# RESTful API design

## Best practices guide

| *Version* | *Date* | *Authors* | *Description* |
|---|---|---|---|
| 1.0 | 2017-12-09 | cln | Initial version (reviewed by pch, mbe) |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# Table of Contents

# Introduction

This guide is intended to architects and developers using the EBX RESTful API and is a set of best practices for developing REST services.

It is not intended to replace the EBX software documentation – the RESTful operations section in particular – nor some of the REST APIs accessible on the internet. You will find a few references on the matter in the appendices.

# Recommendations

## *Recommendation 1: URL Request*

A URL allows a client program to access a unique resource in order to consult it, update it, delete it, or perform any other type of action. A request of the REST API is composed of the following:

- a URL that corresponds to the access path of the resource,
- the HTTP method: [GET], [POST], [PUT], [DELETE], etc.,
- HTTP headers,
- URL parameters,
- an optional message body.

A response is composed of the following:

- an HTTP status code,
- HTTP headers,
- an optional message body.

A URL has to be encoded, for more information see Recommendation 1.4: The URL must be encoded.

An EBX REST URL is composed of:

```
[<method>]
https://<server>[:<port>]/<webapp>/rest/<serviceName>/<serviceVersion>/<pathToResource
>[:<actionName>][?((&)!<parameterName>=<parameterValue>)*]
```

where:

| | |
|---|---|
| `<method>` | HTTP method |
| `http[s]` | HTTP protocol (or HTTPS if secured in SSL) |
| `<server>` | IP address, localhost, DNS (Domain Name Server) |
| `[:<port>]` | (optional) IP conversation port (80 by default in HTTP or 443 in HTTPS) |
| `<webapp>` | Name of the core / addons / specific webapp. For core services, use: ebx-dataservices |
| `rest` | Root path of the URL to access the REST API of the webapp. The operation process is delegated to the mapped Java Servlet, either via web.xml or via a Java annotation (requires Servlet 3.0). |

| | |
|---|---|
| `<serviceName>/<serviceVersion>` | Name and version of the REST API service.<br><br>Corresponds to the category and version of the EBX core API. |
| `<pathToResource>` | Access path to a resource, the path is composed of names and does not include action verbs (in the grammatical sense). For each name, ask yourself if the returned resource is an object or a collection in order to name it accordingly (plural or not). |
| `[:<actionName>]` | (optional) If left blank, the default action will be used. |
| `<parameterName>=<parameterValue>` | Name and value of a parameter.<br><br>Recommendation 4: Request parameters |

Detailed examples:

```
[GET] http://host:port/ebx-dataservices/rest/data/v1/<dataspace>
```

Lists all datasets that the user can access (default action).

```
[GET] http://host:port/ebx-dataservices/rest/data/v1/<dataspace>:permissions
```

Lists all the permissions of the dataspace (extended action). We know we are going to retrieve a collection since the action contains an ‘s’.

```
[GET] http://host:port/ebx-dataservices/rest/data/v1/<dataspace>:information
```

Lists the dataspace information (extended action). It is expected to retrieve an object because the action does not contain an ‘s’.

```
[GET] http://host:port/ebx-
dataservices/rest/data/v1/<dataspace>/<dataset>/<tablePath>/<recordPK>:infor
mation
```

Returns the record information (extended action).

```
[GET] http://host:port/ebx-
dataservices/rest/data/v1/<dataspace>/<dataset>/<tablePath>/<recordPK>/infor
mation
```

Returns the value of the record information field (default action).

## Recommendation 1.1: Naming and versioning the service.

```
[GET]  http://server/webapp/rest/serviceName/serviceVersion/pathToResource:actionName?
parameterName=parameterValue
```

The service name should be picked according to a functional requirement, the version number allows to manage the service evolutions. Two types of evolutions exist:

- **minor** ensuring upward compatibility of the service API (addition of a new parameter, addition of information in the response that does not impact the behavior), it is not necessary to create a new version,

- **major** if the service API is no longer compatible with previous version (deleted parameter, modified structure of the message body), it is necessary to create a new version and to document what has been deprecated.

As a general rule, it is best not to support more than 2 versions, for maintenance cost reasons.

