# Interactive Financial eXchange Forum


## *RESTful Implementation of IFX*

## *Demonstration Design Review*

*Based on IFX Specification Version 2.4*
*July 2018*

*Interactive Financial eXchange Forum*
http://www.ifxforum.org

# Disclaimer

The IFX Forum makes no warranties whatsoever with respect to the contents of this specification. Without limitation, the IFX Forum makes no warranty (i) that the information contained in the specification is accurate, error-free or describes a practically realizable product or service, or (ii) that the product or service described in the specification can be produced or provided without infringing third-party rights or violating applicable laws or regulations.

RESERVATION OF RIGHTS: The contents of this specification are protected by copyright and other intellectual property laws. The IFX Forum expressly reserves all rights in such content.

# Document Change Summary

| Date/ Revision | Who | Section(s) Changed | Revision summary |
|---|---|---|---|
| June 1, 2018 | IFX API WG | All | Original Draft |
| July 13, 2018 | Copy Editors | Many | Style edits and formatting |
| | | | |
| | | | |

# Table of Contents



# 1 Introduction

The Interactive Financial eXchange (IFX) Business Message Specification (BMS) is developed and maintained as a cooperative industry effort among major financial institutions, service providers, and information technology partners to achieve an open messaging standard for the financial services industry. It provides a comprehensive data dictionary organized as an object model and a message framework suitable for developing new financial industry services and software with common Service Oriented Architecture (SOA) design methodologies.

The evolution of the financial services marketplace has begun to open up new business and technical frontiers including Open Banking business practices, PSD2 directives, JavaScript Object Notation (JSON) data representations and RESTful APIs. Many members of IFX Forum concluded that it was necessary to assess how well the IFX Standard would adapt to this changing environment.

As a result, members of IFX Forum collaborated to produce a RESTful implementation of the current standard. To accomplish this objective it was necessary to assess a variety of tools, consider several different design approaches, and examine how the IFX Standard and framework could be adapted to those considerations. The goal of this work has not been to define an exhaustive specification for IFX RESTful implementations or to indicate a preferred implementation of any particular service. Rather, the goal has been to provide an appropriate level of information to guide developers who want to use the IFX Standard as a basis for Open Banking APIs using JSON and RESTful design concepts.

This document reports our analysis of adapting the IFX Standard to a RESTful API model. We compare architectural models, design assumptions, and implementation strategies. Understanding these concepts is essential to understanding design decisions that were made in our adaptation to OpenAPI 2.0 (formerly known as Swagger).[1]

The concepts in this report rely upon the currently published version 2.4 of the IFX BMS which is available at this URL: https://bms.ifxforum.org/rel2.

After this work was completed, IFX Forum merged with NACHA—The Electronic Payments Association and began collaboration with the API Standardization Industry Group (ASIG). Although, this document does not address specific APIs or standardized APIs, it is a necessary precursor to the next stage of work that will include creating IFX-based standardized APIs.

[1] When IFX began this effort Swagger was being rebranded OpenAPI.

# 2 Model Comparisons

The IFX Standard is based on a Service Oriented Architectural model whose key components are Objects and Messages. IFX Messages result in changes to data on the server or back-end systems of record. On the other hand, in a REpresentational State Transfer (REST) framework, both client and server are responsible for maintaining data (resource) state based on the interaction between them.

| IFX SOA Model | REST Model |
|---|---|
| ***Object***<br>IFX Objects can be somewhat simplistically viewed as organized sets of data of a particular type. All of the data in an IFX Object is related to a business concept or artifact. As in any typical banking environment, the IFX Objects are subject to action in more than one service interaction. | ***Resource***<br>Resources in REST are analogous to Objects in RQ-RS framework, but they are shallower in each given step of interaction due mainly to the absence of related resources. When applicable, these relationships, along with the applicable actions, are represented as REST hypermedia controls. |
| ***Message***<br>IFX Messages are defined to affect the state and content of IFX Objects. The standard does not define implementation details, but IFX Messages are readily represented in XML and can easily be mapped to typical database CRUD (Create, Read, Update and Delete) activities. | ***API***<br>At a very high level, the REST-based client-server interaction is a manipulation of a distributed system via sending "images" of data back and forth. These images are resource states. Hence the name REpresentational State Transfer. |
| **IFX messages act on IFX Objects and are therefore readily adapted to REST concepts where APIs (messages) affect the state of a resource (object).** | |

## 2.1 Process Flow

The RQ-RS model may be viewed as a Remote Procedure Call (RPC) model. In RPC, a server is responsible for updating and processing based on a "script" of commands from a client. This script flow is predefined and known to the client before interaction happens. These flows are typically documented as sequence diagrams.

