# ISO 20022 and JSON:  An Implementation Best Practices Whitepaper

January 2018

By members of the ISO 20022 Registration Management Group and the Technical Support Group
Approved for publication by the ISO 20022 RMG 29 January, 2018

# Contents

# Foreword

This document was prepared by a work group which included members of the ISO 20022 Registration Management Group and the Technical Support Group and was approved for publication by the full ISO 20022 RMG on January 29, 2018.

# Introduction

The purpose of this document is to help implementers in the financial services industry define *RESTful* Web Service Application Programming Interfaces (API) with resources represented in XML and/or JSON syntax, based on new and existing models in the Repository as defined by the international standard:

ISO20022:2013 Financial services — Universal financial industry message scheme

This document was developed in response to a resolution by the ISO 20022 Registration Management Group (RMG) indicating worldwide demand for help in understanding the use of ISO 20022 where JSON is used as syntax.

Note: **RMG Resolution 17/383 Proposals from TSG relating to JSON and APIs**

The ISO 20022 RMG resolves to pursue the RESTful APIs and JSON initiatives through the TSG and the RMG secretariat will put out a call for interested parties to engage with that effort, with a view to delivering a more detailed report by July 18th 2017 that can form the basis of a Technical Report or Specification, to be published under the auspices of ISO, that would provide clear guidance as to how to produce standardised APIs that use the ISO 20022 repository artefacts.

This Whitepaper is a static paper that resulted from the work of the TSG. While it will inform the basis of a Technical Specification in ISO (See: ISO TC68/SC9/WG2) it is a separate document.

Since the publication of ISO20022:2013, providing JSON resources via web services has grown in popularity in the financial services industry. There are several alternative styles of JSON which could be used to represent ISO20022 messages as resources. There are several competing specifications for the definition of RESTful Web Service APIs.

The document provides best practice suggestions on how API services can be exposed to the outside world in a consistent way, reusing ISO 20022 repository artefacts that are common across services. These suggestions are provided in the hope that they will help decrease implementation time for developers that consume these APIs, ease mash-ups, and foster reuse.

Note: JSON Schema Draft 4 has been used as the initial resource specification language, as it is used in several API definition languages, including RAML and JSON-LD are broadly used for defining APIs in the financial services industry.

In order to enable interoperability of financial industry web services, this document provides guidance on the generation of Web Service APIs supporting JSON and XML using selected specification languages. In order to explain the choices made, this document may include additional information such as the history of alternatives considered, with explanations, discussions & decisions.

# ISO 20022 web services methodology and JSON schema transformation rules

## 1 Scope

This document:

— gives best practice suggestions for modelling of RESTful Web Service APIs with artefacts currently embodied in the ISO20022:2013 methodology.
— describes transformation rules from the ISO 20022 Logical Level to JSON Schema 0.4.
— does not consider security of communications nor storage.

Note: It may also:

— identify work for further consideration in the next systematic review of ISO20022.
— include references to recommended security and authentication standards such as TC68/SC2 work items and the ISO/IEC JTC 1/SC 27 IT Security Techniques.

## 2 Normative references

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

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

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

*JSON Schema draft 4 specification* *https://tools.ietf.org/html/draft-zyp-json-schema-04*

## 3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 20022 part 1 and the following apply.

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

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

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

**3.0**
**HyperMedia**
the concept of providing links to other resources. Hypermedia is one of the key principles to a REST architecture.

**3.1**
**Remote Procedure Call [RPC]**
architecture style whereby operations are exposed to manipulate data through HTTP as a transport protocol. This is done by putting the action in the URL (as opposed to REST).

**3.2**
**representation**
description of the state of a Resource that is exchanged between a client and a server. Resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as HTML, XML or JSON, none of which are the server's internal representation.

[SOURCE: https://en.wikipedia.org/wiki/Representational_state_transfer]

**3.3**
**resource**
all the information that can be manipulated (created, read, updated, deleted) in the context of a web service (API) solution

**3.4**
**REST**
Representational state transfer (REST) or RESTful web services

# 4 What is Representational State Transfer (REST)?

## 4.1 Introduction

The REST architecture style is described by six constraints. The most relevant (to ISO 20022) are the **uniform interface** and the **stateless sessions**.

## 4.2 Uniform interface

### 4.2.1 Resource-Based

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

### 4.2.2 Manipulation of Resources through Representations

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

### 4.2.3 Self-descriptive Messages

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

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

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

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

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

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

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

The uniform interface that any REST services must provide is fundamental to its design.

### 4.3 Stateless sessions[1]

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

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

Both the state and a resource are needed:

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

## 5 ISO 20022 Web services

### 5.1 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.

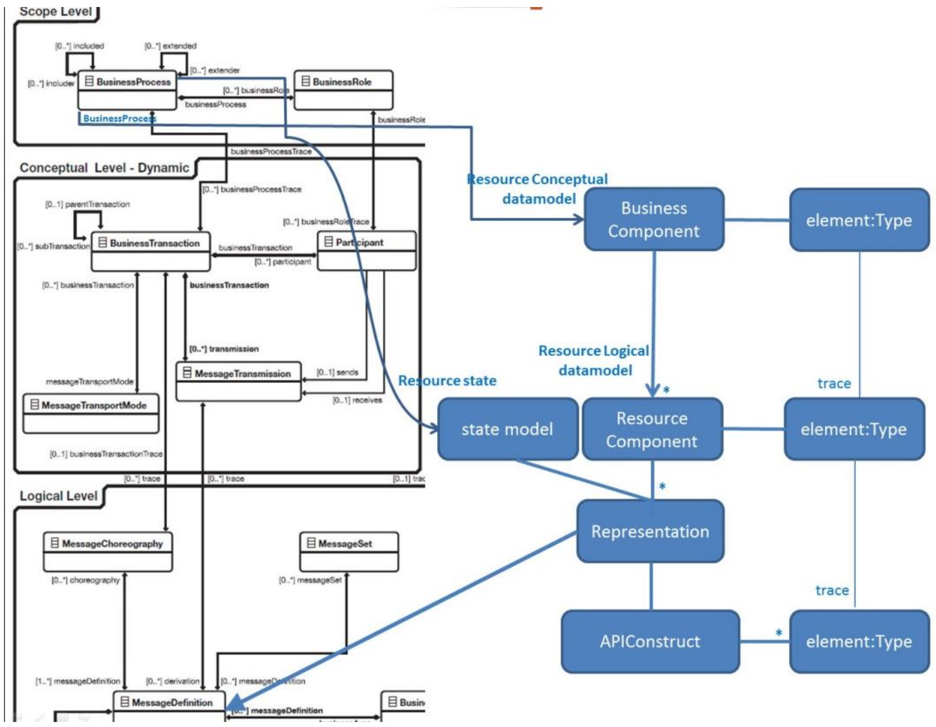


Figure 1 Leveraging components from the existing ISO 20022 methodology

[1] Information about sessions isn't maintained between calls to the service. Instead the URL provides enough information to access the same resource. However, the communication protocol or the application may maintain state

