Next Gen ALM Public Beta

REST API Documentation – Tech Preview

Table of Contents

Disclaimer

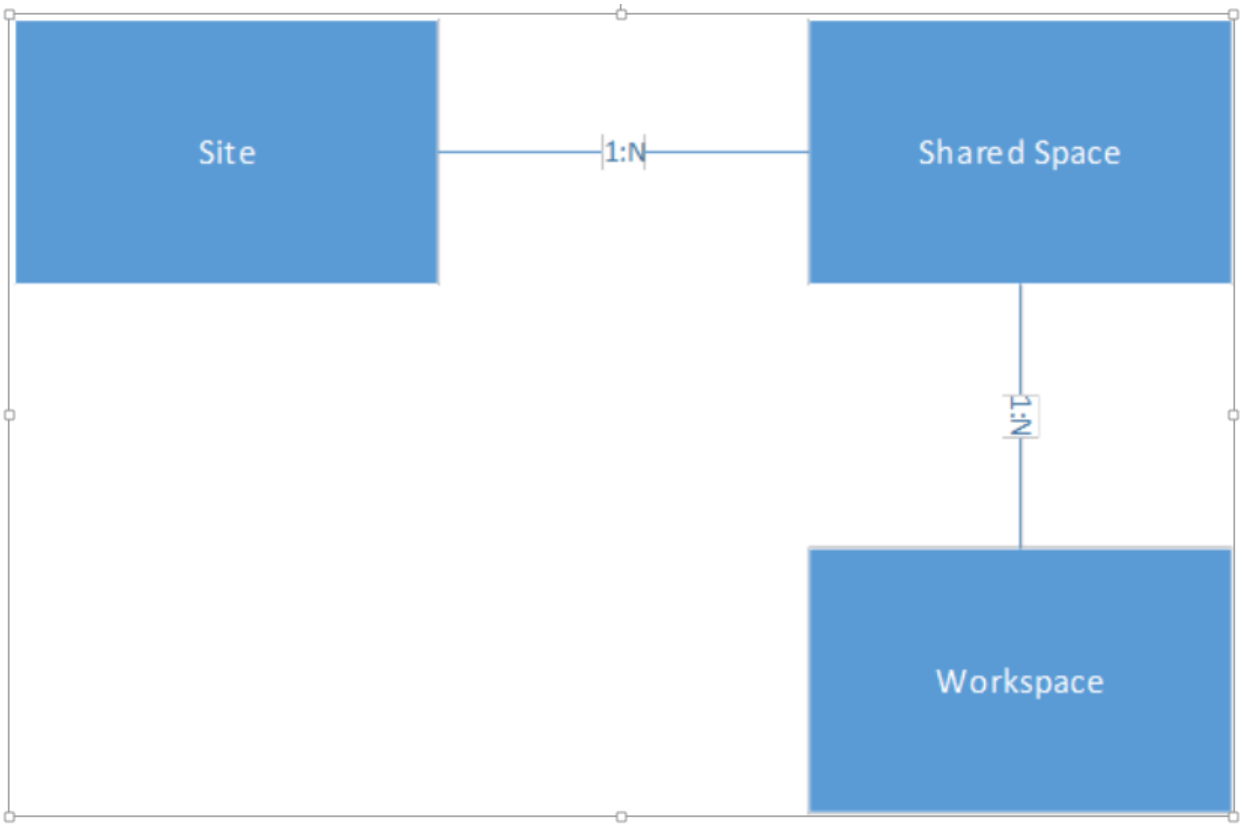
All specifications in this document are in tech preview sate, which can be changed until an API is declared as public.

Introduction

The NGA Model

NGA mostly is an entity - relationship application.
NGA's public REST API purpose is to serve CRUD operations on the various entities and relations between them.

NGA model consists of a *site*, which contains shared spaces.
Each shared space represents, for example a business unit.
Each shared space can contain workspaces. A workspace represents, for example a product within the business unit.



According to this, we have 3 contexts:

- Site
- Shared Space
- Workspace

We will reflect those 3 contexts as 3 entry points in REST API.

Site	http[s]://<server>:<port>/admin/*	N/A in SaaS
Shared Space	http[s]://<server>:<port>/api/shared_spaces/{uid}/*	
Workspace	http[s]://<server>:<port>/api/shared_spaces/{uid}/workspaces/{id}/*	

Any entity in the system exists in one of those contexts.

NOTE: Workspace is an entity (resource) and is located in a shared space context, as can be seen from the workspace context entry point, which fits perfectly the model.

NOTE: Shared space is also represented as an entity. It exists in the *site* context: /admin/shared_spaces

NOTE: */api/shared_spaces* can be accessed either by *id* (numeric) or by *uid = logical_name* (string)

Representation

Currently the API supports only *application/json* representation both on request and response (ECMA-404 standard)

The Metadata REST API

NGA public REST API is fully metadata driven. All the entities described by the metadata resources can be accessed via the REST API as resource collection, which will be described later.

Entity Metadata	<context>/metadata/entities	Entity Metadata - REST API
Fields Metadata	<context>/metadata/fields	Fields Metadata - REST API

Example:
http[s]://<server>:<port>/api/shared_spaces/j8216rp9e0l15co4elzody9m7/workspaces/2001/metadata/fields

NOTE: Each relation is represented as reference field on both of the entities which are involved in the relation

Resource Collection

Each entity described in the *entities* metadata REST API can be accessed with the plural form of the entity name as first level resource in the relevant context.

For example, a *test* is an entity in workspace context. So accessing it (GET) will be the following:

http[s]://<server>:<port>/api/shared_spaces/j8216rp9e0l15co4elzody9m7/workspaces/2001/tests

The response will be of the following JSON format:

```
{
    "data": [
        {
            "id": 1001,
            "type": "test",
            "name": "this is a regression test…",
            …
        },
        {
            "id": 2005,
            "type": "test",
            …
        },
        …
    ]
}
```

The fields of the *entity* are as defined by the *fields* metadata REST API for an *entity*.

Resource collection resource supports the following HTTP methods:

GET	Retrieve the data	Supports filtering, field selection, sorting, pagination
POST	Create new entities	
PUT	Update existing entities	Supports update by filter
DELETE	Delete existing entities	Supports delete by filter

Resource Instance

Each entity described in the *entities* metadata REST API can be accessed directly by *id*.

For example, a *test* is an entity in workspace context. So accessing it (GET) will be the following:

http[s]://<server>:<port>/api/shared_spaces/j86rp9e0l15co4elzody9m7/workspaces/2001/tests/2005

The response will be of the following JSON format:

```
{
    "id": 2005,
    "type": "test",
    "name": "testing steering wheel",
    …
}
```

The fields of the *entity* are as defined by the *fields* metadata REST API for an *entity*.

