Sympa
Integration
API

Sympa Oy
Technopolis Helsinki–Vantaa · Teknobulevardi 3–5 · FI-01530 Vantaa, Finland
+358 290 001 200 · www.sympa.fi · Domicile: Lahti, Finland · Business ID: FI19385975

Table of Contents

1 Version History

Version	Date	Author	Description
3.0.0 / 51.6.0	16.11.2021	Jarmo Sokka	Initial support for approval fields in Sympa2021. Described differences between Sympa Classic and Sympa2021.
3.0.1 / 51.6.3	30.12.2021	Jarmo Sokka	Describe http status codes in case of unsupported OData functionalities and malformed query (since, until, at). Typo fixes.
3.0.2 / 51.7.0	9.5.2022	Jarmo Sokka	Support for table row attachments.
3.0.3 / 51.7.1	1.2.2023	Amelie Archibald	Updated visuals, proofreading.
3.0.4 / 51.7.2	23.2.2023	Amelie Archibald	Correction of coding language/formatting.

2 This Document

This document provides an overview of the Sympa Integration API functionality.

This document contains information on the **latest** public release of the Import API functionality. The Import API is still under construction, so the functionality that has been implemented is somewhat limited and may change.

3 Overview

Sympa Integration API (from here on "API" in this document) provides methods for exposing data from the Sympa HR solution, and for importing data to Sympa for integration purposes. An administrator can dynamically create interfaces that provide access to basically any data stored in the HR system. Interfaces can, at this point, be configured to return data from and/or modify data in a specific card base (e.g. employees).

Configuring the API for your organization's needs can be done by any user with administrator privileges. There are no preset/fixed field lists for the interfaces, which means that basically each interface is unique. This gives admin users the possibility to restrict the transferrable fields to only ones required by the calling system, and no unnecessary data is exposed.
Setting up and configuring an API interface is done at the API panel, sympahr.net ➔ Settings ➔ API.

4 Using the API Interfaces

Calls to the API interfaces that have been configured are done by issuing HTTP GET, POST and PATCH requests to the URLs referring to each interface with the corresponding interface type. The URLs can be found in the list of configured interfaces on the API panel (admin settings). Example URL: https://api.sympahr.net/api/testinterface

Note: For security reasons, Sympa Integration API does not support Cross-Origin Resource Sharing (CORS).

4.1 Access levels

When defining the API in the panel, the user can select one or more of the interface actions to support. Enabling an action adds support to the interface for the given HTTP method (GET, POST or PATCH). If a request is executed against an API using a method that isn't supported by that API, the response will be Method not allowed (405).

- Get – the interface supports reading all data specified in the interface
- Post – the interface supports creating new data cards, and creating new rows to the tables defined in the interface
- Patch – the interface supports updating any existing data cards, and creating new rows and modifying existing rows in the tables defined in the interface
- Currently the system will not support Delete operations.

4.2 Authentication

Each request to any API interface must be authenticated, using the method specified for the API key being used.

The following examples are based on the following key/secret combinations (note: the test interface and credentials do not point to any live API, they are only used as examples for documentation purposes):

```
Key: a447eb14e84e4ecf8eae52cfb932a3b3
Secret: 26c1967a07a04c43b8abada48d1379c2
```

4.2.1 API Key

Authentication with the API interface is done using a unique API key, which enables interaction with one or more API interfaces. The Sympa Integration API supports two types of authentications:

- A plain API key
- Basic Authentication using API key and Secret

The authentication method is chosen for each API key.

ApiKey

If ApiKey authentication is being used, the API key is added as a request header named "X-ApiKey".

Example header values:
`X-ApiKey: a447eb14e84e4ecf8eae52cfb932a3b3`

Basic

Using Basic authentication, the API key function as the username, and the Secret as the password.

Example header values:
`Authorization: Basic YTQ0N2ViMTRlODRlNGVjZjhlYWU1MmNmYjkzMmEzYjM6MjZjMTk2N2EwN2EwNGM0M2I4YWJhZGE0OGQxMzc5YzI=`

Requests where the authentication fails will get an HTTP Unauthorized (401) response code. Since unauthenticated requests are not necessarily associated with any specific ApiKey or interface, these failed requests will not be visible in the API Activity Log.

