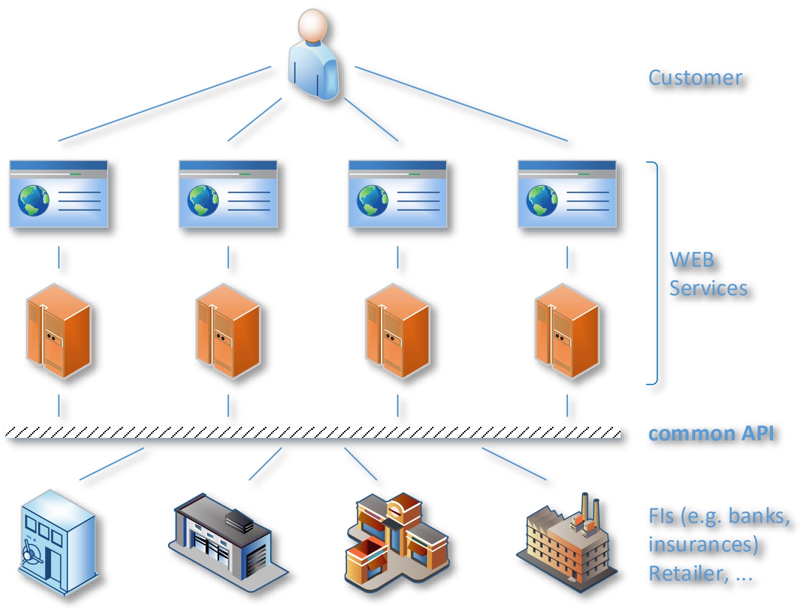
SFTI - working group 'Common API'

Basics



Authorship: Swiss FinTech Innovations
Release: Version 0.05
Date: 20.09.2018

Change Log

Version	Date	Author/s	Comments
0.01	12.03.2018	Jürgen Petry (SFTI)	Document creation
0.02	15.03.2018	Markus Emmenegger (Avaloq) , Dima Dimitrova (Avaloq), Ronny Fuchs (Finstar), Tarmo Ploom (finnova), Patrick Schaller (RED-tec), JP (SFTI)	Review, adjustments
0.03	18.04.2018	Markus Emmenegger (Avaloq) , RF (Finstar), PS (RED-tec), JP (SFTI)	Review, Adjustments
0.04	03.05.2018	Alexander Streule (Avaloq), Markus Emmenegger (Avaloq), Ronny Fuchs (Finstar), Patrick Schaller (RED-tec), Jürgen Petry (SFTI)	Review, Adjustments
0.05	14.06.2018	Markus Emmenegger (Avaloq), Alexander Streule (Avaloq), Tarmo Ploom (Finnova), Ronny Fuchs (Finstar), Patrick Schaller (RED-tec), Jürgen Petry (SFTI)	Review, Adjustments

About SFTI
Swiss Fintech Innovations (SFTI) is an independent association of Swiss financial institutions committed to drive collaboration and digital innovations in the financial services industry. For more information about *Swiss FinTech Innovations*, please refer to http://www.sfti.ch.

Content

1. Preface

The present document contains the basic conventions of the API specification.[1]

In our work, we strive to make the best possible reference to existing API standards. In the banking sector, the pan-European payments interoperability standards and harmonisation initiative *The Berlin Group* (www.berlin-group.org)[2] a strong contributor to such standards.

By examination of their API recommendations, it has been established that their concepts are of good quality and form a good basis for our further work. So the SFTI working group decided to integrate the respective results as foundation for their recommendations of an API standard. This approach is closely coordinated with the responsible task force leaders at *TBG* for substantive and legal reasons.

On the other hand and from a Swiss point of view, SIX provides a set of Swiss Implementation Guidelines (e.g. for Customer-Bank Messages Credit Transfer/Payment Transactions) for the ISO20022 standard. These guidelines form another important foundation for the corresponding API specification.

[1] This does not include security aspects, which are covered in detail by a separate document.

[2] Hereinafter abbreviated by TBG.

