TECHNICAL SPECIFICATION

ISO/TS 23029

First edition
2020-02

Web-service-based application programming interface (WAPI) in financial services

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

Reference number
ISO/TS 23029:2020(E)

© ISO 2020

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

COPYRIGHT PROTECTED DOCUMENT

© ISO 2020

All rights reserved. Unless otherwise specified, or required in the context of its implementation, 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
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword ... **v**

Introduction ... **vi**

1 Scope ... **1**

2 Normative references ... **1**

3 Terms and definitions ... **1**

4 Design principles ... **3**
- 4.1 General ... 3
- 4.2 Standards compatibility ... 3
- 4.3 Extensibility ... 3
- 4.4 Non-repudiation ... 3
- 4.5 Web resource unique identifiers (ID fields) ... 3
- 4.6 Idempotency ... 3
- 4.7 States ... 3

5 Related technology ... **3**
- 5.1 General ... 3
- 5.2 Representational state transfer (REST) and simple object access protocol (SOAP) ... 3
  - 5.2.1 General ... 3
  - 5.2.2 REST ... 4
  - 5.2.3 SOAP ... 4
- 5.3 WebSocket and Webhook ... 5
  - 5.3.1 General ... 5
  - 5.3.2 WebSocket ... 5
  - 5.3.3 Webhook ... 5
- 5.4 HTTPS ... 6
- 5.5 JSON and XML ... 6
  - 5.5.1 General ... 6
  - 5.5.2 JSON ... 6
  - 5.5.3 XML ... 6
- 5.6 Content negotiation ... 7
- 5.7 RESTful API description languages ... 7

6 Naming conventions ... **7**

7 Resource path ... **8**
- 7.1 General ... 8
- 7.2 Resource hops ... 8
- 7.3 Single resource versus collections of resources ... 9

8 WAPI styles ... **9**
- 8.1 General ... 9
- 8.2 REST ... 10
  - 8.2.1 General ... 10
  - 8.2.2 Uniform interface ... 11
  - 8.2.3 Apply the standard HTTP methods ... 12
  - 8.2.4 Stateless sessions ... 13
  - 8.2.5 Idempotency ... 13
  - 8.2.6 Composition of the URI ... 14
  - 8.2.7 Handling associations between resources ... 14
  - 8.2.8 Request parameter usage ... 14
  - 8.2.9 Post usage ... 17
  - 8.2.10 The response ... 18
- 8.3 Asynchronous messaging and server push ... 21
  - 8.3.1 Bidirectional communication model ... 22
  - 8.3.2 Message subscription ... 23

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

8.3.3 Message publish ... 24
9 Data payload syntax ... 24
9.1 JSON ... 24
9.1.1 General ... 24
9.1.2 Syntax and structure ... 24
9.1.3 Data types ... 26
9.2 XML ... 26
9.2.1 General ... 26
9.2.2 Syntax and structure ... 26
9.2.3 Data types ... 28
10 Security and authentication ... 30
10.1 General ... 30
10.2 TLS ... 30
10.2.1 Certificate issuance and verification ... 31
10.3 Application and access layer security ... 32
10.3.1 Introduction ... 32
10.3.2 Overview of the OAuth 2.0 protocol ... 33
10.4 Read-only security profile ... 33
10.5 Read and write security profile ... 33
10.6 Message level integrity, source authentication and non-repudiation ... 34
10.6.1 General ... 34
10.6.2 Signing HTTP requests and responses ... 34
10.6.3 Signing JSON Payload ... 34
10.6.4 HTTP signature ... 35
10.7 Message level encryption ... 35
10.8 Version control ... 35
11 Use cases ... 36
11.1 ISO 20022 Web services ... 36
11.1.1 Introduction ... 36
11.1.2 Modelling guidelines ... 36
11.2 Mapping rules ... 39
11.2.1 RepositoryConcept ... 39
11.2.2 MessageDefinition ... 39
11.2.3 MessageBuildingBlock ... 40
11.2.4 MessageComponent ... 40
11.2.5 ChoiceComponent ... 41
11.2.6 MessageElement ... 42
11.2.7 ISO 20022 DataType transformation to JSON Schema ... 43
Annex A (informative) **Approach to FX trading ... 49**
Bibliography ... 52

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

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).

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).

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

For an explanation of 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 www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 68, *Financial services*, Subcommittee SC 9, *Information exchange for financial services*.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

Introduction

0.1 Opening comments

The purpose of this document is to help implementers in the financial services industry define the framework, function and protocols for an application programming interface (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 – market-, corporate- and regulatory-driven – from different communities and jurisdictions for financial institutions to share data and enable functionality, such as 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 is currently 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 document has been developed in response to meet these current and foreseen requirements that exist in the market. This document does not specify implementations of APIs, but instead takes an international view and references, as appropriate, specific implementation scenarios for illustrative purposes for guidance.

0.2 How to approach this document

This document should be approached as a best-practice framework for developing APIs in financial services. In this sense, some aspects of the document are more mature than others. The document is prescriptive where there is room to be. Where areas are less mature, commentary on best practice has been provided and the considerations set out.

Broadly speaking, this document adopts the following logic and order:

— Clause 3: terms and definitions used in the document;

— Clause 4: the initial considerations for the design of the API;

— Clause 5: overview and commentary on the different technology options;

— Clause 8: specific guidance on APIs under WAPI technical specification.

In Annex A, we set out an example of how to approach the document depending on a specific business area/desired API functionality.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

Web-service-based application programming interface (WAPI) in financial services

1 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, 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;

— defines architectural usage beyond query/response asynchronous messaging towards publish/subscribe to support advanced and existing business models.

