include a network location;
- "Entity Tag (ETag)" means an opaque identifier assigned by a web server to a specific version of a resource found at a URL. If the resource representation at that URL ever changes, a new and different ETag is assigned. ETags can be compared quickly to determine whether two representations of a resource are the same;
- "Service Registry" means a network-based directory that contains available services;
- "RMM" refers to the Richardson Maturity Model a measure of REST API maturity using a scale ranging from 0-3; and
- "Semantic Versioning" means a versioning scheme where a version is identified by the version number MAJOR.MINOR.PATCH, where:
  - MAJOR version when you make incompatible API changes;
  - MINOR version when you add functionality in a backwards-compatible manner; and
  - PATCH version when you make backwards-compatible bug fixes.

4. In terms of conformance in design rules the following keywords should be interpreted, in the same manner as defined in paragraph 8 of WIPO ST.96, that is:

- MUST: An equivalent to "REQUIRED" or "SHALL", means that the definition is an absolute requirement of the specification;
- MUST NOT: Equivalent to "SHALL NOT", means that the definition is an absolutely prohibited by the specification;

- SHOULD: Equivalent to “RECOMMENDED”, means that there may exist valid reasons for ignoring this item, but the implications of doing so need to be fully considered;
- SHOULD NOT: Equivalent to “NOT RECOMMENDED”, means that there may exist valid reasons where this behavior may be acceptable or even useful but the implications of doing so need to be carefully considered; and
- MAY: Equivalent to “OPTIONAL”, means that this item is truly optional, and is only provided as one option selected from many.

NOTATIONS

General notations

5. The following notations are used throughout this document:

- <>: Indicates a placeholder descriptive term that, in implementation, will be replaced by a specific instance value;
- “ ”: Indicates that the text included in quotes must be used verbatim in implementation;
- { }: Indicates that the items are optional in implementation; and
- Courier New font: Indicates keywords or source code.

6. The URLs provided within this Standard are for example purposes only and are not live.

Rule identifiers

7. All design rules are normative. Design rules are identified through a prefix of [XX-nn] or [XXY-nn].

(a) The value “XX” is a prefix to categorize the type of rule as follows:

- WS for SOAP Web API design rules;
- RS for RESTful Web API design rules; and
- CS for both SOAP and RESTful WEB API design rule.

(b) The value “Y” is used only for RESTful design rules and provides further granularity on the type of response that the rule is related to:

- “G” indicates it is a general rule for both JSON and XML response;
- “J” indicates it is for a JSON response; and
- “X” indicates it is an XML response.

(c) The value “nn” indicates the next available number in the sequence of a specific rule type. The number does not reflect the position of the rule, in particular, for a new rule. A new rule will be placed in the relevant context. For example, the rule identifier [WS-4] identifies the fourth SOAP Web API design rule. The rule [WS-4] can be placed between rules [WS-10] and [WS-11] instead of following [WS-3] if that is the most appropriate location for this rule.

(d) The rule identifier of the deleted rule will be kept while the rule text will be replaced with “Deleted”.

SCOPE

8. This Standard aims to guide the Intellectual Property Offices (IPOs) and other Organizations that need to manage, store, process, exchange and disseminate IP data using Web APIs. It is intended that by using this Standard, the development of Web APIs can be simplified and accelerated in a harmonized manner and interoperability among Web APIs can be enhanced.

9. This Standard intends to cover the communications between IPOs and their applicants or data users, and between IPOs through connections between devices-to-devices and devices-to-software applications.

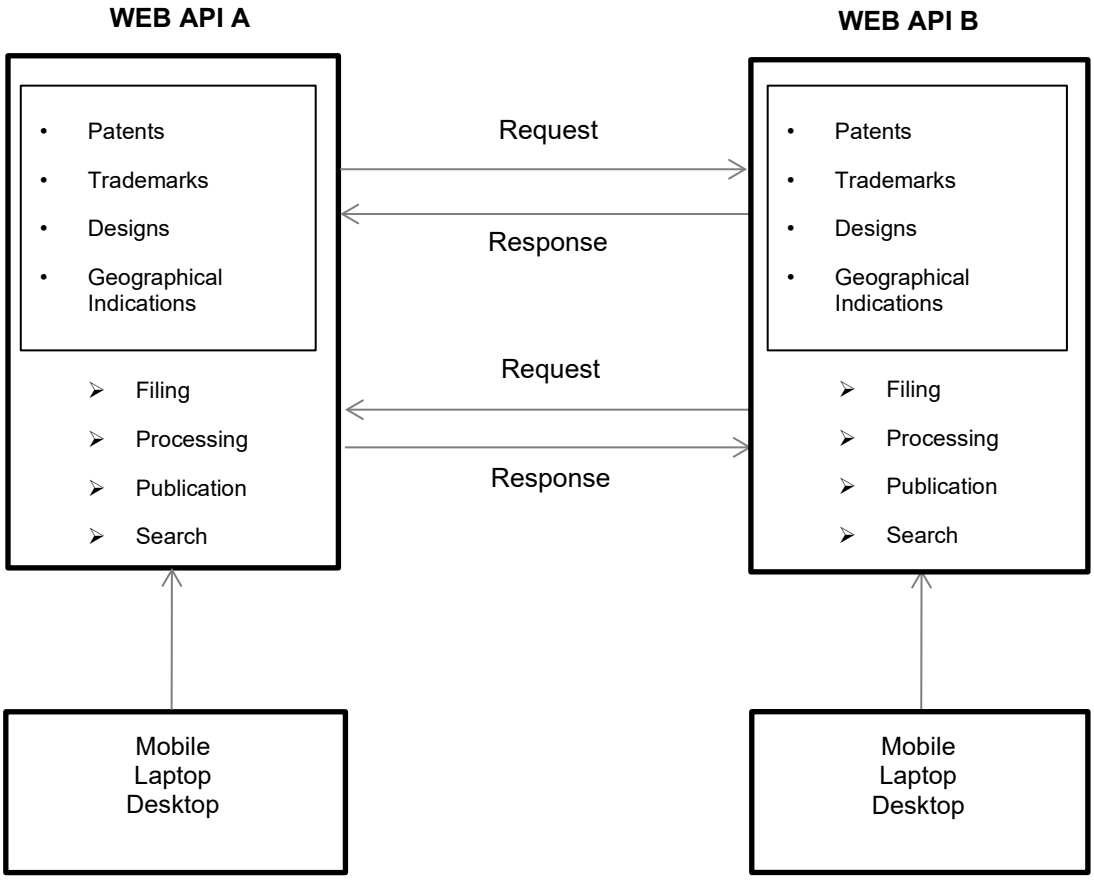


Fig. 1 Scope of the Standard

10. This Standard is to provide a set of design rules and conventions for RESTful and SOAP Web APIs; list of IP data resources which will be exchanged or exposed; and model API documentation or service contract, which can be used for customization, describing message format, data structure and data dictionary in JSON based on WIPO Standard ST.97 and/or XML format based on WIPO Standard ST.96.

11. This Standard provides model Service Contracts for SOAP Web APIs using WSDL and, for RESTful Web APIs using the REST API Modeling Language (RAML) and Open API Specification (OAS). A Service Contract also defines or refers to data types for interfaces (see the Section “Data Type Convention” below). This Standard recommends three types of interfaces: REST-XML (XSD), REST-JSON and SOAP-XML (XSD).

12. This Standard excludes the following:

(a) Binding to specific implementation technology stacks and commercial off-the-shelf (COTS) products

(b) Binding to specific architectural designs (for example, Service Oriented Architecture (SOA) or Microservice Oriented Architecture (MOA)); and

(c) Binding to specific algorithms such as algorithms for the calculation of ETag, i.e., calculation of a unique identifier for a specific version of a resource (for example, used for caching).

WEB API DESIGN PRINCIPLES

13. Both RESTful Web APIs and SOAP Web APIs have proven their ability to meet the demands of big organizations as well as to service the small-embedded applications in production. When choosing between RESTful and SOAP, the following aspects can be considered:

- Security, e.g., SOAP has WS-Security while REST does not specify any security patterns;
- ACID Transaction, e.g., SOAP has WS-AT specification while REST does not have a relevant specification;
- Architectural style, e.g., Microservices and Serverless Architecture Style use REST while SOA uses SOAP web services;
- Flexibility;
- Bandwidth constraints; and
- Guaranteed delivery, e.g., SOAP offers WS-RM while REST does not have a relevant specification.

14. The following service-oriented design principles should be respected when a Web API is designed:

(a) Standardized Service Contract: Standardizing the service contracts is the most important design principle because the contracts allow governance and a consistent service design. A service contract should be easy to implement and understand. A service contract consists of metadata that describes how the service provider and consumer will interact. Metadata also describes the conditions under which those parties are entitled to engage in an interaction. It is recommended that service contracts include:

- Functional requirements: What functionality the Service provides and what data it will return, or typically a combination of the two;
- Non-functional requirements: Information about the responsibility of the providers for providing their functionality and/or data, as well as the expected responsibilities of the consumers of that information and what they will need to provide in return. For example, a consumer's availability, security, and other quality of service considerations.

(b) Service Loose Coupling: Clients and services should evolve independently. Applying this design principle requires:

- Service versioning: Consumers bound to a Web API version should not take the risk of unexpected disruptions due to incompatible API changes; and
- The service contract should be independent of the technology details.

(c) Service Abstraction: The service implementation details should be hidden. The API Design should be independent of the strategies supported by a server. For example, for the REST Web Service, the API resource model should be decoupled from the entity model in the persistence layer;

(d) Service Statelessness: Services should be scalable;

(e) Service Reusability: A well-designed API should provide reusable services with generic contracts. In this regard, this Standard provides a model service contract;

(f) Service Autonomy: The Service functional boundaries should be well defined;

(g) Service Discoverability: Services should be effectively discovered and interpreted;

(h) Service Composability: Services can be used to compose other services;

(i) Using Standards as a Foundation: The API Should follow industry standards (such as IETF, ISO, and OASIS) wherever applicable, naturally favoring them over locally optimized solutions; and

(j) Pick-and-choose Principle: It is not required to implement all the API design rules. The design rules should be chosen based on the implementation of each concrete case.

15. In addition, the following principles should be respected especially with regard to the RESTful Web APIs:

(a) Cacheable: Responses explicitly indicate their cache ability;

(b) Resource identification in requests: Individual resources are identified in requests; for example, using URIs in Web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client;

(c) Hypermedia as the engine of application state (HATEOAS): Having accessed an initial URI for the REST application, analogous to an individual accessing the home page of a website, a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs;

(d) Resource manipulation through representations: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource;

(e) Self-descriptive messages: Each message includes enough metadata to describe how to process the message content;

(f) Web API should follow HTTP semantics such as methods, errors etc.;

(g) Available to the public: Design with the objective that the API will eventually be accessible from the public internet, even if there are no plans to do so at the moment;

(h) Common authentication: Use a common authentication and authorization pattern, preferably based on existing security components, in order to avoid creating a bespoke solution for each API;

(i) Least Privilege: Access and authorization should be assigned to API consumers based on the minimal amount of access they need to carry out the functions required;

(j) Maximize Entropy: The randomness of security credentials should be maximized by using API Keys rather than username and passwords for API authorization, as API Keys provide an attack surface that is more challenging for potential attackers; and

(k) Performance versus security: Balance performance with security with reference to key life times and encryption / decryption overheads.

RESTFUL WEB API

16. A RESTful Web API allows requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations.

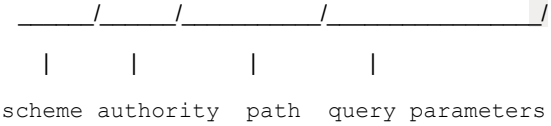
URI components

17. RESTful Web API s use URIs to address resources. According to RFC 3986, an URI syntax should be defined as follows:

```
URI = <scheme> "://" <authority> "/" <path> {"?" query}

authority = {userinfo@}host{:port}
```

For example, https://wipo.int/api/v1/patents?sort=id&offset=10



18. The forward slash “/” character is used in the path of the URI to indicate a hierarchical relationship between resources but the path must not end with a forward slash as it does not provide any semantic value and may cause confusion.

[RSG-01] The forward slash character “/” MUST be used in the path of the URI to indicate a hierarchical relationship between resources but the path MUST NOT end with a forward slash.

19. URIs are case sensitive except for the scheme and host parts. For example, although `https://wipo.int/api/my-resources/uniqueId` and `https://wipo.int/api/my-resources/uniqueId` are the same, `https://wipo.int/api/my-resources/uniqueid` is not. For the resource names, the kebab-case and the lowerCamelCase conventions provide good readability and maps the resource names to the entities in the programming languages with simple transformation. For the query parameters, the lowerCamelCase should be used. For example, `https://wipo.int/api/v1/inventors?firstName=John`. Resource names and query parameters are all case sensitive. Note, that resource names and query parameter names may be abbreviated.

20. A RESTful Web API may have arguments:

- In the query parameter; for example, /inventors?id=1;
- In the URI path segment parameter, for example, /inventors/1; and
- In the request payload such as part of a JSON body.

21. Except for the aforementioned argument types, which are part of the URI, an argument can also be part of the request payload.

Example JSON request payload

```
POST https://wipo.int/api/v1/inventors

 Request body:

{
   "firstName": "John"
}
```

[RSG-02] Resource names MUST be consistent in their naming pattern.

[RSG-03] Resource names in the request SHOULD use kebab-case naming conventions and they MAY be abbreviated.

[RSG-04] Query parameters MUST be consistent in their naming pattern.

[RSG-05] Query parameters SHOULD use the lowerCamelCase convention and they MAY be abbreviated.

22. A Web API endpoint must comply with `IETF RFC 3986` and should avoid potential collisions with page URLs for the website hosted on the root domain. A Web API needs to have one exact entry point to consolidate all requests. In general, there are two patterns of defining endpoints:

- As the first path segment of the URI, for example: `https://wipo.int/api/v1/`; and
- As subdomain, for example: `https://api.wipo.int/v1/`

[RSG-06] The URL pattern for a Web API MUST contain the word "`api`" in the URI.

23. Matrix parameters are an indication that the API is complex with multiple levels of resources and sub-resources. This goes against the service-oriented design principles, previously defined. Moreover, matrix parameters are not standard as they apply to a particular path element while query parameters apply to the request as a whole. An example of matrix parameters is the following: `https://api.wipo.int/v1/path;param1=value1;param2=value2`.

[RSG-07] Matrix parameters MUST NOT be used.

Status codes

24. A Web API must consistently apply HTTP status codes as described in IETF RFCs. HTTP status codes should be used among the ones listed in the standard HTTP status codes (as defined in RFC 9110 and registered by IANA) reproduced in Annex V.

[RSG-08] A Web API MUST consistently apply HTTP status codes as described in IETF RFCs.

[RSG-09] The recommended codes in Annex V SHOULD be used by a Web API to classify the error.