4.3 Unique identifier field

The unique identifier field for the API interface must be defined. The value is a field or property on the chosen interface CardBase that can be used to uniquely identify a DataCard. The selectable values are datafields marked as “unique”, either explicitly or implicitly, on the CardBase and the “DataCardCode” property. DataCardCode isn’t a DataField, it is an automatically generated identifier that every DataCard in Sympa has.

This unique identifier field can be used to uniquely identify a DataCard in data modification requests (PATCH).

4.4 Adding interface fields

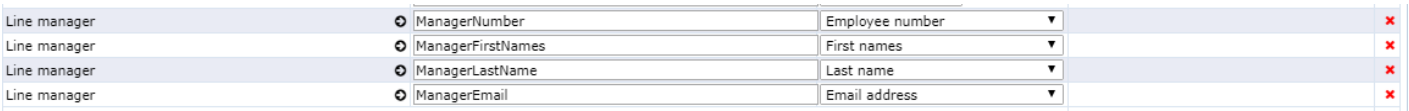
4.4.1 DataCard valued fields

DataCard fields are fields whose value is a DataCard. These include datacard, connection, supervisor and tree fields. These fields are used to establish a relation between datacards (one to one, many to many).

It’s possible to expose information about a related DataCard through an API interface. This is done by adding the field to the interface and selecting a “Target field”. This target field contains the information on the related DataCard that we want to expose.

As an example, if we want to expose an employee’s manager’s email address (employee is one datacard, manager is another datacard), we could add the following field to the API interface.

There is no limit on the amount of information about a related DataCard that can be exposed this way. The following is a valid configuration for exposing multiple pieces of information for a single datacard field.



4.5 Metadata and Service Document

4.5.1 Metadata Document request

Metadata about the interfaces associated to a particular API key can be accessed by appending `/$metadata` to the API service root URL (https://api.sympahr.net/api). The metadata is returned as XML, and it lists the available interfaces as well as their field properties.

4.5.2 Service Document request

The service returns a service document from the root URL (https://api.sympahr.net/api). This document, returned as JSON, lists all the API's available.

5 Reading data with OData queries

The Sympa Customer API implements part of the Open Data Protocol, Version 4. The supported features are described below.

5.1 Fetching data from Sympa: GET Queries

The Sympa Integration API implements part of the Open Data Protocol, Version 4. The supported features are described below. The results from all data queries are returned in JavaScript Object Notation (JSON) format. Only the subset of OData query parameters defined below are supported.

- `$top`
- `$skip`
- `$orderby`
- `$filter`
- `$count`
- `errordata` (non-standard, specific for Sympa Integration API)
- `since` & `until` (non-standard, specific for Sympa Integration API)

By default, all matches to any query are returned, which can lead to slow performance on large data sets. Therefore, it is recommended that the client request a certain number of matches to be returned per query.

In this document, we use shortened URLs, e.g. URL "`api/testinterface?$filter=...`" is just a shortening for `https://api.sympahr.net/api/testinterface?$filter=...`

5.1.1 Top

The query `$top=XX` will return the first XX matches to the query.
Example: `api/testinterface?$top=200`

5.1.2 Skip

The query `$skip=XX` will leave out the first XX matches to the query.
Example: `api/testinterface?$skip=10`
Example: `api/testinterface?$skip=10&$top=10`

5.1.3 Order by

The query `$orderby=somealias, someotheralias desc` will sort the result set, primarily by the alias "somealias", ascending, and secondarily by the alias "someotheralias", descending.
Example: `api/testinterface?$orderby=id`

Example: `api/testinterface?$orderby=id desc`
Example: `api/testinterface?$orderby=surname,age desc`

5.1.4 Filter

The following subset of OData filters are supported by the API.

and	gt	contains
or	lt	startswith
not	ge	endswith
eq	le	

Using any other OData filter operator (e.g., “in”, “has”) or query function (e.g., “now”, “tolower”, “indexof”) will return an HTTP status code *501 (Not Implemented)*.

