**ISO #####-#:####(X)**

ISO TC 68/SC 09/WG 02

WD – 14/11/2018

Secretariat: tom.dunbar@bankofengland.co.uk

**Technical Specification for Web Service Based Application Programming Interfaces in Financial Services (WAPI)**

WD stage

**Warning for WDs and CDs**

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

© ISO 2018, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO’s member body in the country of the requester.

ISO copyright office

Ch. de Blandonnet 8 • CP 401

CH-1214 Vernier, Geneva, Switzerland

Tel. + 41 22 749 01 11

Fax + 41 22 749 09 47

copyright@iso.org

www.iso.org

**Table of Contents**

[To be completed once document is finished]

# Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see [www.iso.org/directives](https://www.iso.org/directives-and-policies.html)).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see [www.iso.org/patents](https://www.iso.org/patents)).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: [www.iso.org/iso/foreword.html](https://www.iso.org/iso/foreword.html).

This document was prepared by Technical Committee ISO/TC 68, Financial Services, Subcommittee SC 09.

# Introduction

Opening comments

The purpose of this Technical Specification is to help implementers in the financial services industry define the framework, function and protocols for an Application Programming Interfaces (API) ecosystem; enabling online synchronised interactions. It documents an international view of an API ecosystem in response to an urgent and significant world-wide demand for guidance and standardisation of APIs in Financial Services.

This has been driven by a number of emerging requirements – both market, corporate and regulatory driven - from different communities and jurisdictions for financial institutions to share data and enable functionality: between third parties acting on behalf of the customer, client or end user; between business to business processes; and within internal processes. It has been widely agreed that standardised APIs provide the most secure, developer friendly, efficient, technically proven way of meeting many of these requirements. Moreover, it is understood that a standardised approach would unlock benefits conducive to promoting interoperability, enhancing security, and supporting innovation. The sharing of data, and the subsequent use of APIs, is not limited to exchanges referenced in this document.

Despite these emerging requirements, there currently is no standardised approach at an international level. Moreover, there is no informative documentation covering the various considerations for developing APIs in financial services, especially given the maturity of some of its components (e.g.: some are in draft). This Technical Specification has been developed in response to meet these current and foreseen requirements that exist in the market. This document will not specify specific implementations of APIs, but instead takes an international view and references as appropriate, specific implementation scenarios for illustrative purposes for guidance.

How to approach this document

This document should be approached as an informative guide / best practice framework to developing APIs in financial services.

In this sense, some aspects of the paper are more mature than others. Where there is room to be prescriptive, we have chosen to be; where areas are less mature, we have chosen to provide commentary on best practice and set out the considerations.

Broadly speaking, the document follows the below logic and order:

* Terms and Definitions: all terms in the document.
* Design Principles: The initial considerations for the design of the API.
* Related Technology: Overview and commentary on the different technology options.
* WAPI Styles: Specific guidance on APIs under WAPI Technical Specification.

In Annex X, we set out some examples of how to approach the document depending on the business area/desired API functionality.

# Scope

# This document defines the framework, function and protocols for an API ecosystem that will enable online synchronised interaction. Specifically, the document:

* defines a logical and technical layered approach for developing APIs, including transformational rules. Specific logical models (such as ISO 20022 models) are not included, but they will be referenced in the context of specific scenarios for guidance purposes;
* will primarily be thought about from a RESTful design point of view, but will consider alternative architectural styles (such as Websocket and Webhook) where other blueprints or scenarios are offered.
* defines for the API ecosystem design principles of an API, rules of a web service based API (WAPI), the data payload, and version control.
* sets out considerations relevant to security, identity, and registration of an API ecosystem. Specific technical solutions will not be defined, but they will be referenced in the context of specific scenarios for guidance purposes.
* Architectural usage beyond query / response asynchronous messaging towards public / subscribe to support advanced and existing business models

The document does not include in its scope:

* a specific technical specification of an API implementation in Financial Services;
* developing JSON APIs based on the ISO 20022 specific Message Formats, such as PAIN, CAMT, PACS; a technical specification that is defined or determined by specific legal frameworks.

# Normative references

# [This section will be completed last to cover the rest of the report].

# ISO 20022-1, *Financial services — Universal financial industry message scheme — Part 1: Metamodel*

ISO 20022-3, *Financial services — Universal financial industry message scheme — Part 3: Modeling*

ISO 20022-4, *Financial services — Universal financial industry message scheme — Part 4: XML Schema Generation*

JSON Schema draft 4 specification[**https://tools.ietf.org/html/draft-handrews-json-schema-01**](https://tools.ietf.org/html/draft-handrews-json-schema-01)

JSON API V1.0 <http://jsonapi.org/about/>

YAML Reference

# Terms and definitions

# [This section will be completed last to cover the rest of the report].

# Design Principles

**This section** covers absolute principles for **developing a web API. It covers both standalone design principles that must be considered upfront when developing an API** in financial services**, as well as principles that apply to specific types of API and are discussed in more detail under WAPI styles.**

## Standards Compatibility

The WAPI advocates the use of ISO data standards wherever possible and practical as a best practice in order to facilitate interoperability.

## Extensibility

Where possible the design decisions taken here for the API ecosystem have been designed to be as extensible as possible. Data standards may change, but the API does not. This is to ensure that their use is able to adapt to future use cases or scenarios.

## Non-Repudiation

Non-repudiation is important for the validity of data of exchange in web APIs and will enforce the confidence in the data to be exchanged. Digital signatures could be used in the context of an API.

## Web Resource Unique Identifiers (ID Fields)

A web resource should have a unique identifier (e.g. a primary key) that can be used to identify the resource. These unique identifiers are used to construct URLs/URIs to identify and address specific resources.

## Idempotency

Idempotency must be considered upfront for a web API. This is covered under WAPI Styles.

## States

# There is a need to consider upfront what state will be required.

# Related Technology

This sections talks through related technology and provides some context on its use on APIs in financial services. It does not cover terms and definitions as this is covered above (Terms and Definitions) and does not cover specific guidance on how to approach the technology as this is covered below (WAPI Styles).

## REST and SOAP

REST and SOAP provide ways to access web services and there are considerations for using both.

### REST

REST or RESTful (Representational state Transfer) web services provides the way to transfer the representational state of a resource and to specify the action to be processed on this resource (see Terms and Definitions).

APIs typically use REST as it more efficient (can use smaller message formats) and requires less extensive processing. In most financial scenarios using REST or RESTful, the API user sends a message to the server and then the server replies shortly after. For example, placing an order in a trade system or requesting the balance on an account.

REST architecture can be exploited to implement these request-response communications. The REST architecture style is described by six constraints:

* Uniform Interface: The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the architecture, which enables each part to evolve independently.
* Stateless: One client can send multiple requests to the server; however, each of them must be independent, that is, every request must contain all the necessary information so that the server can understand it and process it accordingly. In this case, the server must not hold any information about the client state. Any information status must stay on client – such as sessions.
* Cacheable: Because many clients access the same server, and often requesting the same resources, it is necessary that these responses might be cached, avoiding unnecessary processing and significantly increasing performance.
* Client-Server: The uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.
* Layered System: A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. Layers may also enforce security policies.
* Code-On-Demand (Optional): This condition allows the customer to run some code on demand, that is, extend part of server logic to the client, either through an applet or scripts. Thus, different customers may behave in specific ways even using exactly the same services provided by the server. As this item is not part of the architecture itself, it is considered optional. It can be used when performing some of the client-side service which is more efficient or faster.

### SOAP

SOAP (Simple Object Access Protocol) is a messaging [protocol](https://en.wikipedia.org/wiki/Protocol_%28computing%29) specification for exchanging structured information in the implementation of [web services](https://en.wikipedia.org/wiki/Web_service) in [computer networks](https://en.wikipedia.org/wiki/Computer_network) (see Terms and Definitions). Its purpose is to induce [extensibility](https://en.wikipedia.org/wiki/Extensibility), [neutrality](https://en.wikipedia.org/wiki/Neutrality_%28philosophy%29) and independence. It uses [XML Information Set](https://en.wikipedia.org/wiki/XML_Information_Set) for its [message format](https://en.wikipedia.org/wiki/Message_format), and relies on [application layer](https://en.wikipedia.org/wiki/Application_layer) protocols, most often [Hypertext Transfer Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP) or [Simple Mail Transfer Protocol](https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol) (SMTP), for message negotiation and transmission.

SOAP allows processes running on disparate operating systems (such as [Windows](https://en.wikipedia.org/wiki/Microsoft_Windows) and [Linux](https://en.wikipedia.org/wiki/Linux)) to communicate using [Extensible Markup Language](https://en.wikipedia.org/wiki/XML) (XML). Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms.

## Websocket and Webhook

Websocket and Webhook protocols exist for scenarios that REST cannot cater for, such as supporting unsolicited responses. Specifically that:

* For client-to-client architectures, Webhooks may be used.
* For client-to-server architectures, WebSockets may be used.
* Used for any kind of public / subscribe implementation.

### WebSocket

The WebSocket [[RFC6455](https://tools.ietf.org/html/rfc6455)] Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers.

The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g., using XMLHttpRequest or <iframe>s and long polling). In scenario of massive data transporting or event driven by server, WebSocket performs better than HTTP, e.g., market data distribution and market announcement publishing.

Features:

* The WebSocket protocol (IETF RFC 6455) is supported by most browsers, but a client may be any software agent.
* A session is initiated by an HTTP request for protocol upgrade. TLS handshake is supported for authentication, and cipher suites may be negotiated for privacy and non-repudiation.
* After initial handshake, a session becomes a bidirectional, asynchronous messaging protocol. There is no need for a polling mechanism as either side may push unsolicited messages.
* Two subprotocols are defined: text messages, such as JSON or XML, and binary messages, e.g. Simple Binary Encoding. More than one subprotocol may be supported by a peer, if desired. Other subprotocols may be registered with IANA, e.g. ISO 20022 messages.
* WebSocket is layered over TCP. It inherits its reliability and flow control features. However, WebSocket frames messages over a TCP stream. Therefore, applications need not be concerned with framing.
* No session protocol headers are imposed on application messages. Messages are self-contained and therefore can be serialized or forwarded as a unit (in contrast to HTTP where semantic information is dispersed over payload, URI, and headers).
* The session protocol provides no mechanism to correlate requests with responses because it is and event-driven protocol rather than request/response. Correlation of events would be performed at the application layer with transaction IDs and the like.