### Recommendation 1.2: Check the action name.

```
[GET]   http://server/webapp/rest/serviceName/serviceVersion/pathToResource:actionName?
parameterName=parameterValue
```

The action name should be functional, short, straightforward, and should not contain special characters. It should not be prefixed with « get » or « set », because it is the HTTP method that allows to determine the operation type. For more information, Recommendation 2.3: The action is conflicting with the method.

### Recommendation 1.3: Ensure that a URL corresponds to a single resource.

The URL should never be shared by multiple functional needs and should correspond to distinct object types.

### Recommendation 1.4: Strongly encode the URL.

For compliance reasons with the HTTP protocol, it is necessary to use an encoder for the URL parameters.

For more information, see the appendices: Strong encoding constraints for the URL and parameters.

# Recommendation 2: HTTP methods

The HTTP methods allow to define the operation executed in REST on EBX, for example:

- A selection operation correspond to the methods: [GET] or [POST],
- An insert operation corresponds to the method: [POST] with a content,
- An update operation corresponds to the methods: [PUT] for a unit operation, otherwise [POST],
- A delete operation corresponds to the methods: [DELETE] or [POST].
- For other type of operation [POST] is used.

A method is **idempotent** if, and only if, it can be executed several times without changing the server state. A method is **safe** if, and only if, no modification has been introduced to the resource.

More generally, HTTP methods can be used to:

| Method | Idempotent | Safe | Additional information |
|---|---|---|---|
| [GET] | ✓ | ✓ | List data: |

| | | | |
|---|---|---|---|
| | | | • All information is included in the URL (« path info » and parameters).<br>• This method is only for read-only requests.<br>• No request body.<br>• It is possible to create bookmarks in the browser. |
| [PUT] | ✓ | × | Update (or override) data:<br>• The URL resource is similar to the method [GET]. |
| [DELETE] | ✓ | × | Delete (or hide) data:<br>• The URL resource is similar to that of the [GET]. |
| [POST] | × | × | Modify data and any other complex request:<br>• In case of an HTTP 204 response (without content), the method can return the URL of the object created in the « location » HTTP header. |

RFC 7231: Common Method Properties

### Recommendation 2.1: Respect the safe aspect of a method.

The following example illustrates the disregard of the safe aspect of the HTTP method, provided that the method [GET] is defined to safe (does not modify), the creation of a reservation is thus forbidden:

```
[GET] /<serviceName>/<serviceVersion>/reservations:createReservation
```

In this case, the [POST] method should be used.

### Recommendation 2.2: Respect the idempotence of a method.

The example below is not idempotent because, with each call, a reservation is created:

```
[PUT] /<serviceName>/<serviceVersion>/reservations:createReservation
```

In this case, the [POST] method should be used.

### Recommendation 2.3: Check that the action is not conflicting with the method.

The example below illustrates a conflict with the method [GET]: it is not « safe » since it is deleting data:

```
[GET] /<serviceName>/<serviceVersion>/reservations/1234:delete
```

In this case, the [DELETE] method should be used.

## Recommendation 3: HTTP Headers

See the header documentation used by the EBX REST API:

http://dl.orchestranetworks.com/restricted/documentation/en/advanced/references/dataservices_rest_v1.html#id2s3

RFC 7231: Request and response header fields.

To illustrate the management of some headers, see the appendix Example core services headers for EBX.

### Recommendation 3.1: Do not create specific headers.

Only use standard headers. For more information:

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

Warning: some platforms might delete unknown headers.

### Recommendation 3.2: Do not replace a header by a specific parameter.

A standard header is generally better handled than a specific parameter.

### Recommendation 3.3: Track essential headers.

The call trace of the operation, at the INFO level, could include some headers that are required for supporting the REST API.

### Recommendation 3.4: Ignore unsupported headers.

When a transmitted header is not supported by the service operation, do not throw an error.

## Recommendation 4: Request parameters

A parameter should not correspond to a resource identifier and should not duplicate an HTTP header.

### Recommendation 4.1: Strongly encode the parameters.

For compliance reasons with the HTTP protocol, it is necessary to use an encoder for the URL parameters.

For more information, see the appendices: Strong encoding constraints for the URL and parameters.

### Recommendation 4.2: Avoid duplicating core parameters.

If the concept is similar, it is recommended to reuse the parameters defined in the core. Please refer to the software documentation and use the following interface:

`com.orchestranetworks.dataservices.rest.RestConstants`

### Recommendation 4.3: Ensure that sensitive data is not passed as a parameter.

Parameters are logged on the Web server and can be exploited maliciously by an administrator.