Filter examples by field type:

- All the field types can be filtered by null or by text
  `api/testinterface?$filter=not (email eq null)`
  `api/testinterface?$filter=name eq 'no values'`

- Numeric field
  eq =equal, gt=greater than, lt = less than, ge=greater than or equal to, le=less than or equal to
  `api/testinterface?$filter=not (number eq null)`
  `api/testinterface?$filter=number ge 0`
  decimal number as filter
  `api/testinterface?$filter=number ge 1.14d`

- Dates
  eq =equal, gt=greater than, lt = less than, ge=greater than or equal to, le=less than or equal to
  `api/testinterface?$filter=not (date eq null)`
  `api/testinterface?$filter=birthday gt 1980-12-17`
  `between api/testinterface?$filter=(date gt 2020-02-29) and (date lt 2020-03-02)`

- Timestamps
  Datacard's created and modified timestamps can also be used as filter.
  `api/testinterface?$filter=(modified gt 2015-12-31T23:59:59Z)`
  Special case: If you don't specify the time, the timestamp is interpreted as T00:00:00, e.g:
  `api/testinterface?$filter=(created le 2014-01-01)`
  comparison is done with timestamp `2014-01-01T00:00:00`
  A good idea would be to always specify the time with the as a DateTime attribute.
  Note: Datacard's empty created date is shown "`0001-01-01T00:00:00`" but empty modified date is shown as "`null`"

- Identification field
  Identification field can be used in filter as the "raw" numeric value, **not the formatted value**
  e.g. `?$filter=sympaID eq '123'`

Using values that are not numbers (e.g. `sympaID eq 'a123'`) will give an error (`"message": "Error with SQM query: MissingFilterField"`)

5.1.5 Count

The query `$count=true` will add the total count of all matches, after filtering but before skip and top, as a return value named "`odata.count`".
Example:

`api/testinterface?$count=true`

The @odata.count value always contains all the results matched by the filter, i.e. top and skip do not affect the count (`api/testinterface?$skip=2&$top=1&$count=true count value is same as api/testinterface?$count=true`)

Error data

The query `errordata=true` will add a summary of all errors encountered in the data while querying, as a return value named "`odata.errors`". Note: Do not use $, because this a customized parameter for Sympa, not ODATA standard.

5.1.6 Since, until and at

To show the table rows which validity time overlaps with the given time frame use *since*, *until* and *at* (note that this requires effectivity settings in Sympa for the tables in question).
Examples:

```
GET /api/testinterface?since=2011-12-29&until=2012-02-21
GET /api/testinterface?since=2011-12-29
GET /api/testinterface?until=2012-02-21
GET /api/testinterface?at=2012-01-01
```

Note: Do not use $, because this a customized parameter for Sympa, not ODATA standard.

Using *at* together with *since* or *until* is not possible and will return an HTTP response with status code *400 (Bad Request)*.

5.1.7 Expand

`$expand` query parameter allows filtering rows inside a table. Example:

`api/testinterface?$expand=tablename($filter=modified ge 2011-01-01)`

Note: the $expand parameter filters only within the table in question, meaning the result set contains all the datacards (e.g. employees) it would contain otherwise, only the table rows are filtered.

You can combine `$expand` with other filters + attributes:
`api/testinterface?$filter=unit eq 'HR'&$expand=tablename($filter=Approval eq 5)`

In this example the datacards (e.g. employees) are filtered by unit, and the table rows by the row approval status.

5.2 Approvals

Table row approval information is returned as an integer value in the JSON message:

```
Undef = 0
WaitingSave = 1
WaitingRequest = 2
WaitingApproval = 3
Declined = 4
Approved = 5
```

6 Writing data with OData queries

As mentioned at the beginning of this document, this section contains information on the **latest** public release of the Import API functionality. The Import API is still under construction, so the functionality that has been implemented is somewhat limited and may change.

In this section we describe the syntax and format of the queries to:

- create and modify datacards and
- create and modify table rows for the datacards.