### Webhook

Webhook are "user-defined HTTP call-backs". They are usually triggered by some event, such as pushing code to a repository or a comment being posted to a blog. When that event occurs, the source site makes an HTTP request to the URL configured for the webhook. Users can configure them to cause events on one site to invoke behaviour on another. The action taken may be anything. Common uses are to trigger builds with continuous integration systems or to notify bug tracking systems. Since they use HTTP, they can be integrated into web services without adding new infrastructure.

Webhook are sometimes referred to as “Reverse APIs,” as they give you what amounts to an API must API for the webhook to use. The webhook will make an HTTP request (typically a POST), and you will then accordingly. The client that registers to the webhook, needs to provide a server to receive the callback.

## HTTPS

The Hypertext Transfer Protocol (HTTP) is a widely-used for distributed, collaborative, and information systems. Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. HTTP is the most used protocol to exchange or transfer hypertext.

A later version, the successor , was standardized in 2015, and is now supported by major web servers and browsers over TLS using the Application-Layer Protocol Negotiation ( () extension where or newer is required. In this specification, HTTP/1.1[RFC7230-RFC7235] and HTTP/2.0 are recommended. Although HTTP 1.1 used to be the mainstream of the market, due to the advantages of HTTP 2.0, most browsers have begun to support HTTP 2.0. It is believed that HTTP 2.0 will be popular in next few years.

HTTPS (HTTP over SSL or HTTP Secure) is the use of Secure Socket Layer (SSL) or Transport Layer Security (TLS) as a sublayer under regular HTTP application layering. HTTPS encrypts and decrypts user page requests as well as the pages that are returned by the Web server. The use of HTTPS protects against eavesdropping and man-in-the-middle attacks.

## [JSON](https://www.w3schools.com/js/js_json_xml.asp) and XML

In web communication scenarios, the most used two structured data format for data exchanging are XML and JSON [[RFC8259](https://tools.ietf.org/html/rfc8259)].

### JSON

In computing, JSON (JavaScript Object Notation) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types. It is a very common data format used for browser–server communication, including as a replacement for XML in some AJAX-style systems.

JSON has several advantages. One of the most obvious of these is that JSON is slightly less verbose than XML, because XML necessitates opening and closing tags (or in some cases less verbose self-closing tags), and JSON uses name/value pairs, concisely delineated by “{“ and “}” for objects, “[“ and “]” for arrays, “,” to separate pairs, and “:” to separate name from value. Even when zipped (using gzip), JSON is smaller and it takes less time to zip it.

Another advantage that JSON has over XML is that its representation of objects and arrays allows for direct mapping onto the corresponding data structures in the host language, corresponding directly to the object of JavaScript, so code to parse and package it fits very naturally into JavaScript code, but XML needs to have an analytical process.

JSON is simple, since it supports a variety of server-side languages and native new data. JSON format can be directly used for server-side code, which greatly simplifies the server-side and client-side code development, reduces the consumption of network bandwidth and is easy to maintain.

### XML

There are several advantages that XML has over JSON. One of the biggest differences between the two is that XML can put metadata into the tags in the form of attributes.

Another advantage of XML is that most browsers render it in a highly readable and organized way. The tree structure of XML lends itself well to this formatting, and allows for browsers to let users to naturally collapse individual tree elements.

One of the most significant advantages that XML has over JSON is strongitsstrong typing, which is good to communicate mixed content, i.e. strings that contain structured markup.

## Content Negotiation

Content negotiation refers to the mechanism in which a client and server negotiate the style of content that is returned from the server. The client can request a certain style of document using the following HTTP headers: Accept, Accept-Language, Accept-Charset. These refer to the format of the document, language of the document and the character set of the document respectively.

On receipt of the HTTP request, a server must look at the Accept header to determine whether or not it is acceptable. If it is not acceptable then the server must return a  HTTP 406 Not Acceptable status code. If the request is acceptable, the server must attach the correct MIME type for the request in the Content-Type response header. The server may choose to respect the language and character set preferences of the browser to format the response also. If the server chooses to obey the Accept-Language and Accept-Charset headers, the response headers used should be Content-Language for the Accept-Language header and the Content-Type should be appended with the character set information.

## RESTful API Description Languages

RESTful API description languages are formal languages designed to provide a structured description of a RESTful web API that is useful both to a human and for automated machine processing. The structured description might be used to generate documentation for human programmers; such documentation may be easier to read than free-form documentation, since all documentation generated follows the same rules and formatting conventions. Additionally, the description language is usually precise enough to allow automated generation of various software artifacts, like libraries, to access the API from various programming languages, which takes the burden of manually creating them off the programmers.

The RESTful API description language is usually neutral, language-agnostic and industry-agnostic. It does not define API itself, but is useful for designers, programmers and users of an API ecosystem, especially in building up large-scale APIs. In essence, RESTful API description language is a software engineering methodology but a computer architectural or communication technology. The community around RESTful API description languages is active and the landscape is still changing. Up to now, the most active projects in this area are OpenAPI, RAML and API Blueprint:

* OpenAPI Specification

Originally known as the Swagger Specification, is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services. It became an open source collaborative project of the Linux Foundation since 2016. The latest version is 3.0.

URL: https://swagger.io/specification/

* RAML

RAML is a YAML-based language for describing RESTful APIs. It provides all the information necessary to describe RESTful or practically RESTful APIs. Although designed with RESTful APIs in mind, RAML is capable of describing APIs that do not obey all constraints of REST (hence the description "practically RESTful"). It encourages reuse, enables discovery and pattern-sharing and aims for merit-based emergence of best practices.

URL: https://raml.org/

* API Blueprint

API Blueprint is a documentation-oriented web API description language. The API Blueprint is essentially a set of semantic assumptions laid on top of the Markdown syntax used to describe a web API.

URL: https://apiblueprint.org/

# Naming Conventions

The following table lists the applicable character case conventions that are used throughout the world.

|  |  |  |
| --- | --- | --- |
| API element | Rules | example |
| HTTP Headers | Although RFC 2616 specifies that HTTP headers are case insensitive, it should be a best practice to adopt, within an API specification, the same character case for all headers definitions. [RFC6648] Custom proprietary “X-“ type headers should be avoided | Train-Case is the most widely used formalism. Words are capitalised and separated with hyphens (-). | Accept-Charset |
| Acronyms should be capitalised | WWW-Authenticate |
| Query parameters | The most used formalisms are  | snake\_case: Words are in lower case and separated with underscores (\_) | currency\_code |
| lowerCamelCase: Spaces and punctuation are removed and the first letter of each word, except the first one, is capitalised | currencyCode |
| Resource\_path | See below |  |
| Request body | The most used formalisms are | snake\_case: Words are in lower case and separated with underscores (\_). | currency\_code |
| lowerCamelCase: Spaces and punctuation are removed and the first letter of each word, except the first one, is capitalised. | currencyCode |
| Data Types | Above are variables, for data types UpperCammelCase used to define the data typee.g."country": { "title": "Ctry, Country", "description": "Country of the address.", "$ref": "#/definitions/CountryCode” | CountryCode |

# Resource Path

The resource path aims to specify the resources which are relevant for a given API request.

## Resource Hops

This resource path is a chain of one or more resource hops. Each resource hop is built on the following components:

* A resource type is mandatory for all resource hops
* A resource identifier is mandatory except for the last resource hop (see below).

When multiple resource hops are specified within a resource path, it is assumed that each resource, except the first one, is semantically linked to the previous one in the path.

|  |  |  |
| --- | --- | --- |
| Hop element | Rules | example |
| Resource Type | The standard pattern is to use names for resources that are in the plural form to designate the collection of resources of the same type (e.g. ‘orders’ , not ‘order’).The most widely used character case convention for resource types is spinal-case. Words are in lower case and separated with hyphens. | orderspayment-requests |
| Resource Id | A character string that can be put within a HTTP URI in order to identify a single resource on the API server side.This Id is most likely provided by the API server.Any resource hop on the resource path, except the last one must be identified. | 000235698741pmtrqst1652 |

Resource type and ID are distinct elements of the request URI, separated by a slash (/).

**Example**

|  |
| --- |
| Resource Hop |
| /orders/000235698741 |
| /payment-requests/pmtrqst1652 |

### Single resource versus collections of resources

The last resource hop in the resource path is the effective target of the API request.

If this hop has, this means the target is actually the single resource which matches with the ID.

If not, the target is the collection of all accessible resources.IDs.

**Example**

|  |  |
| --- | --- |
| Resource | Target |
| /orders | The collection of all orders |
| /orders/000235698741 | A specific order in the collection of orders |
| /payment-requests/ | The collection of all payment requests |
| /payment-requests/ pmtrqst1652 | A specific item within the collection of payment requests |
| /payment-requests/ pmtrqst1652/instructions | The collection of all instructions within a given payment request |
| /payment-requests/ pmtrqst1652/instructions/1526 | A specific instruction within a given payment request |

# WAPI Styles

This section explains how parts of an API should be designed and implemented. To the extent possible, it provides recommendations and informs decision making for web APIs in financial services.

There are three WAPI styles covered in this document: Rest; Asynchronous Messaging; and Service Push. It is acknowledged that there are other methods in financial services, but these are considered to be the main examples to cover use cases.

## REST

* In most financial scenarios, the API user sends a message to the server and then the server replies it after a short time, e.g., placing an order in a trade system.
* RESTful is an architectural style for developing web services which defines a set of constraints, including client–server architecture, uniform interface, stateless, etc.
* Constraints most relevant to WAPI are the uniform interface and the stateless sessions:
	+ Uniform Interface: The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the architecture, which enables each part to evolve independently. RESTful API usually build uniform interface using HTTP methods GET, POST, PUT and DELETE to operate resources.
	+ Stateless: One client can send multiple requests to the server; however, each of them must be independent, that is, every request must contain all the necessary information so that the server can understand it and process it accordingly. In this case, the server must not hold any information about the client state. Any information status must stay on client – such as sessions.
* REST style is an abstraction of the architectural elements within a distributed hypermedia system, and the nature and state of an architecture’s data elements is a key aspect of REST. Data Element most relevant to WAPI are resources and resource identifiers (e.g. URI), representations and representations metadata(e.g. HTML, JSON, XML).
* A primary benefit of using REST, both from a client and server's perspective, is REST-based interactions happen using constructs that are familiar to anyone who is accustomed to using the internet's Hypertext Transfer Protocol (HTTP). In other word, HTTP is **recommended** to implement RESTful WAPI.
* In summary, the key factors of designing and implementing RESTful WAPI including: Uniform Interface, HTTP methods usage, Stateless, Idempotency, Resources and Resource Identifiers usage.

