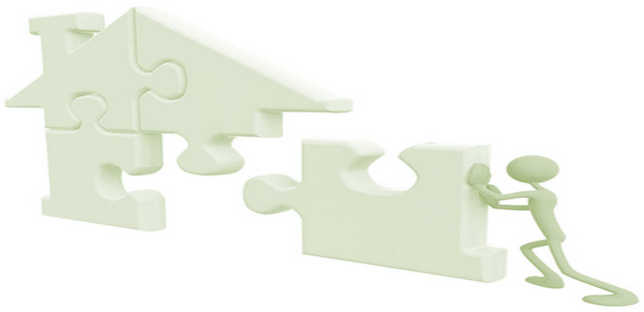
Rest Web services Normalisation



ETO CORPORATE

Corporate Information System Department

Enterprise Reference Architecture

July 2016

Version n°: 1.0

Position Paper

Owner Entity : BNPP Cardif

Scope : BNPP Cardif'Entities

Extended Scope : Solution Providers

Author : Abdeslam SAKTOUN / Enterprise Architect

Status : Validated

Date of Validation : 2016-07-13

Validation Committee : ABC

About the document

Name	Rest web services normalisation
Abstract	This document aims at describing a list of recommendations, principles and best practices to consider when designing Rest web services.

Redaction

Name	Abdeslam SAKTOUN (Author)	Ahmed MERROUCHE (contributor)
Phone	01 41 42 26 80	01 41 42 01 08
Email	abdeslam.saktoun@.bnpparibas.com	ahmed.merrouche@.bnpparibas.com

Document review

Name	**Role**	**Date**
David DREISTADT	Cardif - Responsable Architecture Solutions - Architecture de Patrimoine - DSI Corporate	15/04/2016
Philippe MAROT	Cardif - Architecte Fonctionnel - Architecture de Patrimoine - DSI Corporate	12/04/2016
Cyril PINSON	Cardif - Architecte Fonctionnel - Architecture de Patrimoine - DSI Corporate	12/04/2016
Aurélie HUMBERT	Cardif - Architecte Fonctionnel - Architecture de Patrimoine - DSI Corporate	12/04/2016
Philippe MARETTE	Cardif - Architecte de Données - Architecture de Patrimoine - DSI Corporate	13/04/2016
Pierre SOIGNON	Cardif – Enterprise Architect- Reference Architecture	29/04/2016
Ahmed MERROUCHE	Cardif – Enterprise Architect - Reference Architecture	03/05/2016
Philippe MULLOT	BNPP Group – Enterprise Technical Architect	30/05/2016

Document validation

Name	**Role**	**Date**

Versions

Version	**Date**	**Comment**
V0.1	12/04/2016	Document Draft version
V0.2	20/04/2016	Document modification following comments from Architecture Patrimoine Team
V0.3	04/05/2016	Document modification following comments from reference Architecture Team
V0.4	01/07/2016	Document update following comment from Philippe MULLOT - Group Enterprise Technical Architect
V1.0	13/07/2016	ABC Committee

Reference Documents

Documents	**N° Version**
Cardif_STA-E_IAM Link:https://cyou-cardif-assurance.is.echonet/activities/itcorp/Documents/Cardif_STA-E_IAM.zip	V1.0
PP_Standards Modelisation Services - Schemas draft Link:https://cyou-cardif-assurance.is.echonet/activities/itcorp/Documents/PP_Standards%20Modelisation%20Services%20-%20Schemas%20draft.doc	V1.3

Table of content

1 INTRODUCTION

1.1 Scope

This document is part of the enterprise architecture rules and guidelines produced at Cardif Enterprise Level and must be used and applied by the Cardif's entities and subsidiaries.

The main actors to whom this paper is intended are: project architect, software architect, technical architect, data architect and software developers.

1.2 Aim

Until recently, the use of services within Cardif was based solely on SOAP web services. In the meantime, REST has become a fact standard for building web services.

Many early APIs were written using SOAP but now REST is the dominant force and the publication of REST APIs has been rapidly increasing. The prominent ones being from Amazon, Google, SalesForce and eBay.

The aim of this document is to describe a list of principles, best practices and recommendations to consider when designing Rest web services.

1.3 What is REST

REST stands for Representational State Transfer. It is an architecture style for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

In a REST service, APIs (application programming interfaces) define the media-types that represent resources and drive the application state. The URL and the HTTP method used in the API define the processing rules for a given media type. The HTTP method describes what's being done, and the URL uniquely identifies the resource affected by the action.

1.4 REST Core Concepts

Resources:

In REST everything is a resource. A resource corresponds conceptually to an entity or a set of related entities. A resource could be a document, an image, or information that represents an object such as policy, a product or a person. The notion of a resource could also extend to a service such as a simulation or subscription services.

Application systems expose resources to other applications systems.

Addressability:

Every resource in REST service should have a unique identifier through which it can be addressed. In the HTTP language, these identifiers are URIs.

Uniform Interface:

Resource identifiers and HTTP verbs are used to provide a uniform way of accessing and manipulating resources. The HTTP verbs have dedicated meanings and must be used according to them. The commonly HTTP verbs used are GET, PUT, POST, DELETE. Not every resource has to support all of the HTTP verbs.

Representations:

A client doesn't have to interact directly with a resource but through one of its representations. Representations are defined through media-types, which clearly identify the structure of the representation. Commonly used media-types include ones like xml and json, and more structured ones like atom+xml.