## 5.2 Scope level

### 5.2.1 ISO 20022 metamodel

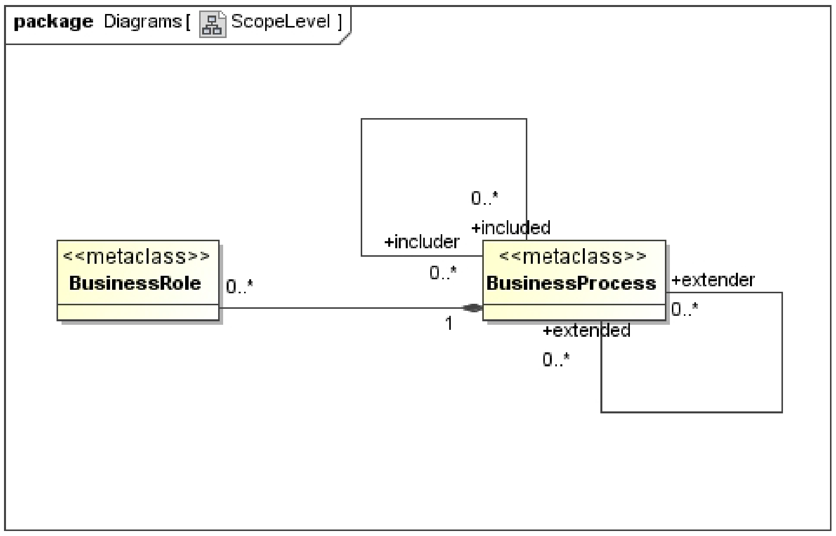


Figure 2 ISO 20022 Scope Level metamodel

### 5.2.2 Modeling guidelines

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

## 5.3 Conceptual Level

### 5.3.1 ISO 20022 metamodel

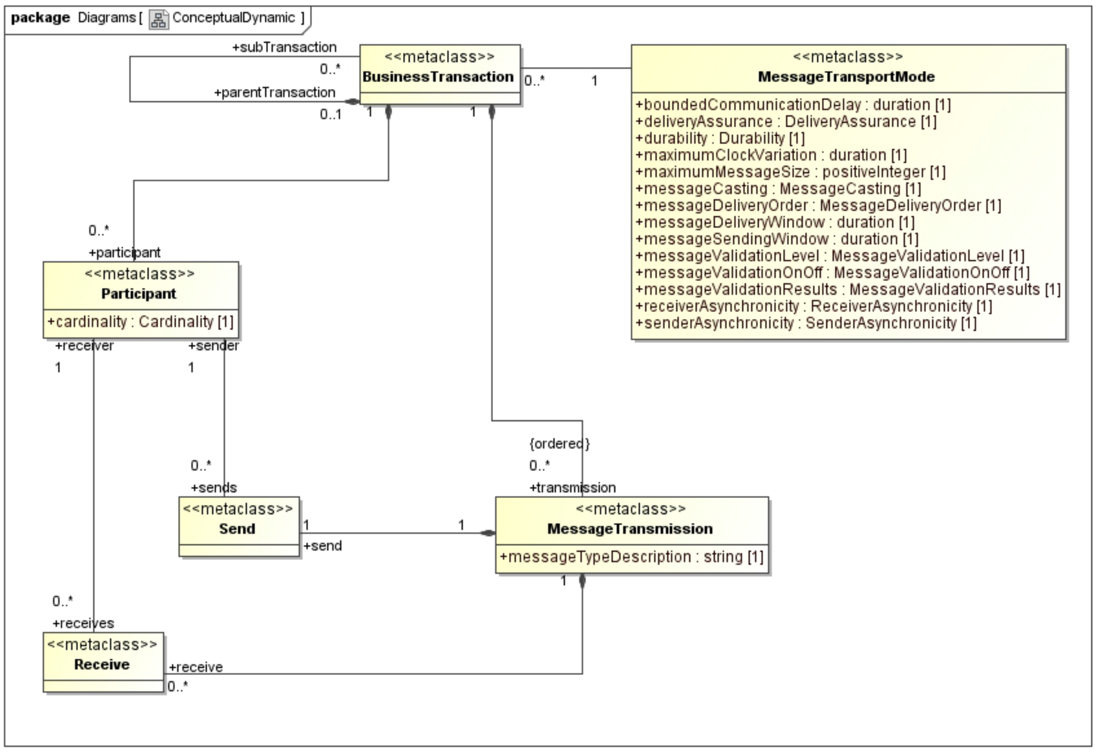

package Diagrams [ ConceptualDynamic ]
<<metaclass>> BusinessTransaction
+subTransaction 0..*
+parentTransaction 0..1
<<metaclass>> MessageTransportMode
+boundedCommunicationDelay : duration [1]
+deliveryAssurance : DeliveryAssurance [1]
+durability : Durability [1]
+maximumClockVariation : duration [1]
+maximumMessageSize : positiveInteger [1]
+messageCasting : MessageCasting [1]
+messageDeliveryOrder : MessageDeliveryOrder [1]
+messageDeliveryWindow : duration [1]
+messageSendingWindow : duration [1]
+messageValidationLevel : MessageValidationLevel [1]
+messageValidationOnOff : MessageValidationOnOff [1]
+messageValidationResults : MessageValidationResults [1]
+receiverAsynchronicity : ReceiverAsynchronicity [1]
+senderAsynchronicity : SenderAsynchronicity [1]
<<metaclass>> Participant
+cardinality : Cardinality [1]
+participant 0..*
+receiver 1
+sender 1
<<metaclass>> Send
+sends 0..*
+send 1
<<metaclass>> MessageTransmission
+messageTypeDescription : string [1]
{ordered} 0..* +transmission
<<metaclass>> Receive
+receives 0..*
+receive 0..*

Figure 3 Conceptual dynamic metamodel

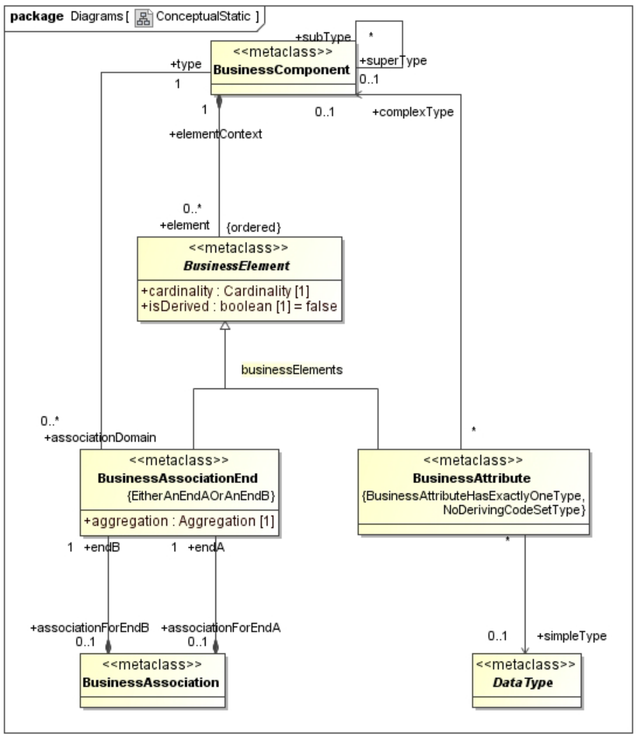

package Diagrams [ ConceptualStatic ]
<<metaclass>> BusinessComponent
+subType *
+superType 0..1
+type 1
+complexType 0..1
+elementContext 1
+element 0..* {ordered}
<<metaclass>> BusinessElement
+cardinality : Cardinality [1]
+isDerived : boolean [1] = false
businessElements
+associationDomain 0..*
<<metaclass>> BusinessAssociationEnd
{EitherAnEndAOrAnEndB}
+aggregation : Aggregation [1]
1 +endB
1 +endA
<<metaclass>> BusinessAttribute
{BusinessAttributeHasExactlyOneType, NoDerivingCodeSetType}
+associationForEndB 0..1
+associationForEndA 0..1
<<metaclass>> BusinessAssociation
0..1 +simpleType
<<metaclass>> DataType

Figure 4 Conceptual Static metamodel

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

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

### 5.3.2 Modeling guidelines

#### 5.3.2.1 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.

#### 5.3.2.2 Model the States[2]

Model the different states by creating a state object for each state of a resource.

Specify for each state the different methods that can be applied. As a state model represents the lifecycle of a resource, it may be necessary to provide different state models for the different resources.

#### 5.3.2.3 Design the BusinessTransactions

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

Avoid chatty conversations resulting in very long interactions.

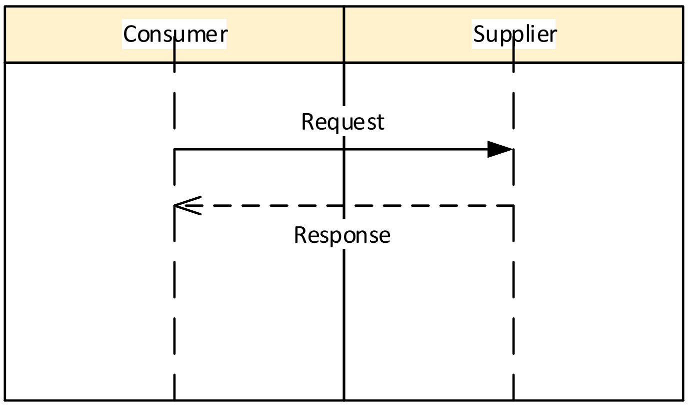


Figure 5 BusinessTransaction with one interaction

[2] A state model is currently not part of the metamodel

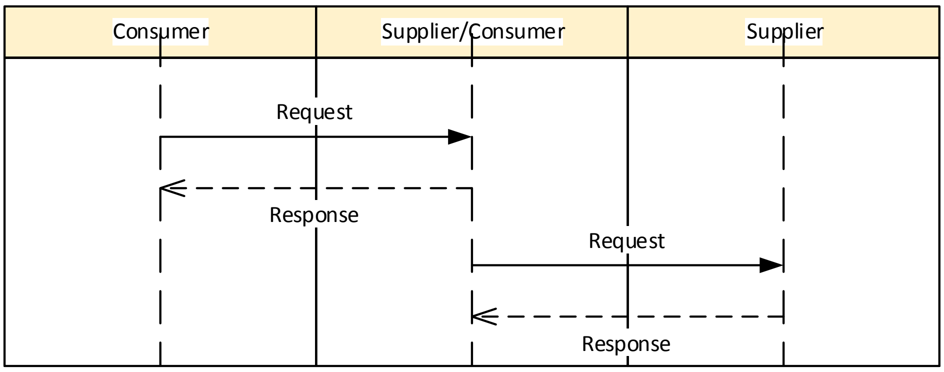


Figure 6 BusinessTransaction with two interactions

## 5.4 Logical Level

### 5.4.1 Introduction

The purpose of the logical level is to define the Resources and the methods that can be applied on each state of the resource(s). Resources form the basis of any RESTful API design.