We use OData 4.0/4.01 protocol [ODATA4].

6.1 Implementation notes

6.1.1 General info

OData queries (GET requests) to the interface are performed against Sympa's pre-calculated Reporting database. There's a small delay from creating or modifying data to it being persisted to the pre-calculated Reporting database. What this means is that there is a small delay from modifying data with a POST or PATCH request before it will be queryable with a GET request. Usually the delay is less than a minute, in exceptional cases it may take up to 30 minutes.

When writing data with a POST or PATCH request, the response includes a "Location" header, the value of which is a URL fragment pointing to the edit URL for the entity being POSTed or PATCHed.

In some of the request examples below, we assume that the implementation has chosen a SerialField on the datacard as the unique identifier field. If the unique identifier field is e.g. string value (like a GUID), the URL pointing to the data card should have single quotes, for example:

```
api/InterfaceName(1234) // serial
api/InterfaceName('8520c9c4-3069-42a7-b25f-831f86ff2706') // guid
api/InterfaceName('john.doe@example.com')                  // email
```

When writing data with a POST or PATCH request, the body of the request should contain the "@odata.type" property, as seen in the examples below. The value of this property should be the interface name.

6.1.2 Current limitations

The Import API currently behaves as though every POST or PATCH request contains the “Prefer” header with value “return=minimal”. This means that if the request is handled successfully, the response code will be Not Content (204) and the response body will be empty.

In the initial implementation, all table rows are inserted in state “Approved”. Additionally, in Sympa Classic table rows that are modified will be updated to state “Approved” regardless of the previous state. In Sympa’s new UI an updated row remains in its previous state.

6.2 Create or update data cards in Sympa: POST/PATCH

In this section we describe how data cards are created to Sympa through the integration API.

We use OData 4.0/4.01 protocol [ODATA4] where applicable.

6.2.1 Create a new data card

Datacards are created with a HTTP POST request to the interface URL (e.g. `api/InterfaceName`). Each POST request creates a single new data card.

Request

```
POST api/InterfaceName
OData-Version: 4.0
Content-Type: application/json;odata.metadata=minimal
Accept: application/json
Prefer: return=minimal

{
    "@odata.type": "InterfaceName",
    "uniqueIdentifierField": "someUniqueValue123",
    "fieldName1": "value",
    // other fields to set, MUST have mandatory fields, MAY set non-mandatory fields ...
}
```

Response

As the request described Prefer: return=minimal, we will not repeat the created content in the response, as in [ODATA4-Protocol], 11.4.2.

```
HTTP/1.1 204 No content
Location: api/InterfaceName(someUniqueValue123)
OData-Version: 4.0
```

6.2.2 Update existing datacard

Datacards are updated with a HTTP PATCH request. When updating a datacard, the URL must contain the unique identifier for the datacard. The field which contains the unique identifier is defined in the API interface settings.

Request

```
PATCH api/InterfaceName(someUniqueValue123)
OData-Version: 4.0
Content-Type: application/json;odata.metadata=minimal
Accept: application/json

{
    "@odata.type": "InterfaceName",
    "fieldName1": "new value"
}
```

Response

```
HTTP/1.1 204 No content
Location: api/InterfaceName(someUniqueValue123)
OData-Version: 4.0
```

6.2.3 Example: Create employee (simplified example)

Request

```
POST api/Person
OData-Version: 4.0
Content-Type: application/json;odata.metadata=minimal
Accept: application/json
{
    "@odata.type" : "Person",
    "FirstName": "John",
    "LastName": "Doe",
    "Email": "johndoe@example.com"
}
```

Response

```
HTTP/1.1 204 No content
Location: api/InterfaceName(someUniqueValue123)
OData-Version: 4.0
```

6.2.4 Example: Update employee (simplified example)

Query:

```
PATCH api/Person(1234)
OData-Version: 4.0
Content-Type: application/json;odata.metadata=minimal
Accept: application/json

{
    "@odata.type" : "Person",
    "FirstName": "Johnny",
    "LastName": "Doe",
    "Email": "johndoe@example.com"
}
```

Response:

```
HTTP/1.1 204 No content
Location: api/InterfaceName(someUniqueValue123)
OData-Version: 4.0
```

The patch went through as it's 20x response code.

