SPHN Connector User Guide

Release 1.4.1 - 03.04.2024

Table of Contents

SPHN Connector User Guide

Release 1.4.1 - 03.04.2024

Introduction

The user guide explains how to use the SPHN Connector. The structure is based on the process of

- Preparing the SPHN Connector for usage
- For #Configure a project
- Preparing and Ingesting data into the SPHN Connector
- Processing the data and retrieving the RDF turtle files
- Json Schema creation and RML mapping (Appendix)
- Pre-checks (Appendix)
- De-Identification (Appendix)

You might find the following documents to be helpful:

- Functional Specification here
- Infrastructure Requirement here
- Installation Guide here
- Test Cases here
- Overview of the Endpoints here
- ReadTheDocs of your instance: https://{your-server}/api/docs

Notation Remark

This user guide documents the format for each endpoint with all parameters and their type:

/endpoint **(param1: type_param1, param2: type_param2, ...)**

Optional parameters are enclosed in square brackets:

/endpoint (param1: type_param1, **[param2: type_param2**], ...)

Please be aware that there are also endpoints that document a set of options as an array string:

/endpoint (param1: type_param1, **param2: [“option1”, ”option2”, ”option3”**], ...)

Note: *the parameter fields are case sensitive!*

General Remarks

The following remarks apply to all endpoints listed in this user guide:

Project - The project is the abbreviation you want to use for the project. The Project will be part of the resulting RDF turtle file. If you setup a new project in the SPHN Connector for the same project it is best to use the format {project}-{number} where the number is a sequence number like 1,2,3 etc.

Patient_id - The pipeline processes data on a patient by patient level and therefore most endpoints will provide a patient_id parameter. This can be just a sequence number or the actual patient pseudonym. It is to be noted that whatever you use as patient_id this will form part of the RDF turtle file generated.

File Compression - For all the endpoints that offer to download files in compressed format (.zip and .tar.gz) the logic is as follows:

- Only one file is returned, the file is **not** compressed even if either .zip or .tar.gz is specified.
- If more than one file is returned the compression is set to **.zip if no option is specified**.
- Otherwise the file is either zipped or tarred according to the specified option.

Concrete examples: some of the examples in the guide (especially the ones with mock data) are based on the 2023.2 RDF schema. We are currency in the process of updating them.

Releases

This section contains references to the deployed releases. The User can access the links to get more details about each release and can download it. All releases are listed here: https://git.dcc.sib.swiss/sphn-semantic-framework/sphn-connector/-/releases.
Each release is related to a specific work package. The proposal for work packages can be found here:
SPHN_Connector_vision.pptx
To better understand the numbering of releases please check out the **Release Plan**.
Suggestion: before building a new release it is recommended to cleanup the previous release setup:
docker-compose down –volumes –rmi all

SPHN Connector releases:
Release 1.4.1
Release 1.4.0
Release 1.4.0-RC2
Release 1.4.0-RC1
Release 1.3.1
Release 1.3.0

Contact/Support

Questions or issues related to the SPHN Connector should be raised via the Slack channel **sphn-connector-issues** or by directly contacting the SPHN Connector team at the email address **sphn-connector-team@lists.sib.swiss**.

Visual user guides

In order to make the navigation between the different endpoints more convenient for the end user we prepared a visual user guide for the API: api_use_guide . This will give you a brief overview in which order to trigger the endpoints. What to do when you want to do X. Please keep in mind that this is done for the most common / basic use cases. All different steps are linked to this document for further explanations. Furthermore there are guides for the Airflow service and as well for the pgadmin service available.

SPHN Connector entry page

From SPHN Connector release 1.2.0, we introduce a new entry page for the Connector. The purpose of this webpage is to give the user a general overview about the main components of the connector and how to directly reach them. All the Connector services are listed there and can be accessed easily by clicking on the corresponding links. There is a documentation section pointing to the most relevant documents needed to operate the Connector. In addition, it contains a short overview on the Connector architecture. Last but not least, the support section specifies the preferred way to contact the support team for every question or issue you may have.
The SPHN Connector entry page is available at the root endpoint of the API: *https://domain_name/*

SPHN Connector

The SPHN Connector is a dockerized software suite that enables the user to create, validate, and provide RDF files from patient data. The connector integrates a variety of other tools like the (*SHACLer*) or the (*Quality Assurance Framework*) to simplify the production of high quality data for the user.

Services

There are several services that the SPHN Connector provides in order to deliver data in an easy and efficient way. For a standard user, it should be sufficient to access the endpoints *SPHN Connector API* and *SPHN Connector API docs*. The other endpoints are mainly meant for debugging purposes and admin users. Clicking on the services' names will redirect you to the corresponding service.

SPHN Connector API

Main entry point for interacting with the SPHN Connector. This endpoint contains all the developed functionalities.

SPHN Connector API docs

Documentation page where all the functionalities of the SPHN Connector API are described.

MinIO

Minio is the object storage of the SPHN Connector. Created projects are stored there as buckets. Inside each bucket we store the configuration files for the project (e.g. schemas, external terminologies) as well as patient data, and other internal files.

pgAdmin

pgAdmin offers a handy interface to access data stored in the internal PostgreSQL database. Admin users can use it to visualize data stored for internal processes, while all users can use it to visualize patient data uploaded.

Apache Airflow

Apache Airflow is the scheduling/pipeline service that is used by the Connector. In the case of unexpected errors of the pipeline, Airflow logs could be useful to for debugging purposes.

Grafana

Grafana is a visualization tool used by the Connector. It offers predefined views for a smoother analyses of the data produced by the Connector.

Documentation

The latest documentation regarding the SPHN Connector usage and installation can be found at the following links

SPHN Connector User Guide

Detailed explanation on how to use the SPHN Connector.

SPHN Connector Installation/Requirements Guide

Reference for the setup and the requirements of the SPHN Connector.

SPHN Readthedocs

SPHN Readthedocs for the SPHN Connector tool.

Gitlab Repository

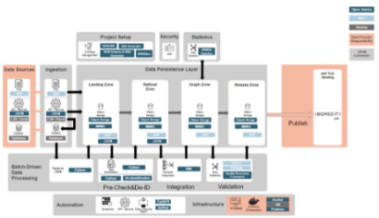
Gitlab repository of the SPHN Connector.

Release Page

Release page of the SPHN Connector with all the released versions.

Architecture

The SPHN Connector has three main ingestion interfaces: RDF, JSON, and database. Patient data goes through three main steps: *Pre-Checks & De-Identification*, *Integration*, and *Validation*. The diagram shows the main components of the SPHN Architecture.



Ingestion process

In addition to the standard ingestion interfaces, the SPHN Connector allows CSV and Excel ingestion. The following diagram gives an overview on the available ingestion interfaces by listing all the steps to execute for a successful ingestion.

Contact/Support

In case of any issues/questions regarding the SPHN Connector, the support team can be contacted either via email sphn-connector-team@lists.sib.swiss or on the *sphn-connector-issues* channel on Slack.

Preparing the SPHN Connector for usage

In order to use the SPHN Connector the users have to be set up. There are two types of users, Administrator and End User/Data Engineer.

The function of the administrator is to:

- Create End Users
- Reset Passwords
- Delete End Users

The End User can:

- Setup and maintain projects
- Ingest data
- Start/stop and monitor the processes
- Retrieve the RDF files

Although the administrator user could also be used at the moment to execute the end users functions, this is not foreseen and an end user should be set up specifically to manage a project and process the data.

Setting up the Admin User

At the moment the admin user has to be specified during the installation process (see installation manual) and there can only be one admin user for the moment. This user holds admin privileges for the API and acts therefore as the Connector admin.

Setting up End Users

End users can only be set up, reseted or deleted by the admin user. For this the admin user will use the functions provided in the API and described below in section User management. The end users, however, can reset their passwords themselves as explained in section Resetting the Password.

Authentication/Authorization

Authentication mechanism is OAuth2 which uses JWT tokens. This authentication is applied to all endpoints activating SPHN Connector's functionalities. It still authenticates with credentials but on the backend, a JWT token with expiration time is generated. In the UI, the user has to login by specifying the credentials.

To access the endpoints with a code-based approach, the user needs to derive a JWT token from the "/token" endpoint and pass it as headers in the endpoint request:

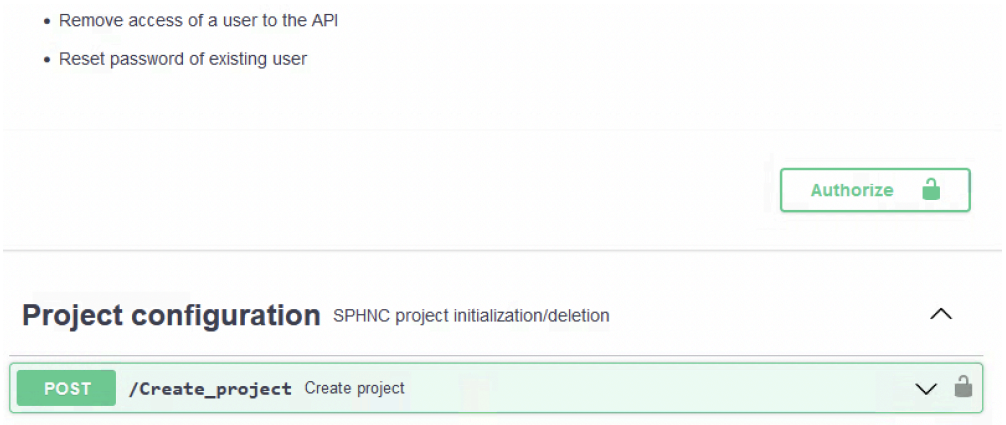
```
token = client.post("/token", data={"username": ”api_user”, "password": “api_password”})

headers = {'Authorization': 'Bearer ' + token.json()['access_token']}

response = client.post("/sphn-connector-endpoint", params={…}, …, headers=headers)
```

For an example on retrieving the header token see also the example from the user testing here: https://git.dcc.sib.swiss/hospfair/sphn-connector-user-testing/-/blob/main/src/config.py?ref_type=heads.

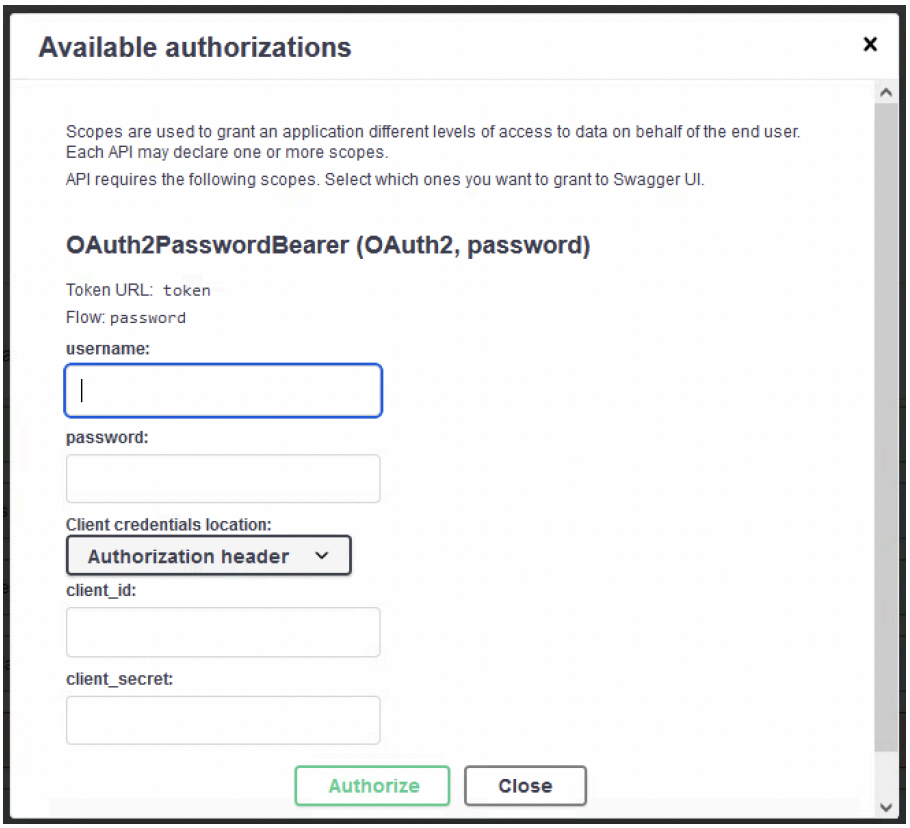
If you want to try it interactively in the FastAPI endpoint enter the following URL into your browser:

https://{your-server}:1443/api

Then click the “Authorize” button on the right (see below).

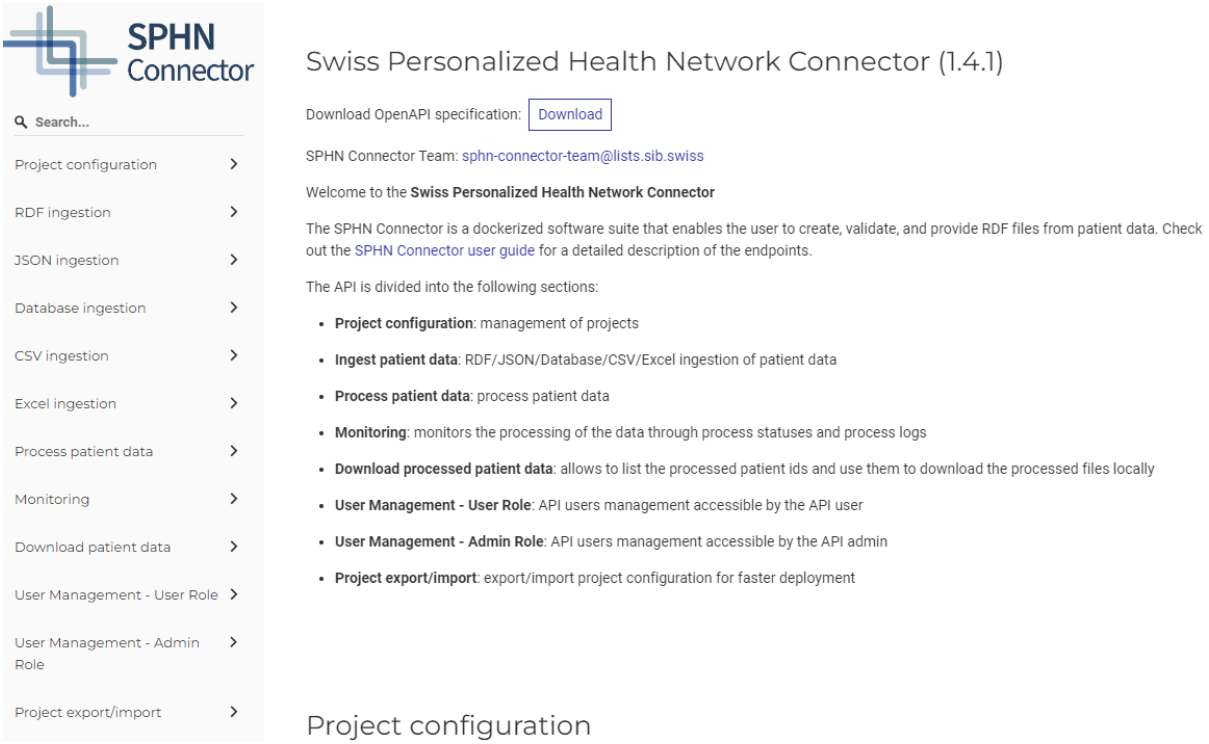


Now you can input the **username** and **password** in the fields below (client_id and client_secret are not used currently, and must be left blank).



Once you have logged in you can start using the API endpoints in the next sections.

This document discusses the available endpoints in detail. If you like you'll also find an overview of the endpoints in your locally running instance via: https://{your-server}/api/docs .



User management

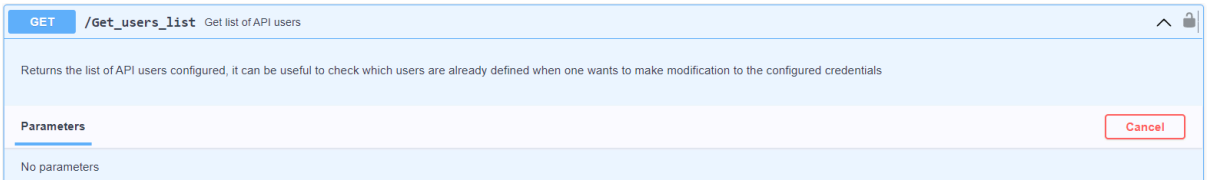
These endpoints need administrator credentials and are foreseen to create specific end users that then can define projects, ingest and process data.

The administrator has the following restricted endpoints that can also be found at the end of the interactive FastAPI user interface:

Get a list of defined users

`/Get_users_list ()`

List all users who are defined and can access the API. The response is divided by user types, i.e. grouped under admin_users, standard_users, ingestion_users. The returned users depend on the user type of the user triggering the endpoint, meaning that for example 'Standard' users can extract a list of users only of type 'Standard' and 'Ingestion', but not 'Admin' type users.



Create a new end user with a default password

`/Create_new_user (user: str = Form(...), password: str = Form(...), user_type: str ['Admin', 'Standard', 'Ingestion'])`

Create a new user by passing username, default password, and type of the user. By default the type is 'Standard', but it can also be set to 'Ingestion' type.

- Admin: user with full permissions on all endpoints, project's schema on database, full permissions on MinIO storage. Can access internal tables in PostgreSQL database.
- Standard: user with all permissions except admin level permissions. On MinIO this user has read permissions on all the content, but write/delete permissions only on prefix Input/batch_ingestion.
- Ingestion: user with permissions only related to data ingestion features, meaning endpoints /Ingest_rdf, /Ingest_json, /Ingest_database, /Ingest_csv, /Ingest_excel, and permissions on the tables in the project's schema on the database. On MinIO the user has read/write/delete permissions only on prefix Input/batch_ingestion.