In REST frameworks, both client and server are responsible for maintaining data (resource) state based on the interaction between them. The interaction flow is driven by client decisions. In contrast with RPC case, the server gives the clients applicable choices of flow during the interaction via hypermedia controls. The results of these interactions are typically documented as state model diagrams.

The table below summarizes two significant and intentional differences between REST and RQ-RS frameworks taking process flows into consideration. The IFX Architecture more closely resembles REST than RPC.

| RQ-RS (RPC) | REST |
|---|---|
| The state of interaction may be kept by the server for the duration of the session. | Client-server interaction in REST intentionally lacks persisted state of the interaction at any moment. |
| Possible flows of an interaction are defined in the RPC before client and server engage, and the clients have knowledge of all possible paths. | In REST, clients do not have to know all possible pathways of conversations. These choices are given to them by servers through hypermedia controls. |

Other practical differences are mostly derived from the above. Here is a somewhat general, but correct explanation of practical aspects. REST APIs are designed around resources, their states, and transitions between these states, while RPC interactions are designed around business functions and processes. REST API data exchange is leaner, but chattier than the RPC interaction.

REST APIs are suitable for distributed systems with eventual data consistency, while RPCs are better suited for centralized systems with immediate data consistency.

## 2.2 Design Considerations

When designing APIs (RESTful or otherwise) developers must be aware that some of the typical design and development disciplines carry additional importance in order to create manageable systems. Perhaps the most significant design challenge is to make APIs resilient to change in the face of broad adoption.

APIs must be robust enough to guarantee a useful lifetime for its clients while simultaneously being responsive to the need for change. API designers must strike a good balance between robust functionality and the time it takes to design and deliver an API to market. These competing requirements result in additional emphasis on designing and implementing transparent, non-breaking extensions.

★★★★★★★★★★★★★
Perhaps the most significant design challenge is to make APIs resilient to change in the face of broad adoption.

APIs must be robust enough to guarantee a useful lifetime for clients while simultaneously being responsive to the need for change.
★★★★★★★★★★★★★

Design considerations to take into account:

- An appropriate versioning approach should be defined and adopted in order to manage change and facilitate upward migration of client software.
- A disciplined extension mechanism should be adopted if customizations are to be offered. It should be capable of permitting incremental changes without breaking interoperability with older clients.

- Choosing tools that readily support these requirements on compatible platforms with necessary programming language and data representation is also essential.

See ***Appendix B*** for a more detailed review of some of the design choices we considered including OData, custom Media-Types, JSON-LD and others.

# 3 The IFX RESTful Lab

To demonstrate the operation of the IFX messages in a RESTful environment, it was necessary to build a server application that could process and respond to RESTful calls. Having no banking core application to interface with, we were limited to operating on the RESTful resource(s) as defined.

## 3.1 General Discussion

Our goal was to apply the RESTful methods of GET, PUT, POST, DELETE and PATCH and build an HTTP server application that could accept and respond to these Methods.

| RESTful Method | IFX Verbs |
|---|---|
| GET | Inq |
| PUT | Add |
| POST | Mod (complete replacement) |
| DELETE | Can, Del |
| PATCH | Mod (differential replacement) |

Users of IFX will recognize that these methods correspond closely with the IFX verbs: Inq, Add, Mod, Can and Del.

We chose to build this application as a J2EE application that we could run in the Tomcat environment we have for the management of the IFX BMS. In addition, we required an environment to store the RESTful Resources the IFX BMS defines. Since we were not attempting to build a core banking system, all we required was the ability to store, retrieve, replace and delete Resource instances. To that end, we leveraged our MySQL environment by creating a database for our Resource instances.

| Method | Parameter | Optional/Required |
|---|---|---|
| GET | IFX-RqUID | Required |
| | IFX-MsgRqHdr | Optional |
| | $filter | Optional |
| | $fields | Optional |
| | $limit | Optional |
| | $offset | Optional |
| | $order_by | Optional |
| | $exclude_url | Optional |
| | $exclude_rec | Optional |
| PUT | IFX-RqUID | Required |
| | IFX-MsgRqHdr | Optional |
| POST | IFX-RqUID | Required |
| | IFX-MsgRqHdr | Optional |
| DELETE | IFX-RqUID | Required |
| | IFX-MsgRqHdr | Optional |
| PATCH | IFX-RqUID | Required |

IFX has from the beginning had the notion of a request identifier (RqUID) that serves many purposes, including duplicate detection and a message header (MsgRqHdr) used to pass security credentials. The IFX API working group decided to carry each of these as parameters to the RESTful environment.