2. Basic Requirements

2.1 Technical API Architecture

The API's interface architecture is based on *representational state transfer* (REST[3]). This decision is motivated by the aim for fast performance, reliability and the ability to scale, against the background of a state-of-the-art approach with a future-proofed perspective.

2.2 Existing Standards

The following protocols and API standards have been or shall be examined:

Name	Business Segment(s)	Syntax	Links
TheBerlin Group	payments	json	www.berlin-group.org
ISO20022	payments	XML	ISO 20022[4]
FinTS/HBCI	payments	XML	finTS
FIX	stock exchange trading	serialized ASCI	Financial Information eXchange[5]
EBICS	several	XML	EBICS[6]

Where it is useful and possible with reasonable effort, we will refer to the basic concepts of these standards. Full compatibility with all these standards is not sought for reasons of complexity.

Due to their legacy character, the following the SWIFT MT message protocol will not be recognized:

The following API standards have been examined and will not be taken into further account due to a lack of importance:

Name	Business Segment(s)	Syntax	Links
OFX	several	-	
open bank project	several		

2.3 Respected directives

The banking part of the API specification will respect the any directive which are issued by entities authorized to do so, e.g. issued by Swiss entities like EFD or finma. PSD2 is not explicitly supported due to proven irrelevance for Switzerland and will also in the future only be supported after necessity is proven.

[3] German version: REST

[4] German version: UNIFI (ISO 20022)

[5] German version: FIX-Protokoll

[6] German version: EBICS

3. Conceptual Foundation

3.1 Banking

This following sections provide an overview of the conceptual foundation of the *Common API Specification for Banking* (CAPS).

3.1.1 CAPS Approach

To ease the design of the API Spec, an existing meta model for the definition of the business domains was wanted. The choice of the spec team fell on BIAN, an organization which together with the meta model it developed is described in the next section.

3.1.2 BIAN Essentials

The *Banking Industry Architecture Network e.V.* (BIAN) is an independent, member owned, not-for-profit association to establish and promote a common architectural framework for enabling banking interoperability. It was established in 2008.

The BIAN meta model has three elements that capture the design of the BIAN Service Landscape.

1. **Business Area**
   This is the highest-level classification. A business area groups together a broad set of business capabilities. For the BIAN Service Landscape they are defined to be aspects of business activity that have similar supporting application and information-specific needs.
2. **Business Domain**
   At the next level, business domains define a coherent collection of capabilities within the broader business area. In the BIAN Service Landscape the business domains are associated with skills and knowledge recognizable in the banking business.
3. **Service Domain**
   This is the finest level of partitioning, each defining unique and discrete business capabilities. The Service Domains are the 'elemental building blocks' of a service landscape. The Service Domain relates to generic capabilities that do not vary in their scope, but the definitions of the Business Domain and Business Area are classifications that are specific to a particular Service Landscape layout. The Service Landscape layout can be varied depending on use.

3.1.3 Business Area overview

The following table shows the upper two layers of the BIAN meta model:

Business Area	Business Domain
Reference Data	Party
	External Agency
	Market data
	Product Management
Sales & Services	Channel Specific
	Cross Channel
	Marketing
	Sales
	Customer Mgmt.
	Servicing
Operation & Execution	Loans & Deposits

Business Area	**Business Domain**
	Cards
	Customer Services
	Investment Mgmt.
	Wholesale Trading
	Market Operations
	Trade Banking
	Corp. Financing & Advisory Services
Cross Product Operations	Payments
	Collateral Administration
	Account Mgmt.
	Operational Services
Risk & Compliance	Bank Portfolio & Treasury
	Models
	Business Analysis
	Regulations & Compliance
Business Support	IT Mgmt.
	Non IT & HR Enterprise Services
	Buildings, Equipment and Facilities
	Finance
	Human Resource Mgmt.
	Knowledge & IP Mgmt.
	Corporate Relations
	Business Direction
	Document Mgmt. & Archive

3.2 Insurance

The overview of the conceptual foundation of the *Common API Specification for Insurance* will be added to this document as soon as the work on this topic has started

4. Character Set[7]

The character set is UTF 8 encoded. This specification is only using the basic data elements “string”, “boolean”, "ISODateTime", "ISODate", "UUID" and “integer” and ISO based code lists (with a byte length of 32 bytes).