This document does not include:

— a specific technical specification of an API implementation in financial services;

— the development of JSON APIs based on the ISO 20022 specific message formats, such as PAIN, CAMT and PACS;

— a technical specification that is defined or determined by specific legal frameworks.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
application programming interface
API
set of well-defined methods, functions, protocols, routines or commands which application software uses with facilities of programming languages to invoke services

Note 1 to entry: An API is available for different types of software, including Web-based system/ecosystem.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

3.2
idempotency
idempotence
operation feature which means there is no additional effect if the operation is called more than once

Note 1 to entry: Idempotent operations are often used to design a Web-based system since it is hard to restrict redundancy access.

3.3
JavaScript object notation
JSON
open and text-based exchange format

Note 1 to entry: Data transmitted in JSON formats make it easy to read and write (for humans), parse and generate (for computers).

3.4
representational state transfer
REST
RESTful
software architectural style for distributed hypermedia systems Note 1 to entry: It was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.

3.5
resource hop
resource locator built on resource type and identifier, mandatory except for the last resource

Note 1 to entry: A resource path is a chain of one or more resource hops.

3.6
RESTful API description language
language designed to provide a structured description of a RESTful Web API that is useful both to humans and for automated machine processing

3.7
simple object access protocol
SOAP
messaging protocol specification for exchanging structured information in the implementation of Web services in computer networks

3.8
WebSocket
protocol which 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

3.9
Webhook
user-defined HTTP call-backs triggered by an event such as pushing code to a repository or a comment being posted to a blog

3.10
XML
eXtensible Markup Language, a data format standard created by W3C

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

4 Design principles

4.1 General

This clause covers absolute principles for developing a Web API. It covers both standalone design principles that shall be considered upfront when developing an API in financial services and principles that apply to specific types of API and are discussed in more detail in Clause 8.

4.2 Standards compatibility

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

4.3 Extensibility

Where possible, design decisions for the API ecosystem should be 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.

4.4 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.

4.5 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.

4.6 Idempotency

Idempotency shall be considered upfront for a Web API. This is covered in Clause 8.

4.7 States

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

5 Related technology

5.1 General

This clause talks through related technology and provides some context on its use on APIs in financial services. It does not include terms and definitions, as these are defined in Clause 3, and does not provide specific guidance on how to approach the technology, as this is covered in Clause 8.

5.2 Representational state transfer (REST) and simple object access protocol (SOAP)

5.2.1 General

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

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

5.2.2 REST

REST or RESTful Web services provide the way to transfer the representational state of a resource and to specify the action to be processed on this resource (see Clause 3). REST is the most popular to build a Web service or define Web APIs due to its simplicity and compatibility with existing Internet standards such as HTTP.

APIs typically use REST as it is more efficient (can use smaller message formats) and requires less extensive processing. In most financial scenarios using RESTful APIs, the end user sends a message to the server, which 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[1]:

— 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 shall be independent, that is, every request shall contain all the necessary information so that the server can understand it and process it accordingly. In this case, the server shall not hold any information about the client state. Any information status shall stay on client – such as sessions.

— Cacheable: because many clients access the same server, and often request the same resources, it is necessary for these responses to 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 they are 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 when 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 faster or more efficient.

5.2.3 SOAP

SOAP is a messaging protocol specification for exchanging structured information in the implementation of Web services in computer networks. Its purpose is to induce extensibility, neutrality and independence. It uses XML information set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

SOAP allows processes running on disparate operating systems (such as Windows and Linux) to communicate using 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

1) R.T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

© ISO 2020 – All rights reserved

5.3 WebSocket and Webhook

5.3.1 General

WebSocket and Webhook protocols exist for scenarios that REST cannot cater for, such as supporting unsolicited responses, specifically:

— client-to-client architectures (Webhook may be used);

— client-to-server architectures (WebSocket may be used);

— any kind of publish/subscribe implementation.

5.3.2 WebSocket

As detailed in RFC6455[2)], the WebSocket 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 the case of massive data transporting or events driven by the server, WebSocket performs better than HTTP, for example 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. Transport Layer Security (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 JavaScript object notation (JSON) or XML, and binary messages, such as simple binary encoding. More than one subprotocol may be supported by a peer, if desired. Other subprotocols may be registered with IANA, for example 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 an event-driven protocol rather than a request/response protocol. Correlation of events is performed at the application layer with transaction IDs and the like.

5.3.3 Webhook

Webhook are sometimes referred to as reverse APIs. 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

2) https://tools.ietf.org/html/rfc6455

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

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.

5.4 HTTPS

HTTP is 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 TLS 1.2 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 become popular in the next few years.

HTTPS (HTTP over SSL or HTTP Secure) is the use of Secure Socket Layer (SSL) or 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

5.5 JSON and XML

5.5.1 General

In Web communication scenarios, the two most used structured data formats for data exchanging are JSON [RFC8259][3] and XML[4].

5.5.2 JSON

JSON is a lightweight, text-based data format widely used in Web-based communication, including as a replacement for XML.

Compared with XML format, JSON has several advantages:

— less verbose than XML, as XML necessitates opening and closing tags, and JSON has concise syntax like using name/value pairs and separated with "{" and "}";

— allows for direct mapping onto the corresponding data structures in the host language, corresponding directly to the object of JavaScript.

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, reducing the consumption of network bandwidth and making it easy to maintain.

5.5.3 XML

XML is a data format standard created by W3C. There are several advantages that XML has over JSON:

— can put metadata into the tags in the form of attributes;

— most browsers render XML in a highly readable and organized way;

— has the concept of schema, supporting strong typing and user-defined types and allowing validation;

3) https://tools.ietf.org/html/rfc8259

4) https://www.w3.org/standards/xml/core

© ISO 2020 – All rights reserved

— supports comments.