Pick-and-choose principle

25. A Service Contract should be tolerant to unexpected parameters (in the request, using query parameters) but raise an error in case of malformed values on expected parameters.

[RSG-10] If the API detects invalid input values, it MUST return the HTTP status code "`400 Bad Request`". The error payload MUST indicate the erroneous value.

[RSG-11] If the API detects syntactically correct argument names (in the request or query parameters) that are not expected, it SHOULD ignore them.

[RSG-12] If the API detects valid values that require features not supported by the server, it MUST return the HTTP status code "`501 Not Implemented`".

Resource model

26. An IP data model should be divided into bounded contexts following a domain-driven design approach. Each bounded context must be mapped to a resource. According to the design principles, a Web API resource model should be decoupled from the data model. A Web API should be modeled as a resource hierarchy to leverage the hierarchical nature of the URI to imply structure (association or composition or aggregation), where each node is either a simple (single) resource or a collection of resources.

27. In this hierarchical resource model, the nodes in the root are called 'top-level nodes' and all of the nested resources are called 'sub-resources'. Sub-resources should be used only to imply compositions, i.e., resources that cannot be top-level resources, otherwise there would be multiple way of retrieving the same entities. Such sub-resources, implying association, are called sub-collections. The other hierarchical structures, i.e., association and aggregation, should be avoided to avoid complex APIs and duplicate functionality.

28. The endpoint always determines the type of the response. For example, the endpoint `https://wipo.int/api/v1/patents` always returns responses regarding patent resources. The endpoint `https://wipo.int/api/v1/patents/1/inventor` always returns responses regarding inventor resources. However, the endpoint `https://wipo.int/api/v1/inventors` is not allowed because the inventor resource cannot be standalone.

29. Only top-level resources, i.e., with a maximum of one level should be used, otherwise these APIs will be very complex to implement. For example, `https://wipo.int/api/v1/patents?inventorId=12345` should be used instead of `https://wipo.int/api/v1/inventors/12345/patents`.

[RSG-13] A Web API SHOULD only use top-level resources. If there are sub-resources, they should be collections and imply an association. An entity should be accessible as either top-level resource or sub-resource but not using both ways.

[RSG-14] If a resource can be stand-alone, it MUST be a top-level resource, or otherwise a sub-resource.

[RSG-15] Query parameters MUST be used instead of URL paths to retrieve nested resources.

30. There are types[1] of Web APIs: the CRUD (Create, Read, Update, and Delete) Web API and the Intent Web API. CRUD Web APIs model changes to a resource, i.e., create/read/update/delete operations. Intent Web APIs by contrast model business operations, e.g., renew/register/publish. CRUD operations should use nouns and Intent Web APIs should use verbs for the resource names. CRUD Web APIs are the most common but both can be combined for example, the service consumer could use an Intent Web API modeling business operation, which would orchestrate the execution of one or more CRUD Web APIs service operations. Using CRUD Web API, the service caller has to orchestrate the business

[1] Alternatively, we could classify APIs according to their archetype. See for instance: "REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces"

logic but with Intent Web APIs it is the service provider who orchestrates the business logic. CRUD Web APIs are not atomic when compared with Intent Web APIs[2].

For example, a trademarks owner wants to renew the ones that will expire soon (for example, on `yyyy-mm-dd`). This is a combination of the following business operations:

- Retrieve marks that will expire on yyyy-mm-dd; and
- Renew the retrieved marks by their international registration number.

Using a CRUD Web API the previous business operations would be modeled with a non-atomic process, requiring two actions such as:

Step 1: Get all the trademarks in XML format[3] that belong to the holder with the name John Smith and will expire, for example, on 2018-12-31:

```
GET /api/v1/trademarks? holderFullName=John%20Smith&expiryDate=2018-12-31. HTTP/1.1

Host: wipo.int

Accept: application/xml
```

The following example HTTP response is returned:

```
HTTP/1.1 200 OK

Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>

<tmk:TrademarkBag xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:tmk="http://www.wipo.int/standards/XMLSchema/ST96/Trademark"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Trademark
TrademarkBag.xsd">

      <tmk:Trademark xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:tmk="http://www.wipo.int/standards/XMLSchema/ST96/Trademark"
com:operationCategory="Delete"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Trademark
Trademark.xsd">

            ...

            <com:RegistrationNumber>

                  <com:IPOfficeCode>IT</com:IPOfficeCode>


      <com:ST13ApplicationNumber>000000000000001</com:ST13ApplicationNumber>
```

[2] An Intent API also enables the application of the Command Query Responsibility Segregation (CQRS) pattern. CQRS is a pattern, where you can use a different model to update information than the model you use to read information. The rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that is not beneficial.

[3] JSON example is skipped since it does not add any value in this case.

```
            </com:RegistrationNumber>

            ...

            <com:ExpiryDate>2018-12-31</com:ExpiryDate>

            ...

      </tmk:Trademark>

      ...

</tmk:TrademarkBag>
```

Step 2: Submit a trademark renewal request for each trademark retrieved in the previous step (depicting here only the first renewal request):

```
POST /api/v1/trademarks/renewalRequests HTTP/1.1

Host: wipo.int

Accept: application/xml

Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>

<tmk:MadridRenewal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:tmk="http://www.wipo.int/standards/XMLSchema/ST96/Trademark"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Trademark
MadridRenewal.xsd">

      ...

      <com:InternationalRegistrationNumber>000000000000001</com:InternationalRegistr
ationNumber>

      ...

</tmk:MadridRenewal>
```

The previous example could also be modeled with an atomic service call using an Intent Web API such as[4]:

```
POST /api/v1/trademarks/findAndRenew?holderFullName=john%20smith&expiryDate=2018-12-
31

Host: wipo.int
```

31. The type of Web API should then place constraints on how the resources are named to provide an indication on which is being used. Note, that resource names that are localized due to business requirements may be in other languages.

[RSG-16] Resource names SHOULD be nouns for CRUD Web APIs and verbs for Intent Web APIs.

[4] The element `InternationalRegistrationNumber` has been removed from the payload to denote all the IRNs. The ST.96 should be not used or relaxed since the example here extends the uses cases allowed from ST.96.

[RSG-17] If resource name is a noun, it SHOULD always use the plural form. Irregular noun forms SHOULD NOT be used. For example, /persons should be used instead of /people.

[RSG-18] Resource names, segment and query parameters SHOULD be composed of words in the English language, using the primary English spellings provided in the Oxford English Dictionary.

Supporting multiple formats

32. Different service consumers may have differing requirements for the data format of the service responses. The media type of the data should be decoupled from the data itself, allowing the service to support a range of media types. Therefore, a Web API must support content type negotiation using the request HTTP header `Accept` and the response HTTP header `Content-Type` as required by IETF RFC 9110. For example, for requesting data in JSON format the header `Accept` should be `Accept: application/json` and for data in XML format the `Accept` should be `Accept: application/xml`. Likewise, for the header `Content-Type`. Additionally, a Web API may support other ways of content type negotiation such as query parameter (for example, `?format`) or URL suffix (for example `.json`).

[RSG-19] A Web API SHOULD use for content type negotiation the request HTTP header `Accept` and the response HTTP header `Content-Type`.

33. APIs must support XML or JSON requests and responses. For XML, responses should be compliant with WIPO Standard using XML such as ST.96 and for JSON, responses should be compliant with WIPO Standard ST.97. A consistent mapping between these two formats should be used.

[RSG-20] A Web API MUST support content type negotiation following IETF RFC 9110.

[RSG-21] JSON format MUST be assumed when no specific content type is requested.

[RSG-22] A Web API SHOULD return the status code "`406 Not Acceptable`" if a requested format is not supported.

[RSG-23] A Web API SHOULD reject requests containing unexpected content type headers with the HTTP status code "`406 Not Acceptable`" or "`415 Unsupported Media Type`".

[RSG-24] The requests and responses (naming convention, message format, data structure, and data dictionary) SHOULD refer to WIPO Standard ST.96 for XML or WIPO Standard ST.97 for JSON.

[RSJ-25] JSON object property names SHOULD be provided in lowerCamelCase, e.g., applicantName.

[RSX-26] XML component names SHOULD be provided in UpperCamelCase.

[RSG-27] A Web API MUST support at least XML or JSON.

HTTP methods

34. HTTP Methods are a type of function provided by a uniform contract to process resource identifiers and data. HTTP Methods must be used as they were intended to according the standardized semantics as specified in IETF RFC 9110 and 5789, namely:

- `GET` – retrieve data
- `HEAD` – like `GET` but without a response payload
- `POST` – submit new data
- `PUT` – update
- `PATCH` – partial update
- `DELETE` – delete data
- `TRACE` – echo

– `OPTIONS` – query verbs that the server supports for a given URL

35. The uniform contract establishes a set of methods to be used by services within a given collection or inventory. HTTP Methods tunneling may be useful when HTTP Headers are rejected by some firewalls.

36. HTTP Methods may follow the 'pick-and-choose' principle, which states that only the functionality needed by the target usage scenario should be implemented. Some proxies support only `POST` and `GET` methods. To overcome these limitations, a Web API may use a POST method with a custom HTTP header "tunneling" the real HTTP method.

[RSG-28] HTTP Methods MUST be restricted to the HTTP standard methods `POST`, `GET`, `PUT`, `DELETE`, `OPTIONS`, `PATCH`, `TRACE` and `HEAD`, as specified in IETF RFC 9110 and 5789.

[RSG-29] HTTP Methods MAY follow the pick-and-choose principle, which states that only the functionality needed by the target usage scenario should be implemented.

[RSG-30] Some proxies support only `POST` and `GET` methods. To overcome these limitations, a Web API MAY use a `POST` method with a custom HTTP header "tunneling" the real HTTP method. The custom HTTP header `X-HTTP-Method` SHOULD be used.

[RSG-31] If a HTTP Method is not supported by the target resource, the HTTP status code "`405 Method Not Allowed`" SHOULD be returned.

37. In some use cases, multiple operations should be supported at once.

[RSG-32] A Web API SHOULD support batching operations (aka bulk operations) in place of multiple individual requests to achieve latency reduction. The same semantics should be used for HTTP Methods and HTTP status codes. The response payload SHOULD contain information about all batching operations. If multiple errors occur, the error payload SHOULD contain information about all the occurrences (in the details attribute). All bulk operations SHOULD be executed in an atomic operation.

GET

38. According to IETF RFC 9110, the HTTP protocol does not place any prior limit on the length of a URI. On the other hand, servers should be cautious about depending on URI lengths above 255 bytes, because some older client or proxy implementations may not properly support these lengths. In the case where this limit is exceeded, it is recommended that named queries are used. Alternatively, a set of rules which determine how to convert between and GET and a POST must be specified. According to the IETF RFC 9110, a GET request must be idempotent, in that the response will be the same no matter how many times the request is run.

[RSG-33] For an end point which fetches a single resource, if a resource is not found, the method `GET` MUST return the status code "`404 Not Found`". Endpoints which return lists of resources will simply return an empty list.

[RSG-34] If a resource is retrieved successfully, the `GET method` MUST return "`200 OK`".

[RSG-35] A `GET` request MUST be idempotent.

[RSG-36] When the URI length exceeds the 255 bytes, the `POST` method SHOULD be used instead of `GET` due to practical `GET` limitations, or else create named queries if possible.

HEAD

39. When a client needs to learn information about an operation, they can use `HEAD`. `HEAD` gets the HTTP header you would get if you made a `GET` request, but without the body. This lets the client determine caching information, what content-type would be returned, what status code would be returned. A `HEAD` request MUST be idempotent according to the IETF RFC 9110.

[RSG-37] A `HEAD` request MUST be idempotent.

[RSG-38] Some proxies support only `POST` and `GET` methods. A Web API SHOULD support a custom HTTP request header to override the HTTP Method in order to overcome these limitations.

POST

40. When a client needs to create a resource, they can use `POST`. For example, the following HTTP request submits a patent application request.

For example, the following submits a patent application request.

Example with XML payloads based on ST.96

The clients submit the patent application request as XML:

```
POST /v1/patents/applications HTTP/1.1

Host: wipo.int

Accept: application/xml

Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>

<pat:ApplicationBody xmlns="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:pat="http://www.wipo.int/standards/XMLSchema/ST96/Patent"
com:languageCode="pl" com:receivingOffice="ST" com:st96Version="V5_0"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Patent
ApplicationBody_V5_0.xsd">

      ...

</pat:ApplicationBody>
```

The following HTTP response is returned to denote the successful submission of the patent application:

```
HTTP/1.1 201 Created

Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>

<pat:ApplicationBody xmlns="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:pat="http://www.wipo.int/standards/XMLSchema/ST96/Patent"
com:languageCode="pl" com:receivingOffice="ST" com:st96Version="V5_0"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Patent
ApplicationBody_V5_0.xsd" applicationBodyStatus=”pending”>

      ...

</pat:ApplicationBody>
```

Example with JSON payloads based on ST.97

The clients submit the patent application request as JSON:

```
POST /v1/patents/applications HTTP/1.1
Host: wipo.int
Accept: application/json
Content-Type: application/json
{
      " applicationBody ": {
            ...
      }
}
```

The following HTTP response is returned to denote the successful submission of the patent application:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
      " applicationBody ": {
            "applicationBodyStatus" : "pending",
            ...
      }
}
```

[RSG-39] A `POST` request MUST NOT be idempotent according to the IETF RFC 9110.

[RSG-40] If the resource creation was successful, the HTTP header `Location` SHOULD contain a URI (absolute or relative) pointing to a created resource.

[RSG-41] If the resource creation was successful, the response SHOULD contain the status code "`201 Created`".

[RSG-42] If the resource creation was successful, the response payload SHOULD by default contain the body of the created resource, to allow the client to use it without making an additional HTTP call.

PUT

41. When a client needs to replace an existing resource entirely, they can use `PUT`. Idempotent characteristics of `PUT` should be taken into account. A `PUT` request has an update semantic (as specified in IETF RFC 9110), and an insert semantic.

[RSG-43] A `PUT` request MUST be idempotent.

[RSG-44] If the target resource is not found and the server does not allow creation at the given URI, `PUT` MUST return the status code "`404 Not Found`". If the server allows creation, `PUT` MUST return the status code "`201 Created`".

[RSG-45] If a resource is updated successfully, `PUT` MUST return the status code "`200 OK`" if the updated resource is returned or a "`204 No Content`" if it is not returned.

PATCH

42. When a client requires a partial update, they can use `PATCH`. Idempotent characteristics of `PATCH` should be taken into account.

For example, the following request updates only a patent language given its number:

```
PATCH /api/v1/patents/publications/100000000000001 HTTP/1.1

Host: wipo.int

If-Match:456

Content-Type: application/merge-patch+json

{ "languageCode": "en" }
```

43. PATCH must not be idempotent according to IETF RFC 9110. In order to make it idempotent, the API may follow the IETF RFC 5789 suggestion of using optimistic locking.

[RSG-46] By default, a `PATCH` request MUST NOT be idempotent.

[RSG-47] If a Web API implements partial updates, idempotent characteristics of `PATCH` SHOULD be taken into account. In order to make it idempotent the API MAY follow the IETF RFC 5789 suggestion of using optimistic locking.

[RSG-48] If a resource is not found `PATCH` MUST return the status code "`404 Not Found`".

[RSJ-49] If a Web API implements partial updates using `PATCH`, it MUST use the JSON Merge Patch format to describe the partial change set, as described in IETF RFC 7396, by using the content type `application/merge-patch+json`.

DELETE

44. When a client needs to delete a resource, they can use `DELETE`. A `DELETE` request must be idempotent according to the IETF RFC 9110

[RSG-50] A `DELETE` request MUST be idempotent.

[RSG-51] If a resource is not found, `DELETE` MUST return the status code "`404 Not Found`".

[RSG-52] If a resource is deleted successfully, `DELETE` MUST return the status "`200 OK`" if the deleted resource is returned or "`204 No Content`" if it is not returned.

TRACE

45. The `TRACE` method does not carry API semantics and is used for testing and diagnostic information according to IETF RFC 9110, for example for testing a chain of proxies. `TRACE` allows the client to see what is being received at the other end of the request chain and uses that data. A `TRACE` request MUST be idempotent according to the IETF RFC 9110.

[RSG-53] The final recipient is either the origin server or the first proxy or gateway to receive a `Max-Forwards` value of zero in the request. A `TRACE` request MUST NOT include a body.

[RSG-54] A `TRACE` request MUST be idempotent.

[RSG-55] The value of the `Via` HTTP header field MUST act to track the request chain.

[RSG-56] The `Max-Forwards` HTTP header field MUST be used to allow the client to limit the length of the request chain.

[RSG-57] If the request is valid, the response SHOULD contain the entire request message in the response body, with a `Content-Type` of "`message/http`".

[RSG-58] Responses to `TRACE` MUST NOT be cached.

[RSG-59] The status code "`200 OK`" SHOULD be returned to `TRACE`.

OPTIONS

46. When a client needs to learn information about a Web API, they can use `OPTIONS`. `OPTIONS` do not carry API semantics. An `OPTIONS` request MUST be idempotent according to the IETF RFC 9110, Custom HTTP Headers.

[RSG-60] An `OPTIONS` request MUST be idempotent.

47. It is a common practice for a Web API using custom HTTP headers to provide "`X-`" as a common prefix, which RFC 6648 deprecates and discourages to use.

[RSG-61] Custom HTTP headers starting with the "`X-`" prefix SHOULD NOT be used.

[RSG-62] Custom HTTP headers SHOULD NOT be used to change the behavior of HTTP Methods unless it is to resolve any existing technical limitations (e.g., see [RSG-39]).

[RSG-63] The naming convention for custom HTTP headers is `<organization>-<header name>`, where `<organization>` and `<header>` SHOULD follow the kebab-case convention.

48. According to the service-oriented design principles, clients and services should evolve independently. Service versioning enables this. Common implementations of service versioning are: Header Versioning (by using a custom header), Query string versioning (e.g., `?v=v1`), Media type versioning (e.g., `Accept: application/vnd.v1+json`) and URI versioning (e.g., `/api/v1/inventors`).

[RSG-64] A Web API SHOULD support a single method of service versioning using URI versioning, e.g., `/api/v1/inventors` or Header versioning, e.g., `Accept-version: v1` or Media type versioning, e.g., `Accept: application/vnd.v1+json`. Query string versioning SHOULD NOT be used.

49. According to the service-oriented design principles, service providers and consumers should also evolve independently. The service consumer should not be affected by minor (backward compatible) changes by the service provider. Therefore, service versioning should use only major versions. For internal non-published APIs (for example, for development and testing) minor versions may also be used such as Semantic Versioning.

[RSG-65] A versioning-numbering scheme SHOULD be followed considering only the major version number (e.g., `/v1`).

50. Service endpoint identifiers include information that can change over time. It may not be possible to replace all references to an out-of-date endpoint, which can lead to the service consumer being unable to further interact with the service endpoint. Therefore, the service provider may return a redirection response. The redirection may be temporary or permanent. The following HTTP status codes are available:

	Permanent	**Temporary**
Allows changing the request method from POST to GET	301	302
Doesn't allow changing the request method from POST to GET	308	307

Since `301` and `302` are more generic they are preferred to increase flexibility and overcome any unnecessary complexity.

[RSG-66] API service contracts MAY include endpoint redirection feature. When a service consumer attempts to invoke a service, a redirection response may be returned to tell the service consumer to resend the request to a new endpoint. Redirections MAY be temporary or permanent:

- Temporary redirect: Using the HTTP response header `Location` and the HTTP status code "`302 Found`" according to IETF RFC 9110; or
- Permanent redirect: Using the HTTP response header `Location` and the HTTP status code "`301 Moved Permanently`" according to IETF RFC 9110.

51. As an API is evolving, it will pass through a series of major phases: planning and designing, developing, testing, deploying and retiring. Rather than providing recommendations for the time periods that an API should preferably remain in a particular phase, it is preferable that the Organization or Service providers instead publish their API lifecycle strategy. A template which provides the basic components which define a life cycle strategy in provided in Annex VII.

[RSG-67] API lifecycle strategies SHOULD be published by the developers to assist users in understanding how long a version will be maintained.

Data query patterns

Pagination options

52. Pagination is a mechanism for a client to retrieve data in pages. Using pagination, we prevent overwhelming the service provider with resource demanding requests according to the design principles. The server should enforce a default page size in case the service consumer has not specified one. Paginated requests may not be idempotent, i.e., a paginated request does not create a snapshot of the data.

[RSG-68] A Web API SHOULD support pagination.

[RSG-69] Paginated requests MAY NOT be idempotent.

[RSG-70] A Web API MUST use query parameters to implement pagination.

[RSG-71] A Web API MUST NOT use HTTP headers to implement pagination.

[RSG-72] Query parameters `limit=<number of items to deliver>` and `offset=<number of items to skip>` SHOULD be used, where `limit` is the number of items to be returned (page size), and `skip` the number of items to be skipped (offset). If no page size limit is specified, a default SHOULD be defined - global or per collection; the default offset MUST be zero "0":

For example, the following is a valid URL:

```
https://wipo.int/api/v1/patents?limit=10&offset=20
```

[RSG-73] The `limit` and the `offset` parameter values SHOULD be included in the response.

Sorting

53. Retrieving data may require the data to be sorted by ascending or descending order. A multi-key sorting criterion may also be used. Sorting is determined through the use of the `sort` query string parameter. The value of this parameter is a comma-separated list of sort keys and sort directions that can optionally be appended to each sort key, separated by the colon ':' character. The supported sort directions are either '`asc`' for ascending or '`desc`' for descending. The client may specify a sort direction for each key. If a sort direction is not specified for a key, then a default direction is set by the server.

For example:

(a) Only sort keys specified:

```
sort=key1,key2
```

'`key1`' is the first key and '`key2`' is the second key and sort directions are defaulted by the server.

(b) Some sort directions specified:

`sort=key1:asc,key2`

where '`key1`' is the first key (ascending order) and '`key2`' is the second key (direction defaulted by the server, i.e., any sort key without a corresponding direction is defaulted).

(c) Each key with specified directions:

`sort=key1:asc,key2:desc`

where '`key1`' is the first key (ascending order) and '`key2`' is the second key (descending order).

54. In order to specify multi-attribute criteria sorting, the value of a query parameter may be a comma-separated list of sort keys and sort directions, with either '`asc`' for ascending or '`desc`' for descending which may be appended to each sort key, separated by the colon ':' character.

[RSG-74] A Web API SHOULD support sorting.

[RSG-75] In order to specify a multi-attribute sorting criterion, a query parameter MUST be used. The value of this parameter is a comma-separated list of sort keys and sort directions either '`asc`' for ascending or '`desc`' for descending MAY be appended to each sort key, separated by the colon ':' character. The default direction MUST be specified by the server in case that a sort direction is not specified for a key.

[RSG-76] A Web API SHOULD return the sorting criteria in the response.

Expansion

55. A service consumer may control the amount of data it receives by expanding a single field into larger objects. This is usually combined with Hypermedia support. Rather than simply asking for a linked entity ID to be included, a service caller can request the full representation of the entity be expanded within the results. Service calls may use expansions to get all the data they need in a single API request:

For example, if Hypermedia is supported, then the following HTTP request retrieves a patent and expands its applicant.

Example with JSON payloads based on ST.97

Retrieve a patent based on its number[5]:

```
GET /api/v1/patents/publications/100000000000001 HTTP/1.1

Host: wipo.int

Accept: application/json
```

The HTTP response is the following:

```
HTTP/1.1 200 OK

Content-Type: application/json
200 OK

{

      "patentPublication":{
```

[5] Patent/PatentNumber.xsd

```
            "languageCode": "en",
            ...
            "bibliographicData": {
                  "st96Version": "V5_0",
                  "applicationIdentification": {
                        "ipOfficeCode": "XX",
                        "applicationNumber": {
                              "applicationNumberText": "13797521"
                        },
                        "inventionSubjectMatterCategory": "Utility",
                        "filingDate": "2013-03-12"
                  },
                  patentGrantIdentification": {
                        "ipOfficeCode": "XX",
                        "patentNumber": "100000000000001"
                  },
                  ...
                  "partyBag": {
                        "applicantBag": {
                              "applicant": {
                                    "href":
"https://wipo.int/api/v1/link/to/applicants"
                              },
                        ...
                        }
                  }
            },
            ...
      }
}
```

Instead of the previous request, using the following HTTP request retrieves the full applicant information of the patent with number 100000000000001:

```
GET /api/v1/patents/publications?id=100000000000001&expand=applicant HTTP/1.1

Host: wipo.int

Accept: application/json
```

The HTTP response is the following:

```
HTTP/1.1 200 OK

Content-Type: application/json
200 OK

{

      "patentPublication":{

      "languageCode": "en",

      ...

      "bibliographicData": {

            "st96Version": "V5_0",

            "applicationIdentification": {

                  "ipOfficeCode": "XX",

                  "applicationNumber": {

                        "applicationNumberText": "13797521"

                  },

                  "inventionSubjectMatterCategory": "Utility",

                  "filingDate": "2013-03-12"

            },

            patentGrantIdentification": {

                  "ipOfficeCode": "XX",

                  "patentNumber": "100000000000001"

            },

            ...

            "partyBag": {

                  "applicantBag": {

                        "applicant": [

                                    {

                                          "sequenceNumber": "001",
```

```
                                    "publicationContact": [
                                          {
                                                "name": {
                                                      "personName": …,
                                    "applicantCategory": "Applicant",
                              },
                              {
                                    "sequenceNumber": "002",
                                    "publicationContact": [
                                          {
                                                "name": {
                                                      "personName": …
                                                }
                                          }
                                    ],
                                    "applicantCategory": "Applicant",
                              },
                              {
                                    "sequenceNumber": "003",
                                    "publicationContact": [
                                          {
                                                "name": {
                                                      "personName": …
                                                }
                                          }
                                    ],
                                    "applicantCategory": "Applicant",
                              },
                        ...
                        }
                  }
            },
...   }}
```

56. A Web API may support expanding the body of returned content.

[RSG-77] A Web API MAY support expanding the body of returned content. The query parameter `expand=<comma-separated list of attributes names>` SHOULD be used.

Projection

57. A Web API should support field projection, which controls how much of an entity's data is returned in response to an API request. The field projection can decrease response time and payload size. If only specific attributes from the retrieved data are required, a projection query parameter must be used instead of URL paths. The query parameter should be formed as follows: "`fields="<comma-separated list of attribute names>`. A projection query parameter is easier to implement and can retrieve multiple attributes. If a projection is supported, the XSD/JSON Schema should not apply in the response since the response will not be valid against the original XSD/JSON Schema.

For example, the following request message returns only the full name of the requested patent inventor:

In case of XML payloads based on ST.96

Get the patent inventor full name with the id equal to id12345:

```
GET

Host: wipo.int

Accept: application/xml
```

An example for the HTTP response message is shown:

```
HTTP/1.1 200 OK

Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>

<pat:Inventor xmlns="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:pat="http://www.wipo.int/standards/XMLSchema/ST96/Patent"
com:sequenceNumber="String" com:id="ID1"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Patent
PatentPublication_V5_0.xsd">

      <Contact>

            <Name>

                  <PersonName>

                        <PersonFullName>John Smith</PersonFullName>

                  </PersonName>

            </Name>

      </Contact>

</pat:Inventor>
```

In case of JSON payloads based on ST.97

Get the patent inventor full name with the id[6] equal to id12345:

```
GET

Host: wipo.int

Accept: application/json
```

An example for the HTTP response message is shown:

```
HTTP/1.1 200 OK

Content-Type: application/json

{

      "inventor": {

            "sequenceNumber": "001",

            "Contact": [

                  {

                        "name": {

                              "personName": [

                                    {

                                          "personFullName": "John Smith"

                                    }

                              ]

                        }

                  }

            ]

      }}
```

[RSG-78] A query parameter SHOULD be used instead of URL paths in case that a Web API supports projection following the format: "`fields="<comma-separated list of attribute names>`.

Number of items

58. In some use cases, the consumer of the API may be interested in the number of items in a collection. This is very common when combined with pagination in order to know the total number of items in the collection.

For example, the following HTTP request retrieves maximum 3 patent publications, skipping the first 4 results and should also contain in the response the total number of the available results:

[6] Common/id.xsd

Example with XML payloads based on ST.96

```
GET /api/v1/patents/publications?count=true&limit=3&offset=4 HTTP/1.1

Host: wipo.int

Accept: application/xml
```

The following example HTTP response is returned:

```
HTTP/1.1 200 OK

Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>

<pat:PatentPublication xmlns="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xmlns:pat="http://www.wipo.int/standards/XMLSchema/ST96/Patent"
com:languageCode="de" com:st96Version="V5_0"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Patent
PatentPublication_V5_0.xsd">

   ...

</pat:PatentPublication>

<pat:PatentPublication>

   ...

</pat:PatentPublication>

   ...

<pat:PatentPublication>

   ...

