senaite.jsonapi Documentation

Release 1.2.3

Riding Bytes
Naralabs

Aug 05, 2020

Contents

This add-on is a RESTful JSON API for SENAITE LIMS, that allows to Create, Read and Update (CRU operations) through http GET/POST requests. It uses JSON as the format for data representation.

The development of SENAITE JSONAPI was strongly driven by the experience gained while developing plone.jsonapi.routes, with which SENAITE JSONAPI shares most of the underlying software design solutions. The main difference between them is that plone.jsonapi.routes is a Plone-specific RESTful JSON API, while *senaite.jsonapi* is SENAITE-specific. For these very same reasons, this documentation is an adapted version of plone.jsonapi.routes's documentation, with the consent of it's author.

This documentation is divided in different parts. We recommend that you get started with *Installation* and then head over to the *Quickstart*. Please check out the *API* documentation for internals about *senaite.jsonapi*.

Table of Contents:

CHAPTER 1

Installation

To install senaite.jsonapi in your SENAITE instance, simply add this add-on in your buildout configuration file as follows, and run *bin/buildout* afterwards:

```
[buildout]

...

[instance]
...
eggs =
    ...
    senaite.jsonapi
```

With this configuration, buildout will download and install the latest published release of senaite.jsonapi from Pypi.

The routes for SENAITE LIMS content types get registered on startup. The following URL should be available after startup:

http://localhost:8080/senaite/@@API/senaite/v1

1.1 JSON Viewers and REST clients

There are plenty of add-ons for browsers that beautify the generated JSON, making it's interpretation more comfortable for humans. Below, some plugins you can install in your browser:

- JSONView for Firefox
- JSON Lite for Firefox
- JSONView for Google Chrome

Below, some applications to send POST requests to senaite.jsonapi:

- RESTClient for Firefox
- Advanced REST Client for Google Chrome

CHAPTER 2

Quickstart

This section gives an introduction about senaite.jsonapi. It assumes you have SENAITE LIMS and *senaite.jsonapi* already installed. The JSON API is therefore located at *http://localhost:8080/senaite/@@API/senaite/v1*. Make sure your SENAITE LIMS instance is located on the same URL, so you can directly click on the links within the examples.

All the coming examples are executed directly in Google Chrome. *JSONView_* is used to beautify the generated JSON and the Advanced Rest Client Application to send POST requests to senaite.jsonapi. See *Installation* for details.

2.1 Version route

The *version* route prints out the current version of *senaite.jsonapi*.

http://localhost:8080/senaite/@@API/senaite/v1/version


```
{
    url: "http://localhost:8080/senaite/@@API/senaite/v1/version",
    date: "2020-03-03",
    version: "1.2.2",
    _runtime: 0.0036830902099609375
}
```


Note: The runtime indicates the time spent in milliseconds until the response is prepared.

2.2 Content Routes

senaite.jsonapi allows you to directly retrieve contents by their *portal_type* name. These *Resources* are automatically generated for **all** available content types in SENAITE.

Each content route is located at the *Base URL*, e.g.

- http://localhost:8080/senaite/@@API/senaite/v1/client

- http://localhost:8080/senaite/@@API/senaite/v1/analysisrequest

The name of each of these content routes is transformed to lower case, so it is also perfectly ok to call these *Resources* like so:

- http://localhost:8080/senaite/@@API/senaite/v1/Client
- http://localhost:8080/senaite/@@API/senaite/v1/AnalysisRequest

For instance, calling a content route like

- http://localhost:8080/senaite/@@API/senaite/v1/client

will return a JSON containing records of type *Client* only:


```
{
    count: 1596,
    pagesize: 25,
    items: [
        {
            uid: "ffce0bba48204c63a62b0744a6b762bf",
            id: "client1",
            ...
            portal_type: "Client",
            ...
        },
        {},
        {},
        ...
    ],
    page: 1,
    _runtime: 0.09960794448852539,
    next: "http://localhost:8080/senaite/@@API/senaite/v1/client?b_start=25",
    pages: 64,
    previous: null
}
```


Some examples of searches for SENAITE-specific portal types below:

- Analysis Services: http://localhost:8080/senaite/@@API/senaite/v1/analysisservice
- Calculations: http://localhost:8080/senaite/@@API/senaite/v1/calculation
- Samples: http://localhost:8080/senaite/@@API/senaite/v1/analysisrequest
- Worksheets: http://localhost:8080/senaite/@@API/senaite/v1/worksheet

Check out senaite.core's types.xml for the full list of portal types that come with SENAITE LIMS by default. Keep in mind that *senaite.jsonapi* will also handle other portal types that might be registered by other add-ons. For instance, SENAITE Health, an extension for health-care labs registers a new portal type named *Patient*. If you have this add-on installed, the url http://locahost:8080/senaite/@@API/senaite/v1/patient will work as well, returning the list of objects from type *Patient*.

From the JSON response above, note the following:

The *Response Format* in *senaite.jsonapi* content URLs is always the same. The top level keys (data after the first `{`) are meta information about the gathered data.

The *items* list will contain the list of results. Each result is a record with just the metadata available in the catalog. Therefore, no object is "waked up" at this stage. This is because of the APIs two step concept, which postpones expensive operations, until the user really wants it.

All *items* are batched to increase performance of the API. The *count* number returns the total number objects found, while the *page* number returns the number of pages in the batch, which can be navigated with the *next* and *previous* links.

2.2.1 Get records full data

To get all data from an object, you can either add the `complete=True` parameter, or you can request the data with the object `UID`.

- http://localhost:8080/senaite/@@API/senaite/v1/client?complete=True
- http://localhost:8080/senaite/@@API/senaite/v1/client/<uid>
- http://localhost:8080/senaite/@@API/senaite/v1/<uid>

The requested content(s) is now loaded by the API and all fields are gathered.

Note: Please keep in mind that large data sets with the *?complete=True* Parameter might increase the loading time significantly.

2.3 UID Route

To fetch the full data of an object immediately, it is also possible to append the UID of the object directly on the root URL of the API, e.g.:

- http://localhost:8080/senaite/@@API/senaite/v1/ffce0bba48204c63a62b0744a6b762bf
- http://localhost:8080/senaite/@@API/senaite/v1/client/ffce0bba48204c63a62b0744a6b762bf

---

Note: The given UID might seem different on your machine.

The response will give the data in the root of the JSON data, so only the object metadata is returned, e.g.:


```
{
    expirationDate: "2019-05-02T11:53:13+02:00",
    _runtime: 0.03150486946105957,
    exclude_from_nav: null,
    BankBranch: null,
    Fax: null,
    title: "Happy Hills",
    parent_id: "clients",
    location: null,
    parent_url: "http://localhost:8080/senaite/@@API/senaite/v1/clientfolder/
↪b7e8d2288af74092afe0cf3a0e172f87",
    PhysicalAddress: {
        city: "Barcelona",
        district: "",
        zip: "",
        country: "Spain",
        state: "Catalonia",
        address: ""
    },
    portal_type: "Client",
```


(continues on next page)

(continued from previous page)


```
    AccountName: null,
    language: "en",
    BulkDiscount: null,
    parent_uid: "b7e8d2288af74092afe0cf3a0e172f87",
    parent_path: "/senaite/clients",
    rights: null,
    AccountNumber: null,
    modified: "2019-07-24T23:14:57+02:00",
    EmailAddress: null,
    BillingAddress: {
        city: "",
        district: "",
        zip: "",
        country: "",
        state: "",
        address: ""
    },
    ...
}
```

CHAPTER 3

Authentication

The API provides a simple way to authenticate a user with SENAITE.

3.1 Login

URL Schema `<BASE URL>/login?__ac_name=<username>&__ac_password=<password>`

The response will set the *__ac* cookie for further cookie authenticated requests.

Note: Currently only cookie authentication works. Other PAS plugins might not work as expected.

Example

`http://localhost:8080/senaite/@@API/senaite/v1/login?__ac_name=admin&__ac_password=admin`

Response


```
{
    url: "http://localhost:8080/senaite/@@API/senaite/v1/users",
    count: 1,
    _runtime: 0.0019960403442382812,
    items: [
        {
            username: "admin",
            authenticated: true,
            last_login_time: "",
            roles: [
                "Manager",
                "Authenticated"
            ],
            url: "http://localhost:8080/senaite/@@API/senaite/v1/users/admin",
            email: null,
            groups: [ ],
```


(continues on next page)

(continued from previous page)


```
            fullname: null,
            id: "admin",
            login_time: ""
        }
    ]
}
```


3.2 Logout

URL Schema `<BASE URL>/users/logout`

The response will expire the *__ac* cookie for further requests.

Example

`http://localhost:8080/senaite/@@API/senaite/v1/users/logout`

Response


```
{
    url: "http://localhost:8080/senaite/@@API/senaite/v1/users",
    _runtime: 0.0009028911590576172,
    success: true
}
```


3.3 Basic Authentication

URL Schema `<BASE URL>/auth`

If the request is not authenticated, this route will raise an unauthorized response with status code 401. Browsers should display the Basic Authentication login.

Example

`http://localhost:8080/senaite/@@API/senaite/v1/auth`

CHAPTER 4

API

This part of the documentation covers all resources (routes) provided by senaite.jsonapi. It also covers all the request parameters that can be applied to these resources to refine the results.

4.1 Concept

The SENAITE JSON API aims to be **as fast as possible**. So the concept of the API is to postpone **expensive operations** until the user really requests it. To do so, the API was built with a **two step architecture**.

An **expensive operation** is basically given, when the API needs to "wake up" an object to retrieve all its field values. This means the full object has to be loaded from the Database (ZODB) into the memory (RAM).