The IFX API working group also decided to use the OData 4.0 specification as a model for record selection and records control that were also implemented as parameters to requests as shown in the table.

Method calls that required passing data records from the client to the server would be placed as JSON strings in the HTTP body. In the case of the PUT and POST, these would be resource instances. In the case of PATCH, this would be a 'difference' document based on the IETF RFC6902.

## 3.2 IFX-Specific Extensions

We also implemented a variant of the GET Method that accepts the URL extension of $count, which will respond with the count of resource records (which may include a $filter parameter). Clients may want to have some concept of the scope of a GET request before requesting large data sets given that banking systems typically contain large databases.

| Method | Extension | Optional/Required |
|---|---|---|
| GET | IFX-RqUID | Required |
| | IFX-MsgRqHdr | Optional |
| | $count | Optional |
| POST | IFX-RqUID | Required |
| | IFX-MsgRqHdr | Optional |
| | $reverse | Optional |
| | $cancel | Optional |

Lastly, we defined variants of the POST Method to accomplish the Rev (Reverse) and Can (Cancel) verbs defined in the IFX BMS. These particular extensions are unique to the IFX specification, but the pattern is quite likely to be generally applicable.

## 3.3 Sample Data and Storage

Given the "demo" scope of our implementation, we were able to keep our database interface layer minimal. Resource instances are stored as text blobs, exactly as received, and are returned in the same form. Since we knew that our database would be small – limited to tens, perhaps hundreds of records – we applied filtration in code after retrieval of all the records of a resource. Similarly, we could apply PATCH requests by reading a record, applying the patch in code, and writing the PATCHed record back to the database.

Importantly, before any record is written to the database we leverage the JSON Schema derived from the IFX BMS with a JSON Schema validator written in our code to ensure that the record matches allowable definitions for that resource as defined in the IFX BMS. This does *not* guarantee that messages have valid business data, but it does ensure that they are well-formed and consistent with the standard.

## 3.4 Our Implementation Platform

We settled on the following platform for our initial project as reasonably representative of a typical RESTful API service platform:

- JSON representation of IFX Objects
- Deployed with SwaggerIO and Azure API management developer portals, which both use OpenAPI v2.0 protocol
- JavaScript with JQuery for client examples
- MySQL Object store for demo data

# 4 Sample APIs

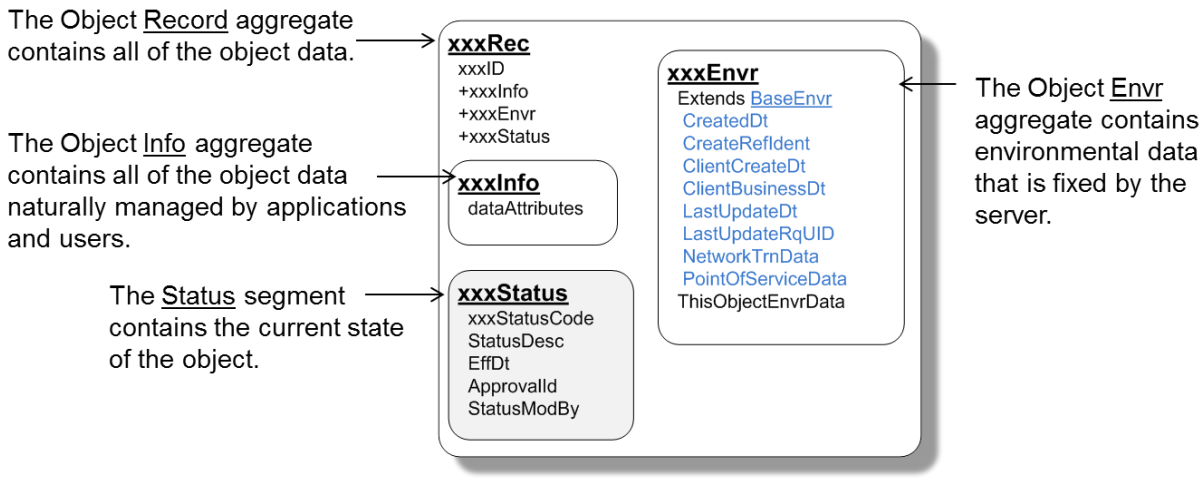
## 4.1 Overview

We created several sample APIs to prove the viability of our design decisions. The APIs described here all use the standard IFX Object definition for Account <Acct> and access data by way of the code we generated from the IFX BMS v2.4.