</pat:PatentPublication>

<count>100</count>
```

Example with JSON payloads based on ST.97

```
GET /api/v1/patents/publications?count=true&limit=3&offset=4 HTTP/1.1

Host: wipo.int

Accept: application/json
```

The following example HTTP response is returned:

```
HTTP/1.1 200 OK

Content-Type: application/json
```

```
{
      "patentPublication": [
            {
            ...
            },
            {
            ...
            },
            {
            ...
            }
      ],
      "count": 100
}
```

59. As one alternative, a Web API may support returning the number of items in a collection inline, i.e., as the part of the response that contains the collection itself. Alternatively, it may form part of a metadata envelope, outside the main body of the response.

[RSG-79] A Web API MUST support returning the number of items in a collection.

[RSG-80] A query parameter SHOULD be used to support returning the number of items in a collection.

[RSG-81] The query parameter `count` SHOULD be used to return the number of items in a collection.

[RSG-82] A Web API MAY support returning the number of items in a collection inline, i.e., as the part of the response that contains the collection itself.

[RSG-83] The query parameter `count=true` SHOULD be used. If not specified, `count` should be set by default to `false`.

[RSG-84] If a Web API supports pagination, it SHOULD support returning inline in the response the number of the collection (i.e., the total number of items of the collection).

Complex search expressions

60. For retrieving data with only a few search criteria, the query parameters are adequate. If there is a use case where we should search for data using complex search expressions (with multiple criteria, Boolean expressions and search operators) then the API has to be designed using a more complex query language. A query language has to be supported by a search grammar.

61. The Contextual Query Language (CQL) is a formal language for representing queries to information retrieval systems such as search engines, bibliographic catalogs and museum collection information. Based on the semantics of Z39.50[7], its design objective is that queries must be readable and writable, and that the language is intuitive and maintains the expression of more complex query languages. This is just one option recommended for use, as it is used broadly by industry.

[7] Please refer the References chapter

[RSG-85] When a Web API supports complex search expressions, a query language SHOULD be specified, such as CQL.

[RSG-86] A Service Contract MUST specify the grammar supported (such as fields, functions, keywords, and operators).

[RSG-87] The query parameter "`q`" MUST be used.

Error handling

62. Error responses should always use the appropriate HTTP status code selected from the standard list of HTTP status codes (RFC 7807), reproduced in Annex V. When the requestor is expecting JSON, return error details in a common data structure. Unless the project requires otherwise, there is no need to define application-specific error codes. Stack trace and other debugging-related information should not be present in the error response body in production environments.

Error payload

63. Error handling is carried out on two levels: on the protocol level (HTTP) and on the application level (payload returned). On the protocol level, a Web API returns an appropriate HTTP status code and on the application level, a Web API returns a payload reporting the error in adequate granularity (mandatory and optional attributes).

64. With regard to the mandatory and optional attributes for the application level error handling,

(a) The following `code` and `message` attributes are mandatory and while the `message` may change in the future, the `code` will not change; it is fixed and will always refer to this particular problem:

- `code` (integer): Technical code of the error situation to be used for support purposes; and
- `message` (string): User-facing (localizable) message describing the error request as requested by the HTTP header `Accept-Language` (see RSG-114).

(b) The following attributes are conditionally mandatory:

- `details`: If error processing requires nesting of error responses, it must use the details field for this purpose. The details field must contain an array of JSON objects that shows code and message properties with the same semantics as described above.

(c) The following attributes are optional:

- `target`: The error structure may contain a target attribute that describes a data element (for example, a resource path);
- `status` : Duplicate of the HTTP status code to propagate it along the call chain or to write it in the support log without the need to explicitly add the HTTP status code every time;
- `moreInfo`: Array of links containing more information about the error situation, for example, giving hints to the end user; and
- `internalMessage`: A technical message, for example, for logging purposes.

65. Error handling should follow HTTP standards (IETF RFC 9110). A minimum error payload is recommended:

For example, the following HTTP response is returned when trademark was not found for the provided international registration number:

Example with XML payload based on ST.96

```
GET /api/v1/trademarks?irn=000000000000001John%20Smith&expiryDate=2018-12-31.
HTTP/1.1

Host: wipo.int
```

```
Accept: application/xml
```

The following example HTTP response is returned:

```
HTTP/1.1 404

Content-Type: application/xml

 <?xml version="1.0" encoding="UTF-8"?>

<com:TransactionError xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:com="http://www.wipo.int/standards/XMLSchema/ST96/Common"
xsi:schemaLocation="http://www.wipo.int/standards/XMLSchema/ST96/Common
TransactionError.xsd">

      <com:TransactionErrorCode>TRADEMARK_NOT_FOUND</com:TransactionErrorCode>

      <com:TransactionErrorText>The trademark with the provided International
Registration Number was not found</com:TransactionErrorText>

</com:TransactionError>
```

Example with JSON payload based on ST.97

```
HTTP/1.1 404

Content-Type: application/json

{

      "transactionError": [

            {

                  "transactionErrorCode": "TRADEMARK_NOT_FOUND"

            },

            {

                  "transactionErrorText": "The trademark with the provided
International Registration Number was not found"

            },

      ]

}
```

[RSG-88] On the protocol level, a Web API MUST return an appropriate HTTP status code selected from the list of standard HTTP Status Codes.

[RSJ-89] On the application level, a Web API MUST return a payload reporting the error in adequate granularity. The `code` and `message` attributes are mandatory, the `details` attribute is conditionally mandatory and `target`, `status`, `moreInfo`, and `internalMessage` attributes are optional.

[RSG-90] Errors MUST NOT expose security-critical data or internal technical details, such as call stacks in the error messages.

[RSG-91] The HTTP `Header: Reason-Phrase` (described in IETF RFC 9112) MUST NOT be used to carry error messages.

Correlation ID

66. Typically consuming a service cascades to triggering multiple other services. There should be a mechanism to correlate all the service activations in the same execution context. For example, including the correlation ID in the log messages, as this uniquely identifies the logged error. A header name should be used e.g., Request-ID or Correlation-ID are commonly used, as taking this into account in design phase of an API, will foster forward compatibility between different APIs and newer implementations.

[RSG-92] Every logged error SHOULD have a unique Correlation ID. A custom HTTP header SHOULD be used and SHOULD be named Correlation-ID.

Service contract

67. REST is not a protocol or an architecture, but an architectural style with architectural properties and architectural constraints. There are no official standards for REST API contracts. This Standard refers to API documentation as a REST Service Contract. The Service Contract is based on the following three fundamental elements:

(a) Resource identifier syntax: How can we express where the data is being transferred to or from?

(b) Methods: What are the protocol mechanisms used to transfer the data?

(c) Media types: What type of data is being transferred? Individual REST services use these elements in different combinations to expose their capabilities. Defining a master set of these elements for use by a collection (or inventory) of services makes this type of service contract "uniform".

[RSG-93] A Service Contract format MUST include the following:

- API version;
- Information about the semantics of API elements;
- Resources;
- Resource attributes;
- Query Parameters;
- Methods;
- Media types;
- Search grammar (if one is supported);
- HTTP Status Codes;
- HTTP Methods;
- Restrictions and distinctive features; and
- Security (e.g., private schemas).

[RSG-94] A Service Contract format SHOULD include requests and responses in XML schema or JSON Schema and examples of the API usage in the supported formats, i.e., XML or JSON.

[RSG-95] A REST API MUST provide API documentation as a Service Contract.

[RSG-96] A Web API implementation deviating from this Standard MUST be explicitly documented in the Service Contract. If a deviating rule is not specified in the Service Contract, it MUST be assumed that this Standard is followed.

[RSG-97] A Service Contract SHOULD allow API client skeleton code generation.

[RSG-98] A Service Contract SHOULD allow server skeleton code generation.

68. Web API documentation can be written for example in RESTful API Modeling Language (RAML), Open API Specification (OAS) and WSDL. As only RAML fully supports both XML and JSON request/response validation (by using XSD schemas and JSON schemas), this Standard recommends RAML[8].

[RSG-99] A Web API documentation SHOULD be written in RAML or OAS. Custom documentation formats SHOULD NOT be used.

Time-out

69. According to the service-oriented design principles, the server usage should be limited.

[RSG-100] A Web API consumer SHOULD be able to specify a server timeout for each request; a custom HTTP header SHOULD be used. A maximum server timeout SHOULD be also used to protect server resources from over-use.

State management

70. If development proceeds following the REST principles, state management must be dealt with on the client side, rather than on the server, since REST APIs are stateless. For example, if multiple servers implement a session, replication should be discouraged.

Response versioning

71. Retrieving multiple times the same data set may result in bandwidth consumption if the data set has not been modified between the requests. Data should be conditionally retrieved only if it has not been modified. This can be done with Content-based Resource Validation or Time-based Resource Validation. If using response versioning, a service consumer may implement optimistic locking.

[RSG-101] A Web API SHOULD support conditionally retrieving data, to ensure only data which is modified will be retrieved. Content-based Resource Validation SHOULD be used because it is more accurate.

[RSG-102] In order to implement Content-based Resource Validation the `ETag` HTTP header SHOULD be used in the response to encode the data state. Afterward, this value SHOULD be used in subsequent requests in the conditional HTTP headers (such as If-Match or If-None-Match). If the data has not been modified since the request returned the `ETag`, the server SHOULD return the status code "`304 Not Modified`" (if not modified). This mechanism is specified in IETF RFC 9110 .

[RSG-103] In order to implement Time-based Resource Validation the `Last-Modified` HTTP header SHOULD be used. This mechanism is specified in IETF RFC 9110.

[RSG-104] Using response versioning, a service consumer MAY implement Optimistic Locking.

Caching

72. A Web API implementation should support cache handling in order to save bandwidth, in compliance with the IETF RFC 9111.

[RSG-105] A Web API MUST support caching of `GET` results; a Web API MAY support caching of results from other HTTP Methods.

[RSG-106] The HTTP response headers `Cache-Control` and `Expires` SHOULD be used. The latter MAY be used to support legacy clients.

[8] OAS is a specification. It also supports Markdown, but RAML does not. On the other hand, although both OAS and RAML support JSON Schema validation for the requests and responses, OAS does not support XSDs. Therefore, in the future, when OAS is feature-complete it may be recommended.

Managed file transfer

73. Transferring (i.e., downloading or uploading) large files has a high probability of causing a network interruption or some other transmission failure. It also consumes a large amount of memory for both the service provider and service consumer. Therefore, it is recommended to transfer large files in multiple chunks with multiple requests. This option also provides an indication of the total download or upload progress. The partial transfer of large files should resume support. The service provider should advertise if it supports the partial transfer of large files.[9]

74. There are two approaches for implementing this type of transfer: the first is to use a `Transfer-Encoding: chunked` header and the second using the `Content-Length` header. These headers should not be used together. `Content-Length` indicates the full size of the file transferred, and therefore the receiver will know the length of the body and will be able to estimate the download completion time. The `Transfer-Encoding: chunked` header is useful for streaming infinitely bounded data, such as audio or video, but not files. It is recommended to use the `Content-Length` header for downloading as the server utilization is low in comparison to `Transfer-Encoding: chunked`. For uploading, the `Transfer-Encoding: chunked` header is recommended.

A Web API should advertise if it supports partial file downloads by responding to `HEAD` requests and replying with the HTTP response headers: `Accept-Ranges` and `Content-Length`. The former should indicate the unit that can be used to define a range and should never be defined as' none'. The latter indicates the full size of the file to download.

[RSG-107] A Web API SHOULD advertise if it supports partial file downloads by responding to `HEAD` requests and replying with the HTTP response headers `Accept-Ranges` and `Content-Length`.

75. A Web API that supports downloading large files should support partial requests according to IETF RFC 7232, i.e.:

- The service consumer asking for a range should use the HTTP header `Range`;
- The service provider response should contain the HTTP headers `Content-Range` and `Content-Length`; and
- The service provider response should have the HTTP status "`206 Partial Content`" in case of a successful range request. In case of a range request that is out of bounds (range values overlap the extent of the resource), the server responds with a "`416 Requested Range Not Satisfiable`" status. In case the range requested is not supported, the "`200 OK`" status is sent back from a server.

[RSG-108] A Web API SHOULD support partial file downloads. Multi-part ranges SHOULD be supported.

76. Multipart ranges may also be requested if the HTTP header `Content-Type: multipart/byteranges; boundary=XXXXX` is used. A range request may be conditional if it is combined with `ETag` or `If-Range` HTTP Headers.

77. There is not any IETF RFC for large files upload. Therefore, in this Standard we do not provide any implementation recommendation for large file uploads.

[RSG-109] A Web API SHOULD advertise if it supports partial file uploads.

[RSG-110] A Web API SHOULD support partial file uploaded. Multi-part ranges SHOULD be supported.

78. The IETF RFC 9110 does not impose any specific size limit for requests. The API Service Contract should specify the maximum limit for the requests. Moreover, on runtime the service provider should indicate to the service consumer if the allowed maximum limit has been exceeded.

[RSG-111] The service provider SHOULD return with HTTP response headers the HTTP header "`413 Request Entity Too Large`" in case the request has exceeded the maximum allowed limit. A custom HTTP header MAY be used to indicate the maximum size of the request.

[9] The service provider may return the location of the file and then the service consumer can call a directory service to download the file. At the end, a partial file download is required. This paragraph does not take into account non-REST protocols such as FTP or sFTP or rsync.

Preference handling

79. A service provider may allow a service consumer to configure values and influence how the former processes the requests of the latter. A standard means for implementing preference handling is outlined in IETF RFC 8144.

[RSG-112] If a Web API supports preference handling, it SHOULD be implemented according to IETF RFC 8144, i.e., the request HTTP header `Prefer` SHOULD be used and the response HTTP header `Preference-Applied` SHOULD be returned (echoing the original request).

[RSG-113] If a Web API supports preference handling, the nomenclature of preferences that MAY be set by using the `Prefer` header MUST be recorded in the Service Contract.

Translation

80. A service consumer may request responses in a specific language if the service provider supports it. A standard specification for handling of a set of natural languages is outlined in IETF RFC 9110.

[RSG-114] If a Web API supports localized data, the request HTTP header `Accept-Language` MUST be supported to indicate the set of natural languages that are preferred in the response as specified in IETF RFC 9110.

Long-running operations

81. There are cases, where a Web API may involve long running operations. For instance, the generation of a PDF by the service provider may take some minutes. This paragraph recommends a typical message exchange pattern to implement such cases, for example:

```
// (a)

GET https://wipo.int/api/v1/patents

Accept: application/pdf

…

// (b)

HTTP/1.1 202 Accepted

Location: https://wipo.int/api/v1/queues/12345

…

// (c1)

GET https://wipo.int/api/v1/queues/12345

…

HTTP/1.1 200 OK

…

// (c2)