In summary, JSON is a lightweight solution, easier for a programmer to use; XML is a heavier solution and, while more functional, is restrictive. Each solution applies to a specific financial Web service.

5.6 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 shall look at the Accept header to determine whether or not it is acceptable. If it is not acceptable then the server shall return an HTTP 406 Not Acceptable status code. If the request is acceptable, the server shall 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 also format the response. 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.

5.7 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 have been OpenAPI Specification, RAML and API Blueprint.

— OpenAPI Specification: originally known as the Swagger Specification, this 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 in 2016. The latest version is 3.0. https://swagger.io/specification/

— RAML: 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 the 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. https://raml.org/

— API Blueprint: 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. https://apiblueprint.org/

6 Naming conventions

Table 1 lists the applicable character case conventions that WAPIs should follow.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

Table 1 — Applicable character case conventions that WAPIs should follow

API element	Rules		Example
HTTP headers	Although RFC 2616 specifies that HTTP headers are case insensitive, it should be best practice to adopt, within an API specification, the same character case for all header definitions [RFC 6648] 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 UpperCamelCase used to define the data type e.g. "country": { "title": "Ctry, Country", "description": "Country of the address.", "$ref": "#/definitions/CountryCode"		CountryCode

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

7 Resource path

7.1 General

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

7.2 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	orders payment-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, shall be identified	000235698741 pmtrqst1652

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

EXAMPLE

Resource hop
/orders/000235698741
/payment-requests/pmtrqst1652

7.3 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 resource 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

8 WAPI styles

8.1 General

This clause explains how parts of an API should be designed and implemented. As much as 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

8.2 REST

8.2.1 General

— It is recommended that financial scenarios which require sending a message with the reply in a short time are built based on RESTful APIs; for example, placing an order in a trade system.

— REST is an architectural style for developing Web services which defines a set of constraints, including client–server architecture, uniform interface and stateless.

— Constraints most relevant to WAPI are the uniform interface and the stateless sessions:

  — uniform interface: 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 the HTTP methods GET, POST, PUT and DELETE to operate resources.

  — stateless: one client can send multiple requests to the server; however, each of them shall be independent, i.e. each request shall contain all the necessary information so that the server can understand it and process it accordingly. In this case, the server shall not hold any information about the client state. Any information status shall 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 elements 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 perspective, is REST-based interactions happen using constructs that are familiar to anyone who is accustomed to using the Internet's HTTP. In other words, HTTP is recommended to implement RESTful WAPI.

— In summary, the key factors in designing and implementing RESTful WAPI include uniform interface, HTTP methods usage, stateless, idempotency, resources and resource identifiers usage.

Figure 1 shows an example of a RESTful API.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

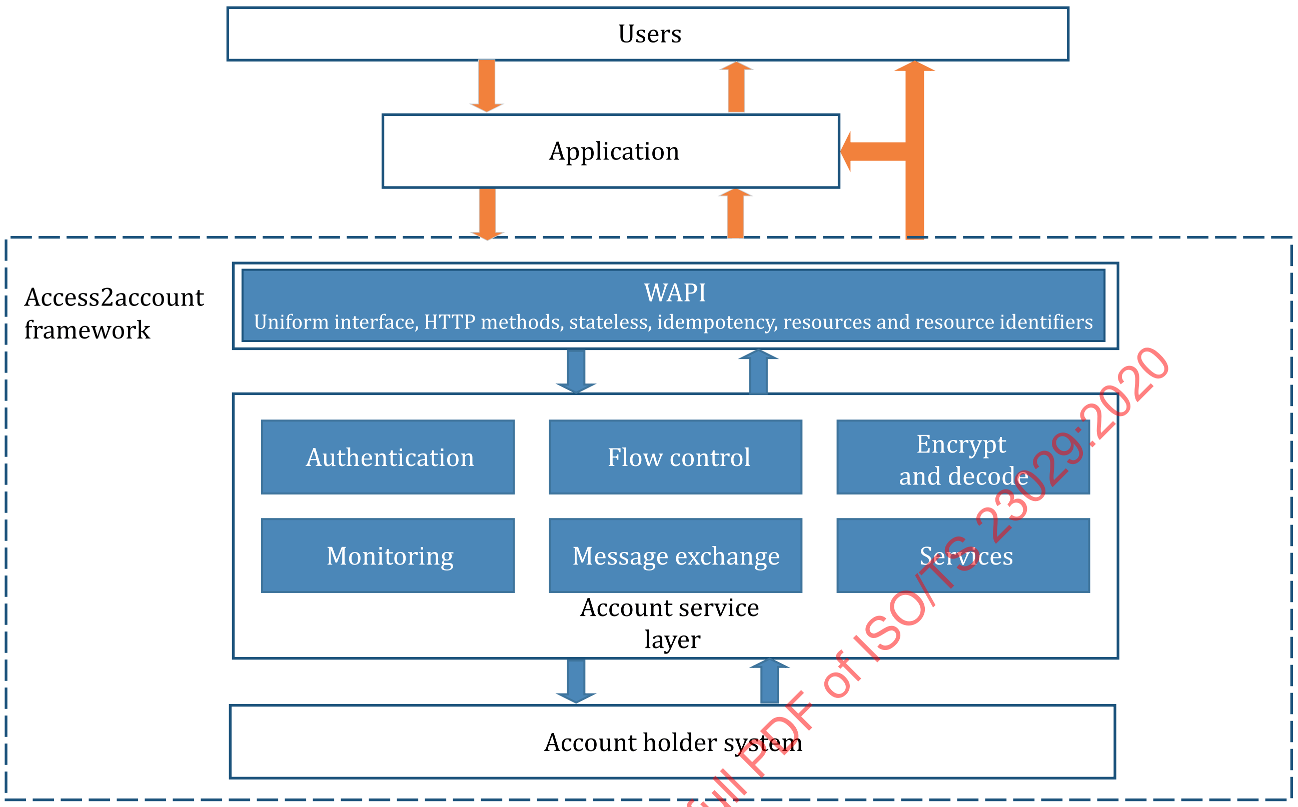

Figure 1 — REST implementation of access to account

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

8.2.2 Uniform interface

8.2.2.1 Resource-based

Requests should use URIs as resource identifiers. The exchanged information should be representations of the resources, expressed in a particular syntax (XML or JSON is recommended), depending on the details of the request and the server implementation.