Resource instance resource supports the following HTTP methods:

GET	Retrieve the data	Supports field selection
PUT	Update existing entity	
DELETE	Delete existing entity	

Specification

General

This page describes the REST API specification in NGA. The principles guiding the specification follow the basic REST guidelines that were introduced by Roy Fielding. You can read more about them here

- Variables are contained within pointy brackets: **<variable name>**
- Optional values are contained within brackets: **[optional value**]
- Repetitive optional values are contained within double brackets **[[repetitive optional value]]**

Brackets Type	Brackets Name	Functionality	Usage Example	Usage Result
<variable name>	Pointy	Variables	**<host name>** Replace the *host name* within the pointy brackets with a string representing the host name	nga.hpe.com
[optional value]	Brackets	Optional	**<host name>[:<port number>]** Replace the *host name* within the pointy brackets with a string representing the host name If a port number exists, add a colon (":") and replace the *port number* with a number representing the port number	nga.hpe.com:8080
[[repetitive optional value]]	Double Brackets	Repetitive optional	**[[/<path segment>]]** Build a path by segments, adding a slash ("/") before each path segment, replacing the *path segment* variable with path segments	/shared_spaces/1/workspaces/2/defects/10

Authentication

The Protocol

The authentication protocol is defined by the *HTTP/1.0 Basic Authentication Scheme* standard as defined in RFC2617

Summary of RFC2617 for *Basic Authentication Scheme* is brought below.

Upon calling the *REST API Authentication URI* with method *POST* the server will pose a challenge:

- A response header *WWW-Authenticate*
- With a value: *Basic realm="Please enter user credentials"*

The consumer should call the *REST API Authentication URI* with *POST* method with the following response to a challenge:

- A request header: *Authorization*
- With a value: *Basic <base64 encoded value <username>:[<password>]>*

Upon successful authentication, status code 200 (OK) is returned.

Upon failed authentication, status code 401 (Unauthorized) is returned.

Example:

Let's assume the username is *user* and the password is *pass*.
So we encode the string ***user:pass*** in base64 encoding, which is: ***dXNlcjpwYXNz***

So we call the *REST API Authentication URI* with the following header:

Authorization: Basic dXNlcjpwYXNz

NOTE:

Of course no need to call the API twice, once - just to get the challenge, and the second for the response to the challenge. The challenge is good to use via browsers, since they support Basic Authentication, and a popup will appear to enter your credentials.
Consumer can call the *REST API Authentication URI* directly with the *Authorization* header with the correct credentials value format.

URI

Sign In

The *sign_in* resource requests authentication

- **http[s]://<server>:<port>/authentication/sign_in**

Supported HTTP Methods: *POST*

Upon successful authentication the following occurs:

- Status code **200 (OK)** is returned
- A cookie with the name ***LWSSO_COOKIE_KEY*** is set as a response cookie
    - This cookie is expected to be sent in each consequent requests
    - This cookie is the authentication cookie
    - The value of this cookie can be refreshed upon specific consequent call – renewal of the cookie
- A cookie with the name ***HPSSO_COOKIE_CSRF*** is set as a response cookie
    - The value of this cookie is expected to be sent in consequent requests via the header named ***HPSSO_HEADER_CSRF***
    - This cookie is required for prevention of CSRF attacks

Upon failed authentication the following occurs:

- Status code **401 (Unauthorized)** is returned

Sign Out

The *sign_out* resource logs the user off the system

- **http[s]://<server>:<port>/authentication/sign_out**

Supported HTTP Methods: *POST*

- Status code **200 (OK)** is returned
- Expires the cookies returned by *sign_in* resource:
    - *LWSSO_COOKIE_KEY*
    - *HPSSO_COOKIE_CSRF*

REST Over HTTP

Overview

This section provides a high-level overview for the guidelines that are required in order to implement REST over HTTP. These guidelines are narrowed down and adapted to the HPE NGA product offering. Note however, that they are still general and remain standard.

URI

- NGA URIs are used to uniquely identify NGA resources
- URIs follow the STD 66/RFC 3986
- URIs have the following generic syntax: **<scheme name>:<hierarchical identifier>[?<query>]**

Scheme

- *Scheme* is mandatory
- The http/https schemes are the only one supported

Hierarchical Identifiers

- *Hierarchical identifier* is mandatory
- *Hierarchical identifier* has the following generic syntax: **//<authority><path>**
    - Begins with two slashes ("//")

Authority

- *Authority* is mandatory
- *Authority* has the following generic syntax: **<host name>[:<port number>]**
    - *Host name* is separated from port by using a colon (":")

Host Name

- *Host name* is mandatory
- *Host name* can be defined by domain name (DNS domain)
- *Host name* can be defined by an IP

Port Number

- *Port number* is optional
- *Port number* is a 16 bit integer
- *Port number* is separated from host name by a colon (":")

Path

- *Path* is mandatory
- *Path* has the following generic syntax: **[[/<Segment Name>]]**
- *Path* represents a sequence of segments
    - *Segments* are separated by a slash ("/")

Queries

- *Queries* are optional
- *Queries* have the following generic syntax: **?<parameter name>=<parameter value>[[&<parameter name>=<parameter value>]]**
    - Separated from hierarchical part by a question mark ("?")
- *Queries* are organized in parameter name/parameter value pairs
    - Pairs are separated by ampersands ("&")

Parameters

- *Parameters* are a pair containing *Parameter name* and *Parameter value*
    - *Parameter name* and *parameter value* are separated by equals sign ("=")
- *Parameter names* are case sensitive
- *Parameter values* are mostly case insensitive

Input

There are three ways to provide inputs to the server:

- Request header
- Query parameters
- Request body

Request Header

All supported input header fields follow the HTTP/1.1 protocol. You can find a list describing HTTP/1.1 header definitions here. Below you can find a list of NGA supported request header fields:

Header Name	Mandatory	Functionality	Example
Accept	No	• Specifies the media types that the client can accept in the response • An empty value or a missing header mean that the client can accept all media types • If the media type cannot be provided by the server then the server should return the 406 (Not Acceptable) HTTP status code	Accept: application/json,text/html,application/xhtml+xml, application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding	No	• Restricts the content coding that the client can accept in the response • An empty value or a missing header mean that the client can accept all content coding • If the coding type cannot be provided by the server then the server should return the 406 (Not Acceptable) HTTP status code	Accept-Encoding: gzip,deflate,sdch
Content-Type	No	• Specifies the media type that is being used for the request body • An empty value or an unknown value will be responded with a 415 (Unsupported Media Type) HTTP status code • If the header is missing the request will be parsed with the web server's default setting	Content-Type: application/json
Host	Yes	• Represents the authority of the server or gateway the resource is requested from • An empty value, an unrecognized value, or a missing header must be responded with the 400 (Bad Request) HTTP status code	Host: nga.hpe.com:8080

Output

Output is returned from the server in three ways:

- Status code
- Response header
- Response body

Status Codes

All supported status codes follow the HTTP/1.1 protocol. You can find a list describing HTTP/1.1 status codes here. Below you can find a list of NGA supported status codes:

Code	Name	Functionality
200	OK	• The request has succeeded
201	Created	• The request was fulfilled • A new resource was created
400	Bad Request	• The request could not be understood due to malformed syntax • The client is expected to perform changes in the request before re-sending
401	Unauthorized	• The request requires user authentication
403	Forbidden	• The server understood the request • The server refuses to fulfill it
404	Not Found	• The server has not found a resource matching the request URI
405	Method Not Allowed	• The request method is not allowed for the resource matching the request URI
406	Not Acceptable	• The resource cannot be generated according to the *Accept* field header provided in the request
408	Request Timeout	• HTTP protocol timeout occurred while processing the request. Handled by the servlet container.
409	Conflict	• The request was partially fulfilled or completely failed because of a conflict in the request
415	Unsupported Media Type	• The server understood the request • The request format is not supported for the request method by the resource matching the request URI
500	Internal Server Error	• The server has encountered a condition which prevented it from fulfilling the request
501	Not Implemented	• The server does not recognize the request method • The server is not capable of supporting it for any resource