The **two step architecture** retrieves only the fields of the catalog results in the *first step*. Only if the user requests the API URL of a specific object, the object will be loaded and all the fields of the object will be returned.

Note: You can add a *complete=yes* parameter to bypass the two step behavior and retrieve the full object data immediately.

4.2 Base URL

After installation, the SENAITE API routes are available below the senaite.jsonapi root URL (`@@API`), with the base `/senaite/api/v1`.

Example: `http://localhost:8080/senaite/@@API/senaite/v1/version`

4.3 Resources

URL Schema `<BASE URL>/<RESOURCE>/<OPERATION>/<uid:optional>`

A resource is equivalent with the portal type name in SENAITE.

This means that all portal types are fully supported by the API simply by adding the portal type to the end of the base url, e.g.:

- http://localhost:8080/senaite/@@API/senaite/v1/Client
- http://localhost:8080/senaite/@@API/senaite/v1/AnalysisService
- http://localhost:8080/senaite/@@API/senaite/v1/AnalysisRequest

Note: Lower case portal type names are also supported.

4.4 Operations

The API understands the basic CRUD operations on the *content resources*. Only the READ operation is accessible via a HTTP GET request. All other operations have to be sent via a HTTP POST request.

OPERATION	URL	METHOD
READ	<BASE URL>/<RESOURCE>/<uid:optional>	GET
CREATE	<BASE URL>/<RESOURCE>/create/<uid:optional>	POST
UPDATE	<BASE URL>/<RESOURCE>/update/<uid:optional>	POST
DELETE	<BASE URL>/<RESOURCE>/delete/<uid:optional>	POST

Note: For traceability reasons, *delete* operation is not supported in SENAITE LIMS. When *delete* operation is used, the system tries to deactivate the object instead.

It is also possible to get the contents by UID directly from the base url, without the need of <RESOURCE>, e.g.:

- http://localhost:8080/senaite/@@API/senaite/v1/<uid>

This principle not applies to VIEW operation only, but to UPDATE and DELETE too. When the UID is directly used, <RESOURCE> becomes optional:

OPERATION	URL	METHOD
READ	<BASE URL>/<RESOURCE:optional>/<uid>	GET
CREATE	<BASE URL>/<RESOURCE:optional>/create/<uid>	POST
UPDATE	<BASE URL>/<RESOURCE:optional>/update/<uid>	POST
DELETE	<BASE URL>/<RESOURCE:optional>/delete/<uid>	POST

Therefore, the following urls are also valid:

- http://localhost:8080/senaite/@@API/senaite/v1/create/<uid>
- http://localhost:8080/senaite/@@API/senaite/v1/update/<uid>
- http://localhost:8080/senaite/@@API/senaite/v1/delete/<uid>

4.5 Users Resource

The API is capable to find SENAITE users, e.g.:

- http://localhost:8080/senaite/@@API/senaite/v1/users
- http://localhost:8080/senaite/@@API/senaite/v1/users/current
- http://localhost:8080/senaite/@@API/senaite/v1/users/<username>


```
{
    count: 50,
    pagesize: 25,
    items: [
        {
            username: "jordi",
            visible_ids: false,
            linked_contact_uid: "e980f398c233488b96d733a49b73c8b8",
            authenticated: false,
            api_url: "http://localhost:8080/senaite/@@API/senaite/v1/users/jordi",
            roles: [
                "Member",
                "LabManager",
                "Authenticated"
            ],
            home_page: "",
            description: "",
            wysiwyg_editor: "",
            location: "",
            error_log_update: 0,
            language: "",
            listed: true,
            groups: [
                "AuthenticatedUsers",
                "Clients",
                "LabManagers",
            ],
            portal_skin: "",
            fullname: "Jordi Puiggené",
            login_time: "2000-01-01T00:00:00",
            email: "jp@naralabs.com",
            ext_editor: false,
            last_login_time: "2000-01-01T00:00:00"
        },
    ],
    page: 1,
    _runtime: 0.008383989334106445,
    next: "http://localhost:8080/senaite/@@API/senaite/v1/users?b_start=25",
    pages: 2,
    previous: null
}
```


The results come as well as batches of 25 items per default. It is also possible to get a higher or lower number of users per batch with the *?limit=n* request parameter, e.g.:

- http://localhost:8080/senaite/@@API/senaite/v1/users?limit=1

Note: This route lists all users for **authenticated** users only.

The username *current* is reserved to fetch the current logged in user:

- http://localhost:8080/senaite/@@API/senaite/v1/users/current

4.5.1 Overview

Resource	Action	Description
users	<username>,current	Resource for SENAITE Users
auth		Basic Authentication
login		Login with __ac_name and __ac_password
logout		De-authenticate

4.6 Catalogs Resource

senaite.jsonapi is capable to retrieve information about the catalogs registered in the system, as well as the indexes and metadata fields (schema) they contain:

- http://localhost:8080/senaite/@@API/senaite/v1/catalogs
- http://localhost:8080/senaite/@@API/senaite/v1/catalogs/<catalog_id>

For each catalog, the following information is provided:

- *id*: the unique identifier of the catalog
- *indexes*: the list of indexes the catalog contains (used for searches)
- *schema*: the list of metadata fields the catalog contains
- *portal_types*: types that are indexed in this catalog

Example:

- http://localhost:8080/senaite/@@API/senaite/v1/catalogs/bika_catalog


```
{
    _runtime: 0.0061838626861572266,
    id: "bika_catalog",
    schema: [
        "Created",
        "Description",
        "Title",
        "Type",
        "UID",
        "creator",
        ...
    ],
    portal_types: [
        "Batch",
        "ReferenceSample",
    ],
    indexes: [
        "BatchDate",
        "Creator",
        "Description",
        "Title",
        "Type",
        "UID",
        ...
    ]
}
```

Note: the *indexes* of a catalog can either be used as filters for searching results and as criteria for sorting the results.

Note: *schema* fields are the keys of the values *senaite.jsonapi* will display in a search query for a given resource and catalog in accordance with the *two step architecture* strategy explained in *Concept*.

4.7 Search Resource

The search route omits the portal type and is therefore capable to search for **any** content type within the portal that is indexed in *portal_type* catalog.

The search route accepts all available indexes which are defined in the portal catalog tool, e.g.:

- http://localhost:8080/senaite/@@API/senaite/v1/search

Returns **all** contents indexed in *portal_catalog*.

- http://localhost:8080/senaite/@@API/senaite/v1/search?id=test

Returns contents that match with the given value of the *id* parameter.

By default, Plone objects are stored in a generalist catalog, named *portal_catalog*. SENAITE LIMS is built on top of Plone and also makes use of this generalist catalog, but **not all objects are stored in this catalog**. Rather, SENAITE LIMS follows a multi-catalog approach given the heterogeneity of object types it contains, with different requirements in terms of indexes for searches. The immediate benefit is that system becomes more performant, but at a cost: the user has to know the catalog to search against.

4.7.1 Searches by catalog

You can check the catalogs registered in the system and locate the portal type you want to search with the route *catalogs*, as explained in *Catalogs Resource*.

Not all catalogs have same indexes, so once you know the catalog to search against, you might need to check the indexes it contains you are using a supported parameter for your search.

The following is a catalog-specific search (note the param *catalog* in the url):

- http://localhost:8080/senaite/@@API/senaite/v1/search?id=WB-00012&catalog=bika_catalog_analysisrequest_listing

Returns the contents indexed with id *WB-00012* in the specified catalog. This catalog only contains objects from type *AnalysisRequest* (aka Sample), so we expect this query to return a single item, a Sample:


```
{
    count: 1,
    pagesize: 25,
    items: [
        {
            getSampleTypeUID: "39cbccd290a64894853d9d28ad297d33",
            getProgress: 40,
            getDueDate: "2020-05-01T16:01:23+02:00",
            getBatchID: "",
            getContactFullName: "Rita Mohale",
            url: "http://localhost:8080/senaite/clients/client-1/WB-00012",
```


(continues on next page)

(continued from previous page)

```
            path: "/senaite/senaite/clients/client-1/WB-00012",
            uid: "19697c28034a4d3a960540b938203b50",
            id: "WB-00012",
            getDateSampled: "2020-04-27T00:00:00+02:00",
            parent_id: "client-1",
            getInternalUse: false,
            api_url: "http://localhost:8080/senaite/@@API/senaite/v1/analysisrequest/
↪19697c28034a4d3a960540b938203b50",
            getClientTitle: "Happy Hills",
            portal_type: "AnalysisRequest",
            ...
        }
    ],
    page: 1,
    _runtime: 9.699778079986572,
    next: null,
    pages: 1,
    previous: null
}
```

Note: Remember that *senaite.jsonapi* follows a **two-step strategy** on searches, so only the catalog metadata of the item is displayed unless you add the parameter *&complete=True* in the URL.

4.7.2 Searches by index

Search of resources supports the use of indexes as filter criteria. Note that we've used the param *id* in the above mentioned searches. In fact, *id* is an index that is present either in default *portal_catalog* and in the catalog for which we've done the catalog-specific search.

Remember you can check the indexes available for any given catalog by using the Catalogs route. For instance:

- http://localhost:8080/senaite/@@API/senaite/v1/search?portal_type=Client

Will return all the objects their value for *portal_type* index is *Client* and that are stored in the default catalog *portal_catalog*. Obviously, this url returns exactly the same result as if we were using the route *client*:

- http://localhost:8080/senaite/@@API/senaite/v1/client

But *portal_catalog* has other indexes that might be of our interest for searches:

- http://localhost:8080/senaite/@@API/senaite/v1/search?review_state=inactive

Will return the items, regardless of the type, that are stored in *portal_catalog* that are in inactive status.