Sensitive parameters should be sent in the request body using [POST] for example.

### Recommendation 4.4: Control useful parameters.

Useful parameters should be controlled syntactically whenever possible, attention: semantic checks may require a subsequent control, or more specifically in the transaction.

### Recommendation 4.5: Track essential parameters.

The call log of the operation, at the INFO level, could include some parameters that are required for supporting the REST API.

### Recommendation 4.6: Ignore unsupported parameters.

When a transmitted parameter is not supported by the service operation, do not throw an error.

# Recommendation 5: Body content

This section applies to the request and response bodies of the REST API.

The service agreement of the response must include information related to the requested resource. A response can have at least two types of structures:

- Successful execution. The structure is open as long as these recommendations are respected:
    - Simple and efficient (more information would be useless, do not omit data that would be useful for the object lifecycle),
    - If links would help improving efficiency, add them with the management of the parameter « includeDetails » in order to be able to remove them if need be (Recommendation 5.10: Simplify the use of the API by returning URLs).
    - Some HTTP status codes do not require a response body, such as: 201 (Created) or 204 (No Content).
- An issue of the REST API, and corresponds to the error case that is supported or unsupported, with an HTTP status code other than 2xx.

### Recommendation 5.1: Support JSON format.

The content format should be in JSON and encoded in UTF-8.

RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format.

### Recommendation 5.2: Respect the design rules of the public API.

The structure of the messages of the REST API should be respected between the request and the response. This recommendation's purpose is to facilitate interoperability. As a consequence, the structure of a request object should correspond to that of the response object. However, this response object can be more detailed, but the structure itself should not be incompatible.

For more information on the public API, see the documentation:

http://dl.orchestranetworks.com/restricted/documentation/en/advanced/references/json_services.html

login: support
password: on@ebx67

## Recommendation 5.3: Limiting data volume.

Some requests could saturate the server's memory if they are incorrectly coded and could lead to an HTTP error status 500. If the flow does not go through the server's memory, it still uses the network.

When the volume of the message itself is unbounded by nature, a paging mechanism should be used. The volume of exchanged data will thus be under control. A paging context is required and described in the EBX documentation. It should be used to navigate with the « `nextPage` » key at the minimum.

http://dl.orchestranetworks.com/restricted/documentation/en/advanced/references/json_services.html#pagination

login: support
password: on@ebx67

## Recommendation 5.4: Avoid duplicating JSON core properties.

It is recommended to reuse the properties defined in the core if the concept is similar. To do so, see the software documentation, then use the interface:

`com.orchestranetworks.dataservices.rest.RestConstants`

## Recommendation 5.5: Use the standard response structure for errors.

This format is managed by the REST API.

Warning: The error can be returned by the server (404 or 500 generally) and thus might differ from the format described hereafter.

The structure of the error message is standardized so that the REST API remains uniform.

```
{
  "userMessage": "...", // Mandatory localized message
  "errorCode": "XXX",   // EBX error code (optional, used mainly for HTTP error 422)
  "errors": [           // Internal messages useful when debugging (optional).
                        // Usually not displayed to the user and not localized.
               "Message 1", "Message 2" }
  ]
}
```

## Recommendation 5.6: Make sure that permissions are granted.

Avoid security breaches by always checking user's permissions.

For example, if permissions grant access to at least one sub-group of the data, this sub-group will be serialized. Other sub-groups will be missing, in case permissions would block access to those.

## Recommendation 5.7: Make sure that no stack is returned.

In case of an error, managed or non-managed, this error should appear in the server log. By no means should the error stack be added to the body of the JSON response. The structure of the standardized error message should be used in this case (cf .#2.7.1.Recommendation 7.1 : The 'managed' and 'non-managed' error cases.|).

## Recommendation 5.8: Simplify the use of the API by returning URLs.

The HATEOAS approach of the EBX REST API also allows to enjoy an intuitive and straightforward navigation, this implies that the information lacking details could be obtained through an URL. These information URLs must be provided in keys called « details ».