Response Header

All supported output header fields follow the HTTP/1.1 protocol. You can find a list describing HTTP/1.1 header definitions here. Below you can find a list of NGA supported response header fields:

Header Name	Mandatory	Functionality	Example	Exceptions
Content-Encoding	Y	• Specifies which coding was applied on the response body • Return coding must be a sub-list of the coding specified in the *Accept-Encoding* request header	Content-Encoding: gzip	
Content-Type	Y	• Specifies the media type that is being used for the response body • Return media types must be a sub-list of the media types specified in the *Accept* request header	Content-Type: application/json;q=0.9	
Date	Y	• Specifies the date and time at which the message was originated • Date and time format follow the guidelines provided in RFC 1123	Date: Mon, 24 Mar 2014 12:11:05 GMT	• Date response header might be omitted by the server when the following status code return: 100, 101, 500, 503
Server	Y	• Specifies the software used by the server in order to handle the request • Forward proxies must not change this response header and must use the Via response header instead	Server: Jetty	
Via	N	• Specifies the proxy or gateway software that forwarded the request to the server	Via: server.com (Apache/2.1)	
WWW-Authenticate	N	• Specifies the authentication challenges required from the HTTP user-agent • Multiple challenges are separated by comas, or by multiple identical headers	WWW-Authenticate: LWSSO realm="http://nga.com/authentication"	• Mandatory when status code 401 returns

Cookies

Cookie Name	Mandatory	Functionality
LWSSO_COOKIE_KEY	N	This is the authentication token. For more info, see Authentication
HPSSO_COOKIE_CSRF	N	This cookie related to the protocol for preventing CSRF attacks. For more info, see Authentication

Request Methods

All supported request methods follow the HTTP/1.1 protocol. You can find a list describing HTTP/1.1 request methods here. The following request methods should be supported at first:

- GET
    - Requests a representation of a resource matching the request URI
- PUT
    - Requests that the enclosed represented resource will be stored under the matching request URI if the resource already exists
- DELETE
    - Requests a deletion of the resource found in the matching request URI
- POST
    - Requests a creation of a new resource under the resource collection matching the request URI

Representations

By default, only JSON representation is supported for all resource instances and resource collections.

JSONs

JSON representations follow the ECMA-404 standard called "The JSON Data Interchange Format".

NGA Over REST

Overview

This section provides an overview for the structure of REST API in NGA. It is based on the previous section called REST over HTTP.

Resources

- REST Resources are identified by URIs
- A resource can be either of two types:
  - Resource Instance (e.g. defect)
  - Resource Collection (e.g. defects)
- Resources should be built in an hierarchy of collections and instances

Hierarchy of Resources by Example

We start with a resource collection *of humans.* This collection contains resource instances of *human*. Since humans have limbs then every *human* will contain a collection of *limbs*. This collection contains resource instances of *limb*. Since limbs have fingers then every *limb* will contain a collection of *fingers*. This collection contains resource instances of *finger*.

This is how the modeling looks like:

- humans
  - human
    - limbs
      - limb
        - fingers
          - finger
          - finger
      - limb
        - fingers
          - finger
          - finger
  - human
    - limbs
      - ...

- A strict set of request methods are allowed for a resource type
- Resources support the following representation types:
  - JSON

Naming Convention

- Since resources represent entities, their naming should represent the entities' names (e.g. *defect*)
- Resource instances should be called by their singular names (e.g. *test*)
- Resource collections should be called by their plural names (e.g. *tests*)

- Resource which name is more than one word will be separated by underscore symbol between the words. E.g. *shared_spaces*

Resource Collection

- A resource collection is a set of *resource instances*
- Resource collections may contain 0->N resource instances
- *Entity Collection* is *resource collection* of *entities* (see below *resource instance*)
- A resource collection is identified using the following general syntax:
    - **<scheme name>:<authority>/api/<resource collection>**
    - **<scheme name>:<authority>/admin/<resource collection>**

Resource Collection URI Example - Using the Defects Collection
http[s]://nga.com/api/shared_spaces/abcdef/workspaces/1003/defects

Supported Request Methods

Resource collections can support the following request methods:

- GET
- PUT
- DELETE
- POST

GET

- Read the entities
- Option to filter
- Option to define paging
- Option to sort
- Option for field selection

GET Request Example for Defects without filtering with default paging and sorting

```
*** Request ***
GET
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/
Accept:             application/json
Host:               nga.com:8080
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx
```

```
{
    "total_count": 2,
    "data": [
        {
            "type": "defect",
            "creation_time": "2014-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "logical_name": "1nq47mymd4o28sxkyeker7l86",
            "version_stamp": 1,
            "release": {
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
1\n</body></html>",
            "id": 1003,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "phase": {
                "type": "phase",
                "id": 1015
            },
            "priority": null,
            "name": "def1",
        },
        {
            "type": "defect",
            "creation_time": "2015-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "logical_name": "4nq47mysfdsfdsxkyeker7l86",
            "version_stamp": 5,
            "release": {
                "type": "release",
                "id": 1040
            },
            "description": "<html><body>\ndefect
2\n</body></html>",
            "id": 1007,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
```

```
                "type": "list_node",
                "id": 1074
            },
            "phase": {
                "type": "phase",
                "id": 1017
            },
            "priority": {
                “type”: “list_node”,
                “id”: 1038
            },
            "name": "def2",
        }
    ]
}
```

PUT

- Request
  - *data* array of entity objects to update by *id*
- Response
  - *data* array of entity objects for which the update was successful

PUT Request Example for Defects

```
*** Request ***
PUT
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/
Accept:             application/json
Content-Type:       application/json
Host:               nga.com:8080

{
    "data": [
        {
            “id”: 1003,
            "type": "defect",
            "release": {
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "name": "def3"
        },
```

```
        {
            “id”: 1004,
            "type": "defect",
            "release": {
                "type": "release",
                "id": 1040
            },
            "description": "<html><body>\ndefect
4\n</body></html>",
            "severity": {
                "type": "list_node",
                "id": 1074
            },
            "name": "def4"
        }
    ]
}
```


```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx
```


```
{
    "total_count": 2,
    "data": [
        {
            "type": "defect",
            "creation_time": "2014-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "logical_name": "1nq47mymd4o28sxkyeker7l86",
            "version_stamp": 2,
            "release": {
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "id": 1003,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "phase": {
```

```
                "type": "phase",
                "id": 1015
            },
            "priority": null,
            "name": "def3"
        },
        {
            "type": "defect",
            "creation_time": "2015-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "logical_name": "4nq47mysfdsfdsxkyeker7l86",
            "version_stamp": 6,
            "release": {
                "type": "release",
                "id": 1040
            },
            "description": "<html><body>\ndefect
4\n</body></html>",
            "id": 1007,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
                "type": "list_node",
                "id": 1074
            },
            "phase": {
                "type": "phase",
                "id": 1017
            },
            "priority": {
                “type”: “list_node”,
                “id”: 1038
            },
            "name": "def4"
        }
    ]
}
```

PUT – Partial Success

- Request
  - *data* array of entity objects to update by *id*
- Response
  - *data* array of entity objects for which the update was successful
  - *errors* array of error objects for entities for which the update was not successful