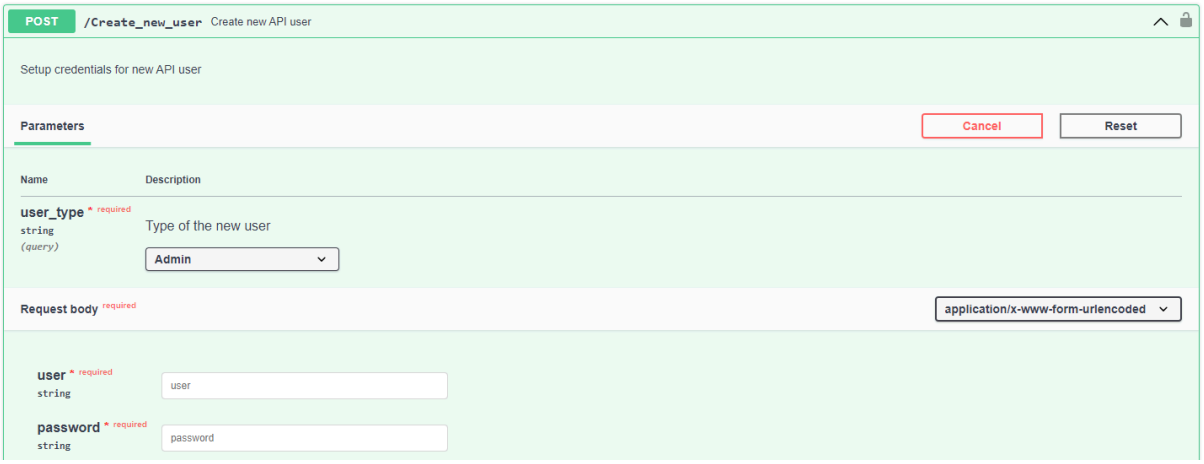
Note: *It is recommended that a user changes his password the first time using the Resetting the Password function described below.*

Note*: this endpoint is available to admin users only. Only the super-user can create admin users. Created admin users can create lower type users.*

Note*: usernames are allowed to contains only a set of characters defined by regex expression:* `^[a-zA-Z0-9\._-]*$`. *This means that only lower/upper case letters, digits, dots, underscores, and hyphens are allowed.*

Note*: when a new API user is created, a corresponding user created for the MinIO microservice with the permissions correlated to its type.*

***Note**: the password must be at least 8 characters long.*

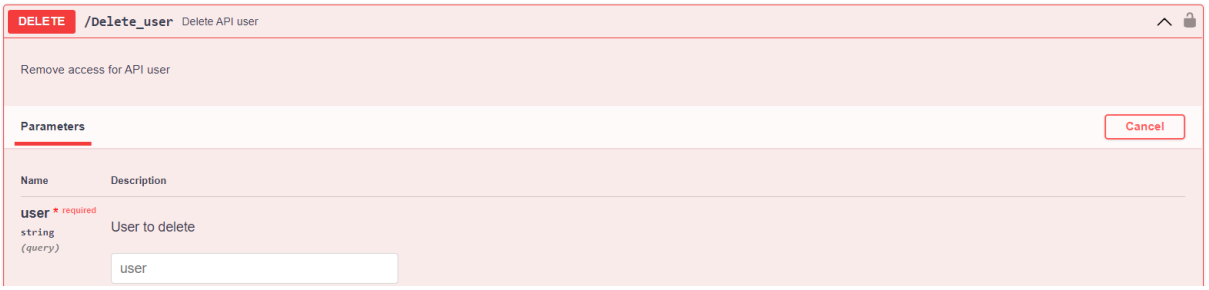


Note: all endpoints that request a password accept input parameters as payload (application/x-www-form-urlencoded). Programmatically it looks like this (Python): `response = client.post("/Create_new_user", data={"user": user,"password": password}, params={"user_type": user_type}, headers=headers)`

Delete a user

`/Delete_user (user: str)`

Deleting a user will not remove the user name from any log entry but it removes the user from the authorized user table.

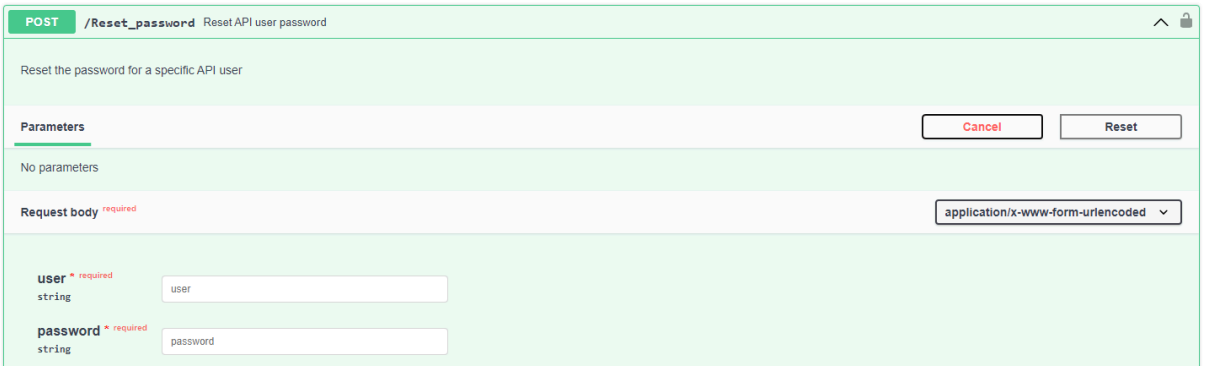


***Note**: this endpoint is available to admin users only. Only the super-user can delete created admin users.*

Reset a password for a user

`/Reset_password (user: str = Form(...), new_password: str = Form(...))`

In case a user doesn't remember his password anymore the administrator can force reset the password of a user by passing a new password.



***Note**: the password must be at least 8 characters long.*

***Note**: this endpoint is available to admin users only. Admin users can reset lower type users or their own password.*

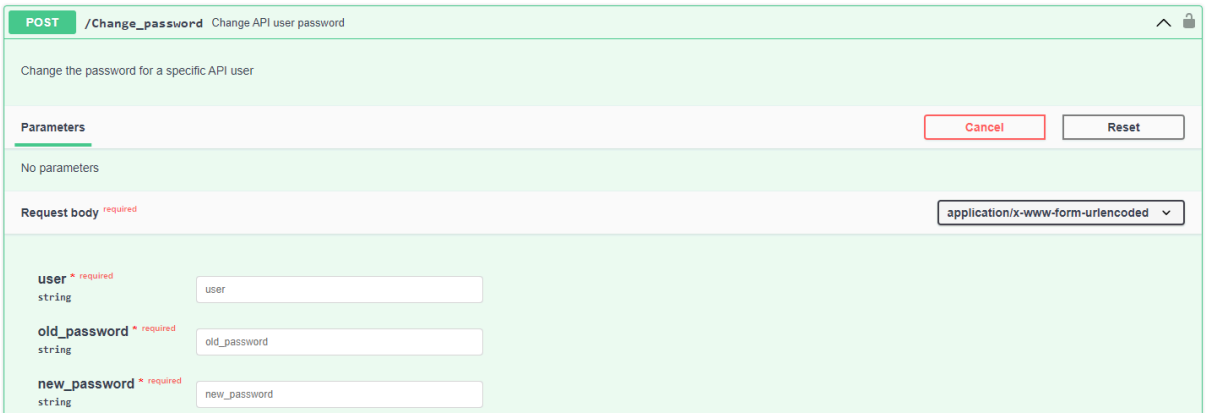
Changing the Password

This function is available for every user of the SPHN Connector that knows his/her password.

`/Change_password (user: str = Form(...), old_password: str = Form(...), new_password: str = Form(...))`

If the user doesn't remember his password, he has to ask the administrator to hard-reset the password.

***Note:** Currently the password functionality implemented is rudimental but will be reviewed for the final release. No password policy is available for the moment.*



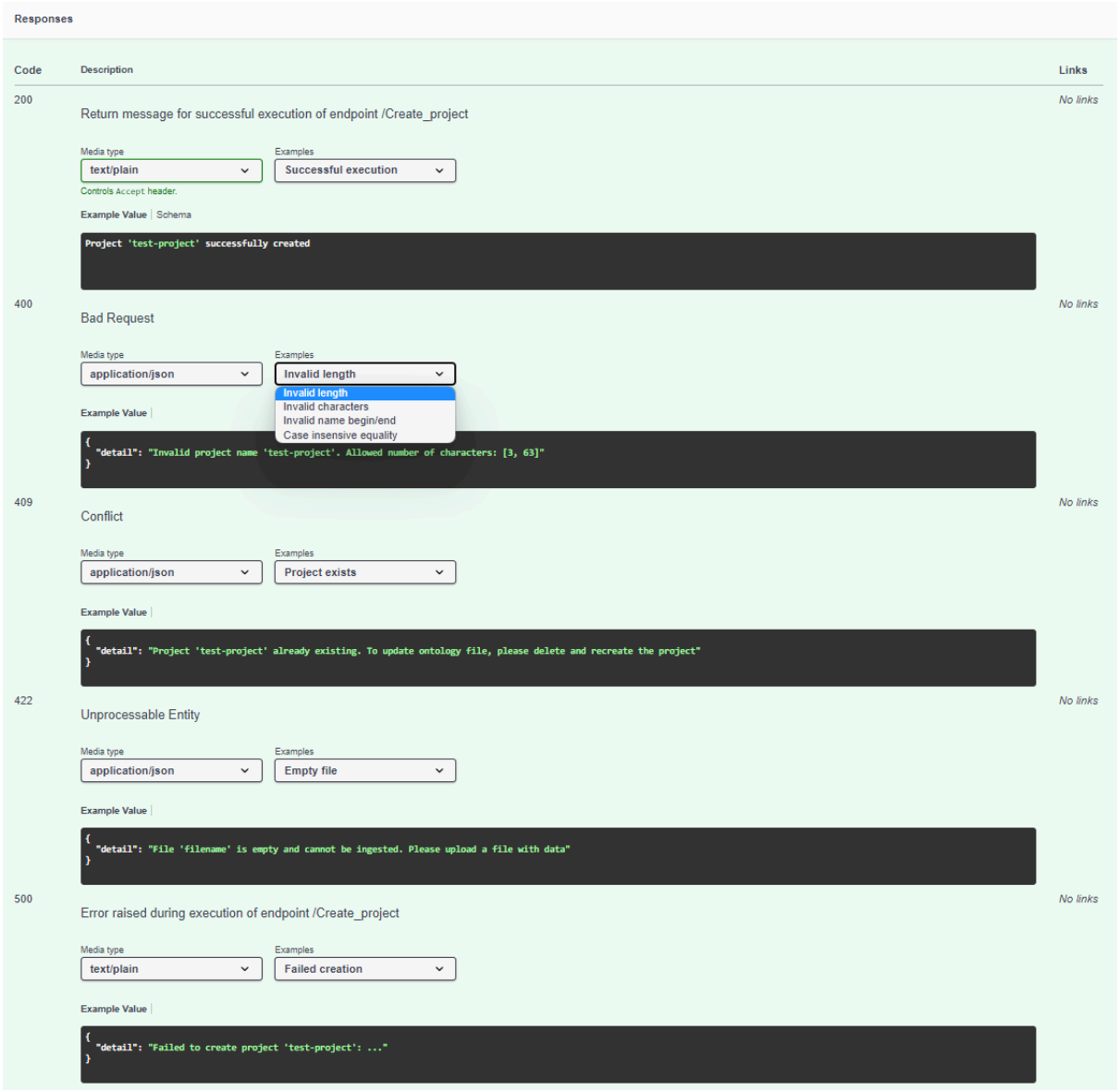
***Note**: the password must be at least 8 characters long.*

***Note**: the super-user can change the password of all users. All other users can only change their own password. Admin users can change other non-admin user passwords with the [Resetting the Password] endpoint.*

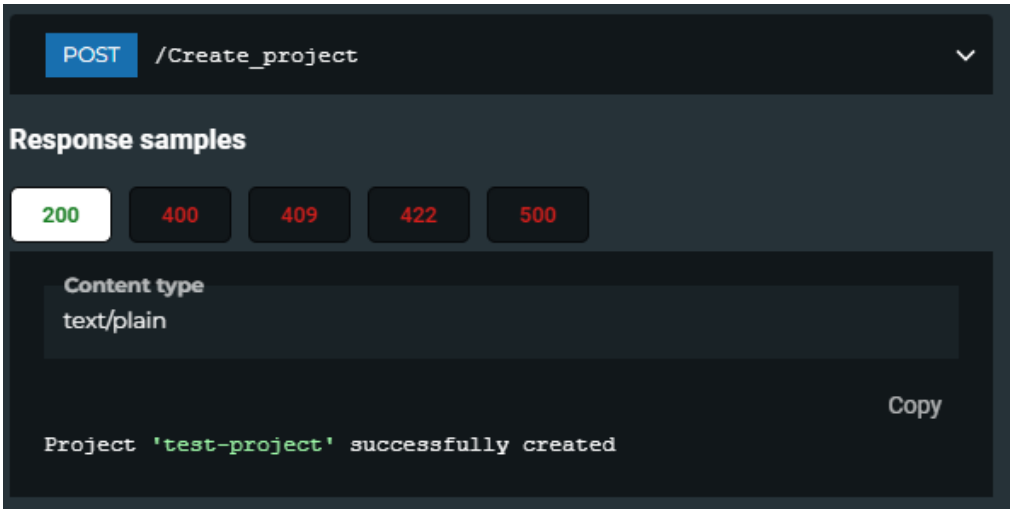
Response Model

Each endpoint of the SPHN Connector API has its own response model. The latter is reported in the `Responses` section below each endpoint. It can also be accessed via the /api/docs. There we tried to summarize the expected HTTP return codes and we listed all the possible responses by defining a set of examples for each response type.

On API user interface:



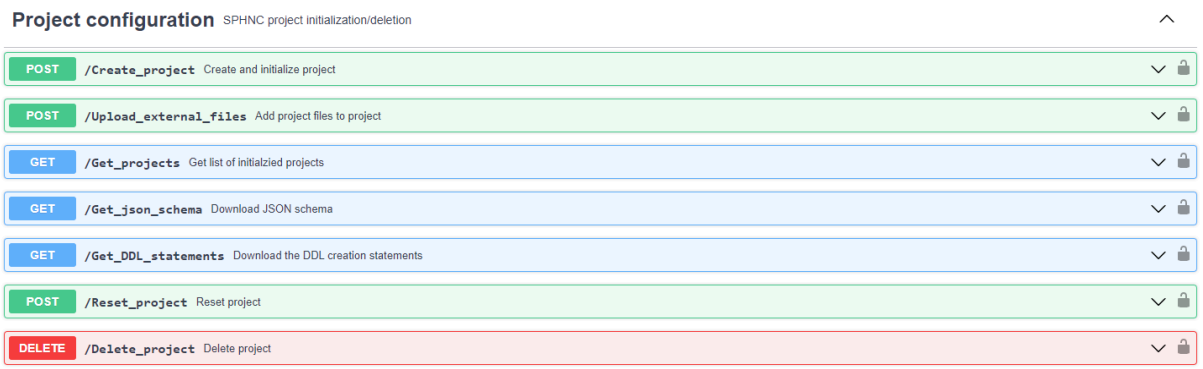
On API documentation page:



Configure a Project

Once a user has been granted access to the API, he can now use all the endpoints of the SPHN Connector, including ingesting data. This section describes all the functions and steps necessary to set up a project. This includes generating a JSON schema that defines the format of a JSON file that can be ingested but also a Postgres DDL with the corresponding database tables used to ingest tabular data.

The following steps are available and should be executed in the sequence as they are listed below:



Create Project

As the SPHN Connector pipeline is based on projects, this is the first thing you have to do in order to use the SPHN Connector. In order to create the project you need to define a project name or code and the RDF schema file you want to upload with it.

```
/Create_project (project: str, sphn_schema: UploadFile,
[project_specific_schema: UploadFile], [data_provider_id: str],
[exception_file: UploadFile], [output_format: [Turtle, NQuads, Trig]
= Turtle] = Turtle], [compression: [True, False] = False])
```

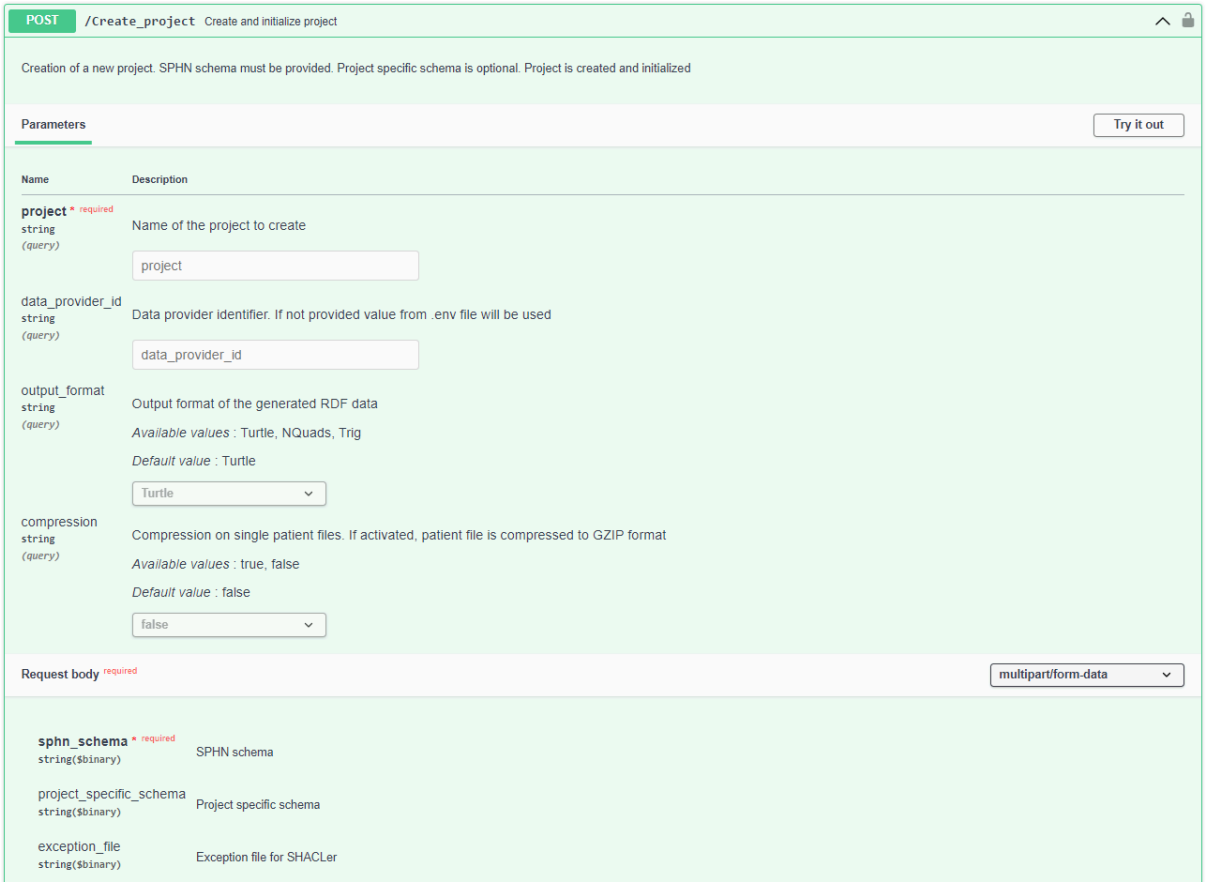
Name of the project and SPHN RDF schema are mandatory parameters to create a new project. In addition, the project specific schema can be uploaded. If that's the case, the latter is used as a project specific schema and the SPHN schema is used as an external schema. If that's not the case, the SPHN schema becomes the project specific schema for the created project. Exception file for Shacler execution can also be uploaded:

Exception File	Under RDF Schema 2022.2 there is an issue with the Shacler and a special exception file is required to correctly validate the RDF file in the QC. The exception file for the SPHN schema can be found here: https://git.dcc.sib.swiss/sphn-semantic-framework/sphn-shacl-generator/-/blob/master/2022-2_exceptions.json **Note**: this might also apply if a project specific schema creates a sub-concept from MeasurementMethod with a different ValueSet, similar to BloodPressure. And a project specific exception file would have to be uploaded. However, from RDF Schema 2023.1 this should not be the case anymore.

Via this endpoint, all the files needed for project initialization and creation are uploaded. When the endpoint is triggered, the files are uploaded and the Shacl file generation is triggered, such as the RML mapping, JSON schema, and database schema generation (JSON schema, RML mapping and DDL generation).

The SPHN Connector distinguishes between two different kinds of schemas: The SPHN schema and the project specific schema. To create a project you need to provide at least the SPHN schema. The designated upload field for this schema is named *"sphn_schema"*. In case you want to create a project that extends on the SPHN schema you can provide an additional schema and upload it via the *"project_specific_schema"* field. Note: while it is sufficient to create a project without a project specific schema you always need to provide the SPHN schema so successfully create a project. In case you are using a schema with known issues (like the SPHN schema version 2022.1) you can furthermore provide an exception file for the Shacler to ensure a correct validation of your data.

***Note**: allowed characters for a project name are only characters, numbers, (dot) . and (hyphen) -*



Note: when a project specific schema is provided, the Connector triggers a compatibility check of the loaded schemas. It checks in the following order:

- The project schema contains the SPHN schema IRI in the list of owl:imports.
- The imported SPHN schema IRI matches the owl:versionIRI defined in the uploaded SPHN schema.

If one of those conditions are not met, then the project creation is stopped and an exception is raised. Therefore, make sure that the project and SPHN schema are compatible when you load them into the SPHN Connector.