Example of a REST API that uses HATEOAS (warning: the actions do not follow our recommendations):

https://developer.paypal.com/docs/api/overview/#hateoas-links

### *Recommendation 5.9: Track the content of requests.*

The track of the requests' content on the server side should be able to be activated in DEBUG mode. Caution, in case of large volume, they should be truncated.

It should also be possible to consider the log level in real time. Consequently, the support team will be able to have access to full logs if need be.

## *Recommendation 6: HTTP status code*

The quality of a REST API can be measured according to how well it respects the HTTP protocol that is returning a status code indicating that the execution has been successful or aborted for any known or unknown reason. The status codes are standardized and thus offer a processing logic to client applications accessing it.

The HTTP status code is represented by a 3-digit number that follows these spaces:

- 1xx: Information: The server is processing, please wait...
- 2xx: Success: The server returned the expected result.
- 3xx: Redirection: Content has moved, check somewhere else (see « location » header).
- 4xx: Client error: Incorrect request, from the client end.
- 5xx: Server error: The request causes an uncontrolled function, from the server end.

http://dl.orchestranetworks.com/restricted/documentation/en/advanced/references/dataservices_rest_v1.html#httpCode

login: support
password: on@ebx67

In addition to the EBX 5.8.1 codes used, we will use for the future release:

| HTTP code | Description |
|---|---|
| 422 (Unprocessable entity) | Used for functional error. |

The HTTP status codes will soon become a new passion, for more information, go to:

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

The development of a REST API in Java is assisted by the use of one of the hierarchies of the following classes:

```
 * REST abstract checked exception. Each implementation is attached
 * to a corresponding HTTP client error code (4xx).
```

`com.orchestranetworks.dataservices.rest.RestException_Checked` inherits from `Exception`

and

```
 * Corresponds to the REST unchecked exception that generates a
 * server error (5xx).
```

`com.orchestranetworks.dataservices.rest.RestException_Unchecked` inherits from `RuntimException`

These state codes have been defined by the RFC 2616, then extended in the RFC 7231.

### Recommendation 6.1: Restrict the use of the status code 500.

If a REST API returns a status code of the 500 type, it is a technical error from the server (example: the database is not accessible, OutOfMemory and Java error).

Note: In case of a Java error, it is recommended to check that it is not a pending bug that should be fixed.

Example: In case of data not found, the service should return a code `404 (Not found).`

### Recommendation 6.2: Ensure the semantics of status codes.

According to the HTTP protocol, some status codes such as `201 (Created)` and `204 (No Content)` specify that they should not return a response body. If a response body is needed, the status code `200 (Ok)` should be used.

## Recommendation 7: Implementation

### Recommendation 7.1: Respect the Stateless architecture.

No context should be recorded on the server corresponding to a client session. This architecture ensures the fault-tolerant operation. For more information, see the appendix section References.

### Recommendation 7.2: Respect the transaction type vs the HTTP method.

EBX offers two types of transactions on a dataspace:

- `com.orchestranetworks.service.Procedure` for read or update,
- `com.orchestranetworks.service.ReadOnlyProcedure` for read-only.

An HTTP method can be « safe » or « non safe », for more information, see Recommendation 2.1: Respect the safe aspect of a method.

A « non safe » HTTP method should not use a `ReadOnlyProcedure` transaction.

# Conclusion

This best practices guide, intended for the REST API design, will undoubtedly allow to positively improve the quality of our REST APIs. They will be more compliant with the market standards and more consistent between themselves (core, add-ons and specific modules). This will highly facilitate the integration of EBX within businesses' systems, mobile applications, and other applications...

The R&D Integration team is ready to help you validate your first RESTful services with EBX. We also remain available should you have any comments or suggestions on how to improve this guide.

# Appendices

## *References*

**Architectural constraints**

https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_constraints

**EBX RESTful API documentation**

http://dl.orchestranetworks.com/restricted/documentation/en/advanced/index.html
login : support
password : on@ebx67

**Google cloud APIs / Design guides**

https://cloud.google.com/apis/design

**HATEOAS**

https://en.wikipedia.org/wiki/HATEOAS

**Hypertext Transfer Protocol (HTTP/1.1)**

RFC 7230, Message Syntax and Routing

RFC 7231, Semantics and Content

RFC 7232, Conditional Requests

RFC 7233, Range Requests

RFC 7234, Caching

RFC 7235, Authentication

RFC 7236, Authentication Scheme Registrations

RFC 7237, Method Registrations

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

**Uniform Resource Identifier (URI)**

RFC 3986, Generic Syntax

**The JavaScript Object Notation (JSON)**
RFC 7159, Data Interchange Format

## Core headers supported by EBX

Management of locales:

| | |
|---|---|
| Request.Header | Accept-Language |
| Response.Header | Content-Language |