6.2.5 Example responses (failure)

If the interface is not opened for POST or PATCH requests:

```
POST api/Person

HTTP/1.1 405 Method Not Allowed
```

If the referred datacard does not exist:

```
PATCH api/Person(123)

HTTP/1.1 404 Not Found
```

6.3 Create or update individual table rows in Sympa: POST/PATCH

6.3.1 Create new table row

```
POST api/InterfaceName(uniqueId)/TableName
{
```

```
    "@odata.type": "InterfaceName.TableName",
    "tableColumnAliasName": "value"
    … table fields to to set on the newly created row..
}
```

6.3.2 Update existing table row

When updating a table row, the URL must contain the unique identifier for the table row. This is the “RowCode” property of the table row, which can be retrieved with a GET request on the API. Additionally, as mentioned before, when creating a table row through the API, the response includes a location header which points to the edit URL. This Location header will contain the RowCode.

```
PATCH api/InterfaceName(someUniqueValue123)/TableName(rowId)
{
    "@odata.type": "InterfaceName.TableName",
    "tablefieldname": "value",
    … table fields to update..
}
```

6.4 Create or update table rows through the datacard

In addition to creating and modifying datacards and their table rows individually, it’s also possible to insert table rows “through” the datacard. This means that it’s possible to create or modify a datacard as well as create new table rows for said datacard in a single request.

Note: It is **not** possible to **modify** table rows “through” the datacard. This may be possible in a later release.

6.4.1 Create new table rows when creating a datacard

```
POST api/InterfaceName
{
    "@odata.type": "InterfaceName",
    "DataCardField1": "some value",
    "DataCardField2": "some other value",
    "TableName": [{
          "tablefield1": "row1 value1",
          "tablefield2": "row1 value2"
        },{
          "tablefield1": "row2 value1",
          "tablefield2": "row2 value2"
        }
```

```
    ]
}
```

6.4.2 Create new table rows when updating a datacard

This is identical to creating new table rows when creating a datacard, except the HTTP method here is PATCH and the URL points to some datacard.

```
PATCH api/InterfaceName(uniqueId)
{
    "@odata.type": "InterfaceName",
    "DataCardField1": "some value",
    "DataCardField2": "some other value",
    "TableName": [{
          "tablefield1": "row1 value1",
          "tablefield2": "row1 value2"
        },{
          "tablefield1": "row2 value1",
          "tablefield2": "row2 value2"
        }
    ]
}
```

6.5 Create table rows for multiple existing cards in a single request

The previous request types supported various operations on a single entity, either a card or a table row. It is also possible to do a request that inserts multiple table rows for multiple cards in a single HTTP request.

This can be done by executing a PATCH request against the interface URL (e.g. `api/InterfaceName`), with a specially formulated request body.

The clearest way to describe the formulation of the request body is with an example. Take the following simplified example:

```
{
    "@context": "#$delta",   1
    "value": [  2
        {
            "EmployeeNumber": "555",  3
            "PaidSalaries@delta": [  4
                {
                    "PaidOn": "2.11.2020",
                    "Amount": "500"
                }
            ]
        }
    ]
}
```

1. The root object must always contain a property “@context” with value “#$delta”
2. The root object also contains an “value” property, whose value is an array of objects. The objects in this array represent existing datacards
3. Each datacard object must contain a value for the interface unique identifier field, which is used to identify the datacard to add table rows to. In this case, datacard with EmployeeNumber = 555 is being modified.
4. The datacard object must also have one or more properties which contain the table rows to be added. The property name is the alias for the table field + “@delta” suffix. The value is an array of objects, which represent the table rows to be added.

The outcome of the above request is that employee with EmployeeNumber “555” has one new row added to the table “PaidSalaries”.