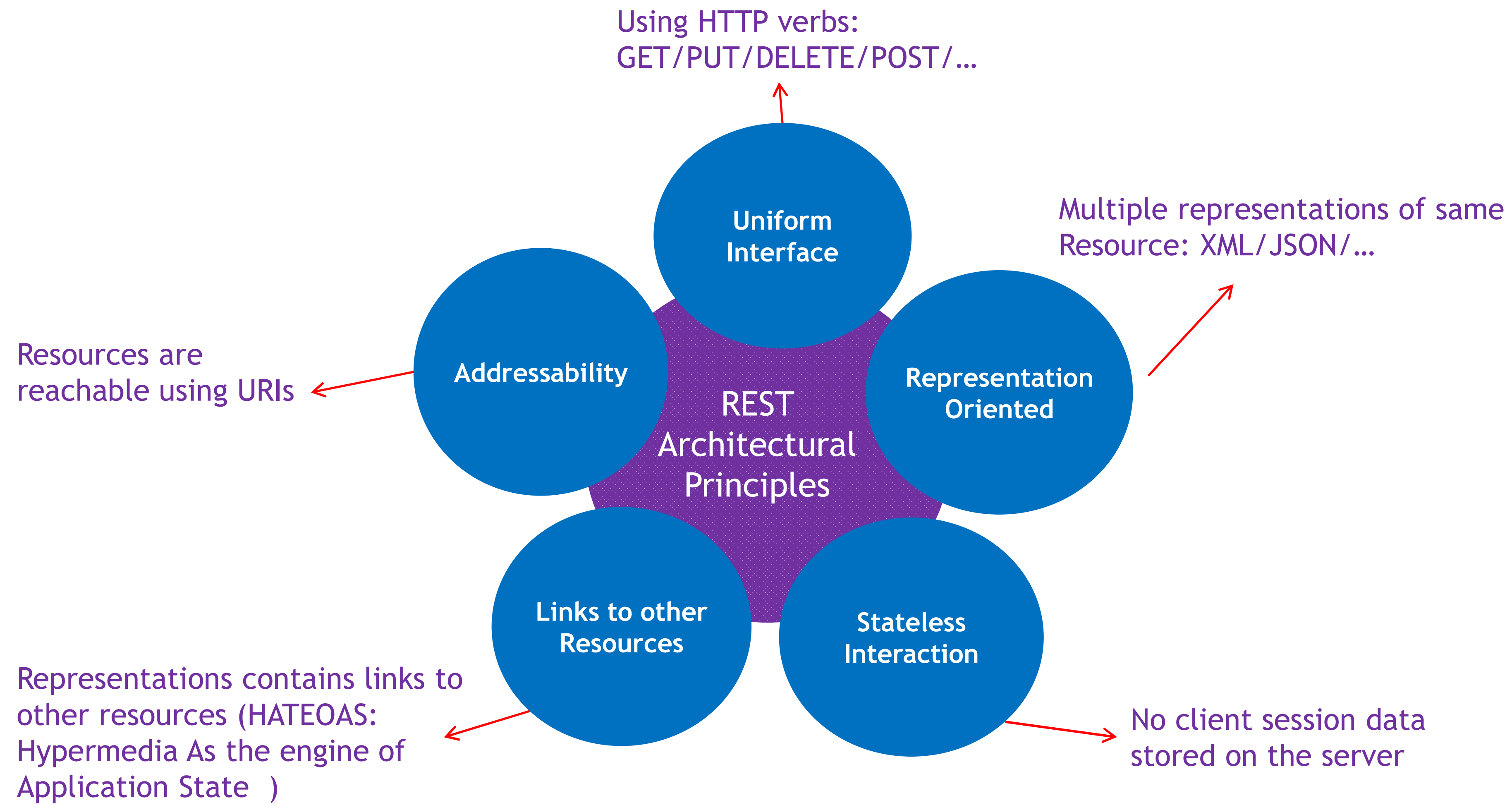
Link to Other Resource:

The representations of a resource can contain links that make reference to other resources, which allows the client to navigate the system based on the resource state and the links provided. This concept is described as Hypermedia as the engine of application state (HATEOAS).

Stateless Interactions:

Sessions state for an application shouldn't rely on client context being stored on the server between requests. Each request made by the application to the server should convey all the information needed to fulfil the request and allow the application to transition to a new state.

1.5 Rest Architectural Principles Summary


Using HTTP verbs: GET/PUT/DELETE/POST/…
Uniform Interface
Multiple representations of same Resource: XML/JSON/…
Addressability
Resources are reachable using URIs
REST Architectural Principles
Representation Oriented
Links to other Resources
Stateless Interaction
Representations contains links to other resources (HATEOAS: Hypermedia As the engine of Application State)
No client session data stored on the server

2 REST Services Normalisation

2.1 Identification of resources

REST uses a resource identifier to identify the particular resource involved in an interaction. The provider that assigned the resource identifier, is responsible for maintaining its semantic validity and ensuring that the identifier does not change over time.

The identifier of a resource can be created either by the application managing the life cycle of the resource or by invoking a common service which is in charge of the generation of standardized identifiers.

2.2 URL Naming

URL Template for a resource:

Below a URL template with its components. The words between curling braces are PATH variables.

http://{apiname}.assurance.bnpparibas.com/{version}/{resource}.{mediatype}

Element Name	Element Description	Example
http	Protocol	
apiname	Name of the API or the application that implements the API	sugar
apiname.assurance.bnpparibas.com	Host name	
version	Major version of the API	v1 / v2
resource	The underlying concept or resource impacted by the action to be done	policies / products / customers
mediatype	The representation of the resource	xml / json / etc…

URL Template for a sub-resource:

Below a URL template with its components. The words between curling braces are PATH variables.

http://{apiname}.assurance.bnpparibas.com/{version}/{resource}/{id}/{sub-ressource}.{mediatype}

Element Name	Element Description	Example
http	Protocol	
apiname	Name of the API or the application that implements the API	sugar
apiname.assurance.bnpparibas.com	Host name	
version	Major version of the API	v1 / v2
resource	The underlying concept or resource impacted by the action to be done	policies / products / customers
Id	The identifier of the parent resource	012057
sub-resource	The underlying sub-resource impacted by the action to be done. The sub-resource is dependent on the resource.	Covers / Addresses /
mediatype	The representation of the sub-resource	xml / json / etc…

2.3 Resource Naming

Resource Names:

Name resources as nouns as opposed to verbs or actions. In other words, a REST URI should refer to a resource that is a thing instead of referring to an action.

Resource Name Form:

Use the plural form of the resource as opposed to a singular form so that you can address two types of resource: a collection of resources or a single resource.

Resource Name Case:

When the name of the resource or a sub-resource is composed of more than one word, it is recommended to use a spinal form: bank-accounts.

So don’t use the camel case form: bankAccounts, because some servers ignore case.

Examples:

- /policies
- /subscriptions
- /customers
- /bank-accounts

- /credit-card

2.4 Scheme Versioning

Scheme versioning is one of the most important considerations to take into account when designing web services.

There are different ways to manage the versioning of rest web services: embedded in the URL or passed in the header of the request.

We recommend to use only the method based on URLs. In this case, versioning must be set up at the highest scope of the URL. The version is specified with a <**v**> prefix and located just before the resource name.

Only the major versions have to be managed using an ordinal number (**1**,**2**, …).

Examples:

- /v1/customers
- /v2/subscriptions

2.5 Media Types

Rest web services can exchange different representations of resources. There are different ways to manage the different representations of resources: based on URLs, based on HTTP Accept Header, etc.

We recommend to use only the method based on URLs. In this case, the formats supported by a service is associated to the name of the resource or the sub-resource.

Examples:

- /v1/cutomers.xml
- /v2/policies.json
- /v1/documents/01280/files.xml

Remark:
The method based on HTTP Accept Header requires using content negotiation to determine the content format that is used to exchange data between servers and clients. The Accept header tells the server what formats the client is looking for.

Accept: application/json;q=1.0, application/xml;q=0.5
The values indicate that JSON is preferred but XML is acceptable as well

3 APIs Normalisation

In the following of this document, we use the word <<host>> to make reference to the common part of the URL of resources and sub-resources.

host=http://apiname.assurance.bnpparibas.com/

3.1 HTTP Methods Commonly Used

We describe below some of the commonly used HTTP methods.

