

API Design Rules

Global IT

19 de diciembre de 2022

Contenido

Version Control	3
Common Vocabulary and acronyms	4
1. Introduction	5
2. APIfication Principles.....	6
2.1. Domain Driven Design.....	6
2.2. API First	7
2.3. Interface standardization. Standardization organisms.	8
2.4. Information Representation Standard	9
3. API Definition	11
3.1. API REST.....	11
HTTP Verbs	13
3.2. HTTP Response Codes	14
3.3. Query Parameters Use	15
3.4. Path Parameters use	16
3.4.1. Good Practices.....	16
3.5. HTTP Headers definition	17
3.5.1. Not allowed HTTP headers.....	18
3.6. MIME Types.....	19
4. API resources definition	20
4.1. URL Definition	20
4.1.1. Good practices.....	20
4.1.2. Hierarchy	21
4.2. Input/Output resources definition.....	21
5. Versioning.....	22
5.1. Versioning Strategy	22
5.2. Backward and forward compatibility	22
5.2.1. Types of modifications	23
5.2.2. Compatibility Management	23
6. Error Responses.....	24
6.1. Internal Errors	24
6.2. External Errors.....	24
6.3. Errors Mapping.....	24
7. Common Data Types (AddressType structure, E164Type, EmailAddressType, ExceptionType structure,...) [Open Title to be updated over the time]	25
8. Filtering, sorting and pagination	25

8.1.	Pagination.....	25
8.2.	Sorting	26
8.3.	Filtering.....	26
9.	Architecture Headers	28
10.	Security.....	29
10.1.	APIs Rest Security	29
10.1.1.	REST security design principles	29
10.1.2.	Good practices to secure REST APIs	30
10.2.	Security Implementation.....	31
10.2.1.	Channel security	31
10.2.1.1.	TLS and mutual authentication	31
10.2.1.2.	Certificate Chain Validation.....	31
10.2.1.3.	Recommended Ciphers	31
10.2.2.	Access Security	32
10.2.2.1.	Oauth.....	32
10.2.2.2.	Scopes	32
10.2.3.	Data Security	33
10.2.3.1.	Headers Validation	33
10.2.3.2.	Response body validation	34
10.2.3.3.	<i>Do not repudiate</i>	34
11.	Definition in Open API.....	35
11.1	General Information.....	35
11.2	Published routes.....	35
11.3	Request Parameters	36
11.4	Response structure	36
11.5	Data definitions	37
11.6	OAuth Definition	38

Version Control

Version	Date	Updates	Author
V1.0.0	04/04/2022	First English Version	Global IT
V2.0.0	25/08/2022	<ul style="list-style-type: none">• Title 6 adapted to new agreements with Telefonica Kernel errors structures.• Snake_case added to common Vocabulary and acronyms.• Title 8.1 Pagination updated with seek filter to systems with more than 1000 records.• Title 9 Headers updated with "X-" due to other Platforms architectures.• Title 7 updated to clarify that it is opened to be updated over the time.• Title 10.2.2.2 updated changing dots (.) by hyphens (-) to rank the scopes' definitions.	Global IT
V3.0.0	16/11/2022	<ul style="list-style-type: none">• HTTP error code to be included in any response code (cf IETF RFC 7807) -- PR#96• TMF notation as for filtering options -- PR#72	Global IT
V4.0.0	19/12/2022	<ul style="list-style-type: none">• Error codes divided between Internal and External ones. Error Code Mapping table added,• Architecture Headers conformed and tagged by showable on swagger or not.	Global IT

Common Vocabulary and acronyms

Término	Definición
API	Application Programming Interface. It is a rule & specification group (code) that applications follow to communicate between them, used as interface among programs developed with different technologies.
Body	HTTP Message body (If exists) is used to carry the entity data associated with the request or response.
Camel Case	It is a kind of define the fields' compound name or phrases without whitespaces among words. It uses a capital letter at the beginning of each word. There are two different uses: <ul style="list-style-type: none">• Upper Camel Case: When the first letter of each word is capital.• Lower Camel Case: Same that Upper one but with the first Word in lowercase.
Header	HTTP Headers allow client and server send additional information joined to the request or response. A request header is divided by name (No case sensitive) followed by colon and the header value (without line breaks). White spaces on the left hand from the value are ignored.
HTTP	Hypertext Transfer Protocol (HTTP) is communication protocol that allows the information transfer using files (XHTML, HTML...) in World Wide Web.
JSON	JavaScript Object Notation
JWT	JSON Web Token (JWT) is an open standard based on JSON proposed by IETF (RFC 7519) for access token creations allowing the identity and purposes spread.
Kebab-case	Practice in the words denomination where the hyphen is used to separate words.
OAuth	Open Authorization is an open standard that allows simple Authorization flows to be used in web sites or applications.
REST	Representational State Transfer
TLS	Transport Layer Security or TLS is a cryptographic protocol that provides secured network communications.
URI	Uniform Resource Identifier
Snake_case	Practice in the words denomination where the underscore is used to separate words.

1. Introduction

The purpose of this document is to technically describe aspects related to the proper design of Application Programming Interfaces (hereinafter API), so that it serves as a recommended reference for the development of APIs in Telefónica operators and projects.

Based on principles of standardization, normalization and good practices, this document explains the guidelines for the design and definition of an API, elaborating the following points:

- How to make a proper API definition
- What are the main aspects to cover
- What are the best practices for designing the API and managing it properly

This document is addressed to all the roles and people that may participate in a Programmable Interface (API) design in Telefónica and supposes that they have basic knowledge of what an API is.

2. APIfication Principles

APIs are critical components of digitization that enable us to expose the functions and capabilities existing in our systems and network in a secure and standardized way (service channels, between systems/platforms and third parties/partners) without needing to redesign or re-create them (enhancing reuse) with the consequent saving of time and investment.

The APIfication program, as Telefónica's value proposition, is based on the following key principles: Domain Driven Design, API First and Standardization.

2.1. Domain Driven Design

The main idea of the Domain Driven Design (DDD) approach is: "to develop software for a complex domain, we need to build a ubiquitous language that embeds the domain's terminology in the software systems we build." (Fowler, 2020)

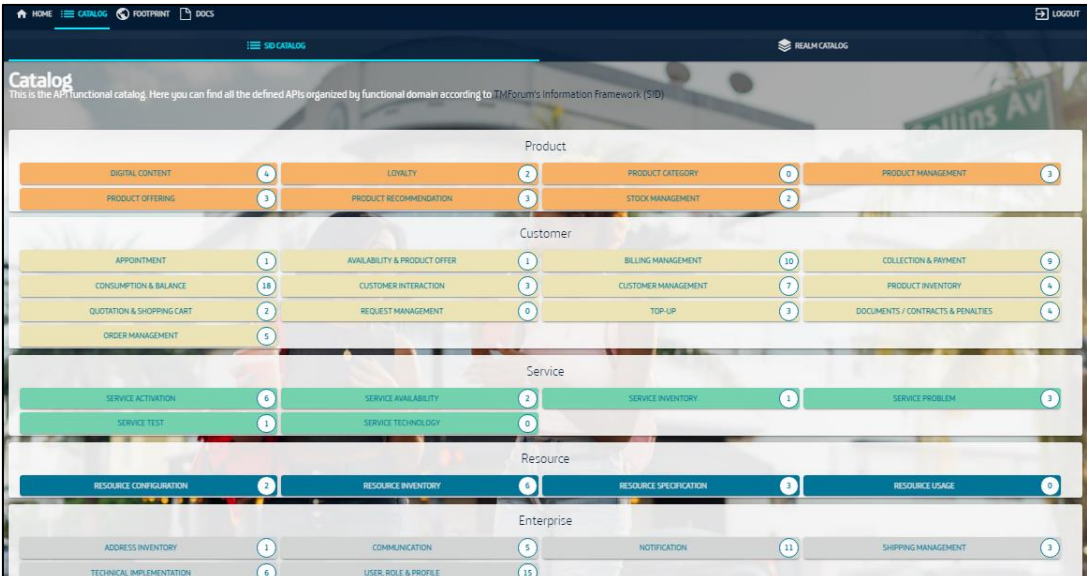
This approach to software development focuses efforts on defining, for each defined domain, a model that includes a broad understanding of the processes and rules of that domain.

DDD approach is particularly suitable for complex domains, where a lot of messy logic needs to be organized.