Authentication:

| | |
|---|---|
| Request.Header | Authorization |
| Response.Header | WWW-Authenticate |

Example for a record creation:

| | |
|---|---|
| Response.Header | Location |

Example of request headers:

```
[GET] http://host:port/ebx-dataservices/rest/data/v1/<dataspace>
Authorization: Basic YWRtaW46YWRtaW4=
Accept: application/json
Accept-Language: fr,en;q=0.8,fr-FR;q=0.5,en-US;q=0.3
Content-Type: application/json;charset=UTF-8
```

Example of response headers:

```
Status code 200 'OK'
Server: Apache-Coyote/1.1
Content-Type: application/json;charset=UTF-8
Content-Language: fr-FR
Content-Length: 1285
Date: Mon, 13 Nov 2017 13:02:08 GMT
```

## Body uses « content » JSON property name

The implementation of properties called « content » in the JSON messages follows the formatting constraint and the necessity to add attributes to exchanged data. Consequently, the value is not directly added as a value of a JSON property, but instead is set at an intermediary level. By following this recommendation, the addition of other values (attributes) is possible, such as:

- label: JSON String, contains the data label
- details: JSON String, contains the URL for more details on the data
- selector: JSON String, contains the URL providing the possible values
- inheritance: JSON String (enumeration), contains the inheritance type of the data
- validation: JSON Array, contains JSON Objects with the severity of the message and the attached message itself
- etc.

Example of a record selection in JSON:

```
{
  "label": "Claude Lévi-Strauss",
  "details": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/individu/1",
  "content": {
    "id": {
      "content": 1
    },
    "gender": {
      "content": "M",
      "label": "M",
      "selector": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/individu/1/gender?
selector=true"
    },
    "prenom": {
      "content": "Claude"
    },
    "patronyme": {
      "content": "Lévi-Strauss"
    },
    "birthDate": {
      "content": "1908-11-28"
    },
    "deathDate": {
      "content": "2009-10-30"
    },
    "jobs": {
      "content": [{
        "content": "11",
        "label": "anthropologue",
        "details": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/nomenclatures/job/11",
        "selector": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/individu/1/jobs?
selector=true"
      },
      {
        "content": "76",
        "label": "ethnologue",
        "details": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/nomenclatures/job/76",
        "selector": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/individu/1/jobs?
selector=true"
      }]
    },
    "memberOf": {
      "content": [{
        "content": "ACA",
        "label": "académie française",
        "details": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/nomenclatures/institutions/
ACA",
        "selector": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/individu/1/memberOf?
selector=true"
      }]
    },
    "distinctions": {
      "content": [{
        "content": {
          "distinction": {
            "content": "12",
            "label": "1. grand-croix de la légion d'honneur",
            "details": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/nomenclatures/distinction/1
2",
            "selector": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/individu/1/distinctions/dis
tinction?selector=true"
          },
          "date": {
            "content": "1973-05-24"
          }
        }
      },
      {
        "content": {
          "distinction": {
            "content": "17",
            "label": "médaille d'or du CNRS",
            "details": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/nomenclatures/distinction/1
7",
            "selector": "http://localhost:8080/ebx-
dataservices/rest/data/v1/Bpersonnalities/genealogie/root/individu/1/distinctions/dis
tinction?selector=true"
          },
          "date": {
            "content": "1967-01-01"
          }
        }
      }]
    },
    "infos": {
      "content": [{
        "content": "http://fr.wikipedia.org/wiki/Claude_L%C3%A9vi-Strauss"
      }]
    }
  }
}
```

## Strong encoding constraints

For compliance reasons with the HTTP protocol, it is necessary to use an encoder for the URL and its parameters. Standard encoding implies identified security threats for some of our clients. Consequently, it is highly recommended to use a strong encoder, that is:

In Java, the following method performs this encoding:

`com.orchestranetworks.dataservices.rest.RESTEncodingHelper.encode(String)`.

The detailed algorithm is described in the JavaDoc.