GET:

The **GET** Method is used to retrieve resources. GET requests don't modify the system state, they are considered to be safe, because they don't cause any changes to system state and idempotent because they produce the same effect whether they are applied once or any number of times.

Remark
Idempotent does not mean that operations must produce the same result on the server. The response itself may not be the same as the state of a resource may change between requests. But a GET does not have to cause side effects and alteration of resources.

POST:

The **POST** method is used to create resources. The POST method is not considered safe because it changes the system state. Also it is non-idempotent because multiple requests would result in multiple resources being created.

The POST method is very flexible and it is often used when no other HTTP method seems appropriate.

PUT:

The **PUT** method is used to modify resources. By using a PUT method, a client can modifies the state of a resource and sends it to the server which is in charge to replace the current state of the resource with the new state.

The PUT method is not considered safe because it changes the state of the system. It is considered idempotent because putting the same resource once or more than once would produce the same result.

DELETE:

The **DELETE** method is used to delete existing resources.

The DELETE method is not considered to be safe because it modifies the state of the system. It is considered idempotent because subsequent DELETE requests would still leave the system in the same state.

3.2 Resources Creation

To create a resource, you need to use the **POST** method of the HTTP protocol with the URL of the resource to be created.

The template for creation of resources is: **POST** host/{version}/{resource}.{mediatype}

Creation of a new resource or collection of resources. Resources to be created are passed in the body of the request by the client.

Example:
- POST host/v1/customers.xml
- POST host/v1/subscriptions.json

3.3 Resources Reading

To retrieve existing resources, you need to use the **GET** method of the HTTP protocol with the URL of the resource to be read.

The template to retrieve a given resource is: **GET** host/{version}/{resource}.{mediatype}/{id}

Getting a resource by Id (identifier). The identifier of the resource is embedded in URL. Resource retrieved is passed in the response by the service.

Example:
- GET host/v1/customers.json/12580
- GET host/v1/products.xml/CGV8V01

The template to retrieve all resources is: **GET** host/{version}/{resource}.{mediatype}

Get all resources. Resources retrieved are passed in the response by the service.

Example:
- GET host/v1/products.json

3.4 Resources Searching

To find resources using search criteria, you need to use the **GET** or the **POST** methods of the HTTP protocol with the URL of the resource to be retrieved.

The use of GET or POST methods depends on how the search criteria are passed to the service: embedded in the URL or passed in the body of the request.

1/ Criteria embedded in the URL

The template to search resources by passing search criteria in the URL is:

GET host/{version}/{resource}.{mediatype}?param1=value1&param2=value2

Search resources based on criteria embedded in URL. Resources retrieved are passed in the response by the service.

Example:

- GET host/v1/customers.json?name=lacosta&surname=paul
- GET host/v1/policies.xml?satus=inforce

2/ Criteria passed in the body of the request

The template to search resources by passing search criteria in the body of the request is:

POST host/{version}/{resource}.{mediatype}/search

In this case we need to specify the action to be done, that is, the search verb in this case.

Example

- POST host/v1/policies.xml/search

3.5 Resources Update

To update existing resources, you need to use the **PUT** method of the HTTP protocol with the URL of the resource to be updated.

There are two different ways to update existing resources:

- Update a resource whose data to be modified are embedded in URL
- Update a resource or a collection of resources whose data to be modified are passed in the request

1/ Data embedded in the URL

The template to update a resource of which data to be modified are embedded in URL is:

PUT host/{version}/{resource}.{mediatype}/{id}?param1=value1&param2=value2

Example:

- PUT host/v1/policies.xml/125689?status=closed

2/ Data passed in the body of the request

The template to update resources of which data to be modified are passed in the body of the request is: **PUT** host/{version}/{resource}.{mediatype}

Example:

- PUT host/v1/policies.xml

3.6 Resources Deletion

To delete existing resources, you need to use the **DELETE** or the **POST** method of the HTTP protocol with the URL of the resource to be deleted.

1/ Delete a single resource

The template to delete a single resource: **DELETE** host/{version}/{resource}.{mediatype}/{id}

Example:

- DELETE host/v1/customers.xml/30450

2/ Delete all resources

The template to delete all resources : **DELETE** host/{version}/{resource}.{mediatype}

Example:

- DELETE host/v1/documents.xml

3/ Delete a list of resources

The template to delete a list of resources: **POST** host/{version}/{resource}.{mediatype}/delete

In this case, we need to specify the action to be done, that is, delete verb in this case.

The identifiers of resources to be deleted are passed in the body of the request.

Example:

- POST host/v1/subscriptions.xml/delete

3.7 Resources Manipulation Summary

Rest API	Action Description	Request Body(Payload)	Response Body
POST host/{version}/{resource}.{mediatype}	Create a single or a collection of resource	One or a collection of resources to be created	Nothing
GET host/{version}/{resource}.{mediatype}/{id}	Read a single resource	Nothing	Resource concerned with its characteristics
GET host/{version}/{resource}.{mediatype}	Read a collection of resources	Nothing	Collection of resources with their characteristics
GET host/{version}/{resource}.{mediatype}?param1=value1&param2=value2	Search (single or collection of resources)	Nothing	Response may contains a collection of resources with their characteristics
POST host/{version}/{resource}.{mediatype}/search	Search (single or collection of resources)	Search criteria	List of found resources with their characteristics
PUT host/{version}/{resource}.{mediatype}/{id}?param1=value1&param2=value2	Update a single resource	Nothing	Nothing
PUT host/{version}/{resource}.{mediatype}	Update a collection of resources	List of resources: identifiers and information to be modified	Nothing
DELETE host/{version}/{resource}.{mediatype}/{id}	Delete a single resource	Nothing	Nothing
DELETE host/{version}/{resource}.{mediatype}	Delete all resources	Nothing	Nothing
POST host/{version}/{resource}.{mediatype}/delete	Delete a list of resources	List of the identifiers of the resources to be deleted	Nothing

3.8 Sub-resources Creation

The life cycle of a sub-resource is dependent of the lifecycle of the parent resource.

Examples of resources and sub-resources:
- Product / Product Covers
- Policy / Policy Covers
- Customer / Customer Bank Accounts

To create sub-resources, you need to use the **POST** method of the HTTP protocol with the URL of the sub-resource to be created.

The template for creation of resources is:

POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}

Sub-resources to be created are passed in the body of the request and the parent resource identifier is passed in the URL

Example:
- POST host/v1/customers.xml/123765/bank-accounts.xml

3.9 Sub-resources Reading

To retrieve sub-resources, you need to use the **GET** method of the HTTP protocol with the URL of the sub-resource to be retrieved.