The DDD principles are based on:

- Placing the organization's business models and rules in the core of the application.
- Obtain a better perspective at the level of collaboration between domain experts and developers, to conceive software with very clear objectives.

As an initial reference to define the different domains and subdomains, we rely on the TM Forum SID model:



The screenshot displays the 'SID CATALOG' interface. At the top, there are navigation links: HOME, CATALOG, FOOTPRINT, and BOOKS. The main header area includes the 'Catalog' title and a description: 'This is the API functional catalog. Here you can find all the defined APIs organized by functional domain according to TM Forum's Information Framework (SID)'. Below this, the catalog is organized into five main functional domains, each with a grid of specific API categories and their counts:

Product			
DIGITAL CONTENT (4)	LOYALTY (2)	PRODUCT CATEGORY (9)	PRODUCT MANAGEMENT (3)
PRODUCT OFFERING (3)	PRODUCT RECOMMENDATION (3)	STOCK MANAGEMENT (2)	

Customer			
APPOINTMENT (1)	AVAILABILITY & PRODUCT OFFER (1)	BILLING MANAGEMENT (10)	COLLECTION & PAYMENT (8)
CONSUMPTION & BALANCE (18)	CUSTOMER INTERACTION (3)	CUSTOMER MANAGEMENT (7)	PRODUCT INVENTORY (4)
QUOTATION & SHOPPING CART (2)	REQUEST MANAGEMENT (0)	TOP-UP (3)	DOCUMENTS / CONTRACTS & PENALTIES (4)
ORDER MANAGEMENT (5)			

Service			
SERVICE ACTIVATION (6)	SERVICE AVAILABILITY (2)	SERVICE INVENTORY (1)	SERVICE PROBLEM (3)
SERVICE TEST (1)	SERVICE TECHNOLOGY (0)		

Resource			
RESOURCE CONFIGURATION (2)	RESOURCE INVENTORY (9)	RESOURCE SPECIFICATION (3)	RESOURCE USAGE (0)

Enterprise			
ADDRESS INVENTORY (1)	COMMUNICATION (5)	NOTIFICATION (11)	SHIPPING MANAGEMENT (3)
TECHNICAL IMPLEMENTATION (6)	USER ROLE & PROFILE (15)		

This set of domains and subdomains is not fixed or static and can evolve over time, depending on the needs for the development and conceptualization of new business or technological elements.

The following figure shows a representation of the concepts handled by the Domain Driven Design approach. The entities located at the bottom are those necessary to analyse the domain as a whole, while those at the top of the graph are related to the more technical part of software architecture. Therefore, Domain Driven Design is an "approach to the design of software solutions that ranges from the most holistic definition to the implementation of the objects in the code".



2.2. API First

API First strategy consists of considering APIs and everything related to them (definition, versioning, development, promotion...) as a product.

This design strategy considers the API as the main interface of the application and initially begins with its design and documentation, to later develop the backend part, instead of setting up the entire backend first and then adapting the API to everything built.

In this way, the technological infrastructure depends directly on the design of the services instead of being a response to their implementation.

Among the main benefits of an "API First" development strategy we can highlight:

- Development teams can work in parallel.
- Reduces the cost of application development.
- Increases speed to market.
- Ensures good developer experiences.

2.3. Interface standardization. Standardization organisms.

In order to ensure the reusability of the different integrations between elements and systems is essential the agreement of the industry (network element providers, system providers, customer service providers,...) defining a series of interfaces that ensure specific functionality and responses.

There are different organizations, standardization forums and collaboration projects that define specific interfaces for certain domains, which are then implemented by different industry agents: integrators, software manufacturers, etc... Some of these organizations, specialized in network domains, are: GSMA, 3GPP, IETF, ONF, Broadband Forum, among others.

At the systems level, the reference organization is, without a doubt, the Telemanagement Forum (TM Forum). TM Forum is a global association that drives collaboration and collective problem solving to maximize the business success of telecommunications companies and their provider ecosystem. Its purpose is to help this ecosystem to transform and prosper in the digital age.

TM Forum Framework is a set of best practices and standards for evaluating and optimizing process performance, using a service approach to operations. The tools available in Framework help to improve end-to-end service management in complex, multi-stakeholder environments. It has been widely adopted by the telecommunications industry, providing a common language of processes, functional capabilities, and information.

Framework is composed of:

- Business Process Framework (eTOM) (enhanced Telecom Operations Map): Reference framework for defining the business processes of telecommunications operators.
- Application Framework Fundamentals (TAM): Grouping functionalities into recognizable applications to automate processes that will help us to plan and optimize an application portfolio.
- The Information Framework (formally Shared Information/Data Model or SID): Unified reference data model that provides a single set of terms for business objects in telecommunications. The goal is to enable people from different departments, companies, or geographic locations to use the same terms to describe the same real-world objects, practices, and relationships.

Over the last years, TMForum is performing a complete transformation of its architecture, moving from Framework, whose paradigm is based on applications, to the Open Digital Architecture (ODA), based on modular components.

TM Forum further defines a set of reference APIs (TMF Open APIs) between the different architecture components.

2.4. Information Representation Standard

As a messaging standard, the use of JSON is proposed by default, since it is a light data exchange format and is commonly adopted by current web technologies, although this does not imply that other types of data cannot be used depending on functional and technical requirements.

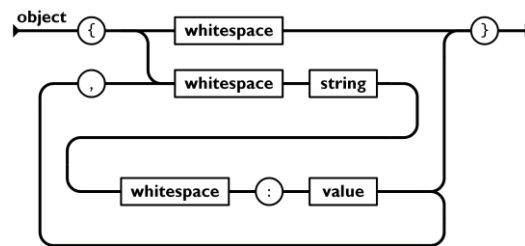
JSON is made up with two structures:

- A collection of key/value pairs. In various languages this is known as an object, record, structure, dictionary, hash table, key list, or an associative array.
- An ordered list of values. In most languages, this is implemented as arrays, vectors, lists, or sequences.

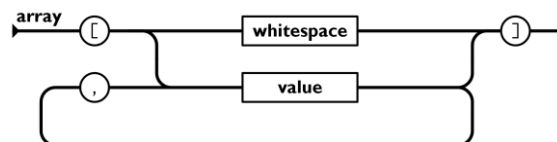
Data exchange format that is independent of the programming language is based on universal structures supported virtually in all programming languages.

In JSON, these structures are represented:

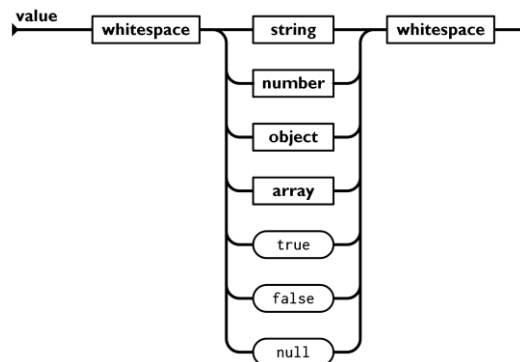
An object is an unordered set of key/value pairs. An object starts with "{" opening brace and ends with "}" closing brace. Each name is followed by a ":" colon and key/value pairs are separated by a "," comma.



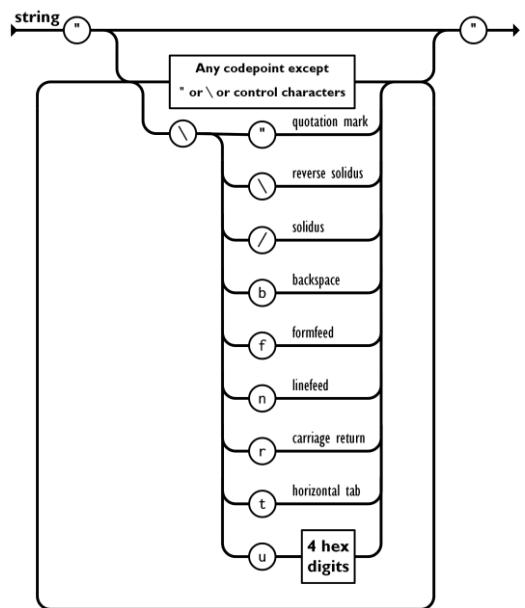
An array is a collection of values. An array starts with “[” left bracket and ends with “]” right bracket. Values are separated by “,” comma.