Every object in the IFX specification is constructed using the pattern illustrated below. For our demonstration, it was simply a matter of implementing a sample database that stores IFX Account objects in the JSON representation that we generated from the IFX BMS Database. The JSON representation starts with the object record that contains the key (ID) and three objects as shown in the table at right.

```
// AcctObj has four items
// 0 - Key (string)
// 1 – AcctInfo (obj)
// 2 - AcctEnvr (obj)
// 3 - AcctStatus (obj)
```

***Appendix A*** shows some functional JavaScript code that illustrates the simplicity of extracting data from the resources returned by the server in response to AJAX calls.)




**A code snippet that shows how data is assigned to JavaScript variables**

```
var acctObj     = data;
var acctid      = acctObj.AcctId; // string
var acctInfo    = acctObj.AcctInfo; // object
var acctStatus  = acctObj.AcctStatus;
var acctEnvr    = acctObj.AcctEnvr;
var acctBal     = acctObj.AcctInfo.AcctBal; // array
var acctObjStr  = JSON.stringify(acctObj);
```

Take a look at this code snippet. For those familiar with the IFX specification, it is obvious how the resources returned from the server are organized, as illustrated above, and assigned to similarly named variables in the front-end application.

## 4.2 Validate Account

In practice, the *Validate Account* scenario would be used to ensure an account number is correct and that the account is active to make and/or receive payments. This scenario may have several variations. Essential to most scenarios is the need to ensure that an account is open at the financial institution handling the request and that the name on the account matches that known by the requester.

In our demonstration, we do not attempt data validation. Instead we display the data attributes that might be typically returned by the API.

## 4.3 Get Account Balance

In practice, the *Get Account Balance* API would be used to retrieve the current balance for a specific account. This might be used to ensure a balance sufficient to make a payment. The *Get Account Balance* API is likely to be a common element of many business scenarios.

## 4.4 Get PSD2 Account Information

PSD2 requires banks to share certain account information more broadly than banks have done historically. This is driving banks to standardize mechanisms and interfaces in order to avoid a chaotic mix of point-to-point interfaces with AISPs (Account Information Service Providers).

The *PSD2 Account Information* API example demonstrates one possible solution to standardizing account information to be shared with aggregators and other AISPs. The IFX Standard includes all of the data elements commonly used in ISO 20022 and much more. Take a look at the *IFX Account Information* API to get a sense of some of the additional data defined in the IFX Standard.

```
var acctObj       = data;
var acctid        = acctObj.AcctId; // string
var acctInfo      = acctObj.AcctInfo; // object
var acctStatus    = acctObj.AcctStatus;
var acctStatusCode = acctObj.AcctStatus.AcctStatusCode;
var acctEnvr      = acctObj.AcctEnvr;
var acctTitle     = acctInfo.AcctTitle;
var acctType      = acctInfo.AcctType.AcctTypeValue;
var acctIBAN      = acctInfo.IBAN;
var acctOpenDt = acctInfo.OpenDt;
var acctBal       = acctObj.AcctInfo.AcctBal; // array
var acctBalType = acctBal[0].BalType.BalTypeValues;
var acctBalAmt  = acctBal[0].CurAmt.Amt;
```

## 4.5 Get IFX Account Information

The IFX Standard includes a very robust definition of the elements and structure of an *Account Object*. This API provides a representative sample of the information available.

In practice, this API might be used to retrieve detailed information about an account. The amount of detail available might be limited by the role and authorization level of the user invoking the API or by business requirements and/or constraints.

## 4.6 Server Interaction

In practice, each interaction between a client and server has certain technical and syntactical requirements that cannot be ignored. The remainder of this section illustrates a few of the details of our implementation that should be informative to potential implementers. These illustrations are not exhaustive.

We used fairly typical AJAX techniques to invoke the server processing at a well-defined endpoint. Also, as would be typical, we created reusable JavaScript functions to handle these calls.