Note about *data_provider_id*: when creating/importing a project the value of data_provider_id can be specified. If that's the case, the value defined in the `.env` file will be overwritten. All the loaded data should then be defined in relation to the provided value (e.g. database patient data). If no value is passed, the default value from the `.env` file is used.

The SPHN Connector generates turtle patient data by default. Nevertheless, when creating a project it is possible to specify the output format and the compression of the generated patient data via parameters **output_format** and **compression**. All the data converted for that specific project will use the format defined at project's creation. The available formats are: **Turtle, NQuads, Trig**. If compression is enabled, the single patient files content will be compressed to GZIP. NQuads and Trig output data use the `sphn:SubjectPseudoIdentifier` resource to identify the graph, e.g. *resource:DATA-PROVIDER-ID-sphn-SubjectPseudoIdentifier-c8a70639-eb11-47b3-a7a9 {...}*. In case this information is not present in the data it will use the data provider ID and Connector patient ID to generate a URI.

Upload External Files

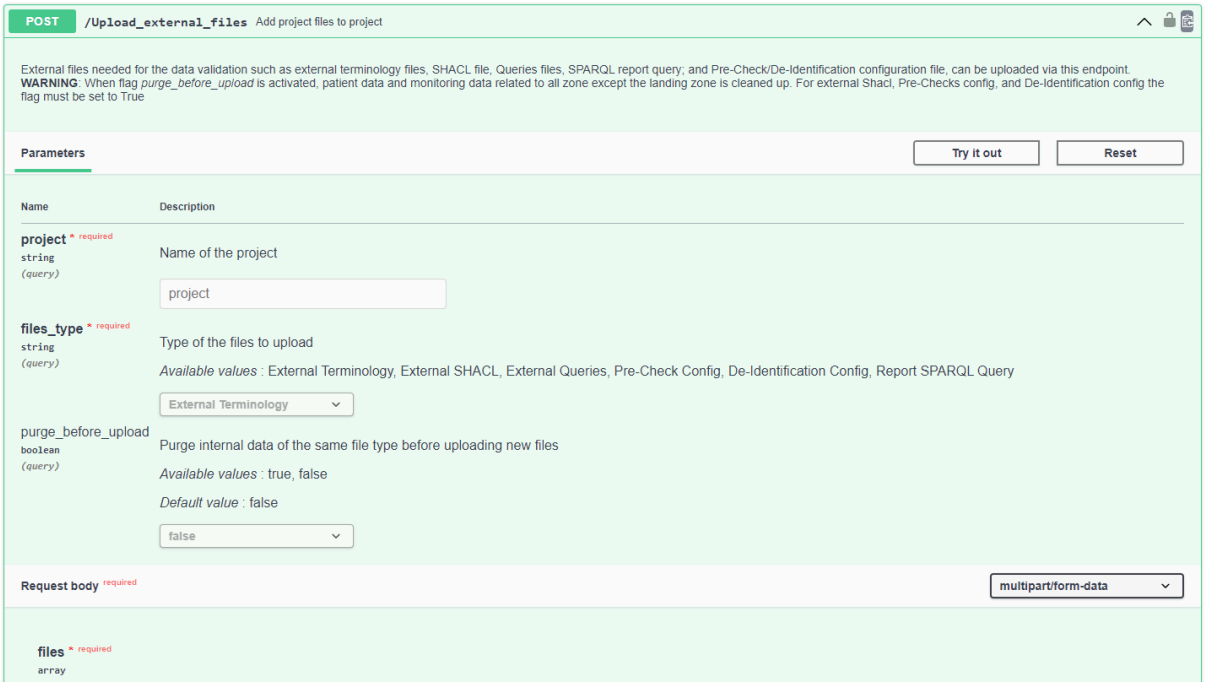
Other types of configuration files have to be uploaded under the upload external files endpoint. The project is initialized at the project creation endpoint, therefore there is no re-initialization logic when new files are uploaded. Though, the endpoint has a parameter `purge_before_upload` which cleans up the previously ingested configuration files for the same type of the files which are currently being uploaded. This flag can be used for example if a new terminology version is released and the user needs to update the external terminologies. For external SHACL, Pre-Check configuration file, and De-Identification configuration file, the user is forced to set the flag to `True`. Another direct consequence of enabling the flag, is that all the patient data and related logging is removed from the Connector for that project, except the data present in the landing zone. That means that the user does not have to reingest all the data, but only reprocess it. It might be possible that you encounter an error when uploading large files. In case you experience an "Bad Gateway - 502" please refer to the troubleshooting guide. For version > 1.2.0-RC2 this issue should not occur since the methodology of validation the rdf files have changed. In case you still encounter this issue please communicate this issue to the development team.

```
/Upload_external_files (project: str, files_type: ["External Terminology", "External SHACL", "External Queries", "Pre-Check Config", "De-Identification Config", "Report SPARQL Query"], files: List[UploadFile], purge_before_upload: bool = False)
```

Following file types can be uploaded via this endpoint:

File Type	Description
External Terminology	All the taxonomies that are used for validating the RDF files are to be uploaded here (SnomedCT, Loinc, ICD-10-GM, CHOP, ATC, UCUM and

	any project specific Terminology or any other future taxonomy that is not yet defined.) The RDF schemas are best retrieved from the BiomedIT Portal (https://portal.dcc.sib.swiss).
External Shacl file	The SPHN Connector will generate the Shacl files from the schema uploaded automatically using the SPHN Shacler tool. If you upload a Shacl file it will replace the ones generated by the Shacler. This means that you have to make sure the file you upload here contains all Shacl rules, also the SPHN ones.
External SparQL script	Any SparQL script you like to execute during the statistics step.
Pre-Check Config	JSON file for Pre-Check configuration (Pre-Checks configuration)
De-Identification Config	JSON file for De-Identification configuration (De-Identification configuration)
Report SPARQL Query	External SPARQL query that is executed against the validation report of a patient and is displayed in the patient logs of the validation step. It expects a .rq file with a valid SPARQL query. If not provided or wrongly provided, it defaults to a pre-defined query.

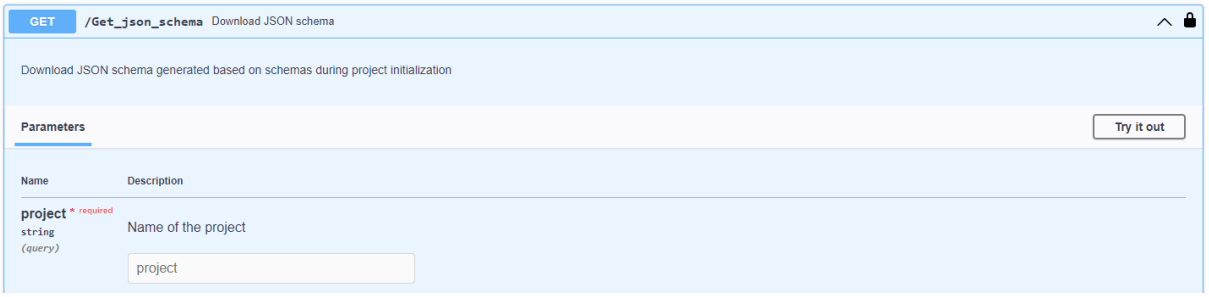


Get JSON Schema

In order to prepare the data in JSON format the JSON schema generated by SPHN Connector can be downloaded with this endpoint:

`/Get_JSON_schema (project: str)`

***Note**: The JSON schema could also be used to compare different versions during a migration to highlight changes in the schema.*



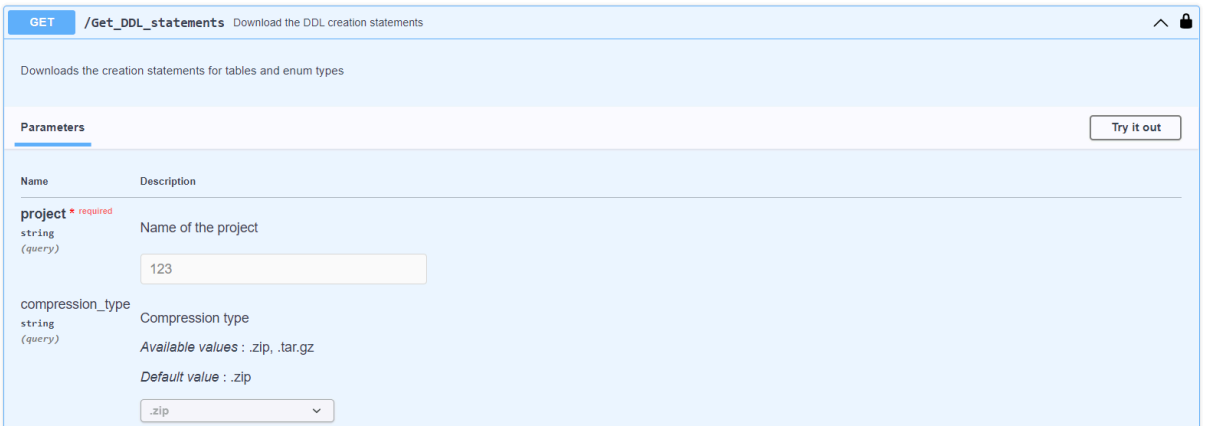
Get DDL Statements

In order to prepare the data in tabular format the actual tables generated by the project initialisation could be viewed or the DDL statements generated by SPHN Connector for the table creation can be downloaded with this endpoint:

```
/Get_DDL_statements (project: str, [compression_type:
[".zip",".tar.gz"]])
```

Since release 0.3.0 it provides the option to download it either as a .zip or .tar.gz file (default is .zip). See also General Remarks, and the get DDL statement always returns two files.

***Note**: The DDL statements could also be used to compare different versions during a migration to highlight changes in the table structures.*



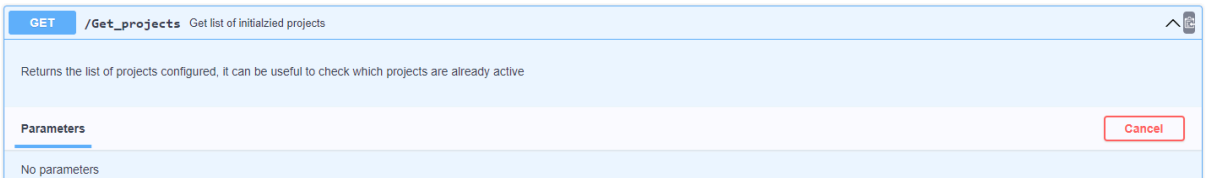
Project Administration Functions

In order to manage the projects that are defined in the SPHN Connector the following options are available:

Get Projects

In order to check which projects are already setup and with which files, you can use the following endpoint:

`/Get_projects ()`



The `/Get_projects` endpoint returns details about the project and its configuration files. It gives an overview on the files uploaded, their size, and some additional details.

Note: after uploading the external terminologies, the user can use this endpoint to compare the `versionIRI` of the uploaded terminologies with the required `owl:imports` of the RDF schema.


```
{
  "projects": [
    {
      "project-name": "test-project",
      "schema-name": "test-project",
      "data-provider-id": "DATA-PROVIDER-ID",
      "created-by": "user",
      "is_initialized": true,
      "uploaded-files": {
        "project_specific_schema": [
          {
            "filename": "test-project_schema.ttl",
            "original_name": "sphn_rdf_schema_2024.2.ttl",
            "file_size": "610.59KB",
            "versionIRI": "https://biomedit.ch/rdf/sphn-schema/sphn/2024/2",
            "owl:imports": [
              "<http://purl.obolibrary.org/obo/eco/releases/2023-09-03/eco.owl>",

"<http://purl.obolibrary.org/obo/genepio/releases/2023-08-19/genepio.owl>",
              "<http://purl.obolibrary.org/obo/geno/releases/2023-10-08/geno.owl>",
              "<http://purl.obolibrary.org/obo/obi/2023-09-20/obi.owl>",
              "<http://purl.obolibrary.org/obo/so/2021-11-22/so.owl>",
              "<http://snomed.info/sct/900000000000207008/version/20231201>",
              "<http://www.ebi.ac.uk/efo/releases/v3.61.0/efo.owl>",
              "<https://biomedit.ch/rdf/sphn-resource/atc/2024/1>",
              "<https://biomedit.ch/rdf/sphn-resource/chop/2024/4>",
              "<https://biomedit.ch/rdf/sphn-resource/edam/1.25>",
              "<https://biomedit.ch/rdf/sphn-resource/emdn/2021-09-29/1>",
              "<https://biomedit.ch/rdf/sphn-resource/hgnc/20231215>",
              "<https://biomedit.ch/rdf/sphn-resource/icd-10-gm/2024/3>",
              "<https://biomedit.ch/rdf/sphn-resource/loinc/2.76/1>",
              "<https://biomedit.ch/rdf/sphn-resource/ucum/2024/1>",
```

```
"<https://www.orphadata.com/data/ontologies/ordo/last_version/ORDO_en_4.4.owl>"
            ]
          }
        ],
        "ext_shacls": [
          {
            "filename": "Shacler_test-project_shacl.ttl",
            "file_size": "568.74KB"
          }
        ],
        "ext_terminologies": [
          {
            "filename": "snomed-ct-CH-20231201.ttl",
            "file_size": "517.93MB",
            "versionIRI": "<http://snomed.info/sct/900000000000207008/version/20231201>"
          },
          {
            "filename": "sphn_atc_2024-2016-1.ttl",
            "file_size": "26.7MB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/atc/2024/1>"
          },
          {
            "filename": "sphn_chop_2024-2013-1.ttl",
            "file_size": "129.17MB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/chop/2024/4>"
          },
          {
            "filename": "sphn_eco_20230903-1.ttl",
            "file_size": "3.12MB",
            "versionIRI":
"<http://purl.obolibrary.org/obo/eco/releases/2023-09-03/eco.owl>"
          },
          {
            "filename": "sphn_edam_1.25-1.ttl",
            "file_size": "1.67MB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/edam/1.25>"
          },
          {
            "filename": "sphn_efo_3.61.0-1.ttl",
            "file_size": "94.79MB",
            "versionIRI": "<http://www.ebi.ac.uk/efo/releases/v3.61.0/efo.owl>"
          },
          {
            "filename": "sphn_emdn_2021-09-29-1.ttl",
            "file_size": "1.01MB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/emdn/2021-09-29/1>"
          },
          {
            "filename": "sphn_genepio_20230819-1.ttl",
            "file_size": "490.6KB",
            "versionIRI":
"<http://purl.obolibrary.org/obo/genepio/releases/2023-08-19/genepio.owl>"
          },
          {
            "filename": "sphn_geno_20231008-1.ttl",
```

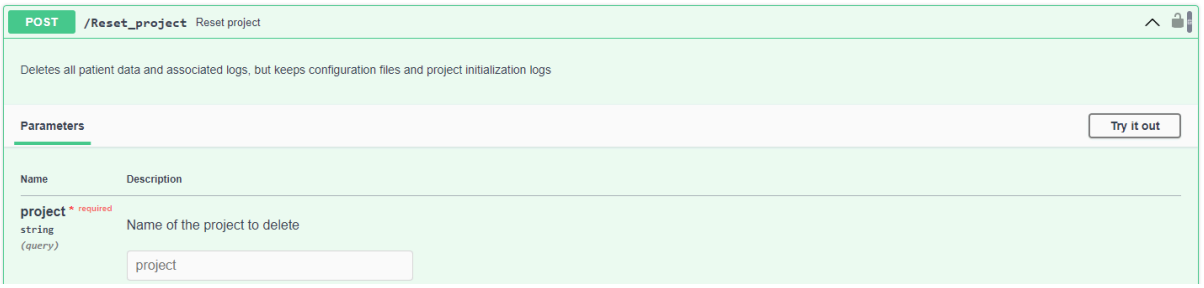
```
            "file_size": "378.96KB",
            "versionIRI":
"<http://purl.obolibrary.org/obo/geno/releases/2023-10-08/geno.owl>"
          },
          {
            "filename": "sphn_hgnc_20231215-1.ttl",
            "file_size": "23.13MB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/hgnc/20231215>"
          },
          {
            "filename": "sphn_icd-10-gm_2024-2013-1.ttl",
            "file_size": "113.24MB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/icd-10-gm/2024/3>"
          },
          {
            "filename": "sphn_loinc_2.76-1.ttl",
            "file_size": "35.96MB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/loinc/2.76/1>"
          },
          {
            "filename": "sphn_obi_20230920-1.ttl",
            "file_size": "4.54MB",
            "versionIRI": "<http://purl.obolibrary.org/obo/obi/2023-09-20/obi.owl>"
          },
          {
            "filename": "sphn_ordo_4.4-1.ttl",
            "file_size": "24.12MB",
            "versionIRI":
"<https://www.orphadata.com/data/ontologies/ordo/last_version/ORDO_en_4.4.owl>"
          },
          {
            "filename": "sphn_so_20211122-1.ttl",
            "file_size": "2.39MB",
            "versionIRI": "<http://purl.obolibrary.org/obo/so/2021-11-22/so.owl>"
          },
          {
            "filename": "sphn_ucum_2024-1.ttl",
            "file_size": "131.82KB",
            "versionIRI": "<https://biomedit.ch/rdf/sphn-resource/ucum/2024/1>"
          }
        ]
      }
    }
  ]
}
```


Reset Project

If you want to reset a project but still use the same configuration with the same schemas, taxonomies, Shacl and SparQL you can execute the following endpoint:

`/Reset_project (project: str)`

This will delete all the data that you have already ingested or are in processing status as well as the RDF files generated and the processing logs. It will not allow you to ingest other schema versions or external files. If you want to do this, you have to delete the project and start again from scratch.



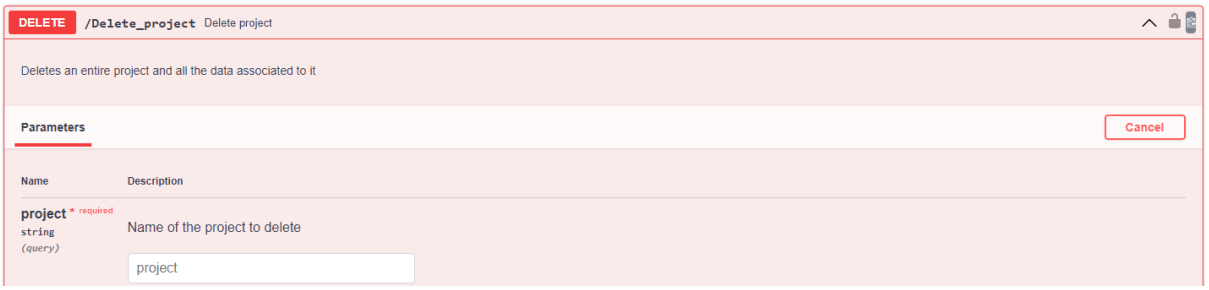
Delete Project

If you want to change the configuration of a project, i.e. load a newer version of the schema you can either create a new project or delete the existing project. In case you want to keep the project name/id you will have to use this endpoint to delete the complete project and start from zero.

`/Delete_project (project: str)`

***Note**: please make sure you download everything you need before you delete the project!*

***Note**: this functionality is available only to admin users and to the user who created the project in the first place.*



Preparing and ingesting data into the SPHN Connector

The SPHN Connector supports three types of patient data ingestion: RDF files ingestion, JSON files ingestion, and ingestion from database.

Note: re-ingestion of data for an existing patient, will cleanup the already processed data and the logging information for that patient.

Note: patient data (RDF, JSON, Database) can be ingested into the Connector for a specific project, only if no pipelines related to that project are currently running.

Preparing RDF Data

In order to be able to ingest data in RDF format the following requirements apply:

- The file has to be prepared based on SPHN RDF Schema version 2022.2 or later.
- The file should contain all concepts for a given patient in order for the validation to work correctly.
- Each patient should be prepared in an individual RDF turtle file in order to manage the processing of a given patient and to obtain the best possible validation result.