1/ Read a single sub-resource

The template to retrieve a single sub-resource:

GET host/{version}/{resource/{id}/{sub-resource}.{mediatype}/{id}

Example:
- GET host/v1/documents/01280/files.xml/001

2/ Read all sub-resources

The template to retrieve all sub-resources:

GET host/{version}/{resource/{id}/{sub-resource}.{mediatype}

Example:
- GET host/v1/documents/01280/files.xml

3.10 Sub-resources Update

To update existing sub-resources, you need to use the **PUT** method of the HTTP protocol with the URL of the sub-resource to be updated.

There are two different ways to update existing sub-resources:
- Update a sub-resource whose data to be modified are embedded in URL

- Update a sub-resource or a collection of sub-resources whose data to be modified are passed in the body of the request

1/ Data embedded in the URL

The template to update a sub-resource of which data to be modified are embedded in URL is:

PUT host/{version}/{resource/{id}/{sub-resource}.{mediatype}/{id}?param1=value1

Example:

- PUT host/v1/policies/125689/covers.json/01?satus=closed

2/ Data passed in the body of the request

The template to update sub-resources of which data to be modified are passed in the body of the request is:

PUT host/{version}/{resource/{id}/{sub-resource}.{mediatype}

Example:

- PUT host/v1/policies/125689/covers.json

3.11 Sub-resources Deletion

To delete existing sub-resources, you need to use the **DELETE** or the **POST** method of the HTTP protocol with the URL of the sub-resource to be deleted.

1/ Delete a single sub-resource

The template to delete a single sub-resource:

DELETE host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/{id}

Example:

- DELETE host/v1/documents/30450/files.xml/01

2/ Delete all sub-resources

The template to delete all sub-resources:

DELETE host/{version}/{resource}/{id}/{sub-resource}.{mediatype}

Example:

- DELETE host/v1/documents/0112/files

3/ Delete a list of sub-resources

The template to delete a list of sub-resources:

POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/delete

In this case, we need to specify the action to be done, that is, delete verb in this case.

The identifiers of sub-resources to be deleted are passed in the body of the request.

Example:

- POST host/v1/documents/0112/files.xml/delete

3.12 Specific Methods

It is not uncommon that the actions to be done cannot map directly to the CRUD actions built around resources or cannot be associated to a given resource. In this case you need to use the verbs that are specific to your the context.

Examples of specific actions:

- Validate a subscription or a bank account
  - POST host/v1/subscriptions.xml/validate
- Send an email
  - POST host/v1/emails.json/send
- Compose a document
  - POST host/v1/documents.json/compose
- Count a number of occurrences of a resource
  - POST host/v1/products.xml/count

3.13 Sub-resources Manipulation Summary

Rest API	Action Description	Request Body (Payload)	Response Body
POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}	Create a single or a collection of sub-resources	One or a collection of sub-resources to be created	Nothing
GET host/{version}/{resource/{id}/{sub-resource}.{mediatype}/{id}	Read a sub-resource	Nothing	Sub-resource concerned with its characteristics
GET host/{version}/{resource/{id}/{sub-resource}.{mediatype}	Read a collection of sub-resources	Nothing	Collection of sub-resources with their characteristics
PUT host/{version}/{resource/{id}/{sub-resource}.{mediatype}/{id}?param1=value1	Update a sub-resource	Nothing	Nothing
PUT host/{version}/{resource/{id}/{sub-resource}.{mediatype}	Update a list of sub-resources	List of sub-resources to be modified and the id of the parent resource	Nothing
DELETE host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/{id}	Delete a sub-resource	Nothing	Nothing
DELETE host/{version}/{resource}/{id}/{sub-resource}.{mediatype}	Delete all sub-resources	Nothing	Nothing
POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/delete	Delete a list of sub-resources	List of identifiers of sub-resources to be deleted	Nothing

4 Other Topics

4.1 Pagination and Sorting

4.1.1 ***Pagination***

Pagination allows to limit the amount of data returned by the rest services and by consequent avoid the performance issues.

By splitting vast dataset into discrete pages, Rest services allow clients to scroll through and access the entire dataset in manageable chunks.

There are different pagination styles but we describe here only the two common, well understood and easy to implement.

1/ Page Number Pagination Style

The template to manage this style of pagination is:

GET host/{version}/{resource}.{mediatype}?page={page_number}&size={page_size}

The page_number parameter indicates the number of the page and the page_size parameter indicates the maximum number of resources to return.

Example:

- GET host/v1/products.xml?page=3&size=20

Retrieving 20 products of the page number 3, so from the 41th to the 60th product, if exists.

2/ Limit Offset Pagination Style

The template to manage this style of pagination is:

GET host/{version}/{resource}.{mediatype}?limit={element_number}&offset={starting_point}

The limit parameter indicates the maximum number of resources to return and the offset parameter indicates the starting point for the return data.

Example:

- GET host/v1/customers.xml?limit=10&offset=30

Retrieving 10 customers starting from the customer number 31.

We recommend using the page number pagination style.

4.1.2 ***Sorting***

Sorting allows the clients of the Rest services to determine the order in which resources in a dataset are arranged.

The Rest services should also allow the clients to specify the sort directions: ascending or descending.

The template to manage sorting is:

GET host/{version}/{resource}.{mediatype}?sort={attribute_name},asc&sort={attribute_name},desc

Example:

- GET host/v1/products.xml?sort=id,asc&sort=status,desc

Retrieving products sorted by ascending id and by descending status.

4.2 Exception Management

HTTP status codes play an important role in REST based web services. It is important to provide helpful , and detailed information regarding the errors in the response body.

Detailed error information can help the APIs consumers to understand issues easily and help them to recover.

The format of response error should include at least the following information:

- Type: functional or technical error
- Code: contains the HTTP status code in the response body
- Short description or Title: provides a brief title for the error. For example errors resulting as a result of input validation will have the title “Validation Failure” for a functional error, and, an “Service Unavailable” will be used for a technical error
- Long description or Detail: contains more description of the error. This information should be human-readable and can be presented to an end user
- Date Time: Timestamp, time in milliseconds when the error occurred
- Reasons: intended for developers. It contains information such as exception class name or stack trace that is relevant to developers
- Errors: code and message of the error: used to report field validation errors

Below a table with the commonly used HTTP status codes.

	Status Code	Description and Message
Success	200 OK	Description: Basic success code
	201 Created	Description: Resource was created
Client Error	400 Bad request	Description: general errors for a request that cannot be processed. Messages: - Resource data pre-validation error. - Resource data invalid. - The paging limit exceeds the allowed number - Resource collection including additional attributes error
	403 Forbidden	Description: rights are not sufficient to access this resource Message: - Access denied
	404 Not Found	Description: Resource requested does not exist. Message: - Resource not found.
	405 Method Not Allowed	Description: method is not supported on this resource or user does not have the permission Message: - Resource does not support method. - Resource method not implemented yet.
Server Error	500 Internal Server Error	The request seems right, but a problem occurred on the server. Message: - Resource internal error.