Our endpoint (URL) for an account lookup:

https://bms.ifxforum.org/api_ifx2_4/accounts/

Our required parameter for these examples (an account ID or token):

01234567-0123-0123-0123-01234567890a

In actual practice, we invoke a function using the URL described above and the account token as parameters *url* and *parmid*.

function xhr_get(url, parmid) {}

The function uses these values to construct a URL as follows:

GET "https://bms.ifxforum.org/api_ifx2_4/accounts/01234567-0123-0123-0123-01234567890a"

Typically, API servers require additional information for authorization and authentication. Our implementations on the RESTful Lab and the Microsoft Azure platform are no different, but these platform and implementation-specific features are not detailed here.

Finally, our implementation requires the HTTP header data to include two fields (media type and IFX RqUID). The IFX RqUID value is unique to each message invocation:

"accept: application/json" , "IFX-RqUID: 01234567-0123-0123-0123-01234567890a"

# 5 Summary

## 5.1 Key Learnings

The IFX Standard and the IFX Message Framework are readily adaptable to REST design concepts. This paper describes many of the design decisions we have made and illustrates how to map the IFX Standard messages (methods) to RESTful concepts and HTTP verbs.

IFX Objects *can* be implemented directly as JSON resources, but more work needs to be done to define resources that are not as deeply nested as many of the structures currently defined in the specification for the purposes of APIs.

We concluded that choosing a query language for all of the APIs we develop will significantly reduce the initial effort associated with learning how to use the APIs and reduce ongoing maintenance costs over time. We chose to adapt a subset of the query language developed for OData. It is a proven protocol that is also very comprehensive and robust.

JSON schema and OpenAPI2.0 tooling are not capable of expressing many business constraints – especially logical data relationships, complex cardinality rules, and if-then-else conditions. Consequently, it places a burden on back-end code to validate these business rules.

## 5.2 Next Steps

Our work up to this point shows that one can immediately start to adapt the IFX standard to RESTful Open Banking APIs, and we have provided some tools to jump-start those efforts. Broadly speaking we need to take steps to formalize specific APIs as industry standards.

Specifically, we have identified the following tasks:

- Define resources from the IFX Object model that satisfy narrowly defined microservices and business use cases;
- Review and formally publish the design choices we have described here as part of the IFX Standard;
- Validate the IFX OpenAPI files with additional development tools;
- Review whether it is appropriate to develop and make available JSON schema that support complex logical rules;
- Create documentation templates that facilitate consistent understanding of the scope and applicability of standard APIs;
- Formalize additional API response mechanisms to support many of the concepts embodied in IFX message status and response codes;
- Review the benefits and advisability of submitting our recommendations as a formal media-type under W3C;
- Review the various versioning and extension strategies in order to adopt or recommend best practices; (See Appendix B.)
- Coordinate with ASIG participants to implement specific IFX-based APIs.

In our discussion and review of the results, we have begun to consider the importance of media-types in RESTful implementations. Further research and analysis are required to determine whether our API standardization efforts would benefit by leveraging hyper-media controls (Hypermedia as the Engine of Application State or HATEOAS) and whether this will contribute to more reliable interoperability and foster adoption.

## 5.3 How You Can Leverage Our Results

Begin with the OpenAPI 2.0 files derived from the IFX BMS. Load the OpenAPI 2.0 file into a UI or API Management environment of your choice. (We have proven that the SwaggerIO and Microsoft Azure tools are capable of dealing with the files we generate. We will be working with members and others to validate other tools.)

Modify model definitions to incorporate your customizations:

- Name new or changed elements with a prefix to easily identify them as extensions to IFX (mimic namespace separation available in XSD). This will help you manage changes as IFX makes further progress advancing the standard to REST.
- Leverage a validating editor like the Swagger Editor.
- Change the 'host' as defined in the OpenAPI 2.0 file, to point to your target host.

Deploy a host that meets these requirements, at a minimum:

- Code support for HTTP methods GET, POST, PUT, DELETE and PATCH for the URLs defined in the OpenAPI file.
- Validate the data passed in by each request, responding with a '400 - Bad Request' for nonconforming requests.

Begin testing and report your experience back to IFX on the IFX Community Forum. We value your feedback and will incorporate it into our thinking as we advance the standard. In addition to API topics, the IFX Community Forum also provides a wealth of information about how to leverage IFX generally and how to use the IFX BMS as a searchable glossary and object model.