There are no requirements on how you generate the RDF file (i.e. which tools you use) except that the resulting file has to comply with above requirements.

Ingest RDF Data

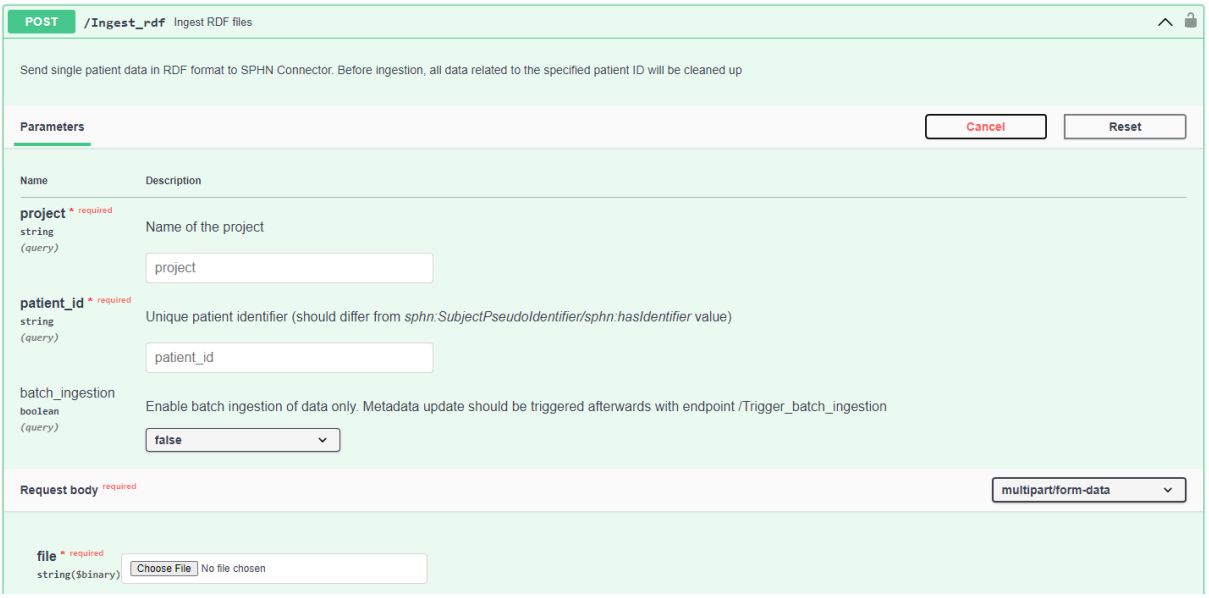
The ingestion of the RDF file has to be done patient by patient and can be done with the following endpoint:

```
/Ingest_rdf (project: str, patient_id: str, file: UploadFile, [batch_ingestion: boolean])
```

***Note**: Technically it is possible to ingest an RDF turtle file that contains more than one patient, however this has below two limitations and is **not** recommended.*

1. *You have to allocate a dummy patient_id when ingesting in order to manage the processing of this file.*
2. *The size of the RDF turtle file should not be too big, as this might cause issues with system resources, especially during validation. The exact size depends on the RAM you allocated to your SPHN Connector server.*

At this stage, there is already a validation step which is performed on RDF files. We assume that the uploaded files with ".ttl" extension are turtle files, and then we load the files into a graph. If the uploaded file cannot be loaded into a graph, we consider it as invalid and we reject it



Batch ingestion: the optional flag `batch_ingestion` allows uploading data in a faster way because it focuses only on the upload of the files into temporary prefix `Input/batch_ingestion` in MinIO and it does not take care of the metadata update. The flag can be activated when there is a large amount of files to ingest into the SPHN Connector. To make those files available for the processing, the user must then trigger the `/Trigger_batch_ingestion` endpoint which takes care of the metadata update and moves the files in the expected landing zone location.

Note: the `patient_id` is an internal ID and should not take the value of `sphn:SubjectPseudoIdentifier/sphn:hasIdentifier`. It is used internally and for the generated file name.

Preparing Data in JSON

In order to prepare data in JSON format it is recommended to download the JSON schema (see Get JSON Schema above). The JSON schema defines how the JSON has to look like and can also be used to validate the JSON file prior to uploading into the SPHN Connector.

The following restriction apply:

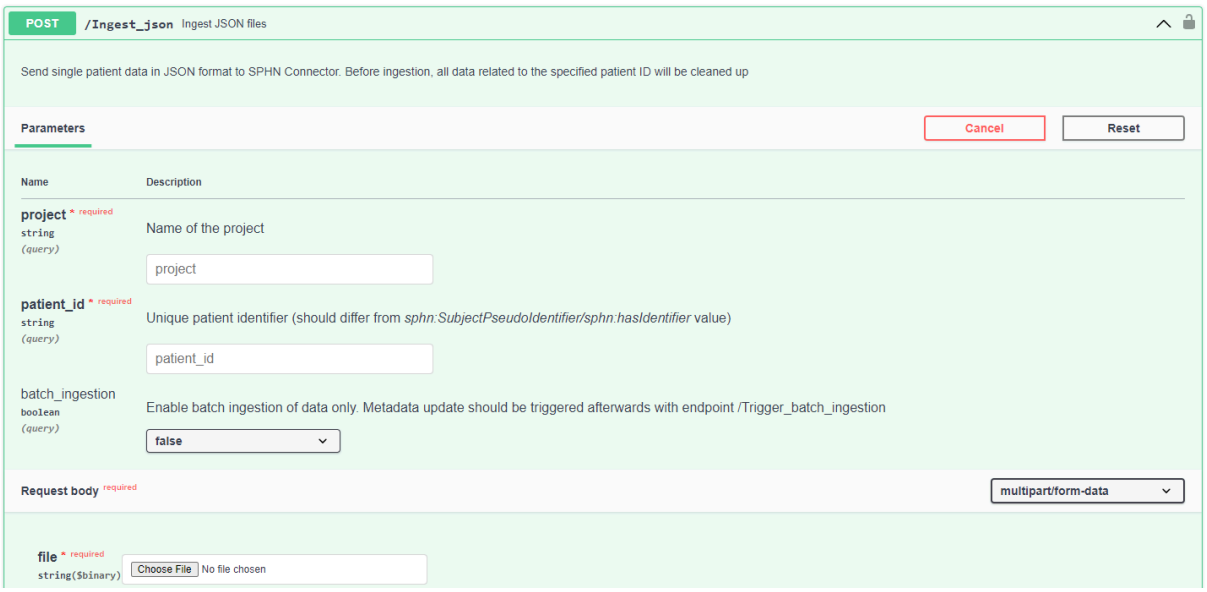
- One JSON file should contain only one patient.
- The JSON file should contain all concepts for a patient.
- The JSON file has to comply with the JSON schema and the translation into the RDF file or its validation will fail.

Note: the JSON data doesn't have to be in the best readable format, but it has to validate against the JSON schema. For example, if we have properties that are defined as an array, we could explode those objects by repeating the core concepts multiple times and only changing the array object properties. Edge cases are documented here: **Json schema, RML mapping and DDL generation**. All JSON files uploaded into the SPHN Connector (holds also for exception files via /Upload_external_files), go through a first validation step, where it is checked that the content of the files is real JSON data. If it's not, the file will be rejected.

Ingest JSON Data

The ingestion of the JSON file has to be done patient by patient and can be done with the following endpoint:

`/Ingest_json (project: str, patient_id: str, file: UploadFile, [batch_ingestion: boolean])`



Batch ingestion: the optional flag `batch_ingestion` allows uploading data in a faster way because it focuses only on the upload of the files into temporary prefix `Input/batch_ingestion` in MinIO and it does not take care of the metadata update. The flag can be activated when there is a large amount of files to ingest into the SPHN Connector. To make those files available for the processing, the user must then trigger the `/Trigger_batch_ingestion` endpoint which takes care of the metadata update and moves the files in the expected landing zone location. It is **highly recommended** to enable batch ingestion for ingestion of large numbers of patients.

Note: the `patient_id` is an internal ID and should not take the value of `sphn:SubjectPseudoIdentifier/sphn:hasIdentifier`. It is used internally and for the generated file name.

Preparing Data in Tabular Format

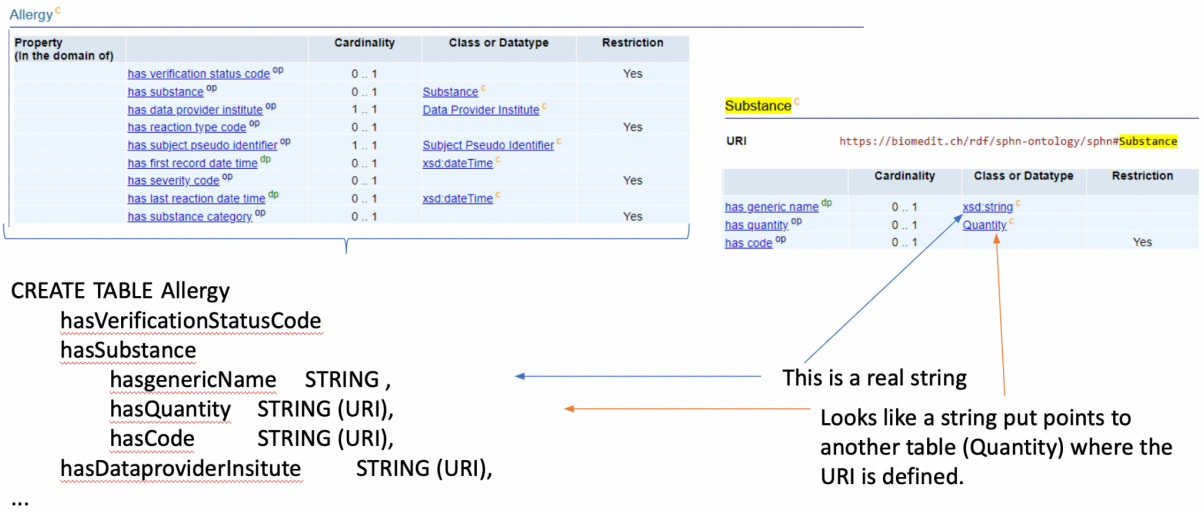
In order to prepare the data in tabular format there are the following options to analyze the necessary structure:

1. Download the DDL statements that were used to create the database in the SPHN Connector for the given project. This can also help in a migration from an older to a new schema by comparing the two DDL statements.
2. Use an ETL tool that can read the postgres database schema and present it in a useful way.
3. Connect to pgAdmin in the SPHN Connector and review the target tables, which you will have to fill in.
   This can be achieved by connecting to the URL https://{full-qualified-server-name}:1443/pgadmin (note that you need the login credentials as specified in the installation). The first time you connect to the database you have to register the server in pgAdmin (see Installation guide).

If you have a data mart that will be used to collect all the data that needs to be ingested into the SPHN Connector it is suggested that the structure is similar to the one of the postgres database in the SPHN Connector.

The tables in the postgres database are grouped by schema, where the schema name corresponds to the project you defined in the SPHN Connector.

Each concept that can be linked to a patient will have its own table including all the properties of the referenced concepts as shown in the following example (based on 2023.2 schema):



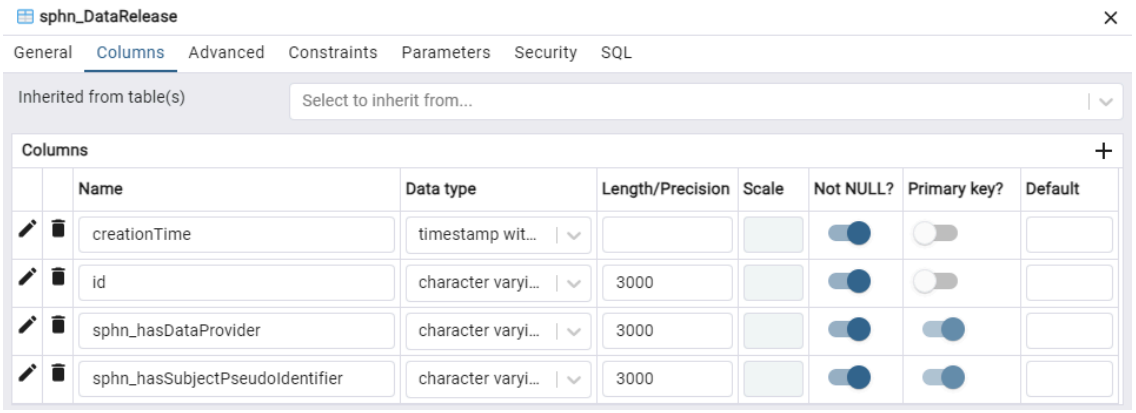
It is also important to note that each property, which has an SPHN defined value set (not external taxonomies), will have a dedicated type. This type consists of all the possible values that the property can have. These values are defined including the full IRI as shown in the following example:

```
CREATE TYPE "test-project"."sphn_BilledDiagnosis__sphn_hasRank__iri_type" AS ENUM
('https://biomedit.ch/rdf/sphn-schema/sphn/individual#Complementary',
'https://biomedit.ch/rdf/sphn-schema/sphn/individual#Principal',
'https://biomedit.ch/rdf/sphn-schema/sphn/individual#Secondary');
```

This means that possible data to be ingested into column `sphn_BilledDiagnosis_sphn_hasRank_iri` in the table

"{schema}"."sphn_BilledDiagnosis" can only contain one of the five values listed in the enum of the type.

Each database schema has a table named sphn_DataRelease. The purpose of this table is to release patient data and make it available for the ingestion into the SPHN Connector.



After all the data for a specific patient has been uploaded into the SPHN tables, then the user reports this patient into the sphn_DataRelease table and therefore makes it available for download.

Note: the value of the column sphn_hasSubjectPseudoIdentifier should be defined by a unique identifier of the patient which should differ from the value of sphn:SubjectPseudoIdentifier/sphn:hasIdentifier. The reason is that this identifier is used internally by the Connector to process the files and it will also be present in the generated RDF filename. As the sphn:SubjectPseudoIdentifier/sphn:hasIdentifier could potentially be identifying data, it's better to define a random identifier. The value defined in the table sphn_DataRelease must then be used to fill the corresponding columns sphn_hasSubjectPseudoIdentifier and the other core concept tables, and the id field in the sphn_SubjectPseudoIdentifier table.

Note: in order to preserve the naming of the columns in PostgreSQL database as they are defined in the schema, when inserting the data double quotes must be used around table names and column names. For example: `INSERT INTO "test-project"."sphn_Allergy" ("sphn_hasSubjectPseudoIdentifier") VALUES ('patient_1' );`

Note: edge cases are documented here.

Note: quantities of type xsd:gDay, xsd:gMonth, xsd:gYear are inserted in the database as integer values. For example, for the concept sphn:BirthDate the column sphn_hasDay takes integer values which should be between 1 and 31. The SPHN Connector then internally takes care of the conversion to the matching pattern, in the example ---01 to ---31.

Ingest Tabular Data

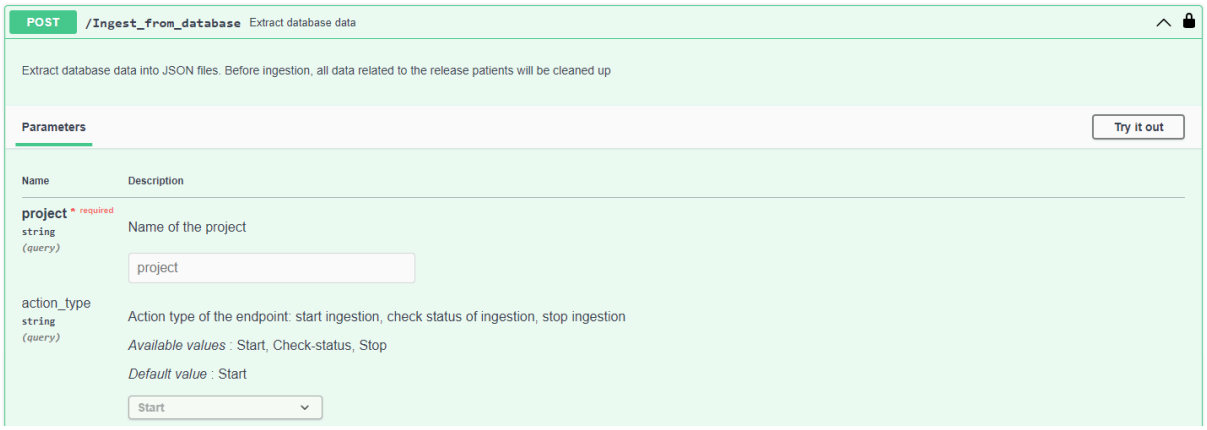
The ingestion of database data is triggered by specifying the project via the endpoint:

/Ingest_from_database (project: str, [action_type:['Start', 'Check-status', 'Stop'] = 'Start')

When the endpoint `/Ingest_from_database` is triggered, we look at the content of the `sphn_DataRelease` table (see above) and we extract the patient and data provider identifiers which are then used as conditions to pull the patient data from all the other concept tables. The process of pulling the data from the database and converting it into JSON format is executed by a DAG in Airflow, therefore it can be monitored from the Airflow UI. This is an asynchronous endpoint, meaning that the endpoint returns directly to the user the message that the extraction has been triggered. Pipelines cannot be started until the extraction is terminated. The user can check on Airflow or they can set the `action_type` to `'Check-status'` to see if the pipeline is still running.

Once a patient has been downloaded and successfully moved to the next stage, that patient will be marked as processed in the landing zone. That means that a re-triggering of the `/Ingest_from_database` with the same data in the `sphn_DataRelease` table won't pull any new data. Let's assume a user needs to update some data for a patient. In order to be able to pull the data for an already processed patient again, the user has to update the `creationTime` column in the `sphn_DataRelease` table and define it with the correct new release time (ideally also the id field should be updated to match the creation time).

Note: `/Ingest_from_database` first checks data availability on the `sphn_DataRelease` table. That means that it looks for released patients with the project name passed to the endpoint, and with column `sphn_hasDataProvider` values that matches the value of `DATA_PROVIDER_ID` defined in the `.env` file (or specified at project creation). It's important that what's defined in the `.env` file is also reported into the SPHN concept tables in order to be consistent.



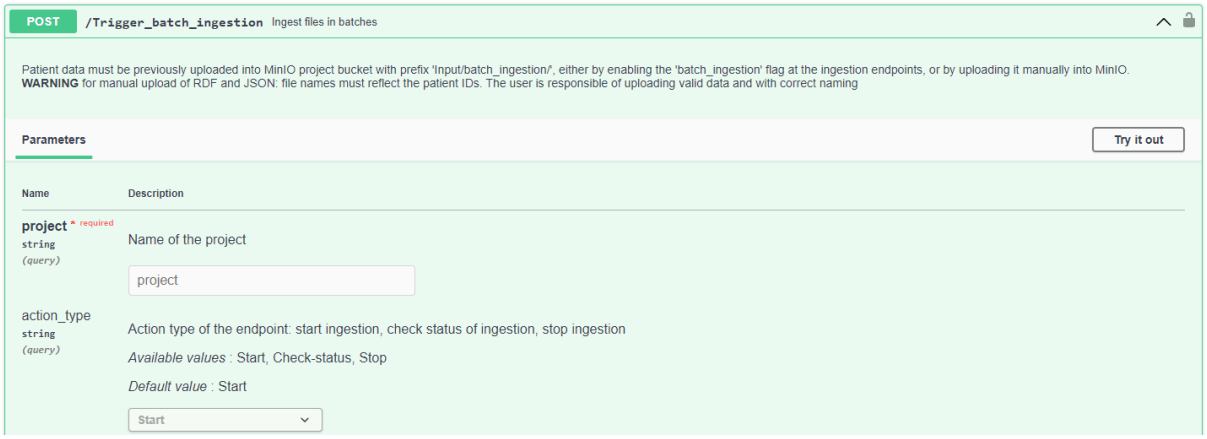
You can find a more complex example of a tabula ingestion in the appendix.

