BNY Mellon NEXEN REST API Guidelines Version 1.0 July 2017

1 Document Purpose

This document covers the guidelines for designing RESTful style APIs. It represents best practices gleaned from studying industry-standard APIs.

The intended audience of this document is a given Line of Business (LOB) that is exposing either an existing service or implementing a new service as a RESTful API that will be used in a B2B or B2C scenario as a part of the NEXEN API program.

This document references the following subject areas that are involved in implementing a RESTful API. Many of these details are evolving and are referenced here to provide clarity. Details of these subject areas are out of scope for this document and will be covered by a separate set of documents.

- **APIs**
  APIs expose services and data to internal and external developers using RESTful API interfaces. APIs leverage the API infrastructure which is composed of:
  - **API Store**
    The API Store implements a catalog for all published APIs and is accessible to developers to be consumed and integrated into their applications. All APIs include consistent documentation via Swagger 2.0.
  - **API Designer**
    The API Store counterpart is the API Designer / Publisher tool where BNY Mellon service providers can publish their APIs to be used by both internal BNY Mellon developers as well as external customers. Mediations may also be designed with this tool, including simple payload mediations as well as full SOAP to REST mediations.
  - **API Gateway**
    The API Gateway exposes RESTful API endpoints for customers. It performs authentication and authorization via the Key Manager, quality of service via throttling, optional caching via response caching, and usage data for business analytics and billing.
  - **Key Manager**
    The Key Manager implements standard OAuth 2.0 authentication / authorization functionality via the front-end façade URI. This is largely transparent to the service provider.
  - **OpenID Connect 1.0 Authorization Server**
    OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It allows clients to verify the identity of the end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user.
  - **ESB Gateway (Mediation)**
    If needed, services will be normalized to a RESTful API interface using a mediation layer. The ESB can provide protocol mediation (e.g., SOAP to REST), security mediation, payload format mediation, and more to retrofit legacy services to become REST-compliant. The ESB Mediation layer will eventually leverage the data dictionary, thus providing a consistent payload interface.
  - **Governance Registry**
    Services can be registered using a Governance Registry. This is primarily for externalizing / governing the service endpoints for each of the SDLC environments, for example: Dev, Test, QA, and Production.
- **Security – Authentication / Authorization**
  (a) Front-end Security (see Key Manager above).
  (b) Back-end Security – Implemented either by a service using a service-specific solution or by using security mediation for legacy services via the ESB (see ESB Gateway above).
- **Services**
  A service, sometimes also called an abstraction layer, is the layer where business functionality is encapsulated. Services are either (a) underlying existing services, using heterogeneous interfaces e.g., RESTful, SOAP, and MQ, or (b) newly developed services that are natively RESTful.
- **Common Data Dictionary**
  A common data dictionary will be developed for all APIs. A common data dictionary will enable developers to define payload schemas based upon standardized attribute names.

2 RESTful API Guidelines

2.1 REST Principals

There are many documents that explain Representational State Transfer (REST) and this introduction touches on some of the basics, highlights best-practices, and addresses common use case patterns.

REST is an architectural style and is embodied as a set of architectural principles, or *REST constraints*, characterized as follows:

2.1.1 Addressable Resources

A key abstraction in REST is a resource, which represents an entity and is addressed by a URI (Uniform Resource Identifier). Employing a standardized address scheme enables a portable and scalable way to find a resource.

A URI is standardized as follows:

URI = Scheme “:” Authority “/” Path [“?” Query] [“#” Fragment]

- **Scheme** is the communication protocol and is one of: http | https. However, https is required for improved security.
- **Authority** is the host and port. It represents the base location of a given resource.
  Authority = Host ":" Port

- **Path** is a set of text strings, or path segments, delimited by “/”.
  Path = PathSegment1 [["/" PathSegment2] [... "/" PathSegmentN]]
  The RESTful path structure is broken down as:
  "/" Context "/" Version "/" Resource
- **Query** is a list of parameters specified as name=value pairs. Each pair is delimited by the “&” character.
  Query = "?" Name1 "=" Value1 ["&" Name2 "=" "Value2"[..."&" "NameN "=" ValueN]]
- **Fragment** is delimited by “#” and points to a specific location within the document. For RESTful APIs that represent services, fragments are not used and only mentioned here for completeness.