Searches by index can also be used against other catalogs:

- http://localhost:8080/senaite/@@API/senaite/v1/search?getClientID=HHILLS&bika_catalog_analysisrequest_listing

Will return all the samples assigned to client with id *HHILLS*. Note this is not the internal ID of the client object, rather the id assigned manually by user on Client creation.

We can also combine multiple indexes in our search:

- http://localhost:8080/senaite/@@API/senaite/v1/search?getClientID=HHILLS&review_state=published&catalog=bika_catalog_analysisrequest_listing

Will return the samples assigned to client with id *HHILLS* their status is *published*.

4.7.3 Sorting and limiting results

Results can also be sorted by any index present in the catalog, by using the *sort_on* parameter:

- http://localhost:8080/senaite/@@API/senaite/v1/search?getClientID=HHILLS&review_state=published&sort_on=getDateSampled&catalog=bika_catalog_analysisrequest_listing

Will return the samples assigned to client with id *HHILLS* their status is *published*, sorted by date sampled ascending. We can also sort the results descending with parameter *sort_order*:

- http://localhost:8080/senaite/@@API/senaite/v1/search?getClientID=HHILLS&review_state=published&sort_on=getDateSampled&sort_order=desc&catalog=bika_catalog_analysisrequest_listing

In addition to sorting, we can also limit the number of results to a given number:

- http://localhost:8080/senaite/@@API/senaite/v1/search?getClientID=HHILLS&review_state=published&sort_on=getDateSampled&sort_order=desc&limit=10&catalog=bika_catalog_analysisrequest_listing

Will return the first 10 samples that are assigned to a client with id *HHILLS*, their status is *published*, sorted by date sampled descending.

4.8 Parameters

URL Schema `<BASE URL>/<RESOURCE>?<KEY>=<VALUE>&<KEY>=<VALUE>`

All content resources accept to be filtered by request parameters.

Key	Value	Description
q	searchterm	Search the SearchableText index for the given query string
path	/physical/path	Specifiy a physical path to only return results below it. See how to Query by path in the Plone docs for details.
depth	0..n	Specify the depth of a path query. Only relevant when using the path parameter.
catalog	catalog name	Search for results against the specified catalog
limit	1..n	Limit the results to the given *limit* number. This will return batched results with *x* pages and *n* items per page
sort_on	catalog index	Sort the results by the given index
sort_order	asc / desc	Sort ascending or descending (default: ascending)
sort_limit	1..n	Limit the result set to n items. The portal catalog will only return n items.
complete	yes/y/1/True	Flag to return the full object results immediately. Bypasses the *two step* behavior of the API
children	yes/y/1/True	Flag to return the folder contents of a folder below the *children* key Only visible if complete flag is true or if an UID is provided
workflow	yes/y/1/True	Flag to include the workflow data below the *workflow* key
filedata	yes/y/1/True	Flag to include the base64 encoded file
recent_created	today, yesterday this-week, this-month this-year	Specify a recent created date range, to find all items created within this date range until today. This uses internally *'range': 'min'* query.
recent_modified	today, yesterday this-week, this-month this-year	Specify a recent modified date range, to find all items modified within this date range until today. This uses internally *'range': 'min'* query.

4.9 Response Format

The response format is for all resources the same.

```
{
    count: 1, // number of found items
    pagesize: 25, // items per page
    items: [  // List of all item objexts
        {
            id: "front-page", // item data
            ...
        }
    ],
    page: 1, // current page
    _runtime: 0.00381,  // calculation time to generate the data
    next: null,  // URL to the next batch
    pages: 1,  //  number of total pages
    previous: null  // URL to the previous batch
}
```

count The number of found items – can be more than displayed on one site

pagesize Number of items per page

items List of found items – only catalog brain keys unless you add a *complete=yes* parameter to the request or request an URL with an UID at the end.

page The current page of the batched result set

_runtime The time in milliseconds needed to generate the data

next The URL to the next batch

pages The number of pages in the batch

previous The URL to the previous batch

CHAPTER 5

CRUD

Each content route provider shipped with this package, provides the basic CRUD *Operations* functionality to *create*, *read*, *update* and *delete* the resource handled, except that the *delete* operation tries to deactivate the resource instead of deleting it. The reason is that for traceability reasons, *delete* operation is not supported in SENAITE LIMS.

Keep in mind that available operations are strongly bound to permissions, so the operation will only take place if the user has enough privileges for that operation and resource status.

5.1 Unified API

URL Schema `<BASE URL>/<OPERATION>/<uid:optional>`

There is a convenient and unified way to fetch the content without knowing the resource. This unified resource is directly located at the *Base URL*.

5.2 CREATE

The *create* route will create the content inside the container located at the given UID.

http://localhost:8080/senaite/@@API/senaite/v1/<RESOURCE:optional>/create/<uid:optional>

The given RESOURCE defines the type of object to create. You can omit this value and specify the type with *portal_type* variable in the HTTP POST body. Check *Operations*: for more information.

The given optional UID defines the target container. You can omit this UID and specify all the information in the HTTP POST body.

The following are the POST parameters required for the creation of any type of object:

- *portal_type*: The type name of to object to be created (e.g. *Client*), Required if *<RESOURCE>* is omitted in the url.
- *parent_path*: Physical path of the parent container (e.g. */senaite/clients*), Required if *<uid>* is omitted in the url.

Note: *parent_uid* (the UID of the parent container) can be used instead of *parent_path*

Additional fields might be required depending on the resource to be created. For instance, for the creation of a *Client* object, values for two additional fields are required: *Name* and *ClientID*.

Important: SENAITE.JSONAPI does not allow the creation of objects when:

- the container is the portal root (*senaite* path)
- the container is senaite's setup (*senaite/bika_setup* path)
- the container does not allow the specified *portal_type*

In such cases, *senaite.jsonapi* will always return a 401 response.

The examples below show possible variations of a HTTP POST body sent to the JSON API with the header **Content-Type: application/json** set. Remember you can use the Advanced Rest Client Application to send POST requests. See *Installation* for details.

5.2.1 Example: Client creation

Request URL:

http://localhost:8080/senaite/@@API/senaite/v1/create

Body Content type (application/json):

```
{
    "portal_type": "Client",
    "title": "Test Client",
    "ClientID": "TEST-01",
    "parent_path": "/senaite/clients"
}
```

5.2.2 Example: Sample Type creation

Request URL:

http://localhost:8080/senaite/@@API/senaite/v1/create

Body Content type (application/json):

```
{
    "portal_type": "SampleType",
    "title": "Test Sample Type",
    "description": "This is a new Sample Type",
    "Hazardous": false,
    "Prefix": "TST",
    "MinimumVolume": "10 mL",
    "RetentionPeriod": {
      "days": 5,
      "hours": 0,
      "minutes": 0
    },
```

(continues on next page)

(continued from previous page)

```
    "parent_path": "/senaite/bika_setup/bika_sampletypes"
}
```

5.2.3 Example: Sample Creation

Request URL:

http://localhost:8080/senaite/@@API/senaite/v1/AnalysisRequest/create/<client_uid>

Body Content type (application/json):

```
{
    "Contact": <client_contact_uid>,
    "SampleType": <sample_type_uid>,
    "DateSampled": "2020-03-05 14:21:20",
    "Template": <ar_template_uid>,
}
```

where:

- <*client_uid*> is the UID of the Client
- <*client_contact_uid*> is the UID of a Contact from the Client
- <*sample_type_uid*> is the UID of the Sample Type
- <*ar_template_uid*> is the UID of the Sample Template

Note: In this example, the RESOURCE (*AnalysisRequest*) has been defined in the url, as well as the parent container. This is also supported, as explained in *Operations*. Remember that in SENAITE LIMS, the portal type that represents samples is *AnalysisRequest*.

5.3 READ

The *read* route does not exist, use the base url to retrieve a content by uid, as explained in *Operations*. E.g.:

http://localhost:8080/senaite/@@API/senaite/v1/<uid>

Please, refer to *Search Resource* section to learn how to search objects.

5.4 UPDATE

The *update* route will update the content located at the given UID.

http://localhost:8080/senaite/@@API/senaite/v1/update/<uid:optional>

The given optional UID defines the object to update. You can omit this UID and specify all the information in the HTTP POST body by using either:

- *path* parameter, as the physical path to the object, or
- *uid* parameter, as the UID of the object

Alternatively, you can use *id* and *parent_path* parameters with the values from the parent container as well.

Important: SENAITE.JSONAPI does not allow the update of objects when:

- the container is the portal root (*senaite* path)
- the container is senaite's setup (*senaite/bika_setup* path)

In such cases, *senaite.jsonapi* will always return a 401 response.

The *update* route can also be used to perform transitions by using the keyword *transition* in the HTTP POST body.

The examples below show possible variations of a HTTP POST body sent to the JSON API with the header **Content-Type: application/json** set. Remember you can use the Advanced Rest Client Application to send POST requests. See *Installation* for details.

5.4.1 Example

Given this Request URL:

http://localhost:8080/senaite/@@API/senaite/v1/update

the following POSTs are equivalent, all them update the "Priority" of sample DBS-00012 to 2:

```
{
    "path": "/senaite/clients/client-1/DBS-00012",
    "Priority": 2,
}
```

```
{
    "uid": <uid_of_sample_DBS-00012>,
    "Priority": 2,
}
```

```
{
    "id": "DBS-00012",
    "parent_path": "/senaite/clients/client-1",
    "Priority": 2,
}
```

Using the same URL with this HTTP POST body:

```
{
    "uid": <uid_of_sample_DBS-00012>,
    "Priority": 2,
    "transition": "receive"
}
```

will update the "Priority" field of the sample to *2* and will perform the transition "receive" to the Sample with id *DBS-00012*. This transition will only take place if the sample is in a suitable status and the user has enough privileges for the transition to take place.