Here’s a more useful example:

```
PATCH api/Employees
{
    "@context": "#$delta",
    "value": [
        {
            "EmployeeNumber": "555",
            "PaidSalaries@delta": [
                {
                    "PaidOn": "2020.11.2",
                    "Amount": "500"
                },
                {
                    "PaidOn": "2020.11.9",
                    "Amount": "525"
                },
                {
                    "PaidOn": "2020.11.16",
```

```
                    "Amount": "500"
                }
            ]
        },
        {
            "EmployeeNumber": "556",
            "PaidSalaries@delta": [
                {
                    "PaidOn": "2020.11.2",
                    "Amount": "200"
                },
                {
                    "PaidOn": "2020.11.16",
                    "Amount": "150"
                }
            ]
        }
    ]
}
```

The outcome of the above request is that employee with EmployeeNumber “555” has 3 new rows added to the table “PaidSalaries”, and employee with EmployeeNumber “556” has 2 new rows added to the table “PaidSalaries”.

It is also possible to add rows to more than 1 table in such a request. For example:

```
PATCH api/Employees
{
    "@context": "#$delta",
    "value": [
        {
            "EmployeeNumber": "555",
            "PaidSalaries@delta": [
                {
                    "PaidOn": "2020.11.2",
                    "Amount": "500"
                }
            ],
            "PlannedVacations@delta": [
                {
                    "StartDate": "2020.11.2",
            "EndDate": "2020.11.5"
```

```
                },
                {
                    "StartDate": "2020.11.10",
            "EndDate": "2020.11.15"
                }
            ]
        }
    ]
}
```

The outcome of the above request is that employee with EmployeeNumber “555” has 1 new row added to the table “PaidSalaries”, and 2 new rows added to the table “PlannedVacations”.

6.6 Importing different types of values to Sympa

6.6.1 Unique identifier field

Unique identifier field in this context means the field which has been selected as the “Unique identifier field” for a given API interface.

It is always possible when inserting data into Sympa through the import API (POST request) to specify a value for the unique identifier field. However, if the selected “Unique identifier field” is an identification field then it is not necessary to include a value in the request. If there is no value for the field in the request, Sympa will automatically generate a value for it. These automatically generated identifier will be returned in the Location header of the response, as usual.

6.6.2 Attachments

Attachments are supported when creating a single table row with a POST request (see section 6.3.1). Attachment support is currently limited to table row attachment. Also, attachments are not supported when querying data with GET request of modifying data with PATCH requests. GET and PATCH requests can be made against API interface containing attachment fields, but attachment fields are ignored (no data is returned etc.).

Read more about uploading attachment in section 7 (Uploading attachments).

6.6.3 Approval fields

Passing values for approval fields **is not currently supported** and it will lead to *FieldTypeNotSupported* error. See section 6.1.2 for current limitations and behavior of Approval fields.

6.6.4 Option fields

Option field (CheckBox, DropDown, Radio etc.) values should be strings and must match a value in the Mapping Table defined in the API interface definition.

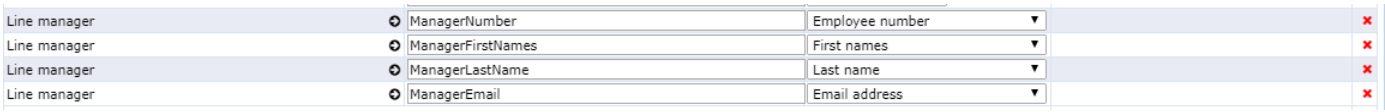
6.6.5 Multi-select option fields (CheckBox field)

Multi-select option field values must be passed as an array of strings, which must match a value in the Mapping Table defined in the API interface definition. Passing an empty array results in all of the options for the checkbox field being deselected.

6.6.6 DataCard fields

DataCard field values can be inserted to create relations between DataCards. To create this relation, a value must be passed for the **target field** of an interface field on the API interface. The target field must be **unique**, either explicitly or implicitly, and multiple values for the same DataCard valued field cannot be included in one request.