**Use Case Example: Access to Account**

REST implementation in Access2account

### Uniform interface

#### Resource-Based

It is recommended resources identified in requests using URIs as resource identifiers. The resources themselves are conceptually separate from the representations that are exchanged with the client. The exchanged information is expressed in a particular syntax (e.g. XML or JSON), depending on the details of the request and the server implementation.

#### Manipulation of Resources through Representations

When a client knows a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

#### Self-descriptive Messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cache-ability.

#### Hypermedia as the Engine of Application State (HATEOAS)

Clients deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name). Services deliver state to clients via body content, response codes, and response headers. This is technically referred-to as hypermedia (or hyperlinks within hypertext).

Aside from the description above, HATEOAS also means that, where necessary, links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects.

In such architectures, each response would thus contain the subsequent representations the resource can be in. So the response to get to the “Authorised” state would also contain three new URI, pointing to three states (Authorised, Rejected and Accepted).

Hypermedia may also be used to provide information about what next the client is able to do:

* pagination features
* further manipulations of resources or sub-resources or linked resources

When HATEOAS is used, to ensure that the naming of different sections within the response is consistent, it should adhere to the principles of jasonapi.com v1.0: <http://jsonapi.org/about/>

### Apply the standard HTTP methods

**Usage guidelines**

* Use the standard HTTP methods POST, GET, PUT and DELETE, (and PATCH) to operate on the resources, with the following meanings: POST = create, GET = read, DELETE = delete, PUT (or PATCH) = update.
* Use POST to create resources without any specified resource identifier. After successful resource creation with POST, the best practice for the server is:
* To assign an identifier to the newly created resource
* To answer the client with a “201 Created” and HTTP status along with the location of the newly created resource.
* Use GET to retrieve either a given resource that is specified using its identifier, or a set of resources that might be specified through some search conditions. Some other notes on GET requests:
* GET requests can be cached
* GET requests remain in the browser history
* GET requests can be bookmarked
* GET requests should never be used when dealing with sensitive data
* GET requests have length restrictions
* Since restrictions on GET, use POST to read is allowed when dealing with sensitive data or too many parameters.
* Use DELETE to delete either a given resource that is specified using its identifier, or a set of resources that might be specified through some search conditions.
* Use PUT for ‘full’ updates, i.e. when the entire resource is in the body of the PUT request, and will replace the previous resource in its entirety. In other word, PUT must be a full resource update, you MUST send all attribute values in a PUT request to guarantee idempotency.
* Use PATCH for partial updates when only one or a few attributes of a resource are present in the update request, but the resource itself is not replaced. Use RFC 7396 for the body of the PATCH. This is the simplest, most intuitive way, and preferred. Although RFC 7396 relies on using JSON as the syntax, the rules specified therein can also apply when using XML.

**Example**

| **URL** | **POST** | **GET** | **PUT** | **PATCH** | **DELETE** |
| --- | --- | --- | --- | --- | --- |
| /orders | Create an order (returns an id for the created order) | List orders | Bulk replace or create orders[[1]](#footnote-1) | Bulk update orders[[2]](#footnote-2) | Delete all orders |
| /orders/000235698741 | Error[[3]](#footnote-3) | Get the details on this order | If it exists, replace the order, else create it[[4]](#footnote-4) | If exists, update the order or fail otherwise | Delete this order |

### Stateless sessions

Stateless sessions indicate that the necessary state to handle the request is defined by the client and contained within the request itself, whether as part of the URI, query-string parameters, body, or headers. The URI uniquely identifies the resource and the body contains the state (or state change) of that resource. Then after the server does its processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, HTTP status codes and response body.

This is the opposite of a “session” which maintains state across multiple HTTP requests. In REST, the client must include all information for the server to fulfil the request, resending state as necessary if that state must span multiple requests.

Both the state and a resource are needed:

* State, or application state, is that which the server cares about to fulfil a request—data necessary for the current session or request.
* A resource, or resource state, is the data that defines the resource representation—the data stored in the database, for instance. Consider application state to be data that could vary by client, and per request. Resource state, on the other hand, is constant across every client who requests it.

### Idempotency

Normally, the Web APIs shall not be idempotent for ‘read’ applications without additional design. The Web APIs are recommended to be idempotent for ‘write’ applications, based on the below principles:

* The APIs for creating, updating or deleting resources are **recommended** to be idempotent. The intent of this capability is to allow an API user to retry API requests that failed with a timeout or an unexpected error.
* For operations that may be disruptive it is recommended that an idempotency key is **implemented** in API requests.
* It is recommended, the user of an API **must not** change the request body while using the same idempotency key. If the API user changes the request body, the API provider **must not** modify the end resource.
* An API provider **must** treat a request as idempotent if it had received the first request with the same key.
* The API Provider **must** respond to the request with the current status of the resource, if it is successful. An API provider **may** use the message signature (if implemented) along with the idempotency Key to ensure that the request body has not changed.

### Composition of the URI



The resource\_path may include more than one resource where applicable, accompanied by resource ID’s. Usually URLs don’t contain verbs as they are based on resource paths.

In a Type, Issuer, Version concept the URI can reflect this by defining the {service} in the way, that the Type (what I call) and the Issuer (who provided the call) are reflected. Allows an ecosystem of similar services from different provides (Issuer).

**Example**

https://api.example.com/refdata/v1/bics

### Handling associations between resources

Expose associations between resources in the URI path. Keep however those relations in the URI to a minimum. Usually the primary key and the resource affected suffice.

**Example**

GET https://api.example.com/refdata/v1/bics/PVRBRU4VXXX/ssis

means:

“Get all SSIs for BIC PVRBRU4VXXX”

Where BIC PVRBRU4VXXX is considered the primary key.

### Request parameter usage

There may be various types of "parameters" in a request:

* Locators (resource identifiers or a specific action)
* Filters (parameters that provide a search for, sorting or limit results)
* Content (data to be stored)

There are different placeholders where to put those parameters:

* URI query parameters (the portion of the URI that follows the ‘?’ question mark )
* URI paths (the portion of the URI that follows the hostname or ‘fully qualified domain name’, and that precedes the ‘?’ question mark if present)
* HTTP request body
* HTTP request headers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **URI query** | **URI paths** | **Request body** | **Request header**  |
| **Locators** |  | X |  |  |
| **Filters** | X |  |  |  |
| **State** |  |  |  | X |
| **Content** |  |  | X |  |

#### State

State is set in headers, depending on what type of state information it is.

#### Content

Content has only one place where it belongs, and that is the request body, either as payload/body content (XML or JSON) or as multipart/form-data request. Content of the resource should only be in the body and not anywhere else, such as the Header.

#### Resource Filters

Locators belong in the URL path.

**Example**

/bics/CITIUS33, where CITIUS33 is the id of a specific resource of type ‘bic’

Filters go in the query string, because while they are a part of getting the correct data, they are only there to return a subset of what the Locators return.

There are two exceptions whereby Locators and Filters can go into the HTTP Request body instead of the URI:

* When the length of the URL exceeds the maximum length (2K) that need to process these URLs.
* When the information in the Locator or Filter needs to be encrypted. These would normally be put in the URL, but URLs tend to end up in logfiles or tracetools in unencrypted form, so URLs are not a good place for confidential information.

If your API needs to support these two exceptional cases, then for these cases allow using a POST instead of a GET, with method=GET in the URI query part of the POST, and with the body of the POST containing the URI query part as it would have appeared in the normal GET. Where possible – it is not RESTful to use POSTs for GETs.

#### Attributes and Search Criteria

Put resource attributes and search criteria behind the ‘?’

Distinguish relations between resources from their attributes. Typically a relation between two resources is expressed through the path in the URL whereas an attribute of a resource is put behind the ‘?’.

Example

/bics/PVRBRU4VXX/ssis?currency\_code=USD&ssi\_category=COPA

means:

“All SSIs for BIC PVRBRU4VXX for currency code “USD” and SSI category “COPA”

Attributes in the URI query should be child elements of the resource.

The order of attributes in the URI query has no meaning.

#### Specifying multiple allowed values for an attribute

Use a comma separated list of values after the attribute’s name and the ‘=’ sign to specify

**Example**

GET /bics?address.country=Belgium,Germany

Returns bics where the country of their address is either Belgium or Germany.

#### Filtering

Reducing the list of attributes returned per resource (filtering)

Use ‘fields=’ in the query part of the URL, followed by a comma separated list of attribute names. Other fields or attributes of the resource will then not be returned in the response.
It is recommended that your API supports this, in case it deals with big objects, and it needs to work over restricted bandwidth e.g. mobile phone apps.

**Example**

GET /bics?fields=institution\_name,address.city

returns

{

 “name” “My Bank Inc.”

 “address” : {

 “city” : “London”

 }

}

#### Modeling of ‘OR’ filters vs ‘AND’ filters

**Simple AND queries**: send individual calls for each leg of the query. The different responses may contain duplicates.

**Simple (X)OR queries**: send individual calls for each leg of the query. If the first response is negative, then send the second call etc...

**Complex, nested AND/(X)OR queries**: define a generic structure in the URL behind a parameter that is called ‘q’

* Use ‘&’ to indicate AND
* Use ‘|’ to indicate OR
* Use the dot notation to name resource attributes that are subfields of a hierarchically nested resource structure, E.g. address.city

#### Pagination requests

The most commonly used technique, which is recommended, is to use **limit** to indicate the maximum number of objects that should be returned in a ‘page’, and **offset** to indicate the starting object of the page.

The limit is a maximum, since e.g. the last returned page might contain fewer elements than the limit.

This will work for all simple paginations, and is easy to use for the application developer. For complex cases (graphs, dynamically changing sets …), other techniques might be applied.

 **Example**

orders?limit=25&offset=50

Returns maximum 25 elements, starting at 50 places beyond the first element in the returned list of orders. In a numbered list where the first element has index 0, this returns objects 50 through 74. In a numbered list where the first element has index 1, this returns objects 51 to 75.

See section [XXXX] for pagination responses.

Actually, HATEOAS is a smart way for the server to specify in a partial response which links should be used by the client for getting the other pages of the result.

### Post usage

There are two usages recommended using POST: create resources, read resources.