Example of exception in xml/xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<ExceptionInformation xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ExceptionInformation.xsd">
	<Type>FE</Type>
	<Code>400</Code>
	<ShortDescription>validation failed</ShortDescription>
	<LongDescription>input validation failed</LongDescription>
	<DateTime>2001-12-17T09:30:47Z</DateTime>
	<Reasons>org.springframework.web.bind.MethodArgumentNotValidException</Reasons>
	<Error>
		<Field>Name</Field>
		<Code>not null</Code>
		<Message>Name may not be null</Message>
	</Error>
	<Error>
		<Field>BirthDate</Field>
		<Code>format not correct</Code>
		<Message>day and month of birth are mandatory</Message>
	</Error>
</ExceptionInformation>
```

Example of exception in json

```
{
      Type  : "FE",
      Code :  "400",
      ShortDescription : “validation failed”,
      LongDescription : “input validation failed”,
      DateTime :12358979,
      Reasons :  “org.springframework.web.bind.MethodArgumentNotValidException”,
      Error : [ {
                Field : “Name”
                Code : “not null”
                Message : “Name may not be null”
             },
              {
              Field : “BirthDate”
                Code : “format not correct”
                Message : “day and month of birth are mandatory”
             }
           ]
}
```

4.3 Security Management

SESAME is a TSP (Transverse and Strategic Platform) in charge of the management of authentication and authorization needs within BNP Paribas Cardif.

Rather than having each application maintains its own user database with usernames, accounts and passwords, it seems more appropriate to use a centralized approach to identity management.

Therefore access to any application and using its resources has to go through Sesame platform. The principle consists in generating a token for a client application and its verification by a server application within Sesame.

There is an STA (Solution Technical Application) which describes the functioning of Sesame and the services provided.

A new version of Sesame is planned and it will include an implementation of SAML2.0 specification

Another specification OAuth2 exists. It already has widespread adoption by companies like Google, Facebook, Salesforce, and Twitter to name a few.

OAuth2 provides a simpler and more standardized solution which covers the same needs and avoids the use of workarounds for interoperability with native applications (mobile applications).

<u>Remark:</u>

Until now, it is not expected to provide an implementation of OAuth2 in Sesame.

5 Rest Services Tooling

5.1 JAX-RS Specification

JAX-RS stands for Java API for REST Web Services. It is a Java API for creating web services with REST.

This specification defines a set of Java APIs for the development of web services built according to the Representational State Transfer (REST) architectural style.

5.2 Server Side Development Frameworks

5.2.1 *Spring Rest*

Spring REST is a project part of the spring portfolio projects. It is intended for designing and developing REST APIs using the spring framework.

Spring Rest is based on the Jersey JAX-RS reference implementation from Oracle.

One of the Spring Framework's goals is to reduce plumbing code so that developers can focus on their efforts on implementing core business logic.

5.2.2 *CXF*

CXF is a framework which completely implements the JAX-RS 2.0 including Client API.

5.2.3 *RESTEasy*

RESTEasy is the JBOSS's JAX-RS Implementation.

5.3 Testing Tools

5.3.1 *SoapUI*

SoapUI is one of the best tools available to test a REST API. SoapUI goes well beyond basic status code testing and allows for very complex data driven testing of the given API as well as a slew of options for asserting response data. It supports multiple environments, multiple schema(xml, html, json), wsdl (soap), wadl(rest), dynamic variables, mock testing, load testing, security testing, database connectivity and testing, conditional logic and reporting.

5.3.2 *Postman*

Postman is a Google Chrome browser extension for making HTTP requests. It provides a lot of features that makes it easy to develop and test a REST API.

Postman's features are only a subset of SoapUI's features but it is very lean, easy to use and supports many known web API's out of the box via shared community projects (Twitter, Facebook, Google Drive, etc.). It also allows to save a Postman project file and share it by uploading to the postman server. It is a really neat feature when developers plan on exposing a newly created web API to other developers.

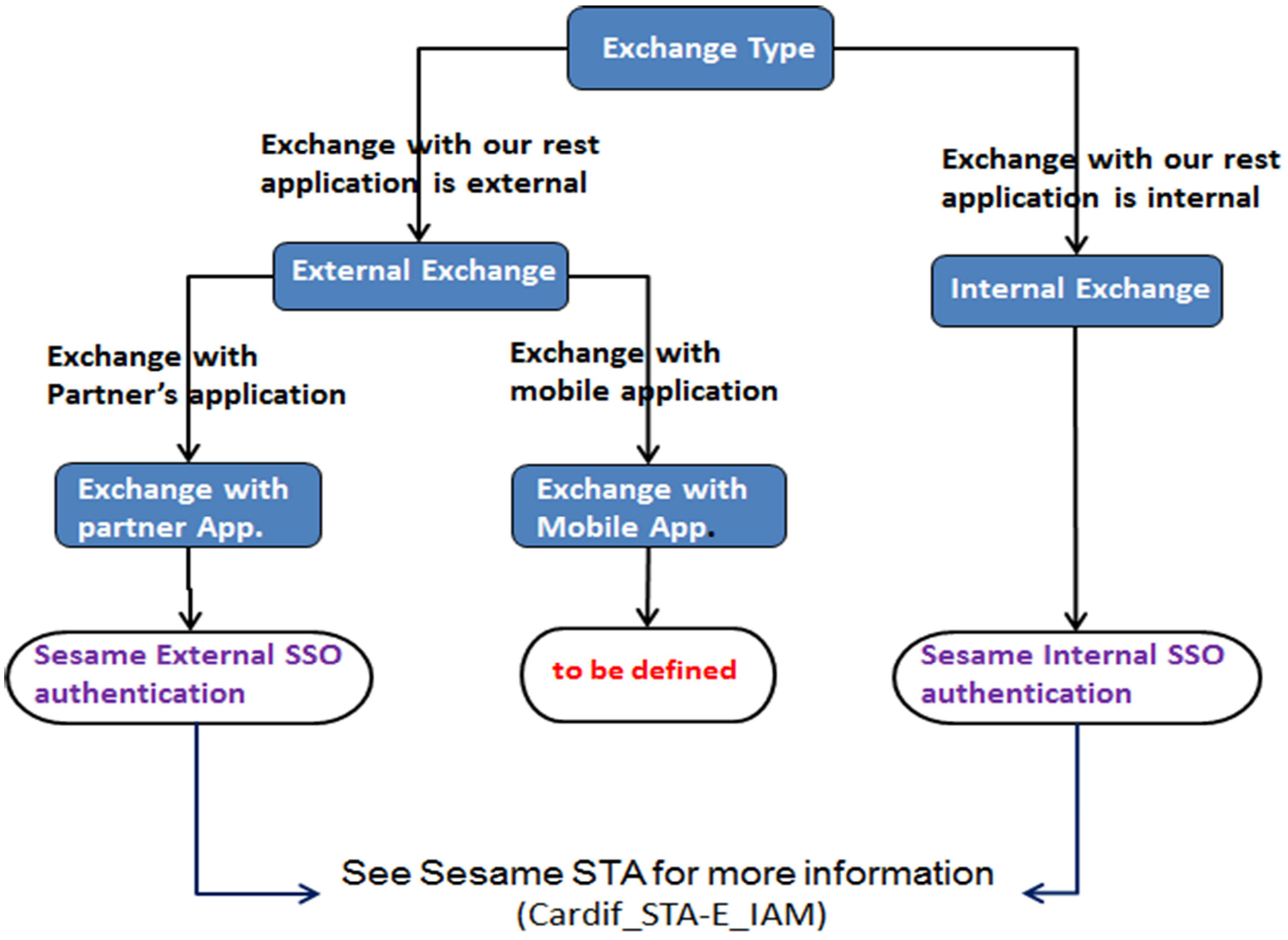
5.4 Client Side Development Frameworks

There are also client side development frameworks that help in building the requests and communicating with server side services.

They are not mentioned here because they are out of scope of this document.

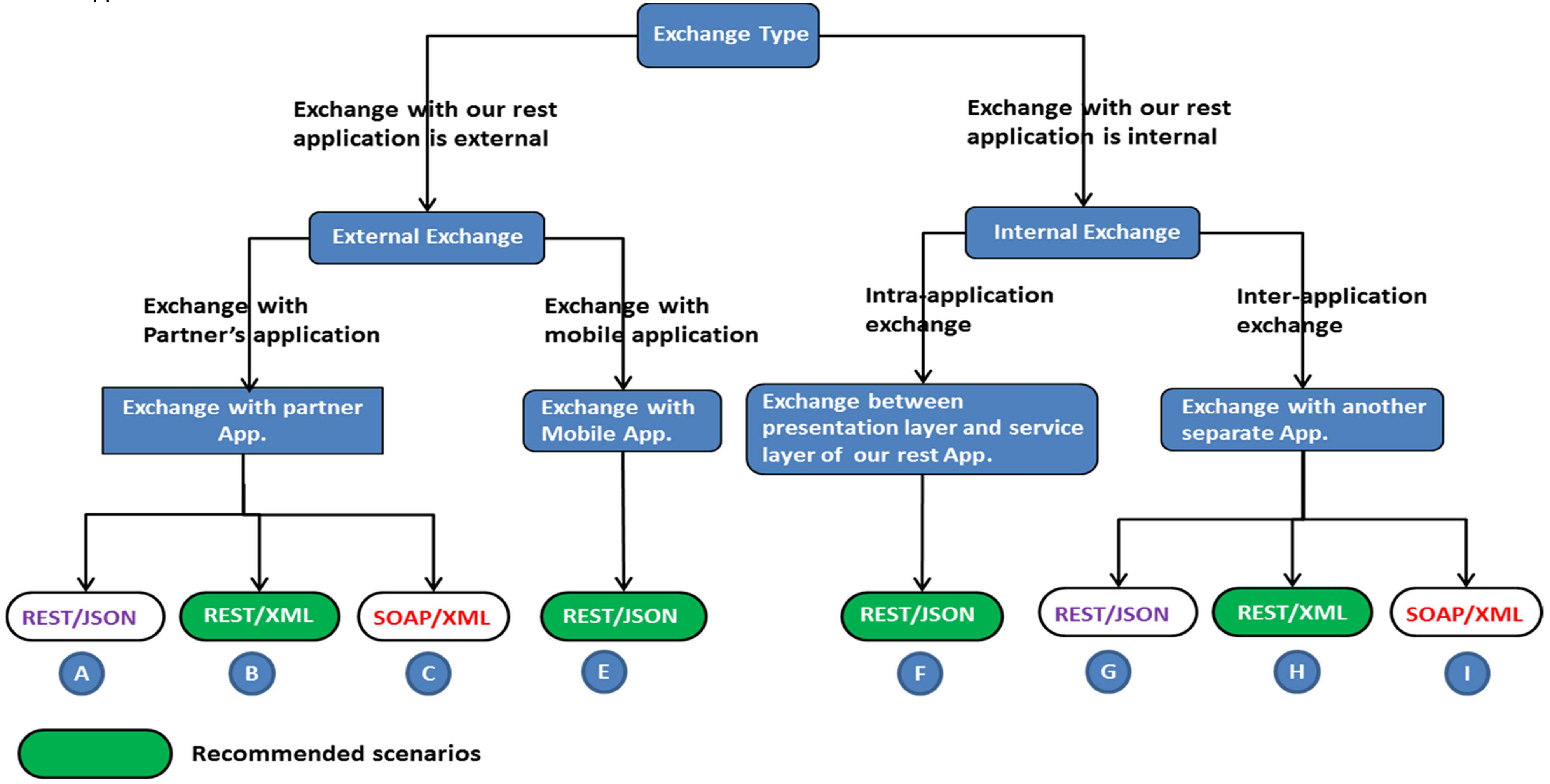
6 Rules & Guiding Principles

6.1 Rest Services Security Decision Tree


Exchange Type
Exchange with our rest application is external
Exchange with our rest application is internal
External Exchange
Internal Exchange
Exchange with Partner's application
Exchange with mobile application
Exchange with partner App.