### 5.4.2 Metamodel

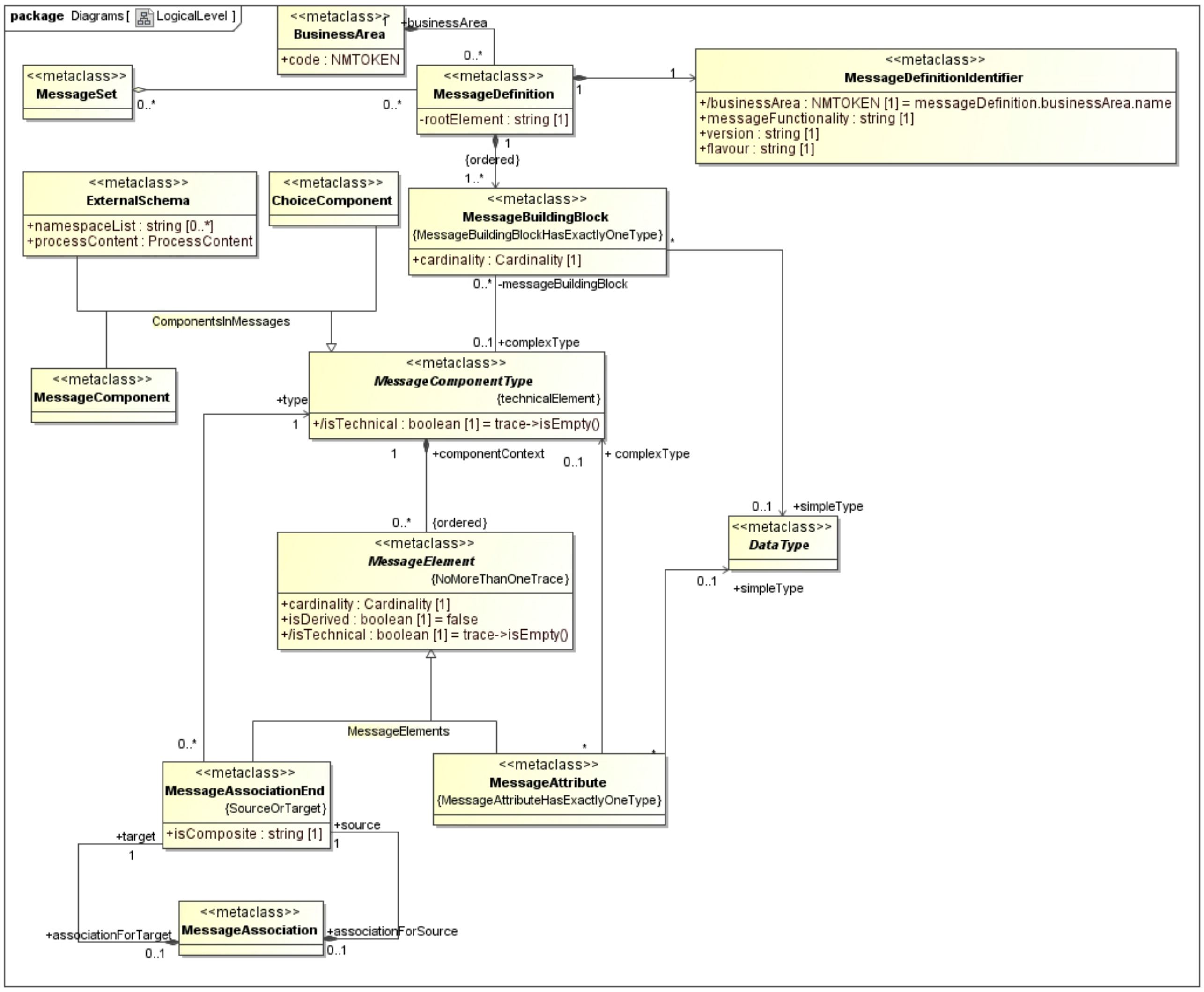

package Diagrams [ LogicalLevel ]
<<metaclass>> BusinessArea
+code : NMTOKEN
+businessArea
<<metaclass>> MessageSet
<<metaclass>> MessageDefinition
-rootElement : string [1]
<<metaclass>> MessageDefinitionIdentifier
+/businessArea : NMTOKEN [1] = messageDefinition.businessArea.name
+messageFunctionality : string [1]
+version : string [1]
+flavour : string [1]
{ordered}
<<metaclass>> ExternalSchema
+namespaceList : string [0..*]
+processContent : ProcessContent
<<metaclass>> ChoiceComponent
<<metaclass>> MessageBuildingBlock
{MessageBuildingBlockHasExactlyOneType}
+cardinality : Cardinality [1]
-messageBuildingBlock
ComponentsInMessages
+complexType
<<metaclass>> MessageComponent
<<metaclass>> MessageComponentType
{technicalElement}
+/isTechnical : boolean [1] = trace->isEmpty()
+type
+componentContext
+ complexType
+simpleType
<<metaclass>> DataType
<<metaclass>> MessageElement
{NoMoreThanOneTrace}
+cardinality : Cardinality [1]
+isDerived : boolean [1] = false
+/isTechnical : boolean [1] = trace->isEmpty()
MessageElements
<<metaclass>> MessageAssociationEnd
{SourceOrTarget}
+isComposite : string [1]
<<metaclass>> MessageAttribute
{MessageAttributeHasExactlyOneType}
+target
+source
<<metaclass>> MessageAssociation
+associationForTarget
+associationForSource

Figure 7 ISO 20022 Logical metamodel

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.

### 5.4.3 Modeling guidelines

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

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

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

#### 5.4.3.2 Design the api calls

##### 5.4.3.2.1 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)

##### 5.4.3.2.2 The representation

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

**A representation**

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

##### 5.4.3.2.3 Type and structure of request messages

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

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

**Note**

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

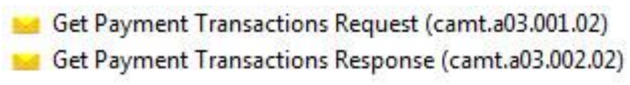
##### 5.4.3.2.4 The messagedefinition identifier

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

The API message identifier maps onto the MessageDefinitionIdentifier as follows:

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

**Example**


Get Payment Transactions Request (camt.a03.001.02)
Get Payment Transactions Response (camt.a03.002.02)

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

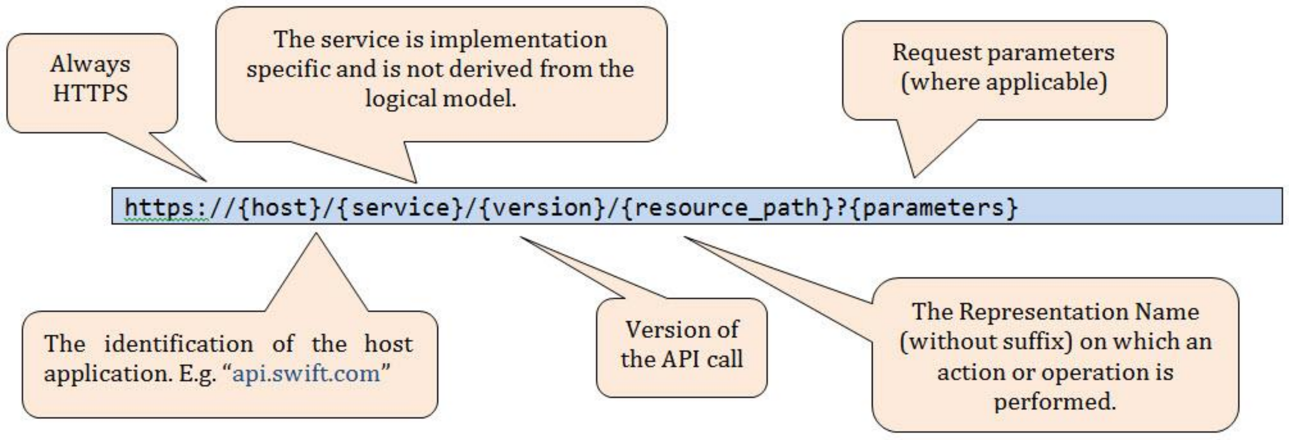
# 6 Implementation tutorial

## 6.1 Introduction

This chapter contains guidelines and considerations to follow when these logical representations are actually transformed into an actual solution, using HTTPS.

The syntax transformation is following the ISO 20022:2013 part 4 XML transformation rules (in case the exchange is using XML), or following the XML – JSON mapping rules specified below.

## 6.2 Composition of the URI


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.swift.com"
Version of the API call
The Representation Name (without suffix) on which an action or operation is performed.

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

URL Example

```
https://api.swift.com/refdata/v1/bics
```

### 6.2.1 Apply the standard HTTP methods

#### 6.2.1.1 Usage guidelines

- Use the standard HTTP methods POST, GET, PUT and DELETE, (and PATCH) to operate on the resources, with the following meanings: POST = create, GET = read, DELETE = delete, PUT (or PATCH[3]) = update.
- 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.
- Use PUT instead of POST in case the API needs to be idempotent.
- Use PUT when you allow the client to specify the resource identifier of the newly created resource. But since PUT is idempotent, you must send all possible values.
- If you want to use PUT to update a resource, it must be a full resource update; you MUST send all attribute values in a PUT request to guarantee idempotency.
- Use PUT when you want or need to send all available values in order to follow idempotency requirements, for instance in the case of a full resource update.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. There are two types of partial updates :
    1. simple partial updates, e.g. changing the value of one or more attributes.
       Use RFC 7396 [4] for the body of the PATCH. This is the simplest, most intuitive way, and preferred.
    2. complex partial updates i.e. cases that cannot be handled by RFC 7396 [4], like partial updates on complex structures,.
       Use RFC 6902 [5] for the body of the PATCH.
       For example, see RFC 7644, where the SCIM API does do complex partial updates using a portion of RFC 6902.[5]

    Although RFC 7396, 6902 and 7644 are using JSON as the syntax, the rules specified therein can also apply when using XML.

- Use POST to create resources when you do not know the resource identifier. With POST creates, it is best practice to return the status of "201 Created" and the location of the newly created resource, since its location was unknown at the time of submission. This allows the client to access the new resource later if they need to.
- You can use POST to send either all available values or just a subset of available values

[3] The patch operation standard for XML is documented in RFC 5261

**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**[4] | **Bulk update orders**[5] | **Delete all orders** |
| **/orders/00023569 8741** | **Error**[6] | **Get the details on this order** | **If it exists, replace the order, else create it**[7] | **If exists, update the order or fail otherwise** | **Delete this order** |

### 6.2.2 Handling associations between resources

#### 6.2.2.1 Usage guidelines

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

***Example***

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

means:

*"Get all SSIs for BIC PVRBRU4VXXX"*

Where BIC PVRBRU4VXXX is considered the primary key.

### 6.2.3 Request parameter usage

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

- Locators (resource identifiers or a specific action)
- Filters (parameters that provide a search for, sorting or limit results)
- State (session identification, API keys…).
- Content (data to be stored)

There are different placeholders where to put those parameters:

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

[4] In case of a replacement, the IDs of the concerned resources must be mentioned in the payload.
[5] A bulk patch is a dangerous and not recommended way to bulk update resources.
[6] A 'POST' is a creation of resource that does not exist yet. Doing a POST on an already existing resource is thus considered an error. Except if the POST is overridden to be a PUT or PATCH
[7] It is bad practice to have the client creates the resource ID. It should always be the server that creates it.

The following table summarises this:

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

#### 6.2.3.1 State

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

#### 6.2.3.2 Content

Content has only one place where it belongs, and that is the request body, either as payload/body content (XML or JSON) or as multipart/form-data request.

#### 6.2.3.3 Locators & 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) of common HTTP tools that need to process these URLs.
- When the information in the Locator or Filter needs to be encrypted. These would normally be put in the URL, but URLs tend to end up in logfiles or tracetools in unencrypted form, so URLs are not a good place for confidential information.

If your API needs to support these two exceptional cases, then for these cases allow using a POST instead of a GET, with method=GET in the URI query part of the POST, and with the body of the POST containing the URI query part as it would have appeared in the normal GET, and with the HTTP header Content-Type set to application/x-www-form-urlencoded. But try to avoid this if at all possible – it is not RESTful to use POSTs for GETs, so only do this if you have to work around these 2 limitations.

Put resource attributes and search criteria behind the '?'

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

***Example***

```
/bics/PVRBRU4VXX/ssis?currency_code=USD&ssi_category=COPA
```

means:

*"All SSIs for BIC PVRBRU4VXX for SSI currency code "USD" and SSI category "COPA"*

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

Order of the attributes in the URI query has no meaning.

#### 6.2.3.4 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.

#### 6.2.3.5 Reducing the list of attributes returned per resource (filtering)

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

**Example** :

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

returns

```
{
  "name"  "My Bank Inc."
 "address" :     {
                         "city" : "London"
                 }
}
```

#### 6.2.3.6 Modeling of 'OR' filters vs 'AND' filters

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

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

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

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

#### 6.2.3.6.1 Pagination requests

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

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

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

**Example**

```
/orders?limit=25&offset=50
```

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

See section 6.3.5 for pagination responses.

## 6.3 The response

### 6.3.1 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.

### 6.3.2 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, etc.). In that case, no 'error' is returned: the HTTP status code will be '200 – OK', 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.