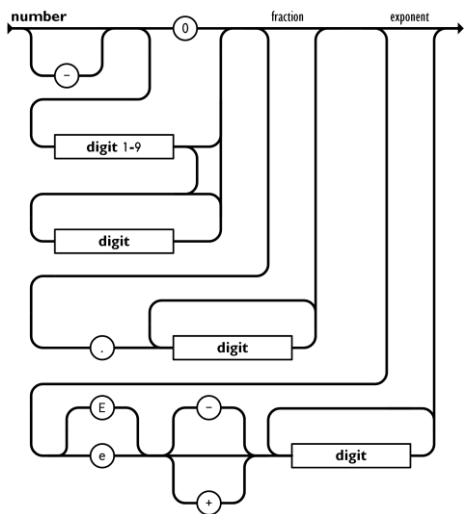
A value can be a string with double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.



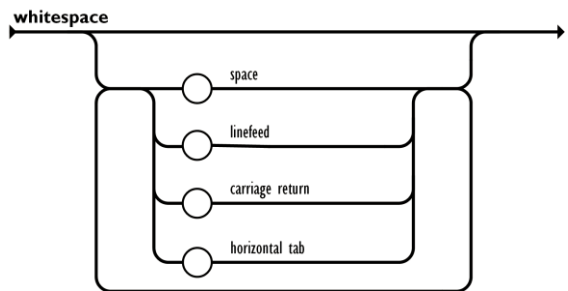
A character string is a collection of zero or more Unicode characters, enclosed in double quotes, using backslash escaping. A character is represented by a character string of a single character. A character string is similar to a C or Java character string.



A number is similar to a C or Java number, except that octal and hexadecimal formats are not used.



White spaces can be inserted between any pair of symbols.



Except for minor encoding details, this fully describes the language.

3. API Definition

API definition aims to capture 100% of the functional, syntax and semantic documentation of a contract. It is a critical element for the business strategy and will tag the product offered success.

Outstanding aspects in the API definition are:

- API must be defined from a product view focused to API consumers. It means, it should be prepared a top-down definition of all APIs.
- Product should be focused on the usability.
- API contract must be correctly documented, and it will be published in a shared catalog
- API exposure definition will be based on “resources could be used by different services and product consumers”. That means, API definition should be a 360° product view.

API definition must consider the next assertions:

- API resources definition contains the URI and enterprise resources.
- Rest specification compliance aims make the API fully interoperable.
- API resource security.
- Product consumption experience.
- API measurement over how many, who and when it is used.
- API must be versioned, and version must figure in all operations URL. It allows a better last develops management and surveillance, it avoids out of date URLs request and it makes available coexistence of more than one version productive in the same environment.
- Expose the required fields by the API consumers only. Sometimes, all the response body is not necessary for consumers. Allows more security, less the network traffic and agile the API usability.
- One of the API Quality requirements will be the evolution and management scalability, lined with versioning and backward compatibility considerations detailed in this document.
- At API definition time is necessary include audit parameters to allow make surveillance and next maintenances.
- English use must be applied in the OpenAPI definition.

3.1. API REST

Representational state transfer (REST) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave.

The formal REST constraints are as follows:

- Client–server architecture: The client-server design pattern enforces the principle of separation of concerns: separating the user interface concerns from the data storage

concerns. Portability of the user interface is thus improved. In the case of the Web, a plethora of web browsers have been developed for most platforms without the need for knowledge of any server implementations. Separation also simplifies the server components, improving scalability, but more importantly it allows components to evolve independently (anarchic scalability), which is necessary in an Internet-scale environment that involves multiple organisational domains.

- **Statelessness:** In computing, a stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood in isolation, without context information from previous packets in the session. This property of stateless protocols makes them ideal in high volume applications, increasing performance by removing server load caused by retention of session information.
- **Cacheability:** As on the World Wide Web, clients and intermediaries can cache responses. Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from providing stale or inappropriate data in response to further requests. Well-managed caching partially or eliminates some client-server interactions, further improving scalability and performance.
- **Layered system:** A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. If a proxy or load balancer is placed between the client and server, it won't affect their communications, and there won't be a need to update the client or server code.
- **Uniform interface:** The uniform interface constraint is fundamental to the design of any RESTful system. It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:
 - **Resource identification in requests** - Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server could send data from its database as HTML, XML or as JSON—none of which are the server's internal representation.
 - **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's state.
 - **Self-descriptive messages** - Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.

- Hypermedia as the engine of application state (HATEOAS) - Having accessed an initial URI for the REST application—analogous to a human Web user 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 resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application.

HTTP Verbs

HTTP standard should be followed to the verbs usability, resulting an API definition oriented to the RESTful design. Any HTTP correct verb usability are allowed.

Principal methods (Verbs) available to use are next ones:

Verbo HTTP	Operación CRUD	Descripción
GET	Read	Retrieve the actual resource status.
POST	Create	Creates a new resource in the collection. Returns the resource URL when the creation ends.
PUT	Update	Replaces a specific resource. Returns the resource URL when the replace ends.
DELETE	Delete	Delete a specific resource.
PATCH	Update	Updates a specific resource, applying all changes at the same time. If resource does not exist, it will be created. Returns the resource URL when the update ends. If any error occurs during the update, all of them will be cancelled.
OPTIONS	Read	Returns a 200 OK with an allowed methods list in the specific resource destined to the header allowed joined to an HTML document about the resource + an API Doc link.
HEAD	Read	Returns the resource actual status without message body.

In this document will be defined the principal verbs to use in the API definition.

- POST: Used to send data to the server.
- GET: This operation allows realize all the read and retrieve operations. GET operation should be based on data retrieving, that's why the retrieve operation must be realized directly from the URI including some identifiers and query params.
- PUT: Allows update entity data, deleting one or more fields from this entity body if there are not informed. New information to update must be informed by JSON language sent in the body request. If operation requires extra information, Query params could be used. PUT case, if registry does not exist in server data storage, it will be created, that means this operation could be used to create new resources.

- DELETE: Allows delete full entities from server. From consumer perspective, it is not a reversible action. (Rollback action is not available).
- PATCH: Allows partial fields update of a resource.

3.2. HTTP Response Codes

HTTP status response codes indicate if a request has been completed successfully. Response codes are group by five classes.

1. Inform responses (1XX)
2. Success responses (2XX)
3. Redirections (3XX)
4. Client Errors (4XX)
5. Server Errors (5XX)

Status codes has been defined in the [RFC 2616](#) 10th section. You can get the updated specifications from [RFC 7231](#).

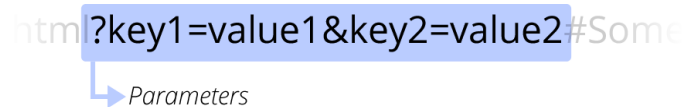
Common errors are indicated below:

Código	Descripción
200	200 (OK) status code indicates that the request result successfully. GET -> 200 HTTP Code by default. POST/PUT/PATCH -> Resource update actions, data is returned in a body from server side. DELETE -> Resource delete action, data is returned in a body from server side.
201	201 (Created) HTTP code indicates that the request has created one or more resource successfully. POST/PUT -> When a resource is created successfully.
202	202 (Accepted) code indicated that the request has been accepted to be processed, but it has not ended. Usually, when a DELETE is requested but the server cannot make the action immediately. It should applies to async processes.
203	203 (Unathourized information) code indicated that the request has been successfully, but the attached payload was modified from the 200 (OK) response from the origin server using a transformation proxy. It is used when data sent in the request could be modified as a third data subset.
204	204 (No Content) indicated that the server has ended successfully the request and there are nothing to return in the body response. POST -> When it's used to modify a resource and output is not returned. PUT/PATCH -> When it modify a resource and output is not returned. DELETE -> Resource delete action and output is not returned. PD: This code is recommended to be used in DELETE operations.