PUT – Partial Success - Request Example for Defects, where trying to update defect by id 1004 referencing a non-existing release by id 1040

```
*** Request ***
PUT
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/
Accept:             application/json
Content-Type:       application/json
Host:               nga.com:8080

{
    "data": [
        {
            “id”: 1003,
            "type": "defect",
            "release": {
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "name": "def3"
        },
        {
            “id”: 1004,
            "type": "defect",
            "release": {
                "type": "release",
                "id": 1040
            },
            "description": "<html><body>\ndefect
4\n</body></html>",
            "severity": {
                "type": "list_node",
                "id": 1074
```

```
            },
            "name": "def4"
        }
    ]
}
```

```
*** Response ***
HTTP/1.1 409 Conflict
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

{
    "total_count": 1,
    "data": [
        {
            "type": "defect",
            "creation_time": "2014-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "logical_name": "1nq47mymd4o28sxkyeker7l86",
            "version_stamp": 2,
            "release": {
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "id": 1003,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "phase": {
                "type": "phase",
                "id": 1015
            },
            "priority": null,
            "name": "def3",
        }
    ],
    “errors”: [
        {
            “error_code”: “platform.entity_not_found”,
            “description”: “The entity by id 1040 of type release
does not exist”,
```

```
            “properties”: {
                “entity_type”: “release”,
                “entity_id”: “1040”
            }
        }
    ]
}
```

PUT – By Filter

- Request
  - *query* parameter, which is the filter which entities to update
  - *data* array of one entity object to update the entities specified by the *query* parameter
- Response
  - *data* array of entity objects for which the update was successful
  - *errors* array of error objects for entities for which the update was not successful

DELETE

- A Request for a delete on a collection deletes the entire collection
- Deleting can be done by using filtering
- Supports partial success, similar to POST and PUT

DELETE all defects

```
*** Request ***
DELETE
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/
Accept:             application/json
Host:               nga.com:8080
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx
```

POST

- Request
  - *data* array of entity objects to create
- Response
  - *data* array of entity objects created successfully – each object contains an *id*

PUT Request Example for Defects

```
*** Request ***
PUT
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/
Accept:             application/json
Content-Type:       application/json
Host:               nga.com:8080

{
    "data": [
        {
            "type": "defect",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "release": {
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "name": "def3"
        },
        {
            "type": "defect",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "release": {
                "type": "release",
                "id": 1040
            },
            "description": "<html><body>\ndefect
4\n</body></html>",
            "severity": {
                "type": "list_node",
```

```
                "id": 1074
            },
            "name": "def4"
        }
    ]
}
```


```
*** Response ***
HTTP/1.1 201 Created
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx
```


```
{
    "total_count": 2,
    "data": [
        {
            "type": "defect",
            "creation_time": "2016-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "logical_name": "1nq47mymd4o28sxkyeker7l86",
            "version_stamp": 1,
            "release": {
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "id": 1003,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "phase": {
                "type": "phase",
                "id": 1015
            },
            "priority": null,
            "name": "def3"
        },
        {
            "type": "defect",
            "creation_time": "2016-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
```

```
                "id": 1002
            },
            "logical_name": "4nq47mysfdsfdsxkyeker7l86",
            "version_stamp": 1,
            "release": {
                "type": "release",
                "id": 1040
            },
            "description": "<html><body>\ndefect
4\n</body></html>",
            "id": 1007,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
                "type": "list_node",
                "id": 1074
            },
            "phase": {
                "type": "phase",
                "id": 1015
            },
            "priority": null,
            "name": "def4"
        }
    ]
}
```


POST – Partial Success

- Request
  - *data* array of entity objects to create
- Response
  - *data* array of entity objects created successfully – each object contains an *id*
  - *errors* array of error objects for entities for which the creation was not successful

POST – Partial Success - Request Example for Defects, where trying to create defect by id 1004 referencing a non-existing release by id 1040

```
*** Request ***
POST
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/
Accept:             application/json
Content-Type:       application/json
Host:               nga.com:8080

{
    "data": [
        {
            "type": "defect",
            "release": {
```

```
                "type": "release",
                "id": 1002
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "name": "def3"
        },
        {
            "type": "defect",
            "release": {
                "type": "release",
                "id": 1040
            },
            "description": "<html><body>\ndefect
4\n</body></html>",
            "severity": {
                "type": "list_node",
                "id": 1074
            },
            "name": "def4"
        }
    ]
}
```


```
*** Response ***
HTTP/1.1 409 Conflict
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx
```


```
{
    "total_count": 1,
    "data": [
        {
            "type": "defect",
            "creation_time": "2016-01-27T13:01:52Z",
            "parent": {
                "type": "work_item_root",
                "id": 1002
            },
            "logical_name": "1nq47mymd4o28sxkyeker7l86",
            "version_stamp": 1,
            "release": {
                "type": "release",
                "id": 1002
```

```
            },
            "description": "<html><body>\ndefect
3\n</body></html>",
            "id": 1003,
            "last_modified": "2016-01-27T13:01:52Z",
            "severity": {
                "type": "list_node",
                "id": 1073
            },
            "phase": {
                "type": "phase",
                "id": 1015
            },
            "priority": null,
            "name": "def3",
        }
    ],
    “errors”: [
        {
            “error_code”: “platform.entity_not_found”,
            “description”: “The entity by id 1040 of type release
does not exist”,
            “properties”: {
                “entity_type”: “release”,
                “entity_id”: “1040”
            }
        }
    ]
}
```

Filtering

Filtering entities is achieved by filtering values of fields. Therefore, the language of the values provided in the filter should kept in consistency with the data types declared by the fields.

Allowed Operations per Data Type

Data Type	Equals To	Less than	Greater than	Less than or equals to	Greater than or equals to
Integer	👍	👍	👍	👍	👍
Boolean	👍	👎	👎	👎	👎
Date / DateTime	👍	👍	👍	👍	👍
String	👍	👎	👎	👎	👎
Memo	👎	👎	👎	👎	👎
Reference	👍	👎	👎	👎	👎

Query Statement

- The *query* statement is contained within double quotes: *"query statement"*
- Query statement has the following generic syntax:
    - **<query phrase>[[<logical operator><query phrase>]]**

Query Phrase

Query phrase has the following generic syntax:

[opening parenthesis][negate keyword][opening parenthesis]<field name><comparison operator><value>[closing parenthesis][closing parenthesis]

Opening and Closing Parenthesis

- Query phrase can be enclosed in parenthesis
- Parenthesis is an operator which is the highest precedence between the operators
- The meaning of this operator is the order in which the conditions are evaluated
- Opening parenthesis has the following generic syntax: **(**
- Closing parenthesis has the following generic syntax: **)**

Negate Keyword

- Negate keyword is optional
- Has the following generic syntax: **!**

Comparison Operator

- Comparison operators are used to separate between field names and their values
- The table below lists the available comparison operators:

Operator Symbol	Functionality	Example
EQ	Equals to	id EQ 1001
LT	Less than	id LT 1001
GT	Greater than	id GT 1001
LE	Less than or equals to	id LE 1001
GE	Greater than or equals to	id GE 1001

Value

- Values are either numerical, boolean, string based (*Date* and *DateTime* type values in filter notation behave like *strings*) or reference