Example: https://apigateway.bnymellon.com/asset-coverage/v1/funds?coverageDate=2017-06-20T19:20:09.610Z&expand=ApprovedFunds

In this example, the parts are:

Scheme: https
Authority: apigateway.bnymellon.com
Path: /asset-coverage/v1/funds

> **Context**: asset-coverage
> **Version**: v1
> **Resource**: funds

Query: ?coverageDate=2017-06-20T19:20:09.610Z&expand=ApprovedFunds

> **Name1**: coverageDate
> **Value1**: 2017-06-20T19:20:09.610Z
> **Name2**: expand
> **Value2**: ApprovedFunds

2.1.2 A Uniform, Constrained Interface

An important part of the complete REST interface is the standard set of *methods* that interact with resources:

- **GET – Retrieve a resource**
  Used to query the server. It is both an idempotent and safe operation. There should not be any side effects, other than possible meta-data (e.g., last accessed timestamp). The idempotent property enables various caching strategies to be leveraged by browsers and proxy servers.

- **PUT – Update a resource**
  Updates the representation of an entity. The updated representation is passed in the message body. It should contain a complete representation of the object. Partial PUTs are not allowed. The semantic of PUT is modeled as an update. It is also idempotent.

- **DELETE – Delete a resource**
  Removes a given entity of a resource. It is also idempotent.

- **POST – Create a new resource**
  Creates a new entity. POST is both a non-idempotent and an unsafe operation.

- **OPTIONS – Query capabilities**
  Allows the requester to retrieve the capabilities of a server without any effect on a resource. This is generally not needed for standard REST calls except for implementing pre-flight CORS requests.

Some of the advantages of uniform interfaces are familiarity, interoperability, and scalability. See more about methods in **2.4 HTTP Methods for RESTful APIs** below.

2.1.3 Representation-oriented

Interaction with a service takes place using resource representations of that service. From the client point of view, the **Content-Type** header specifies the data format of the representation for describing input, while the **Accept** header specifies the data format of the representation for describing output. The client and server can negotiate the representation format using these headers to indicate a media type (also known as MIME).

The most commonly used media type in APIs is **application/json**.

Others media types used are:

- application/x-www-form-urlencoded (for PUT and POST only)
- application/xml or text/xml
- application/javascript
- text/plain
- text/csv (facilitates “export” capability)

2.1.4 Stateless

Services are implemented as stateless.

There is no client session data stored in the service. The client is responsible for storing any session data. The stateless design helps in scaling services and reduces complexity on the server side since there is no need to replicate session state nor ensure session affinity.

2.1.5 Hypermedia as the Engine of Application State (HATEOAS)

Hyperlinks enable references to additional data. Embedded hyperlinks in the data also enable discovery of additional interactions that can subsequently be performed. This REST constraint has several possible implementations. Because the solution to this constraint is not completely yet standardized, these API Guidelines deem it to be optional.

2.2 Uniform Resource Identifier (URI)

A Uniform Resource Identifier (URI) is used to identify a resource. See https://tools.ietf.org/html/rfc6570.

Note: The following examples are for front-end façade URIs. The exact base path for a given front-end URI is determined as a part of the API Publishing task. Front-end URIs can map to any backend URIs up to the point of the first resource.

Backend URIs that are the endpoints of the actual service implementation should follow a similar pattern.