400	400 (Bad Request) status code indicates that the server cannot or will not process the request due to something perceived as a client error (for example, malformed request syntax, invalid request message structure, or incorrect routing). This code must be documented in all the operations in which it is necessary to receive data in the request.
401	401 (Unauthorized) status code indicates that the request has not been applied because it lacks valid authentication credentials for the target resource. This code has to be documented in all API operations that require subscription by a client.
403	403 (Forbidden) status code indicates that the server understood the request, but is refusing to authorize it. A server that wants to make public why the request was prohibited can describe that reason in the response payload (if applicable). This code is usually documented in the operations. It will be returned when the OAuth token access is invalid or when the user fails operational security.
404	404 (Not Found) status code indicates that the origin server either did not find a current representation for the target resource or is unwilling to reveal that it exists. This code will occur on GET operations when the resource is not available, so it is necessary to document this return in such situations.
405	405 (Method Not Allowed) status code indicates that the origin server knows about the method received in the request line, but the target resource does not support it. This code is documented at the API portal level, it should not be documented at the API level.
406	406 (Not Acceptable) status code indicates that the target resource does not have a current representation that would be acceptable to the user, based on the proactive negotiation header fields received in the request, and the server is unwilling to provide a predetermined representation. It must be reported when there is no response by default, and header fields are reported to carry out the content negotiation (Accept, Accept-Charset, Accept-Encoding, Accept-Language).
408	Status code 408 (Request Timeout) indicates that the server did not receive the complete request message within the expected time. This code is documented at the API portal level, it should not be documented at the API level.
409	The 409 (Conflict) status code indicates when a request conflicts with the current state of the server.
500	Status code 500 (Internal Server Error) indicates that the server encountered an unexpected condition that prevented it from fulfilling the request. This code must always be documented. It should be used as a general system error.
501	Status code 501 (Not Implemented) indicates that the requested method is not supported by the server and cannot be handled. The only methods that servers require support (and therefore should not return this code) are GET and HEAD.
502	Status code 502 (Bad Gateway) indicates that the server, while working as a gateway to get a response needed to handle the request, got an invalid response.
504	Status code 504 (Gateway Timeout) indicates that the server is acting as a gateway and cannot get a response in time.

3.3. Query Parameters Use

API query parameters can be defined as key-value pairs that appear after the question mark in the URL. Basically, they are URL extensions used to help determine the specific content or action based on the data being delivered. Query parameters are added at the end of the URL, using a "?". The question mark is used to separate the path and the query parameters.



If you want to add multiple query parameters, an '&' is placed between them to form a query string. You can present lot of objects types with different lengths, such as arrays, strings, and numbers.

3.4. Path Parameters use

A path param is a unique identifier for the resource. For example:

- /users/{user-id}

Multiple path params can be entered if there is a logical path of mutually dependent resources.

- /users/{user-id}/documents/{document-id}

3.4.1. Good Practices

1. Path params cannot be concatenated. A path param must be preceded by the resource represented. If we did this the URL would be incomprehensible:
 - /users/{user-id}/{document-id}
 - /users/13225365/647658
2. The attribute must be identifying itself, it is not enough with "{id}"
 - /users/{id}

Reason is that if this resource is "extended" in the future and includes other identifiers, we would not know which of the entities the {id} parameter refers to. For example:

- Incorrect:
 - i. /users/{id}/documents/{document-id}
 - Correct:
 - i. /users/{user-id}/documents/{document-id}
3. It is recommended that the identifier have a similar morphology on all endpoints. For example, "xxx-id", where xxx is the name of the entity it references:
 - /users/{user-id}
 - /accounts/{account-id}
 - /vehicles/{vehicle-id}
 - /users/{user-id}/vehicles/{vehicle-id}
 4. Care must be taken not to create ambiguities in the URIs when defining paths. For example, if the "user" entity can be identified by two unique identifiers and we will create two URIs
 - /users/{user-id}
 - /users/{nif}

5. Identifiers must be, as far as possible, of a hash type or similar so that we avoid enumeration or brute force attacks for their deduction.

Upon API invocation, one of the options would be chosen and we would not be able to distinguish which one was chosen.

3.5. HTTP Headers definition

Request header parameters are a great addition to the design of our API functionality. The purpose of this document is not to describe all the possibilities offered by the HTTP protocol, but to try to take the use of HTTP headers into account during the definition and design of APIs, in order to improve their characteristics.

The main HTTP headers are described below:

- **Accept:** The "Accept" header can be used to specify certain types of data that are acceptable for the response. Accept headers can be used to indicate that the request is specifically limited to a small set of desired types, as in the case of a request for an image.
- **Accept-Encoding:** This field is like 'accept' header but restricts the content encodings that are acceptable in the response.
- **Accept-Language:** Consumer defines the list of languages in order of preference. The server answer with the "Content-Language" field in the header with the response language.
- **Authorization:** Field to send the authorization token for API access, initially OAuth and JWT.
- **Content-Type:** This field indicates the type of message sent to the recipient or, in the case of the HEAD method, the type of message that would have been sent if the request had been a GET. The MIME type of the response, or the content uploaded via POST/PUT in case it is a request.
- **Content-Length:** The content length indicates the message size, in octets, sent to the recipient or, in the case of the HEAD method, the message size that would have been sent if the request had been a GET. The size of the response in octets (8 bits)
- **Content-Encoding:** The "Content-Encoding" field is used as a message type modifier. The type of encoding used in the response is indicated.

Optional recommended security headers by OWASP:

- **HTTP Strict Transport Security:** web security policy mechanism which helps to protect websites against protocol downgrade attacks and cookie hijacking. It allows web servers to declare that web browsers (or other complying user agents) should only interact with it using secure HTTPS connections, and never via the insecure HTTP protocol.
- **X-Frame-Options:** response header (also named XFO) improves the protection of web applications against clickjacking. It instructs the browser whether the content can be displayed within frames.
- **X-Content-Type-Options:** Setting this header will prevent the browser from interpreting files as a different MIME type to what is specified in the Content-Type HTTP header (e.g. treating text/plain as text/css).

- **Content-Security-Policy:** requires careful tuning and precise definition of the policy. If enabled, CSP has significant impact on the way browsers render pages (e.g., inline JavaScript is disabled by default and must be explicitly allowed in the policy). CSP prevents a wide range of attacks, including cross-site scripting and other cross-site injections.
- **X-Permitted-Cross-Domain-Policies:** A cross-domain policy file is an XML document that grants a web client, such as Adobe Flash Player or Adobe Acrobat (though not necessarily limited to these), permission to handle data across domains. When clients request content hosted on a particular source domain and that content makes requests directed towards a domain other than its own, the remote domain needs to host a cross-domain policy file that grants access to the source domain, allowing the client to continue the transaction.
- **Referrer-Policy:** governs which referrer information, sent in the Referer header, should be included with requests made.
- **Clear-Site-Data:** clears browsing data (cookies, storage, cache) associated with the requesting website. It allows web developers to have more control over the data stored locally by a browser for their origins.
- **Cross-Origin-Embedder-Policy:** prevents a document from loading any cross-origin resources that don't explicitly grant the document permission
- **Cross-Origin-Opener-Policy:** allows you to ensure a top-level document does not share a browsing context group with cross-origin documents. COOP will process-isolate your document and potential attackers can't access to your global object if they were opening it in a popup, preventing a set of cross-origin attacks dubbed XS-Leaks
- **Cross-Origin-Resource-Policy:** response header (also named CORP) allows to define a policy that lets web sites and applications opt in to protection against certain requests from other origins (such as those issued with elements like <script> and), to mitigate speculative side-channel attacks, like Spectre, as well as Cross-Site Script Inclusion (XSSI) attacks
- **Cache-Control:** holds directives (instructions) for caching in both requests and responses. If a given directive is in a request, it does not mean this directive is in the response

3.5.1. Not allowed HTTP headers

- **Server:** This header offers information, it is recommended disable all the relevant information about the version and services raised on the server side to avoid disclosing information about the server that provides the services.
- **X-Powered-By:** This header describes the technology used to implement the exposed service. This information can be useful to potential attackers and should be avoided.
- **X-Frame-Options:** This header is generally used to prevent clickjacking attacks, but there is a more standard header to do this called "Content-Security-Policy". This header is no longer needed.
- **X-UA-Compatible:** This header was introduced by Microsoft to provide compatibility with legacy versions of IE (IE8, IE7...). APIs are not considered to have a web interface, so this header is not useful.
- **Expires:** Because the "Cache-Control" header can offer an expiration date/time for cached values, this header is no longer needed and should be avoided.

- **Pragma:** The "Cache-Control" header will do the same job as "Pragma" too, it is more standard, so should be avoided.

3.6. MIME Types

During the API definition process, API MIME types must be identified, explaining how the data will be sent to the resource and how the resource will return it to the consumption.

Due to interoperability reasons and in order to comply as closely as possible with REST, it is recommended to use standardized mime-types and avoid creating new mime-types.