GET https://wipo.int/api/v1/queues/12345

HTTP/1.1 303 See Other

Location: https://wipo.int/api/v1/path/to/pdf

…
// (c3)
GET https://wipo.int/api/v1/path/to/pdf
…
```

82. If an API supports long-running operations, then they should be performed asynchronously to ensure the user is not made to wait for a response. The rule below sets out a recommended approach for implementation.

[RSG-115] If the API supports long-running operations, they SHOULD be asynchronous. The following approach SHOULD be followed:

(a) The service consumer activates the service operation;

(b) The service operation returns the status code "`202 Accepted`" according to IETF RFC 9110 (section 15.3.3) i.e., the request has been accepted for processing but the processing has not been completed. The location of the queued task that was created is also returned with the HTTP header Location; and

(c) The service consumer calls the returned Location to learn if the resource is available. If the resource is not available, the response SHOULD have the status code "`200 OK`", contain the task status (for example pending) and MAY contain other information (for example, a progress indicator, and/or a link to cancel or delete the task using the DELETE HTTP method). If the resource is available, the response SHOULD have the status code "`303 See Other`" and the HTTP header Location SHOULD contain the URL to retrieve the task results.

Security model

General rules

83. Within the scope of this standard, API security is concerned with pivotal security attributes that will ensure that information accessible by an API and APIs themselves are secure throughout their lifecycle. These attributes are confidentiality, integrity, availability, trust, non-repudiation, compartmentalization, authentication, authorization and auditing.

[RSG-116] Confidentiality: APIs and API Information MUST be identified, classified, and protected against unauthorized access, disclosure and eavesdropping at all times. The least privilege, zero trust, need to know and need to share[10] principles MUST be followed.

[RSG-117] Integrity-Assurance: APIs and API Information MUST be protected against unauthorized modification, duplication, corruption and destruction. Information MUST be modified through approved transactions and interfaces. Systems MUST be updated using approved configuration management, change management and patch management processes.

[RSG-118] Availability: APIs and API Information MUST be available to authorized users at the right time as defined in the Service Level Agreements (SLAs), access-control policies and defined business processes.

[RSG-119] Non-repudiation: Every transaction processed or action performed by APIs MUST enforce non-repudiation through the implementation of proper auditing, authorization, authentication, and the implementation of secure paths and non-repudiation services and mechanisms.

[RSG-120] Authentication, Authorization, Auditing: Users, systems, APIs or devices involved in critical transactions or actions MUST be authenticated, authorized using role-based or attribute based access-control services and maintain segregation of duty. In addition, all actions MUST be logged, and the authentication's strength must increase with the associated information risk.

Guidelines for secure and threat-resistant API management

84. APIs should be designed, built, tested, and implemented with security requirements and risks in mind. The appropriate countermeasures and controls should be built directly into the design and not as an after-thought. It is recommended to use best practices and standards, such as OWASP.

[RSG-121] While developing APIs, threats, malicious use cases, secure coding techniques, transport layer security and security testing MUST be carefully considered, especially:

[10] https://www.owasp.org/index.php/Security_by_Design_Principles

- `PUTs` and `POSTs`: i.e., which change to internal data could potentially be used to attack or misinform;
- `DELETEs`: i.e., could be used to remove the contents of an internal resource repository;
- Whitelist allowable methods to ensure that allowable HTTP Methods are properly restricted while others would return a proper response code; and
- Well known attacks should be considered during the threat-modeling phase of the design process to ensure that the threat risk does not increase. The threats and mitigation defined within OWASP Top Ten Cheat Sheet[11] MUST be taken into consideration.

[RSG-122] While developing APIs, the standards and best practices listed below SHOULD be followed:

- Secure coding best practices: OWASP Secure Coding Principles;
- Rest API security: REST Security Cheat Sheet;
- Escape inputs and cross site scripting protection: OWASP XSS Cheat Sheet;
- SQL Injection prevention: OWASP SQL Injection Cheat Sheet, OWASP Parameterization Cheat Sheet; and
- Transport layer security: OWASP Transport Layer Protection Cheat Sheet.

[RSG-123] Security testing and vulnerability assessment MUST be carried out to ensure that APIs are secure and threat-resistant. This requirement MAY be achieved by leveraging Static and Dynamic Application Security Testing (SAST/DAST), automated vulnerability management tools and penetration testing.

Encryption, integrity and non-repudiation

85. Protected services must be secured to protect authentication credentials in transit: for example, passwords, API keys or JSON Web Tokens. Integrity of the transmitted data and non-repudiation of action taken should also be guaranteed. Secure cryptographic mechanisms can ensure confidentiality, encryption, integrity assurance and non-repudiation. Perfect forward secrecy is one means of ensuring that session keys cannot be compromised.

[RSG-124] Protected services MUST only provide HTTPS endpoints using TLS 1.2, or higher, with a cipher suite that includes ECDHE for key exchange.

[RSG-125] When considering authentication protocols, perfect forward secrecy SHOULD be used to provide transport security. The use of insecure cryptographic algorithms and backwards compatibility to SSL 3 and TLS 1.0/1.1 SHOULD NOT be allowed.

[RSG-126] For maximum security and trust, a site-to-site IPSEC VPN SHOULD be established to further protect the information transmitted over insecure networks.

[RSG-127] The consuming application SHOULD validate the TLS certificate chain when making requests to protected resources, including checking the certificate revocation list.

[RSG-128] Protected services SHOULD only use valid certificates issued by a trusted certificate authority (CA).

[RSG-129] Tokens SHOULD be signed using secure signing algorithms that are compliant with the digital signature standard (DSS) FIPS –186-4. The RSA digital signature algorithm or the ECDSA algorithm SHOULD be considered.

[11] https://owasp.org/www-project-top-ten/2017/

Authentication and authorization

86. Authorization is the act of performing access control on a resource. Authorization does not just cover the enforcement of access controls, but also the definition of those controls. This includes the access rules and policies, which should define the required level of access agreeable to both provider and consuming application. The foundation of access control is a provider granting or denying a consuming application and/or consumer access to a resource to a certain level of granularity. Coarse-grained access should be considered at the API or the API gateway request point while fine-grained control should be considered at the backend service, if possible. Role Based Access Control (RBAC) or the Attribute Based Access Control (ABAC) model can be considered.

87. If a service is protected, then Open ID Connect should be favored over OAuth 2.0 because it fills many of the gaps of the latter and provides a standardized way to gain a resource owner's profile data, JSON Web Token (JWT) standardized token format and cryptography. Other security schemes should not be used such as HTTP Basic Authorization which requires that the client must keep a password somewhere in clear text to send along with each request. Also the verification of this password would be slower because it will have to access the credential store. OAuth 2.0 does not specify the security token. Therefore, the JWT token should be used in comparison for example to SAML 2.0, which is more verbose.

[RSG-130] Anonymous authentication MUST only be used when the customers and the application they are using accesses information or feature with a low sensitivity level which should not require authentication, such as, public information.

[RSG-131] Username and password or password hash authentication MUST NOT be allowed.

[RSG-132] If a service is protected, Open ID Connect SHOULD be used.

[RSG-133] Where a JSON Web Token (JWT) is used, a JWT secret SHOULD possess high entropy to increase the work factor of a brute force attack; token TTL and RTTL SHOULD be as short as possible; and sensitive information SHOULD NOT be stored in the JWT payload.

88. A common security design choice is to centralize user authentication. It should be stored in an Identity Provider (IdP) or locally at REST endpoints.

89. Services should be careful to prevent leaking of credentials. Passwords, security tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable. For example, the following is incorrect (API Key in URL): `https://wipo.int/api/patents?apiKey=a53f435643de32.`

[RSG-134] In `POST/PUT` requests, sensitive data SHOULD be transferred in the request body or by request headers.

[RSG-135] In `GET` requests, sensitive data SHOULD be transferred in an HTTP Header.

[RSG-136] In order to minimize latency and reduce coupling between protected services, the access control decision SHOULD be taken locally by REST endpoints.

90. API Keys Authentication: API keys should be used wherever system-to-system authentication is required and they should be automatically and randomly generated. The inherent risk of this authentication mode is that anyone with a copy of the API key can use it as though they were the legitimate consuming application. Hence, all communications should comply with RSG-124, to protect the key in transit.

The onus is on the application developer to properly protect their copy of the API key. If the API key is embedded into the consuming application, it can be decompiled and extracted. If stored in plain text files, they can be stolen and re-used for malicious purposes. An API Key must therefore be protected by a credential store or a secret management mechanism. API Keys may be used to control services usage even for public services.

[RSG-137] API Keys SHOULD be used for protected and public services to prevent overwhelming their service provider with multiple requests (denial-of-service attacks). For protected services API Keys MAY be used for monetization (purchased plans), usage policy enforcement (QoS) and monitoring.

[RSG-138] API Keys MAY be combined with the HTTP request header user-agent to discern between a human user and a software agent as specified in IETF RFC 9110.

[RSG-139] The service provider SHOULD return along with HTTP response headers the current usage status. The following response data MAY be returned:

- Rate limit: Rate limit (per minute) as set in the system;
- Rate limit remaining: Remaining amount of requests allowed during the current time slot (-1 indicates that the limit has been exceeded); and
- Rate limit reset: Time (in seconds) remaining until the request counter will be reset.

[RSG-140] The service provider SHOULD return the status code "`429 Too Many Requests`" if requests are coming in too quickly.

[RSG-141] API Keys MUST be revoked if the client violates the usage agreement, as specified by the IPO.

[RSG-142] API Keys SHOULD be transferred using custom HTTP headers. They SHOULD NOT be transferred using query parameters.

[RSG-143] API Keys SHOULD be randomly generated.

91. While there is an overhead with the use of public key cryptography and certificates, certificate-based mutual authentication should be used when a Web API requires stronger authentication than offered by API keys to provide additional security. Secure and trusted certificates must be issued by a mutually trusted certificate authority (CA) through a trust establishment process or cross-certification. To mitigate identity security risks peculiar to sensitive systems and privileged actions, strong authentication can be leveraged. Certificates shared between the client and the server should be used, e.g., X.509.

[RSG-144] Secure and trusted certificates MUST be issued by a mutually trusted certificate authority (CA) through a trust establishment process or cross-certification.

[RSG-145] Certificates shared between the client and the server SHOULD be used to mitigate identity security risks particular to sensitive systems and privileged actions, e.g., X.509.

[RSG-146] For highly privileged services, two-way mutual authentication between the client and the server SHOULD use certificates to provide additional protection.

[RSG-147] Multi-factor authentication SHOULD be implemented to mitigate identity risks for application with a high-risk profile, a system processing very sensitive information or a privileged action.

Availability and threat protection

92. Availability in this context covers threat protection to minimize API downtime, looking at how threats against exposed APIs can be mitigated using basic design principles. Availability also covers scaling to meet demand and ensuring the hosting environments are stable etc. These levels of availability are addressed across the hardware and software stacks that support the delivery of APIs. Availability is normally addressed under business continuity and disaster recovery standards that recommend a risk assessment approach to define the availability requirements.

Cross-domain requests

93. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy. Under the same-origin policy, a web browser permits scripts contained in a first web page to access data in a second web page, only if both web pages have the same origin (i.e., combination of URI scheme, host name, and port number).

94. The Cross-Origin Resource Sharing (CORS) is a W3C standard to flexibly specify which Cross-Domain Requests are permitted. By delivering appropriate CORS HTTP headers, your REST API signals to the browser which domains or origins are allowed to make JavaScript calls to the REST service.

95. The JSON with padding (JSONP) is a method for sending JSON data without worrying about cross-domain request issues. It introduces callback functions for the loading of JSON data from different domains. The idea behind it is based on the fact that the HTML `<script>` tag is not affected by the same origin policy. Anything imported through this tag is executed immediately in the global context. Instead of passing in a JavaScript file, one can pass in a URL to a service that returns JavaScript code.

96. The following approaches are usually followed to bypass this restriction:

- JSONP is a workaround for cross-domain requests. It does not offer any error-detection mechanism, i.e., if there was an issue and the service failed or responded with an HTTP error, there is no way to determine what

the issue was on the client side. The result will be that the AJAX application will just 'hang'. Moreover, the site that uses JSONP will unconditionally trust the JSON provided from a different domain;

– Iframe is an alternative workaround for cross-domain requests. Using the JavaScript `window.postMessage (message, targetOrigin)` method on the iframe object, it is possible to pass a request a site of a different domain. Iframe approach has good compatibility even in old browsers. Moreover, it only supports GET. The source of the Iframes page should always be checked due to security issues; and

– CORS is a standardized approach to perform a call to an external domain. It can use `XMLHttpRequest` to send and receive data and has better error handling mechanism than JSONP. It supports many types of authorization in comparison to JSONP, which only supports cookies. It also supports HTTP Methods in comparison to JSONP, which only supports `GET`. On the other hand, it is not always possible to implement CORS because the browsers have to support it and because the API consumers have to be enlisted in the CORS whitelist.

[RSG-148] If the REST API is public, the HTTP header `Access-Control-Allow-Origin` MUST be set to '*'.

[RSG-149] If the REST API is protected, CORS SHOULD be used, if possible. Else, JSONP MAY be used as fallback but only for GET requests, for example, when the user is accessing using an old browser. Iframe SHOULD NOT be used.

API maturity model

97. It is common to classify a REST API using a maturity model. While various models are available, this Standard refers to the Richardson Maturity Model (RMM). RMM defines three levels, and this Standard recommends Level 2 for REST API because Level 3 is complex to implement and requires significant conceptual and development-related investment from service providers and consumers. At the same time, it does not immediately benefit service consumers.

98. If a Web API implements Level 3 of RMM, a hypermedia format must be put in place. Hypertext Application Language (HAL) is simple and is compatible with JSON and XML responses. However, it is only a draft recommendation, along with other hypermedia formats, such as JSON-LD. JSON-Schema should be used because as although there is currently no specification for Level 3 of RMM, this is considered the most mature. The following hypermedia formats should not be considered: IETF RFC 8288 and `Collection+JSON`.

99. It is recommended that instances described by a schema provide a link to a downloadable JSON Schema using the link relation "`describedby`", as defined by Linked Data Protocol 1.0, section 8.1 [W3C.REC-ldp-20150226].

In HTTP, such links can be attached to any response using the `Link` header [RFC8288]. An example of such a header would be:

```
Link: <http://example.com/my-hyper-schema#>; rel="describedby"
```

[RSJ-150] If using instances described a schema, the `Link` header SHOULD be used to provide a link to a downloadable JSON schema according to RFC 8288.

[RSJ-151] A Web API SHOULD implement at least Level 2 (Transport Native Properties) of RMM. Level 3 (Hypermedia) MAY be implemented to make the API completely discoverable.

100. A custom hypermedia format may be designed. In which case, a set of attributes is recommended. For example:

```
{
  "link": {
     "href": "/patents",
     "rel": "self"
  },
   ...
}
```

[RSJ-152] For designing a custom hypermedia format the following set of attributes SHOULD be used enclosed into an attribute link:

- `href` – the target URI;
- `rel` – the meaning of the target URI;
- `self` – the URI references the resource itself;
- `next` – the URI references the next page (if used during pagination);
- `previous` – the URI references the previous page (if used during pagination); and
- arbitrary name `v` denotes the custom meaning of a relation.

SOAP WEB API

101. This standard recommends the REST architectural style as the preferred approach to API design. RESTful architectures are generally simpler to design, extend, integrate than SOAP. Coverage of SOAP is included here for completeness; examples and use cases are not provided.

102. A SOAP Web API is a software application identified by URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts. It also supports direct interactions with other software applications using XML-based messages, via internet protocols such as SOAP and HTTP.

103. A SOAP-based contract is described in a Web Service Definition Language (WSDL), a W3C standard document. Throughout this document “Web Service Contract WSDL document” will be referred as just “WSDL”.

104. When creating web services, there are two development styles: Contract Last and Contract First. When using a contract-last approach, you start with the code, and let the web service contract be generated from that. When using contract-first, you start with the WSDL contract, and use code to implement said contract.

General rules

105. The Web Service Interoperability (WS-I) Profile is one of the most important standards in regard to SOAP-based APIs, and it provides a minimum foundation for writing Web Services that can work together. WS-I provides a guideline on how services are “exposed” to each other and how they transfer information (referred to as ‘messaging’). It is a profile for implementing specific versions of some of the most important Web Service standards such as WSDL, SOAP, XML, etc. Adhering to certain profiles implicitly indicates adhering to specific versions of these Web Services standards. WS-I Basic Profile v1.1 provides guidance for using XML 1.0, HTTP 1.1, UDDI, SOAP 1.1, WSDL 1.1, and UDDI 2.0. WS-I Basic Profile 2.0 provides guidance for using SOAP 1.2, WSDL 1.1, UDDI 2.0, WS-Addressing, and MTOM. SOAP 1.2 provides a clear processing model and leads to better interoperability. WSDL 2.0 was designed to solve the interoperability issues found in WSDL 1.1 by using improved SOAP 1.2 bindings.

[WS-01] All WSDLs MUST conform to WS-I Basic Profile 2.0. WSDL 1.2 MAY be used.

106. A WSDL SOAP binding can be either a Remote Procedure Call (RPC) style binding or a document-style binding. A SOAP binding can also have an encoded use or a literal use. This gives you five style/use models: RPC/encoded, RPC/literal, document/encoded, document/literal, document/literal wrapped.

[WS-02] Services MUST follow document-style binding and literal use models (either document/literal or document/literal wrapped). When there are graphs, the RPC/encoded style MUST be used.

[WS-03] When there are exceptional use cases, such as when there are overloaded operations in the WSDL, all the other styles SHOULD be used.

107. The concrete WSDL should be separated from the abstract WSDL in order to provide a more modular and flexible interface. The abstract WSDL defines data types, messages, operation, and the port type. The concrete WSDL defines the binding, port and service.

[WS-04] The WSDL SHOULD be separated into an abstract and a concrete part.

[WS-05] All data types SHOULD be defined in an XSD file and imported in the abstract WSDL.

[WS-06] The concrete WSDL MUST define only one service with one port.

Schemas

108. Schemas used in the WSDL must be compliant with WIPO Standard ST.96 Standard. For re-use purposes and modularity, a schema must be a separate document that is either included or imported into the WSDL, instead of defining directly it in the WSDL. This will permit changes in XML structure without changing the WSDL.

[WS-07] The schema defined in the `wsdl:types` element MUST be imported from a self-standing schema file, to allow modularity and re-use.

[WS-08] Import of an external schema MUST be implemented using an `xsd:import` technique, not an `xsd:include`.

[WS-09] Element `xsd:any` MUST NOT be used to specify a root element in the message body.

[WS-10] The target namespace for the WSDL (attribute `targetNamespace` on `wsdl:definitions`) MUST be different from the target namespace of the schema (attribute `targetNamespace` on `xsd:schema`).

[WS-11] The requests and responses (naming convention, message format, data structure, and data dictionary) SHOULD follow WIPO Standard ST.96.

Naming and versioning

109. Appropriate naming conventions should also be applied when naming Services and WSDL elements. Naming conventions should follow those implemented in WIPO Standard ST.96.

[WS-12] Services MUST be named in UpperCamelCase and have a 'Service' suffix, for example `https://wipo.int/PatentsService`.

[WS-13] WSDL elements message, part, portType, operation, input, output, and binding SHOULD be named in UpperCamelCase.

[WS-14] Request message names SHOULD have a ‘Request’ suffix.

[WS-15] Response message names SHOULD have a ‘Response’ suffix.

[WS-16] Operation names SHOULD follow the format of `<Verb><Object>{<Qualifier>}`, where `<Verb>` indicates the operation (preferably Get, Create, Update, or Delete where applicable) on the `<Object>` of the operation, optionally finally followed by a `<Qualifier>` of the `<Object>`.

110. All operation names will have at least two parts. An optional third part may be included to further clarify and/or specify the business purpose of the operation. The three parts are: `<Verb> <Object> <Qualifier - Optional>`. Each part will be described in detail below.

Verb: Each operation name will start with a verb. The verb examples in common usage are described below:

Verb	Description	Example
Get	Get a single object	`GetBibData`
Create	Get a new object	`CreateBibData`
Update	Update an object	`UpdateBibData`
Delete	Delete an object	`DeleteCustomer`

Object: A noun following a verb will be a succinct and unambiguous description of the business function the operation is providing. The goal is to provide consumers with a better understanding of what the operation does with no ambiguity. Given that the definition of some entities are not common across the various cost centers, the object may be a composite field with the first node being the cost center and the second node the entity, for example, `PatentCustomer`.

Qualifier: The purpose of the object qualifier (optional) attribute is, to further clarify the business domain or subject area, for example, `GetCustomerList`. `Get` denotes the operation to be acted upon the Customer and `List` further describes the fact that the intention is to get a list of Customers not just one customer as in `GetCustomer`.

111. According to the service-oriented design principles, service providers and consumers should evolve independently. The service consumer should not be affected from minor (backward compatible) changes by the service provider. Therefore, service versioning should use only major version numbers. For internal APIs (for example, for development and testing) minor versions may also be used such as Semantic Versioning.

[WS-17] The name of the WSDL file SHOULD conform the following pattern: `<service name>_V<major version number>`

[WS-18] The namespace of the WSDL file SHOULD contain the service version; e.g., `https://wipo.int/PatentsService/V1"`

112. The description of service and its operations is provided as WSDL documentation.

[WS-19] Element `wsdl:documentation` SHOULD be used in WSDL with description of service (as the first child of `wsdl:definitions` in the WSDL) and its operations.

Web service contract design

113. A Web Service Contract should include a technical interface comprised of a Web Service Definition Language (WSDL), XML Schema definitions, WS-Policy descriptions as well as a non-technical interface comprised of one or more service description documents.

114. The WSDL, part of the “Service Contract,” must be designed prior to any code development. No WSDL should ever be auto-generated from the code. The motto is “Contract First” and NOT “Code First”. All Web Service Contracts must conform to Web Service Interoperability Basic Profile (WS-I BP). Any project that auto-generates from code will be liable to amendments to ensure conformance to these standards.

Attaching policies to WSDL definitions

115. Web Service Contracts can be extended with security policies that express additional constraints, requirements, and qualities that typically relate to the behaviors of services. Security policies can be human-readable and become part of a supplemental service-level agreement or can be machine-readable processed at runtime. Machine-readable policies are defined using the WS-Policy language and related WS-Policy specifications.

[WS-20] Policy expressions MUST be isolated into a separate WS-Policy definition document, which is then referenced within the WSDL document via the `wsp:PolicyReference` element.

[WS-21] Global or domain-specific policies SHOULD be isolated and applied to multiple services.

[WS-22] Policy attachment points SHOULD conform the WSDL 1.1 or later version, preferably version 2.0, attachment point elements and corresponding policy subjects (service, endpoint, operation, and message).

SOAP – web service security

116. Web Services Security (WSS): SOAP Message Security is a set of enhancements to SOAP messaging that provides message integrity and confidentiality. WSS: SOAP Message Security is extensible and can accommodate a variety of security models and encryption technologies. WSS: SOAP Message Security provides three main mechanisms that can be used independently or together:

- The ability to send security tokens as part of a message, and for associating the security tokens with message content;

- The ability to protect the contents of a message from unauthorized and undetected modification (message integrity); and
- The ability to protect the contents of a message from unauthorized disclosure (message confidentiality).

WSS: SOAP Message Security can be used in conjunction with other Web service extensions and application-specific protocols to satisfy a variety of security requirements.

[WS-23] Web Services using SOAP message SHOULD be protected accordance with WSS:SOAP Standard recommendations.

DATA TYPE FORMATS

117. This Standard recommends primitive data type formats such as time, date and language to be consistent with the recommendations of WIPO Standards ST.96 and ST.97 which are used for XML and JSON requests and responses respectively and for query parameters.

[CS-01] Time objects MUST be formatted as specified in IETF RFC 9557 (it is a profile of ISO 8601).

[CS-02] Time zone information along with time SHOULD be used as specified in IETF RFC 9557 (it is a profile of ISO 8601). Time along with time zone format is hh:mm:ss±hh:mm. For example: `20:54:21+00:00`

[CS-03] Date objects MUST be formatted as specified in IETF RFC 9557 (it is a profile of ISO 8601). Date format is YYYY-MM-DD. For example: `2018-10-19`

[CS-04] Datetime (i.e., timestamp) objects MUST be formatted as specified in IETF RFC 9557 (it is a profile of ISO 8601).

[CS-05] The relevant time zone to Datetime SHOULD be used as specified in IETF RFC 9557 (it is a profile of ISO 8601). Date with time along with time zone format is YYYY-MM-DDThh:mm:ss±hh:mm. For example: `2017-02-14T20:54:21+00:00`

[CS-06] ISO 4217-Alpha (3-Letter Currency Codes) MUST be used for Currency Codes. The precision of the value (i.e., number of digits after the decimal point) MAY vary depending on the business requirements.

[CS-07] WIPO Standard ST.3 two-letter codes be used for representing IPOs, states, other entities, organizations and for priority and designated countries/organizations.

[CS-08] ISO 3166-1-Alpha-2 Code Elements (2 letter country codes) MUST be used for the representation of the names of countries, dependencies, and other areas of particular geopolitical interest, on the basis of lists of country names obtained from the United Nations.

[CS-09] ISO 639-1 (2-Letter Language Codes) MUST be used for Language Codes.

[CS-10] Units of Measure SHOULD use the units of measure as described in The Unified Code for Units of Measure (based on ISO 80000 definitions). For example, for weight measuring using kilograms (kg)

[CS-11] Characters used in enumeration values MUST be restricted to the following set: {a-z, A-Z, 0-9, period (.), comma (,), spaces ( ), dash (-) and underscore (_)}.

[CSJ-12] The Representational Terms in Annex VI MUST be used for atomic property names.

[CSJ-13] Acronyms and abbreviations appearing at the beginning of a property name MUST be in lower case. Otherwise, all values of an enumeration, acronyms and abbreviation values MUST appear in upper case.

CONFORMANCE

118. This Standard is designed as a set of design rules and conventions that can be layered on top of existing or new Web Service APIs to provide common functionality. Not all services will support all of the conventions defined in the Standard due to business (for example, QoS may not be required) or technical constraints (for example, OAuth 2.0 may already be used).

119. This Standard defines two levels of conformance: A and AA Conformance Levels. Note that rules indicates by MAY are not considered important when determining conformance.

120. The Web Service APIs are encouraged to support as much additional functionality beyond their level of conformance as is appropriate for their intended scenario.

121. Two conformance levels are defined:

- **Level A**: For Level A conformance, the API indicates that the required general design rules (RSG), which are identified as 'MUST' in this Standard, are followed. In addition, the rules specific to the type of response returned must also be complied with, in other words, the following conformance sub-level are indicated:
  - Level AJ: Returning an ST.97 JSON response, must comply with all general level rules (RSG) identified as MUST as well as all JSON specific rules (RSJ) identified as MUST;
  - Level AX: Returning an ST.96 XML instance, must comply with all general level rules (RSG) identified as MUST as well as all XML specific rules (RSX) identified as MUST; and
  - Level A: Returning either a JSON or XML response, must comply with all general level rules (RSG) identified as MUST as well as all JSON specific rules (RSJ) identified as MUST and all XML specific rules (RSX) identified as MUST.
- **Level AA**: For Level AA conformance, the API indicates that is Level A compliant and all the recommended design rules, which are identified as 'SHOULD' in this Standard, are followed. As with Level A, there are sub-levels dependent upon the type of response:
  - Level AAJ: Level AJ compliance as well as the recommended SHOULD rules applicable to a JSON response; and