8.2.2.2 Manipulation of resources through representations

A representation of a resource which a client holds should has enough information to modify or delete the resource on the server if permitted.

8.2.2.3 Self-descriptive messages

Each message should include enough information to describe how to process it. For example, an Internet media type should be used to specify which parser to invoke. Responses should also explicitly indicate their cache ability.

8.2.2.4 Hypermedia as the engine of application state (HATEOAS)

Clients should deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name). Services should deliver state to clients via body content, response codes and response headers.

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 URIs, pointing to three states (Authorised, Rejected and Accepted).

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

— pagination features;

— further manipulations of resources, 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/.

8.2.3 Apply the standard HTTP methods

Usage guidelines are as follows:

— 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, using 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 words, PUT shall be a full resource update, all attribute values in a PUT request shall be sent 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

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[a]	Bulk update orders[b]	Delete all orders
/orders/000235698741	Error[c]	Get the details on this order	If it exists, replace the order, or else create it[d]	If exists, update the order or fail otherwise	Delete this order

[a] In case of a replacement, the IDs of the concerned resources can't be mentioned in the payload.

[b] A bulk patch is a dangerous way to bulk update resources and not recommended.

[c] 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, unless the POST is overridden to be a PUT or PATCH.

[d] It is bad practice to have the client create the resource ID. It should always be the server that creates it.

8.2.4 Stateless sessions

Stateless sessions indicate that the necessary state to handle the request should be defined by the client and contained within the request itself, whether as part of the URI, query-string parameters, body or headers. After the server does its processing, the appropriate state should be communicated back to the client.

This is the opposite of a "session" which maintains state across multiple HTTP requests. In REST, the client shall include all information for the server to fulfil the request, resending state as necessary if that state spans 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, i.e. 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.

8.2.5 Idempotency

Normally, the Web APIs shall not be idempotent for 'read' applications without additional design. It is recommended that Web APIs are idempotent for 'write' applications, based on the following principles:

— it is recommended that the APIs for creating, updating or deleting resources are 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 that the user of an API shall not change the request body while using the same idempotency key. If the API user changes the request body, the API provider shall not modify the end resource;

— an API provider shall treat a request as idempotent if it had received the first request with the same key;

— the API provider shall 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

8.2.6 Composition of the URI

The composition of the URI is detailed in Figure 2.

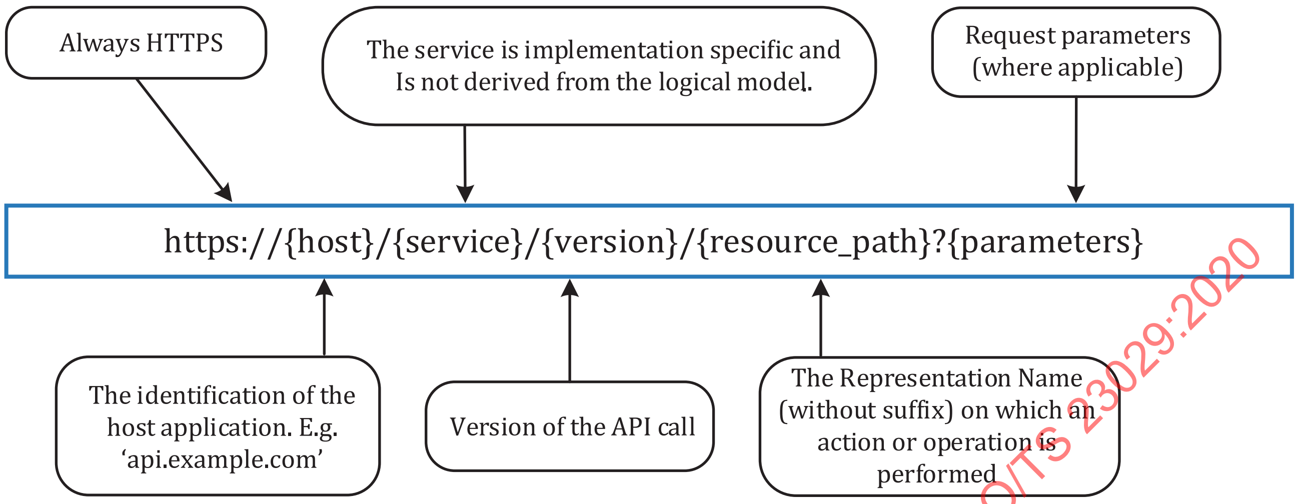
Always HTTPS
The service is implementation specific and Is not derived from the logical model.
Request parameters (where applicable)
https://{host}/{service}/{version}/{resource_path}?{parameters}
The identification of the host application. E.g. 'api.example.com'
Version of the API call
The Representation Name (without suffix) on which an action or operation is performed