### 6.3.3 The error element

The API should use a generic construct to report an error on API requests.

| Attribute | Definition |
|---|---|
| http | a repeat of the HTTP status code[(1)] |
| code | a more detailed error code [(2)] (which is API-service specific) |
| text | A relatively short text that explains the error in a language that is useful for the end-user of the service, and that can be displayed as such to the end user or logged in a log that is visible to the end-user, if the application developer wants |
| developer_text | a text that explains the possible reason(s) of the error, and the possible corrections, in language geared towards the developer of the application that made the API request. Can be quite long text. Not meant to be displayed to the end-user, mainly intended to assist debugging. |
| more_info | a URI to documents that may help the developer to understand or solve the problem |

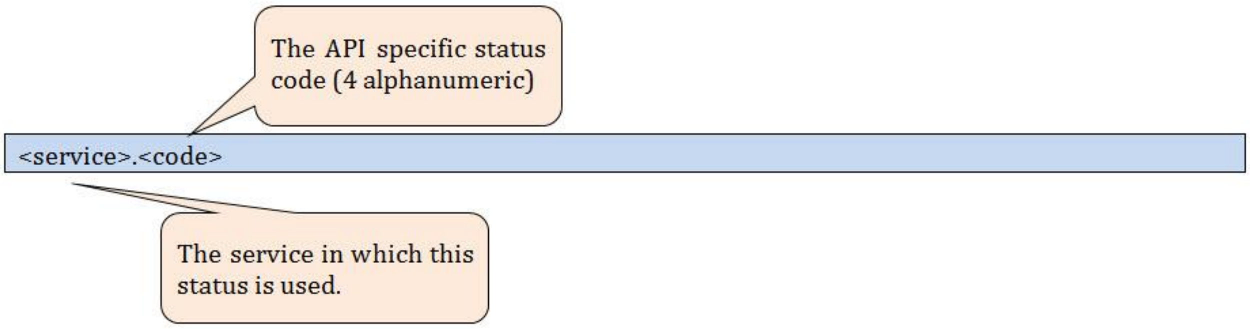
**(1) Following HTTP status codes must be supported:**

| *Code* | *Definition* |
|---|---|
| **200** | OK. This is used for any successful request (GET, POST, PUT or DELETE). The response body contains the result, but no error element. |
| **400** | Bad Request. Something is wrong in the parameters, headers or body of the request. The response body contains an error element with more details. E.g. validation error. |
| **401** | Not Authorized Similar to 403, but specifically for use when authentication is required and has failed or has not yet been provided. The response body contains an error element with more details. |
| **403** | Forbidden. The server understood the request, but is refusing to fulfil it as the user is not authorised to execute this action (but he is already authenticated). [8] |
| **404** | Not Found. The requested resource is not found. The response body contains an error element with more details. |

[8] This code has to be used with consideration as a user can use it to fish for ids that don't belong to him. When using sensitive data, code 404 is recommended for both missing and forbidden resources.

| | |
|---|---|
| **429** | Too many requests. The request exceeded the allowed quota or throttling limits. The response body contains an error element with more details |
| **500** [9] | Server error. The request is valid but could not be executed due to some problem on the server. The response body contains an error element with more details. |
| **503** | Service Temporarily Unavailable. The response body contains an error element with more details. |

(2) Code is composed of two parts, delimited by a '.'


The API specific status code (4 alphanumeric)
<service>.<code>
The service in which this status is used.

**Example**

Using JSON:

```
"error":{
        "http":"400",
        "code":"SWIFTRef.1234",
        "text":"Missing parameter",
        "developer_text":"a parameter was missing in the request",
        "more_info":"http://swiftref.swift.com/sites/sdcref/files/ssi_plus_v2_txt_files_t
ech_spec.pdf"
   }
```

Using XML:

```
<error>
        <http>400</http>
        <code>SWIFTRef.1234</code>
        <text>Missing parameter</text>
        <developer_text>a parameter was missing in the request</developer_text>
        <more_info>http://swiftref.swift.com/sites/sdcref/files/ssi_plus_v2_txt_files_tec
h_spec.pdf"</more_info>
</error>
```

[9] Care must be applied with 5xx responses not to reveal too much detail that may be misused to attack the server. E.g Stack traces should never be sent to a client.

### 6.3.4 Suppressing HTTP status codes

To assist developers that program in an environment that cannot pass HTTP errors to the application, developers can put in the URL of the API request, after the "?":

```
suppress_response_codes
```

If *suppress_response_codes* is present in the URL, then the HTTP response always returns status 200 – "OK", even if there is an error, which guarantees that the response will reach the application. In all cases, the response then contains a "metadata" object at the start, where the first element in "metadata" is a "*http_status*" element which shows the HTTP status code as it would have been without the *suppress_response_codes*. Everything else remains the same i.e. in case of error, there is an "*error*" object following the "*meta*" object, and if there was no error, the "*meta*" object is followed by the actual response data, if any.

### 6.3.5 Pagination responses

See section 6.2.3.6.1 for pagination requests.

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

```
"page_header" : { "total_count" : x, "offset": y, "count" : z },
```

Use a separate element **page_header** to indicate information about the paging in the response. It contains 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. Should be the same as the value of limit in the request, but that's not guaranteed. E.g. in the last page, count can be lower than the value of limit in the request (see section 3.6).
- **offset** to indicate the starting object of the page. Should be the same as the value of offset in the request (see section 3.6)

### 6.3.6 Empty list

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

**Example**

```
GET /orders
```

Response

```
{
   "list_header" : { "total_count" : 0, "offset": 0, "count" : 0 },
   "orders" : []
}
```

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

### *Example "single resource not found"*

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

# 7 Naming Conventions

## 7.1 Names

XML or JSON Element names may use snake_case, i.e. all lowercase, with underscore characters to separate words.

**Examples**

- ssi_category
- currency_code

## 7.2 Single element versus collections of elements

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'), and to use resource IDs to address a single resource within a collection.

This is shown in the table below:

- The first URI is for a collection. E.g. a GET operation on this URI would return a set of all orders.
- the second URI is for a specific element in the collection, with a given 'id'. A GET operation on this URI would return that specific order.

**Example**

| **Resource** | |
|---|---|
| /orders | The collection of all orders |
| /orders/000235698741 | A specific order in the collection of orders |

# 8 Version control

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

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

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

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

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

# 9 JSON Schema transformation rules

## 9.1 Introduction: the ISO 20022 Logical Level

The JSON Schema transformation rules use the ISO 20022 Logical Level as the starting point and specifies for each artefact that can be part of a MessageDefinition the equivalent construct in JSON Schema.

Below depiction shows the current ISO 20022 metamodel of the Logical Level.

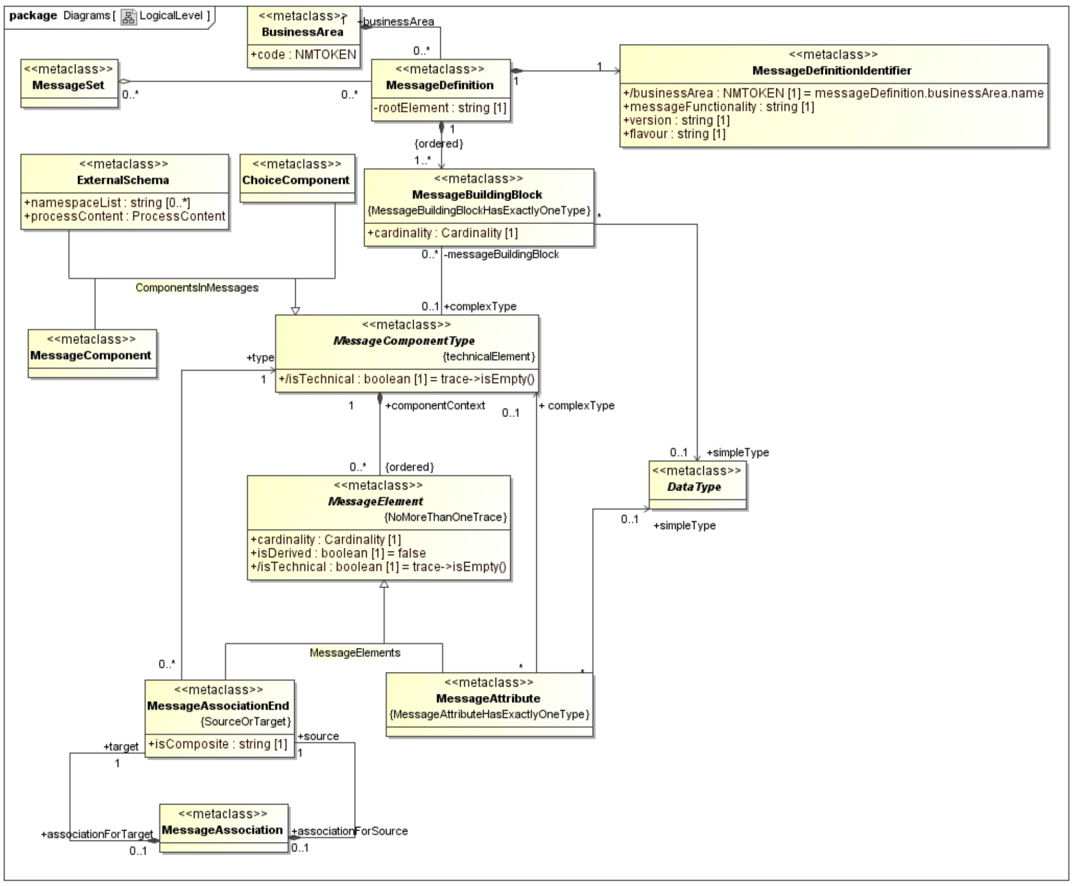

package Diagrams [ LogicalLevel ]
<<metaclass>> BusinessArea
+code : NMTOKEN
1
+businessArea
0..*
<<metaclass>> MessageSet
0..*
0..*
<<metaclass>> MessageDefinition
-rootElement : string [1]
1
1
<<metaclass>> MessageDefinitionIdentifier
+/businessArea : NMTOKEN [1] = messageDefinition.businessArea.name
+messageFunctionality : string [1]
+version : string [1]
+flavour : string [1]
1
{ordered}
1..*
<<metaclass>> ExternalSchema
+namespaceList : string [0..*]
+processContent : ProcessContent
<<metaclass>> ChoiceComponent
<<metaclass>> MessageBuildingBlock
{MessageBuildingBlockHasExactlyOneType}
+cardinality : Cardinality [1]
*
0..*
-messageBuildingBlock
ComponentsInMessages
0..1
+complexType
<<metaclass>> MessageComponent
+type
1
<<metaclass>> MessageComponentType
{technicalElement}
+/isTechnical : boolean [1] = trace->isEmpty()
1
+componentContext
0..1
+ complexType
0..1
+simpleType
<<metaclass>> DataType
0..*
{ordered}
<<metaclass>> MessageElement
{NoMoreThanOneTrace}
+cardinality : Cardinality [1]
+isDerived : boolean [1] = false
+/isTechnical : boolean [1] = trace->isEmpty()
0..1
+simpleType
MessageElements
0..*
*
*
<<metaclass>> MessageAssociationEnd
{SourceOrTarget}
+isComposite : string [1]
<<metaclass>> MessageAttribute
{MessageAttributeHasExactlyOneType}
+target
1
+source
1
<<metaclass>> MessageAssociation
+associationForTarget
0..1
+associationForSource
0..1

Figure 8 ISO 20022 Logical Level metamodel

## 9.2 JSON or XML

The API may support two formats: JSON and XML. JSON is the default response format. If the default syntax is not used, the requested syntax must be specified.

The HTTP "Accept" header[10] is the recommended way to request a response in a specific format.

For JSON:

```
Accept: application/json
```

For XML:

```
Accept: text/xml
and
Accept application/xml[11]
```

The usage of the 'file extension' is also possible. If used, it overrides the Accept header.

```
https://api.service1.com/v1/bics.xml
```

returns a collection of BICs, in an XML structure.

```
https://api.service1.com/v1/bics
```

returns a collection of BICs, in a JSON structure.