POST is used to read when GET method restricted, like parameter length limit or security requirement. When using POST to read, URI query parameters is not allowed, instead query parameters are passed via request body as multipart/form-data content type.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **URI query** | **URI paths** | **Request body** | **Request header**  |
| **Locators** |  | X |  |  |
| **Filters** |  |  | X |  |
| **State** |  |  |  | X |
| **Content** |  |  | X |  |

**Example**

read order via GET and POST

https://api.example.com/orders?orderId=xxxx

POST /orders HTTP/2.0

Content-Type: application/x-www-form-urlencoded;charset=utf-8

Host: api.example.com

orderId=xxxx

POST is used to create resources with data payload(contains idempotency key) as xml or json content request. See section [11] for data payload syntax.

**Example**

create an order with idempotency key "orderId"

POST /orders HTTP/2.0

Content-Type: application/json;charset=utf-8

Host: api.example.com

{

 “orderId” “order0000000001”

 “orderBody” : {

 “price” : “xxx”

 }

}

### The response

#### Object and status

The normal response to a request is an HTTP response with HTTP status code *200 – “OK”,* followed by a response body that contains the representation of an object and its state, either in JSON or XML format.

#### Success vs errors

**Success** : When the request can be processed, i.e. a ‘business response’ can be returned. A business response is a representation of an object and/or its status, or a result of an action (e.g. a creation of a resource, or a calculation, etc.). In that case, no ‘error’ is returned: the HTTP status code will  be any success 2XX class response as per the HTTP standard, and no ‘error’ block will be present in the response.

**Error** : When the API request cannot be processed. There is no business response: no representation of an object or its state is returned, and no result of an action is returned. Instead, an error element is returned: the HTTP status code will be in the 4xx range or 5xx range, and an ‘error’ element will be present in the response.

#### The error element

It is noted that not all fields, as per the below, are necessary and some are options.

HTTP status codes are sometimes not sufficient to convey enough information about an error to be helpful. This specification defines simple JSON formats to suit this purpose. They are designed to be reused by HTTP APIs, which can identify distinct "problem types" specific to their needs. Thus, API clients can be informed of both the high-level error class (using the status code) and the finer-grained details of the problem (using one of these formats). The error information format is

The canonical model for problem details is a JSON object. The [RFC7807](https://tools.ietf.org/html/rfc7807) defines error details as "application/problem+json" media type.

A problem details object can have the following members:

"type" (string) - A URI reference that identifies the problem type. This specification encourages that, when dereferenced, it provide human-readable documentation for the problem type. When this member is not present, its value is assumed to be "about:blank", returning a blank page.

"title" (string) - A short, human-readable summary of the problem type. It SHOULD NOT change from occurrence to occurrence of the problem, except for purposes of localization (e.g., using proactive content negotiation).

"status" (number) - The HTTP status code ([RFC7231]) generated by the origin server for this occurrence of the problem.

"detail" (string) - A human-readable explanation specific to this occurrence of the problem.

"instance" (string) - A URI reference that identifies the specific occurrence of the problem. It may or may not yield further information if dereferenced.

Consumers MUST use the "type" string as the primary identifier for the problem type; the "title" string is advisory and included only for users who are not aware of the semantics of the URI and do not have the ability to discover them (e.g., offline log analysis). Consumers SHOULD NOT automatically dereference the type URI.

The "status" member, if present, is only advisory; it conveys the HTTP status code used for the convenience of the consumer. Generators MUST use the same status code in the actual HTTP response, to assure that generic HTTP software that does not understand this format still behaves correctly.

Consumers can use the status member to determine what the original status code used by the generator was, in cases where it has been changed (e.g., by an intermediary or cache), and when message bodies persist without HTTP information. Generic HTTP software will still use the HTTP status code.

The "detail" member, if present, ought to focus on helping the client correct the problem, rather than giving debugging information.

For example, an HTTP response carrying JSON problem details:

 HTTP/1.1 403 Forbidden

 Content-Type: application/problem+json

 Content-Language: en

 {

 "type": "https://example.com/probs/out-of-credit",

 "title": "You do not have enough credit.",

 "detail": "Your current balance is 30, but that costs 50.",

 "instance": "/account/12345/msgs/abc",

 "balance": 30,

 "accounts": ["/account/12345",

 "/account/67890"]

 }

Here, the out-of-credit problem (identified by its type URI) indicates the reason for the 403 in "title", gives a reference for the specific problem occurrence with "instance", gives occurrence-specific details in "detail", and adds two extensions; "balance" conveys the account's balance, and "accounts" gives links where the account can be topped up.

The ability to convey problem-specific extensions allows more than one problem to be conveyed. For example:

 HTTP/1.1 400 Bad Request

 Content-Type: application/problem+json

 Content-Language: en

 {

 "type": "https://example.net/validation-error",

 "title": "Your request parameters didn't validate.",

 "invalid-params": [ {

 "name": "age",

 "reason": "must be a positive integer"

 },

 {

 "name": "color",

 "reason": "must be 'green', 'red' or 'blue'"}

 ]

 }

Note that this requires each of the subproblems to be similar enough to use the same HTTP status code. If they do not, the 207 (Multi-Status) [[RFC4918](https://tools.ietf.org/html/rfc4918)] code could be used to encapsulate multiple status messages.

#### Pagination responses

See section [XXXX] for pagination requests.

Every page of the response may contain metadata, structured as follows:

“page\_header” : { “total\_count” : x, “offset”: y, “count” : z },

The response will use a separate element page\_header to indicate information about the paging, containing the following pieces of data:

* total\_count to indicate the total number of objects available in the returned list across all pages. This is not mandatory.
* count to indicate the number of objects in the page. This value should be the same as the value of limit in the request, but that’s not guaranteed. E.g. in the last page, count can be lower than the value of limit in the request (see section XXXX).
* offset to indicate the starting object of the page. This value should be the same as the value of offset in the request (see section XXXX).

#### Empty list

If an API call searches for resources (objects, records…), it will return a list of multiple resources that match the search criteria, and that list can be empty if no matching resources were found. The empty list is not an error since in this case; the resource is the list itself which is being accessed. The best practice for returning lists is to indicate how many objects (or ‘ records’) are in the list – this is called the metadata - plus the list of resulting records. In case of the empty list, the HTTP status will be ‘success’ (HTTP ‘200 – OK’), the metadata will show “total\_count” = 0, and the list will be the empty list (an empty array in JSON).

**Example**

GET /orders

Response

{

 “list\_header” : { “total\_count” : 0, “offset”: 0, “count” : 0 },

 “orders” : []

}

This is different from requesting a single resource, where the expected response is a representation of this single resource (not a list of resources). If the request for a single resource cannot return a representation of this resource then this is an error.

Example “single resource not found”

A request to retrieve an order with a particular order ID which does not match an existing order: this will return a HTTP ‘404 – Not found’ with an error block that indicates that the requested order ID does not exist.

## Asynchronous Messaging & Server Push

* Like any other architectural style, REST or RESTful API can't fit in with all the needs of the web API ecosystem; specifically it is not useful for implementation of any type of services where the server updates the client or allows the client to continue processing without an immediate response.
* To implement these facilities, we may need to introduce asynchronous messaging/server push styles to API ecosystem. Asynchronous messaging is a communication method allows an application, either client or server, send out a message does not require an immediate response to continue processing.
* Server push is a style of Internet-based communication where the request for a given transaction is initiated by the publisher (central server). Push services are often based on information preferences expressed in advance, also called a publish/subscribe model.
* Neither asynchronous messaging nor server push is restricted to certain technology. In early days, people used a raw TCP socket like Java applet or used HTTP streaming/long polling which called Comet framework to make a web server to push data to a browser. Recent web technologies realized push mechanism (HTTP/2.0) or full-duplex TCP connection (WebSocket) can support asynchronous messaging and server push much easier.
* These are criteria for using asynchronous messaging & server push as opposed to a RESTful API.
	+ The application requires real-time performance or a very high transaction rate. Features that contribute to lower latency are: no polling required, no session headers required.
	+ Further performance gains could be derived from binary encoding of messages. Highly transactional applications are machines talking to each other; humanly readable messages like JSON and XML are not only unnecessary, they are a significant performance drag.
	+ The application is suited to event-based applications with unsolicited messages and subscriptions for subsequent events.
* In summary, the key factors of designing and implementing asynchronous messaging or server push WAPI including: bidirectional communication, message publish/subscribe model.

WAPI implementation in Trading Services

### Bidirectional Communication

A general model for push services includes three basic actors: a user agent, a push service, and an application (server).



At the very beginning of the process, a new message subscription is created by the user agent and then distributed to its application server. This subscription is the basis of all future interactions between the actors. A subscription is used by the application server to send messages to the push service for delivery to the user agent. The user agent uses the subscription to monitor the push service for any incoming message.

To offer more control for authorization, a message subscription is modelled as two resources with different capabilities:

* A subscription resource is used to receive messages from a subscription and to delete a subscription. It is private to the user agent.
* A push resource is used to send messages to a subscription. It is public and shared by the user agent with its application server.

It is expected that a unique subscription will be distributed to each application; however, there are no inherent cardinality constraints in the protocol. Multiple subscriptions might be created for the same application, or multiple applications could use the same subscription. Note, however, that sharing subscriptions has security and privacy implications.

Subscriptions have a limited lifetime. They can also be terminated by either the push service or the user agent at any time. User agents and application servers must be prepared to manage changes in the subscription state.

A push service is usually identified as a web resource, whose URI is as follows:

{protocol}://{host}/{push service}/{version}/

**Example**

Push service via WebSocket

wss://api.example.com/orders/v1

### Message Subscription

Beyond the technology this allows customer self-service and adjustment of any kind if subscription. As may for larger data content still messaging is used, this subscription generates a message in a other format over another channel. E.g. asking for all statements of this year, may end up in a battery of CAMT.053 over FileAct or EBICS.

In the scenario of message subscription, the user agent, push service and application server communicate through predefined JSON objects, which are as follow:

Subscription JSON:

{“Type”:”subscription”,”Topic”:”topic name”}

A subscription creation JSON object is sent from user agent to push service when the user agent wants to get a push message from the push Subscription. Here the “Type” element indicates the operation of the push service and the “Topic” element indicates the topic of the push subscription. If the push subscription does not exist, then the push service will create a new push subscription with the topic name.

Subscription Obsolete JSON:

{“Type”:”subscription\_obsolete”,”Topic”:”topic name”}

A subscription obsolete JSON object is sent from the user agent to the push service when a user agent wants to cancel an existing push subscription. Here the “Type” element indicates the operation of the push service and the “Topic” element indicates the topic of the messages in the push subscription. If no user agent belongs to a push subscription, the service would be obsoleted from the push service.