As described above, the endpoint is asynchronous. When called, the extraction of the database data is triggered in an Airflow DAG. When there is a large amount of database data that needs to be ingested into the SPHN Connector, we have to take into account some computation time. During the extraction, the user is not allowed to trigger the Connector pipeline steps. In case there is the need to stop the extraction for example because some data should be added to the tables, then the user can set `action_type` to `'Stop'` and trigger the endpoint.

Trigger batch ingestion

After uploading the data via `/Ingest_json` or `/Ingest_rdf` with the `batch_ingestion` flag activated, the metadata of the new files must be updated and the files must be moved to the standard landing zone location where the SPHN Connector expects them to be to start the processing. This can be achieved with the following endpoint:

`/Trigger_batch_ingestion (project: str, [action_type:['Start', 'Check-status', 'Stop'] = 'Start')`



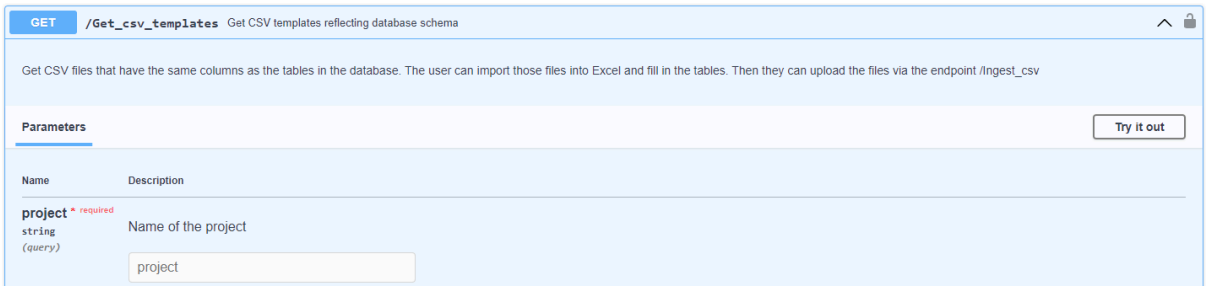
The user should favor batch ingestion via the described ingestion endpoints by setting the `batch_ingestion` flag. Nevertheless, there is also the possibility to upload data directly via MinIO service. Issues can arise when running the SPHN Connector if not used properly. Just upload good data with names reflecting the patient IDs into the prefix `Input/batch_ingestion` in MinIO. The service can be accessed with the same credentials as for the API. In fact a MinIO user is directly created with some policies when a new API user is created.

Batch ingestion is triggered asynchronously in Airflow. The status can be checked with `action_type = "Check status"`. The start pipeline endpoint is blocked until the batch ingestion terminates.

Preparing data in CSV format

The upload of CSV patient data is introduced from release 1.3.0 and at that point it is marked as **future-outlook** functionality (will be improved if needed in future releases). The upload of patient data via CSV files could be useful for the users that have large amounts of data stored in a database, where this data can be extracted in CSV files which reflect the SPHN Connector database schema. Examples shown are based on 2023.2 RDF schema. The CSV files must adhere to the structure defined by the templates extracted with following endpoint:

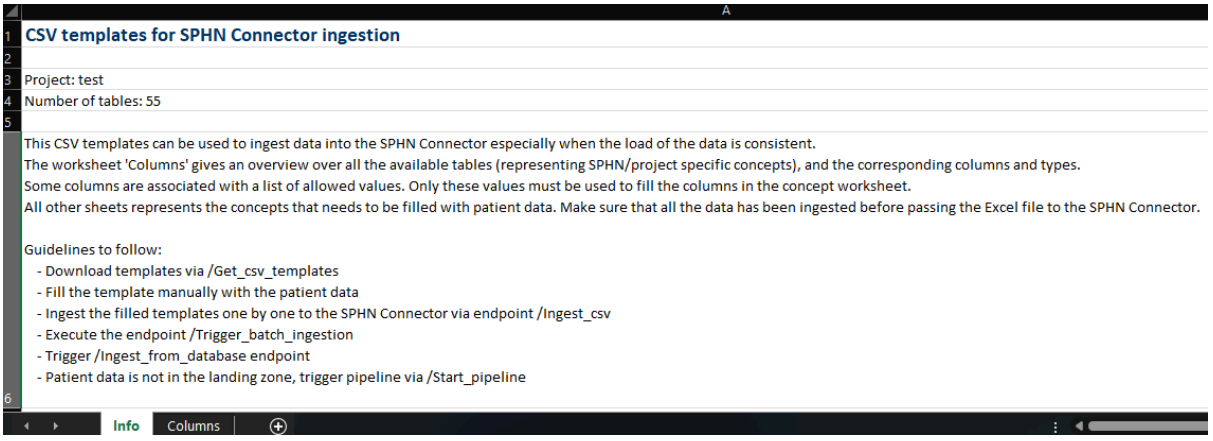
`/Get_csv_templates(project: str)`



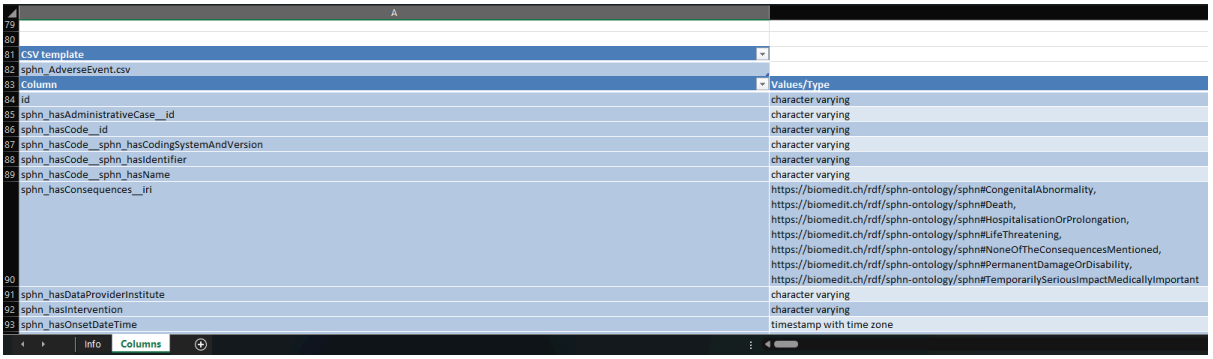
The endpoint returns a zipped file containing a CSV template for each table defined in the database schema and a readme file (available in .txt and .xlsx format) describing all the templates, their columns, and column types.

Name	Type	Compressed size	Password ...	Size
README.txt	Text Source File	8 KB	No	112 KB
README.xlsx	Microsoft Excel Worksheet	52 KB	No	68 KB
sphn_AccessDevicePresence.csv	Microsoft Excel Comma S...	1 KB	No	3 KB
sphn_AdministrativeCase.csv	Microsoft Excel Comma S...	1 KB	No	1 KB
sphn_AdministrativeGender.csv	Microsoft Excel Comma S...	1 KB	No	1 KB
sphn_AdverseEvent.csv	Microsoft Excel Comma S...	1 KB	No	1 KB
sphn_Age.csv	Microsoft Excel Comma S...	1 KB	No	1 KB
sphn_Allergy.csv	Microsoft Excel Comma S...	1 KB	No	1 KB

The ingestion process is briefly described in the *Info* sheet:



Templates description are defined in the *Columns* sheet of *README.xlsx*:

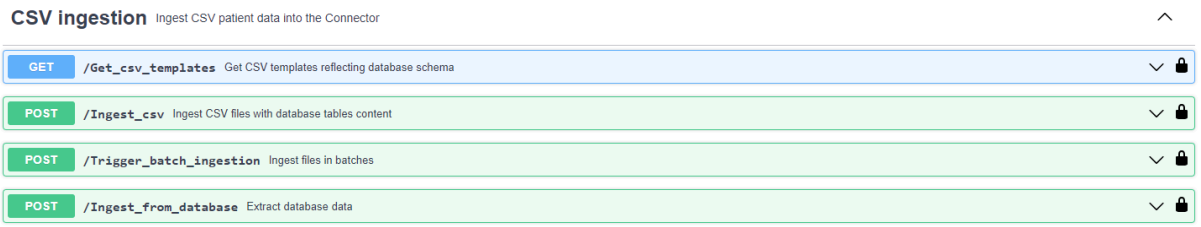


CSV template	
sphn_AdverseEvent.csv	
Column	**Values/Type**
id	character varying
sphn_hasAdministrativeCase__id	character varying
sphn_hasCode__id	character varying
sphn_hasCode__sphn_hasCodingSystemAndVersion	character varying
sphn_hasCode__sphn_hasIdentifier	character varying
sphn_hasCode__sphn_hasName	character varying
sphn_hasConsequences__iri	https://biomedit.ch/rdf/sphn-ontology/sphn#CongenitalAbnormality, https://biomedit.ch/rdf/sphn-ontology/sphn#Death, https://biomedit.ch/rdf/sphn-ontology/sphn#HospitalisationOrProlongation, https://biomedit.ch/rdf/sphn-ontology/sphn#LifeThreatening, https://biomedit.ch/rdf/sphn-ontology/sphn#NoneOfTheConsequencesMentioned, https://biomedit.ch/rdf/sphn-ontology/sphn#PermanentDamageOrDisability, https://biomedit.ch/rdf/sphn-ontology/sphn#TemporarilySeriousImpactMedicallyImportant
sphn_hasDataProviderInstitute	character varying
sphn_hasIntervention	character varying
sphn_hasOnsetDateTime	timestamp with time zone

Info | Columns

Ingest CSV data

The steps to follow to ingest CSV data into the SPHN Connector are also described in the extracted readme file (*Info* sheet). The API already gives the structured steps to follow to successfully ingest CSV data into the Connector:



After extracting and filling the CSV templates, those can be passed to the Connector via the endpoint

`/Ingest_csv(project:str , file: UploadFile, [file_type: [".csv", ".zip"] = ".csv"])`

If the CSV files are big, it's better to ingest them one by one into the connector via this endpoint (by using file type ".csv"). Otherwise, it's also possible to zip them into one file and pass it to the Connector after setting file type ".zip".

The uploaded CSV data needs then to be ingested into the Connector internal database and consequently extracted to JSON format. To achieve it just trigger sequentially endpoint

`/Trigger_batch_ingestion` and then `/Ingest_from_database`. At this point, the patient data is available in the landing zone and the processing can be triggered via `/Start_pipeline` as usual.

Note: at every ingestion the database schema tables are truncated.

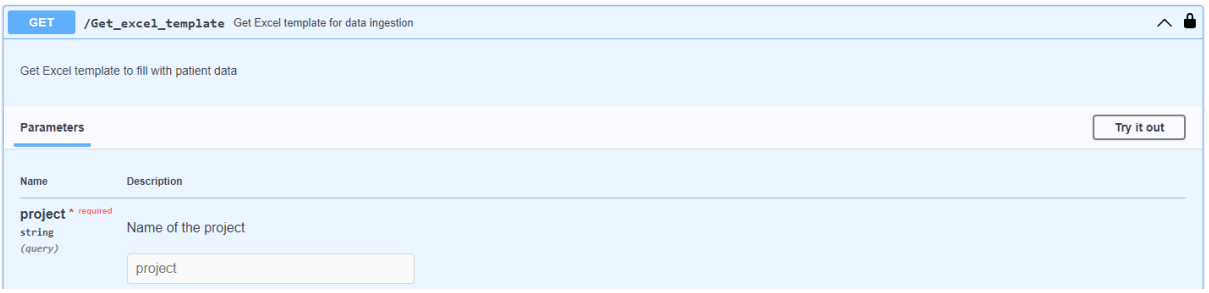
Note: it is important to upload all the necessary data and make sure that the `sphn_DataRelease.csv` template is correctly filled out and uploaded as well. It is the responsibility of the user to ingest data which matches the expected format.

Preparing data in Excel format

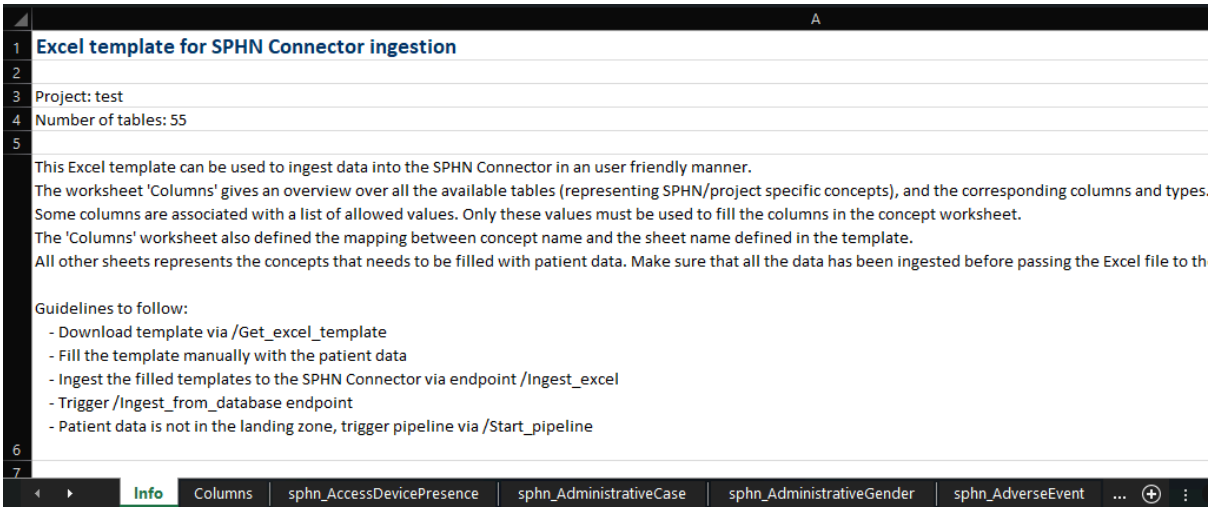
The upload of Excel patient data is introduced from release 1.3.0 and at that point it is marked as **future-outlook** functionality (will be improved if needed in future releases). The upload of patient data via Excel template could be useful for data providers that have a small amount of data to share with SPHN. Examples shown are based on 2023.2 RDF schema. A user-friendly interface like Excel could potentially motivate smaller data providers to share their routine data with SPHN.

The first step in the data preparation is the extraction of an Excel template constructed on the database schema of the create project. This is obtained with the endpoint

`/Get_excel_tempalte(project: str)`



The template is very similar to the CSV templates described in the previous sections, it is just reported in a single Excel file. It has an *Info* sheet with the description of the process;



a *Columns* sheet containing the mapping between database table name and sheet name, column names, column types for the defined tables;

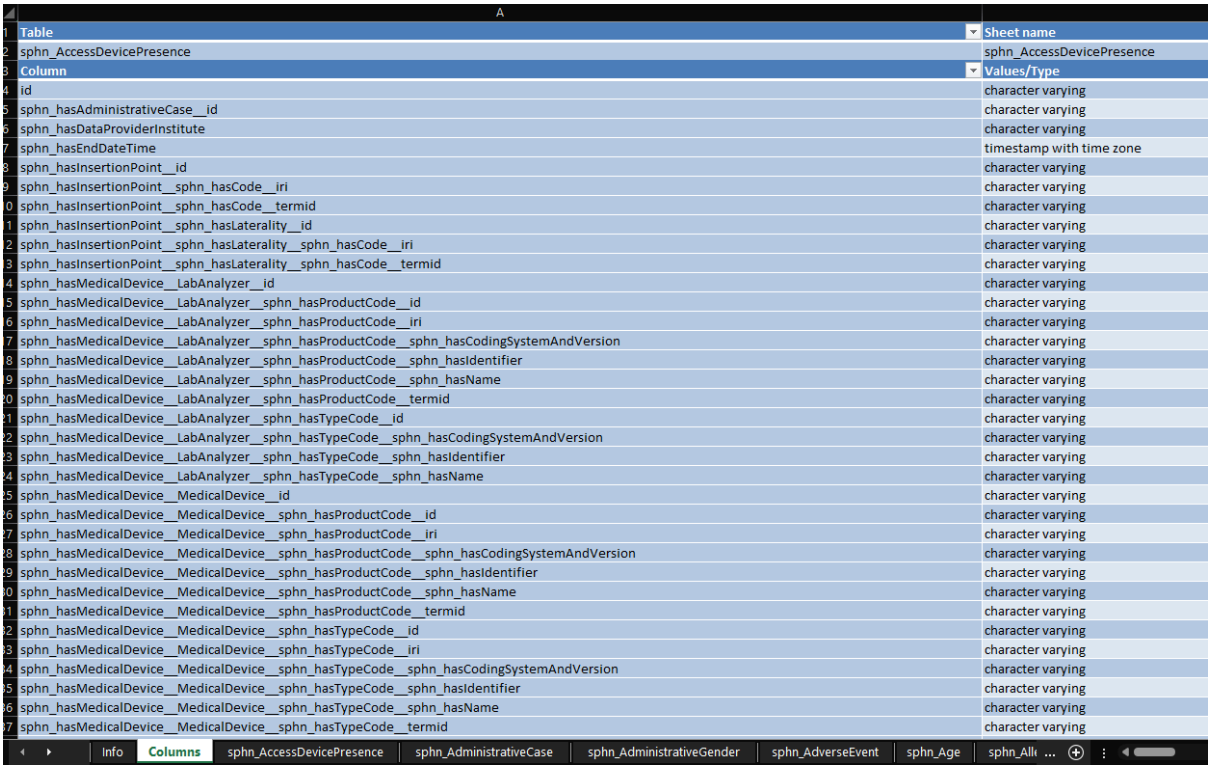
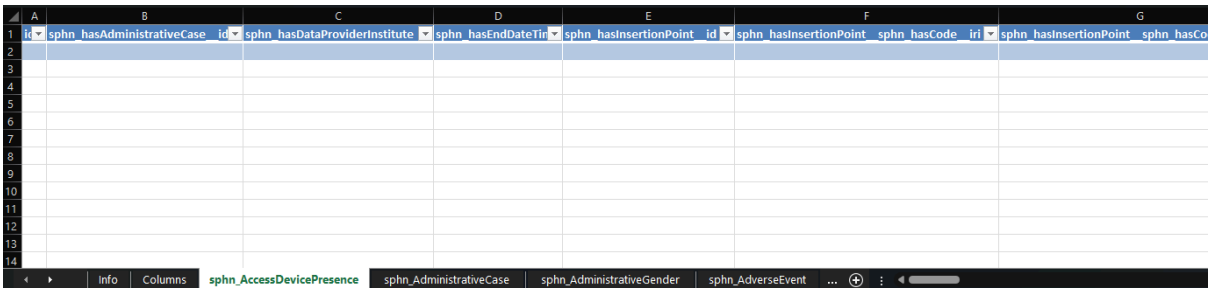


Table	Sheet name
sphn_AccessDevicePresence	sphn_AccessDevicePresence
Column	**Values/Type**
id	character varying
sphn_hasAdministrativeCase__id	character varying
sphn_hasDataProviderInstitute	character varying
sphn_hasEndDateTime	timestamp with time zone
sphn_hasInsertionPoint__id	character varying
sphn_hasInsertionPoint__sphn_hasCode__iri	character varying
sphn_hasInsertionPoint__sphn_hasCode__termid	character varying
sphn_hasInsertionPoint__sphn_hasLaterality__id	character varying
sphn_hasInsertionPoint__sphn_hasLaterality__sphn_hasCode__iri	character varying
sphn_hasInsertionPoint__sphn_hasLaterality__sphn_hasCode__termid	character varying
sphn_hasMedicalDevice__LabAnalyzer__id	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasProductCode__id	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasProductCode__iri	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasProductCode__sphn_hasCodingSystemAndVersion	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasProductCode__sphn_hasIdentifier	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasProductCode__sphn_hasName	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasProductCode__termid	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasTypeCode__id	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasTypeCode__sphn_hasCodingSystemAndVersion	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasTypeCode__sphn_hasIdentifier	character varying
sphn_hasMedicalDevice__LabAnalyzer__sphn_hasTypeCode__sphn_hasName	character varying
sphn_hasMedicalDevice__MedicalDevice__id	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__id	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__iri	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__sphn_hasCodingSystemAndVersion	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__sphn_hasIdentifier	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__sphn_hasName	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__termid	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasTypeCode__id	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasTypeCode__iri	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasTypeCode__sphn_hasCodingSystemAndVersion	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasTypeCode__sphn_hasIdentifier	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasTypeCode__sphn_hasName	character varying
sphn_hasMedicalDevice__MedicalDevice__sphn_hasTypeCode__termid	character varying

Info | Columns | sphn_AccessDevicePresence | sphn_AdministrativeCase | sphn_AdministrativeGender | sphn_AdverseEvent | sphn_Age | sphn_All ...

and all the sheets that should be filled with patient data

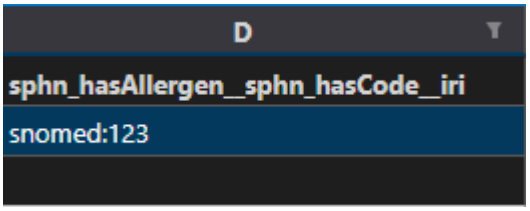


id	sphn_hasAdministrativeCase__id	sphn_hasDataProviderInstitute	sphn_hasEndDateTir	sphn_hasInsertionPoint__id	sphn_hasInsertionPoint__sphn_hasCode__iri	sphn_hasInsertionPoint__sphn_hasCo

Info | Columns | sphn_AccessDevicePresence | sphn_AdministrativeCase | sphn_AdministrativeGender | sphn_AdverseEvent ...