It is up to the API designer to decide whether to support XML, JSON or both.

### 9.3 Encoding

Both JSON and XML are encoded using UTF-8.

### 9.4 RepositoryConcept

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

- Registered
- Provisionally Registered (Draft Schemas)

### 9.5 MessageDefinition

MessageDefinition is transformed into an Object with following content:

- JSON value pair with Name `"$schema"` and Value "http://json-schema.org/draft-04/schema#"
- JSON value pair with Name `"type"` and Value "object"
- JSON value pair with Name `"additionalProperties"` and Value "false"
- JSON Object `"properties"` with following content:
  - JSON object `"@xmlns"` with following content:
    - JSON value pair with Name `"default"` and as Value the concatenation of "`urn:iso:std:iso:20022:tech:json:`" with the MessageDefinitionIdentifier.
  - JSON object with Name the value of MessageDefinition.Name but without the Version, converted to "snake case", and with following content:

[10] A comma separated list of the preferred content type in order of preference.
[11] https://en.wikipedia.org/wiki/XML_and_MIME

  - 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 9.7 below.

**Example**

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

## 9.6 MessageBuildingBlock

## 9.7 See “MessageElement

MessageElement is typed by a MessageComponentType”

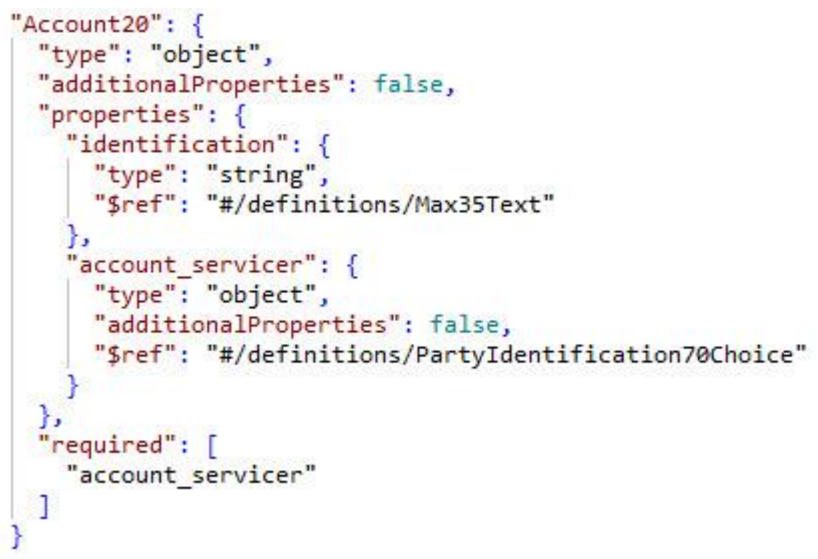
## 9.8 MessageComponent

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

- MessageComponent.Name is the name of the JSON object.
- JSON value pair with Name `"type"` and Value “object”
- JSON value pair with Name `"additionalProperties"` and Value “false”
- JSON Object `"properties"` containing 1 or more MessageElements (see 9.10 )
- 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.



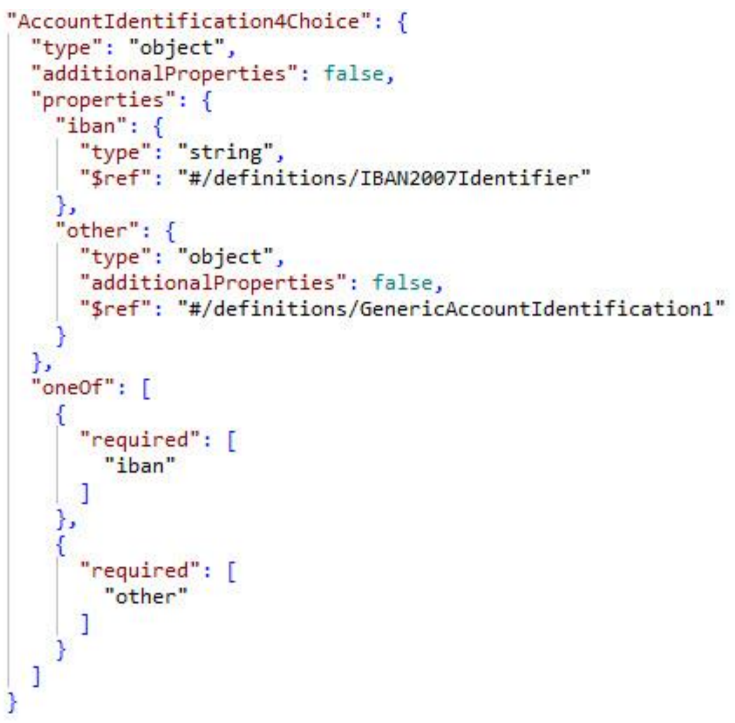
```
"Account20": {
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "identification": {
      "type": "string",
      "$ref": "#/definitions/Max35Text"
    },
    "account_servicer": {
      "type": "object",
      "additionalProperties": false,
      "$ref": "#/definitions/PartyIdentification70Choice"
    }
  },
  "required": [
    "account_servicer"
  ]
}
```

## 9.9 ChoiceComponent

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

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

**Example**



```
"AccountIdentification4Choice": {
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "iban": {
      "type": "string",
      "$ref": "#/definitions/IBAN2007Identifier"
    },
    "other": {
      "type": "object",
      "additionalProperties": false,
      "$ref": "#/definitions/GenericAccountIdentification1"
    }
  },
  "oneOf": [
    {
      "required": [
        "iban"
      ]
    },
    {
      "required": [
        "other"
      ]
    }
  ]
}
```

## 9.10 MessageElement

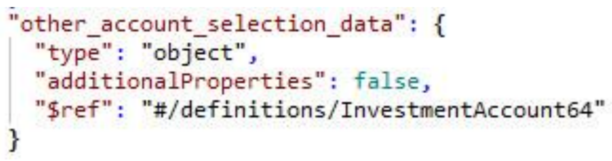
### 9.10.1 MessageElement is typed by a MessageComponentType

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 following characteristics:

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

**Example**

other_account_selection_criteria typed by InvestmentAccount64



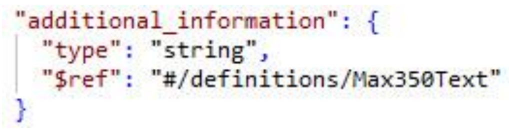
```
"other_account_selection_data": {
  "type": "object",
  "additionalProperties": false,
  "$ref": "#/definitions/InvestmentAccount64"
}
```

### 9.10.2 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 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"
}
```

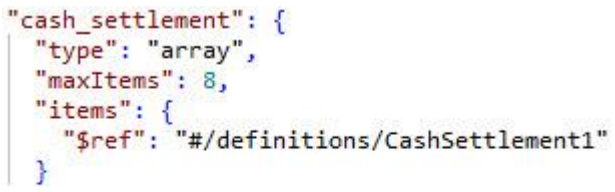
## 9.11 MessageElement is an array

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

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

**Example**

Below example shows the element "cash_settlement" may occur max 8 times:



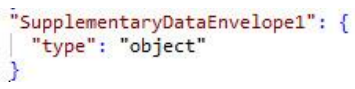
```
"cash_settlement": {
  "type": "array",
  "maxItems": 8,
  "items": {
    "$ref": "#/definitions/CashSettlement1"
  }
```

## 9.12 ExternalSchema

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

- ExternalSchema.Name is the name of the JSON object.
- JSON value pair with Name `"type"` and Value "object"

**Example**



```
"SupplementaryDataEnvelope1": {
  "type": "object"
}
```

## 9.13 ISO 20022 DataType transformation to JSON Schema

### 9.13.1 General

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

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

[12] Default value is zero in which case minItems may be omitted.
[13] The default is that the array is unbounded in which case maxItems may be omitted.

### 9.13.2 DataType Amount

#### 9.13.2.1 CurrencyIdentifierSet is not empty

DataType Amount is transformed into two JSON objects.

A JSON object with following characteristics:

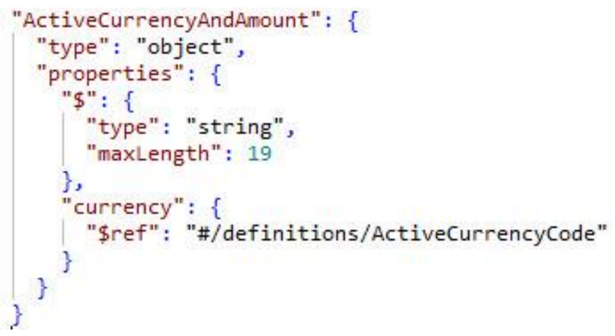
- Amount.Name is the name of the JSON object.
- JSON value pair with Name `"type"` and Value "object"
- JSON Object `"properties"` with following content:
  - JSON value pair with Name "$" and value a JSON object with following values:
    - JSON value pair with Name `"type"` and Value "string"
    -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

A JSON object with following characteristics:

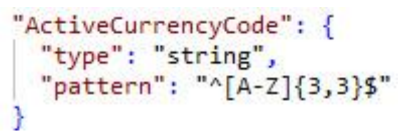
- CurrencyIdentifier.Type is the name of the JSON object.
- JSON value pair with Name `"type"` and Value "string"
- If Property Pattern of CurrencyIdentifierSet's Type is not empty: JSON value pair with Name `"pattern"` and Value the content of that Property.

**Example**

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



```
"ActiveCurrencyAndAmount": {
  "type": "object",
  "properties": {
    "$": {
      "type": "string",
      "maxLength": 19
    },
    "currency": {
      "$ref": "#/definitions/ActiveCurrencyCode"
    }
  }
}
```



```
"ActiveCurrencyCode": {
  "type": "string",
  "pattern": "^[A-Z]{3,3}$"
}
```

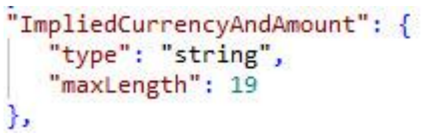
#### 9.13.2.2 CurrencyIdentifierSet is empty

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

- Amount.Name is the name of the JSON object.
- JSON value pair with Name `"type"` and Value "string"
- If Amount.totalDigits is not empty: JSON value pair with Name `"maxLength"` and Value the content of datatype Amount's Property totalDigits.

**Example**

Below example shows an amount of maximum 19 characters.