## 5.4 Conclusion

The IFX Standard and the IFX Message Framework are readily adaptable to REST design concepts. This paper describes many of the design decisions we have made and illustrates how to map the IFX Standard messages (methods) to RESTful concepts and HTTP verbs.

IFX Objects *can* be implemented directly as JSON resources, but more work needs to be done for the purposes of APIs to define resources that are not as deeply nested as many of the structures currently defined in the specification.

We have not committed to a timeline for the work implied here, but we recognize that addressing these considerations will improve the viability of IFX standard APIs, making them more understandable to developers and reducing the effort necessary to adopt the standard.

# 6 Appendix A

## 6.1 Example Client-side JavaScript Code

This Appendix includes some samples of JavaScript functions (using JQuery) we developed to demonstrate the use of IFX.

```
function showOneAcct(parmid) {
	// IFX Object representation
	xhr_get(baseurl + '/'+parmid, parmid).done(function(data){
		var acctObj = data;
		var acctid = acctObj.AcctId; // string
		var acctInfo = acctObj.AcctInfo; // object
		var acctStatus = acctObj.AcctStatus;
		var acctEnvr = acctObj.AcctEnvr;
		var acctBal = acctObj.AcctInfo.AcctBal; // array
		var acctObjStr = JSON.stringify(acctObj);

		// call a simple table formatter for the Object (recursively displays nested objects)
		tableform(acctObj);

		// We could substitute any of the returned objects for display
		tableform(acctInfo);
		tableform(acctStatus);
		tableform(acctEnvr);
	})
};
function showPSD2Acct(parmid) {
	// PSD2 Example
	xhr_get(baseurl + '/'+parmid, parmid).done(function(data){

		var acctObj = data;
		var acctid = acctObj.AcctId; // string
		var acctInfo = acctObj.AcctInfo; // object
		var acctStatus = acctObj.AcctStatus;
		var acctStatusCode = acctObj.AcctStatus.AcctStatusCode;
		var acctEnvr = acctObj.AcctEnvr;
		var acctTitle =    acctInfo.AcctTitle;
		var acctType= acctInfo.AcctType.AcctTypeValue;
		var acctIBAN = acctInfo.IBAN;
		var acctOpenDt = acctInfo.OpenDt;
		var acctBal = acctObj.AcctInfo.AcctBal; // array
		var acctBalType = acctBal[0].BalType.BalTypeValues;
```

```
                var acctBalAmt = acctBal[0].CurAmt.Amt;
        })
};

function getAcctBal(parmid) {
        xhr_get(baseurl + '/'+parmid, parmid).done(function(data){
                var acctObj = data;
                var acctBal = acctObj.AcctInfo.AcctBal; // array
                var acctBalType =acctBal[0].BalType.BalTypeValues;
                var acctBalAmt = acctBal[0].CurAmt.Amt;
        })
};

function validateAcct(parmid) {
        //var acctid = $("#enterAccount").val();
        // alert("validate acct " + acctid);
        // alert("parmid ="+parmid);
        xhr_get(baseurl + '/'+parmid, parmid).done(function(data){
                var acctObj = data;
                var acctBal = acctObj.AcctInfo.AcctBal; // array
                var acctStatus = acctObj.AcctStatus.AcctStatusCode;
        })
};
```

# 7 Appendix B

## 7.1 Further Exploration of Alternative Design Options

When designing APIs (RESTful or otherwise) developers must be aware that some of the typical design and development disciplines carry additional importance in order to create manageable systems. Perhaps the biggest challenge is to make APIs resilient to change in the face of broad adoption. APIs should be made robust enough to guarantee a useful lifetime for its clients while simultaneously being responsive to the need for change. API designers must strike a good balance between robust functionality and the time it takes to design and deliver an API to market. These competing requirements result in additional emphasis on designing and implementing transparent, nonbreaking extensions.

## 7.2 Further Discussion of Version Management

REST principles advise that clients must be able to discover objects/resources at run-time since they are unaware of a back-end schema. Media-types are one mechanism that can allow RESTful APIs to work in the absence of pre-agreed object model schema. One of extensively developed media types designed for this purpose is JSON-LD.

For our purposes, we view media-types as a meta-language used to design other aspects of APIs. It includes a set of syntactical assumptions and agreements between clients and servers.

JSON-LD allows for discovery of resource structures. We did not use this technique when we built our client examples, relying instead upon our knowledge of the IFX object and message structures. We have not ruled out using this approach in the future since APIs built using this approach are resilient to change.