Max35Text, Max70Text, Max140Text resp. Max512Text are defining strings with a maximum length of 35, 70, 140 resp. 512 characters.

Banks and insurances will accept for strings at least the following character set:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

/ - ? : ( ) . , ' +

Space

Banks and insurances may accept further character sets for text fields like names, addresses, text. Corresponding information will be contained in the company's API documentation. Banks and insurances might convert certain special characters of these further character sets, before e.g. forwarding submitted data.

[7] Based on section 2.1 of the Payment API Implementation Guidelines (Version 1.0) published by TBG.

5. Application Layer: Guiding Principles

5.1 Versioning Concept

As versioning concept for this *Common API Specification*, ***versioning through content negotiation*** has been selected.[8]

REMINDER: This selection also affects section **5.3 Interface Structure** of this document!

5.2 Location of Message Parameters[9]

The API definition follows the REST approach. This approach allows to transport message parameters at different levels:

- message parameters as part of the http level (http header)
- message parameters by defining the resource path (URL path information) with additional query parameters and
- message parameters as part of the http body.

The content parameters in the corresponding http body will be encoded in JSON.

The following principle is applied when defining the API:

Message parameters as part of the http header:

- Definition of the content syntax
- Certificate and Signature Data where needed
- Customer identification data
- Protocol level data like Request Timestamps or Request/Transaction Identifiers

Message parameters as part of the path level:

- All data addressing a resource:
  - provider identification,
  - Service identification,
  - Payment product identification,
  - Account Information subtype identification,
  - Resource ID,
  - Additional information needed to process the request as process steering flags or filtering information,

Message parameters as part of the http body:

- Business data content,
- Customer authentication data,
- Messaging Information
- Hyperlinks to steer the full counterpart/client – bank process

5.3 Interface Structure

The API is resource oriented. Resources can be addressed under the API endpoints[10]

https://{provider}/{service-endpoint}

using additional content parameters

{parameters}

where

- {provider}

[8] Decision from workshop on Mai 4th, 2018, active since version 0.04 of this document.

[9] Based on section 4.1 of the Payment API Implementation Guidelines (Version 1.0) published by TBG.

[10] Versioning is not implemented through URI path, but through content negotiation, cf. section 6.1.

is the host of the API, which is not further mentioned

- {service}
has values as consents, payments, bulk-payments, standing-orders, accounts, card-accounts or funds-confirmations, eventually extended by more information on product types and request scope

- {parameters}
are content attributes encoded in JSON

URIs of service endpoints are made up by hyphenated pattern, e.g. `/payments/sepa-credit-transfer`.

The structure of the request/response is described in the following in the categories

- Path:
Attributes encoded in the Path, e.g. “payments/sepa-credit-transfers” for {resource}

- Query Parameters:
Attributes added to the path after the ? sign as process steering flags or filtering attributes for GET access methods

- Header:
Attributes encoded in the http header of request or response

- Request:
Attributes within the content parameter set of the request

- Response:
Attributes within the content parameter set of the response, defined in JSON

5.4 Body Parameters Naming Conventions[11]

The body parameters in JSON encoding are defined in LowerCamelCase syntax.

This also includes acronyms: No uppercase for standard acronyms, even if it may be less readable (e.g. the abbreviation for the *International Bank Account Number*, which is named *iban* here).

5.5 Pagination

To support id-based pagination, the query parameter `entryReferenceFrom` can be added to query strings. This data attribute indicates that the inquirer is in favour to get all objects after the one with the given ID.[12]

Besides that, a `_links` section shall be added to the response of queries.[13]

[11] In accordance with TBG conventions.

[12] This pagination concept is aligned with TBG's account information API, transaction list.

[13] In accordance with section 4.7 "API steering" of the Payment API Implementation Guidelines (Version 1.0) published by TBG, containing `first`, `next`, `previous` and `last` links.