Figure 2 — Composition of the URI

The resource_path may include more than one resource where applicable, accompanied by resource IDs. 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
```

8.2.7 Handling associations between resources

Expose associations between resources in the URI path. However, keep 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.

8.2.8 Request parameter usage

8.2.8.1 General

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

— Locators (resource identifiers or a specific action);

— Filters (parameters that provide a search for, sort or limit results);

— Content (data to be stored).

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

There are different placeholders where to put those parameters:

— URI query parameters (the portion of the URI that follows the '?');

— URI paths (the portion of the URI that follows the hostname or 'fully qualified domain name', and that precedes the '?' if present);

— HTTP request body;

— HTTP request headers.

	URI query	URI paths	Request body	Request header
Locators		X		
Filters	X			
State				X
Content			X	

8.2.8.2 State

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

8.2.8.3 Content

Content only belongs in one place, 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.

8.2.8.4 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) needed 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 the use of 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.

8.2.8.5 Attributes and search criteria

Put resource attributes and search criteria behind the '?'.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

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.

8.2.8.6 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.

8.2.8.7 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, such as mobile phone apps.

EXAMPLE

```
GET /bics?fields=institution_name,address.city

Returns

{

    "name"  "My Bank Inc."

    "address" :{

        "city" : "London"

    }

}
```

8.2.8.8 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, and so on.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

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, for example address.city.

8.2.8.9 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, for example, 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 (e.g. 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 to 74. In a numbered list where the first element has index 1, this returns objects 51 to 75.

See 8.2.10.4 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.

8.2.9 Post usage

There are two usages recommended using POST: create resources and 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
```

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

POST is used to create resources with data payload (contains idempotency key) as XML or JSON content request. See Clause 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"

    }

}
```

8.2.10 The response

8.2.10.1 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.

8.2.10.2 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). 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.

8.2.10.3 The error element

It is noted that not all fields, as indicated below, are necessary and some are optional.

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 canonical model for problem details is a JSON object. The 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 provides 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

— "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 shall 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 shall use the same status code in the actual HTTP response, to ensure 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
```