The standard headers that allow managing the data format are:

- Accept
- Content-Type

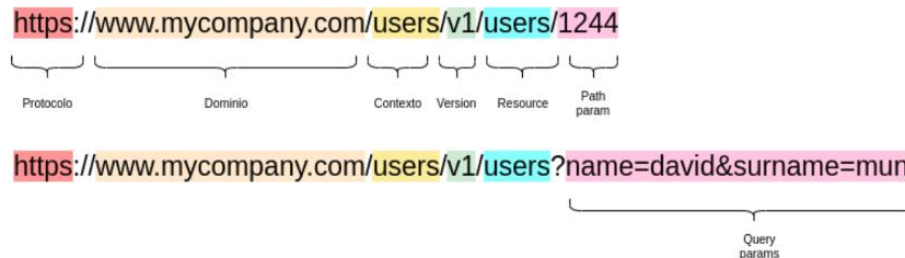
As a MIME Types example we can identify:

- application/xml
- application/json

4. API resources definition

4.1. URL Definition

The figure below shows an example of URL definition:



As seen, the full URL consists of:

1. Protocol: Protocol on which it works. We will always use HTTPS.
2. Domain: Our machine name or domain. It will be defined at the server level.
3. Context: The name of the API. Our system may have several differentiated APIs according to its objectives and the relationship of its resources.
4. Version: MAJOR version. Part of semantic versioning.
5. Resource: Specific resource that we are accessing. It can be made up of several levels.
6. Path param: Part of the resource identifier that precedes it. Indicates that it is the user unequivocally identified by "1244".
7. Query Param: Resource filter parameters. They are preceded by "?" and as many as defined with "&" can be concatenated.

4.1.1. Good practices

URIs should be designed according to the following considerations:

- URI with lowercase and hyphens. URIs must be "human readable" to facilitate identification of the offered resources. Lowercase words and hyphenation (kebab-case) help achieve this best practice. For example: `/customer-segments`
- URIs must contain the exposed resource.
- Verbs use is not allowed.
- URIs must contain the "major version" of the API.
- The resource chain will be defined in the API URI following a hierarchical relationship.
 - `<Resource1>/<id>/<Resource2>/<id>`
 - A collection must always be followed by a member of the collection.
 - The API response should return, if possible, the hypermedia resource which it refers to. Although it is recommended that the REST API be more pragmatic to return in a single call what is necessary to avoid concatenated calls from clients.
 - Short URIs use is recommended for relevant entities. Failure to define a clear hierarchical model will lead to inconsistencies in publishing interfaces through the API and will make it difficult to consume.
- Avoiding usage of the package type nomenclature (e.g. `"com.mycompany.api..."`) in the API URI, because it makes it difficult for the developer or consumer to use the API.

- URIs are defined per entity based on CRUD operations. Generally, we should only have one operation verb per functional entity (GET, PUT, POST, PATCH, DELETE).
- The URI at the business entity level will always be a plural noun.
- OperationIds are defined in lowerCamelCase: For example: helloWorld
- Objects are defined in CamelCase inside properties field. For example: Greetings, ExampleObject.

4.1.2. Hierarchy

Hierarchy could be introduced with the concepts of entity and sub-entity:

- **Entity:** It is understood as a enough relevant business object to be identified as a product. A single entity is defined by an API.
- **Sub-entity:** It is understood as a business object that by itself has no business relevance. It is an object that is hierarchically related to an entity.

To make the hierarchy, the following aspects must be applied:

- Two levels of hierarchy should not exceed and should not be more than 8 resources (6-8).
- A resource has multiple operations identified by HTTP Verbs.
- Resources defined through URIs establish a hierarchical relationship with each other:
 - /<entity>
 - /<entity>/<entity_id>
 - /<entity>/<entity_id>/<sub_entity>
 - /<entity>/<entity_id>/<sub_entity>/<sub_entity_id>

4.2. Input/Output resources definition

At this point, some considerations are outlined about the business input and output data of the API resources. This data can be informed by different means: QueryString, Header, Body...

These considerations are below:

- API input and output business data will follow the lowerCamelCase notation.
- The field names in JSON and XML will follow the same URIs standard.
 - Business data must be human readable.
 - commercial data name must be a noun, that means, it corresponds to an entity information.
- Business data sent as part of the HTTP protocol will be exempt from these rules. In these cases, compliance with the standard protocol will apply.
- Sensitive data (considered this way for security) must go in the body if it is a POST request and in the header if it is a GET, encrypted by default by the HTTP protocol.
- Description of the input and output data must have:
 - Functional description
 - Data Type
 - Value range supported

5. Versioning

5.1. Versioning Strategy

Service versioning is a practice by which, when a change occurs in the API of a service, a new version of that service is released so that the new version and the previous one coexist for a period of time.

Consumers will be migrated to the new version of the service sequentially. When everyone is consuming the latest version of the service, the previous version is removed.

Consumers can distinguish between one version of the service and another, the technique of adding the API version to the context of the base URL will be used, since this technique is the most used in the main reference APIs.

The structure of the URL would have the following form:

`https://host:port/api/v1/resource`

When we version through the URL, only the "MAJOR version" is included since this would change when a change incompatible with the previous version occurs.

API implementation versioning will follow semantic versioning. Given a MAJOR.MINOR.PATCH version number, it increments:

- 1) The MAJOR version when you make an incompatible API change.
- 2) The MINOR version when you add functionality that is backwards compatible.
- 3) The PATCH version when you fix backward compatible bugs.

Related to the versioning of rest parts involved in Apification projects, best practises are detailed below:

SHARED CODE ON REPOSITORIES

1. MAJOR - Major of API Contract
2. MINOR - Minor of API Contract
3. PATCH - New Updates / Contributions of shared code

MICROSERVICE DEPLOYMENTS (NOT MANDATORY BUT RECOMMENDED)

1. MAJOR - Major of API Contract
2. MINOR - Minor of API Contract
3. PATCH - New Microservice Deployments

5.2. Backward and forward compatibility

Avoid breaking backwards compatibility unless strictly necessary, that means, new versions should be compatible with previous versions.

Bearing in mind that APIs are continually evolving and certain operations will no longer be supported, the following considerations must be taken into account:

- Agree to discontinue an API with consumers.
- Establish the obsolescence of the API in a reasonable period of time (6 months).
- Monitor the use of deprecated APIs.
- Remove deprecated APIs documentation.

- Never start using already deprecated APIs.

5.2.1. Types of modifications

Not all API changes have an impact on API consumers. These changes are often referred to as backward compatible changes. If API undergoes changes of this type, it should not be necessary to release a new version, it would suffice to replace the current one. What would be very convenient is to notify our consumers with the new changes so that they take them into account.

This is a list of changes to an API that DO NOT affect consumers:

- Add new operations to the service. Translated to REST, it would be to add new actions on a resource (PUT, POST...)
- Add optional input parameters to requests on existing resources. Ex: a new filter parameter in a GET on a collection of resources.
- Modify input parameters from required to optional. Eg: when creating a resource, a property of said resource that was previously mandatory becomes optional.
- Add new properties in the representation of a resource returned by the server. Eg: now to a Person resource that was previously made up of DNI and name, we add a new age field.

This other list shows changes that DO affect consumers:

- Delete operations or actions on a resource. For example: POST requests on a resource are no longer accepted.
- Add new mandatory input parameters. For example: Now, to register a resource, a new required field must be sent in the body of the request.
- Modify input parameters from optional to mandatory. For example: now when creating a Person resource, the age field, which was previously optional, is now mandatory.
- Modify a parameter in existing operations (resource verbs). Also applicable to parameter removal. For example, when consulting a resource, a certain field is no longer returned. Another example: a field that was previously a string is now numeric.
- Add new responses to existing operations. For example: creating a resource can return a 412 response code.

5.2.2. Compatibility Management

To ensure this compatibility, the following rules must be followed:

As API provider

- New fields should always be added as optional.
- Postel's Law: "Be conservative in what you do, be liberal in what you accept from others". When you have input fields that need to be removed, mark them as unused so they can be ignored.
- Do not change the field's semantics.
- Do not change the field's order.
- Do not change the validation rules of the request fields to more restrictive ones.
- If you use collections that can be returned with no content, then answer with an empty collection and not null.

- Layout pagination support from the start.

As API Consumer

- **Tolerant reader:** if it does not recognize a field when faced with a response from a service, do not process it, but record it through the log (or resend it if applicable).
- Ignore fields with null values.
- **Variable order rule:** DO NOT rely on the order in which data appears in responses from the JSON service, unless the service explicitly specifies it.
- Clients MUST NOT transmit personally identifiable information (PII) parameters in the URL. If necessary, use headers.