For all the column names representing an IRI value (i.e. column names ending with `__iri`), the SPHN Connector implicitly replaces a list of prefixes with the corresponding full IRI. This allows the user to fill in the data faster and in a compacted way.

Example for the `sphn_Allergy` table:



D
sphn_hasAllergen__sphn_hasCode__iri
snomed:123

The prefix `snomed:` will be implicitly replaced with `http://snomed.info/id/` and the resulting value in the database will be

sphn_hasAllergen__sphn_hasCode__iri character varying (3000)
http://snomed.info/id/123

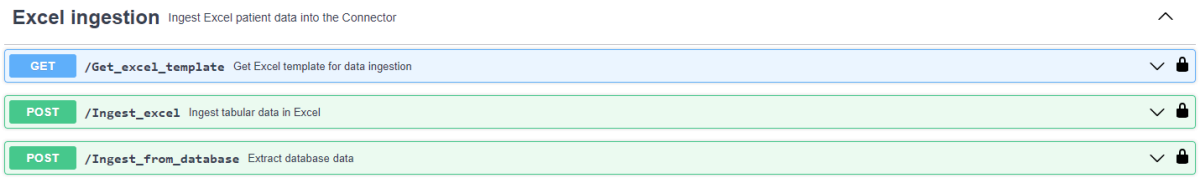
Of course, it is still possible to insert in the Excel sheet the full IRIs.

The prefixes that are automatically replaced are the following:


```
{
    "snomed:":"http://snomed.info/id/",
    "chop:":"https://biomedit.ch/rdf/sphn-resource/chop/",
    "geno:":"http://purl.obolibrary.org/obo/GENO_",
    "icd-10-gm:":"https://biomedit.ch/rdf/sphn-resource/icd-10-gm/",
    "loinc:":"https://loinc.org/rdf/",
    "so:":"http://purl.obolibrary.org/obo/SO_",
    "atc:":"https://www.whocc.no/atc_ddd_index/?code=",
    "hgnc:":"https://biomedit.ch/rdf/sphn-resource/hgnc/",
    "ucum:":"https://biomedit.ch/rdf/sphn-resource/ucum/",
    "sphn:":"https://biomedit.ch/rdf/sphn-schema/sphn#",
    "eco:":"http://purl.obolibrary.org/obo/ECO_",
    "emdn:":"https://biomedit.ch/rdf/sphn-resource/emdn/",
    "genepio:":"http://purl.obolibrary.org/obo/GENEPIO_",
    "obi:":"http://purl.obolibrary.org/obo/OBI_",
    "ordo:":"http://www.orpha.net/ORDO/",
    "efo:":"http://www.ebi.ac.uk/efo/EFO_",
    "edam:":"http://edamontology.org/"
}
```


Ingest Excel data

After filling in the Excel sheets with all the necessary patient data, the data can be ingested into the SPHN Connector with this structured process:



The first step is the ingestion of the Excel template via endpoint

`/Ingest_excel_template(project: str, file: UploadFile)`

The uploaded Excel template will be parsed and the data will be directly ingested into the database. After this step, the user triggers `/Ingest_from_database` to extract the data into the landing zone and then they can start processing it with `/Start_pipeline` as usual.

Note: at every ingestion the database schema tables are truncated.

Note: it is important to upload all the necessary data and make sure that the sheet `sphn_DataRelease` is correctly filled out. It is the responsibility of the user to ingest data which matches the expected format.

Processing the data and retrieving the RDF turtle files

Start and Stop processing of data

Start

With the start pipeline function the processing is triggered in the SPHN Connector. Without starting the process the data just waits where it was ingested. There are two basic option on how to start the pipeline:

- Start complete pipeline - This will implicitly trigger all the different steps to process ingested data and produce validated RDF turtle files in the release zone. This is the default you want to use the pipeline with, in a production environment. If you want to start the complete pipeline, you will not provide a step as a parameter (see command below).
- Start an individual process step - This option is foreseen for testing and investigating errors and you would not use this in the production. The possible steps are (parameters to specify are listed in the command below):
    - **Pre-check/De-ID** - Pre-Check phase will validate the ingested data in order to guarantee that it can be processed and that with this data an RDF file can be generated at all. Documentation: **Pre-Check.** De-Identification phase is an optional step that allows data providers to de-identify the data before they are converted into RDF. Documentation: **De-Identification.** Pre-Check/De-ID step is implemented from WP5/release 0.5.0.
    - **Integration** - This will apply the RML mapping and generate the RDF turtle files which will be validated in the next step. In case of RDF input data, the step moved the files to the graph zone. RDF conversion with RML is implemented from release 0.3.0.
    - **Validation** - This step does validate the RDF files using the SPHN QC Framework and is currently the only step implemented in release 0.2.0.
    - **Publish** - This step is foreseen for the future where the SPHN Connector could also call the SETT tool or similar to transmit a file directly to the project space. However, this will **not** be available in the release 1.0.0.

Additionally there are two more options when starting the pipeline:

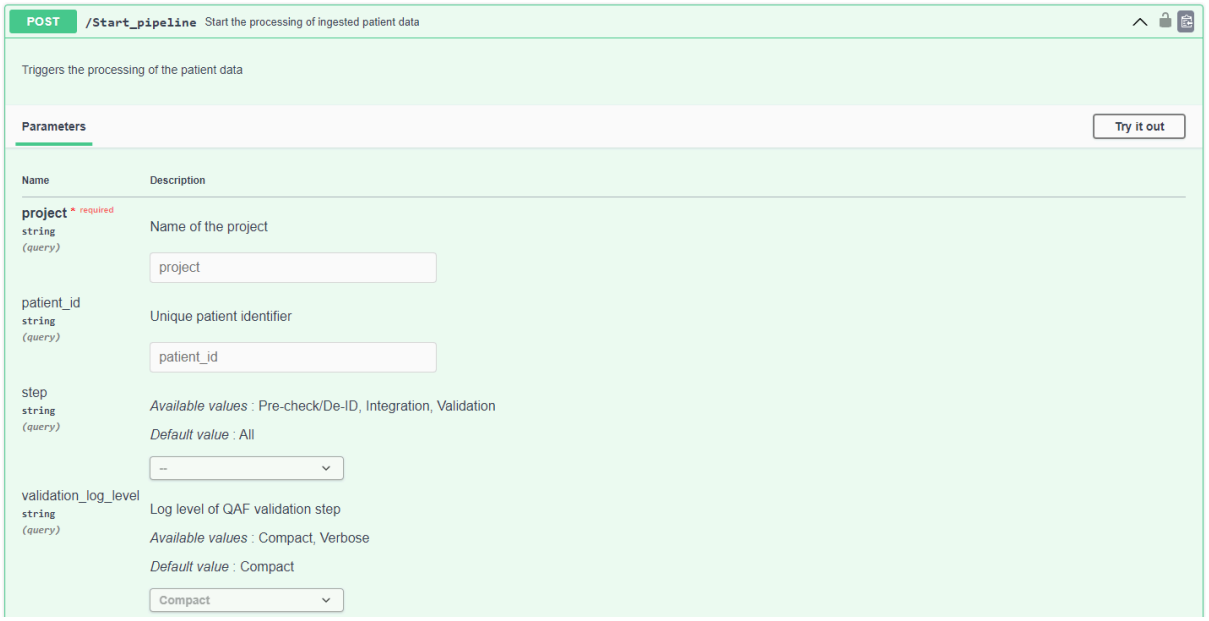
- Per Patient (patient_id) - You can start the pipeline for a specific patient if you supply the patient id in the request. This might be useful if you ingest the data in a tabular format and your ETL tool knows when the data for a patient is completely ingested. In this case you could start the processing for this patient. Alternatively you could also use this for testing where you want to limit the testing to one patient only.
- Logging details (validation_log_level) - The level of information logged for each patient and process step depends on this parameter. If you set it to verbose you get all the details, including the QC report set to “logfulldetails=true”. However, the default is compact, which will run the QC framework with the default “logfulldetails=false”. Accessing these log files is described in the next section. The options for this in the API are not boolean but are reflected as “compact” / “verbose”

The pipeline is started on a project basis, this is why the project parameter is mandatory. The pipeline is always started, except when there is no patient data available for the step that needs to be started.

```
/Start_pipeline (project: str, [patient_id: str], [step:
["Pre-check/De-ID", "Integration", "Validation"]],
[validation_log_level: ["compact", "verbose"]])
```

If you start the pipeline for a given step, you have to make sure that the data is ready for this step. Therefore you may want to trigger one step after another to make sure the data is always available at the right step.
Each pipeline triggering, regardless of parameters, will be defined by a universal unique identifier (run_uuid). Its value is returned in the JSON response body together with the return message. It can be used together with the results of /Get_status to locate the execution status of a specific pipeline triggered.

***Note**: At the moment if you start the pipeline it will process the data as specified and after a while it will stop automatically. If you have started the pipeline for one patient only, you can not start another pipeline for another patient and you have to wait until the pipeline is automatically stopped. Challenge here is that it is difficult to foresee how long the processing of one patient takes and especially the loading of taxonomies, which happens in this case by patient, takes some time. For the future we are looking into how to synchronize processes better.*

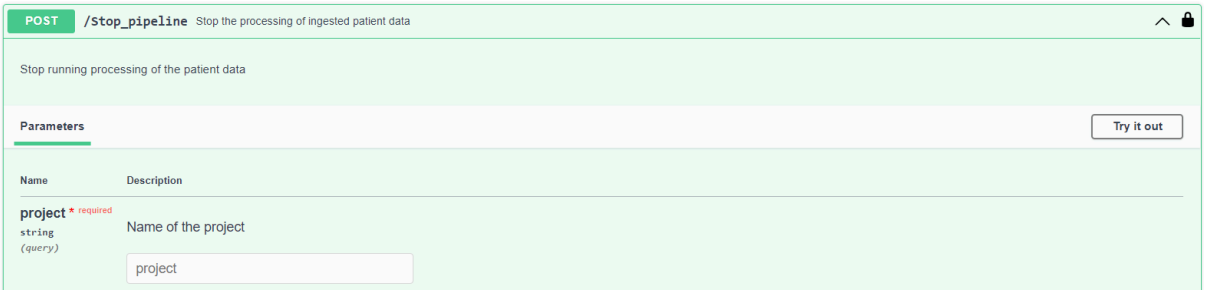


Stop

As the pipeline usually stops automatically after processing its data the stop pipeline is useful if you want to prematurely stop the processing. Starting with release 0.3.0 there are no more options beside the project name. The /Stop_pipeline will stop the running pipeline no matter how it was started..

```
/Stop_pipeline (project: str)
```

***Note**: You can only stop a pipeline that is actually running. A pipeline that is not running will return a non 200 return code.*

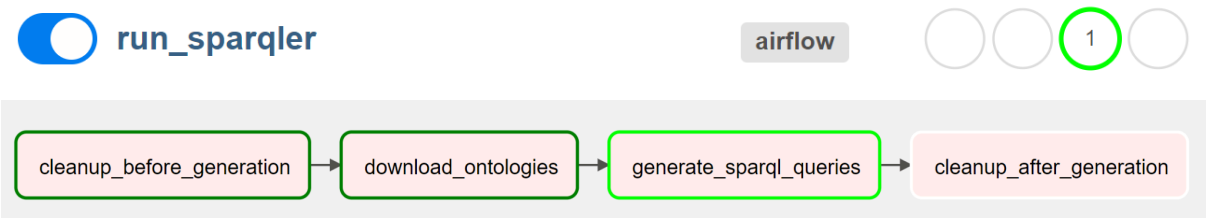

POST
/Stop_pipeline
Stop the processing of ingested patient data
Stop running processing of the patient data
Parameters
Try it out
Name
Description
project * required
string
(query)
Name of the project
project

Start statistics

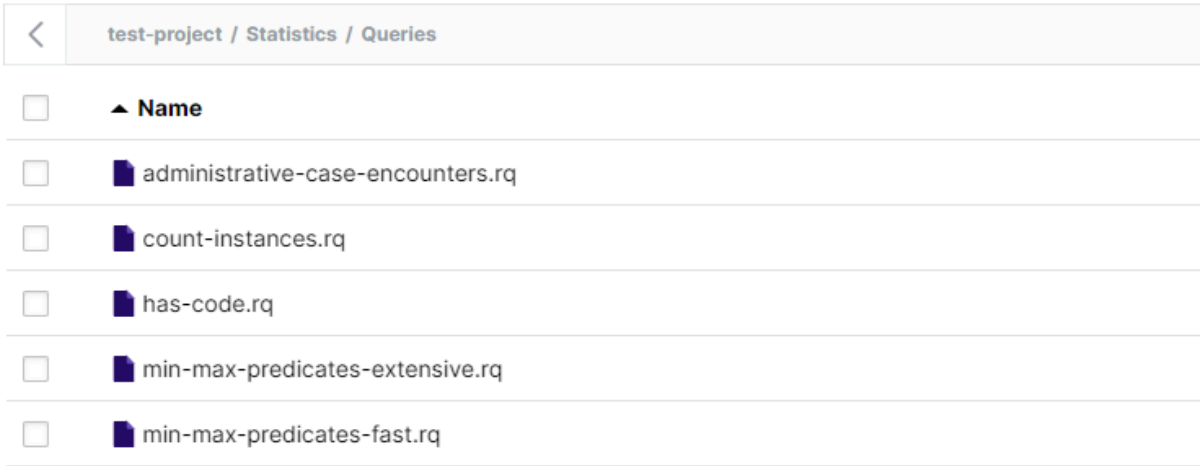
SPHN Connector release 1.3.0 introduces a new step for the generation of patient data statistics. SPARQL queries for statistical purposes are generated with a modified version of the SPARQLer Tool. Statistics are performed on patient files available in the `graph_zone` persistence layer and must be triggered separately. Once the data is available in the graph zone the statistics can be triggered, meaning that it is not mandatory to execute the validation step on the data before executing the statistics. The statistics execution can be monitored via the `/Get_status` endpoint as the other steps. Once completed, aggregated reports can be extracted via the endpoint `/Get_statistics_reports`. Those reports contain both the standardized reports which are constructed on the auto-generated SPARQL queries, and the individual reports which report the results of the external QC queries uploaded via the `/Upload_external_files`.

SPARQL queries generation

The generation of the SPARQL queries is triggered automatically after the creation of a SPHN Connector project. The SPHN Connector can be used as usual while the queries are being generated via Airflow DAG:



The generated SPARQL queries are stored in MinIO in the project's bucket under the prefix */Statistics/Queries*.



Summary of the queries:

- *administrative-case-encounters.rq*: this is not generated by the SPARQLer but by a custom query and it is needed to keep track of the encounters, i.e. the references to different values of property `sphn:hasAdministrativeCase`.

Note: the encounters count is constructed by matching the property name `sphn:hasAdministrativeCase`. According to DCC standards, the projects should not define a project specific property for the `sphn:AdministrativeCase` concept. If it's the case just keep in mind that that won't be counted for the encounters count.

- *count-instances.rq*: modified version of the SPARQL query to count the number of instances of each concept.
- *has-code.rq*: modified version of the SPARQL query to count the used codes.
- *min-max-predicates-extensive.rq*: modified version of the SPARQL query collecting min/max values of concept/attribute and all the related distinct values.
- *min-max-predicates-fast.rq*: modified version of the SPARQL query collecting min/max values of concept/attribute and the distinct count per patient.

Individual reports

In addition to the automated SPARQL queries generation, the user can also provide arbitrary SPARQL queries to execute on patient data. The queries must be provided via the /Upload_external_files endpoint and will be executed during the statistics step. From the fact we do not have any information about the structure of the results returned by the queries, we cannot apply any aggregation on the generated data. Nevertheless, we report all the results in a table and return them in a single tabular file after adding information about data provider ID, project name, and patient ID. The user can then use the extract to perform their own statistics. Unless proven otherwise, the user can upload arbitrary QC queries into the Connector.