Exchange with Mobile App.
Sesame External SSO authentication
to be defined
Sesame Internal SSO authentication
See Sesame STA for more information (Cardif_STA-E_IAM)

6.2 Rest Services Usage Decision Tree

The decision tree, given below, shows how an application exposing rest web services should communicate with other internal, external or mobile applications.


Exchange Type
Exchange with our rest application is external
Exchange with our rest application is internal
External Exchange
Internal Exchange
Exchange with Partner's application
Exchange with mobile application
Intra-application exchange
Inter-application exchange
Exchange with partner App.
Exchange with Mobile App.
Exchange between presentation layer and service layer of our rest App.
Exchange with another separate App.
REST/JSON
REST/XML
SOAP/XML
REST/JSON
REST/JSON
REST/JSON
REST/XML
SOAP/XML
A
B
C
E
F
G
H
I
Recommended scenarios

A - Partner App uses rest/json → Our rest App provides a rest/json exposition

B - Partner App uses rest/jxml → Our rest App provides a rest/xml exposition

C - Partner App uses soap/xml → **Option_1:** Our rest App provides a rest/xml exposition, and an intermediation layer provides a soap/xml exposition with an integration component in charge of the translation soap/xml - rest/xml. **Option_2:** Our rest App should provide also soap/xml exposition

E - Mobile App uses rest/json → Our rest App provides a rest/json exposition

F - Presentation and Service layers of our rest App use rest/json for communication

G - Internal App uses rest/json → Our rest App provides a rest/json exposition

H - Internal App uses rest/xml → Our rest App provides a rest/xml exposition

I -Internal App uses soap/xml → **Option_1:** Our rest App provides a rest/xml exposition, and an intermediation layer provides a soap/xml exposition with an integration component in charge of the translation soap/xml - rest/xml. **Option_2:** Our rest App should provide also soap/xml exposition

6.3 Services Documentation, Publication & API Management

Documenting REST services for consumers to use and interact with is difficult task because there are no real established standards.

With SOAP based services, a WSDL serves as a contract for the client and provides detailed descriptions of the operations and associated request/response payloads. The WADL, Web Application Description Language specification tried to fill this gap in the REST Web services world, but it didn't get a lot of adoption.

There is also no standard for the publication of the rest services versus SOAP UDDI directory.

Recently there has been a growth in the number of tools and products for describing, discovering REST web services and managing APIs.

The APIs management products include other functionalities such as:

- Securing the access to APIs
- Controlling the use of APIs using quotas
- Billing the use of APIs
- Managing the life cycle of APIs
- Etc..

Examples of tools that offer solutions for documenting REST services and managing APIs are: Swagger, 3scale, IBM, Mashape, Apigee, Restlet, WSO2 API management.

In the case where there is no standard tool or product that meets your requirements, you can rely on manually edited documents to expose REST contracts to your consumers.

6.4 API Scope & Versioning Management

A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE resources.

Various application programming interfaces (APIs) are now using REST web services as a logical choice and way for building APIs that allow end users and client applications to connect and interact with remote services. RESTful APIs are used by many sites, including Google, Amazon, Twitter and LinkedIn.

Number of Versions
The number of versions supported by an API should be limited to only two major versions. The deprecation of old versions must always take place only after informing the concerned consumers and communicating the retirement deadline.

Versions that will no longer be maintained need to be retired.

Backward Compatibility
A new version of a rest API contract that continue to support consumers designed to work with old version, is considered backward compatible. This means that old API capabilities are still present and continue to support the old combinations of method, resource identifier and media type.

<u>Examples of compatible changes:</u>

- Adding new capability with an existing method and new resources that were not previously available
    - Existing: GET v1/orders.xml/id
    - New: GET v1/orders/id/items.xml

- Adding new capability using existing resource but with new method
    - Existing: POST v1/orders.xml
    - New: GET v1/orders.xml

- Adding new supported media type to an existing capability using existing method and resource
    - Existing: GET v1/orders.xml/id
    - New: GET v1/orders.json/id

- Adding new optional XML elements, used in an operation input message, that don't need to be understood by old consumers

<u>Examples of incompatible changes:</u>

- Removing and old capability
    - Before
        - POST v1/orders.json
        - GET v1/orders.json
    - After
        - POST v1/orders.json

- Removing an old media type for a given capability
    - Before
        - POST v1/orders.xml
        - POST v1/orders.json