URI = Scheme “:” Authority “/” Path [“?” Query] [“#” Fragment]

For the internal services that are developed for NEXEN Access Cloud (BXP), the following is the suggested URI pattern:

Scheme: https

Authority:

- **Dev**: apigateway.dev.example.com
- **Test**: apigateway.test.example.com
- **QA**: apigateway.qa.example.com
- **Production**: apigateway.example.com

Path: /<context>/<version>/<resource>

2.2.1 URI Examples

Given:

context = positions
version = v1
resources = holdings, transactions

Dev:

https://apigateway.dev.example.com/positions/v1/holdings
https://apigateway.dev.example.com/positions/v1/transactions

Test:

https://apigateway.test.example.com/positions/v1/holdings
https://apigateway.test.example.com/positions/v1/transactions

QA:

https://apigateway.qa.example.com/positions/v1/holdings
https://apigateway.qa.example.com/positions/v1/transactions

Production:

https://apigateway.example.com/positions/v1/holdings
https://apigateway.example.com/positions/v1/transactions

2.3 Versioning

RESTful API versioning is a bit different than traditional versioning of libraries or maven artifacts. This is due to the fact that (a) REST APIs are always hosted and there is an obligation to maintain a version for a certain period of time, and (b) the nature of REST APIs is that they are very loosely coupled to the implementation. For those reasons, versions are not expected to change often. For a REST API, the version is embedded in the API's base URI.

A subset of Semantic Versioning is used and is specified here: http://semver.org. For our purposes, the MAJOR version in the semantic versioning scheme is sufficient.

Precede the version number with the letter “v**”**, for example, “v1”.

Whenever backward incompatible API changes are introduced, the version number should be incremented. Conversely, for backward *compatible* changes, do not change the version number.

The MINOR and PATCH parts of the full semantic version are not included in the path, but may be optionally supported via HTTP headers: **Version-Minor** and **Version-Patch**. But in general, the MINOR and PATCH should be irrelevant to the API consumer. Using version numbers in URLs forces incompatible changes to be communicated to clients and forces clients to make required changes before using the new API.

Example: https://apigateway.example.com/positions/v2/holdings

As new versions of an API are released, at least one previous version should be maintained. An old version that is still being supported can be explicitly called, as in, https://apigateway.example.com/positions/v1/holdings.

2.4 HTTP Methods for RESTful APIs

2.4.1 Safety and Idempotency

Method	**Safe?** ***No side effects***	**Idempotent?** ***Request can be repeated with the same result***	**Description**
GET	Yes	Yes	Read an entire resource collection or resource entity
PUT	No	Yes	Update a single resource entity
DELETE	No	Yes	Delete a single resource entity
POST	No	No	Create a single resource entity

2.4.2 HTTP Methods Applied to Specific Entities and Collections

Method	**Specific Entity** **e.g., /holdings/12345**	**Entire Collection** **e.g., /holdings**
GET	200 OK 404 Not Found	200 OK *An array is returned.*
PUT	200 OK 404 Not Found	404 Not Found *It is not desirable to update every resource in a collection.*
DELETE	200 OK 204 No Content 404 Not Found	404 Not Found *It is not desirable to delete an entire collection.*
POST[1]	404 Not Found *Use PUT to update an item.*	201 Created 202 Accepted (*for job workflows*)

[1] Sometimes POST is used as a *catch-all* method for special cases that do not fit any of the standard use cases.

2.4.3 Special Cases

- **POST for long-running request**s. Use a “job” or “task” to wrap the response. See 2.7.3.2 Modeling Asynchronous below.
- **POST for sensitive data**. To ensure that sensitive data (e.g., credentials) is encrypted via HTTPS, always use POST so that the sensitive data is not in the URL which otherwise could be exposed in web logs or a browser cache.
- **POST for complicated queries**. POST can be used when it would be too cumbersome to use query parameters with GET. For example, supposing the use case is to search for several customers simultaneously and each customer is represented as **First Name**, **Last Name,** and **ZIP Code**. Then, the input is an array of objects which is cumbersome to represent in a *flattened* way represented by query parameters.

2.5 Media Types

2.5.1 Content-Type Header

To describe the request input payload format and / or response payload output, use the HTTP **Content-Type** header.