This technique might be viewed as a late binding protocol, where the API elements are discovered, and agreement is achieved at the execution time. Furthermore, a schema-less, vocabulary-based design approach relies on external vocabulary of terms, possibly defined by standards like IFX, FIBO and ISO 20022.

Our working definition of media-type may differ from some common assumptions. For our purposes we view media-types as a meta-language used to design other aspects of APIs. It includes a set of syntactical assumptions and agreements between clients and servers. It is a technology layer supporting business logic and structures. It is defined on top of the transport layer and language such as JSON and HTTP. Media-types define generic API controls for static and dynamic elements. For example, media-type could facilitate a constant agreement about where and how a provider would be specifying relationships or resource actions, and thus the clients would know where to look and how to interpret them.

Correctly chosen media-type can make or break the API. So, designing or finding an existing media-type that has all desired features for a RESTful API is a significant effort. During this phase, an API development team would be making a lot of decisions and compromises. For now, we have chosen OpenAPI as our base media-type.

IFX may choose to complete the exercise of formalizing the IFX RESTful format as a meta-syntax that describes the structure or our message format. This technical work would effectively define an unregistered media-type. However, it would not address the standardization of business definitions.

# 8 Appendix C

## 8.1 IFX Messages adapted to HTTP Verbs

The following general use of HTTP Verbs is proposed:

- POST = Add
- PUT = Replace
- PATCH = Update
- GET = Inquiry
- DELETE = Delete

The verbs will act on IFX Object, usually in JavaScript Object Notation (JSON) syntax, whose top-level contents consist of the following:

| begin Aggregate | | | |
|---|---|---|---|
| SvcIdent | Aggregate | Optional | Service Identifier |
| xxxId | Identifier | Required | Account Identifier |
| xxxxInfo | Aggregate | Required | Account Information Aggregate |
| xxxEnvr | Aggregate | Optional | Account Environment Aggregate |
| xxxStatus | Aggregate | Required | Account Status Aggregate |
| end Aggregate | | | |

For the purposes of RESTful APIs, SvcIdent is deprecated from the above. The general use of JSON is recommended to encode the data contents of parameters and other data values exchanged via IFX REST-based messages.

Where HTTP header values are used, it is proposed that they be prefixed with “IFX-” to keep them from conflicting with other headers. For example, when the “RqUID” field is placed in an HTTP header, it is named “IFX-RqUID”.

| IFX message Replaced with-> | Http VERB | HTTP URL Example | HTTP Headers (JSON) | HTTP Body (JSON) |
|---|---|---|---|---|
| xxxxAdd | POST | ../Acct | IFX-RqUID<br>IFX-MsgRqHdr | xxxxInfo |

| IFX message Replaced with-> | Http VERB | HTTP URL Example | HTTP Headers (JSON) | HTTP Body (JSON) |
|---|---|---|---|---|
| xxxxCan (b) | POST | ../CardOrder/$cancel | IFX-RqUID<br>IFX-MsgRqHdr | xxxxId<br>xxxxInfo |
| xxxxDel | DELETE | ../Acct/{ID} | IFX-RqUID<br>IFX-MsgRqHdr | |
| xxxxInq (a) | GET | ../Acct/{ID} | IFX-RqUID<br>IFX-MsgRqHdr | |
| xxxxInq (b) | GET | ../Acct<br><br>(See Notes below on optional system query parameters.) | IFX-RqUID<br>IFX-MsgRqHdr | |
| xxxxMod (replace) | PUT | ../Acct/{ID} | IFX-RqUID<br>IFX-MsgRqHdr | AcctInfo |
| xxxxMod (a) (update) | PATCH | .. /Acct/{ID} | IFX-RqUID<br>IFX-MsgRqHdr | patch update directions based on RFC 6902 or RFC 5261 |
| xxxxRev | POST | ../Acct/$reverse | IFX-RqUID<br>IFX-MsgRqHdr | RevReasonCode, Desc, RqUIDrev |

Regarding (a) and (b) versions of IFX message mappings to HTTP verbs and URLs, the (a) versions of the mappings embed an IFX Object ID in the URL to the REST resource where the (b) versions of the mappings instead pass an IFX data section parameter such as xxxSel, xxxRec, xxxKeys, etc. Use of either form of these mappings is at the discretion of the IFX implementer.

* * *