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

```
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": "shall be a positive integer"

    },

    {

        "name": "color",

        "reason": "shall 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] code could be used to encapsulate multiple status messages.

8.2.10.4 Pagination responses

See 8.2.8.9 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. For example in the last page, count can be lower than the value of limit in the request;

— offset to indicate the starting object of the page. This value should be the same as the value of offset in the request.

8.2.10.5 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

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

'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.

8.3 Asynchronous messaging and 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 which allows an application, either client or server, to send out a message which 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, 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 and 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 and 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 include bidirectional communication and the message publish/subscribe model.

WAPI implementation in trading services is detailed in Figure 3.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

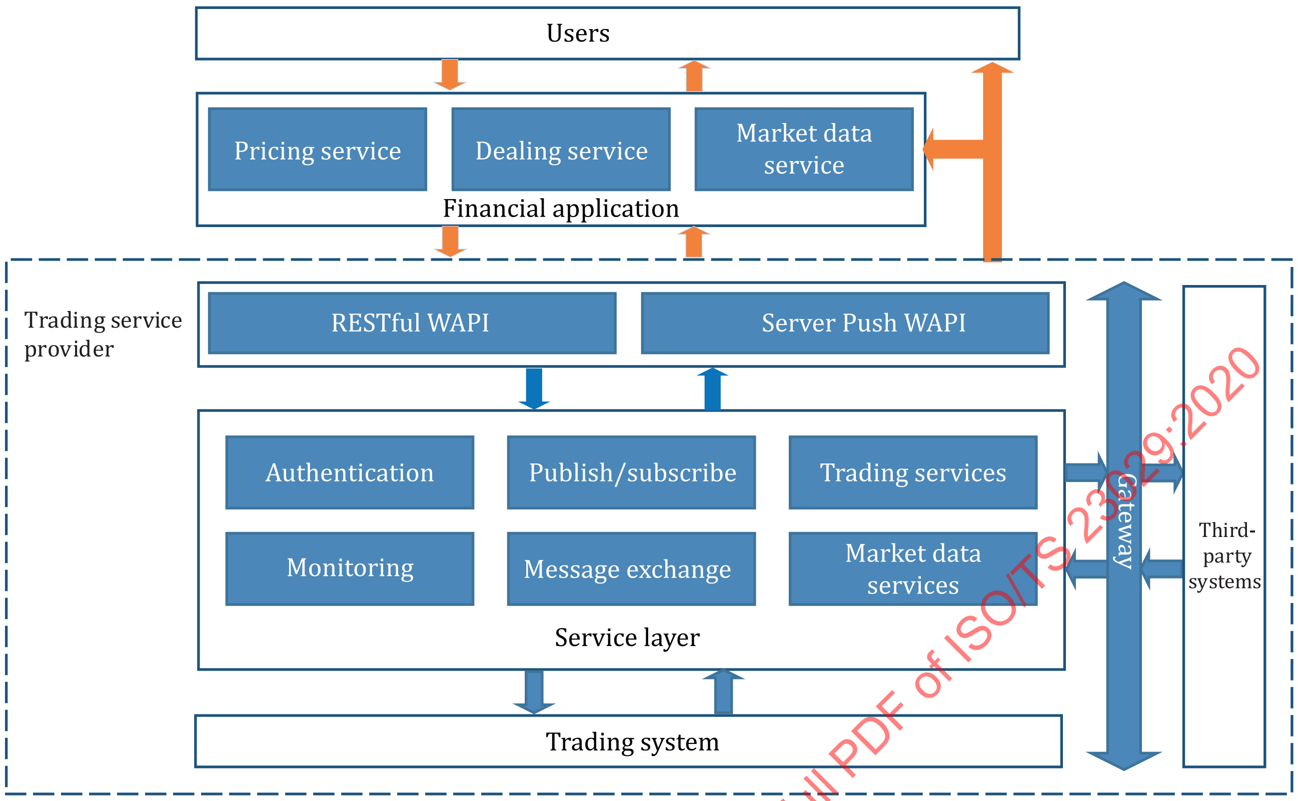

Figure 3 — WAPI implementation in trading services

8.3.1 Bidirectional communication model

As detailed in RFC8030[5], a general model for push services should include three basic actors: a user agent, a push service and an application (server). Figure 4 shows a general model of a push service.

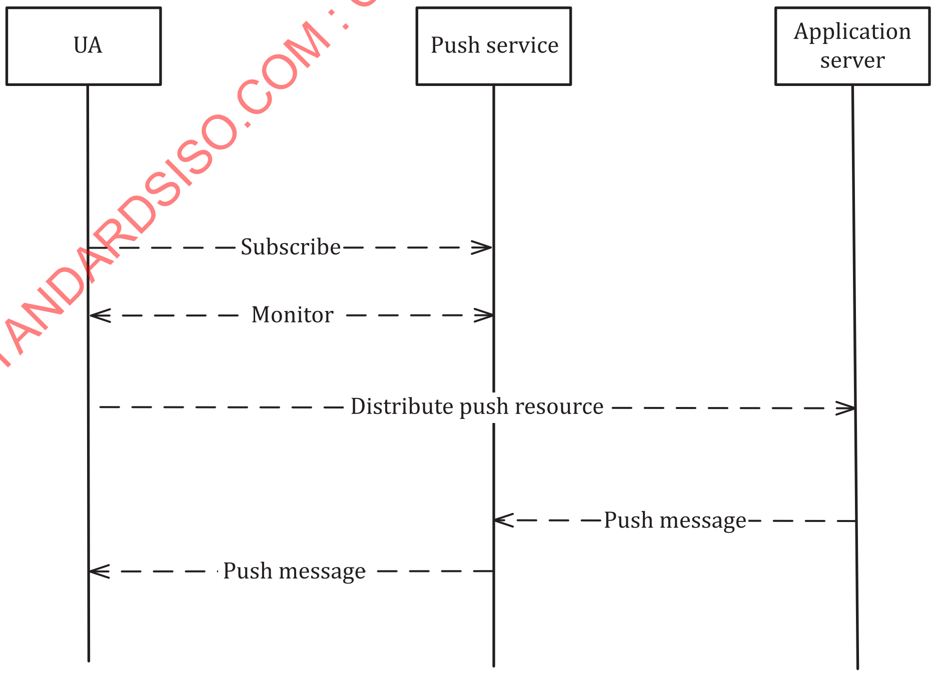

Figure 4 — General model of push service

5) https://tools.ietf.org/html/rfc8030

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

The message subscription is used as follows:

— a new message subscription is created by the user agent and then distributed to its application server at the very beginning of the process;

— a subscription is the basis of all 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;

— a subscription is used to monitor the push service for any incoming message;

— a subscription is modelled as two resources with different capabilities to offer more control for authorization:

  — 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.

A unique subscription should be distributed to each application. In case multiple subscriptions might be created for the same application, or multiple applications could use the same subscription, the sharing subscriptions have security and privacy implications.

Subscriptions should have a limited lifetime. The push service or the user agent can terminate subscriptions at any time. User agents and application servers shall manage changes in the subscription state.

A push service is usually identified as a Web resource, with a URI as follows:

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

EXAMPLE

Push service via WebSocket

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

8.3.2 Message subscription

Beyond the technology this allows customer self-service and adjustment of any kind of subscription. As messaging may still be used for larger data content, this subscription generates a message in another format over another channel. For example, 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 follows:

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

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

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.

8.3.3 Message publish

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

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.

9 Data payload syntax

9.1 JSON

9.1.1 General

As described in 5.5, JSON is a lightweight solution of format for sharing data. JSON is derived from the JavaScript programming language, but available for use by many languages including Python, Ruby, PHP, and Java. Beside JSON, YAML is also an opportunity and better for human readability.

Data payload of financial APIs represented by JSON should follow the following rules.

9.1.2 Syntax and structure

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

JSON syntax is derived from JSON 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": {
```

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

```
            "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 consist 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

9.1.3 Data types

JSON values are found to the right of the colon. At the granular level, these need to be one of six 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.

```
"IBAN2007Identifier": {

    "type": "string",

    "pattern": "^[A-Z]{2,2}[0-9]{2,2}[a-zA-Z0-9]{1,30}$"

}
```

There are some solutions to validate JSON object, for example JSON schema.

9.2 XML

9.2.1 General

As described in 5.5, XML is a heavy solution of format for sharing data compared with JSON. The latest versions of XML published by W3C are XML1.0[6] and XML1.1[7].

Data payload of financial APIs represented by JSON should follow the following rules.

9.2.2 Syntax and structure

XML uses a self-descriptive syntax. The main syntax rules to build an XML document are as follows:

— shall contain one root element;

— optional contain XML prolog. If prolog exists, it shall come first in the document:

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

— UTF-8 is the default character encoding;

— an XML element is everything from (including) the element's start tag to (including) the element's end tag:

  <price>100.35</price>;

— an element may contain text, attributes, other elements or a mix of these;

— an element shall have a closing tag;

6) https://www.w3.org/TR/2008/REC-xml-20081126/s

7) https://www.w3.org/TR/2006/REC-xml11-20060816/

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

— XML tags are case sensitive

— an element may have attributes in name/value pairs just like in HTML. The attribute values shall be quoted:

<price required="Y">100.35</price>;

— elements shall be properly nested:

<quote><price required="Y">100.35</price></quote>;

— empty elements are allowed:

<element></element>;

— an element shall follow these naming rules:

  — element names are case-sensitive;

  — element names shall start with a letter or underscore;

  — element names cannot start with the letters xml, XML or Xml;

  — element names can contain letters, digits, hyphens, underscores and periods;

  — element names cannot contain spaces;

— naming conventions used in financial APIs should follow Clause 8;

— syntax for writing comments is similar in HTML:

<!-- This is a comment -->

— support namespace to avoid element name conflicts:

<fx:quote><price required="Y">100.35</price></fx:quote>

<fi:quote><price required="Y">100.35</price></fi:quote>

Following the above rules, an XML document forms a tree structure that starts at the root element.

EXAMPLE