### Message Publish

In the scenario of message publish, the user agent, push service and application server communicate through predefined JSON objects, which are as follow:

Push Message JSON:

{“Type”:”push\_message”,”Topic”:”{topic name}”, “Body”:{message body}}

A push message JSON object is sent from user agent to push service or application server to push service when the push message is created. Here the “Type” element indicates the object is a push message and the “Topic” element indicates the topic of the push subscription that the push message belongs to. Body is the message body.

When a push message JSON object is sent to a push service, the server copies and pushes the push message to all the user agents associated with the push subscription.

# Data Payload Syntax

## JSON

**JSON** is a format for sharing data. As its name suggests, JSON is derived from the JavaScript programming language, but it’s available for use by many languages including Python, Ruby, PHP, and Java.

This format is easy to transmit between web server and client or browser. Very readable and lightweight, JSON offers a good alternative to XML and requires much less formatting. This informational guide will get you up to speed with the data you can use in JSON files, and the general structure and syntax of this format.

Beside JSON, YAML is as well an opportunity and better human readable.

### Syntax and Structure

A JSON object is a key-value data format that is typically rendered in curly braces.

JSON syntax is derived from JavaScript object notation syntax:

* Data is in name/value pairs
* Data is separated by commas
* Curly braces hold objects
* Square brackets hold arrays

A JSON object looks something like this:

"definitions": {

 "PostalAddress6-API": {

 "title": "PostalAddress6 Swiss Corporate API definition equivalent",

 "description": "Information that locates and identifies a specific address, as defined by postal services or in free format text.",

 "type": "object",

 "properties": {

 "streetName": {

 "title": "StrtNm, StreetName",

 "description": "Name of a street or thoroughfare.",

 "$ref": "#/definitions/Max70Text”

 },

 "buildingNumber": {

 "title": "BldgNb, BuildingNumber",

 "description": "Number that identifies the position of a building on a street.",

 "$ref": "#/definitions/Max16Text”

 },

 "postCode": {

 "title": "PstCd, PostCode",

 "description": "Identifier consisting of a group of letters and/or numbers that is added to a postal address to assist the sorting of mail.",

 "$ref": "#/definitions/Max16Text”

 },

 "townName": {

 "title": "TwnNm, TownName",

 "description": "Name of a built-up area, with defined boundaries, and a local government.”

, "$ref": "#/definitions/Max35Text”

 },

 "country": {

 "title": "Ctry, Country",

 "description": "Country of the address.",

 "$ref": "#/definitions/CountryCode”

 }

 },

 "additionalProperties": false },

In a financial WAPI JSON object may be consisted of two parts:

{“header”:{header content},”body”:{body content}}

Header content contains the specific information of different Financial API standards(e.g., financial standard name), and body content is the actual data.

### Data Types

JSON **values** are found to the right of the colon. At the granular level, these need to be one of 6 simple data types:

* strings
* numbers
* objects
* arrays
* Booleans (true or false)
* null

At the broader level, values can also be made up of the complex data types of JSON object or array. Schemes can be used to define the expected data structure and explain the constraint of each field used within the message.



There are some solutions to validate JSON object, e.g., [JSON schema](http://json-schema.org/).

## XML

By using tree structure, you can get to know all succeeding branches and sub-branches starting from the root. The parsing starts at the root, then moves down the first branch to an element, the branch from there, and so on to the nodes.

XML tags identify the data and are used to store and organize the data HTML used to in .

There are three important characteristics of XML that make it useful in a variety of systems and solutions −

XML is extensible − XML allows you to create your own self-descriptive tags, or language, that suits your application.

XML carries the data and allows you to store the data irrespective of how it will be presented.

 (it),XML is a public standard − XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

### Syntax and Structure

Following is a complete XML document

<?xml version = "1.0"?>

<contact-info>

 <name>Tanmay Patil</name>

 <company>TutorialsPoint</company>

 <phone>(011) 123-4567</phone>

</contact-info>

There are two kinds of information in the above example

Markup, like <contact-info>

The text, or the character data, Tutorials Point and (040) 123-4567.

The following diagram depicts the syntax rules to write different types of markup and text in an XML document.



An XML file is structured by several XML-elements, also called XML-nodes or XML-tags. The names of XML-elements are enclosed in triangular brackets < >. An attribute specifies a single property for the element, using a name/value pair. An XML-element can have one or more attributes. References usually allow you to add or include additional text or markup in an XML document. References always begin with the symbol "&" which is a reserved character and end with the symbol ";". The names of XML-elements and XML-attributes are case-sensitive, which means the name of start and end elements need to be written in the same case.

The tree structure contains root (parent) elements, child elements and so on. By using tree structure, you can get to know all succeeding branches and sub-branches starting from the root. The parsing starts at the root, then moves down the first branch to an element, take the first branch from there, and so on to the leaf nodes.

Example

Following example demonstrates simple XML tree structure

<?xml version = "1.0"?>

<Company>

 <Employee>

 <FirstName>Tanmay</FirstName>

 <LastName>Patil</LastName>

 <ContactNo>1234567890</ContactNo>

 <Email>tanmaypatil@xyz.com</Email>

 <Address>

 <City>Bangalore</City>

 <State>Karnataka</State>

 <Zip>560212</Zip>

 </Address>

 </Employee>

</Company>

Following tree structure represents the above XML document



XML Trees Structure

In the above diagram, there is a root element named as <company>. Inside that, there is one more element <Employee>. Inside the employee element, there are five branches named <FirstName>, <LastName>, <ContactNo>, <Email>, and <Address>. Inside the <Address> element, there are three sub-branches, named <City> <State> and <Zip>.

### Data Types

The following table lists primitive XML schema data types, facets that can be applied to the data type, and a description of the data type.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Facets** | **Description** |
| **string** | length, pattern, maxLength, minLength, enumeration, whiteSpace | Represents character strings. |
| **boolean** | pattern, whiteSpace | Represents Boolean values, which are either **true** or **false**. |
| **decimal** | enumeration, pattern, totalDigits, fractionDigits, minInclusive, maxInclusive, maxExclusive, whiteSpace | Represents arbitrary precision numbers. |
| **float** | pattern, enumeration, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents single-precision 32-bit floating-point numbers. |
| **double** | pattern, enumeration, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents double-precision 64-bit floating-point numbers. |
| **duration** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a duration of time.The pattern for **duration** is PnYnMnDTnHnMnS, where nY represents the number of years, nM the number of months, nD the number of days, T the date/time separator, nH the number of hours, nM the number of minutes, and nS the number of seconds. |
| **dateTime** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a specific instance of time.The pattern for **dateTime** is CCYY-MM-DDThh:mm:ss where CC represents the century, YY the year, MM the month, and DD the day, preceded by an optional leading negative (-) character to indicate a negative number. If the negative character is omitted, positive (+) is assumed. The T is the date/time separator and hh, mm, and ssrepresent hour, minute, and second respectively. Additional digits can be used to increase the precision of fractional seconds if desired. For example, the format ss.ss... with any number of digits after the decimal point is supported. The fractional seconds part is optional.This representation may be immediately followed by a "Z" to indicate Coordinated Universal Time (UTC) or to indicate the time zone. For example, the difference between the local time and Coordinated Universal Time, immediately followed by a sign, + or -, followed by the difference from UTC represented as hh:mm (minutes is required). If the time zone is included, both hours and minutes must be present. |
| **time** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents an instance of time that recurs every day.The pattern for **time** is hh:mm:ss.sss with optional time zone indicator. |
| **date** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a calendar date.The pattern for **date** is CCYY-MM-DD with optional time zone indicator as allowed for **dateTime**. |
| **gYearMonth** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a specific Gregorian month in a specific Gregorian year. A set of one-month long, nonperiodic instances.The pattern for **gYearMonth** is CCYY-MM with optional time zone indicator. |
| **gYear** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a Gregorian year. A set of one-year long, nonperiodic instances.The pattern for **gYear** is CCYY with optional time zone indicator as allowed for **dateTime**. |
| **gMonthDay** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a specific Gregorian date that recurs, specifically a day of the year such as the third of May. A **gMonthDay** is the set of calendar dates. Specifically, it is a set of one-day long, annually periodic instances.The pattern for **gMonthDay** is --MM-DD with optional time zone indicator as allowed for **date**. |
| **gDay** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a Gregorian day that recurs, specifically a day of the month such as the fifth day of the month. A **gDay** is the space of a set of calendar dates. Specifically, it is a set of one-day long, monthly periodic instances.The pattern for **gDay** is ---DD with optional time zone indicator as allowed for **date**. |
| **gMonth** | enumeration, pattern, minInclusive, minExclusive, maxInclusive, maxExclusive, whiteSpace | Represents a Gregorian month that recurs every year. A **gMonth** is the space of a set of calendar months. Specifically, it is a set of one-month long, yearly periodic instances.The pattern for **gMonth** is --MM-- with optional time zone indicator as allowed for **date**. |
| **hexBinary** | length, pattern, maxLength, minLength, enumeration, whiteSpace | Represents arbitrary hex-encoded binary data. A **hexBinary** is the set of finite-length sequences of binary octets. Each binary octet is encoded as a character tuple, consisting of two hexadecimal digits ([0-9a-fA-F]) representing the octet code. |
| **base64Binary** | length, pattern, maxLength, minLength, enumeration, whiteSpace | Represents Base64-encoded arbitrary binary data. A **base64Binary** is the set of finite-length sequences of binary octets. |
| **anyURI** | length, pattern, maxLength, minLength, enumeration, whiteSpace | Represents a URI as defined by RFC 2396. An **anyURI** value can be absolute or relative, and may have an optional fragment identifier. |
| **QName** | length, enumeration, pattern, maxLength, minLength, whiteSpace | Represents a qualified name. A qualified name is composed of a prefix and a local name separated by a colon. Both the prefix and local names must be an NCName. The prefix must be associated with a namespace URI reference, using a namespace declaration. |
| **NOTATION** | length, enumeration, pattern, maxLength, minLength, whiteSpace | Represents a **NOTATION** attribute type. A set of QNames. |

#

# Security and Authentication

Web based Application Programming Interfaces implemented in the Financial Services industry are often of high value and need to be appropriately secured.