```
"ImpliedCurrencyAndAmount": {
    "type": "string",
    "maxLength": 19
},
```

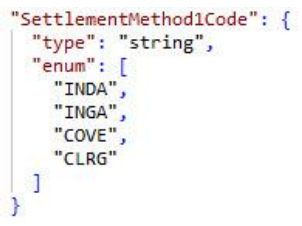
### 9.13.3 DataType CodeSet

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

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

**Example**

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



```
"SettlementMethod1Code": {
  "type": "string",
  "enum": [
    "INDA",
    "INGA",
    "COVE",
    "CLRG"
  ]
}
```

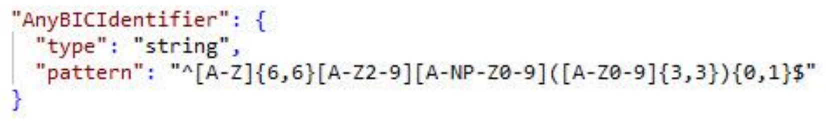
### 9.13.4 DataType Text

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

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

**Example**

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



```
"AnyBICIdentifier": {
  "type": "string",
  "pattern": "^[A-Z]{6,6}[A-Z2-9][A-NP-Z0-9]([A-Z0-9]{3,3}){0,1}$"
}
```

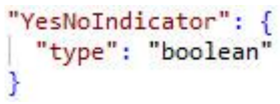
### 9.13.5 DataType Indicator

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

- Indicator.Name is the name of the JSON object.
- JSON value pair with Name `"type"` and Value "boolean"

**Example**

Below example shows datatype YesNoIndicator.



```
"YesNoIndicator": {
  "type": "boolean"
}
```

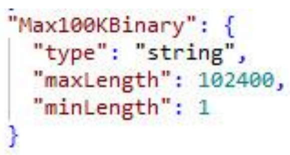
### 9.13.6 DataType Binary

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

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

**Example**

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



```
"Max100KBinary": {
  "type": "string",
  "maxLength": 102400,
  "minLength": 1
}
```

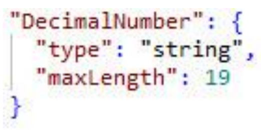
### 9.13.7 DataType Quantity

DataType Quantity[14] is transformed into a JSON object with following characteristics:

- Quantity.Name is the name of the JSON object.
- JSON value pair with Name `"type"` and Value "string"
- If Quantity.totalDigits is not empty: JSON value pair with Name "maxLength" and Value the value of datatype Quantity's Property totalDigits.

**Example**

Below example shows a datatype of representation Quantity.



```
"DecimalNumber": {
  "type": "string",
  "maxLength": 19
}
```

[14] Technically, a Quantity can have a "unit" property, but at present none of the Quantity definitions in the ISO 20022 repository use that attribute.

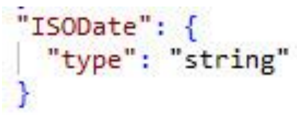
### 9.13.8 DataType DateTime[15]

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

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

**Example**

Below example shows a datatype of representation DateTime.



```
"ISODate": {
  "type": "string"
}
```

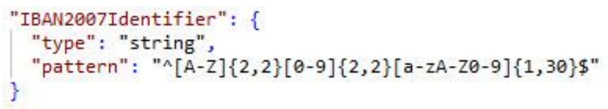
### 9.13.9 DataType Identifier

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

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

**Example**

Below example shows a datatype of representation Identifier.



```
"IBAN2007Identifier": {
  "type": "string",
  "pattern": "^[A-Z]{2,2}[0-9]{2,2}[a-zA-Z0-9]{1,30}$"
}
```

### 9.13.10 DataType Rate

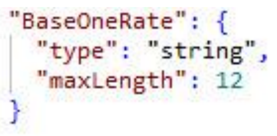
DataType Rate[16] is transformed into a JSON object with following characteristics:

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

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

**Example**

Below example shows a datatype of representation Rate.



```
"BaseOneRate": {
  "type": "string",
  "maxLength": 12
}
```

[15] The same transformation applies for Date and Time.
[16] Technically, a Rate can have a “baseUnitCode” property, but none of the Rate definitions in the ISO 20022 repository use that attribute

# Annex A
(informative)

# Converting ISO 20022 XML into JSON

## A.1 Introduction

To check that the generated ISO 20022 JSON schemas are valid, we need to test them. The best way to test them is to convert ISO 20022 XML messages into JSON, and that validate that JSON against the JSON schemas.

While XML and JSON are structurally similar, there are some complications in converting ISO 20022 XML into JSON:

- You need to convert the abbreviated ISO 20022 XML names into unabbreviated snake_case names.
    - There is no algorithmic way to do this, so instead we used name mapping tables creates during the JSON schema generation process.
- You need to know the type of element/attribute values, and you need to know whether an element is repeatable or not. A non-repeatable XML element is converted to a JSON property with a data type or object value, while a repeatable XML element is converted to a JSON property with an array value.
    - The only way to get this information is to use the XML Schema. In particular, if you validate the XML against the XML Schemas, you can use the XML parser's PSVI (post-Schema validation infoset) API to give you access to the XML Schema information that is required.

### A.1.1 Conversion Procedure

Using the name mapping tables and the PSVI API, the conversion process for each parsed file is as follows:

1. Walk the XML tree top-down from the root element, and populate properties of a JSON object
2. The "xmlns" value is converted to a JSON "@xmlns" property
    a. This assumes that the default namespace is the message's ISO 20022 namespace.
3. The "Document" root element of the ISO 20022 XML message is ignored. Process its single child element as described below.
4. If an element has a simple type with no attributes, the name is mapped to produce a JSON property with a data type value.
    a. XML Booleans become JSON Booleans.
    b. All other data type values become strings – JSON numbers are not used, because JSON parsers do not have a standard decimal precision for numbers.
5. For an element with attributes or child elements, the name is mapped to produce a JSON property with an **object** value if the "maxOccurs" of the element is <= 1.
6. For an element with attributes or child elements, the name is mapped to produce a JSON property with an **array** value if the "maxOccurs" of the element is >= 2.
7. For an element with a simple type and one or more attributes, the simple type value becomes the value of a property called "$" within the element's JSON object.
8. For each attribute within an element, the name is mapped and prefixed with "@" to produce a JSON property with a data type value.
9. For each child element within an element, the name is mapped to produce a JSON property with a data type value, object value or array value as appropriate.
10. Where the same child element occurs twice or more in a parent element, the child element is converted to a single JSON property with an array value.

## A.2 Example

### A.2.1 ISO 20022 XML message

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:tsmt.002.001.04"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ActvtyRpt>
    <RptId>
      <Id>ARPMessage25</Id>
      <CreDtTm>2009-09-09T11:38:00</CreDtTm>
    </RptId>
    <RltdMsgRef>
      <Id>ARRMessage24</Id>
      <CreDtTm>2009-09-09T11:37:00</CreDtTm>
    </RltdMsgRef>
    <Rpt>
      <TxId>01190799181-6940-48</TxId>
      <RptdNtty>
        <BIC>ADIABE22</BIC>
      </RptdNtty>
      <RptdItm>
        <DtTm>2009-09-06T08:52:00</DtTm>
        <Actvty>
          <MsgNm>tsmt.020.001.02</MsgNm>
        </Actvty>
        <Initr>
          <BIC>ADIABE22</BIC>
        </Initr>
      </RptdItm>
      <RptdItm>
        <DtTm>2009-09-06T08:54:00</DtTm>
        <Actvty>
          <MsgNm>tsmt.011.001.02</MsgNm>
        </Actvty>
        <Initr>
          <BIC>SWHQBEBB</BIC>
        </Initr>
      </RptdItm>
    </Rpt>
  </ActvtyRpt>
</Document>
```

### A.2.2 ISO 20022 JSON message

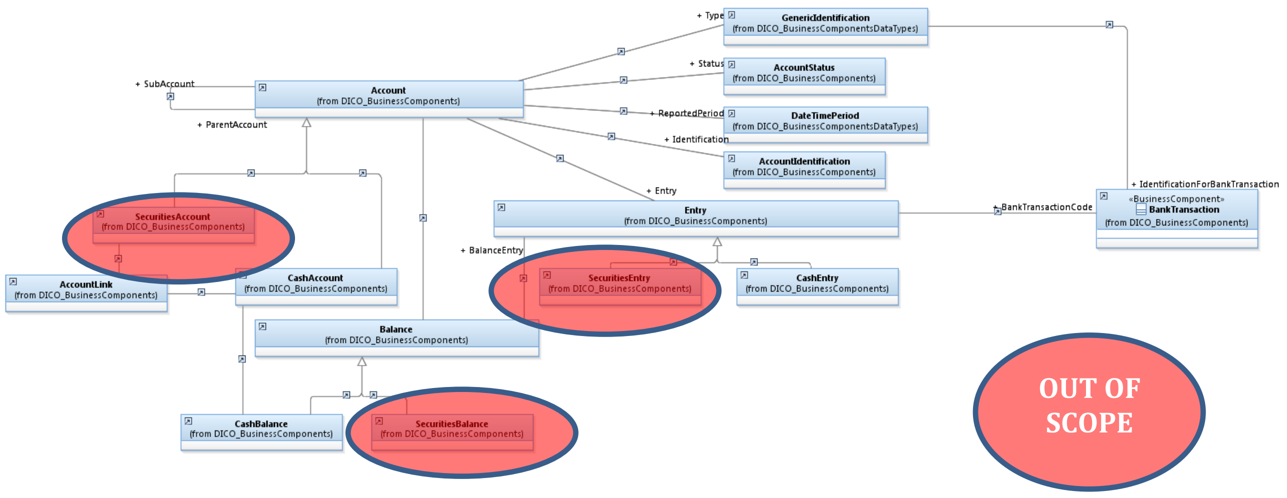
```json
{
  "activity_report" : {
    "related_message_reference" : {
      "creation_date_time" : "2009-09-09T11:37:00",
      "identification" : "ARRMessage24"
    },
    "report" : [ {
      "transaction_identification" : "01190799181-6940-48",
      "reported_entity" : [ {
        "bic" : "ADIABE22"
      } ],
      "reported_item" : [ {
        "activity" : {
          "message_name" : "tsmt.011.001.02"
        },
        "initiator" : {
          "bic" : "SWHQBEBB"
        },
        "date_time" : "2009-09-06T08:54:00"
      }, {
        "activity" : {
          "message_name" : "tsmt.020.001.02"
        },
        "initiator" : {
          "bic" : "ADIABE22"
        },
        "date_time" : "2009-09-06T08:52:00"
      } ]
    } ],
    "report_identification" : {
      "creation_date_time" : "2009-09-09T11:38:00",
      "identification" : "ARPMessage25"
    }
  },
  "@xmlns" : "urn:iso:std:iso:20022:tech:xsd:tsmt.002.001.04"
}
```

# Annex B
(informative)

**Example**

## B.1 Conceptual Level

The below model shows a data model defined at the conceptual level that represents a Cash Account.