As an example, given the following configuration where “Employee number” and “Email address” are unique, and “First names” and “Last name” are not unique:



Here’s the expected behavior for a few inputs:

Valid input – ManagerNumber points to Employee number, which is unique

```
{
"ManagerNumber": "555"
}
```

Valid input – ManagerEmail points to Email address, which is unique

```
{
"ManagerEmail": "something@something.com"
}
```

Invalid input – ManagerNumber and ManagerEmail point to the same DataCard valued field “Line manager”. Multiple values for the same DataCard valued field cannot be included in one request.

```
{
"ManagerNumber": "555",
"ManagerEmail": "something@something.com"
}
```

Invalid input – ManagerFirstNames points to the First names field, which is not unique.

```
{
"ManagerFirstNames": "something"
}
```

6.6.7 Identification fields

Identification fields in Sympa generally have their values automatically generated by the system when a datacard, or table row, is inserted. This is true of the Import API, **but** the Import API also allows the caller to specify specific values for identification fields. If there is an identification field on the cardbase and/or table being imported to, and the request does not contain a value for said identification field, then a new value for it is automatically generated.

6.6.8 Date field

Date field values must be given in format “yyyy-MM-dd”. This means that a time portion must not be included.

6.6.9 Numeric field

Numeric field values must use the period decimal separator. There is a limit of 13 integral digits, that is digits before the decimal separator.

6.6.10 Boolean field

Boolean field values must be either “true” or “false”. This is not case sensitive.

6.6.11 Active property

Same as Boolean field. Values must be either “true” or “false”. This is not case sensitive.

6.6.12 Username property

Username value must be at least 5 characters and less than 50 characters in length. Username must also be unique in the organization.

6.6.13 Empty or undefined value

Saving an empty or undefined value can be achieved by passing either a null or empty string value in the request.

Note: mandatory fields, for the most part, can’t have empty or undefined value saved to them.

6.6.14 Explicitly undefined value for mandatory datacard tree fields

Mandatory datacard tree fields **can** have an “undefined” value saved for them, but only by passing a special string value of “Sympa_ExplicitSetNotDefined”.

This is possible because datacard tree fields are intended to represent non-circular hierarchies, meaning at least one node in the hierarchy must have no parents. The typical use case of this is a line manager field. There is always at least one employee at the top of the hierarchy that has no line manager (the CEO for instance).

Note: null or empty string values are not allowed for mandatory datacard tree fields.

6.7 Validations

6.7.1 Extra fields

Only fields that are on the API Interface can be imported. If the request body contains properties that do not match an alias on the API interface, the request fails.

6.7.2 Mandatory fields

Fields in on the API Interface that are marked as “Mandatory” in Sympa must have a non-null or empty string value in POST requests. Non-null or empty string values for Mandatory fields in PATCH requests are also not allowed, but the Mandatory properties themselves may be left out of the quest.

6.7.3 Unique identifier modification

Modifying an entity’s unique identifier is not allowed through the import API.

6.8 Error responses

There are many different errors that can occur during an import request. These include, but are not limited to:

- Request body is invalid JSON
- First layer API validation failures
    - mandatory fields *
    - extra data in request *
    - Filed value format validation *
    - attempted modification of unique identifier field *
- Second layer API validation fails
    - Datacard for given unique identifier doesn’t exist
    - Table row for given unique identifier doesn’t exist
    - Imported value for datacard or option field not found *
    - No mapping for option field defined *
- Import request contains an unsupported field type *
- Persistence layer validation fails *

- An unknown or unexpected error occurs

In all of the above cases, the response body contains a “code” property, which contains the HTTP status code of the request, and a “message” property which contains a description of the error that occurred.

In cases where the error relates to a specific field, or set of fields, in the request (marked with * above), the response includes a "fieldErrors” property containing field-specific error information.

Example response:

```
{
    "code": "400",
    "message": "Api request failed. See errors list for details.",
    "fieldErrors": [
        {
            "fieldAlias": "FirstName",
            "message": "Field is mandatory. Value cannot be null or empty."
            "errorCode": "FieldValueMandatory"
        }
    ]
}
```

7 Uploading attachments

API currently supports uploading table row attachments when creating a single table row. Attachments are uploaded with POST request using `multipart/form-data` content type. Single POST request can contain zero or multiple attachments. The same request can also contain the rest of the row data as JSON.