```
<?xml version = "1.0"?>

<fx:trade>

    <fx:quote>

        <price required="Y">100.35</price>

    </fx:quote>

    <fx:product>100001</fx:product>

</fx:trade>
```

An example of the tree structure is shown in Figure 5.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

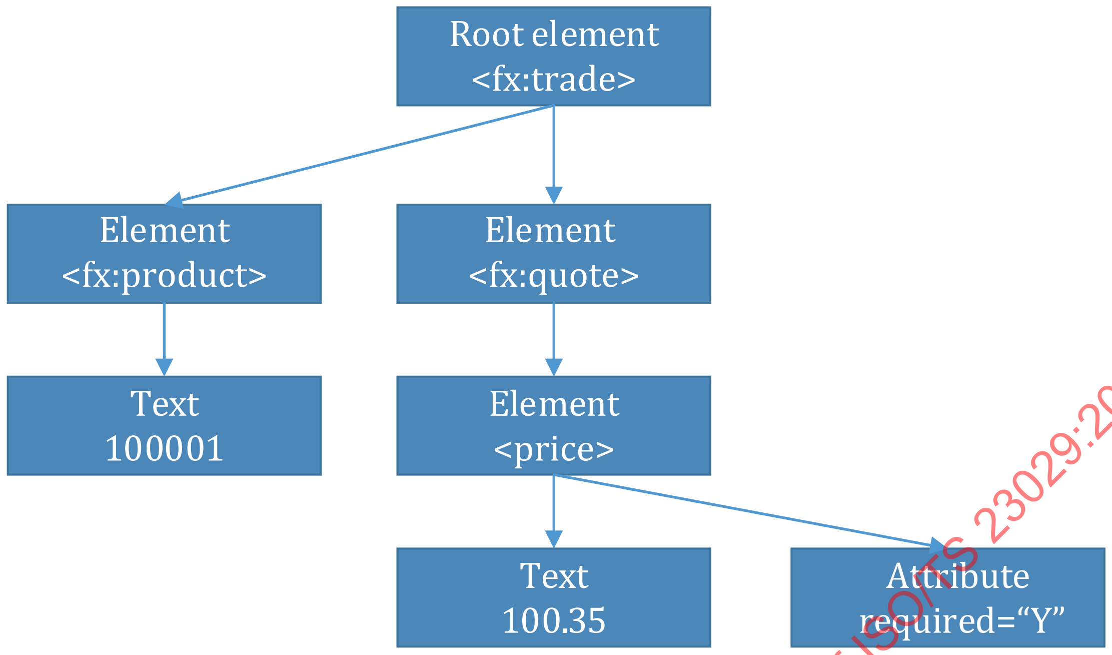

Figure 5 — XML tree structure

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

9.2.3 Data types

Table 2 lists primitive XML schema data types, facets that can be applied to the data type, and a description of the data type.

Table 2 — Primitive XML scheme data types

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.

© ISO 2020 – All rights reserved

Table 2 *(continued)*

Data type	Facets	Description
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 ss represent 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 shall 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**

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

Table 2 *(continued)*

Data type	Facets	Description
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 shall be an NCName. The prefix shall 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.

10 Security and authentication

10.1 General

Web-based APIs implemented in the financial services industry are often of high value and need to be appropriately secured.

This clause 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.

10.2 TLS

All WAPI endpoints shall be protected by 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 TLS in BCP195[8) shall be followed, with the following additional requirements:

— TLS version 1.2[9) or later shall be used for all communications;

— a TLS server certificate check shall be performed, as per RFC6125[10).

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.

8) https://tools.ietf.org/html/bcp195

9) https://tools.ietf.org/html/rfc5246

10) https://tools.ietf.org/html/rfc6125

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

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).

10.2.1 Certificate issuance and verification

10.2.1.1 General

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.

10.2.1.2 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 at 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 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.

10.2.1.3 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.

10.2.1.4 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

10.3 Application and access layer security

10.3.1 Introduction

As detailed in RFC6749[11], 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 a 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, consisting 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 document there are two 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 and write API security profile: to be implemented for APIs where write access is performed, for example the initiation of payments.

The Read and 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

11) https://tools.ietf.org/html/rfc6749

10.3.2 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.

Financial APIs that involve exchange between institutions should use OAUTH 2.0 client credentials 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 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.

10.4 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.

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.

Counterparties who adopt an API that uses OAUTH 2.0 shall implement and execute a conformance test.

10.5 Read and write security profile

Implementers of WAPI for read and write use cases should use the OpenID Foundation's Financial-grade API - Part 2: 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

10.6 Message level integrity, source authentication and non-repudiation

10.6.1 General

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.

10.6.2 Signing HTTP requests and 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

10.6.3 Signing JSON Payload

The application layer security profiles defined in this document 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 its 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 RFC7515 Appendix F[12]) this is not recommended as there is a greater chance of implementation errors. JWTs can be signed

12) https://tools.ietf.org/html/rfc7515#appendix-F

© ISO 2020 – All rights reserved

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 document we recommend the use of JSON Web Key[13] endpoints as described in OpenID Connect Core, 10.1.1[14]. This will allow both servers and clients to rotate keys according to their key management policies without any manual changes required to API implementations.

10.6.4 HTTP signature

WARNING — As noted previously there are many issues with general purpose HTTP signing. However, if implementers have a requirement for such a feature they may want to consider the following mechanism. This mechanism is not an international standard but has been used by other financial API initiatives.

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 and some HTTP headers.

To implement this mechanism, the WAPI client shall:

— 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 shall 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.

10.7 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.

10.8 Version control

An API shall 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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

13) https://tools.ietf.org/html/rfc7517

14) http://openid.net/specs/openid-connect-core-1_0.html#RotateSigKeys

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. The following 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.

11 Use cases

11.1 ISO 20022 Web services

11.1.1 Introduction

The premise for defining a modelling methodology for ISO 20022 REST APIs is to start from 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 6 shows leveraging components in the existing ISO 20022 methodology.

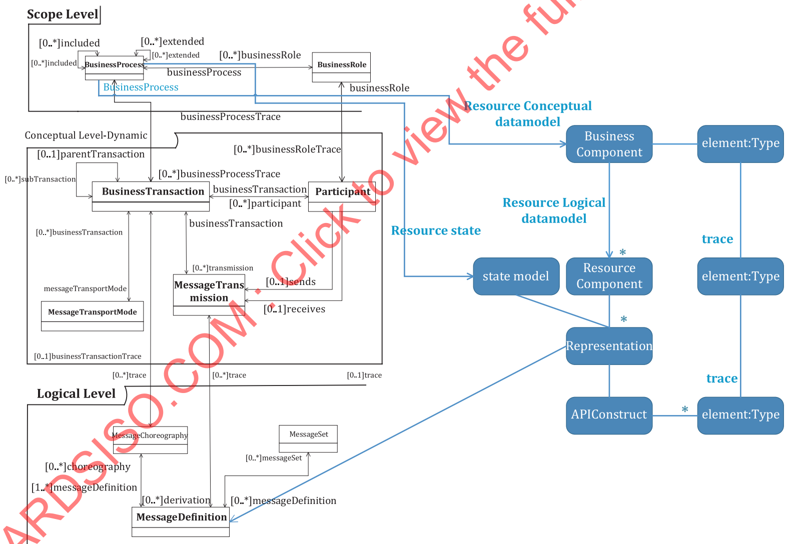

Figure 6 — Leveraging components from the existing ISO 20022 methodology

11.1.2 Modelling guidelines

11.1.2.1 Scope level

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.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

In essence, BusinessTransaction provides for grouping and sequencing of MessageTransmissions relevant to a Business Process. MessageTransmissions correspond to the API requests and responses.

11.1.2.2 Conceptual level

11.1.2.2.1 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.

11.1.2.2.2 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. An interaction between two Participants therefore constitutes 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.

11.1.2.3 Logical level

11.1.2.3.1 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 shall 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.

11.1.2.3.2 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

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

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).

11.1.2.3.3 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 shall 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, this is 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 shall be 'a'
flavour	Request or Response	'001' or '002'
version	version	

EXAMPLE

✉ Get Payment Transactions Request (camt.03.001.02)

✉ Get Payment Transactions Response (camt.03.002.02)

GetPaymentTransactions has ref 'a03'.

The request has ref '001'.

The response has ref '002'.

The version is '02'.

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved

11.2 Mapping rules

11.2.1 RepositoryConcept

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

— Registered; or

— Provisionally Registered (Draft Schemas).

11.2.2 MessageDefinition

MessageDefinition is transformed into an Object with the following content:

— JSON value pair with Name "$schema" and Value 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 the following content:

  — JSON object "@xmlns" with the 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 the 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 11.2.4.

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"
        }
    },
```