    - After
        - POST v1/orders.xml

- Adding new mandatory XML elements, used in an operation input message , that are not supplied by old consumers

<u>API scope and Version Change Rules</u>

There are two different categories of API to consider:

- API organized around major business objects such product, policy and customer
- API organised around business processes such as subscription, registration, or an after sales operation registration

All the capabilities within an API must have the same version.

Adding a new capability that doesn't break the backward compatibility doesn't require to change the current version of the API.

Changing or removing a capability that breaks the backward compatibility requires the creation of a new version of the API.

Remember that the number of versions to be supported should be limited to only two major versions.

6.5 Granularity of Requests To be handled

Operations of a rest service should be able to operate on a single or a collection of requests.

7 Operational usage of Rest Services

The rules and best practices which have been described throughout this document are in accordance and aligned with the principles of the Rest architectural style.

In practice, these principles could not be easily applied for several reasons such as:

- Being easily assimilated and understood by developers or people in charge of the implementation of these interfaces
- Existence of client frameworks that support only a limited subset of HTTP methods. It is not uncommon to see that some clients have only support for GET and POST and not for PUT and DELETE

Below, we provide new interfaces for both resource and sub-resources using only the HTTP methods GET and POST.

7.1 Resource Interfaces

Rest API	Action Description	Request Body (Payload)	Response Body (Payload)
POST host/{version}/{resource}.{mediatype}/create	Create a single or a collection of resource	One or a collection of resources to be created	Nothing
GET host/{version}/{resource}.{mediatype}/{id}/get	Read a single resource	Nothing	Resource concerned with its characteristics
GET host/{version}/{resource}.{mediatype}/getAll	Read a collection of resources	Nothing	Collection of resources with their characteristics
POST host/{version}/{resource}.{mediatype}/search	Search (single or collection of resources)	Search criteria	List of found resources with their characteristics
POST host/{version}/{resource}.{mediatype}/update	Update a collection of resources	List of resources: identifiers and information to be modified	Nothing
POST host/{version}/{resource}.{mediatype}/{id}/delete	Delete a single resource	Nothing	Nothing
POST host/{version}/{resource}.{mediatype}/deleteAll	Delete all resources	Nothing	Nothing
POST host/{version}/{resource}.{mediatype}/deleteList	Delete a list of resources	List of the identifiers of the resources to be deleted	Nothing

7.2 Sub-Resource Interfaces

Rest API	Action Description	Request Body (Payload)	Response Body (Payload)
POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/create	Create a single or a collection of sub-resources	One or a collection of sub-resources to be created	Nothing
GET host/{version}/{resource/{id}/{sub-resource}.{mediatype}/{id}/get	Read a sub-resource	Nothing	Sub-resource concerned with its characteristics
GET host/{version}/{resource/{id}/{sub-resource}.{mediatype}/getAll	Read a collection of sub-resources	Nothing	Collection of sub-resources with their characteristics
POST host/{version}/{resource/{id}/{sub-resource}.{mediatype}/update	Update a list of sub-resources	List of sub-resources to be modified and the id of the parent resource	Nothing
POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/{id}/delete	Delete a sub-resource	Nothing	Nothing
POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/deleteAll	Delete all sub-resources	Nothing	Nothing
POST host/{version}/{resource}/{id}/{sub-resource}.{mediatype}/deleteList	Delete a list of sub-resources	List of identifiers of sub-resources to be deleted	Nothing

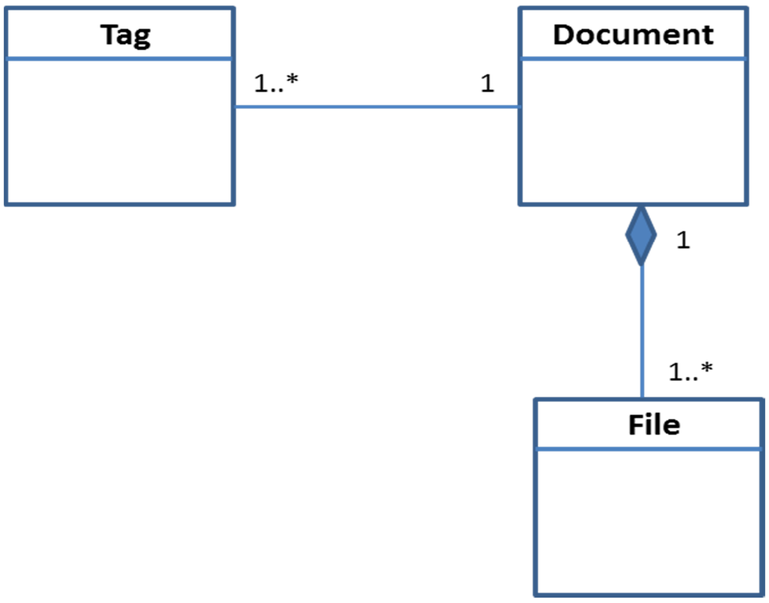
8 Example of API – Sugar Soap Services versus Rest Services

Sugar is a TSP (transverse and strategic platform) which represents the master and reference application in terms of documents storage; As such, it has a role to identify and store the master documents, but also to publish and notify all the status changes of reference objects under its responsibility (particularly the documents).

The core functionalities of sugar are:

- Storage of documents
- Searching and viewing of documents
- Modification of document metadata

8.1 Simplified Sugar Data Model



8.2 APIs In Accordance with Rest Architectural Style

Soap Service Name	Soap Operation	Underlying Concept	Operation Description	Rest API	Request Body (Payload)	Response Body (Payload)	Comment
sugar-document	store	Document	Store one or a collection of documents (including associated files, annotations, meta-data and contents) that must be provided in the request body	POST host/v1/documents.xml	Collection of documents	None	Content of files can be embedded in the message or can be provided as an attachment