5.5 DELETE

The *delete* route will deactivate the content located at the given UID.

http://localhost:8080/senaite/@@API/senaite/v1/delete/<uid:optional>

The given optional UID defines the object to deactivate. You can omit this UID and specify all the information in the HTTP POST body.

5.5.1 Example

Deactivate an object by its **physical path**:

http://localhost:8080/senaite/@@API/senaite/v1/delete?path=/senaite/clients/client-1

Or you can specify the **parent path** and the **id** of the object

http://localhost:8080/Plone/@@API/plone/api/1.0/delete?parent_path=/senaite/clients&id=client-1

Or you can specify these information in the request body:

```
{
    uid: "<object_uid>"
}
```

CHAPTER 6

Customizing

This package is built to be extended. You can either use the *Zope Component Architecture* and provide an specific Adapter to control what is being returned by the API or you simply write your own route provider.

This section will show how to build a custom route provider for an example content type. It will also show how to write and register a custom data adapter for this content type. It is even possible to customize how the fields of a specific content type can be accessed or modified.

6.1 Adding a custom route provider

Each route provider shipped with this package, provides the basic CRUD functionality to *get*, *create*, *delete* and *update* the resource handled.

The same functionality can be used to provide this behavior for custom content types. All necessary functions are located in the *api* module within this package.

```
# CRUD
from senaite.jsonapi.api import get_batched
from senaite.jsonapi.api import create_items
from senaite.jsonapi.api import update_items
from senaite.jsonapi.api import delete_items

# route dispatcher
from senaite.jsonapi import add_route

# GET
@add_route("/todos", "todos", methods=["GET"])
@add_route("/todos/<string:uid>", "todos", methods=["GET"])
def get(context, request, uid=None):
    """ get all todos
    """
    return get_batched("Todo", uid=uid, endpoint="todo")
```

You can also specify an own *query* and pass it to the *get_batched* function of the api. This gives full control over the executed query on the catalog:

```
@add_route("/mytodos", "mytodos", methods=["GET"])
def mytodos(context, request):
    """ Returns all my todos
    """
    myself =
    query = {"portal_type": "Todo",
             "creator": api.get_current_user().getId() }
    return get_batched(query=query)
```

Note: Other keywords (except *uid*) are ignored, if the *query* keyword is detected.

The upper example registers a function named *get* with the *add_route* decorator. This ensures that this function gets called when the */todos* route is called, e.g. *http://localhost:8080/senaite/@@API/senaite/v1/todos*.

The second argument of the decorator is the endpoint, which is kind of the registration key for our function. The last argument is the methods we would like to handle here. In this case we're only interested in GET requests.

All route providers get always the *context* and the *request* as the first two arguments. The *uid* keyword argument is passed in, when a UID was appended to the URL, e.g *http://localhost:8080/senaite/@@API/v1/senaite/todo/a3f3f9efd0b4df190d16ea63d*.

The *get_batched* function we call inside our function will do all the heavy lifting for us. We simply need to pass in the *portal_type* as the first argument, the *UID* and the *endpoint*.

To be able to create, update and delete our *Todo* content type, it is necessary to provide the following functions as well. The behavior is analogue to the upper example but as there is no need for batching, the functions return a Python *<list>* instead of a complete mapping as above.

```
ACTIONS = "create,update,delete,cut,copy,paste"

# http://werkzeug.pocoo.org/docs/0.11/routing/#builtin-converters
# http://werkzeug.pocoo.org/docs/0.11/routing/#custom-converters
@route("/<any(" + ACTIONS + "):action>",
       "senaite.jsonapi.v1.action", methods=["POST"])
@route("/<any(" + ACTIONS + "):action>/<string(maxlength=32):uid>",
       "senaite.jsonapi.v1.action", methods=["POST"])
@route("/<string:resource>/<any(" + ACTIONS + "):action>",
       "senaite.jsonapi.v1.action", methods=["POST"])
@route("/<string:resource>/<any(" + ACTIONS + "):action>/<string(maxlength=32):uid>",
       "senaite.jsonapi.v1.action", methods=["POST"])
def action(context, request, action=None, resource=None, uid=None):
    """Various HTTP POST actions

    Case 1: <action>
    <site_id>/@@API/v1/senaite/<action>

    Case 2: <action>/<uid>
    -> The actions (update, delete) will performed on the object identified by <uid>
    -> The action (create) will use the <uid> as the parent folder
    <site_id>/@@API/v1/senaite/<action>/<uid>

    Case 3: <resource>/<action>
    -> The "target" object will be located by a location given in the request body␣
↪(uid, path, parent_path + id)
```

(continues on next page)

(continued from previous page)

```
    -> The actions (update, delete) will performed on the target object
    -> The action (create) will use the target object as the container
    <site_id>/@@API/v1/senaite/<resource>/<action>

    Case 4: <resource>/<action>/<uid>
    -> The actions (update, delete) will performed on the object identified by <uid>
    -> The action (create) will use the <uid> as the parent folder
    <Plonesite>/@@API/plone/api/1.0/<resource>/<action>
    """

    # Fetch and call the action function of the API
    func_name = "{}_items".format(action)
    action_func = getattr(api, func_name, None)
    if action_func is None:
        api.fail(500, "API has no member named '{}'".format(func_name))

    portal_type = api.resource_to_portal_type(resource)
    items = action_func(portal_type=portal_type, uid=uid)

    return {
        "count": len(items),
        "items": items,
        "url": api.url_for("senaite.jsonapi.v1.action", action=action),
    }
```

6.2 Adding a custom data adapter

The data returned by the API for each content type is extracted by the *IInfo* Adapter. This Adapter simply extracts all field values from the content.

To customize how the data is extracted from the content, you have to register an adapter for a more specific interface on the content.

This adapter has to implement the *IInfo* interface.

```
from senaite.jsonapi.interfaces import IInfo
from zope import interface


class TodoAdapter(object):
    """ A custom adapter for Todo content types
    """
    interface.implements(IInfo)

    def __init__(self, context):
        self.context = context

    def to_dict(self):
        return {} # whatever data you need

    def __call__(self):
        # just implement it like this, don't ask x_X
        return self.to_dict()
```

Register the adapter in your *configure.zcml* file for your special interface:

```
<configure
    xmlns="http://namespaces.zope.org/zope">

    <!-- Adapter for my custom content type -->
    <adapter
        for="my.addon.interfaces.ITodo"
        factory=".adapters.TodoAdapter"
        />

</configure>
```

6.3 Adding a custom data manager

The data sent by the API for **each content type** is set by the *IDataManager* Adapter. This Adapter has a simple interface:

```
class IDataManager(interface.Interface):
    """ Field Interface
    """

    def get(name):
        """ Get the value of the named field with
        """

    def set(name, value):
        """ Set the value of the named field
        """

    def json_data(name, default=None):
        """ Get a JSON compatible structure from the value
        """
```

To customize how the data is set to each field of the content, you have to register an adapter for a more specific interface on the content. This adapter has to implement the *IDataManager* interface.

Note: The *json_data* function is called by the Data Provider Adapter (*IInfo*) to get a JSON compatible return Value, e.g.: DateTime('2017/05/14 14:46:18.746800 GMT+2') -> "2017-05-14T14:46:18+02:00"

Important: Please be aware that you have to implement security for field level access on your own.

```
from persistent.dict import PersistentDict
from senaite.jsonapi.interfaces import IDataManager
from zope import interface
from zope.annotation import IAnnotations


class TodoDataManager(object):
    """ A custom data manager for Todo content types
    """
    interface.implements(IDataManager)
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, context):
        self.context = context

    @property
    def storage(self):
        return IAnnotations(self.context).setdefault('my.addon.todo',␣
↪PersistentDict())

    def get(self, name):
        self.storage.get("name")

    def set(self, name, value):
        self.storage["name"] = value
```

Register the adapter in your *configure.zcml* file for your special interface:

```
<configure
    xmlns="http://namespaces.zope.org/zope">

    <!-- Adapter for my custom content type -->
    <adapter
        for="my.addon.interfaces.ITodo"
        factory=".adapters.TodoDataManager"
        />

</configure>
```

6.4 Adding a custom field manager

The default data managers (*IDataManager*) defined in this package know how to *set* and *get* the values from fields. But sometimes it might be useful to be more granular and know how to *set* and *get* a value for a **specific field**.

Therefore, *senaite.jsonapi* introduces Field Managers (*IFieldManager*), which adapt a field.

This Adapter has a simple interface:

```
class IFieldManager(interface.Interface):
    """A Field Manager is able to set/get the values of a single field.
    """

    def get(instance, **kwargs):
        """Get the value of the field
        """

    def set(instance, value, **kwargs):
        """Set the value of the field
        """

    def json_data(instance, default=None):
        """Get a JSON compatible structure from the value
        """
```

To customize how the data is set to each field of the content, you have to register a more specific adapter to a field.

This adapter has to implement then the *IFieldManager* interface.