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

```
    "definitions": {

        "AccountDetailsConfirmationV07": {
```

11.2.3 MessageBuildingBlock

See “MessageElement

MessageElement is typed by a MessageComponentType”

11.2.4 MessageComponent

A MessageComponent is transformed into a JSON object with the 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 one or more MessageElements (see 11.2.6);

— 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.

```
"Account20": {

    "type": "object",

    "additionalProperties":false,

    "properties":{

        "identification":{

            "type": "string",

            "$ref": "#/definitions/Max35Text"

        },

        "account_servicer":{

            "type": "object",

            "additionalProperties": "false",

            "$ref": "#/definitions/PartyIdentification70Choice"

        }

    },

    "required": [

        "account_servicer"

    ]

}
```

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

11.2.5 ChoiceComponent

A ChoiceComponent is transformed into a JSON object with the 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 one 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

```
"AccountIdentification4Chioce": {

    "type": "object",

    "additionalProperties": false,

    "properties": {

        "iban":{

            "type" : "string",

            "$ref" : "#/definitions/IBAN2007Identifier"

        },

        "other" : {

            "type": "object",

            "additionalProperties": false ,

            "$ref":"#/definitions/GenericAccountIdentification1"

        }

    },

    "oneOf":[

    {

        "required":[

            "iban"

        ]

    },

    {

        "required":[

            "other"
```

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

```
        ]
    }
    ]
}
```

11.2.6 MessageElement

11.2.6.1 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 the 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 the 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

```
"other_account_selection_data": {
    "type": "object",
    "additionalProperties":false,
    "$ref": "#/definitions/InvestmentAccount64"
}
```

11.2.6.2 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 the 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

```
"additional_information": {
    "type": "string",
    "$ref": "#/definitions/Max350Text"
}
```

STANDARDSISO.COM : Click to view the full PDF of ISO/TS 23029:2020

© ISO 2020 – All rights reserved