  - Level AAX: Level AX compliance as well as the recommended SHOULD rules applicable to an XML response.

122. The traceability matric between the design rules and the conformance levels is listed in Annex I.

REFERENCES

WIPO Standards

WIPO Standard ST.3	Two-letter codes for the representation of states, other entities and organizations
WIPO Standard ST.96	Processing of Intellectual Property information using XML
WIPO Standard ST.97	Processing of Intellectual Property information using JSON

Standards and conventions

Note that these external standards tend to evolve on their own. As IETF standards evolve the IETF documentation identifies which standards have become obsolete.

IETF RFC 2518:	HTTP Extensions for Distributed Authoring – WEBDAV - https://www.rfc-editor.org/rfc/rfc2518
IETF RFC 3986	Uniform Resource Identifier (URI): Generic Syntax – www.ietf.org/rfc/rfc3986.txt
IETF RFC 4918:	HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV) – https://www.rfc-editor.org/rfc/rfc4918
IETF RFC 5842:	Binding Extensions to Web Distributed Authoring and Versioning (WebDAV) – https://www.rfc-editor.org/rfc/rfc5842
IETF RFC 5789	PATCH Method for HTTP – https://tools.ietf.org/rfc/rfc5789.txt
IETF RFC 6648	Deprecating the "X-" Prefix and Similar Constructs in Application Protocols - https://tools.ietf.org/rfc/rfc6648.txt
IEFT RFC 7396	JSON Merge Patch – https://www.rfc-editor.org/rfc/rfc7396
IETF RFC 8144:	Use of the Prefer Header Field in Web Distributed Authoring and Versioning (WebDAV) – https://www.rfc-editor.org/rfc/rfc8144
IETF RFC 8288:	Web Linking – https://datatracker.ietf.org/doc/html/rfc8288
IETF RFC 8297:	An HTTP Status Code for Indicating Hints – https://www.rfc-editor.org/rfc/rfc8297
IETF RFC 9110	HTTP Semantics – https://www.ietf.org/rfc/rfc9110.pdf
IETF RFC 9111	HTTP Caching – https://datatracker.ietf.org/doc/html/rfc9111
IETF RFC 9557	Date and Time on the Internet: Timestamps – https://datatracker.ietf.org/doc/html/rfc9557
ISO 639-1	Language codes – https://www.iso.org/iso-639-language-code
ISO 3166-1 alpha-2	Two-letter acronyms for country codes – https://www.iso.org/iso-3166-country-codes.html
ISO 4217	Currency Codes – www.iso.org/iso/home/standards/currency_codes.htm
ISO 8601	Date and Time Formats – https://www.iso.org/iso-8601-date-and-time-format.html

IANA Internet Assigned Number authority: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

Odata https://www.odata.org/

OASIS OData Metadata Service Entity Model: http://docs.oasisopen.org/odata/odata/v4.0/os/models/MetadataService.edmx

OASIS OData JSON Format Version 4.0. Edited by Ralf Handl, Michael Pizzo, and Mark Biamonte. Latest version https://docs.oasis-open.org/odata/odata/v4.0/os/models/MetadataService.edmx

OASIS OData Atom Format Version 4.0. Edited by Martin Zurmuehl, Michael Pizzo, and Ralf Handl. Latest version:

http://docs.oasis-open.org/odata/odata-atom-format/v4.0/odata-atom-format-v4.0.html

OASIS OData OData Version 4.0

- Part 1: Protocol – http://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.html
- Part 2: URL Conventions – http://docs.oasis-open.org/odata/odata/v4.0/os/part2-url-conventions/odata-v4.0-os-part2-url-conventions.html
- Part 3: Common Schema Definition Language (CSDL) – http://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html

OASIS ABNF components: OData ABNF Construction Rules Version 4.0 and OData ABNF Test Cases:

http://docs.oasis-open.org/odata/odata/v4.0/os/abnf/

OASIS Vocabulary components: OData Core Vocabulary, OData Measures Vocabulary and OData Capabilities Vocabulary:
http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/

OASIS XML schemas:

OData EDMX XML Schema and OData EDM XML Schema

http://docs.oasis-open.org/odata/odata/v4.0/os/schemas/

OASIS SAML 2.0 http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html

RAML (ReSTful API Modeling Language) http://raml.org

OpenAPI Initiative www.openapis.org

Richardson's REST API Maturity Model https://martinfowler.com/articles/richardsonMaturityModel.html

HAL http://stateless.co/hal_specification.html

JSON-LD https://json-ld.org

Collection+JSON Document Format http://amundsen.com/media-types/collection/format/

BadgerFish http://badgerfish.ning.com/

Semantic Versioning https://semver.org/

REST https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

CQL https://en.wikipedia.org/wiki/Contextual_Query_Language

Z39.50 https://www.loc.gov/z3950/agency/Z39-50-2003.pdf

WS-I Basic Profile 2.0 http://ws-i.org/profiles/BasicProfile-2.0-2010-11-09.html

W3C SOAP 1.2 Part 1 Messaging Framework – https://www.w3.org/TR/soap12-part1/

W3C SOAP 1.2 Part 2 Adjuncts – https://www.w3.org/TR/soap12-part2/

W3C WSDL Version 2.0 Part 1 Core Language – https://www.w3.org/TR/wsdl20/

W3C CORS https://www.w3.org/TR/cors/

W3C Matric Parameters https://www.w3.org/DesignIssues/MatrixURIs.html

IP Offices' REST APIs

EPO Open Patent Services OPS v 3.2 https://developers.epo.org

USPTO PatentsView	https://patentsview.org
WIPO ePCTv1.1	https://pct.wipo.int/
EUIPO TMview	https://www.tmdn.org/tmview/#/tmview
EUIPO Designview	https://www.tmdn.org/tmdsview-web/#/dsview
TMclass	https://tmclass.tmdn.org/ec2/
DESIGNclass	https://euipo.europa.eu/designclass/

Industry REST APIs and Design Guidelines

Facebook	https://developers.facebook.com/docs/graph-api/reference
GitHub	https://developer.github.com/v3
Google APIs Design Guide	https://cloud.google.com/apis/design/
Azure	https://docs.microsoft.com/en-us/rest/api/
OpenAPI	https://swagger.io/docs/specification/about/
OData	http://www.odata.org/documentation/
JSON API	http://jsonapi.org/format/
Microsoft API Design	https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design
JIRA REST API	https://developer.atlassian.com/server/jira/platform/jira-rest-api-examples
Confluence REST API	https://developer.atlassian.com/server/confluence/
Ebay API	https://developer.ebay.com/api-docs/static/ebay-rest-landing.html
Oracle REST Data Services	http://www.oracle.com/technetwork/developer-tools/rest-data-services/overview/index.html
PayPal REST API	https://developer.paypal.com/docs/api/overview/
Data on the Web Best Practices	https://www.w3.org/TR/dwbp/#intro
SAP Guidelines for Future	
REST API Harmonization	https://help.sap.com/docs/api-style-guide/sap-api-style-guide-public/rest-and-odata-api-documentation
GitHub API	https://developer.github.com/v3/
Zalando	https://github.com/zalando/ReSTful-api-guidelines
Dropbox	https://www.dropbox.com/developers
X	https://docs.x.com/home

Others

CQRS	https://martinfowler.com/bliki/CQRS.html
ITU	https://www.itu.int/en/ITU-T/ipr/Pages/open.aspx
OWASP Rest Security Cheat Sheet	https://www.owasp.org/index.php/REST_Security_Cheat_Sheet
DDD	https://martinfowler.com/bliki/BoundedContext.html
REST Principles	https://en.wikipedia.org/wiki/Representational_state_transfer

Open/Closed Principle	https://en.wikipedia.org/wiki/Open/closed_principle
Which style of WSDL should I use?	https://www.ibm.com/developerworks/library/ws-whichwsdl/
New Zealand Government	
API Standard and Guidelines	https://www.digital.govt.nz/standards-and-guidance/technology-and-architecture/application-programming-interfaces-apis/api-guidelines
Cross site scripting prevention cheat sheet:	
	https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
OWASP Cheat Sheet Series	https://cheatsheetseries.owasp.org/
Digital Signature Standard (DSS)	https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf
SOAP Message Security 1.0, OASIS Standard 200401	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf
SOA Principles of Service Design, Thomas Erl (2008)	

[Annex I follows]

ANNEX I

LIST OF RESTFUL WEB SERVICE DESIGN RULES AND CONVENTIONS AND CONFORMANCE INDICATORS

Version 2.0

Revision approved by the Committee on WIPO Standards (CWS)
at its thirteenth session on November 14, 2025

Annex I of WIPO Standard ST.90 provides the list of design rules and conventions for restful web services, and their relevant indicators which identify basic conformance requirements in terms of which conformance level, Web Services API implementation support.

The List of Restful Web Service Design Rules and Conventions and Conformance Indicators is available at:
https://www.wipo.int/documents/d/standards/docs-en-03-90-01-annex-i-v2-0.xlsx

- Letter “X” in the column “C” of the table indicates that the design rule must be complied with, in order to achieve a Level AJ compliance (for a JSON response);
- Letter “X” in the column “D” of the table indicates that the design rule must be complied with, in order to achieve a Level AX compliance (for an XML response);
- Letter “X” in the column “E” of the table indicates that the design rule must be complied with, in order to achieve a Level AAJ compliance (for a JSON response); and
- Letter “X” in the column “F” of the table indicates that the design rule must be complied with, in order to achieve a Level AAX compliance (for an XML response).

[Editorial Note: In order achieve a Level A compliance, it is just necessary to follow rules that have an “X” in both Column “C” and “D”. In order to achieve a Level AA compliance, it is necessary to follow rules that have an “X” in both Column “E” and “F”. The third letter indicates the type of response provided.]

[Annex II follows]

ANNEX II

REST IP VOCABULARY

Version 2.0

Revision approved by the Committee on WIPO Standards (CWS)
at its thirteenth session on November 14, 2025

1. The following IP Vocabulary is provided in Table 1 as examples of /basic RESTful Service Request parameters. IP Offices will likely encounter the need to develop more complex requests and varied response payloads according to their business needs. The parameters in this table are examples of ST.97 elements, used for a JSON response. The complete ST.97 IP JSON Schemas can be consulted in the Annex II of WIPO ST.97, or alternatively, when referring to XML-based APIs, these parameters correspond to the ST.96 elements in lowerCamelCase. The complete ST.96 IP data dictionary and IP XML Schemas can be accessed from this location: https://www.wipo.int/standards/en/st96/.

[Editorial Note: In the future, it is planned to provide a link to a more comprehensive list of REST IP XML and JSON vocabulary which will be dynamically maintained on an ongoing basis as IP elements and vocabulary continue to evolve.]

Table 1: Example API Business Vocabulary

Business Domain(s)	Resource Name(s)	Parameter Name	Description
ALL	/trademarks /patents /designs	`st13ApplicationNumber`	The application number for the filed IP, using WIPO ST.13 format which is a string of several values including the national application number, IP Type, and the country/organization.
ALL	/trademarks /patents /designs	`applicationNumber`	The application number for the filed IP in the format of the national office.
MULTIPLE	/trademarks /designs	`internationalRegistrationNumber`	The International Registration Number of the IP right. For Trademarks this pertains to the Madrid System. For Industrial Designs, this pertains to the Hague system.
ALL	/trademarks /patents /designs	`availableDocument`	Single document entry relevant to the search criteria provided to DocList API.
ALL	/trademarks /patents /designs	`sortingCriteria`	Sorting Criterion used by the DocList API.

ALL	/trademarks /patents /designs	receivingOfficeCode	The IP Office, in WIPO ST.3 format.
ALL	/trademarks /patents /designs	receivingOfficeDate	The date received at the IP office.
Trademarks	/trademarks	applicationDate	The date of the application.
		registrationDate	The date registered at the IP office.
		markFeatureCategory	The category of mark feature.
		markCurrentStatusCode	Code of the current legal status of the application.
		markCurrentStatusDate	Date of the current legal status of the application.
Patents	/patents	filingDate	The date that the application was filed.
		grantPublicationDate	The date that the grant was published.
		fileReferenceIdentifier	Applicants reference number.
		applicationBodyStatus	Status of the application body.
		statusEventData	Data associated with a legal status event in relation to a specific patent application.
		keyEventCode	A code indicating a broad, high-level event that covers the most general and important situations in a category.
Industrial Designs	/designs	applicationDate	The date that the application was filed.
		designApplicationCurrentStatus	Category of current legal status of the design application.
		designApplicationCurrentStatusDate	Date of the current legal status of the design application.

2. The following technical query parameters defined in Table 2 should apply to all the REST API services:

Table 2: API Technical Vocabulary

Query/Path Parameter	Parameter Value Data Type	Constraint	Format	Description	Design Rule
format	string		`type/subtype; parameter=value` according to RFC7231, 3.1.1.1. Media Type	Used for content-type negotiation (prefer a HTTP request header).	[RSG-19]
v	string		`v%` where % is a positive integer	Used for service versioning (prefer indicating version as path segment of the URL).	[RSG-64]
limit	integer	positive	`limit=10`	The page size used for pagination.	[RSG-73]
offset	integer	positive; default is 0	`offset=5`	The offset used for pagination.	[RSG-73]
sort	comma-separated list of strings	Possible values: a. asc b. desc	`sort=key1:asc,key2:desc`	Multi-attribute sorting criterion.	[RSG-74] – [RSG-76]
expand	comma-separated list of strings		`expand=key1,key2`	Used for expanding the body of the returned content.	[RSG-77]
count	boolean	Default is false	`count=true`	Returns the number of items in a collection (may be inline).	[RSG-81]
apiKey	string		`apiKey=abcdef12345`	Used to indicate a Web API Key (a HTTP header should be preferred).	[RSG-137] – [RSG-138]

[Annex III follows]

ANNEX III

RESTFUL WEB API GUIDELINES AND MODEL SERVICE CONTRACT

Version 1.1

*Revision approved by the Committee on WIPO Standards (CWS)
at its tenth session on November 25, 2022*

1. Annex III provides two example models of Standard-compliant API specifications which intend to provide guidance to Intellectual Property Offices (IPOs) which wish to develop web services according to this Standard. Details regarding two example models are provided below and Appendixes A and B.

2. It should be noted that the example models were produced using a hybrid-approach of contract-first and code-first approaches.

DOCLIST EXAMPLE MODEL

3. The first of the example models was inspired by the IP5[12] Office Open Portal Dossier (OPD) set of web services, provided with the same name. The DocList API provides a list of relevant patent documents associated with at least an application or publication number.

PATENT LEGAL STATUS EXAMPLE MODEL

4. The second of the example models is the patent legal status API which provides either the history of legal status events for a particular application number or else the details of a particular legal status event.

[Appendices A and B to Annex III follow]

[12] The IP5 Offices are comprised of Chinese National Intellectual Property Administration (CNIPA), European Patent Office (EPO), Japan Patent Office (JPO), Korean Intellectual Property Office (KIPO) and the United States Patent and Trademark Office (USPTO).