This chapter describes how implementers can ensure the confidentiality and integrity of data exchanged via such interfaces. In addition it provides a secure framework for authorised access of resources by first and third party client software.

Existing widely used internet standards published by the Internet Engineering Task Force & OpenID Foundation are referenced and should be consulted by implementers.

## Transport Layer Security

All WAPI endpoints shall be protected by transport layer security (TLS), this is sometimes referred to as HTTPS. Without the use of TLS it is trivially easy for financial data in transit to be intercepted or maliciously modified.

The recommendations for Secure Use of Transport Layer Security in BCP195[[5]](#footnote-5) shall be followed, with the following additional requirements:

* TLS version 1.2[[6]](#footnote-6) or later shall be used for all communications.
* A TLS server certificate check shall be performed, as per RFC6125[[7]](#footnote-7).

The implementation of these standards will ensure:

1. the confidentiality of the data sent over the connection - as the connection will be encrypted;
2. the Integrity of the data sent over the connection - as it is not possible for an attacker to inject or tamper with the data

In addition, where the server certificate used in the TLS handshake is issued by a trusted certificate authority the client has a strong assurance of the identity of the server it is interacting with.

Should the server wish to authenticate the client at the transport layer, mutual TLS authentication can be implemented. This involves the client presenting its certificate to the server as part of the TLS handshake and will give the server a strong assurance as to the identity of the client. This is common practice for many existing financial API implementations and is recommended for WAPI implementations.

Technical implementation details for TLS and “mutual TLS” are available in the core TLS specs: TLS 1.2 (<https://tools.ietf.org/html/rfc5246>) and TLS 1.3 (https://tools.ietf.org/html/rfc8446). [Sec Note – need section on Content Revocation?]

### Certificate Issuance and Verification

Key to the authentication and identity characteristics of TLS is the issuance and verification of certificates. There are a number of different approaches that can be applied to financial APIs - each with a different trust model.

#### Standard website certificates issued by globally trusted certificate authorities

Internet Browser and Operating System providers maintain lists of globally trusted certificate authorities, the Mozilla one is widely used and is available [here](https://ccadb-public.secure.force.com/mozilla/IncludedCACertificateReport). A financial API provider can obtain certificates signed by one of these globally trusted certificate authorities from multiple vendors in the market, who will validate that the provider controls a domain before issuing a certificate corresponding to that domain.

These standard "domain validated" certificates provide no authoritative identity assurances themselves, but are often times sufficient when the client has other means of confirming the identity of the financial API provider. It is possible for a provider to obtain an "extended validation" certificate which does provide assurance of the corporate identity of the provider. In practice these provide little additional benefit as the client will still need to make a decision based on external criteria as to whether they trust the API provider.

#### Regional trust frameworks, such as eIDAS in the EU

The eIDAS family of legislation in the EU provides a trust framework with qualified trust service providers. These service providers can issue both server and client certificates and there are various standards to support additional metadata in such certificates, such as the regulatory status of the owner of the certificate. For region specific deployments this approach may be of merit.

#### Federation Operators

Some ecosystems, particularly closed ecosystems, delegate the responsibility for issuing certificates to a federation operator. This entity will have the responsibility for verifying the identity of the participants according to rules specific to the federation. Each participant in the ecosystem will trust the federation operator and may well have a specific contract or terms of service in place with such an operator. Examples of this approach in the UK are the UK Open Banking Directory and the Origo Unipass system.

## Application and Access Layer Security

### Introduction

As detailed in RFC6749[[8]](#footnote-8), in the traditional client-server authentication model, the client software requests an access-restricted resource (protected resource) on the server by authenticating with the server using credentials issued by the server. This could be the username and password of an individual user or an API key granted to a company.

Using such credentials directly via an API creates several problems and limitations, especially in the case where a user wants to allow third party client software to access their resources:

* Third-party applications are required to store the user’s credentials for future use, typically a password in clear-text;
* Servers are required to support password authentication, despite the security weaknesses inherent in passwords;
* Third-party applications gain overly broad access to the user’s protected resources, leaving users without any ability to restrict duration or access to a limited subset of resources;
* Users cannot revoke access to an individual third party without revoking access to all third parties;
* Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth 2.0 addresses these issues by introducing an authorisation layer and separating the role of the client from that of the resource owner. In OAuth, the client software requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner.

Instead of using the resource owner's credentials to access protected resources, the client obtains an access token -- a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to client software by an authorization server with the approval of the resource owner. The client software uses the access token to access the protected resources hosted by the resource server.

This separation of concerns between authorisation servers and resource servers delivers security, scalability and extensibility benefits. Such architecture is now considered best practice and is required for the implementation of WAPI.

The OAuth 2.0 Authorization Framework that consists of [RFC6749], [RFC6750], [RFC7636], and other specifications, is the core protocol used for application layer security and authorisation in the WAPI standard. However OAuth 2.0 itself has a broad application and multiple implementation profiles - not all of which are suitable for financial APIs.

In this standard there are 2 security profiles for implementing secure OAuth 2.0 based APIs:

1. Read Only API Security Profile - to be implemented for APIs where read-only financial data is exchanged
2. Read & Write API Security Profile - to be implemented for APIs where write access is performed, for example the initiation of payments

The Read & Write API Security Profile extends the Read Only API Security Profile and provides additional security guarantees, appropriate due to the high value of the APIs it is designed to protect.

### Overview of the OAuth 2.0 protocol

While there are many variants, this is the primary flow used to allow end-users to grant access to client software:

1. Client software registers with the authorization server and is given a set of credentials (this happens once)
2. Client software redirects user to the authorisation server with several parameters including an identifier for the client and the scope of access that is being requested
3. Authorization server authenticates the user and requests their authorisation to grant access to the client software
4. Authorization server redirects the user back to the client software with an authorization code
5. Client software presents its client credentials and this authorization code to the authorization server
6. The authorization server verifies the credentials and the code and returns an access token to the client software
7. The client software uses this access token to access resources belonging to the user at the resource server

There are many financial APIs that don’t involve end-users, but are exchanges between two institutions. For such APIs it is recommended to use the OAuth 2.0 client credentials token grant. This will allow the API provider to separate the roles of the authorisation server and resource server. Such a flow, would be as follows:

1. Client software registers with the authorization server and is given a set of credentials (this happens once)
2. Client software presents its client credentials along with the a scope value denoting the access it is requesting
3. The authorization server verifies the credentials and the scope and returns an access token to the client software
4. The client software uses this access token to access protected resources.

Such an approach is better than the client software presenting raw credentials on every API call as it separates the actions of granting access and fulfilling access requests.

## Read Only Security Profile

Implementers of WAPI for read-only use-cases should use the OpenID Foundation’s Financial-grade API - Read-Only API Security Profile ([https://openid.net/specs/openid-financial-api-part-1.htmlhttps://openid.net/specs/openid-financial-api-part-1.html](https://openid.net/specs/openid-financial-api-part-1.html))

This is a profile of OAuth 2.0 specifically designed for financial APIs. By using this profile, implementers can ensure that the API is protected from many well known attacks.

It also has a conformance test suite for both authorization servers and client software (<https://gitlab.com/fintechlabs/fapi-conformance-suite>).<https://gitlab.com/fintechlabs/fapi-conformance-suite>). The use of a conformance test suite is required, as it:

* Ensures that the API is secure by checking that the security profile has been applied correctly
* Ensures that the API is interoperable

## Read & Write Security Profile

Implementers of WAPI for read and write use cases should use the OpenID Foundation’s Financial-grade API - Read and Write API Security Profile (https://openid.net/specs/openid-financial-api-part-2.html)

This profile builds on the “Read Only Security Profile” and adds further protection through the use of signed requests, signed responses and proof of possession tokens. This profile also has conformance test suite, the use of which is highly recommended.

## Message Level Integrity, Source Authentication & Non-Repudiation

The use of TLS provides the client software with guarantees of the integrity of the data received and the identity of the server the data is received from. However such guarantees only apply at the time of the exchange and at the transport layer. These guarantees are hard to capture for further verification or auditing at a later date. For this reason, it may be beneficial to implement cryptographic signatures at the application layer.

### Signing HTTP Requests & Responses

There is no international standard for signing HTTP requests or responses and given the use of JSON in this standard, implementers are recommended to consider signing JSON payloads rather than attempting to sign raw HTTP messages.

Some of the reasons why no standard has emerged in this area are:

* HTTP bodies are often benignly tampered with by proxy servers (e.g. to add compression)
* HTTP headers are often added to, or modified in some way

General purpose HTTP signing is therefore likely to result in many false negatives and furthermore is not a good format for archiving - as the signature is detached from the content.

### Signing JSON Payload

The application layer security profiles defined in this standard provide further guarantees around the OAuth 2.0 exchanges through the use of JSON Web Tokens (JWTs). These tokens allow the receiver to authenticate the source of the token and verify it’s integrity. Because they are a self-contained data structure they are easy to audit at a later date and can be used to provide non-repudiation guarantees.

For implementations where it is important to have such properties for all data exchanged the use of JWTs can be extended to all endpoints.

JWTs are already in wide use and their adoption is expanding rapidly. They are defined with the following family of standards:

RFC7519 - JSON Web Token

RFC7515 - JSON Web Signature

RFC7518 - JSON Web Algorithms

RFC7797 - JSON Web Signature Unencoded Payload Option

The design of the JWT family of specifications took learnings from SAML and the problems with canonicalization that have plagued implementations. The simplicity and extensibility of the JWT structure makes them suitable for a wide variety of use cases. They also fit neatly with the REST practices through the use of the “application/jwt” content type. Servers can require that clients send JWT payloads rather than JSON”. This may be appropriate when the client is calling an endpoint to change sensitive data and the server wants to be able to keep a copy of the request that is independently auditable and non-repudiable.

A server may elect to serve either JSON or JWT responses. In such cases the client can specify which response it wants to receive through the HTTP “Accept” header.

While it is possible to separate the signature from the payload (as described in Appendix F of RFC7515[[9]](#footnote-9)) this is not recommended as there is a greater chance of implementation errors. JWTs can be signed with either symmetric keys or using asymmetric keys, both RSA and EC key pairs are supported. For financial APIs implementers should use asymmetric keys and follow the guidelines in the Read Only Security Profile concerning the strength of such keys.

The JWT family of standards provide a number of different options for how the receivers of JWTs can obtain the appropriate public key to verify the signature. For this standard we recommend the use of JSON Web Key[[10]](#footnote-10) endpoints as described in OpenID Connect Core 10.1.1[[11]](#footnote-11). This will allow both servers and clients to rotate keys according to their key management policies without any manual changes required to API implementations.