Minimally support this media type:

- application/json

Optionally support these media types:

- application/x-www-form-urlencoded
- application/xml

If a request call includes a **Content-Type** header that indicates an unsupported media type, a *415 Unsupported Media Type* error should be returned.

2.5.2 Accept Header

To request a specific response format (content negotiation) other than the default, use the HTTP **Accept** header.

Minimally support this media type:

- application/json

Optionally support these media types:

- application/xml
- text/csv
- text/plain

If a request call includes an **Accept** header that indicates an unsupported media type, a *406 Not Acceptable* error should be returned.

2.6 URI Template

A URI Template is a compact sequence of characters for describing a range of Uniform Resource Identifiers through variable expansion. This specification defines the URI Template syntax and the process for expanding a URI Template into a URI reference, along with guidelines for the use of URI Templates: https://tools.ietf.org/html/rfc6570.

2.7 Resource Modeling

> *“The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g., "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g., a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.”*
>
> *- Roy Fielding’s dissertation.*

Resources can be:

- **A collection**, for example, “/customers”.
- **A single entity** within a collection, also known as a document, for example, “/customers/C12345”

A resource may contain sub-collection resources. This is desirable to minimize the number of API calls needed to perform a task and thus reduce “chattiness”. On the other hand, embedding too many sub-collections in a response may make the response payload large and unwieldy. The right balance needs to be considered. One possible solution is to include a query parameter to indicate whether or not the sub-collection should be expanded in the response payload. For example, **?expand=true**.

Sometimes there will be functionality in the service that defies being factored into resource nouns. This is sometimes true of actions, for example, calculate, translate, convert, etc. For this situation, verbs are used, but they are a rare exception.

2.7.1 Resource Naming

- Resource names should be simple, single-word nouns that are pluralized.
- When applicable, the resource identifier should be in the URI template – not a query string parameter.
- Use lower case with hyphens to separate words, but try to use a single word whenever possible.

- Use the resource path hierarchy to signify structure and scoping.

Example - Good:

/customers/C12345/orders/O12345/items/I12345

Example - Bad:

/customer?customerId=C12345&orderId=O12345&itemId=I12345

2.7.2 Resource Usage

Where relationships are complex and deeply nested, it is not advisable to have a resource structure that is more than 3 levels deep.

Do not use a trailing forward slash (/) in URIs.

Ideally there should be only 2 base URLs per resource. The first URL is for a collection; the second is for a specific element in the collection. For example:

- /transactions
- /transactions/<id>

Resource	GET	PUT	DELETE	POST
	Read	Update	Delete	Create
/transactions	List transactions.	*Bulk update is generally not done.*	*Bulk delete is generally not done.*	Create a new transaction.
/transactions/<id>	Retrieve the transaction entity referred to by **id**.	Update the transaction entity referred to by **id**.	Delete the transaction entity referred to by **id**.	Create a new transaction with the supplied **id.** *This is generally not done as the system should assign the ID. This pattern is reserved for another use case (partial PUT), so if an ID must be supplied, then an HTTP header should be used.*

Use the query component of a URI to filter collections. For example: GET /positions?asOf=2017-06-20

Use the query component of a URI to paginate a collection. For example: GET /positions?offset=0&limit=10

Use path variables for a direct ID belonging to a collection. For example, to retrieve an account with ID 65248: GET /accounts/65248

Use parameters for non-direct IDs. For example: GET /positions?isdn=35346546

Do not include file extensions in a URI.

2.7.3 Modeling Special Patterns

2.7.3.1 Modeling Workflows

There are situations where an event or action changes the state of an object, and subsequent new events change the state again. For this situation, a REST workflow model works well. The recommended solution is to include an "/actions" sub-resource. When an action is POSTed to the "/actions" sub-resource, the state can be changed.