+ Type
GenericIdentification
(from DICO_BusinessComponentsDataTypes)
+ Status
AccountStatus
(from DICO_BusinessComponents)
+ SubAccount
Account
(from DICO_BusinessComponents)
+ ParentAccount
+ ReportedPeriod
DateTimePeriod
(from DICO_BusinessComponentsDataTypes)
+ Identification
AccountIdentification
(from DICO_BusinessComponents)
+ IdentificationForBankTransaction
+ Entry
Entry
(from DICO_BusinessComponents)
+ BankTransactionCode
«BusinessComponent»
BankTransaction
(from DICO_BusinessComponents)
SecuritiesAccount
(from DICO_BusinessComponents)
+ BalanceEntry
AccountLink
(from DICO_BusinessComponents)
CashAccount
(from DICO_BusinessComponents)
SecuritiesEntry
(from DICO_BusinessComponents)
CashEntry
(from DICO_BusinessComponents)
Balance
(from DICO_BusinessComponents)
CashBalance
(from DICO_BusinessComponents)
SecuritiesBalance
(from DICO_BusinessComponents)
OUT OF SCOPE

## B.2 Logical Level

Based on the high level business process for customer-to-bank payment the example model below shows the different **states** a Customer to Bank Payment Order process has and the methods/actions that must be applied to get to the different states. An “authorise payment” method will have the resource PaymentOrder change state into an authorised Payment. In that state, a request to accept the payment, a request to reject the payment, or a status request on that payment may be performed. The latter does (status request) does not lead to a new state.

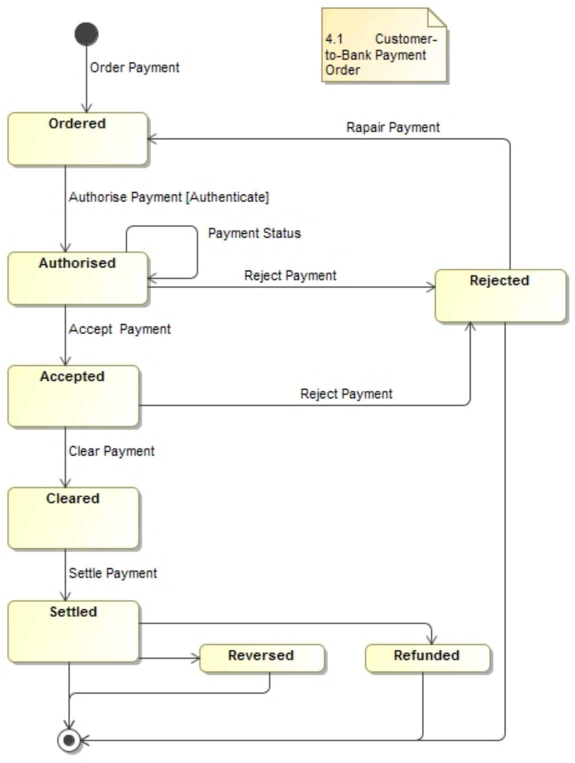
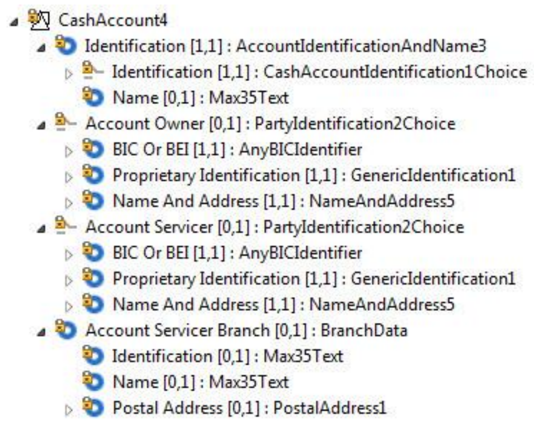


**Figure 9 Payment Initiation State mode**[17]

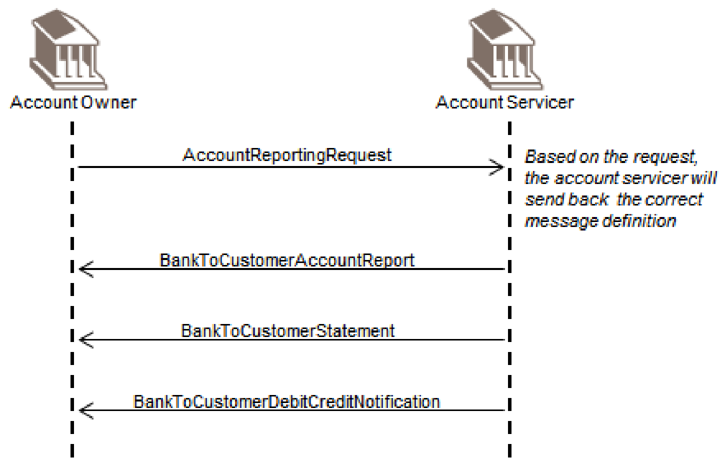
Below shows the datamodel defined at the logical level of a CashAccount. It has an Identification, an Owner, a Servicer and a Branch.



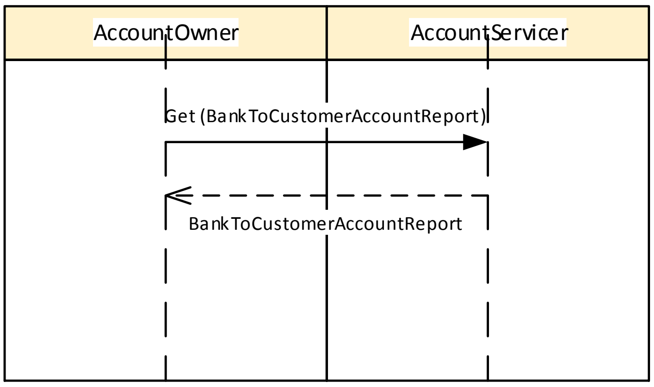
```
CashAccount4
    Identification [1,1] : AccountIdentificationAndName3
        Identification [1,1] : CashAccountIdentification1Choice
        Name [0,1] : Max35Text
    Account Owner [0,1] : PartyIdentification2Choice
        BIC Or BEI [1,1] : AnyBICIdentifier
        Proprietary Identification [1,1] : GenericIdentification1
        Name And Address [1,1] : NameAndAddress5
    Account Servicer [0,1] : PartyIdentification2Choice
        BIC Or BEI [1,1] : AnyBICIdentifier
        Proprietary Identification [1,1] : GenericIdentification1
        Name And Address [1,1] : NameAndAddress5
    Account Servicer Branch [0,1] : BranchData
        Identification [0,1] : Max35Text
        Name [0,1] : Max35Text
        Postal Address [0,1] : PostalAddress1
```

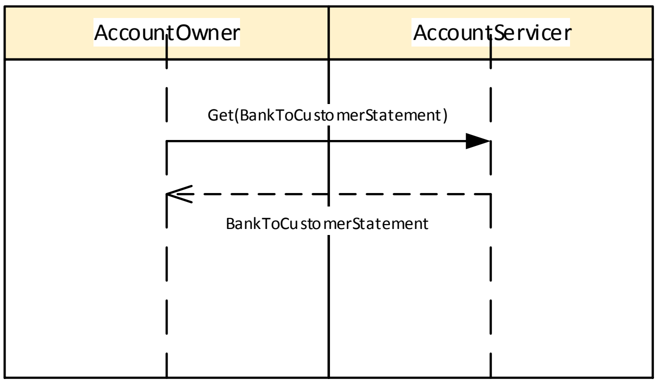
Below shows an Interaction from ISO 20022's Bank-to-Customer Cash Management, showing a BusinessTransaction for a request with three potential responses.

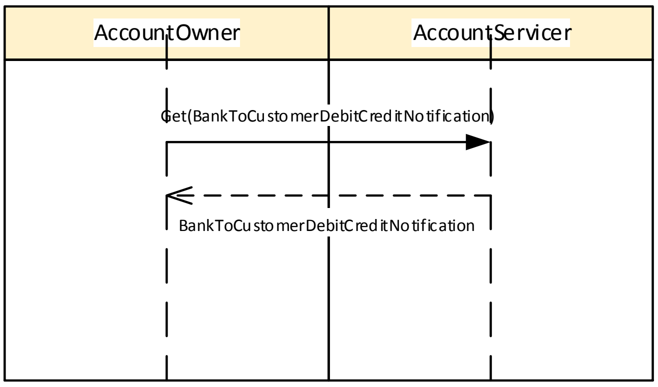
[17] This state model is based on following MDR: https://www.iso20022.org/documents/general/Bank_Account_Management_Maintenance2013.zip


Account Owner
Account Servicer
AccountReportingRequest
Based on the request, the account servicer will send back the correct message definition
BankToCustomerAccountReport
BankToCustomerStatement
BankToCustomerDebitCreditNotification

In a RESTful architecture, the equivalent could be three BusinessTransactions:

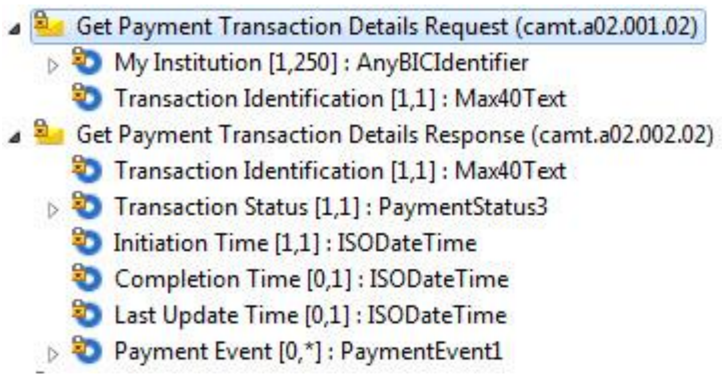

AccountOwner
AccountServicer
Get (BankToCustomerAccountReport)
BankToCustomerAccountReport




AccountOwner
AccountServicer
Get(BankToCustomerDebitCreditNotification)
BankToCustomerDebitCreditNotification

The resource representations in all three exchanges are based on the above specified Resource CashAccount.