sugar-document	reindex	Document	Update meta-data (indexation tags) of documents	PUT host/v1/documents.xml	List of documents : identifiers and meta-data to be modified	List of documents modified with their meta-data up to date	
sugar-document	get	Document	Get a specific document	GET host/v1/documents.xml/{id}	None	The document concerned with its characteristics	
sugar-document	find	Document	Find documents by providing flexible search criteria passed within the request body	POST host/v1/documents.xml/search	Search criteria	List of documents found with their meta-data	
sugar-document	delete	Document	Delete a specific document	DELETE host/v1/documents.xml/{id}	None	None	
sugar-document	findstreamed	Document	Find documents by providing flexible search criteria passed within the request body	POST host/v1/documents.xml?content-sending=streamed	Search criteria	Documents and related files found	

sugar-documentFile	add	File	Upload one or a collection of files related to a specific document	POST host/v1/documents/{id}/files.xml	Collection of files to be uploaded	None	Content of files can be embedded in the input message or can be provided as an attachment
sugar-documentFile	get	File	Get a specific file content	GET host/v1/documents/{id}/files.xml/{id}	None	The file concerned with its meta-data	
sugar-documentFile	delete	File	delete a specific file	DELETE host/v1/documents/{id}/files.xml/{id}	None	None	

8.3 APIs Using Only the POST and GET HTTP Methods

Soap Service Name	Soap Operation	Underlying Concept	Operation Description	Rest API	Request Body (Payload)	Response Body (Payload)	Comment
sugar-document	store	Document	Store one or a collection of documents (including associated files, annotations, meta-data and contents) that must be provided in the request body	POST host/v1/documents.xml /store	Collection of documents	None	Content of files can be embedded in the message or can be provided as an attachment
sugar-document	reindex	Document	Update meta-data (indexation tags) of documents	POST host/v1/documents.xml /reindex	List of documents : identifiers and meta-data to be modified	List of documents modified with their meta-data up to date	
sugar-document	get	Document	Get a specific document	GET host/v1/documents.xml /{id}/get	None	The document concerned with its characteristics	
sugar-document	find	Document	Find documents by providing flexible search criteria passed within the request body	POST host/v1/documents.xml /search	Search criteria	List of documents found with their meta-data	
sugar-document	delete	Document	Delete a specific document	POST host/v1/documents.xml /{id}/delete	None	None	
sugar-document	findstreamed	Document	Find documents by providing flexible search criteria passed within the request body	POST host/v1/documents.xml /search?Content-sending=streamed	Search criteria	Documents and related files found	
sugar-	add	File	Upload one or a collection of files related to a specific	POST host/v1/documents/{id}	Collection of files to	None	Content of files can be embedded in the

documentFile			document	/files.xml/store	be uploaded		input message or can be provided as an attachment
sugar-documentFile	get	File	Get a specific file content	GET host/v1/documents/{id}/files.xml/{id}/get	None	The file concerned with its meta-data	
sugar-documentFile	delete	File	delete a specific file	POST host/v1/documents/{id}/files.xml/{id}/delete	None	None	

9 SOAP & REST Services Comparisons

Underlying Topic	SOAP	REST
HTTP Usage	Used as a transport mode	Use as the backbone
HTTP Methods Used	POST	GET/POST/UPDATE/DELETE/etc.
Representation or Media Type	Mono-representation - XML	Multi-representation - XML/JSON/etc.
Orientation	RPC-Oriented	Resource-Oriented
Software Infrastructure Required	Stub / Skeleton	Nothing
Payload Schema Structure	Simple & Complex Structures	Only simple structures are recommended
Service Documentation & Contract	Static contract - WSDL serves as a contract for the client and provides detailed descriptions of the operations and associated request/response payloads	Dynamic contract - There is no standard for the publication of the rest services versus SOAP UDDI directory. But recently there has been a growth in the number of tools and products for describing, discovering REST web services and managing APIs.
Testing	SOAP services testing is hard because of the added abstraction	For consuming REST resources you only need an HTTP client
Comparison Summary	- Transport independent - Standardized (W3C) - Pre-build extensibilities in the form of the WS* standards -Require expensive processing	- REST requires use of HTTP - Interaction with web services doesn't require expensive tools - Easy to learn -Doesn't require extensive processing -Similar to other Web technologies in design philosophy

10 Glossary

- **REST:** Representational State Transfer. It is an architecture style for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines
- **URIs**: Uniform Resource Identifier. They are simply sequences of characters that conform to the standardized syntax defined by the IETF (an open standards organization)
- **URLs,** Uniform Resource Locators, are URIs that can be used in requests
- **URNs,** Uniform Resource Name, are URIs that can be used as unique identifiers for resources as business entities. Many identifiers in REST are URIs, URLs and URNs at the same time