For example, suppose you were trying to model a vacation approval workflow:

Step	Action	State
1	Employee requests time off	Request pending
2	Manager reviews request	Request pending
3	Manager queries available actions	Request pending
4	Manager approves request Manager denies request (Go to step 6)	Request approved Request rejected

5	Employee checks status of request (then goes to Tahiti)	Request approved
6	Employee continues to work	Request rejected

Step 1 Request – *Request some time off*
POST /vacations

```
{
"startDate": "2017-06-26",
"numberOfDays": "5"
}
```

Step 1 Response
HTTP/1.1 201 CREATED

```
{
"id": 4455,
"employeeComitId": "xbbxx86",
"employeeName": "Maxwell Smart",
"startDate": "2017-06-26",
"numberOfDays": "5",
"state": "pending"
}
```

Step 2 Request – *Get a list of all pending vacation requests*
GET /vacations?state=pending

Step 2 Response
HTTP/1.1 200 OK

```
[
{
"id": 4455,
"employeeComitId": "xbbxx86",
"employeeName": "Maxwell Smart",
"startDate": "2017-06-26",
"numberOfDays": "5"
}
]
```

Step 3 Request – *Get a list of available actions for a given request item*
GET /vacations/4455/actions

Step 3 Response
HTTP/1.1 200 OK

```
[
{"action":"Approve"},
{"action":"Reject"}
]
```

Step 4 Request – *Manager grants the time off for the employee*
POST /vacations/4455/actions

```
{
"action": "Approve"
}
```

Step 4 Response
HTTP/1.1 200 OK

```
{
"id": 4455,
"employeeComitId": "xbbxx86",
"employeeName": "Maxwell Smart",
"startDate": "2017-06-26",
"numberOfDays": "5",
"state": "approved"
}
```

Step 5 Request – *Employee checks to see if the request was approved or denied*
GET /vacations/4455

Step 5 Response
HTTP/1.1 200 OK
```
{

"id": 4455,
"employeeComitId": "xbbxx86",
"employeeName": "Maxwell Smart",
"startDate": "2017-06-26",
"numberOfDays": "5",
"state": "approved"
}
```

2.7.3.2 Modeling Asynchronous

A RESTful API call should return fairly quickly. Sometimes, there are requests that take a long time to execute. For this situation, an asynchronous job model works well.

For example, if a long-executing report is run, it can be wrapped in a job and represented this way:

Step 1 Request
POST /reports

<report data>

Step 1 Response
HTTP/1.1 202 Accepted
Location: /jobs/12345

```
{

"links": {
"status": {
"method": "GET",
"href": "/jobs/12345"
}
}
}
```

The location URI refers to a resource where the asynchronous job processing status can be subsequently queried:

Step 2 Request
GET /jobs/12345

Step 2 Response
HTTP/1.1 200 OK
```
{

"status": "Pending",
"estimatedCompletionTime": 10000,
"links": {
"cancel": {
"method": "DELETE",
"href": "/jobs/12345"
}
}
}
```

The "cancel" link is an optional link that when present allows the user to delete the job before processing has begun.

Once the job has started, a subsequent call would look like this:

Step 3 Request
GET /jobs/12345

Step 3 Response
HTTP/1.1 200 OK
```
{

  "status": "Running",
  "estimatedCompletionTime": 8000
}
```

When the job has completed:

Step 4 Request
GET /jobs/12345

Step 4 Response
HTTP/1.1 200 OK
{

```
"status": "Completed",
"links": {
"result": {
"method": "GET",
"href": "/reports/3343"
}
}
}
```

2.7.3.3 Cloning an Entity

Sometimes it's desirable to create a new entity based upon an existing entity with a few new values.