- "No Value" notion (all types except of *boolean* and *string* values. For *string* see further below):
  - Whenever a value does not exist (e.g. *defect closing date* was not defined yet, since the defect is not closed yet), special keyword ***null*** should be defined for such a case. This ***null*** keyword can be used also in filtering in order to specify the notion of "no value".
- Boolean values should be put the value ***true*** or ***false***
  - Example: **/<some_entities>?query="<some_boolean_field_name> EQ true"**
- Numeric values are placed after the comparison operator
  - Example: **/<some_entities>?query="<some_numeric_field_name> GE 35"**
- Date / DateTime values:
  - Must be wrapped in carets: ***^date^***
    - Currently, there is a known defect, and wrapping in carets does not work. Please use quotes wrapping instead – ***'date'***. This will be fixed in next release, and carets will be used.
  - Expected date and time format is ISO-8601. Examples:
    - 2015-02-25T16:42:11Z
    - 2015-02-25T16:42:11+02:00
  - The date and time is UTC

- For filtering purposes, should be in UTC and ISO-8601 format
- Example: **/<some_entities>?query="<some_date_field_name> LT ^2015-02-25T16:42:11Z^**

- String / Memo values:
  - Must be wrapped in carets: ***^string^***

    NOTE:
    Currently, there is a known defect, and wrapping in carets does not work. Please use quotes wrapping instead – ***'string'***. This will be fixed in next release, and carets will be used.

  - Escaping of special character in *string* value (if the string you are searching contains one of the following characters, and you would like to filter by the character):

Character	Escaped Character (URI Encoded)	Comments
"	\" (%5C%22)	
^	\^ (%5C%5E)	
\	\\ (%5C%5C)	
'	\q (%5Cq)	
<	\l (%5Cl)	
>	\g (%5Cg)	
*	N/A	Filtering by this character is not supported
{	\{ (%5C%7B)	
(	\((%5C( )	
)	\) (%5C))	
[	\[(%5Cb)	
?	\? (%5C%3F)	

- No value / empty strings - represented as ***null***
    - This implies that whenever user nullifies a string field from existing value (via PUT), client can send *null* or *empty string* ("") and server will store in DB ***null*** – this implies that empty string is a non-valid value for a non-nullable field.
- REST API doesn't trims string/memo fields
    - The only manipulation of a string / memo field values from the server side can occur only due to output sanitization functionality
- Wildcard: '****'*** – star character
    - Means – any match
    - Examples:
        - String ends with 'ending': ***ending**
            - Matches: lala_ending ; lalaending; ending
        - String starts with 'starting': **starting***
            - Matches: starting_lala ; startinglala, starting
        - String which starts with 'starting' and ends with 'ending': **starting*ending**
            - Matches: starting_ending ; startingending ; startinglalalaending
- Examples:
    - **/<some_entities>?query="<some_string_field_name> EQ ^existence^"**
    - **/<some_entities>?query="<some_string_field_name> EQ ^test*^"**

- Reference values:
    - Filtering on reference value (see field metadata API for explanation about reference fields) means ability to filter on field values of the referenced entity
    - Reference value has the following generic form:
        - **{<query phrase>[[<logical operator><query phrase>]]}**

Example: *defect* entity has a reference field to *release* entity which is called *detected_in_release*. We want to filter all *defects* which were detected in *release* by name *'release1'*

```
/defects?query="detected_in_release EQ {name EQ ^release1^}"
```

- Reference field can reference a single entity or many entities - multi-reference field. In case of multi-reference field, the *equality* operator will work as a *containment* operator.
    - Example: Consider the following scenario – defect which is tagged with multiple user tags

```
{
    “type”: “defect”,
    “user_tags”: [
        {
            “id”: 1001,
            “type”: “user_tag”
        },
        {
            “id”: 2005,
            “type”: “user_tag”
        },
        {
            “id”: 3008,
            “type”: “user_tag”
        }
    ]
}
```

The following filters will all retrieve the above defect:
/defects?query="user_tags EQ {id EQ 1001}"
/defects?query="user_tags EQ {id EQ 1001||id EQ 2005}" /defects?query="user_tags EQ {id EQ 1001}||user_tags EQ {id EQ 2005}"
/defects?query="user_tags EQ {id EQ 1001||id EQ 500000}" /defects?query="user_tags EQ {id EQ 1001}||user_tags EQ {id EQ 500000}"
/defects?query="user_tags EQ {id EQ 1001;id EQ 3008}" /defects?query="user_tags EQ {id EQ 1001};user_tags EQ {id EQ 3008}"
/defects?query="user_tags EQ {(id EQ 1001;id EQ 2005;id EQ 3008)||id EQ 50000000}" /defects?query="user_tags EQ {id EQ 1001;id EQ 2005;id EQ 3008}||user_tags EQ {id EQ 50000000}"
/defects?query="user_tags EQ {id EQ 1001||(id EQ 2005;id EQ 50000000)}" /defects?query="user_tags EQ {id EQ 1001}||(user_tags EQ {id EQ 2005}; user_tags EQ {id EQ 50000000})"
/defects?query="user_tags EQ {id EQ 1001}||user_tags EQ {null}"

The following filters will <u>not</u> retrieve the above defect:
`/defects?query="user_tags EQ {id EQ 1001;id EQ 50000000}"` `/defects?query="user_tags EQ {id EQ 1001};user_tags EQ {id EQ 50000000}"`

`/defects?query="user_tags EQ {id EQ 1001;(id EQ 5000000||id EQ 7000000)}"`
`/defects?query="user_tags EQ {id EQ 1001};user_tags EQ {id EQ 5000000||id EQ 7000000}"`
`/defects?query="user_tags EQ {id EQ 1001};(user_tags EQ {id EQ 5000000}|| user_tags EQ {id EQ 7000000})"`

`/defects?query="user_tags EQ {id EQ 1001};user_tags EQ {null}"`

Logical Operator

- Logical operators are used to separate between query phrases or query statements
- The table below lists the available logical operators:

Logical Operator	Functionality
;	And
||	Or

Operator Precedence

Operator	Rank (low number = higher rank)
()	1
!	2
;	3
||	4

Example: The following statements are equivalent

```
"((!name EQ ^test^));(flag EQ true)"
"(!name EQ ^test^);flag EQ true"
"!name EQ ^test^;flag EQ true"
```

Pagination

- Allows retrieving a limited collection of results from the server
- Allows offsetting a collection of results from the server
- Keywords are *limit* and *offset*

Limit

- Limits the number of results in a resource collection returned from the server to a specific number
- *total_count* property will state the real number of entities which answer the filter (besides the ones which are brought by the specific page)
- If no *limit* parameter is provided a default is being used
- Limit has the following generic syntax: **limit=<limit_value>**
    - limit_value is an integer
    - limit_value > 0

Offset

- Offsets the starting point of the collection returned from the server in the results
- If no offset parameter is provided - the number 0 (zero) is assumed
- Offset has the following generic syntax: **offset=<offset_value>**
    - offset_value is an integer
    - offset_value >= 0

Example: Consider that we have 100 defects in the system. To get 10 defects in places 40 – 49 the following construct should be sent:

```
/defects?limit=10&offset=40
```

Fields

- Allows limiting the set of fields returned from the server
- The fields *id* and *type* are brought always whatever the fields selection
- Has the following generic syntax: **fields=<field name>[[,<field name>]]**
    - Field names are separated by commas: ","
- If no *fields* query parameter is provided a default set of fields is being used