APPENDIX A

DOCLIST EXAMPLE MODEL

1. Appendix A provides a link to a zip file which includes the requirements document which outlines the request and response formats, the YAML specification and the XSD components.

2. Appendix A is available at:

https://www.wipo.int/standards/en/st90/annex-iii_appendix_a_V1_0.zip

APPENDIX B

PATENT LEGAL STATUS EXAMPLE MODEL

1. Appendix B provides a link to zip file provided here include the API specification provided in RAML, example data and WIPO Standard ST.96 enumeration lists.

2. Appendix B is available at:

https://www.wipo.int/standards/en/st90/annex-iii_appendix_b_V1_0.zip

[Annex IV follows]

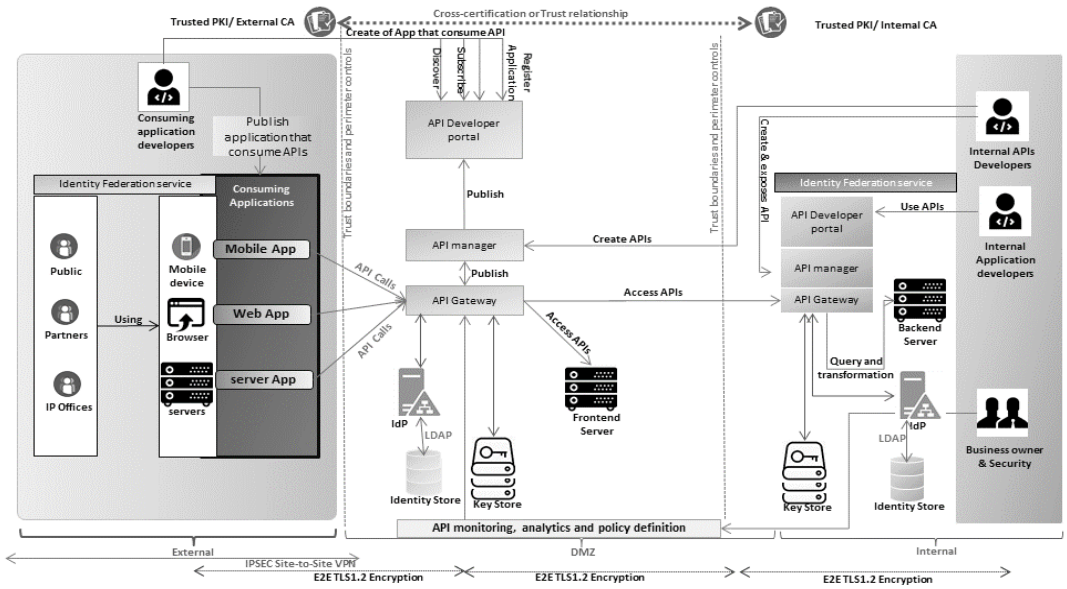
ANNEX IV

HIGH LEVEL SECURITY ARCHITECTURE BEST PRACTICES

Version 2.0

Revision approved by the Committee on WIPO Standards (CWS)
at its thirteenth session on November 14, 2025

1. The security architecture defines the services and mechanisms that should be implemented to enforce defined policies and rules while also providing a framework to further standardize and automate security. The core services and mechanisms of this API Security Framework (the development portal, API manager and API gateway) provide a grouping of functionalities. These functions can be delivered by discrete applications, bespoke code development, via COTS products or through leveraging existing technologies that can be configured to provide these functions / services. Some of the functionality may overlap or be combined into one or more products depending on the vendor used.



2. The recommended security architecture SHOULD have the following API security services and mechanisms:

- A Web API portal to provide functions such as API discovery, API analytics, access to specifications and description including SLAs, social network and FAQs;
- A Web API manager to provide centralized API administration and governance for API catalogues, management of registration and on-boarding of various API developer communities, API lifecycle management, application of pre-defined security profiles, and security policies lifecycle management;
- A Web API gateway to provide security automation capabilities including but not limited to centralized threat protections, centralized API authentication, authorization, logging, security policy enforcement, message encryption, monitoring, and analytics;
- A Web API monitoring and analytics service to provide functions such as advanced API services monitoring, analytics, profile usage for security baselines, changes of usage and demand;
- A credential store to provide capabilities to securely store API keys, secrets, certificates, etc.;

- A trusted Certificate Authority (CA) to issue secure certificates and enable trust establishment between the various Offices;
- A Security Information and Event Management system (SIEM) to enable security logs correlation and advanced security analytics and monitoring;
- An Identity Provider to manage the identities stored in the LDAP directories and enable authentication; and
- A Web application scanning product that performs regular security scans and performs analysis based on a trusted security baseline such as OWASP Top 10.

[Annex V follows]

ANNEX V

HTTP STATUS CODES

Version 2.0

*Revision approved by the Committee on WIPO Standards (CWS)
at its thirteenth session on November 14, 2025*

1. It is important to align responses around the appropriate HTTP status code and to follow the standard HTTP codes. In addition to an appropriate status code, there should be a useful and concise description of the error in the body of your HTTP response. Responses should be specific and clear so consumers can come to a conclusion very quickly when using the API.

2. The set of HTTP status codes is defined on the basis of in IETF RFC 9110. The status codes listed below should be used by an API, where applicable.

3. The following response status code categories are defined:

- 1xx: Informational - Communicates transfer protocol-level information;
- 2xx: Success - Indicates that the client's request was accepted successfully;
- 3xx: Redirection - Indicates that the client must take some additional action in order to complete their request;
- 4xx: Client Error - This category of error status codes points the finger at clients; and
- 5xx: Server Error - The server takes responsibility for these error status codes.

4. The following table consolidates the HTTP Status Codes and provides references to the relative IETF RFCs.

Value	Description	Reference
100	Continue	[IETF RFC 9110, Section 15.2.1]
101	Switching Protocols	[IETF RFC 9110, Section 15.2.2]
102	Processing	[IETF RFC 2518]
103	Early Hints	[IETF RFC 8297]
104-199	Unassigned	
200	OK	[IETF RFC 9110, Section 15.3.1]
201	Created	[IETF RFC 9110, Section 15.3.2]
202	Accepted	[IETF RFC 9110, Section 15.3.3]
203	Non-Authoritative Information	[IETF RFC 9110, Section 15.3.4]
204	No Content	[IETF RFC 9110, Section 15.3.5]
205	Reset Content	[IETF RFC 9110, Section 15.3.6]
206	Partial Content	[IETF RFC 9110, Section 15.3.7]
207	Multi-Status	[IETF RFC 4918]
208	Already Reported	[IETF RFC 5842]
209-225	Unassigned	
226	IM Used	[IETF RFC 3229]

227-299	Unassigned	
300	Multiple Choices	[IETF RFC 9110, Section 15.4.1]
301	Moved Permanently	[IETF RFC 9110, Section 15.4.2]
302	Found	[IETF RFC 9110, Section 15.4.3]
303	See Other	[IETF RFC 9110, Section 15.4.4]
304	Not Modified	[IETF RFC 9110, Section 15.4.5]
305	Use Proxy	[IETF RFC 9110, Section 15.4.6]
306	(Unused)	[IETF RFC 9110, Section 15.4.7]
307	Temporary Redirect	[IETF RFC 9110, Section 15.4.8]
308	Permanent Redirect	[IETF RFC 9110, Section 15.4.9]
309-399	Unassigned	
400	Bad Request	[IETF RFC 9110, Section 15.5.1]
401	Unauthorized	[IETF RFC 9110, Section 15.5.2]
402	Payment Required	[IETF RFC 9110, Section 15.5.3]
403	Forbidden	[IETF RFC 9110, Section 15.5.4]
404	Not Found	[IETF RFC 9110, Section 15.5.5]
405	Method Not Allowed	[IETF RFC 9110, Section 15.5.6]
406	Not Acceptable	[IETF RFC 9110, Section 15.5.7]
407	Proxy Authentication Required	[IETF RFC 9110, Section 15.5.8]
408	Request Timeout	[IETF RFC 9110, Section 15.5.9]
409	Conflict	[IETF RFC 9110, Section 15.5.10]
410	Gone	[IETF RFC 9110, Section 15.5.11]
411	Length Required	[IETF RFC 9110, Section 15.5.12]
412	Precondition Failed	[IETF RFC 9110, Section 15.5.13] [IETF RFC 8144, Section 3.2]
413	Content Too Large	[IETF RFC 9110, Section 15.5.14]
414	URI Too Long	[IETF RFC 9110, Section 15.5.15]
415	Unsupported Media Type	[IETF RFC 9110, Section 15.5.16] [IETF RFC 7694, Section 3]
416	Range Not Satisfiable	[IETF RFC 9110, Section 15.5.17]
417	Expectation Failed	[IETF RFC 9110, Section 15.5.18]
418-420	Unassigned	
421	Misdirected Request	[IETF RFC 9110, Section 15.5.20]
422	Unprocessable Entity	[IETF RFC 9110, Section 15.5.21] [IETF RFC 4918]
423	Locked	[IETF RFC 4918]
424	Failed Dependency	[IETF RFC 4918]
425	Unassigned	
426	Upgrade Required	[IETF RFC 9110, Section 15.5.22]
427	Unassigned	
428	Precondition Required	[IETF RFC 6585]

429	Too Many Requests	[IETF RFC 6585]
430	Unassigned	
431	Request Header Fields Too Large	[IETF RFC 6585]
432-450	Unassigned	
451	Unavailable For Legal Reasons	[IETF RFC 7725]
452-499	Unassigned	
500	Internal Server Error	[IETF RFC 9110, Section 15.6.1]
501	Not Implemented	[IETF RFC 9110, Section 15.6.2]
502	Bad Gateway	[IETF RFC 9110, Section 15.6.3]
503	Service Unavailable	[IETF RFC 9110, Section 15.6.4]
504	Gateway Timeout	[IETF RFC 9110, Section 15.6.5]
505	HTTP Version Not Supported	[IETF RFC 9110, Section 15.6.6]
506	Variant Also Negotiates	[IETF RFC 2295]
507	Insufficient Storage	[IETF RFC 4918]
508	Loop Detected	[IETF RFC 5842]
509	Unassigned	
510	Not Extended	[IETF RFC 2774]
511	Network Authentication Required	[IETF RFC 6585]
512-599	Unassigned	

[Annex VI follows]

ANNEX VI

REPRESENTATIONAL TERMS

Version 1.1

*Revision approved by the Committee on WIPO Standards (CWS)
at its tenth session on November 25, 2022*

Term	Definition	Data Type
Amount	A monetary value.	`Number`
Category	A specifically defined division or subset in a system of classification in which all items share the same concept of taxonomy.	`String`
Code	A combination of one or more numbers, letters, or special characters, which is substituted for a specific meaning. Represents finite, predetermined values or free format.	`String`
Date	The notion of a specific point in time, expressed by year, month, and day.	`String`
Directory	Always preceded by PATH.	`String`
Document	A CLOB stands for "Character Large OBject," which is a specific data type for almost all databases. Quite simply, a CLOB is a pointer to text stored outside of the table in a dedicated block. Used for XML documents. Comprised of textual information of International Trademark Registration being exchanged. XML tags identify the data items concerned with such information. TIS - Madrid development team may define the attribute XML_DOC as CLOB, pointer to Tagged Data stored outside of the table in a dedicated block.	`String`
Identifier	A combination of one or more integers, letters, special characters which uniquely identifies a specific instance of a business object, but which may not have a readily definable meaning.	`String`
Indicator	A signal of the presence, absence, or requirement of something. Recommended values are "`Y`", "`N`", and, "`?`" if needed.	`Boolean`
Measure	A measure is a numeric value determined by measuring an object along with the specified unit of measure. `MeasureType` is used to represent a kind of physical dimension such as temperature, length, speed, width, weight, volume, latitude of an object. More precisely, `MeasureType` should be used to measure intrinsic or physical properties of an object seen as a whole.	`Number`
Name	The designation of an object expressed in a word or phrase.	`String`
Number	A string of numeral or alphanumeric characters expressing label, value, quantity or identification.	`Number, String`
Percent	A number which represents a part of a whole, which will be divided by 100.	`Number`

Term	Definition	Data Type
Quantity	A quantity is a counted number of non-monetary units, possibly including fractions. `Quantity` is used to represent a counted number of things. `Quantity` should be used for simple properties of an object seen as a composite or collection or container to quantify or count its components. `Quantity` should always express a counted number of things, and the property will be such as total, shipped, loaded, stored. `QuantityType` should be used for components that require unit information; and `xsd:nonNegativeInteger` should be used for countable components which do not need unit information.	`Number`
Rate	A quantity or amount measured in relation to another quantity or amount.	`Number`
Text	An unformatted character string, generally in the form of words. (includes: Abbreviation, Comments.)	`String`
Time	A designation of a specified chronological point within a period.	`Date`
DateTime	The captured date and time of an event when it occurs.	`Date`
URI	The Uniform Resource Identifier that identifies where the file is located.	`String`

[Annex VII follows]

ANNEX VII

API LIFECYCLE MANAGEMENT PUBLICATION

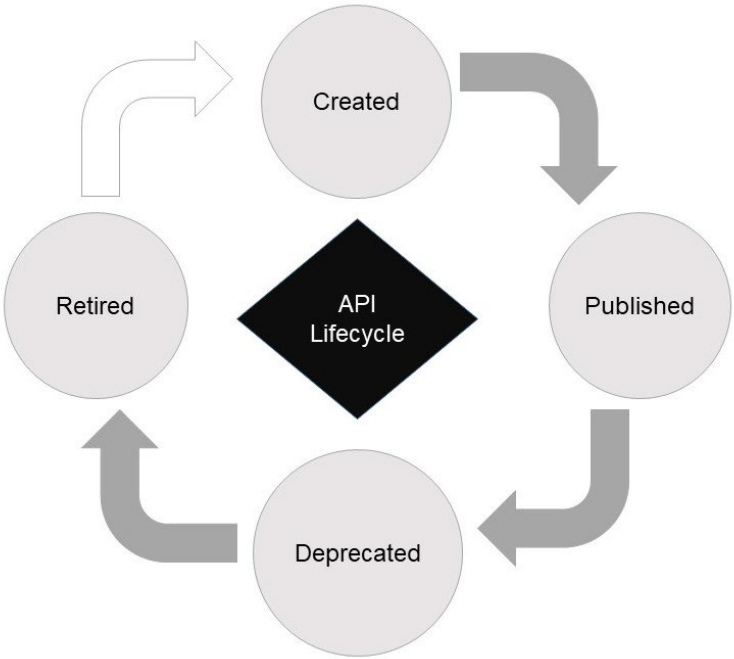
Version 1.1

*Revision approved by the Committee on WIPO Standards (CWS)
at its tenth session on November 25, 2022*

1. This Annex provides a brief overview of API Lifecycle management and suggests key pieces of information that should be published in a policy document by an IP Office to assist API consumers in understanding how best to use these APIs.

2. API Lifecycle management is a critical aspect of an API strategy as it provides the framework for the life of an API from creation through to retirement. It is useful both internally for the developers and operations teams and also externally for API consumers. For internal developers, it helps create a structure and set expectations for developing an API, and for the operations teams it assists with the understanding of support requirements. For API consumers, both internally and externally, it provides an informal contract of expectations for when a particular API is used. This will become clear as each stage in the lifecycle is presented below.

3. Published API lifecycles can be comprised of simple 4-step processes or more complex with up to 10 or more steps. However, for the most part, the lifecycles with more steps are considered more detailed versions of the lifecycles with fewer steps. As such, this document will focus on the basic 4-step process necessary to capture an API lifecycle: Created -> Published -> Deprecated -> Retired. Any published API lifecycle document should incorporate at least a description of these four stages are managed by an IP Office.



CREATED

4. Creating an API focuses on designing, implementing and documenting the API. The critical consideration during the creation phase is to consider the purpose of the API and the overall structure necessary to 'future-proof' the API as much as possible. Ideally, the API should adhere to a set of internal and external standards, such those recommendations incorporated in the current Standard. If the API is to be monetized, then consideration should be given at this stage to define the monetisation strategy.

PUBLISHED

5. Once an API is created it needs to be published. It should be versioned using a standard versioning strategy and documentation should be provided including the API specification and sample requests and responses (see [RSG-64]-[RSG-65]). Once published, the API is consumed by applications. Note that fixes and enhancements may be incorporated during the Publish stage.

DEPRECATED

6. At some point an API is no longer useful. It has either been superseded by a newer version of an API or is the no longer relevant, because of some external or internal factor. API Consumers should be contacted and preparation made to remove the API from the catalogue. At this stage it is likely to only major bugs with the API will be fixed.

RETIRED

7. This is the stage where the API is decommissioned. This should include disabling access to the API and removing it from API platform. Consideration should be given as to whether "extended support" will be offered or if there are any cases in which retirement would be delayed.

8. The last two stages are the most important to document in terms of the lifecycle management, the deprecation and retirement stages. It is critical for API consumers to understand the expectations placed on them when they start to use an API to avoid disappointment or challenges when trying to remove an API from the catalogue. This should include, for example, management of major and minor versions and any timelines for notification of changes. At a high level, there tends to be two approaches to API deprecation/retirement: either retaining a previously stated number of versions or retaining old versions for a specified time period. A combination of these approaches can also be used but either the number of older versions which are to be supported or the length of time that old versions are retained must be clearly stated in the published lifecycle document.

[End of Annex VII and of Standard]