Note: The *json_data* function is called by the Data Manager Adapter (*IDataManager*) to get a JSON compatible return Value, e.g.: DateTime('2017/05/14 14:46:18.746800 GMT+2') -> "2017-05-14T14:46:18+02:00"

Note: The *json_data* method is defined on context level (*IDataManger*) as well as on field level (*IFieldManager*). This is to handle objects w/o fields, e.g. Catalog Brains, Portal Object etc. and Objects which contain fields and want to delegate the JSON representation to the field.

Important: Please be aware that you have to implement security for field level access on your own.

```
class DateTimeFieldManager(ATFieldManager):
    """Adapter to get/set the value of DateTime Fields
    """
    interface.implements(IFieldManager)

    def set(self, instance, value, **kw):
        """Converts the value into a DateTime object before setting.
        """
        try:
            value = DateTime(value)
        except SyntaxError:
            logger.warn("Value '{}' is not a valid DateTime string"
                        .format(value))
            return False

        self._set(instance, value, **kw)

    def json_data(self, instance, default=None):
        """Get a JSON compatible value
        """
        value = self.get(instance)
        return api.to_iso_date(value) or default
```

Register the adapter in your *configure.zcml* file for your special interface:

```
<configure
    xmlns="http://namespaces.zope.org/zope">

  <!-- Adapter for AT DateTime Field -->
  <adapter
      for="Products.Archetypes.interfaces.field.IDateTimeField"
      factory=".fieldmanagers.DateTimeFieldManager"
      />

</configure>
```

6.5 Adding a custom catalog tool

Note: Remember *senaite.jsonapi* searches against *portal_catalog* by default, but you can search against other catalogs by using the *catalog* parameter in the search query. See _Search_Resource for further information.

All search is done through a catalog adapter. This adapter has to provide at least a *search* method. The others are optional, but recommended.

```
class ICatalog(interface.Interface):
    """ Catalog interface
    """

    def search(query):
        """ search the catalog and return the results
        """

    def get_catalog():
        """ get the used catalog tool
        """

    def get_indexes():
        """ get all indexes managed by this catalog
        """

    def get_index(name):
        """ get an index by name
        """

    def to_index_value(value, index):
        """ Convert the value for a given index
        """
```

To customize the catalog tool to get full control of the search, you have to register an catalog adapter for a more specific interface on the portal. This adapter has to implement the *ICatalog* interface.

```
from senaite.jsonapi.interfaces import ICatalog
from senaite.jsonapi import api
from zope import interface


class MyCatalog(object):
    """My Catalog adapter
    """
    interface.implements(ICatalog)

    def __init__(self, context):
        self._catalog = api.get_tool("my_catalog")

    def search(self, query):
        """search the catalog
        """
        catalog = self.get_catalog()
        return catalog(query)
```

Register the adapter in your *configure.zcml* file for your special interface:

```
<configure
    xmlns="http://namespaces.zope.org/zope">
```

(continues on next page)

(continued from previous page)

```
    <!-- Adapter for a custom catalog adapter -->
    <adapter
        for=".interfaces.ICustomPortalMarkerInterface"
        factory=".catalog.MyCatalog"
        />

</configure>
```

6.6 Adding a custom catalog query adapter

Note: Remember *senaite.jsonapi* searches against *portal_catalog* by default, but you can search against other catalogs by using the *catalog* parameter in the search query. See _Search_Resource for further information.

All search is done through a catalog adapter. The *ICatalogQuery* adapter provides a suitable query usable for the *ICatalog* adapter. It should at least provide a *make_query* method.

```
class ICatalogQuery(interface.Interface):
    """ Catalog query interface
    """

    def make_query(**kw):
        """ create a new query or augment an given query
        """
```

To customize a custom catalog tool to perform a search, you have to register an catalog adapter for a more specific interface on the portal. This adapter has to implement the *ICatalog* interface.

```
from senaite.jsonapi.interfaces import ICatalogQuery
from zope import interface


class MyCatalogQuery(object):
    """MyCatalog query adapter
    """
    interface.implements(ICatalogQuery)

    def __init__(self, catalog):
        self.catalog = catalog

    def make_query(self, **kw):
        """create a query suitable for the catalog
        """
        query = {"sort_on": "created", "sort_order": "descending"}
        query.update(kw)
        return query
```

Register the adapter in your *configure.zcml* file for your special interface:

```
<configure
    xmlns="http://namespaces.zope.org/zope">

```

(continues on next page)

(continued from previous page)

```
    <!-- Adapter for a custom query adapter -->
    <adapter
        for=".interface.ICustomCatalogInterface"
        factory=".catalog.MyCatalogQuery"
        />

</configure>
```

6.7 Adding an adapter for create operation

SENAITE JSONAPI is *portal_type-naive*. This means that this add-on delegates the responsibility of creation operation to the underlying add-on where the given portal type is registered. This is true in most cases, except when:

- the container is the portal root (*senaite* path)
- the container is senaite's setup (*senaite/bika_setup* path)
- the container does not allow the specified *portal_type*

For the cases above, *senaite.jsonapi* will always return a 401 response.

Sometimes, one might want to handle the creation of a given object differently, either because:

- you want a portal type to never be created through *senaite.jsonapi*
- you want a portal type to only be created in some specific circumstances
- you want to add some additional logic within the creation process
- etc.

SENAITE.JSONAPI provides the *ICreate* interface that allows you to handle the *create* operation with more granularity. An Adapter of this interface is initialized with the container object to be created. This interface provides the following signatures:

```
class ICreate(interface.Interface):
    """Interface to handle creation of objects
    """

    def is_creation_allowed(self):
        """Returns whether the creation of this portal type for the given
        container is allowed
        """

    def is_creation_delegated(self):
        """Return whether the creation of this portal type has to be delegated
        to this adapter
        """

    def create_object(self, **data):
        """Creates an object
        """
```

6.7.1 Allow/disallow the creation of a content type

For instance, say you don't want to allow the creation of objects from type *Todo* through the *senaite.jsonapi*:

```
from senaite.jsonapi.interfaces import ICreate
from zope import interface


class TodoCreateAdapter(object):
    """Custom adapter for the creation of Todo type
    """
    interface.implements(ICreate)

    def __init__(self, container):
        self.container = container

    def is_creation_allowed(self):
        """Returns whether the creation of the portal_type is allowed
        """
        return False
```

Register the adapter in your *configure.zcml* file for your special interface:

```
<configure
    xmlns="http://namespaces.zope.org/zope">

    <!-- Adapter for a creation custom adapter -->
    <adapter
      name="Todo"
      factory=".TodoCreateAdapter"
      provides="senaite.jsonapi.interfaces.ICreate"
      for="*" />

</configure>
```

Note: This is a "named" adapter in which the name is the portal type.

Note that if you wanted this *Todo* type to be created through *senaite.jsonapi*, except inside the container *Client*, you could do so by registering the adapter for *IClient* type only:

```
<configure
    xmlns="http://namespaces.zope.org/zope">

    <!-- Adapter for custom creation of Todo -->
    <adapter
      name="Todo"
      factory=".TodoCreateAdapter"
      provides="senaite.jsonapi.interfaces.ICreate"
      for="bika.lims.interfaces.IClient" />

</configure>
```

Note: We've used here a custom *Todo* type, but you can use this approach for any type registered in the system, being it from *senaite.core* (e.g. *Client', 'SampleType*, etc.) or from any other add-on.

6.7.2 Custom creation of a content type

As we've explained before, you might want to have full control on the creation of a given portal type because you have to add additional logic. You can use the same adapter as before:

```
from Products.CMFPlone.utils import _createObjectByType
from senaite.jsonapi.interfaces import ICreate
from zope import interface


class TodoCreateAdapter(object):
    """Custom adapter for the creation of Todo type
    """
    interface.implements(ICreate)

    def __init__(self, container):
        self.container = container

    def is_creation_allowed(self):
        """Returns whether the creation of the portal_type is allowed
        """
        return True

    def is_creation_delegated(self):
        """Returns whether the creation of this portal type has to be
        delegated to this adapter
        """
        return True

    def create_object(self, **data):
        """Creates an object
        """
        obj = _createObjectByType("Todo", self.container, tmpID())
        obj.edit(**data)
        obj.unmarkCreationFlag()
        obj.reindexObject()
        return obj
```

With this example, *senaite.jsonapi* will not follow the default procedure of creation, but delegate the operation to the function *create_object* of this adapter. Note the creation will only be delegated when the function *is_creation_delegated* returns True.

6.8 Adding an adapter for update operation

Sometimes, one might want to handle the update of a given object differently, either because:

- you want an object to never be updated through *senaite.jsonapi*
- you want an object to only be updated in some specific circumstances
- you want to add some additional logic within the update process
- etc.

Adding a custom data manager or *Adding a custom field manager* allows to achieve these goals partially, cause their scope is at field level. If you need full control over the update process, you can also create an adapter implementing *IUpdate* interface. This interface allows you to handle the *update* operation by your own. This interface provides the folllowing signatures:

```
class IUpdate(interface.Interface):
    """Interface to handle update of objects
    """

    def is_update_allowed(self):
        """Returns whether the update of the object is allowed
        """

    def update_object(self, **data):
        """Updates the object
        """
```

6.8.1 Allow/disallow to update an object

For instance, say you don't want to allow the update of objects from type *Todo* through the *senaite.jsonapi*:

```
from senaite.jsonapi.interfaces import IUpdate
from zope import interface


class TodoUpdateAdapter(object):
    """Custom adapter for the update of objects from Todo type
    """
    interface.implements(IUpdate)

    def __init__(self, context):
        self.context = context

    def is_update_allowed(self):
        """Returns whether the update of the object is allowed
        """
        return False
```

Register the adapter in your *configure.zcml* file for your special interface:

```
<configure
    xmlns="http://namespaces.zope.org/zope">

    <!-- Adapter for custom update -->
    <adapter
      factory=".TodoUpdateAdapter"
      provides="senaite.jsonapi.interfaces.IUpdate"
      for="my.addon.interfaces.ITodo" />

</configure>
```

Note: This adapter is initialized with *context*, the object to be updated.