Example:

```
/defects?fields=id,name
```

Sorting

- Allows sorting the result entities returned from the server
- Has the following generic syntax:
  order_by=[<direction>]<field name>[[,[<direction>]<field name>]]
    - Field names are separated by commas: “,”
- If no *order_by* query parameter is defined, the sorting on *id* field is assumed

Direction

- Determines the ordering direction
- Allowed values are described in the following table:

Direction	Functionality
(empty)	Ascending (default)
- (minus)	Descending

Example:

```
/defects?order_by=severity,-creation_time
```

Resource Instance

- A resource instance represents an object on the server side
- A resource instance is identified by its unique ID in relevant context
- An *entity* is *resource instance*
- A resource instance is identified using the following generic syntax:
    - **<context>/<resource collection>/<resource instance id>**

Resource Collection URI Example - Using the Defects Resource
http[s]://nga.com/api/shared_spaces/abcdef/workspaces/1003/defects/2005

Supported Request Methods

Resource instance can support the following request methods:

- GET
- PUT
- DELETE

GET

- Read the entity
- Option for field selection

GET Request Example for Defect Resource Instance

```
*** Request ***
GET
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/1003
Accept:             application/json
Host:               nga.com:8080
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

{
    "type": "defect",
    "creation_time": "2014-01-27T13:01:52Z",
    "parent": {
        "type": "work_item_root",
        "id": 1002
    },
    "logical_name": "1nq47mymd4o28sxkyeker7l86",
    "version_stamp": 1,
    "release": {
        "type": "release",
        "id": 1002
    },
    "description": "<html><body>\ndefect 1\n</body></html>",
    "id": 1003,
    "last_modified": "2016-01-27T13:01:52Z",
    "severity": {
        "type": "list_node",
        "id": 1073
    },
    "phase": {
        "type": "phase",
        "id": 1015
    },
    "priority": null,
    "name": "def1",
}
```

PUT

- Request
  - An *entity* object to update by *id*
- Response
  - An *entity* object if the update was successful
  - An *error* object if error occurred

PUT Request Example for Defect Resource Instance

```
*** Request ***
PUT
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/1003
Accept:             application/json
Content-Type:       application/json
Host:               nga.com:8080

{
    “id”: 1003,
    "type": "defect",
    "release": {
        "type": "release",
        "id": 1002
    },
    "description": "<html><body>\ndefect 3\n</body></html>",
    "severity": {
        "type": "list_node",
        "id": 1073
    },
    "name": "def3"
}
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

{
    "type": "defect",
    "creation_time": "2014-01-27T13:01:52Z",
    "parent": {
        "type": "work_item_root",
        "id": 1002
    },
    "logical_name": "1nq47mymd4o28sxkyeker7l86",
    "version_stamp": 2,
    "release": {
        "type": "release",
        "id": 1002
```

```
    },
    "description": "<html><body>\ndefect 3\n</body></html>",
    "id": 1003,
    "last_modified": "2016-01-27T13:01:52Z",
    "severity": {
        "type": "list_node",
        "id": 1073
    },
    "phase": {
        "type": "phase",
        "id": 1015
    },
    "priority": null,
    "name": "def3"
}
```


DELETE

- Deletes the resource instance

DELETE all defects
*** Request *** DELETE /api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/defects/2005 Accept: application/json Host: nga.com:8080
*** Response *** HTTP/1.1 200 OK Content-Encoding: gzip Content-Type: application/json;q=0.9 Date: Mon, 27 Mar 2014 12:11:05 GMT Server: nginx

Entity Metadata

Entity metadata is all about the features the entity supports. In this part of the document we will explore all the features of an entity, and how they represented in the REST API.

URI

- Workspace level: **http[s]://<server>:<port>/api/shared_spaces/<id>/workspaces/<id>/metadata/entities**
- Shared space level: **http[s]://<server>:<port>/api/shared_spaces/<id>/metadata/entities**
- Site level (N/A in SaaS): **http[s]://<server>:<port>/admin/metadata/entities**

Supports filtering by *name*

Supported HTTP Methods

This API supports only GET operation

Properties

Name

- The entity name for which the metadata is defined
- Field name: ***name***

Features

- Array of features supported by an entity
  - Each item is a JSON object
- Field name: ***features***
- Below are the available features

REST

- REST API related definitions
- Fields:
  - ***name:*** the name of the feature – constant value: *rest*
  - ***url:*** the resource URL relative to context root
  - ***methods:*** an array of strings, the HTTP protocol methods supported. Values: *GET*, *POST*, *PUT* or *DELETE*

Example for *test* entity:

```
{
    "name": "rest",
    "url": "tests",
    "methods": [ "GET", "POST", "PUT", "DELETE" ]
}
```

Mailing

- Defines whether the entity supports mailing
- Fields:
  - ***name:*** the name of the feature – constant value: *mailing*

This is how the *mailing* feature looks like:

```
{
    “name”: “mailing”
}
```

Attachments

- Defines whether the entity supports *attachments*
  - By accessing the *<context>/attachments* API
- Fields:
  - ***name:*** the name of the feature – constant value: *attachments* (note: currently it is called *has_attachments*)

This is how the *attachments* feature looks like:

```
{
    “name”: “attachments”
}
```

Comments

- Defines whether the entity supports *comments*
  - By accessing the *<context>/comments* API
- Fields:
  - ***name:*** the name of the feature – constant value: *comments* (note: currently it is called *has_comments*)

This is how the *comments* feature looks like:

```
{
    “name”: “comments”
}
```

Business Rules

- Defines whether the entity supports *business rules*
- Fields
  - ***name:*** the name of the feature – constant value: *business_rules*

This is how the *business rules* feature looks like:

```
{
    “name”: “business_rules”
}
```

Subtypes

- Defines that the entity has *subtypes*
- Fields:
  - ***name:*** the name of the feature – constant value: *subtypes*
  - ***types:*** A string array of the names of the entities, which are the subtypes of this entity
- The entity will have a field called *subtype* which will store the *subtype* of the entity instance

Example:

```
{
    “name”: “subtypes”,
    “types”: [
        “defect”,
        “feature”,
        “theme”,
        “work_item_root”,
        “story”
    ]
}
```

Subtype Of

- Defines, that an entity is a subtype of another entity
- Fields:
  - ***name:*** the name of the feature – constant value: *subtype_of*
  - ***type:*** the name of the entity, which has this entity in the *subtypes* feature definition

Example:

```
{
    “name”: “subtype_of”,
    “type”: “work_item”
}
```

Hierarchy

- Defines whether the entity is hierarchical
- Fields:
  - ***name:*** the name of the feature – constant value: *hierarchy* (note: currently it is called *hierarchical_entity*)
  - ***root***: the root entity in the hierarchy. An object with the following fields:
    - ***type:*** the type of the root entity
    - ***id:*** the id of the root entity
  - ***child_types***: an array of strings, allowed types for child entities
    - for non-empty array – *multi-reference* field by name *children* exists on the entity (this field will be defined in the *fields metadata API*)
  - ***parent_types***: an array of string, allowed types for parent entities
    - for non-empty array – *reference* field by name *parent* exists on the entity (this field will be defined in the *fields metadata API*)

Example 1: this is how the *hierarchy* feature will look for the '*feature*' entity (subtype of '*work_item*')

```
{
   "name": "hierarchy",
   "root": {
      "type": "work_item_root",
      "id": 1001
   },
   "parent_types": ["theme"],
   "child_types": ["defect", "story"]
}
```