Attachment support is behind a feature flag and must be enabled separately for each organization. Please contact your Sympa contact person to enable this feature.

7.1 Request structure

HTTP request with content type `multipart/form-data` consists of one or more subparts. Request must have a Content-Type header with value `multipart/form-data`. Subparts can contain either JSON data or a binary file.

Example request can be seen below. Request creates a row with four columns: Document_type, Additional_information, Agreements, and Attachments. The first two are basic text columns and are in a subpart named “json”. The other two are attachment fields and there is a subpart for each file in the request. See section 7.1.2 to understand how files are linked to attachment fields defined in API interface.

Example request:

```
POST /api/Contracts(36)/Employment_agreements HTTP/1.1
Host: localhost:44384
Authorization: Basic
MjJiZTcxYWZiOWVkMjFkMzClYJBhMDYwMTQyMjkhZjI6M2EwoTFmZDUzYTAzN2VmNTg1ZmNiNGM1MDI2MmE2MmQ=
Content-Length: 631
Content-Type: multipart/form-data; boundary=-----WebKitFormBoundary7MA4YWxkTrZu0gW

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="json"
{ "Document_type": "Employment agreement", "Additional_information": "Employment agreement and
attachments" }

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="Agreements";
filename="/C:/Users/sympa.user/Downloads/agreement.pdf"
Content-Type: <Content-Type header here>

(data)

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="Attachments";
filename="/C:/Users/sympa.user/Downloads/attachment_1.pdf"
Content-Type: <Content-Type header here>

(data)

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="Attachments";
filename="/C:/Users/sympa.user/Downloads/attachment_2.pdf"
Content-Type: <Content-Type header here>

(data)

----WebKitFormBoundary7MA4YWxkTrZu0gW
```

7.1.1 JSON subpart

Single request can contain **zero or one** JSON data as a subpart. If the JSON subpart is present, it must have `Content-Disposition` header containing `name` parameter. The value of the parameter doesn't matter but it must be empty string at least. For the sake of clarity, for example, JSON or data can be used as a value. Posting more than one JSON subpart causes an error with HTTP status code 400 (BadRequest).

7.1.2 File subpart

Attachments are posted as binary data and for each file there must be a subpart. Single request can contain **zero or multiple** files as subparts.

Every file subpart **must have** `Content-Disposition` header containing `name` and `filename` parameter. The parameter `filename` is essential when request is parsed since it defines that the subpart is a file. `Content-Type` for subpart is optional and API doesn't use it when parsing the request.

The value of the subpart's `name` parameter must match to a field alias defined in API interface. For example, in the example above there are two fields: Attachments and Agreements. Meaning that API interface must contain attachment fields Agreements and Attachments on table Employment_agreements.

Request can also contain multiple subparts with the same name (subpart named Attachments in the example above). That simply means that multiple files can be uploaded to the same field.

7.2 Limitations

There are currently following limitations in API's attachment support:

- Only table row attachments are supported
- Attachment can be uploaded only when creating a single table row:
  `POST /api/InterfaceName(cardIdentifier)/TableName`
- Attachment fields are not returned when fetching data with a GET request
- Only PDF files can be uploaded
- Maximum size of a single attachment is around 30 MB. Note that 30 MB is also a maximum size of the whole request (including JSON, files, and HTTP request metadata). Meaning that it's possible to upload single file around 30 MB or multiple files smaller than that.

8 API Activity Log

The ten most recent requests to your organization's API interfaces, along with their statuses, metadata and any error messages, will be shown in a table on the bottom of the API panel.

9 Throttling

The default throttling level is up to 100 requests in 1h time per interface. If a more allowing level is required, please contact your Sympa contact person.

This limit of 100 requests is a shared limit across all request types, so no more than 100 requests per hour can be executed regardless of type (GET, POST, PATCH).

References

[ODATA4] OData Version 4.01 documentation, OASIS, https://www.odata.org/documentation/.

[ODATA4-Part1] OData Version 4.01. Part 1: Protocol, http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part1-protocol.html