6. Error Responses

In order to guarantee interoperability, one of the most important points is to carry out error management aimed at strictly complying with the error codes defined in the HTTP protocol.

An error representation must not be different from the representation of any resource.

Errors should be focused on environment where there are going to be raised, that's why there are two different standard error definitions.

6.1. Internal Errors

Internal errors definitions focused on gateway previous side, allowing interoperability, observability and granularity based on error codes.

Internal Error Codes Definition

6.2. External Errors

External errors definitions focused on gateway back side, allowing consumers usability, understandability and final users showability.

External Error Codes Definition

6.3. Errors Mapping

Correlation between internal and external error must be strongly defined, that's why a mapping table between both worlds has been created and grow maintenance.

Error Codes Mapping Table

7. Common Data Types (AddressType structure, E164Type, EmailAddressType, ExceptionType structure,...) [Open Title to be updated over the time]

The aim of this clause is to detail standard data types that will be used over time in all definitions, as long as they satisfy the information that must be covered.

It should be noted that this point is open to continuous evolution over time through the addition of possible new data structures. To allow for a proper management of this ever-evolving list, an external repository has been defined to that end.

This repository is referenced below:

[API Desing Common Data Types](#)

8. Filtering, sorting and pagination

Exposing a resources collection through a single URI can cause applications to fetch large amounts of data when only a subset of the information is required. For example, suppose a client application needs to find all orders with a cost greater than a specific value. You could retrieve all orders from the /orders URI and then filter these orders on the client side. Clearly, this process is highly inefficient. It wastes network bandwidth and processing power on the server hosting the web API.

8.1. Pagination

Services can answer with a resource or article collections. Sometimes these collections may be a partial set due to performance or security reasons. Elements must be identified and arranged consistently on all pages. Paging can be enabled by default on the server side to mitigate denial of service or similar.

Services must accept and use these query headers when paging is supported:

- per_page: number of resources requested to be provided in the response
- page: requested index to indicate the start of the resources to be provided in the response
- seek: index of last result read to create the next/previous number of results. This query parameter is used for pagination in systems with more than 1000 records.

Services must accept and use these headers when paging is supported:

- Content-Last-Key: it allows specifying the key of the last resort provided in the response
- X-Total-Count: it allows indicating the total number of elements in the collection

If the server cannot meet any of the required parameters, it should return an error message.

The HTTP codes that the server will use as a response are:

- 200: response includes the complete list of resources

- 206: the response does not include the complete list of resources
- 400: request outside the range of the resource list

Petitions examples:

- page=0 per_page=20, which returns the first 20 resources
- page=10 per_page=20, which returns 20 resources from the 10th element

8.2. Sorting

Sorting the result of a query on a resources collection requires two main parameters:

- order_by: it contains the names of the attributes on which the sort is performed, with comma separated if there is more than one criteria.
- order: by default, sorting is done in descending order.

If you may want to specify which sort criteria you need to use "asc" or "desc" as query value.

For example: The list of orders is retrieved, sorted by rating, reviews and name with descending criteria.

https://api.mycompany.com/v1/orders?order_by=rating,reviews,name&order=asc

8.3. Filtering

Filtering consists of restricting the number of resources queried by specifying some attributes and their expected values. It is possible to filter a collection on multiple attributes at the same time and allow multiple values for a filtered attribute.

Next, it is specified how it should be used according to the filtering based on the type of data being searched for: a text, a number or a date and the type of operation:

ration.

Operation	Text	Numbers	Dates
Equal	GET .../?name=Juan	GET .../?amount=807.24	GET .../?executionDate=2018-30-05
Greater or equal	N/A	GET .../?amount.gte=807.24	GET.../?execution_date.gte=2018-30-05
Strictly greater	N/A	GET .../?amount.gt=807.24	GET.../?execution_date.gt=2018-30-05
smaller or equal	N/A	GET .../?amount.lte=807.24	GET.../?execution_date.lte=2018-30-05
Strictly smaller	N/A	GET .../?amount.lt=807.24	GET.../?execution_date.lt=2018-30-05

Contains	GET .../?name =~Juan	N/A	N/A
-----------------	----------------------------	-----	-----

Additional rules:

- The operator & is evaluated as an AND between different attributes.
- A Query Param (attribute) can contain 1 or n values separated by ','.
- For operations on numeric, date or enumerated fields, the use of the suffixes .(gte|gt|lte|lt)\$ will be allowed, which will act as comparators for “greater - equal to, greater than, smaller - equal to, smaller than”.

Examples:

- Equals: to search users with first name "david" and last name "munoz":
 - GET /users?name=david&surname=munoz
 - GET /users?name=David,Noelia
 - Search for several values separating them by ",".
- Inclusion: if we already have a filter that searches for "equal" and we want to provide it with the possibility of searching for "inclusion", we must include the character "~"
 - GET /users?name=dav
 - Search for the exact name "dav"
 - GET /users?name=~dav
 - Look for names that include "dav"
- Greater than / less than: new attribute will be created and it will be preceded with the suffixes .(gte|gt|lte|lt)\$.
 - GET /users?creation_date.gte=2021-01-01T00:00:00
 - Find users with creation Date greater than 2021
 - GET /users?creation_date.gt=2021-11-31T23:59:59
 - Find users with creationDate less than 2022
 - GET /users?creation_date.gte=2020-01-01T00:00:00&creation_date.lte=2021-11-31T23:59:59
 - Search for users created between 2020 and 2021

9. Architecture Headers

With the aim of standardizing the request observability and traceability process, common headers that provide a follow-up of the E2E processes should be included. The table below captures these headers.

Name	Description	Type	Pattern	Required on Swagger	Required	Example
X-Version	Service version indentificator	String	N/A	No	No	
X-Correlator	Service correlator to make observability	String	UUID (8-4-4-4-12)	Yes	No	b4333c46-49c0-4f62-80d7-f0ef930f1c46
Authorization	Contains Access token	String	JWT	Yes	Yes	Bearer eyJhbGciOiJIUzI1NiIsIn...
Content-Language	Describe the language(s) intended for the audience	String	ISO 639-1	No	No	es
Accept-Language	Indicates the natural language and locale that the client prefers	String	N/A	No	No	
Service	Service indentificator	String	N/A	No	No	
Service Callback	This header indicates the endpoint that will receive a pending response if it is necessary.	String	N/A	Yes, if some async endpoint is present on contract	Yes, if some async endpoint is present on contract	
Application	Application identifier	String	N/A	No	No	
User identifier	Unique indentificator for the user	String	N/A	No	No	
Originator	Service identifier from the request originator	String	country:entity:system:subsystem	No	No	
Business-process	Business process where the service is located	String	N/A	No	No	
Country	Contains the country of the entity originating the call.	String	ISO 3166-1 alpha-2	No	No	ES

work-station	Origin requestor equipment identification	String	N/A	No	No	
--------------	---	--------	-----	----	----	--

10. Security

One of the key points in the API definition process is to specify and validate the security needs that will be maintained to guarantee data integrity and access control. There are multiple ways to secure a RESTful API, e.g. basic authentication, OAuth, etc., but one thing is for sure: RESTful APIs should be stateless, so authentication/authorization requests should not rely on cookies or sessions. Instead, each API request must come with some form of authentication credentials that must be validated on the server for each request.

Basic idea in terms of security is to understand that various types of data will require different levels of security, depending on the confidentiality of the data you are trying to obtain and the level of trust between Telefónica and the consumer.