Note: We've used here a custom *Todo* type, but you can use this approach for any type registered in the system, being it from *senaite.core* (e.g. *Client', 'SampleType*, etc.) or from any other add-on.

6.8.2 Custom update of an object

Imagine that besides updating your object, you want to add a *Remarks* at the same time. You can use the same adapter as before:

```
from senaite.jsonapi.interfaces import IUpdate
from zope import interface


class TodoUpdateAdapter(object):
    """Custom adapter for the update of objects from Todo type
    """
    interface.implements(IUpdate)

    def __init__(self, context):
        self.context = context

    def is_update_allowed(self):
        """Returns whether the update of the object is allowed
        """
        return True

    def update_object(self, **data):
        """Updates the object
        """
        self.context.setRemarks("Updated through json.api")
        self.context.edit(**data)
        self.context.reindexObject()
```

With this example, *senaite.jsonapi* will not follow the default procedure of update, but delegate the operation to the function *update_object* of this adapter.

CHAPTER 7

Doctests

7.1 AUTH

Running this test from the buildout directory:

bin/test test_doctests -t auth

7.1.1 Test Setup

Needed Imports:

```
>>> import transaction
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

Functional Helpers:

```
>>> def get(url):
...     browser.open("{}/{}".format(api_url, url))
...     return browser.contents
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> api_url = "{}/@@API/senaite/v1".format(portal_url)
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

7.1.2 Login

User can login with a GET:

```
>>> get("login?__ac_name={}&__ac_password={}".format(TEST_USER_ID, TEST_USER_
↪PASSWORD))
'..."authenticated": true...'
```

And once logged, *auth* route does not rise an unauthorized response 401:

```
>>> get("auth")
'{"_runtime": ...}'
```

7.1.3 Logout

User can logout easily too:

```
>>> get("users/logout")
'..."authenticated": false...'
```

And *auth* route rises an unauthorized response 401:

```
>>> get("auth")
Traceback (most recent call last):
[...]
HTTPError: HTTP Error 401: Unauthorized
```

7.2 VERSION

Running this test from the buildout directory:

bin/test test_doctests -t version

7.2.1 Test Setup

Needed Imports:

```
>>> import transaction
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

Functional Helpers:

```
>>> def logout():
...     browser.open(portal_url + "/logout")
...     assert("You are now logged out" in browser.contents)
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

JSON API:

```
>>> api_base_url = portal_url + "/@@API/senaite/v1"
```

7.2.2 Authenticated user

The version route should be visible to authenticated users:

```
>>> browser.open(api_base_url + "/version")
>>> browser.contents
'{"url": "http://nohost/plone/@@API/senaite/v1/version", "date": "...", "version": ...
↪, "_runtime": ...}'
```

7.2.3 Unauthenticated user

Log out:

```
>>> logout()
```

The version route should be visible to unauthenticated users too:

```
>>> browser.open(api_base_url + "/version")
>>> browser.contents
'{"url": "http://nohost/plone/@@API/senaite/v1/version", "date": "...", "version": ...
↪, "_runtime": ...}'
```

7.3 USERS

Running this test from the buildout directory:

bin/test test_doctests -t users

7.3.1 Test Setup

Needed Imports:

```
>>> import json
>>> import transaction
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

Functional Helpers:

```
>>> def get(url):
...     browser.open("{}/{}".format(api_url, url))
...     return browser.contents
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> api_url = "{}/@@API/senaite/v1".format(portal_url)
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

7.3.2 Get all users

The API is capable to find SENAITE users:

```
>>> response = get("users")
>>> data = json.loads(response)
>>> items = data.get("items")
>>> sorted(map(lambda it: it["username"], items))
[u'test-user', u'test_analyst_0',...u'test_labmanager_1']
```

And for each user, the roles and groups are displayed:

```
>>> analyst = filter(lambda it: it["username"] == "test_analyst_0", items)[0]
>>> sorted(analyst.get("roles"))
[u'Analyst', u'Authenticated', u'Member']
```

```
>>> sorted(analyst.get("groups"))
[u'Analysts', u'AuthenticatedUsers']
```

As well as other properties:

```
>>> sorted(analyst.keys())
[u'api_url', u'authenticated', u'description', u'email', ...]
```

7.3.3 Get current user

Current user can also be retrieved easily:

```
>>> response = get("users/current")
>>> data = json.loads(response)
>>> data.get("count")
1
>>> current = data.get("items")[0]
>>> current.get("username")
u'test-user'
```

and includes all properties too:

```
>>> sorted(current.keys())
[u'api_url', u'authenticated', u'description', u'email',...u'groups',...u'roles'...]
```

7.3.4 Get a single user

A single user can be retrieved too:

```
>>> get("users/test_analyst_0")
'..."username": "test_analyst_0"...'
```

7.4 CATALOGS

Running this test from the buildout directory:

    bin/test test_doctests -t catalogs

7.4.1 Test Setup

Needed Imports:

```
>>> import json
>>> import transaction
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

Functional Helpers:

```
>>> def get(url):
...     browser.open("{}/{}".format(api_url, url))
...     return browser.contents
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> api_url = "{}/@@API/senaite/v1".format(portal_url)
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

7.4.2 Get all catalogs

senaite.jsonapi is capable to retrieve information about the catalogs registered in the system:

```
>>> response = get("catalogs")
>>> data = json.loads(response)
>>> items = data.get("items")
>>> catalog_ids = map(lambda cat: cat["id"], items)
>>> sorted(catalog_ids)
[u'auditlog_catalog', ..., u'portal_catalog']
```

Catalogs for internal use are not included though:

```
>>> "uid_catalog" in catalog_ids
False
```

```
>>> "reference_catalog" in catalog_ids
False
```

For each catalog, indexes, schema fields and allowed portal types are listed:

```
>>> cat = filter(lambda it: it["id"]=="portal_catalog", items)[0]
>>> sorted(cat.get("indexes"))
[u'Analyst', u'Creator', u'Date', ...]
```

```
>>> sorted(cat.get("schema"))
[u'Analyst', u'CreationDate', u'Creator',...]
```

```
>>> sorted(cat.get("portal_types"))
[u'ARReport', u'ARTemplate', u'ARTemplates',...]
```

7.4.3 Get a single catalog

A single catalog can also be retrieved by it's id:

```
>>> response = get("catalogs/portal_catalog")
>>> cat = json.loads(response)
>>> sorted(cat.get("indexes"))
[u'Analyst', u'Creator', u'Date', ...]
```

```
>>> sorted(cat.get("schema"))
[u'Analyst', u'CreationDate', u'Creator',...]
```

```
>>> sorted(cat.get("portal_types"))
[u'ARReport', u'ARTemplate', u'ARTemplates',...]
```

7.5 SEARCH

Running this test from the buildout directory:

bin/test test_doctests -t search

7.5.1 Test Setup

Needed Imports:

```
>>> import json
>>> import transaction
```

```
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

```
>>> from bika.lims import api
```

Functional Helpers:

```
>>> def get(url):
...     browser.open("{}/{}".format(api_url, url))
...     return browser.contents
```

```
>>> def get_count(response):
...     data = json.loads(response)
...     return data.get("count")
```

```
>>> def get_items_ids(response, sort=True):
...     data = json.loads(response)
...     items = data.get("items")
...     items = map(lambda it: it["id"], items)
...     if sort:
...         return sorted(items)
...     return items
```

```
>>> def init_data():
...     api.create(portal.clients, "Client", title="Happy Hills", ClientID="HH")
...     api.create(portal.clients, "Client", title="ACME", ClientID="AC")
...     api.create(portal.clients, "Client", title="Fill the gap", ClientID="FG")
...     api.create(setup.bika_sampletypes, "SampleType", title="Water", Prefix="W")
...     api.create(setup.bika_sampletypes, "SampleType", title="Dust", Prefix="D")
...     transaction.commit()
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> api_url = "{}/@@API/senaite/v1".format(portal_url)
>>> setup = api.get_setup()
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
```

Initialize the instance with some objects for testing:

```
>>> init_data()
```

7.5.2 Basic search

We can directly search by resource:

```
>>> response = get("client")
>>> get_count(response)
3
>>> get_items_ids(response)
[u'client-1', u'client-2', u'client-3']
```

We can also add search criteria as well:

```
>>> response = get("client?id=client-1")
>>> get_count(response)
1
>>> get_items_ids(response)
[u'client-1']
```

```
>>> response = get("client?getName=ACME")
>>> get_count(response)
1
>>> get_items_ids(response)
[u'client-2']
```

7.5.3 Sort and limit

We can use sort and limit too:

```
>>> response = get("client?sort_on=id&sort_order=asc")
>>> get_items_ids(response, sort=False)
[u'client-1', u'client-2', u'client-3']
```

```
>>> response = get("client?sort_on=id&sort_order=desc")
>>> get_items_ids(response, sort=False)
[u'client-3', u'client-2', u'client-1']
```

```
>>> response = get("client?sort_on=id&sort_order=desc&limit=2")
>>> get_items_ids(response, sort=False)
[u'client-3', u'client-2']
```

7.5.4 Search without resource

We can also omit the resource and search directly by portal_type:

```
>>> response = get("search?portal_type=Client")
>>> get_items_ids(response)
[u'client-1', u'client-2', u'client-3']
```

Additional search criteria and sorting works as well:

```
>>> response = get("search?portal_type=Client&getName=ACME")
>>> get_items_ids(response)
[u'client-2']
```

```
>>> response = get("search?portal_type=Client&sort_on=id&sort_order=desc&limit=2")
>>> get_items_ids(response, sort=False)
[u'client-3', u'client-2']
```

7.5.5 Catalog search

We can specify the catalog to use in searches. Sample Types are stored in both portal_catalog and setup_catalog:

```
>>> response = get("sampletype")
>>> get_items_ids(response)
[u'sampletype-1', u'sampletype-2']
```

```
>>> response = get("sampletype?catalog=portal_catalog")
>>> get_items_ids(response)
[u'sampletype-1', u'sampletype-2']
```

```
>>> response = get("sampletype?catalog=bika_setup_catalog")
>>> get_items_ids(response)
[u'sampletype-1', u'sampletype-2']
```

But Sample Types are not stored in "bika_catalog":

```
>>> response = get("sampletype?catalog=bika_catalog")
>>> get_items_ids(response)
[]
```

7.6 CREATE

Running this test from the buildout directory:

bin/test test_doctests -t create

7.6.1 Test Setup

Needed Imports:

```
>>> import json
>>> import transaction
>>> import urllib
>>> from DateTime import DateTime
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

```
>>> from bika.lims import api
```

Functional Helpers:

```
>>> def get(url):
...     browser.open("{}/{}".format(api_url, url))
...     return browser.contents
```

```
>>> def post(url, data):
...     url = "{}/{}".format(api_url, url)
...     browser.post(url, urllib.urlencode(data, doseq=True))
...     return browser.contents
```

```
>>> def create(data):
...     response = post("create", data)
...     assert("items" in response)
...     response = json.loads(response)
...     items = response.get("items")
...     assert(len(items)==1)
...     item = response.get("items")[0]
...     assert("uid" in item)
...     return api.get_object(item["uid"])
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> api_url = "{}/@@API/senaite/v1".format(portal_url)
>>> setup = api.get_setup()
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

7.6.2 Create with resource

We can create an object by providing the resource and the parent uid directly in the request:

```
>>> clients = portal.clients
>>> clients_uid = api.get_uid(clients)
>>> url = "client/create/{}".format(clients_uid)
>>> data = {"title": "Test client 1",
...         "ClientID": "TC1"}
>>> post(url, data)
'...clients/client-1"...'
```

We can also omit the parent uid while defining the resource, but passing the uid of the container via post:

```
>>> data = {"title": "Test client 2",
...         "ClientID": "TC2",
...         "parent_uid": clients_uid}
>>> post("client/create", data)
'...clients/client-2"...'
```

We can use *parent_path* instead of *parent_uid*:

```
>>> data = {"title": "Test client 3",
...         "ClientID": "TC3",
...         "parent_path": api.get_path(clients)}
>>> post("client/create", data)
'...clients/client-3"...'
```

7.6.3 Create without resource

Or we can create an object without the resource, but with the parent uid and defining the portal_type via post:

```
>>> url = "create/{}".format(clients_uid)
>>> data = {"title": "Test client 4",
...         "ClientID": "TC4",
...         "portal_type": "Client"}
>>> post(url, data)
'...clients/client-4"...'
```

7.6.4 Create via post only

We can omit both the resource and container uid and pass everything via post:

```
>>> data = {"title": "Test client 5",
...         "ClientID": "TC5",
...         "portal_type": "Client",
...         "parent_path": api.get_path(clients)}
>>> post("create", data)
'...clients/client-5"...'
```

```
>>> data = {"title": "Test client 6",
...         "ClientID": "TC6",
...         "portal_type": "Client",
...         "parent_uid": clients_uid}
>>> post("create", data)
'...clients/client-6"...'
```

If we do a search now for clients, we will get all them:

```
>>> output = get("client")
>>> output = json.loads(output)
>>> items = output.get("items")
>>> items = map(lambda it: it.get("getClientID"), items)
>>> sorted(items)
[u'TC1', u'TC2', u'TC3', u'TC4', u'TC5', u'TC6']
```

7.6.5 Required fields

System will fail with a 400 error when trying to create an object without a required attribute:

```
>>> data = {"portal_type": "SampleType",
...         "parent_path": api.get_path(setup.bika_sampletypes),
...         "title": "Fresh Egg",
...         "Prefix": "FE"}
>>> post("create", data)
Traceback (most recent call last):
[...]
HTTPError: HTTP Error 400: Bad Request
```

7.6.6 Create a Client

```
>>> data = {"portal_type": "Client",
...         "parent_path": api.get_path(clients),
...         "title": "Omelette corp",
...         "ClientID": "EC"}
>>> client = create(data)
>>> client.getClientID()
'EC'
>>> api.get_parent(client)
<ClientFolder at /plone/clients>
```

7.6.7 Create a Client Contact

```
>>> data = {"portal_type": "Contact",
...         "parent_path": api.get_path(client),
...         "Firstname": "Proud",
...         "Surname": "Hen"}
>>> contact = create(data)
>>> contact.getFullname()
'Proud Hen'
>>> api.get_parent(contact)
<Client at /plone/clients/client-7>
```

7.6.8 Create a Sample Type

```
>>> data = {"portal_type": "SampleType",
...         "parent_path": api.get_path(setup.bika_sampletypes),
...         "title": "Fresh Egg",
...         "MinimumVolume": "10 gr",
...         "Prefix": "FE"}
>>> sample_type = create(data)
>>> sample_type.Title()
'Fresh Egg'
>>> sample_type.getPrefix()
'FE'
>>> api.get_parent(sample_type)
<SampleTypes at /plone/bika_setup/bika_sampletypes>
```

7.6.9 Create a Laboratory Contact

```
>>> data = {"portal_type": "LabContact",
...         "parent_path": api.get_path(setup.bika_labcontacts),
...         "Firstname": "Lab",
...         "Surname": "Chicken"}
>>> lab_contact = create(data)
>>> lab_contact.getFullname()
'Lab Chicken'
>>> api.get_parent(lab_contact)
<LabContacts at /plone/bika_setup/bika_labcontacts>
```

7.6.10 Create a Department

```
>>> data = {"portal_type": "Department",
...         "parent_path": api.get_path(setup.bika_departments),
...         "title": "Microbiology",
...         "Manager": api.get_uid(lab_contact)}
>>> department = create(data)
>>> department.Title()
'Microbiology'
>>> api.get_parent(department)
<Departments at /plone/bika_setup/bika_departments>
```

7.6.11 Create an Analysis Category

```
>>> data = {"portal_type": "AnalysisCategory",
...         "parent_path": api.get_path(setup.bika_analysiscategories),
...         "title": "Microbiology identification",
...         "Department": api.get_uid(department)}
>>> category = create(data)
>>> category.Title()
'Microbiology identification'
>>> api.get_parent(category)
<AnalysisCategories at /plone/bika_setup/bika_analysiscategories>
>>> category.getDepartment()
<Department at /plone/bika_setup/bika_departments/department-1>
```

7.6.12 Create an Analysis Service

```
>>> data = {"portal_type": "AnalysisService",
...         "parent_path": api.get_path(setup.bika_analysisservices),
...         "title": "Salmonella",
...         "Keyword": "Sal",
...         "ScientificName": True,
...         "Price": 15,
...         "Category": api.get_uid(category),
...         "Accredited": True}
>>> sal = create(data)
>>> sal.Title()
'Salmonella'
>>> sal.getKeyword()
'Sal'
>>> sal.getScientificName()
True
>>> sal.getAccredited()
True
>>> sal.getCategory()
<AnalysisCategory at /plone/bika_setup/bika_analysiscategories/analysiscategory-1>
```

```
>>> data = {"portal_type": "AnalysisService",
...         "parent_path": api.get_path(setup.bika_analysisservices),
...         "title": "Escherichia coli",
...         "Keyword": "Ecoli",
...         "ScientificName": True,
...         "Price": 15,
...         "Category": api.get_uid(category)}
>>> ecoli = create(data)
>>> ecoli.Title()
'Escherichia coli'
>>> ecoli.getKeyword()
'Ecoli'
>>> ecoli.getScientificName()
True
>>> ecoli.getPrice()
'15.00'
>>> ecoli.getCategory()
<AnalysisCategory at /plone/bika_setup/bika_analysiscategories/analysiscategory-1>
```

7.6.13 Creating a Sample

The creation of a Sample (*AnalysisRequest* portal type) is handled differently from the rest of objects, an specific function in *senaite.core* must be used instead of the plone's default creation.