Below shows a model of a representation (request and response)



Get Payment Transaction Details Request (camt.a02.001.02)
My Institution [1,250] : AnyBICIdentifier
Transaction Identification [1,1] : Max40Text
Get Payment Transaction Details Response (camt.a02.002.02)
Transaction Identification [1,1] : Max40Text
Transaction Status [1,1] : PaymentStatus3
Initiation Time [1,1] : ISODateTime
Completion Time [0,1] : ISODateTime
Last Update Time [0,1] : ISODateTime
Payment Event [0,*] : PaymentEvent1

## JSON Schema

### Request

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "description": "Prototype JSON ISO 20022 MessageDefinition Schema GetPaymentTransactionDetailsRequest
(camt.a02.001.02) *NOT FOR PRODUCTION USAGE* Generated by SWIFT Standards 2017-07-19 10:29:57",
  "additionalProperties": false,
  "properties": {
    "@xmlns": {
      "default": "urn:iso:std:iso:20022:tech:json:camt.a02.001.02"
    },
    "get_payment_transaction_details_request": {
      "$ref": "#/definitions/GetPaymentTransactionDetailsRequest"
    }
  },
  "definitions": {
    "GetPaymentTransactionDetailsRequest": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "my_institution": {
          "type": "array",
          "maxItems": 250,
          "items": {
            "$ref": "#/definitions/AnyBICIdentifier"
          }
        },
        "transaction_identification": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/Max40Text"
        }
      },
      "required": [
        "my_institution",
        "transaction_identification"
      ]
    },
    "AnyBICIdentifier": {
      "type": "string",
      "pattern": "^[A-Z]{6,6}[A-Z2-9][A-NP-Z0-9]([A-Z0-9]{3,3}){0,1}$"
    },
    "Max40Text": {
      "type": "string",
      "maxLength": 40,
      "minLength": 1
    }
  }
}
```

### Response

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "description": "Prototype JSON ISO 20022 MessageDefinition Schema GetPaymentTransactionDetailsResponse
(camt.a02.002.02) *NOT FOR PRODUCTION USAGE* Generated by SWIFT Standards 2017-07-19 10:29:57",
  "additionalProperties": false,
  "properties": {
    "@xmlns": {
      "default": "urn:iso:std:iso:20022:tech:json:camt.a02.002.02"
    },
    "get_payment_transaction_details_response": {
      "$ref": "#/definitions/GetPaymentTransactionDetailsResponse"
    }
  },
  "definitions": {
    "GetPaymentTransactionDetailsResponse": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "transaction_identification": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/Max40Text"
        },
        "transaction_status": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/PaymentStatus3"
        },
        "initiation_time": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/ISODateTime"
        },
        "completion_time": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/ISODateTime"
        },
        "last_update_time": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/ISODateTime"
        },
        "payment_event": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/PaymentEvent1"
          }
        }
      },
      "required": [
        "transaction_identification",
        "transaction_status",
        "initiation_time"
      ]
    },
    "ActiveCurrencyAndAmount": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "$": {
          "type": "string",
          "maxLength": 19
        },
        "currency": {
          "$ref": "#/definitions/ActiveCurrencyCode"
        }
      }
    },
    "ActiveCurrencyCode": {
      "type": "string",
      "pattern": "^[A-Z]{3,3}$"
    },
    "ActiveOrHistoricCurrencyAndAmount": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "$": {
          "type": "string",
          "maxLength": 19
        },
        "currency": {
          "$ref": "#/definitions/ActiveOrHistoricCurrencyCode"
        }
      }
    },
    "ActiveOrHistoricCurrencyCode": {
      "type": "string",
      "pattern": "^[A-Z]{3,3}$"
    },
    "AnyBICIdentifier": {
      "type": "string",
      "pattern": "^[A-Z]{6,6}[A-Z2-9][A-NP-Z0-9]([A-Z0-9]{3,3}){0,1}$"
    },
    "BaseOneRate": {
      "type": "string",
      "maxLength": 12
    },
    "ChargeBearerType3Code": {
      "type": "string",
      "enum": [
        "SHAR",
        "DEBT",
        "CRED"
      ]
    },
    "ForeignExchangeTerms32": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "from_currency": {
          "type": "string",
          "$ref": "#/definitions/ActiveOrHistoricCurrencyCode"
        },
        "to_currency": {
          "type": "string",
          "$ref": "#/definitions/ActiveOrHistoricCurrencyCode"
        },
        "exchange_rate": {
          "type": "string",
          "$ref": "#/definitions/BaseOneRate"
        }
      },
      "required": [
        "from_currency",
        "to_currency",
        "exchange_rate"
      ]
    },
    "ISODate": {
      "type": "string"
    },
    "ISODateTime": {
      "type": "string"
    },
    "InvalidPaymentsEvent1Code": {
      "type": "string"
    },
    "Max30Text": {
      "type": "string",
      "maxLength": 30
    },
    "Max350Text": {
      "type": "string",
      "maxLength": 350,
      "minLength": 1
    },
    "Max35Text": {
      "type": "string",
      "maxLength": 35,
      "minLength": 1
    },
    "Max40Text": {
      "type": "string",
      "maxLength": 40,
      "minLength": 1
    },
    "Max50Text": {
      "type": "string",
      "maxLength": 50,
      "minLength": 1
    },
    "PaymentEvent1": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "network_reference": {
          "type": "string",
          "$ref": "#/definitions/Max350Text"
        },
        "message_name_identification": {
          "type": "string",
          "$ref": "#/definitions/Max35Text"
        },
        "business_service": {
          "type": "string",
          "$ref": "#/definitions/Max35Text"
        },
        "valid": {
          "type": "boolean",
          "$ref": "#/definitions/YesNoIndicator"
        },
        "invalidity_reason": {
          "type": "string",
          "$ref": "#/definitions/InvalidPaymentsEvent1Code"
        },
        "invalidity_description": {
          "type": "string",
          "$ref": "#/definitions/Max350Text"
        },
        "sender_reference": {
          "type": "string",
          "$ref": "#/definitions/Max30Text"
        },
        "instruction_identification": {
          "type": "string",
          "$ref": "#/definitions/Max35Text"
        },
        "transaction_status": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/PaymentStatus3"
        },
        "forwarded_agent": {
          "type": "string",
          "$ref": "#/definitions/AnyBICIdentifier"
        },
        "funds_available": {
          "type": "string",
          "$ref": "#/definitions/ISODateTime"
        },
        "from": {
          "type": "string",
          "$ref": "#/definitions/AnyBICIdentifier"
        },
        "to": {
          "type": "string",
          "$ref": "#/definitions/AnyBICIdentifier"
        },
        "originator": {
          "type": "string",
          "$ref": "#/definitions/AnyBICIdentifier"
        },
        "creditor_agent": {
          "type": "string",
          "$ref": "#/definitions/AnyBICIdentifier"
        },
        "previous_instructing_agent": {
          "type": "string",
          "$ref": "#/definitions/AnyBICIdentifier"
        },
        "sender_acknowledgement_receipt": {
          "type": "string",
          "$ref": "#/definitions/ISODateTime"
        },
        "received_date": {
          "type": "string",
          "$ref": "#/definitions/ISODateTime"
        },
        "instructed_amount": {
          "type": "object",
          "$ref": "#/definitions/ActiveOrHistoricCurrencyAndAmount"
        },
        "confirmed_amount": {
          "type": "object",
          "$ref": "#/definitions/ActiveOrHistoricCurrencyAndAmount"
        },
        "interbank_settlement_amount": {
          "type": "object",
          "$ref": "#/definitions/ActiveCurrencyAndAmount"
        },
        "interbank_settlement_date": {
          "type": "string",
          "$ref": "#/definitions/ISODate"
        },
        "charge_bearer": {
          "type": "string",
          "$ref": "#/definitions/ChargeBearerType3Code"
        },
        "charge_amount": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/ActiveOrHistoricCurrencyAndAmount"
          }
        },
        "foreign_exchange_details": {
          "type": "object",
          "additionalProperties": false,
          "$ref": "#/definitions/ForeignExchangeTerms32"
        },
        "update_payment": {
          "type": "string",
          "$ref": "#/definitions/Max35Text"
        },
        "duplicate_message_reference": {
          "type": "string",
          "$ref": "#/definitions/Max50Text"
        },
        "copied_business_service": {
          "type": "string",
          "$ref": "#/definitions/Max35Text"
        },
        "last_update_time": {
          "type": "string",
          "$ref": "#/definitions/ISODateTime"
        }
      },
      "required": [
        "network_reference",
        "message_name_identification",
        "business_service",
        "valid",
        "instruction_identification",
        "sender_acknowledgement_receipt",
        "last_update_time"
      ]
    },
    "PaymentReason1Code": {
      "type": "string",
      "enum": [
        "G002",
        "G004",
        "G000",
        "G001",
        "G003"
      ]
    },
    "PaymentStatus3": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "status": {
          "type": "string",
          "$ref": "#/definitions/TransactionIndividualStatus4Code"
        },
        "reason": {
          "type": "string",
          "$ref": "#/definitions/PaymentReason1Code"
        }
      },
      "required": [
        "status"
      ]
    },
    "TransactionIndividualStatus4Code": {
      "type": "string",
      "enum": [
        "RJCT",
        "ACSP",
        "ACSC"
      ]
    },
    "YesNoIndicator": {
      "type": "boolean"
    }
  }
}
```

# Annex C
(informative)

# Open Issues

## C.1 Introduction

Following issues have not yet been discussed or no resolution has been agreed on.

## C.2 Open Issues

1. JSON Schema: Can the Property Namespace be copied into the value pair with Name "$ref"?
2. JSON Schema: Support for ProcessContents?
3. JSON Schema: Support for the equivalent of ID – IDREF. In the logical model this is a relation with isComposite is false.
4. JSON Schema: Support for facets is incomplete.
5. JSON Schema: Support of "format date-time" in time based types.
6. BusinessComponents are not ideal to model Resources, as it is impossible to use subsets of codesets, pick elements from different BC, etc…This may mean a new artefact will need to be created and in the meantime, until the metamodel is updated, another artefact (MessageComponents) may need to be used.
7. Metaclass MessageChoreography is yet to be fleshed out.
8. MessageDefinitionIdentifier: Either we use the flavour, or the MessageFunctionality, or an extension (e.g. XXXX.001.001.01Rq and XXXX.001.001.01Rs), to identify whether it is a request or a response. The flavour is currently proposed, but there are also voices to use a messageFunctionality scheme like: XXXX.01Q.001.01 (for the request) and XXXX.01A.001.01 (for the response)

9. Version control: Two options:

    - A Service URL SHOULD include a {version} to distinguish instances of the service that have a different interface or capability. A service implementation might improve over time, providing better quality information (eg. more of the optional information), but retaining the same interface at the same Service URL so that client can continue to use it without change. When a service implementation changes to require more information from a client, or to provide information the client mustn't ignore, a service version change may be warranted in order to allow clients to adapt to the new service interface.

    - An API must have a version. The version is part of the URL. It is derived from the version of the service.

10. JSON Schema: transformation of the MessageDefinitionIdentifier: should xmlns be used? How should the prefixed portion be constructed?

11. Logical model: additional properties specific to REST design need to be added to the metamodel, such as the HTTP verb, the endpoint(s), etc…

12. JSON Schema: version control: additionalProperties: should this be kept to 'false' or should we allow (some) components to be extensible.

13. What about empty (business) responses? How to model these?

# Bibliography

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[2] http://restcookbook.com/Basics/hateoas/

[4] RFC 7396 https://tools.ietf.org/html/rfc7396

[5] RFC 6902 https://tools.ietf.org/html/rfc6902