10.1. APIs Rest Security

10.1.1. REST security design principles

REST security design principles The document "The Protection of Information in Computer Systems", by Jerome Saltzer and Michael Schroeder, outlines eight design principles to secure information in computer systems, which are listed and elaborated below:

- 1. Least privilege:** An entity should only have the set of permissions necessary to perform the actions for which it is authorized, and no more. Permissions can be added as needed and should be revoked when no longer in use.
- 2. Fail-safe defaults:** A user's default level of access to any system resource should be "denied" unless "permission" has been explicitly granted.
- 3. Mechanism economics:** The design should be as simple as possible. All component interfaces and interactions between them should be simple enough to understand.
- 4. Full mediation:** A system must validate access rights to all of its resources to ensure that they are allowed and must not rely on the cached permissions array. If the level of access to a certain resource is revoked, but that is not reflected in the permissions matrix, it would violate security.
- 5. Open Design:** This principle underlines the importance of building a system in an open way, without secret and confidential algorithms.
- 6. Separation of privileges:** Granting permissions to an entity should not be based purely on a single condition, a combination of conditions based on resource type is a better idea..
- 7. Least Common Mechanism:** It refers to the risk of sharing state between different components. If one can corrupt the shared state, then it can corrupt all other components that depend on it.
- 8. Psychological acceptance:** States that the security mechanisms must not make the resource more difficult to access than if the security mechanisms were not present. In short, security should not worsen the user experience. (restfulapi.net,2021)

10.1.2. Good practices to secure REST APIs

The following points can serve as a checklist to design the security mechanism of the REST APIs.

1. Simple Management

Securing only the APIs that need to be secure. Any time the more complex solution is made "unnecessarily", it is also likely to leave a hole.

2. HTTPs must be used always

By always using SSL, authentication credentials can be simplified to a randomly generated access token. The token is delivered in the username field of HTTP Basic Auth. It is relatively easy to use, and you get a lot of security features for free.

If HTTP 2 is used, to improve performance, you can even send multiple requests over a single connection, this way you will avoid the complete overhead of TCP and SSL on subsequent requests.

3. Using hash password

Passwords should never be sent in API bodies, but if it is necessary it must be hashed to protect the system (or minimize damage) even if it is compromised in some hacking attempts. There are many hashing algorithms that can be really effective for password security, for example PBKDF2, bcrypt and scrypt algorithms.

4. Information must not be exposed in the URLs

Username, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, making them easily exploitable. For example, this URL (*https://api.domain.com/user-management/users/{id}/someAction?apiKey=abcd123456789*) exposes the API key. Therefore, never use this kind of security.

5. OAuth should be considered

While basic authentication is good enough for most APIs and, if implemented correctly, also secure, you may want to consider OAuth as well. The OAuth 2.0 authorization framework allows a third-party application to gain limited access to an HTTP service, either on behalf of a resource owner, by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party app to get access on your behalf.

OpenID Connect is built on the OAuth 2.0 protocol and uses an additional JSON Web Token (JWT), called an ID token, to standardize areas that OAuth 2.0 leaves up to choice, such as scopes and endpoint discovery. It is specifically focused on user authentication and is widely used to enable user logins on consumer websites and mobile apps.

6. Add request time flags should be considered

Along with other request parameters, a request timestamp can be added as a custom HTTP header in API requests. The server will compare the current timestamp with the timestamp of the request and will only accept the request if it is within a reasonable time frame (1-2 minutes maybe).

This will prevent very basic replay attacks from people trying to hack your system without changing this timestamp.

7. Entry params validation

Validate the parameters of the request in the first step, before it reaches the application logic. Put strong validation checks and reject the request immediately if validation fails. In the API response, send relevant error messages and an example of the correct input format to improve the user experience.

(restfulapi.net, 2021)

10.2. Security Implementation

The security implemented in an API can be divided into three different layers: i) channel security; ii) access security; and iii) data security.

10.2.1. Channel security

The API must ensure that the channel where the consumer and the API will exchange information is secure.

10.2.1.1. TLS and mutual authentication

As for today, it is commonly agreed that all communications over the Internet must be done via secure HTTP (HTTPS) using Transport Layer Security (TLS) to generate a secure and recorded communication channel. In some cases, the TLS channel must be more strictly added a mutual authentication process to identify both actors.

Protect communications with TLS is mandatory for all APIs. Any API that accepts requests without TLS will not be published to the API Manager. The TLS version to use for APIs is TLS 1.2 (for compatibility) and TLS 1.3 (for security and because it is the latest version available). The API Manager should not accept requests made with some of the older versions of TLS.

In case the request does not use the proper TLS, the API should send an HTTP 403 (Forbidden) code with the corresponding response body.

10.2.1.2. Certificate Chain Validation

When establishing a TLS with mutual authentication, the API must validate that the consumer's certificate is validated by one of the allowed CAs.

In case the request does not use the proper TLS, the API should send an HTTP 401 (Unauthorized) code with the corresponding response body.

10.2.1.3. Recommended Ciphers

These include:

- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

10.2.2. Access Security

The API must ensure that the consumer is known and can access the requested resources.

10.2.2.1. OAuth

All APIs must be protected by the OAuth 2.0 Framework. All API requests must include an HTTP header called "Authorization" with a valid OAuth access token.

The following controls will be performed on the access token:

- Check if the access token is still valid. If it is expired or revoked, the API will return an HTTP 401 (Unauthorized) code.
- Check if the access token was created to be used by the requestor. Otherwise, the API will return an HTTP 401 (Unauthorized) code.
- Check the scope of the access token, if it is expired or revoked, the API will return an HTTP 403 (Forbidden) code.

10.2.2.2. Scopes

The scopes allow defining the permission scopes that a system or a user has on a resource, ensuring that they can only access the parts they need and not have access to more. These restrictions are done by limiting the permissions that are granted to OAuth tokens.

Scopes should be represented as:

- API Name: address-management, numbering-information...
- API Protected Resource: orders, billings...
- Grant-level: read, write...

To correctly define the scopes, when creating them, the following recommendations should be taken:

- **Appropriate granularity:** Scopes should be granular enough to match the types of resources and permissions you want to grant.
- Use a **common nomenclature for all resources:** Scopes must be descriptive names and that there is no conflict between the different resources.
- Use the **kebab-case nomenclature** to define API names, resources, and scope permissions.
- **Recommended Format:**
 - <API_name>-<Resource>-<Permission>

Example:

address-management-validations-write

Next, we illustrate an example on how to define series of scopes for a OpenAPI file.

The first step is to create the security definitions according to the nomenclature that we have defined.

```
securityDefinitions:
  three_legged:
    tokenUrl: https://auth.baikalplatform.com/token
    authorizationUrl: https://auth.baikalplatform.com/authorize
    scopes:
      explore-content-modules-user-read: No description available
      explore-content-modules-phone-number-read: No description available
      explore-content-send-feedback: No description available
    type: oauth2
    flow: accessCode
```

Then, each operation is assigned the necessary scope:

```
0 - /users/{user_id}/modules:
1 -   get:
2 -     security:
3 -       - three_legged:
4 -         - explore-content-modules-user-read
5 -     description: Get a list of modules for the UserId matching the query.
6 -     tags:

/users/{user_id}/cards/feedback:
  post:
    security:
      - three_legged:
        - explore-content-send-feedback
    description: >-
      Send feedback on impression and user actions on the cards of a given
      container. List of possible values for the outcome is `presented`
```

10.2.3. Data Security

The API must ensure that the information sent is as expected and has not been altered by possible attackers.

10.2.3.1. Headers Validation

An attacker can collect information from an API by sending malicious data via headers, so the API must verify that:

1. API must receive supported headers only.

```
HEADER.HACK: SELECT * FROM USERS;
Authorization: Bearer 7894df-ds8f7-sdf84-sdf878u
```

2. Supported headers must have values and longitudes agreed.

```
Authorization: INSERT INTO ...
```

To avoid this, validate that the "Authorization" header is JWT type and consists of three concatenated Base64url-encoded strings, separated by dots

If a header contains a malicious value or an unaccepted header is received, the API should send an HTTP 400 response.

10.2.3.2. Response body validation

A more common way to attack an API is to send incorrect values or malicious codes within the data sent in the request to obtain relevant information from internal services. Mainly, the APIs do not know if the transmitted data is valid or not, so in the end the malicious data can navigate to the backend and cause serious problems.

To avoid these undesirable situations, the APIs can carry out a previous control of the payload structure. These validations are performed using JSON definitions with a description of the JSON structure, fields, and value formats in this payload.

In case the payload does not follow these definitions, the API must send an HTTP 400 response.

10.2.3.3. Do not repudiate

Sometimes the attacker just wants to modify the payload values to change the behavior of the Customer Service Provider ecosystem to their advantage. For example, if you are running a payment, you can modify the payment payload to change the account id to your own account. This will not change either the structure or the value format of the payload.

In these situations, it is mandatory to require the consumer to send the payload in JWT format, signed with one of the allowed consumer signing certificates.