### Http Signature

Warning: The Http-Signature specification is now registered as a draft by IETF

Reference: <https://datatracker.ietf.org/doc/draft-cavage-http-signatures>

The Http-Signature specification provides a way to check the integrity and the authenticity of a request sent by a WAPI client.

The authentication mechanism applies on

* The HTTP request body
* Some HTTP headers as well

To implement this mechanism, the WAPI client must:

* Compute a digest of the HTTP body and add this digest as an extra HTTP header.
* Create a header field string covering all the desired headers and especially the previously computed body digest
* Use a specific Signature Certificate in order to apply a signature on this header field string and add this signature as an extra-header embedding
	+ The key identifier which must specify the way to get the relevant qualified certificate
	+ The algorithm that has been used
	+ The list of headers that have been signed
	+ The signature itself.

## Message Level Encryption

The use of JWTs allows implementers to support message encryption through the use of RFC7516 - JSON Web Encryption. Please refer to the RFC for details of how to implement this.

## Version control

An API must have a version. The version is part of the URL. It is derived from the version of the service interface exposed to the client.

Specify the version with a 'v' prefix. Position it between the {service} name and the {resource\_path}. Use a simple ordinal number.

The version id of the implementation of the service MAY be included with a service to aid debugging.

The rule of thumb is that every change to the API that breaks the client is a version change. If a change to the API modifies the logic the client application needs to write to handle the response, change the version number in the URL. Below changes to an API do not automatically mean the API must receive a new version:

* Addition of new resources
* Addition of new data items in the response where allowed by the Schema
* Changed technologies (e.g. Java to Ruby)
* Changes to the application back-end services that offer the API

#

# Use Cases

##  ISO20022

### ISO 20022 Web services

#### Introduction

[Note: This section is being updated at the moment. See Minutes from Singapore Meeting]

The premise for defining a modelling methodology for ISO 20022 REST APIs is to start fom the current ISO 20022 methodology and extend it where necessary. Although the current methodology doesn’t have custom REST API artefacts, many of the metamodel artefacts can be reused to design REST APIs.



Figure 1 Leveraging components from the existing ISO 20022 methodology

#### Scope level

##### Modeling guidelines

The starting point is to analyse the business domain by specifying the Business Process and extracting the business concepts that are relevant to the business needs.

##### \\

A key aspect of modelling REST APIs is the ability to expose the different states REST resources can have and the different methods that can be applied at each state.

The resource’s interface is a BusinessTransaction. The description of Business Transaction in the Metamodel and Modelling allows it to describe an interface, whether to single or multiple resources and participants.
In essence BusinessTransaction provides for grouping and sequencing of MessageTransmissions relevant to a Business Process. MessageTransmissions correspond to the API requests and responses.

##### Modeling guidelines

###### Model the Resources (using MessageComponents)

The Business Components that are involved in the Business Process are selected or created. The resulting datamodel is the foundation for the various Resources that will be derived in the context of a solution or service at the logical level.

###### Design the BusinessTransactions

A BusinessTransaction for a collection of resources specifies the **interactions** between a client and a server corresponding with Business Roles in the Business Process. The client sends MessageTransmission requests to, and receives responses from the server. So an interaction between two Participants constitutes of a **request** and a **response**. One Participant will be the **supplier** (aka the server) of the API service, the other the **consumer** (aka the client) of the API service.

Resources are

* a set of MessageComponents representing the data model
* a refinement, custom to the solution, of the data model (which is a set of BusinessComponents) that was defined earlier during the conceptual analysis.

##### Modeling guidelines

###### Refine the Resource(s)

Resources specify all information as a data model that can be managed (created, read, updated and deleted) in the context of an API solution.

Resources are complemented by a model showing the different states each resource can have and the methods that can be applied on that resource for each state.

At least one state model must be defined which is the life-state model containing at least the creation and the destruction.

A resource may be linked with other resources. This link may be a composition, an aggregation or a simple association.

A resource may be further refined to meet the requirements of a data provider. The resource identifier may be distinct from identifiers provided at the business layer.

###### Design the API calls

Introduction

Each api call consists of a request message and a response message.

The response may either be

* a business response message (modelled as a representation response)
* an acknowledgment message without any business content.
* an error message (error structures are discussed in the tutorial)

The representation

A Representation specifies the method’s data model. The data model for the api call uses only elements from the Resource data model. It may be complemented with technical elements that are useful in the context of the method such as page numbers (discussed in the tutorial)

**A representation**

* Contains the data elements that are used to represent a state of a resource for a specific interaction. It is a precise description of the information that can be exchanged between two participants in the context of an interaction. The composition of the data that goes into a representation is similar to the composition of ISO 20022 messages except for the “Document” envelope element which is missing in representations.
* has a very precise scope, which means very little context needs to be added to the elements.
* is a composition of elements (MessageElements) that are collected from the resource(s) from which it is derived.

Type and structure of request messages

The type of message describes what kind of action has to be processed on a given resource (see the tutorial for additional information on the use of the different HTTP methods).

The message must specify the chain of resources that are to be used (resource path). The path provided by the URL of the request lists the type and the identifier of each relevant resource.

**Note**

If an API has too many actions, then that’s an indication that either it was designed with an RPC viewpoint rather than using RESTful principles, or that the API in question is naturally a better fit for an RPC type model.

The messagedefinition identifier

The ISO 20022 MessageDefinitionIdentifier is used to uniquely identify an API method within ISO 20022. It identifies a MessageDefinition which may be used in several MessageTransmissions – in both requests and responses to a web service.
The MessageChoreography defines which message types may be used in each API method.

The API message identifier maps onto the MessageDefinitionIdentifier as follows:

|  |  |  |
| --- | --- | --- |
| **MessageDefinitionIdentifier** | **API message identifier** | **notes** |
| businessArea | BusinessArea |  |
| messageFunctionality | API Method/State  | Alphanumeric. First character must be ‘a’ |
| flavour | Request or Response | ‘001’ or ‘002’ |
| version | version |  |

**Example**



GetPaymentTransactions has ref ‘a03’.
The request has ref ‘001’.
The response has ref ‘002’.
The version is ‘02’.

### Mapping Rules

####  RepositoryConcept

RepositoryConcepts (MessageDefinition, etc.) are only converted into JSON schemas (or components thereof) if their RepositoryConcept.RegistrationStatus is one of

* Registered
* Provisionally Registered (Draft Schemas)

#### MessageDefinition

MessageDefinition is transformed into an Object with following content:

* JSON value pair with Name "$schema" and Value “[http://json-schema.org/draft-04/schema#](http://json-schema.org/draft-04/schema)”
* JSON value pair with Name "type" and Value “object”
* JSON value pair with Name "additionalProperties" and Value “false”
* JSON Object "properties" with following content:
	+ JSON object "@xmlns" with following content:
		- JSON value pair with Name "default" and as Value the concatenation of “urn:iso:std:iso:20022:tech:json:” with the MessageDefinitionIdentifier.
	+ JSON object with Name the value of MessageDefinition.Name but without the Version, converted to “snake case”, and with following content:
		- JSON value pair with name "$ref" and as Value the concatenation of "#/definitions/” and MessageDefinition.Name
* JSON Object "definitions" comprising the comma separated definitions of the rest of the message.

MessageDefinition’s MessageBuildingBlocks are transformed into a MessageComponents. See 13.1.2.4 below.

**Example**

{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "additionalProperties": false,
 "properties": {
 "@xmlns": {
 "default": "urn:iso:std:iso:20022:tech:json:acmt.002.001.07"
 },
 "account\_details\_confirmation": {
 "$ref": "#/definitions/AccountDetailsConfirmationV07"
 }
 },
 "definitions": {
 "AccountDetailsConfirmationV07": {

#### MessageBuildingBlock

See “MessageElement

MessageElement is typed by a MessageComponentType”

#### MessageComponent

A MessageComponent is transformed into a JSON object with following characteristics:

* MessageComponent.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “object”
* JSON value pair with Name "additionalProperties" and Value “false”
* JSON Object "properties" containing 1 or more MessageElements (see 13.1.2.6 )
* JSON value pair with Name "required" and Value the array containing only the mandatory elements of the object

**Example**

JSON Object Account20 has two elements “identification” and “account\_servicer” whereby “account\_servicer” is mandatory.

****

#### ChoiceComponent

A ChoiceComponent is transformed into a JSON object with following characteristics:

* MessageComponent.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “object”
* JSON value pair with Name "additionalProperties" and Value “false”
* JSON Object "properties" containing 1 or more MessageElements (see below on how MessageElements is transformed)
* JSON value pair with Name "oneOf" and Value the array containing all the elements of the ChoiceComponent whereby each element is a JSON value pair with Name "required" and Value an array containing the MessageElement.Name.

**Example**

****

#### MessageElement

##### MessageElement is typed by a MessageComponentType

A MessageElement is transformed into a JSON object (if the MessageElement is not an array) or a JSON array (if the MessageElement is an array), with following characteristics:

* MessageElement.Name is the name of the JSON object or the JSON array.
* JSON value pair with Name "type" and Value “object”
* JSON value pair with Name "additionalProperties" and Value “false”
* JSON Object "properties" with following content:
	+ MessageElement Type is the JSON value pair with name "$ref" and as Value the concatenation of "#/definitions/” with MessageComponentType.Name

**Example**

other\_account\_selection\_criteria typed by InvestmentAccount64



##### MessageElement is typed by a DataType

A MessageElement is transformed into a JSON object (if the MessageElement is not an array) or a JSON array (if the MessageElement is an array), with following characteristics:

* MessageElement.Name is the name of the JSON object or the JSON array.
* JSON value pair with Name "type" and Value “string”
* JSON value pair with name "$ref" and as Value the concatenation of "#/definitions/” with Datatype.Name

**Example**

****

##### MessageElement is an array

A MessageElement is transformed into a JSON array with following characteristics:

* MessageElement.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “array”
* Optional[[12]](#footnote-12) JSON value pair with name "minItems" containing as Value the minimal number of occurrences
* Optional[[13]](#footnote-13) JSON value pair with name "maxItems" containing as Value the maximum number of occurrences
* JSON Object "items" with following content:
	+ MessageElement Type is the JSON value pair with name "$ref" and as Value the concatenation of "#/definitions/” with Datatype.Name

**Example**

Below example shows the element “cash\_settlement” may occur max 8 times:



##### ExternalSchema

An ExternalSchema is transformed into a JSON object with following characteristics:

* ExternalSchema.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “object”

**Example**



#### ISO 20022 DataType transformation to JSON Schema

##### General

There are two kinds of Datatypes, JSON Datatypes and user-defined DataTypes, each with their own set of transformation rules. This covers the user-defined DataTypes.