```
>>> data = {"portal_type": "AnalysisRequest",
...         "parent_uid": api.get_uid(client),
...         "Contact": api.get_uid(contact),
...         "DateSampled": DateTime().ISO8601(),
...         "SampleType": api.get_uid(sample_type),
...         "Analyses": map(api.get_uid, [sal, ecoli]) }
>>> sample = create(data)
>>> sample
<AnalysisRequest at /plone/clients/client-7/FE-0001>
```

```
>>> analyses = sample.getAnalyses(full_objects=True)
>>> sorted(map(lambda an: an.getKeyword(), analyses))
['Ecoli', 'Sal']
```

```
>>> sample.getSampleType()
<SampleType at /plone/bika_setup/bika_sampletypes/sampletype-2>
```

```
>>> sample.getClient()
<Client at /plone/clients/client-7>
```

```
>>> sample.getContact()
<Contact at /plone/clients/client-7/contact-1>
```

7.6.14 Creation restrictions

We get a 401 error if we try to create an object inside portal root:

```
>>> data = {"title": "My clients folder",
...         "portal_type": "ClientsFolder",
...         "parent_path": api.get_path(portal)}
>>> post("create", data)
Traceback (most recent call last):
[...]
HTTPError: HTTP Error 401: Unauthorized
```

We get a 401 error if we try to create an object inside setup folder:

```
>>> data = {"title": "My Analysis Categories folder",
...         "portal_type": "AnalysisCategories",
...         "parent_path": api.get_path(setup)}
>>> post("create", data)
Traceback (most recent call last):
[...]
HTTPError: HTTP Error 401: Unauthorized
```

We get a 401 error when we try to create an object from a type that is not allowed by the container:

```
>>> data = {"title": "My Method",
...         "portal_type": "Method",
```

(continues on next page)

(continued from previous page)

```
...         "parent_path": api.get_path(clients)}
>>> post("create", data)
Traceback (most recent call last):
[...]
HTTPError: HTTP Error 401: Unauthorized
```

7.7 READ

Running this test from the buildout directory:

bin/test test_doctests -t read

7.7.1 Test Setup

Needed Imports:

```
>>> import json
>>> import transaction
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

```
>>> from bika.lims import api
```

Functional Helpers:

```
>>> def get(url):
...     browser.open("{}/{}".format(api_url, url))
...     return browser.contents
```

```
>>> def get_count(response):
...     data = json.loads(response)
...     return data.get("count")
```

```
>>> def get_items_ids(response, sort=True):
...     data = json.loads(response)
...     items = data.get("items")
...     items = map(lambda it: it["id"], items)
...     if sort:
...         return sorted(items)
...     return items
```

```
>>> def init_data():
...     api.create(portal.clients, "Client", title="Happy Hills", ClientID="HH")
...     api.create(portal.clients, "Client", title="ACME", ClientID="AC")
...     api.create(portal.clients, "Client", title="Fill the gap", ClientID="FG")
...     transaction.commit()
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> api_url = "{}/@@API/senaite/v1".format(portal_url)
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Initialize the instance with some objects for testing:

```
>>> init_data()
```

7.7.2 Get resource objects

We can get the objects from a resource type:

```
>>> response = get("client")
>>> get_count(response)
3
>>> get_items_ids(response)
[u'client-1', u'client-2', u'client-3']
```

7.7.3 Get by uid

We can directly fetch a given object by it's UID and resource:

```
>>> client = api.create(portal.clients, "Client", title="Woow", ClientID="WO")
>>> uid = api.get_uid(client)
>>> transaction.commit()
>>> response = get("client/{}".format(uid))
>>> get_count(response)
1
>>> response
'..."title": "Woow"...'
```

Even with only the uid:

```
>>> response = get(uid)
>>> response
'..."title": "Woow"...'
```

but with no items in the response:

```
>>> "items" in response
False
```

```
>>> sorted(json.loads(response).keys())
[u'AccountName', u'AccountNumber', u'AccountType',...]
```

7.8 UPDATE

Running this test from the buildout directory:

bin/test test_doctests -t update

7.8.1 Test Setup

Needed Imports:

```
>>> import json
>>> import transaction
>>> import urllib
>>> from plone.app.testing import setRoles
>>> from plone.app.testing import TEST_USER_ID
>>> from plone.app.testing import TEST_USER_PASSWORD
```

```
>>> from bika.lims import api
```

Functional Helpers:

```
>>> def get(url):
...     browser.open("{}/{}".format(api_url, url))
...     return browser.contents
```

```
>>> def post(url, data):
...     url = "{}/{}".format(api_url, url)
...     browser.post(url, urllib.urlencode(data, doseq=True))
...     return browser.contents
```

```
>>> def get_item_object(response):
...     assert("items" in response)
...     response = json.loads(response)
...     items = response.get("items")
...     assert(len(items)==1)
...     item = response.get("items")[0]
...     assert("uid" in item)
...     return api.get_object(item["uid"])
```

```
>>> def create(data):
...     response = post("create", data)
...     return get_item_object(response)
```

Variables:

```
>>> portal = self.portal
>>> portal_url = portal.absolute_url()
>>> api_url = "{}/@@API/senaite/v1".format(portal_url)
>>> setup = api.get_setup()
>>> browser = self.getBrowser()
>>> setRoles(portal, TEST_USER_ID, ["LabManager", "Manager"])
>>> transaction.commit()
```

Initialize the instance with some objects for testing:

```
>>> clients = api.get_portal().clients
>>> data = {"portal_type": "Client",
...         "parent_path": api.get_path(clients),
...         "title": "Chicken corp",
```

(continues on next page)

(continued from previous page)

```
...          "ClientID": "CC"}
>>> client1 = create(data)
```

```
>>> data = {"portal_type": "Client",
...          "parent_path": api.get_path(clients),
...          "title": "Beef Corp",
...          "ClientID": "BC"}
>>> client2 = create(data)
```

```
>>> data = {"portal_type": "Client",
...          "parent_path": api.get_path(clients),
...          "title": "Octopus Corp",
...          "ClientID": "OC"}
>>> client3 = create(data)
```

7.8.2 Update by resource and uid

We can update an object by providing the resource and the uid of the object:

```
>>> client_uid = api.get_uid(client1)
>>> data = {"ClientID": "CC1"}
>>> response = post("client/update/{}".format(client_uid), data)
>>> obj = get_item_object(response)
>>> obj.getClientID()
'CC1'
```

7.8.3 Update by uid without resource

Even easier, we can update with only the uid:

```
>>> data = {"ClientID": "CC2"}
>>> response = post("update/{}".format(client_uid), data)
>>> obj = get_item_object(response)
>>> obj.getClientID()
'CC2'
```

7.8.4 Update via post only

When updating by resource (without an UID explicitly set), the system expects a the data to passed via POST to contain the item to be updated.

The object to be updated can be send in the HTTP POST body by using the *uid*:

```
>>> data = {"uid": client_uid,
...          "ClientID": "CC3"}
>>> response = post("update", data)
>>> obj = get_item_object(response)
>>> obj.getClientID()
'CC3'
```

By using the *path*, as the physical path of the object:

```
>>> data = {"path": api.get_path(client1),
...         "ClientID": "CC4"}
>>> response = post("update", data)
>>> obj = get_item_object(response)
>>> obj.getClientID()
'CC4'
```

Or by using the *id* of the object together with *parent_path*, as the physical path of the container object:

```
>>> data = {"id": api.get_id(client1),
...         "parent_path": api.get_path(clients),
...         "ClientID": "CC5"}
>>> response = post("update", data)
>>> obj = get_item_object(response)
>>> obj.getClientID()
'CC5'
```

7.8.5 Do a transition

We can transition the objects by using the keyord *transition* in the data sent via POST:

```
>>> api.is_active(client1)
True
>>> data = {"uid": api.get_uid(client1),
...         "transition": "deactivate"}
>>> response = post("update", data)
>>> obj = get_item_object(response)
>>> api.is_active(obj)
False
```

We can update and transition at same time:

```
>>> data = {"uid": api.get_uid(client1),
...         "ClientID": "CC6",
...         "transition": "activate"}
>>> response = post("update", data)
>>> obj = get_item_object(response)
>>> api.is_active(obj)
True
>>> obj.getClientID()
'CC6'
```

7.8.6 Update restrictions

We get a 401 error if we try to update an object from inside portal root:

```
>>> data = {"title": "My clients folder",
...         "uid": api.get_uid(clients),}
>>> post("update", data)
Traceback (most recent call last):
[...]
HTTPError: HTTP Error 401: Unauthorized
```

We get a 401 error if we try to update an object from inside setup folder:

```
>>> cats_uid = api.get_uid(api.get_setup().bika_analysiscategories)
>>> data = {"title": "My Analysis Categories folder",
...         "uid": cats_uid,}
>>> post("update", data)
Traceback (most recent call last):
[...]
HTTPError: HTTP Error 401: Unauthorized
```

We cannot update the *id* of an object:

```
>>> original_id = api.get_id(client1)
>>> data = {"id": "client-123123",
...         "uid": client_uid }
>>> response = post("update", data)
>>> obj = get_item_object(response)
>>> api.get_id(obj) == original_id
True
```

CHAPTER 8

Changelog

8.1 1.2.3 (2020-08-05)

- #40 Prevent the id of objects of being accidentally updated
- #40 Do not allow to update objects from setup folder
- #40 Do not allow to update objects from portal root
- #40 Fix upgrade does not work on post-only mode
- #40 Adapter for custom handling of *update* operation
- #37 Do not allow to create objects in setup folder
- #37 Do not allow to create objects in portal root
- #37 Adapter for custom handling of *create* operation
- #37 Make the creation operation to be portal_type-naive
- #35 Added *catalogs* route
- #34 Make senaite.jsonapi catalog-agnostic on searches

8.2 1.2.2 (2020-03-03)

- Missing package data

8.3 1.2.1 (2020-03-02)

- Fixed tests and updated build system

8.4 1.2.0 (2018-01-03)

Added

- Added *parent_path* to response data
- Allow custom methods as attributes in adapter

Removed

Changed

- Integration to SENAITE CORE
- License changed to GPLv2

Fixed

- #25 Null values are saved as 'NOW' in Date Time Fields
- Fixed Tests

Security

8.5 1.1.0 (2017-11-04)

- Merged PR https://github.com/collective/plone.jsonapi.routes/pull/90
- Get object by UID catalog

8.6 1.0.1 (2017-09-30)

- Fixed broken release (missing MANIFEST.in)

8.7 1.0.0 (2017-09-30)

- First release