Here is the recommended pattern for cloning an existing object:

POST
/<collection>?source=/<collection>/<cloneFromId>&<attribute1>={newAttribute1Value}&<attribute2>={newAttribute2Value}...&<attributeN>={new AttributeNValue}

For example, suppose there is a collection named "products", and it's desirable to create a newer version of the product that is mostly the same as the original:

Request
GET /products/123

Response
HTTP/1.1 200 OK
{

```
"id": 123,
"name": "BoringWidget",
"price": 50.00,
"description": "Just a widget",
"class": "widgets"
}
```

Request
POST /products?source=/products/123&price=70.00&name=LatestAndGreatest

Response
HTTP/1.1 201 CREATED
{

```
"id": 456,
"name": "LatestAndGreatest",
"price": 70.00,
"description": "Just a widget",
"class": "widgets"
}
```

2.8 Disallowed URL Characters

Due to cross-site vulnerability reasons, the following characters should not be used in a URL:

Bad CSS Characters	<,>,',%22	These are considered to be cross-site scripting characters. SiteMinder will block URLs containing these. A SiteMinder exception to these restrictions requires IRM approval.
Bad Form Characters	<,>,&,%22	Characters used in HTML Forms that should be blocked by SiteMinder. A SiteMinder exception to these restrictions requires IRM approval.

Bad URL Characters and Character Sequences	/,./,/.,/*,*.,~,\,%00-%1f,%7f-%ff,%25,%25U,%25u	These are considered "bad" URL characters. A SiteMinder exception to these restrictions requires IRM approval.

2.9 Partial Response

Pagination and partial fields response allows developers to ask for exactly just the information they need.

2.9.1 Pagination

It's almost always a bad idea to return every resource in a given database table, assuming that table is not fixed and contains a potentially large number of rows.

There should be a reasonable attempt of consistency in results for collections that change often, given that each query is independent and stateless.

Use query parameters **limit**, **offset**, **sortKey**, and **dir** (**asc** or **desc**). These names are common, well understood, and the concepts are present in database interfaces and therefore easy for developers to use. The parameters **offset** and **limit** are zero-based.

The **offset** parameter defines the number of skipped records, and **limit** defines the number of records to be returned. For example, ?offset=100&limit=50 means that we are expecting the service to return any records that exist within the inclusive range 101-150 from the result set.

When the **limit** parameter is not present, a given service should choose a reasonable default limit. For some services, the default might be unlimited. For example, ?offset=100 would mean return all records starting with record 101. For other services, the default limit might be something finite such as 100. For example, ?offset=100 would mean return records that exist in the inclusive range 101-200.

When the **offset** parameter is not present, the default is 0.

The parameter **sortKey** specifies a field or field list, by which to sort first, before applying **offset** or **limit**.

The parameter **dir** specifies the sort direction – **asc** for ascending or **desc** for descending. The default sort direction is ascending.

2.9.2 Partial Fields Response

In order to facilitate selection of which fields to include in the response, a partial fields response specifier can be used with the **fields** query parameter. The recommended syntax is a subset of Google's partial response syntax which is inspired by the XPath syntax.

For example:

```
GET /positions?fields=accountName,isin,cusip,localPrice
```

Additionally, a sub-selector can be used to request a set of specific sub-fields of arrays or objects by placing elements in parentheses following the array or object name, as in:

```
GET /holdings-filters?fields=name,assetTypes(code, description)
```

2.9.3 Simple Filtering

Simple equality filtering can be achieved by simply representing each attribute as a query parameter directly in the query parameter string as in:

GET /holdings?clearingInstitution=BNYMELLON

2.9.4 Advanced Filtering

For more advanced filtering such as using relational operators as well as compound expressions, a simple filtering expression language that is easy to encode in a query string is recommended:

2.9.4.1 Advanced Filtering Operators

Operator	Description	Example

EQ	Equal	GET /holdings?filter=clearingInstitution EQ 'BNYMELLON'
NE	Not Equal	GET /holdings?filter=accountId NE 'FSW'
GT	Greater Than	GET /holdings?filter=marketValue GT 100000
GE	Greater Than or Equal	GET /holdings?filter=priorDateMarketValue GE 120000
LT	Less Than	GET /holdings?filter=newBalance LT 1000000
LE	Less Than or Equal	GET /holdings?filter=newBalance LE 1000000
AND	Logical And	GET /holdings?filter=marketValue GT 100000 AND marketValue LT 150000
OR	Logical Or	GET /holdings?filter=marketValue LE 10000 OR marketValue GT 200000