NOTE See Figure 15 and Figure 7 in ISO 20022-1.

###### DataType Amount

CurrencyIdentifierSet is not empty

DataType Amount is transformed into two JSON objects.

A JSON object with following characteristics:

* Amount.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “object”
* JSON Object "properties" with following content:
	+ JSON value pair with Name “$” and value a JSON object with following values:
		- JSON value pair with Name "type" and Value “string”
		- If Amount. totalDigits is not empty: JSON value pair with Name "maxLength" and Value the value of datatype Amount’s Property totalDigits.
	+ JSON value pair with the Name of the Property CurrencyIdentifierSet.Name and value a JSON object with following value:
		- JSON value pair with name "$ref" and as Value the concatenation of "#/definitions/” with CurrencyIdentifierSet.Type

A JSON object with following characteristics:

* CurrencyIdentifier.Type is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”
* If Property Pattern of CurrencyIdentifierSet’s Type is not empty: JSON value pair with Name "pattern" and Value the content of that Property.

Example

Below example shows element “$” to contain an amount of maximum 19 characters and “currency” to contain a currency code constrained by a pattern.

****

****

CurrencyIdentifierSet is empty

DataType Amount is transformed into a JSON object with following characteristics:

* Amount.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”
* If Amount.totalDigits is not empty: JSON value pair with Name "maxLength" and Value the content of datatype Amount’s Property totalDigits.

Example

Below example shows an amount of maximum 19 characters.

****

###### DataType CodeSet

DataType CodeSet is transformed into a JSON object with following characteristics:

* CodeSet.Name is the name of the JSON object.
* If CodeSet.property is not empty: JSON value pair with Name "pattern" and Value the content of datatype CodeSet’s Property pattern
* JSON value pair with Name "type" and Value “string”
* JSON value pair with Name "enum" and Value the array containing all the CodeSetLiteral values of the CodeSet.

Example

Below example shows the codeSet “SettlementMethod1Code” containing codes INDA, INGA, COVE and CLRG:



###### DataType Text

DataType Text is transformed into a JSON object with following characteristics:

* Text.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”
* If Property minLength is not emply, JSON value pair with Name "minLength" and Value the content of Property minLength.
* If Property maxLength is not emply, JSON value pair with Name "maxLength" and Value the content of Property maxLength.
* If Property length is not emply, both JSON value pair with Name "minLength" and Value the content of Property length and JSON value pair with Name "maxLength" and Value the content of Property length.
* If Property Pattern is not emply, JSON value pair with Name "pattern" and Value the content of Property Pattern.

Example

Below example shows a datatype that validates the correct syntax for a BIC.



###### DataType Indicator

DataType Indicator is transformed into a JSON object with following characteristics:

* Indicator.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “boolean”

Example

Below example shows datatype YesNoIndicator.



###### DataType Binary

DataType binary is transformed into a JSON object with following characteristics:

* Binary.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”
* If Property minLength is not emply, JSON value pair with Name "minLength" and Value the content of Property minLength.
* If Property maxLength is not emply, JSON value pair with Name "maxLength" and Value the content of Property maxLength.

Example

Below example shows a binary datatype of minimum 1 character and maximum 102400 characters.



###### DataType Quantity

DataType Quantity[[14]](#footnote-14) is transformed into a JSON object with following characteristics:

* Quantity.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”
* If Quantity.totalDigits is not empty: JSON value pair with Name "maxLength" and Value the value of datatype Quantity’s Property totalDigits.

Example

Below example shows a datatype of representation Quantity.



###### DataType DateTime[[15]](#footnote-15)

DataType DateTime is transformed into a JSON object with following characteristics:

* DateTime.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”

Example

Below example shows a datatype of representation DateTime.



###### DataType Identifier

DataType Identifier is transformed into a JSON object with following characteristics:

* Identifier.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”
* If Property Pattern is not emply, JSON value pair with Name "pattern" and Value the content of Property Pattern.

Example

Below example shows a datatype of representation Identifier.



###### DataType Rate

DataType Rate[[16]](#footnote-16) is transformed into a JSON object with following characteristics:

* Rate.Name is the name of the JSON object.
* JSON value pair with Name "type" and Value “string”
* If Rate.totalDigits is not empty: JSON value pair with Name "maxLength" and Value the value of datatype Rate’s Property totalDigits.

A Rate has a “baseValue” property, but this is not represented in the JSON schema.

Example

Below example shows a datatype of representation Rate.



## IMIX

## Any other standards transformation...

# Annex 1: Approach to FX trading

For Maker and Taker API in market, major scenarios of FX trading include RFQ trading, indicative pricing, etc. Based on characteristics of scenarios, we may develop WAPI with applicable technologies and styles.

## RFQ trading

**Workflows:**



RFQ quoting and execution messages are supported on a single RFQ session. An RFQ is initiated by the taker sending a QuoteRequest for a particular instrument, size and optionally currency. The QuoteRequest contains an optional RFQ expiration time. If no expiration time is provided on the QuoteRequest, a default amount of time will be applied. The RFQ is identified by the QuoteReqID, which will appear on all messages related to the RFQ. For disclosed bilateral trading for RFQ, the Liquidity Consumer must have available credit for each LP counterparty LC submitting an RFQ to. The system will submit the QuoteRequests to the Liquidity Providers with available credit for the Liquidity Consumer and that have been selected by the Liquidity Consumer.

**Apply WAPI:**

Usually, interactions between client and server can be considered with two factors: communication model and payload format. There are several common communication models: one request one response (synchronous or asynchronous), one request multiple responses and push. In technical perspective, we may have different technology choices to realize one model, but there is always an optimum solution. For example, before WebSockets, clever engineer used HTTP/1.1 to simulate push model which called Comet framework. But nowadays, we should always use WebSockets except for special reason. Therefore we can design WAPI for RFQ trading as below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Actor | Action | Model | Technology | WAPI Style | URL | Payload |
| Taker | QuoteRequest | 1R1R Sync | HTTP/2.0 | RESTFul | .../rfq/taker/v1/quoterequest | JSON |
| Taker | Quote | 1RMR  | HTTP/2.0 | RESTFul | .../rfq/taker/v1/quoterequest | JSON |
| Taker | NOS | 1R1R Sync | HTTP/2.0 | RESTFul | .../rfq/taker/v1/nos | JSON |
| Taker | ER | 1RMR | HTTP/2.0 | RESTFul | .../rfq/taker/v1/nos | JSON |
| Maker | QuoteRequest | Push | WebSockets | Server Push | .../rfq/maker/v1/quoterequest | JSON |
| Maker | Quote | 1R1R Sync | HTTP/2.0 | RESTFul | .../rfq/maker/v1/quote | JSON |
| Maker | NOS | Push | WebSockets | Server Push | .../rfq/maker/v1/nos | JSON |
| Maker | ER | 1R1R Sync | HTTP/2.0 | RESTFul | .../rfq/maker/v1/er | JSON |

Note:

* 1R1R Sync - One Request One Response Synchronous
* 1RMR - One Request Multiple Responses

## Indicative Pricing

In addition to the tradable pricing segments, indicative pricing will be set up on its own segment. Indicative prices are provided in exactly the same way that tradable prices are provided, and are indicative solely.

This section describes the application level of pricing indicative rate workflows supported and exchanged between the market maker and the system:

|  |  |  |  |
| --- | --- | --- | --- |
| Workflow | Pricing/ Trading | Direction | Messages |
| Start/Stop a market data subscription  | Pricing | To Maker | MarketDataRequest |
| Reject a MarketDataRequest | Pricing | From Maker | MarketDataRequestReject |
| Full view of price book | Pricing | From Maker | MarketDataSnapshotFullRefresh |

**Workflows:**



When the Liquidity Provider logs on, the trading platform will subscribe for indicative rates whilst providing the trading band.

The Liquidity Provider will feed the indicative rates to the system. If the quote does not satisfy the system checks(Price band check), the quote will be dropped.

**Apply WAPI:**

Overall solution is much similar to RFQ trading. The only difference is as market data is time sensitive, so maker is continually sending data to system regardless success or not. Based on this, WAPI design for indicative pricing could be as below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Actor | Action | Model | Technology | WAPI Style | URL | Payload |
| Taker | Indicative Rates | Push | WebSockets | Server Push | .../ip/taker/v1/ir | JSON |
| Maker | MarketData | 1R1R Async | HTTP/2.0 | RESTFul | .../ip/taker/v1/marketdata | JSON |

1. In case of a replacement, the IDs of the concerned resources must be mentioned in the payload. [↑](#footnote-ref-1)
2. A bulk patch is a dangerous and not recommended way to bulk update resources. [↑](#footnote-ref-2)
3. A ‘POST’ is a creation of resource that does not exist yet. Doing a POST on an already existing resource is thus considered an error. Except if the POST is overridden to be a PUT or PATCH [↑](#footnote-ref-3)
4. It is bad practice to have the client create the resource ID. It should always be the server that creates it. [↑](#footnote-ref-4)
5. https://tools.ietf.org/html/bcp195 [↑](#footnote-ref-5)
6. https://tools.ietf.org/html/rfc5246 [↑](#footnote-ref-6)
7. https://tools.ietf.org/html/rfc6125 [↑](#footnote-ref-7)
8. https://tools.ietf.org/html/rfc6749 [↑](#footnote-ref-8)
9. https://tools.ietf.org/html/rfc7515#appendix-F [↑](#footnote-ref-9)
10. https://tools.ietf.org/html/rfc7517 [↑](#footnote-ref-10)
11. http://openid.net/specs/openid-connect-core-1\_0.html#RotateSigKeys [↑](#footnote-ref-11)
12. Default value is zero in which case minItems may be omitted. [↑](#footnote-ref-12)
13. The default is that the array is unbounded in which case maxItems may be omitted. [↑](#footnote-ref-13)
14. Technically, a Quantity can have a “unit” property, but at present none of the Quantity definitions in the ISO 20022 repository use that attribute. [↑](#footnote-ref-14)
15. The same transformation applies for Date and Time. [↑](#footnote-ref-15)
16. [↑](#footnote-ref-16)