Example 2: this is how the *hierarchy* feature will look for the ' *'application_module'* entity

```
{
   "name": "hierarchy",
   "root": {
      "type": "application_module",
      "id": 1001
   },
   "parent_types": ["application_module"],
   "child_types": ["application_module"]
}
```

Example: *defect* entity metadata

```
{
    “name”: “defect”,
    “features”: [
        {
            “name”: “rest”,
            “methods”: [
                “DELETE”,
                “POST”,
                “GET”,
                “PUT”
            ],
            “url”: “defects”
        },
        {
            “name”: “comments”
        },
        {
            “name”: “mailing”
        },
        {
            “name”: “subtype_of”,
            “type”: “work_item”
        },
        {
            “name”: “attachments”
        },
        {
            “name”: “business_rules”
        },
        {
            “name”: “hierarchy”,
            “parent_types”: [
                “feature”,
                “work_item_root”
            ],
            “child_types”: [],
            “root”: {
                “type”: “work_item_root”,
                “id”: 1002
            }
        }
    ]
}
```

Field Metadata

Field metadata API define the fields available for an entity. For each field there are properties and features, e.g. type of the field, whether the field's values are audited, validations on the field, etc.

URI

- Workspace level: **http[s]://<server>:<port>/api/shared_spaces/<id>/workspaces/<id>/metadata/fields**
- Shared space level: **http[s]://<server>:<port>/api/shared_spaces/<id>/metadata/fields**
- Site level (N/A in SaaS): **http[s]://<server>:<port>/admin/metadata/fields**

Supports filtering by *entity_name*

Supported HTTP Methods

This API supports only GET operation

Properties

Here we will enumerate the various properties a field can have in order to describe itself.

Name

- The name of the field as it appears in API representation
- Field name: ***name***

Label

- The label of the field as might appear in UI
- Field name: ***label***

Entity

- The entity the field belongs to
- Field name: ***entity_name***

Filterable

- Marks a field as one which can be filtered by or not
- Field name: ***filterable***
- Value: ***Boolean***

Sortable

- Marks a field as one which can be sorted by or not
- Field name: ***sortable***
- Value: ***Boolean***

Returned by Default

- Marks a field as one which is brought by default or not when no field selection is
- Field name: ***returned_by_default***

Field Type

- The type of the field's expected value
- Field name: ***field_type***
- Below we will list the available field types

Integer

- Field which contains integer values
- Value of this type can be in the range of -2147483648 to +2147483647
  - Deprecated. See *Long* type.

Here is how the *Integer* type looks like:

```
"field_type": "integer"
```

Long

- Field which contains integer values
- Value of this type can be in the range of -9223372036854775808 to +9223372036854775807
  - Deprecated. Will be renamed to *Integer* type.

Here is how the *Long* type looks like:

```
"field_type": "long"
```

Boolean

- Field which contains Boolean values
- Value: *true* / *false*

Here is how the *Boolean* type looks like:

```
"field_type": "boolean"
```

DateTime

- Field which contains date and time information
- Expected date format is: ISO-8601
  - Examples: 2015-02-25T16:42:11+00:00 , 2015-02-25T16:42:11Z
- The date and time is UTC
- For create, update and filtering purposes, API consumer must use the UTC and ISO-8601 formats

Here is how the *DateTime* type looks like:

```
"field_type": "date_time"
```

Date

- Field which contains date information
- Expected date format is: ISO-8601
  - Examples: 2015-02-25T16:42:11+00:00 , 2015-02-25T16:42:11Z

NOTE:
Although the field is defined as *Date* we still keep the *time* portion of the date for multi-zonal filtering capabilities on *Date* fields. This means that with regards to the data representation, the *Date* and *DateTime* types are the same. The *Date* field type exists in order to sign the consumer, that the application uses only the *Date* portion of the data. (UI will show accordingly the *Date* picker control and not *DateTime* picker control)

- The date and time is UTC

- For create, update and filtering purposes, API consumer must use the UTC and ISO-8601 formats

Here is how the *Date* type looks like:

```
"field_type": "date"
```

String

- Field which contains a string value
- The length of the value is determined by the *max_length* input validation property

Here is how the *String* type looks like:

```
"field_type": "string"
```

Memo

- Field which contains a text value
  - Usually represented in HTML

Here is how the *Memo* type looks like:

```
"field_type": "memo"
```

Object

- Field which has an arbitrary structure
  - Represented as JSON object
  - The format of the JSON object is defined in documentation

Here is how the *Object* type looks like:

```
"field_type": "object"
```

NOTE: currently the *object* type is called *custom*

Reference

- Field which references an *entity collection*
- Value sent in *Reference* type is sent as *link*
  - *Link* is an object
  - *Link* contains the following information
    - *Entity type*
    - *Entity id*

Example of *link* object:

```
{
    "id": 1001,
    "type": "defect"
}
```

- The field ***field_type_data*** will contain information on the referenced types
- Multiple references can be represented if the ***multiple*** field is set to *true*

When the ***multiple*** field is set to *true*, values are expected in the following form (example):

```
{
    “total_count”: 2,
    “data”: [
        {
            “id”: 1001,
            “type”: “release”
        },
        {
            “id”: 1002,
            “type”: “release”
        }
    ]
}
```

For *POST* and *PUT* requests, the *total_count* property can be omitted.

- *List Node* entity is a special one, and reference field to *list node* will look different than all the other entities
- *Reference* field can reference more than one entity type
- Here is how the *Reference* type looks like:

Reference to single entity type except *list_node* entity:

```
"field_type": "reference",
"field_type_data": {
    "targets": [
        {
            "type": "release"
        }
    ],
    "multiple": false
}
```

NOTE: in current version it looks like:

```
"field_type": "reference",
"field_type_data": {
    "target": {
        "type": "release",
        “types”: [ “release” ]
    },
    "multiple": false
}
```

Reference to *list_node* entity:

```
"field_type": "reference",
"field_type_data": {
    "targets": [
        {
            "type": "list_node",
            "logical_name": "list_node.severity"
        }
    ],
    "multiple": false
}
```

NOTE: in current version it looks like:

```
"field_type": "reference",
"field_type_data": {
    "target": {
        "logical_name": "list_node.severity"
        “types”: [ "list_node" ],
        “type”: “list_node”
    },
    "multiple": false
}
```

Reference to multiple entity types except *list_node* entity:

```
"field_type": "reference",
"field_type_data": {
    "targets": [
        {
            "type": "defect"
        },
        {
            "type": "story"
        }
    ],
    "multiple": false
}
```

NOTE: in current version it looks like:

```
"field_type": "reference",
"field_type_data": {
    "target": {
        “types”: [ “defect”, “story” ],
        “type”: “work_item”
    ],
    "multiple": false
}
```

Data Validations

- *Data validations* allow performing validations on inputs and outputs
    - They provide a mean for the consumer to validate inputs before they are being sent to the server, or expect certain outputs returned from the server
- *Data validations* are represented as fields on the *field metadata* object

Input

- Values that are being sent to the server can be validated
    - Some validations are optional
- Some validations are allowed for specific data types only

	Integer / Long	Boolean	Date / DateTime	String	Memo	Reference
Not Null	✓	✓	✓	✓	✓	✓
Maximum Length				**Mandatory**		
Unique	✓			✓		
Read Only	✓	✓	✓	✓	✓	✓