2.9.4.2 Advanced Filtering BNF

```
<expression>     ::= <term> | <term> <connector> <term>

<term>           ::= <variable> <operator> <constant>

<operator>       ::= "EQ" | "NE" | "GT" | "GE" | "LT" | "LE"

<connector>      ::= "AND" | "OR"

<constant>       ::= <anystring> | <number> | <boolean>

<anystring>      ::= <quotedstring> | <string>

<quotedstring>   ::= "'" <string> "'"

<string>         ::= <character> | <string> <character>

<character>      ::= <lowercharacter> | <uppercharacter>

<lowercharacter> ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

<uppercharacter> ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

<boolean>        ::= "true" | "false"

<number>         ::= <integer> | <pointfloat>

<pointfloat>     ::= <integer> "." <integer>

<integer>        ::= <digit> | <integer> <digit>

<digit>          ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

2.10 Data Representation Formats

The default data representation is JSON (Content-Type: application/json).

Tip: Always use a standard library to stream to JSON to ensure the data is well-formed and is valid JSON (e.g., com.google.gson.Gson).

2.10.1 Attribute Names

Attribute names should be formatted in lowerCamelCase.

Attribute names, if query-able, should match their associated query parameter name exactly.

For example:

GET /positions?accountNumber=12345

should return:

```
{
"accountNumber" : 12345,
...
}
```

and not:

```
{
"acctNo" : 12345,
...
}
```

2.10.2 Null Values

For the Java value **Null** or C value **NULL** (0), use the text-based literal value “null”.

2.10.3 Date and Timestamp

Dates can use either Coordinated Universal Time (UTC) or an offset from the UTC, thus avoiding issues with time zones and daylight saving time.

Dates should be returned with a timestamp with a zero UTC offset in the format YYYY-MM-DDTHH:MMZ so that conversion to the respective time zone is accurate. For example: 2014-10-17T10:12:30Z.

Timestamp formats with seconds and milliseconds can be used for specific use cases: YYYY-MM-DDTHH:MM:SSZ or YYYY-MM-DDTHH:MM:SS.ssssssZ.

Additional details on Date and Timestamp formats are available at W3, ISO 8601 Wiki and ISO 8601.

2.10.4 Countries and Territories

Countries should be represented as 2-letter code. For example: US, DK, IN. For more information, see ISO_3166-1.

Subdivisions should be represented as a 2-letter code, followed by a hyphen and then followed by a 2-letter code. For example: WS-WA, US-CO, CA-BC, IN-AP. For more information, see ISO-3166-2.

2.10.5 Currencies

Currencies should be represented following ISO 4217 which specifies 3-letter currency codes for names of currencies.

Examples:

Entity	Currency	Alphabetic Code
AFGHANISTAN	Afghani	AFN
ALAND ISLANDS	Euro	EUR
ALBANIA	Lek	ALL
ALGERIA	Algerian Dinar	DZD
AMERICAN SAMOA	US Dollar	USD

A full list of alphabetic and numeric codes are available at List One.

2.10.6 Language Tags

Best practices for the language tag are available at W3C.

The IANA registry is the authoritative reference for sub tags.

Examples:

Code	Language	Subtag
en	English	language
es	Spanish	language
fr-CA	French as used in Canada	language+region

2.11 Data Validation

Validation of incoming payloads is performed by the backend service.

Validation may include:

- Minimum and maximum length.
- Enumeration facets validation.
- Special characters.
- Application-dependent inter-object validation rules.

Tip: Use a standard REST framework to implement the validation functionality such as **Spring MVC**, **Spring Boot Jersey** or **REST WS with Jackson**.

2.12 Error Handling

Use a consistent form across the application to represent errors. For example:

- Construct the errorCode using the format“<Application Mnemonic><code>”. For example: "MFD1001".
- A code can belong to a range to classify:
    - 0001-0999: Warning
    - 1000-2000: Invalid request or request parameters
    - 2001-3000: Error
- A Log file entry should include errorCode, userId, and timestamp. Log4j by default will include the timestamp.
- The error description is a user-friendly message. Due to security restrictions, it is not advisable to include text descriptions for internal errors, as a text description may expose internal implementations and could thusly be exploited.

Use HTTP status codes and map them cleanly to relevant standard-based codes.

2.13 Response Codes

Minimally, use the following HTTP Status codes:

Status Code	Status Message	Note
200	OK	
201	Created	Always use this for POST when POST creates a new entity.
202	Accepted	Use this in conjunction with POST when implementing the job workflow pattern.
204	No Content	Only use this for DELETE, but it is preferable to instead use a 200 with response content indicating status.
400	Bad Request	
401	Unauthorized	
404	Not Found	
406	Not Acceptable	
415	Unsupported Media Type	

500	Internal Server Error	Normally this is generated by the server and not the application code. Never expose stack exceptions in a response.

2.14 Standard Response Schema Wrapper

A standard response schema wrapper is recommended to separate the representation of the actual object and any additional metadata.

The format of the wrapper is as follows:

When returning a collection or subset (filtered) of a collection:

```
{

"metadata": {
},
"data": [
],
"pagination": {
}
}
```

When returning a single entity within a collection:

```
{

"metadata": {
},
"data": {
}
}
```

Example 1:

GET /products

```
{
"metadata": {
"status": "success",
"description": "string",
},
"data": [
{
"id": 1,
"name": "Product One",
"createdOn": "2017-05-26T00:25:07.542Z"
},
{
"id": 2,
"name": "Product Two",
"createdOn": "2017-06-21T00:25:07.542Z"
}
],
"pagination": {
"offset": 0,
"limit": 100,
"total": 2
}
}
```

Example 2:

GET /products/2

```
{
"metadata": {
"status": "success",
"description": "string",
},
"data": {
"id": 2,
"name": "Product Two",
"createdOn": "2017-06-21T00:25:07.542Z"
}
}
```

Note that for brevity, the examples in this document do not show this full wrapper.

3 Terms and Abbreviations

The table below provides a central location for all terms and abbreviations used throughout this document. These terms may include legal, technical or specific operational / business terms, which need to be clarified, confirmed, or described to this document's audience.

Term	Meaning
LOB	Line of Business
JSON	JavaScript Object Notation
CORS	Cross-Origin Resource Sharing
ESB	Enterprise Service Bus
JWT	JSON Web Token
OIDC	OpenID Connect

4 Reference Documents

Document Name	Brief Description	Source
Web API Design	This document was written by Brian Mulloy from Apigee and it covers most of the standards for RESTful web services.	https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf
RESTful Web APIs	Book by Leonard Richardson, Sam Ruby, Mike Amundsen	http://shop.oreilly.com/product/0636920028468.do
REST API Design Rulebook	Book by Mark Masse	http://shop.oreilly.com/product/0636920021575.do
RESTful Java with JAX-RS 2.0	Book by Bill Burke	http://shop.oreilly.com/product/0636920028925.do
RESTful Web Services Cookbook	Solutions for Improving Scalability and Simplicity Book by Subbu Allamaraju	http://shop.oreilly.com/product/9780596801694.do
SOA with REST	Book by Thomas Erl, Benjamin Carlyle, Cesare Pautass, Raj Balasubramanian	https://www.amazon.com/SOA-REST-Principles-Constraints-Enterprise/dp/0137012519

5 Notices

BNY Mellon is the corporate brand of The Bank of New York Mellon Corporation and may be used as a generic term to reference the corporation as a whole and/or its various subsidiaries generally. Products and services may be provided under various brand names in various countries by