The API must validate the signature of the JWT in the payload following next requirements:

- The API must validate that the certificate used by the consumer is accepted by one of the allowed CAs (see Certificate Chain Validation section for a list of accepted CAs).
- Validate that the payload has not been modified during its transmission. Below options should be checked:
 - Encryption/decryption of the JWT signature using the appropriate consumer public key. The JWT is encoded in Base64.
 - Making sure that the decrypted and encrypted signature has the following String value ("JWT_Header.JWT_Payload")
 - Making sure that the JWT payload has the same structure and values as the decrypted part of "JWT_Signature".

11. Definition in Open API

API documentation helps customers integrate with the API by explaining what it is and how to use it. All APIs must include documentation for the developer who will consume your API.

API documentation usually consists of:

- Conceptual information to introduce clients to the API and the domain
- Practical information to help customers get involved with the API
- Reference information to inform customers of every detail of your API

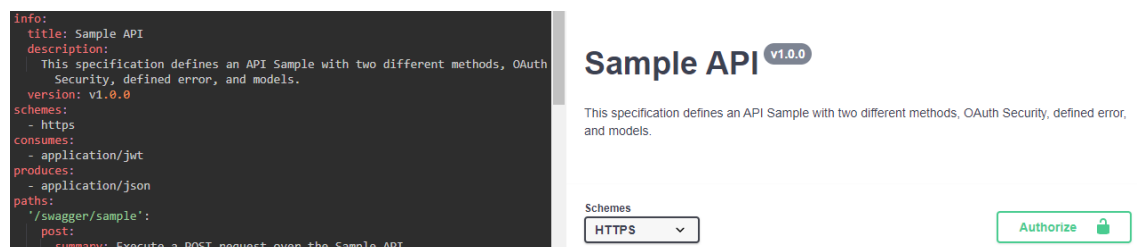
Below considerations should be checked when an API is documented:

- The API functionalities must be implemented following the specifications of the Open API latest version using the .yaml or .json file extension.
- The API specification structure should have the following parts:
 - General information (Section 11.1)
 - Published Routes (Section 11.2)
 - Request Parameters (Section 11.3)
 - Response Structure (Section 11.4)
 - Data Definitions (Section 11.5)
 - OAuth Definition (Section 11.6)

11.1 General Information

The “API General Info” part must contain the following information:

- API Version in the next format: vX.X.X.
- API title with public name.
- A brief description of the main functions of the API.
- API Terms of Service.
- Contact information, with name, email and website of the API Holder.
- License information (name, website...)
- Schemes supported (HTTP, HTTPS...)
- Allowed response formats (“application/json”, “text/xml”...)
- Response format (“application/json”...)



11.2 Published routes

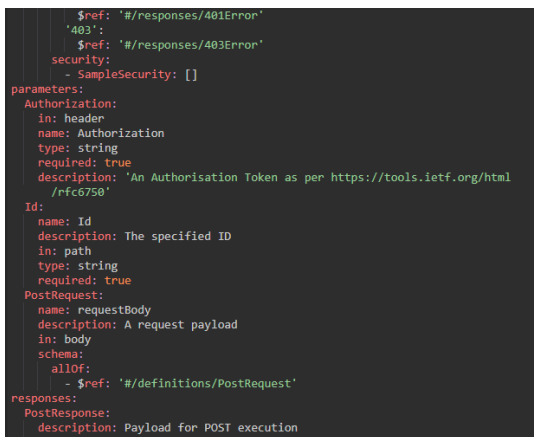
This part must contain the list of published functions, with the following description:

- URI functionality.
- HTTP Methods. For each one, the following shall be included:

- ```
paths:
 '/swagger/sample':
 post:
 summary: Execute a POST request over the Sample API
 description: Endpoint will be secured by way of Mutual Authentication
 over TLS.
 tags:
 - Sample POST
 parameters:
 - $ref: '#/parameters/PostRequest'
 responses:
 '201':
 $ref: '#/responses/PostResponse'
 '400':
 $ref: '#/responses/400Error'
 security:
 - SampleSecurity: []
 '/swagger/sample/Id':
 get:
 summary: Get information via ID
 description: Endpoint will be secured by way of Mutual Authentication
 over TLS.
 tags:
 - Sample GET
 parameters:
 - $ref: '#/parameters/Id'
 - $ref: '#/parameters/Authorization'
 responses:
 '200':
 $ref: '#/responses/GetResponse'
 '401':
 $ref: '#/responses/401Error'
```



- Parameter name, used to reference it in other sections
- Parameter description
- Parameter place (header, route...)
- Type (basic types like chains, integers, complex objects,...)
- Required field or optional flag



- Name of the response object, used to refer to it in other sections.
- Response object description.
- Object type (basic types like string, integer... or even more complex objects defined in the "Data definition" part...)

- Metadata links (HATEOAS)

The image displays an OpenAPI specification and its visual representation. On the left, the raw JSON specification is shown, defining response schemas and error codes. On the right, the Swagger UI visualizes this information, showing a table of responses with their status codes, descriptions, and example JSON payloads.

**OpenAPI Specification (Left Pane):**

```

responses:
 PostResponse:
 description: Payload for POST execution
 schema:
 allOf:
 - $ref: '#/definitions/PostResponse'
 GetResponse:
 description: Response from GET method
 schema:
 allOf:
 - $ref: '#/definitions/GetResponse'
 400Error:
 description: 'Request failed due to incorrect parameter.'
 schema:
 $ref: '#/definitions/Error'
 401Error:
 description: 'Request failed due to invalid access token.'
 403Error:
 description: 'The consumer does not have permission to access this API.'
 405Error:
 description: Method Not Allowed
 definitions:
 PostRequest:
 type: object
 required:
 - type
 properties:
 name:
 description: 'The short name.'
 type: string
 minLength: 0
 maxLength: 36
 description:

```

**Swagger UI (Right Pane):**

Responses | Response content type: application/json

| Code | Description                                |
|------|--------------------------------------------|
| 201  | Payload for POST execution                 |
| 400  | Request failed due to incorrect parameter. |

Example Value | Model

For 201:

```

{
 "id": "string",
 "name": "string",
 "description": "string",
 "type": 0,
 "enabled": true
}

```

For 400:

```

{
 "error_code": "name_too_long",
 "error_description": "string"
}

```

## 11.5 Data definitions

This part captures a detailed description of all the data structures used in the API specification. For each of these data, the specification must contain:

- Name of the data object, used to reference it in other sections.
- Data type (String, Integer, Object...).
- If the data type is an object, list of required properties.
- List of properties within the object data:
  - Property name
  - Property description
  - Property type (String, Integer, Object ...)
  - Other properties by type:
    - String ones: min and max longitude
    - Integer ones: Format (int32, int64...), min value.
- In this part, the error response structure must also be defined, which must be as follows:
  - Type (Array, Integer, Object ...)
  - Mandatory fields of the structure
  - Properties:
    - Error Code
      - Type (Array, Integer...)
      - Error codes supported, as Enum list
    - Error description
      - Type (Array)
      - Min longitude
      - Max longitude

```

GetResponse:
 type: object
 properties:
 id:
 description: 'A unique identifier in format UUID v4.'
 type: string
 minLength: 1
 maxLength: 36
 name:
 description: 'The short name.'
 type: string
 minLength: 1
 maxLength: 36
 description:
 description: 'A brief description.'
 type: string
 minLength: 1
 maxLength: 500
 type:
 description: 'Code of type'
 type: integer
 format: int32
 minimum: 1
 enabled:
 type: boolean
 Error:
 type: object
 required:
 - error_code
 properties:
 error_code:
 type: string
 enum:
 - name_too_long
 - description_too_long

```

| Response from GET method                                                                                                                                      |                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| Example Value                                                                                                                                                 | Model                                                     |
| <pre> {   id: "A unique identifier in format UUID v4.",   name: "The short name.",   description: "A brief description.",   type: 1,   enabled: true } </pre> |                                                           |
| 401                                                                                                                                                           | Request failed due to invalid access token.               |
| 403                                                                                                                                                           | The consumer does not have permission to access this API. |

## 11.6 OAuth Definition

Finally, the “OAuth Definition” part describes the OAuth security applied to the API. This spec is for client testing purposes only, but there should be as similar as possible to the OAuth flows in your production environment. This definition has the following aspects:

- Security Type: oauth2, oauth...
- Security Flow (Depends of security type): implicit, password...
- Security Flow description applied (String)
- Endpoint token URL
- URL to endpoint authorization ( If flow is based on "authorizationCode").