Not Null

- Determines whether a ***null*** value is allowed
- Field name: ***required***
- Value: boolean

Example:

```
"required": true
```

Maximum Length

- Enforces a maximum length of a *string* field
- Field name: ***max_length***
- Value: integer > 0

Example:

```
"max_length": 70
```

Unique

- Enforces uniqueness on the level of the whole *entity* type
- Field name: ***unique***
- Value: boolean

Example:

```
"unique": true
```

Read Only

- Determines whether the field is read only
- Field name: ***editable***
- Value: boolean

Example:

```
“editable”: true
```

Output

- Values that are being returned from the server can be validated
    - Validations are optional
- Some validations are allowed from specific data types only

	Integer / Long	**Boolean**	**Date / DateTime**	**String**	**Memo**	**Reference**
Sanitization						

Sanitization

- Enforces *sanitization* for the field’s output
    - Provides options for types of sanitization
- Field name: ***sanitization***
- Value: a string value represents an enum: *{text, html}*

Example:

```
“sanitization”: “html”
```

Example for field metadata object:

```
{
    "name": "description",
    "entity_name": "run",
    "filterable": false,
    "editable": true,
    "returned_by_default": true,
    "label": "Description",
    "sortable": false,
    "required": false,
    "sanitization": "HTML",
    "unique": false,
    "field_type": "memo",
    "max_length": null
}
```

Attachments

Attachments REST API is a special one, and thus has its own section. The difference between the *attachments* API and all the rest, is that attachment represents entity data and in addition binary content.

- Attachments currently are supported only in workspace context (as entity metadata reflects).
- Attachments API is generic access for all attachments in the workspace. As a result, for each entity which has the *attachments* feature, the *attachment* entity has a reference field to the entity.
    - The reference field names are by the following convention: ***owner_<entity name>***, example: *owner_test*
    - *Attachment* entity must have a reference to one and only one *owner* entity

Resource Collection

- *Resource collection* in part of the HTTP methods behaves as any other entity *resource collection* with reflecting only the *attachment* entity data without the binary content, and in part with reference to binary content.

URI

- Currently, *attachments* are relevant only the workspace context
- **http[s]://<server>:<port>/api/shared_spaces/{uid}/workspaces/{id}/attachments**

GET

- GET request on *attachments* returns only the *entity* data of the attachment, without the binary content

GET Request - Attachments

```
*** Request ***
GET
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/attachments
Accept:             application/json
Host:               nga.com:8080
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

{
    "total_count": 2,
    "data": [
        {
            "type": "attachment",
            "creation_time": "2016-02-23T09:18:29Z",
            "version_stamp": 1,
            "owner_work_item": {
```

```
                "type": "defect",
                "id": 5003
            },
            "name": "tests.txt",
            "owner_test": null,
            "description": null,
            "id": 4003,
            "last_modified": "2016-02-23T09:18:29Z"
        },
        {
            "type": "attachment",
            "creation_time": "2016-02-23T09:41:48Z",
            "version_stamp": 1,
            "owner_work_item": null,
            "name": "temp2.txt",
            "owner_test": {
                "type": "test_manual",
                "id": 2002
            },
            "description": null,
            "id": 4004,
            "last_modified": "2016-02-23T09:41:48Z"
        }
    ]
}
```


PUT

- Supports updating only the *attachment* entity data, as described in *PUT resource collection*
- In order to update the binary content of the attachment, use DELETE existing and POST new

PUT Request - Attachments

```
*** Request ***
PUT
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/attachments
Accept:             application/json
Content-Type:       application/json
Host:               nga.com:8080

{
    "data": [
        {
            "type": "attachment",
            "description": “description test 1”,
            "id": 4003
        },
        {
            "type": "attachment",
```

```
            "description": “description test 2”,
            "id": 4004
        }
    ]

}
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

{
    "total_count": 2,
    "data": [
        {
            "type": "attachment",
            "creation_time": "2016-02-23T09:18:29Z",
            "version_stamp": 1,
            "owner_work_item": {
                "type": "defect",
                “id”: 5003
            },
            "name": "tests.txt",
            "owner_test": null,
            "description": null,
            "id": 4003,
            "last_modified": "2016-02-23T09:18:29Z"
        },
        {
            "type": "attachment",
            "creation_time": "2016-02-23T09:41:48Z",
            "version_stamp": 1,
            "owner_work_item": null,
            "name": "temp2.txt",
            "owner_test": {
                "type": "test_manual",
                "id": 2002
            },
            "description": null,
            "id": 4004,
            "last_modified": "2016-02-23T09:41:48Z"
        }
    ]
}
```

DELETE

- As described in *DELETE resource collection*

POST

- Bulk POST is not supported
- In POST we have 2 different types of data
  - The *attachment's* entity data
  - The *attachment's* binary content
- We will support only *multipart/form-data* content type

POST Request - Attachments

```
*** Request ***
POST
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/attachments
Accept:             application/json
Content-Type:       multipart/form-data; boundary=----------------------
-----25852296509330
Host:               nga.com:8080

-----------------------------25852296509330
Content-Disposition: form-data; name="entity"
Content-Type: application/json

{
    "name":"test.txt",
    "owner_work_item": {
        "id":5003,
        "type": "defect"
    }
}
-----------------------------25852296509330
Content-Disposition: form-data; name="content"; filename="test.txt"
Content-Type: text/plain

text file testing

-----------------------------25852296509330--
```

```
*** Response ***
HTTP/1.1 201 Created
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

{
    "total_count": 1,
    "data": [
        {
            "type": "attachment",
```

```
            "creation_time": "2016-02-23T09:18:29Z",
            "version_stamp": 1,
            "owner_work_item": {
                "type": "defect",
                “id”: 5003
            },
            "name": "test.txt",
            "owner_test": null,
            "description": null,
            "id": 4005,
            "last_modified": "2016-02-23T09:18:29Z"
        }
    ]
}
```

Resource Instance

Attachment resource instance contains both the *attachment's* entity data and the binary content

GET

Supports 2 content types:

- application/json – retrieve only the *attachment's* entity data
- application/octet-stream – retrieve only the *attachment's* binary content

GET Request – Attachment resource instance – entity data

```
*** Request ***
GET
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/attachments/4003
Accept:             application/json
Host:               nga.com:8080
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/json;q=0.9
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

{
    "type": "attachment",
    "creation_time": "2016-02-23T09:18:29Z",
    "version_stamp": 1,
    "owner_work_item": {
        "type": "defect",
        "id": 5003
    },
    "name": "tests.txt",
    "owner_test": null,
```

```
    "description": null,
    "id": 4003,
    "last_modified": "2016-02-23T09:18:29Z"
}
```

GET Request – Attachment resource instance – entity binary content (the content is a text file with the following content: “text file testing”)

```
*** Request ***
GET
/api/shared_spaces/421m74dkge2omf0elrvvy7r3d/workspaces/1003/attachments/4003
Accept:             application/octet-stream
Host:               nga.com:8080
```

```
*** Response ***
HTTP/1.1 200 OK
Content-Encoding:   gzip
Content-Type:       application/octet-stream
Date:               Mon, 27 Mar 2014 12:11:05 GMT
Server:             nginx

text file testing
```

PUT

- Only the *attachment’s* entity data can be updated
- As described in *PUT resource instance*

DELETE

- As described in *DELETE resource instance*

POST

- Not supported