5.6 HTTP Response Codes

The Common API Specification lists only those error codes for which an implementer should be prepared to handle for business reasons and which are closely linked to the business logic implemented in the API.

Examples of business specific response codes for a request to delete a pending payment:

Status Code	Standard HTTP Status Msg.	Description
200	OK	payment has been deleted
201	Created	payment is marked for deletion
404	Not found	payment does not exist

Highly standardized and technical error codes (e.g. *5xx* server errors, security related codes, bad request) need not be listed explicitly, because any implementer needs to do standard HTTP error handling anyway.

Examples of technical response codes for a request to delete a pending payment:

Status Code	Standard HTTP Status Msg.
400	Bad Request
401	Unauthorized
403	Forbidden
503	Service Unavailable

In general, the entire scope of existing http return codes[14] should be supported by this API's implementations.

5.7 Additional/detailed error information

With regard to the handling of additional/detailed error information, this specification refers to methods described in RFC 7807 "Problem Details for HTTP APIs". For simplicity's sake, only two examples are given here:

Example 1:

```
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"]
   }
```

[14] As published by IANA in its HTTP Status Code Registry: http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

Example 2:

```
      HTTP/1.1 400 Bad Request
      Content-Type: application/problem+json
      Content-Language: en

      {
      "type": "https://example.net/validation-error",
      "title": "Your request parameters didn't validate.",
      "invalid-params": [ {
                            "name": "age",
                            "reason": "must be a positive
   integer"
                          },
                          {
                            "name": "color",
                            "reason":  "must  be  'green',
   'red' or 'blue'"}
                        ]
      }
```

For a detailed description see tools.ietf.org/html/rfc7807.

6. Appendix

6.1 Intro to API Versioning

With regard to the implementation strategy, four different ways of versioning a REST API may be distinguished:

- Versioning through URI Path
- Versioning through query parameters
- Versioning through custom headers
- Versioning through content negotiation

In the following sections, these ways are described in detail.[15] Following this description, the versioning selected for the *Common API Specification* is indicated.

6.1.1 Versioning through URI Path

One way to version a REST API is to include the version number in the URL path:[16]

http://www.example.com/api/**1**/products

This solution often uses URI routing to point to a specific version of the API. Because cache keys (in this situation URIs) are changed by version, clients can easily cache resources. When a new version of the REST API is released, it is perceived as a new entry in the cache. This solution has a pretty big footprint in the code base as introducing breaking changes implies branching the entire API.

6.1.2 Versioning through query parameters

Another option for versioning a REST API is to include the version number as a query parameter:

http://www.example.com/api/products?**version=1**

This is a straightforward way of versioning an API from an implementation point of view. It is also easy to default to the latest version if a query parameter is not specified.

The main drawback comparing to the URI versioning is the difficulty of routing. Query parameters are in fact more difficult to use for routing requests to the proper API version.

Versioning is a crucial part of API design that gives developers the freedom to refactor their code and work on better representations for the resources of their API.

6.1.3 Versioning through custom headers

REST APIs can also be versioned by providing custom headers with the version number included as an attribute:

curl -H "**Accepts-version: 1.0**" http://www.example.com/api/products

The main difference between this approach and the two previous ones is that it doesn't clutter the URI with versioning information.

6.1.4 Versioning through content negotiation

The last strategy we are addressing is versioning through content negotiation:

curl -H "Accept: application/vnd.xm.device+json**; version=1**" http://www.example.com/api/products

[15] Source: https://www.xmatters.com/integrations/blog-four-rest-api-versioning-strategies (reference kindly provided by Markus Emmenegger/Avaloq)

[16] This strategy is used by companies such as Facebook, Twitter, Airbnb, and others.

This approach allows us to version a single resource representation instead of versioning the entire API which gives us a more granular control over versioning. It also creates a smaller footprint in the code base as we don’t have to fork the entire application when creating a new version. Another advantage of this approach is that it doesn’t require implementing URI routing rules introduced by versioning through the URI path.

One of the drawbacks of this approach is that it is less accessible than URI-versioned APIs: Requiring HTTP headers with media types makes it more difficult to test and explore the API using a browser.