Example (based on 2023.2 RDF schema):
We upload an external QC query named *test2.rq*:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ucum: <https://biomedit.ch/rdf/sphn-resource/ucum/>
PREFIX sphn: <https://biomedit.ch/rdf/sphn-ontology/sphn#>

SELECT ?this ?value ?code
WHERE {
    ?this rdf:type sphn:Age .
    ?this sphn:hasQuantity ?quantity .
    ?quantity sphn:hasValue ?value .
    ?quantity sphn:hasUnit ?unit .
    ?unit sphn:hasCode ?code .
}
```

We start the statistics on the graph zone data and we extract the Excel report. We see the sheet test2 in addition to the standardized reports. It contains the results of the query execution on the 2 available patients:

	A	B	C	D	E	F
1	data_provider_id	project_name	patient_id	this	value	code
2	DATA-PROVIDER-ID	123	1	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-b3b68b57-da54-4267-9d13-8266d26d5396	2520.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/min
3	DATA-PROVIDER-ID	123	1	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-555a4085-4578-4ab3-a6a9-74652371ea2c	6958.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/wk
4	DATA-PROVIDER-ID	123	1	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-033c1469-4f06-4a18-b4a2-64447d3c279b	8600.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/a
5	DATA-PROVIDER-ID	123	1	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-a36bcb01-67e9-4363-905c-053b25fdacbe	4550.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/h
6	DATA-PROVIDER-ID	123	1	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-2c623ac3-ad70-47cf-a358-cb1dbe3feafd	1081.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/h
7	DATA-PROVIDER-ID	123	1	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-ecadbb86-f16f-4487-b7f2-1e5cce0755e3	277.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/mo
8	DATA-PROVIDER-ID	123	2	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-b3b68b57-da54-4267-9d13-8266d26d5396	2520.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/min
9	DATA-PROVIDER-ID	123	2	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-555a4085-4578-4ab3-a6a9-74652371ea2c	6958.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/wk
10	DATA-PROVIDER-ID	123	2	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-033c1469-4f06-4a18-b4a2-64447d3c279b	8600.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/a
11	DATA-PROVIDER-ID	123	2	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-a36bcb01-67e9-4363-905c-053b25fdacbe	4550.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/h
12	DATA-PROVIDER-ID	123	2	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-2c623ac3-ad70-47cf-a358-cb1dbe3feafd	1081.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/h
13	DATA-PROVIDER-ID	123	2	https://biomedit.ch/rdf/sphn-resource/CHE-229_707_417-DataProviderInstitute-Age-ecadbb86-f16f-4487-b7f2-1e5cce0755e3	277.0e0	https://biomedit.ch/rdf/sphn-resource/ucum/mo

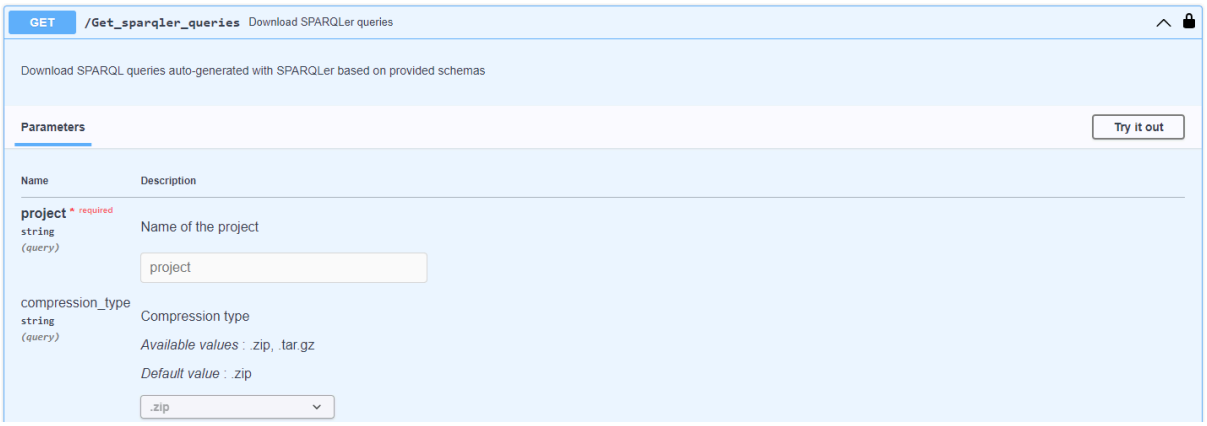
concepts_count | concepts_min_max | concepts_codes | test2

Columns A, B, and C have been extracted from the Connector metadata. While columns D, E, F are simply the results returned by the SPARQL query provided (`SELECT ?this ?value ?code`).

Extracting the generated queries

In case the user is interested in the used queries, they can either download them via MinIO, or trigger the following endpoint and download the returned response.

`/Get_sparqler_queries(project: str)`



Trigger the statistics

The statistics on the patient data available in the graph zone (i.e. RDF patient files ready for validation step) is triggered with the endpoint:

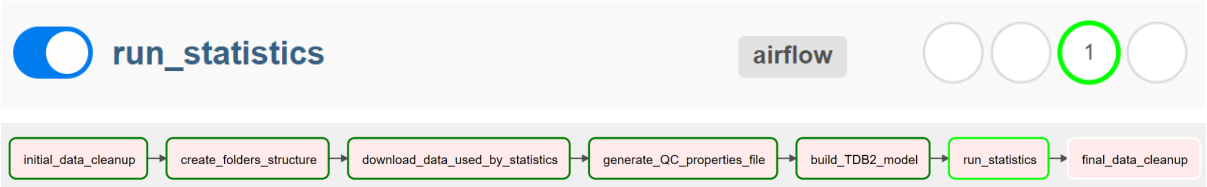
`/Start_statistics(project: str, [profile: ["Fast", "Extensive"]])`

Here we can distinguish between two statistics profiles:

- **Fast**: the fast profile is the less resources/time consuming. Statistics do not collect any values from the patient data but only keeps track of the min/max/count per patient. When aggregating the statistics over multiple patients for the final reports, we won't get any sample values nor distinct counts over all the patients. Instead we'll get the min/max value

of the distinct count over the single patients. Consider using this profile for less descriptive statistics that can be produced with less resource usage and in a shorter time.

- **Extensive**: the extensive profile is more resources/time consuming. The SPARQL queries collect min/max values and distinct values and then report the overall distinct count and sample values in the final report.



When triggering the statistics, a modified version of the QAF is launched and it will execute the loaded external QC queries and the autogenerated SPARQL queries discussed above. The results are firstly written to CSV files locally and then loaded into the internal database into some ad-hoc tables. These tables will be queried and the results aggregated when the final reports are extracted.
Statistics related database tables:

- *sparqler_count_instances*
- *sparqler_min_max_predicates_extensive_numeric*
- *sparqler_min_max_predicates_fast*
- *sparqler_min_max_predicates_fast_numeric*
- *sparqler_has_code*
- *sparqler_administrative_case_encounters*
- *stats_individual_reports*

The status of the statistics execution can also be monitored via the `/Get_status` endpoint:

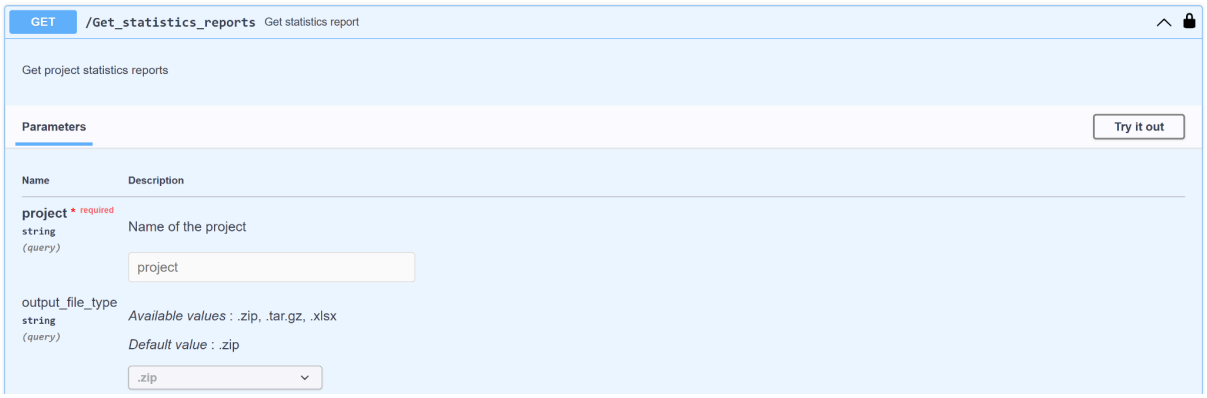

```
{
  "run_uuid": "d667c920-c88a-4270-b62a-eed0f19532e5",
  "project_name": "test-project",
  "pipeline_step": "Statistics",
  "pipeline_status": "SUCCESS",
  "patients_processed": 1,
  "patients_with_warnings": 0,
  "patients_with_errors": 0,
  "execution_time": "2023-08-29 16:23:12",
  "elapsed_time": "0:00:02"
},
```


After the statistics execution, the generated data is extracted and aggregated and the statistics reports are stored in MinIO at prefix */Statistics/Reports*. The reports can then be extracted with the `/Get_statistics_reports` endpoint.

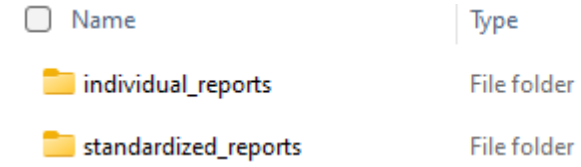
Extracting the reports

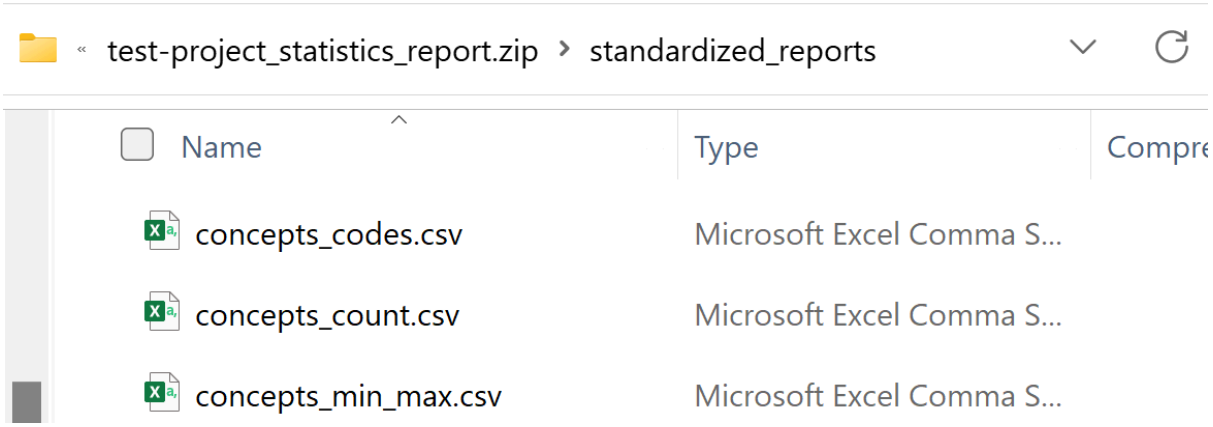
Once the statistics has been successfully executed, the standardized and individual reports can be extracted with the `/Get_statistics_reports` endpoint:

`/Get_statistics_reports (project: str, [output_file_type: [.zip, .tar.gz, .xlsx]])`



When the reports are downloaded in compressed format, the downloaded bundle will contain two subfolders *standardized_reports* and *individual_reports*, each one containing CSV files with the statistics data.





On the other hand, the Excel output gives a more readable representation of the statistics. Standardized reports will be reported in the sheets concepts_count, concepts_min_max, and concepts_code, while individual reports (if any external QC query has been provided) will be reported in additional sheets with sheet name reflecting the uploaded query file name.

Note: the statistics Excel report is added to the download bundle returned by the endpoint `/Download_data`.

	A	B	C	D
1	concept	instances	patients	encounters
2	sphn:AccessDevicePresence	18	3	2
3	sphn:AdministrativeCase	3	3	0
4	sphn:AdministrativeGender	9	3	0
5	sphn:AdverseEvent	18	3	2
6	sphn:Age	54	3	0
7	sphn:Allergen	18	3	0
8	sphn:Allergy	9	3	0
9	sphn:AllergyEpisode	18	3	2
10	sphn:Biobanksample	18	3	2
11	sphn:BirthDate	9	3	0
12	sphn:BloodPressure	18	3	2
13	sphn:BodyHeight	18	3	2
14	sphn:BodyMassIndex	18	3	2
15	sphn:BodyPosition	18	3	2
16	sphn:BodySite	162	3	0
17	sphn:BodySurfaceArea	18	3	2
18	sphn:BodyTemperature	18	3	2
19	sphn:BodyWeight	18	3	2
20	sphn:CardiacIndex	18	3	2
21	sphn:CardiacOutput	18	3	2
22	sphn:CareHandling	3	3	0
23	sphn:ChromosomalLocation	9	3	0
24	sphn:Chromosome	9	3	0
25	sphn:CircumferenceMeasure	18	3	2
26	sphn:CivilStatus	9	3	0
27	sphn:Code	402	3	0
28	sphn:Consent	9	3	0
29	sphn:DataDetermination	18	3	0
30	sphn:DataFile	36	3	0
31	sphn:DataProviderInstitute	3	3	0
32	sphn:DataRelease	3	3	0
33	sphn:DeathDate	18	3	0

concepts_count | concepts_min_max | concepts_codes | qc_query

(Report based on 2023.2 RDF schema)

Monitoring the Pipeline/processes/jobs

Get Status

This endpoint will return all the results of pipeline steps that were executed for a given project. The project steps, as of release 0.5.0, are `Pre-check/De-ID`, `Integration`, and `Validation`. The returned statuses are an array of all the pipelines that did run and for every step the following data is returned:

```
{
    "project_name": "Hospfair-1",
    "pipeline_step": "Integration",
    "pipeline_status": "SUCCESS",
    "patients_processed": 3,
    "patients_with_warnings": 0,
    "patients_with_errors": 1,
    "execution_time": "2022-10-06 10:07:57",
    "elapsed_time": "0:06:10"
}
```

To get the status of a pipeline for a project this endpoint can be requested:

`/Get_status (project: str, [step: ["Pre-check/De-ID", "Integration", "Validation", "Statistics"]])`

The following status codes could be returned:

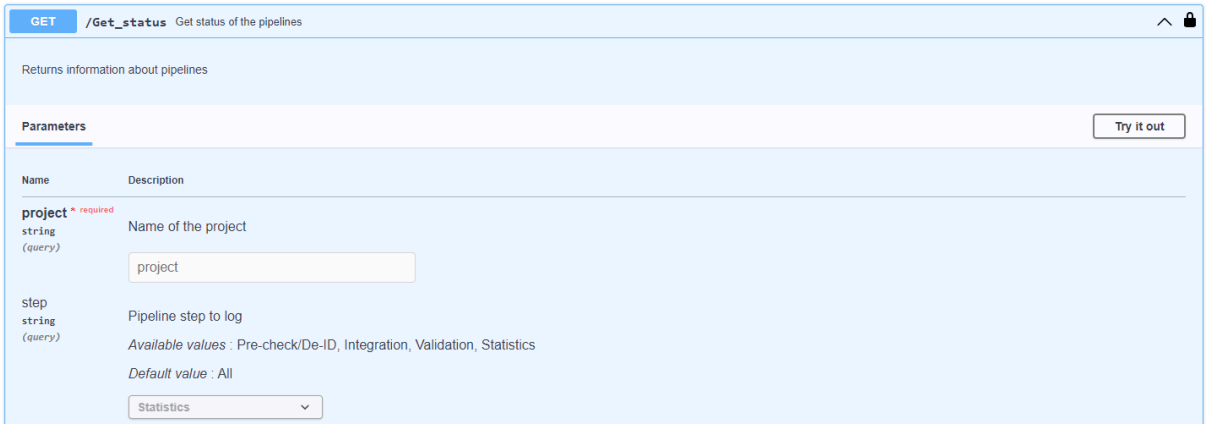
- Success - the pipeline step completed correctly and the pipeline step is stopped again.
- Running - the pipeline step is still running.
- Skipped - no files found to trigger the step
- Failed - the execution of the pipeline step failed due to some exceptions.

Patients counting:

- *processed_patients*: number of patients that were processed in that step
- *patients_with_warnings*: number of patients that raised a warning. Currently, this can happen only during the Pre-Check/De-Identification step where a Pre-Check with severity level 'WARN' raises an error, and during the Validation step, when the imported external terminologies do not match the expected imports of the loaded schema.
- *patients_with_errors*: number of patients that raised errors. Could be due to failed execution of the step, or due to some Pre-Check errors. Check patient logs to get more details.

We recommend using the */Get_status* endpoint to determine the progress in the SPHN Connector.

Note: *In order to identify the pipeline step to be checked you will have to parse the returned JSON from the beginning (latest ones should be at the beginning) and filter by the time the pipeline was last started (to make sure the step is from last run, in case something went wrong and no step was started at all).*

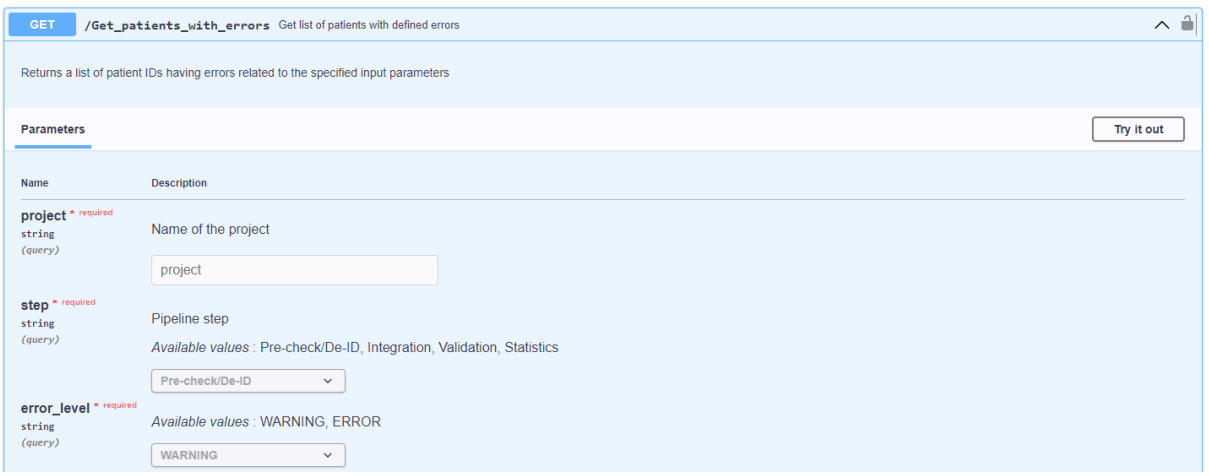


Note: from release 1.4.0 we introduced real-time reporting of the status. This is made possible after checking the files that have already been processed directly on the database or in the local file system. Once a thread terminates, there is a small fraction of time where the local files are deleted; if the status is checked during that period it is possible to get inconsistent results. The suggestion is just to wait a few seconds and retry. The integration scenario from RDF to RDF is not monitored in a real-time manner but only thread per thread once they finish. Nevertheless, that process should be faster.

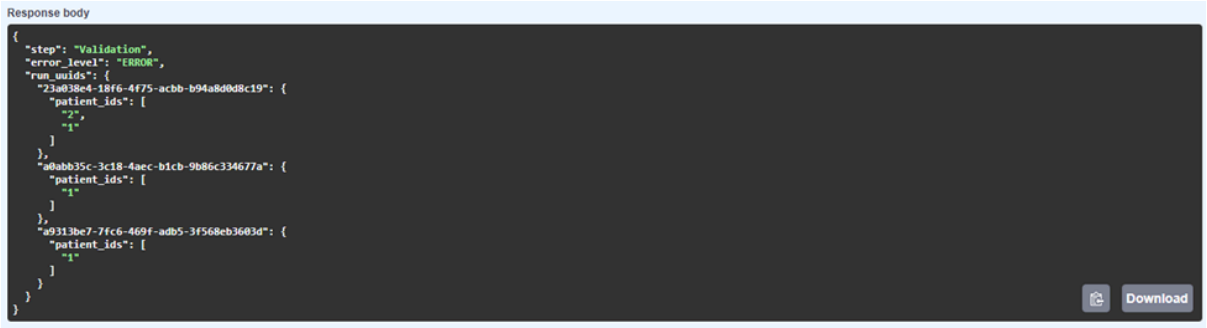
Get patients with errors

Sometimes, when we check the status of a pipeline execution (via endpoint `/Get_status`) we see that the number of patients with warnings or with errors for a specific step is different from zero. In this case we do not have any information about the patients that are failing. To fill this gap, we can use the endpoint `/Get_patients_with_errors` which for a specific combination of project name, step name, and error level, returns the list of patient IDs with the specified error for all the executed pipelines.

`/Get_patients_with_errors (project: str, step: ["Pre-Check/De-Id", "Integration", "Validation", "Statistics"], error_level: ["WARNING", "ERROR"])`



The response body has information about the input parameters passed and a list of run_uuids with the corresponding patient IDs. The information is grouped by run_uuids and sorted by the run datetime. The latest run will be the first entry and the first run will be the last entry. For example:



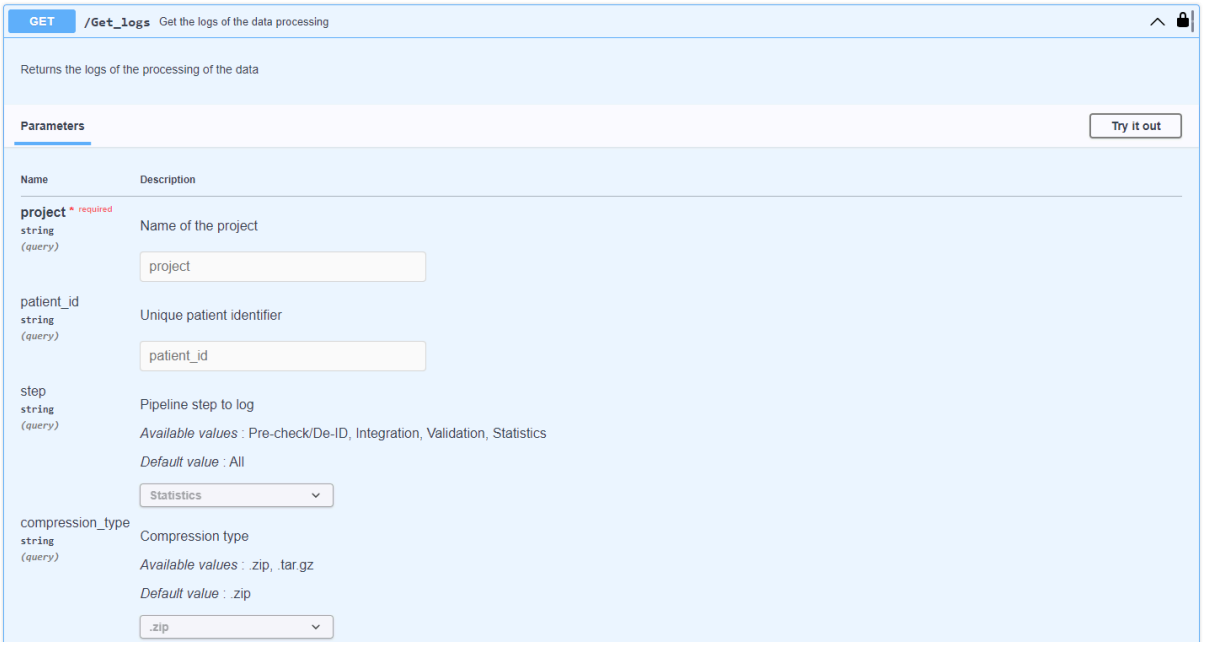
With this information the user can extract only the logs for the patients they are interested in and they don't have to extract the full logs and search for errors there.

Get Logs

To get the logs of the processing of the ingested data the following endpoint can be requested:

`/Get_logs (project: str, patient_id: str, [step: ["Pre-check/De-ID", "Integration", "Validation", "Statistics"]], [compression_type: [".zip",".tar.gz"]])`

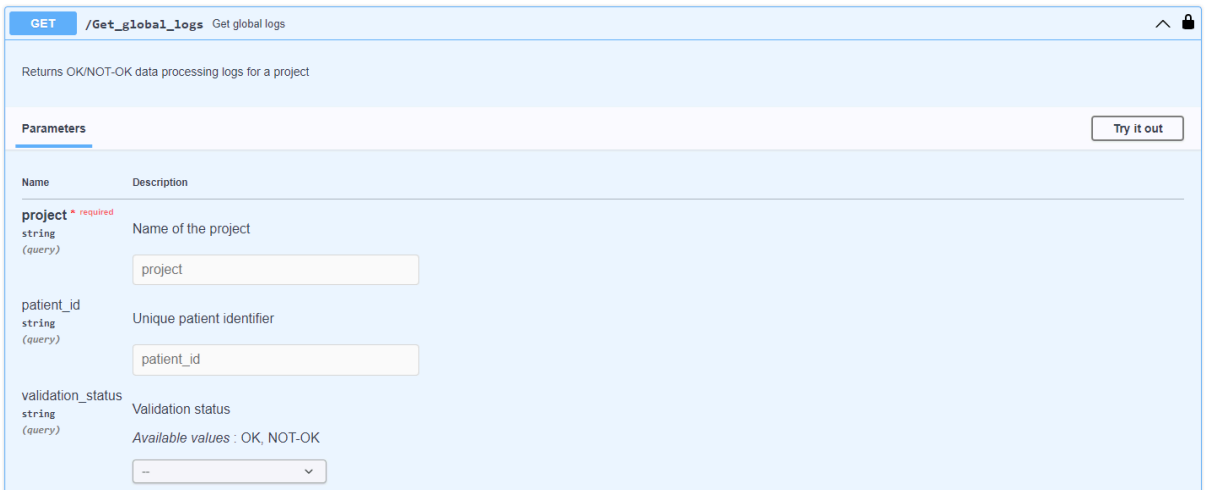
In order to request the logs the process has to be at least started for the given project/patient(s)/step. If you start the full pipeline for a patient but you only want to get the RDF validation log you can specify that only the log for the step "Validation" should be returned.
From release 0.3.0 you can also specify if you like to download the logs as .zip or .tar.gz file (default is .zip). See also General Remarks.



Get Global Logs

Global logs for a project can be requested via following endpoint:

/Get_global_logs (project: str, [patient_id: str], [validation_status: ["OK", "NOT-OK"]])



It is possible to filter the results by patient ID and/or validation status of the QAF validation. If those fields are not specified, then we get all the logs for the given project. In addition to the validation logs for each patient, the response also reports information about all the patient data that has been ingested, such as timestamp and ingestion type. Moreover, the history of upload/deletion of configuration files is also reported in the logs.

Quality Check logs

The patient logs returned by `/Get_logs` or `/Get_global_logs` contains the quality check report if the validation step has been executed on the patient (the QC logs are also available when downloading the data via `/Download_data` in case the validation was unsuccessful or if there are warning messages). When a patient is marked with status `validated_ok: false` it means that during the SHACL validation some violations are raised. The QC report has the following columns:

- *resultSeverity*: severity level of the violation: sh:Violation, sh:Warning, sh:Info.
- *resultPath*: property related to the focusNode.
- *sourceConstraintComponent*: type of SHACL constraint applied.
- *sourceShape*: name of the constraint raising a violation.
- *count*: count of violations grouped on (resultPath, resultSeverity, sourceConstraintComponent, and sourceShape).
- *focusNode*: resource on which the constraint is applied and failed.
- *resultMessage*: message of the violation.
- *value*: value related to the violation.

To have a successful validation of the patient data is important to manually fix all the violations with severity sh:Violation.

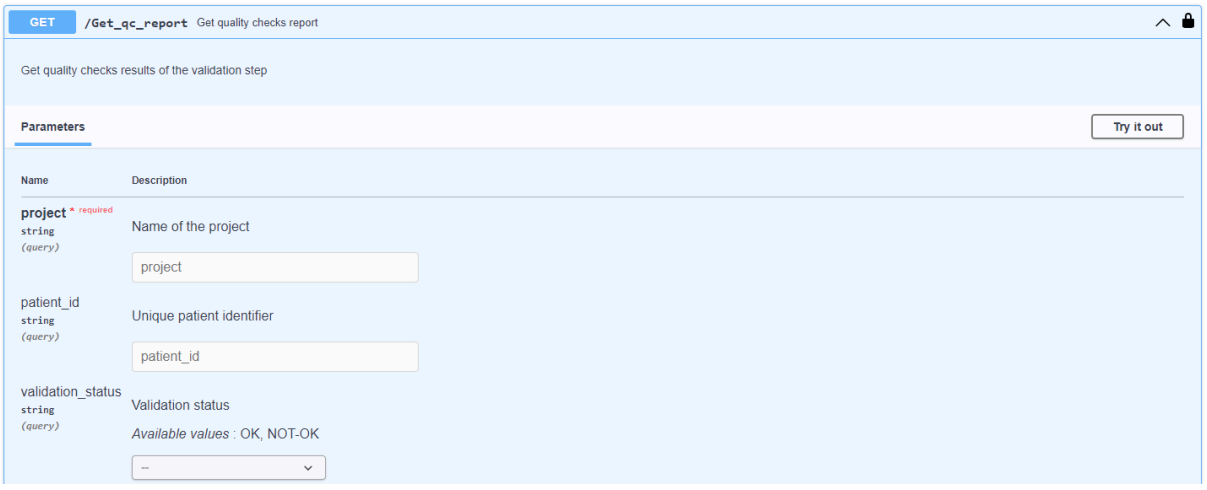
Violations of severity sh:Info and sh:Warning are introduced from release 1.3.0 (Shacler Repo). This is a small summary of the different types of violations available at that point:

Constraint	Description	Severity
constraints:sphnConcept	Main constraints extracted from the	sh:Violation

	schema when building the SHACL file. For example cardinality constraints, code existence, terminologies constraints, … In addition from release 1.3.0, we add a SPARQL constraint checking that the value of *sphn#hasStartDateTime* is not antecedent of *sphn#hasEndDateTime*.	
sphnConcept_Warning_Naming	Checks that the IRI creation of subjects follow naming conventions.	sh:Warning
sphnConcept_Warning_Codes	Look for unversioned codes that are marked as deprecated.	sh:Warning
UnversionedCodeHasMeaningChange	Look for unversioned codes that have changed their meaning over time.	sh:Warning
OldVersionedCodeHasBeenValid	Look for versioned codes that had a meaning change and are not valid anymore.	sh:Info
OldVersionedCodeStillValid	Look for versioned codes that had a meaning change but are still valid.	sh:Info

For a more detailed overview over all the processed patients, the user can inspect the internal database table *quality_checks* or extract an Excel report of the quality checks results via the following endpoint:

`/Get_qc_report (project: str, [patient_id :str], [validation_status: str])`



Consider that if the number of patients is very high, it might be safer and more efficient to directly check on the database table.

Note: if a custom report SPARQL query is uploaded via `/Upload_external_files endpoint`, the results are not stored into the *quality_checks* table and will not be visible in the

report extracted with this endpoint. This is due to the fixed structure of the database table which cannot dynamically adapt to the uploaded report query.

Get Init Logs

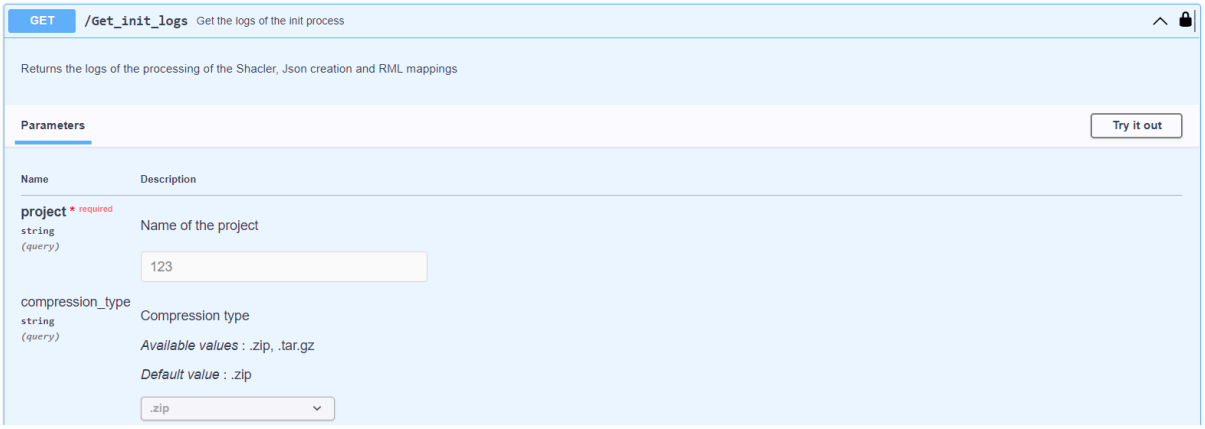
This endpoint was called in release 0.2.0 `/Get_shacler_logs` but is now renamed to `/Get_init_logs` in release 0.3.0. During the project initialization the SPHN Connector runs two processes:

1. The SHACLer generate the SHACL file based on the uploaded ontologies,
2. The script to generate RML mappings, JSON schema and DDL statements.

These processes produce two different log files which can be downloaded with this endpoint to check if the project initialisation did run correctly. The endpoint always provides both log files in a compressed .zip or .tar.gz file. From release 0.3.0 you can also specify if you like to download the logs as .zip or .tar.gz file (default is .zip). See also General Remarks.

The logs can be downloaded with this endpoint:

`/Get_Shacler_logs (project:str, [compression_type: [".zip",".tar.gz"]])`



Get Airflow logs

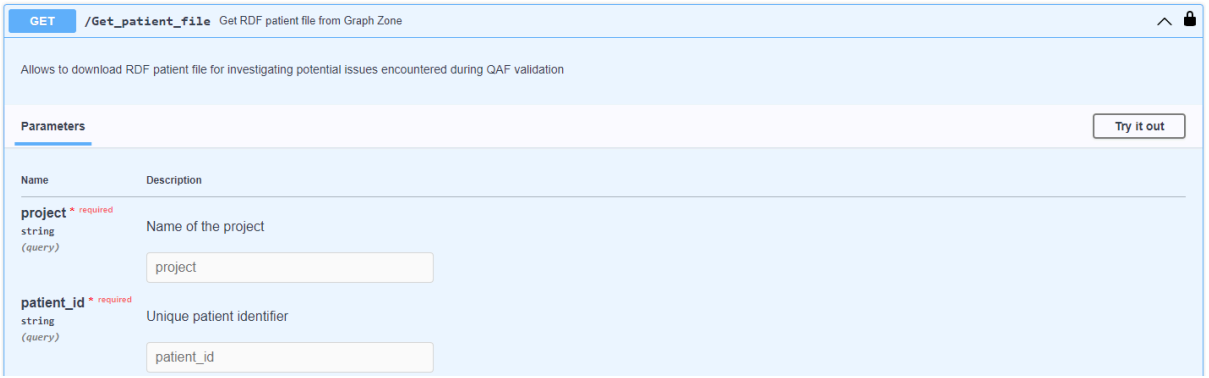
In the event of an unexpected failure of a pipeline step, it is usually useful to investigate the Airflow logs of the failed task. This could potentially give additional information about the failure cause. Nevertheless, the standard user sometimes does not have direct access to the Airflow UI and therefore they cannot access the logs. The endpoint `/Get_Airflow_logs` extracts the Airflow logs of the latest executed tasks for each DAG and returns them to the user, allowing standard users to have more insights on the failure cause. If the user is interested in a specific DAG, they can pick the DAG name from the available list.

`/Get_Airflow_logs(project: str, [DAG: ['all', 'run_shacler', 'run_sparqler', 'run_database_extraction', 'run_whole_pipeline', 'run_pre_check_and_de_identification', 'run_integration', 'run_validation', 'run_statistics', 'run_connector_backup', 'run_minio_download', 'run_real_time_count' ]]`

Get Patient File

Sometimes, it happens that the Validation step fails for a specific patient file. Usually, this is due to wrong RDF data (e.g. special characters in concepts definition). This can be deduced from the processing logs, where the stack trace of the Quality Assurance Framework reports an issue at a specific line in the RDF patient data. From the fact that this data wasn't successfully moved to the release zone, the user cannot download it via `/Download_data` endpoint. Nevertheless, this can be achieved with the endpoint `/Get_patient_file`. It requires the project name and the patient ID as usually the user is investigating an issue on a single patient file. The endpoint returns the file present in the graph zone, that is the file on which the validation was executed.

`/Get_patient_file (project: str, patient_id: str)`



Get Patients Report

This endpoint gives the user an overview of the patients' status in the SPHN Connector workflow. The user can check if there are some patients that are stuck somewhere in the pipeline and need to be manually handled. For each stage, the report gives the total number of patients, the number of processed patients (the meaning of processed depends on the stage, it could be for example ingested, transformed, downloaded, …), the number of not processed patients, and the list of the patient IDs which are marked as not processed. Example:


```
"Pre-check & De-Identification (Pre-check and De-Identification not run or failed)": {
  "total": 400,
  "processed": 400,
```

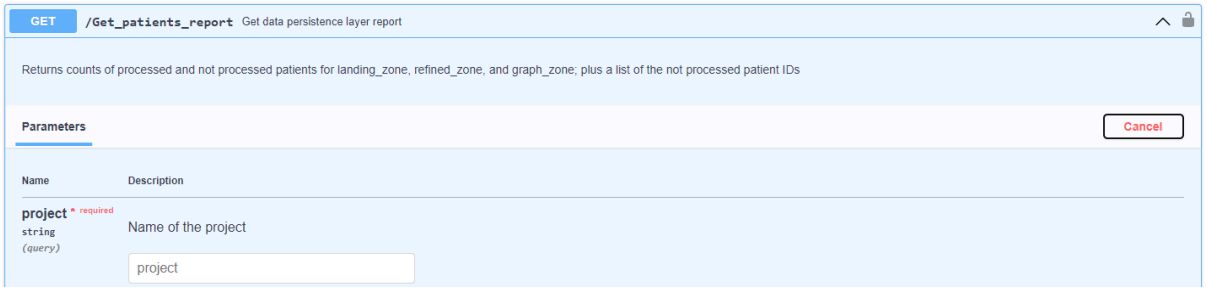
```
    "not-processed": 0,
    "not-processed-patients": []
  }
```

The report gives and overview of the following stages:

- **Pre-Release DB ingestion**: compares all the existing patient IDs in the database tables with the IDs of the patients which have been released to the `sphn_DataRelease` table. All the patients that have references in the tables but are not released yet will be marked as *not-released*.
- **Ingested into DB**: extracts all the patients that are currently released to the `sphn_DataRelease` table in the DB, but that have not yet been extracted into the SPHN Connector landing zone (i.e. `/Ingest_from_database` not yet run for those patients). The comparison is not done only with patient IDs, but also after checking changes for the `creation_time` values.
- **Pre-check & De-Identification**: gives an overview of the patients that either didn't go through the Pre-Check/De-Identification step yet, or for which the step failed.
- **Transformation**: gives an overview of the patients that either didn't go through the Integration step yet, or for which the step failed.
- **Validation**: gives an overview of the patients that either didn't go through the Validation step yet, or for which the step failed.
- **Downloaded**: gives an overview of all the patients that have been already downloaded (meaning the endpoint `/Download_data` has been triggered for those patients).

We recommend to use the /Get_patients_report to identify potential issues with the data in the different SPHN Connector steps.

`/Get_patients_report(project: str)`



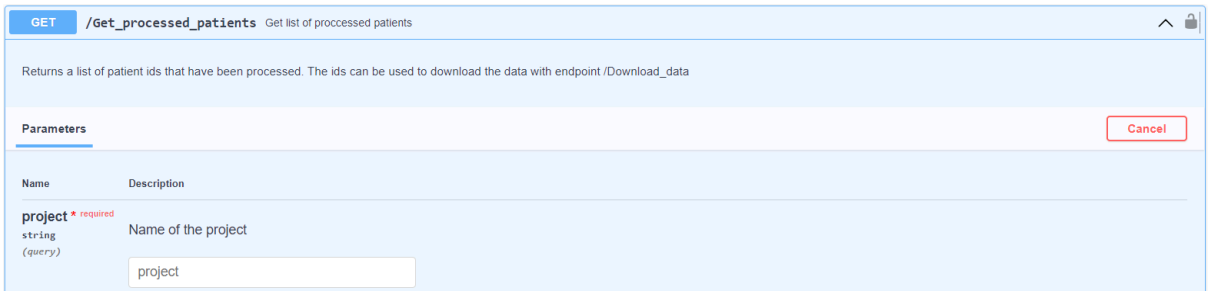
Manage and Download Data

The final step is to download the RDF turtle files. For this there are two endpoints:

Get list of processed patients

The endpoint gives a list of all patients that have been processed for a given project with their validation status OK or NOT-OK. This list can be useful when deciding which files to download.

`/Get_processed_patients (project: str)`



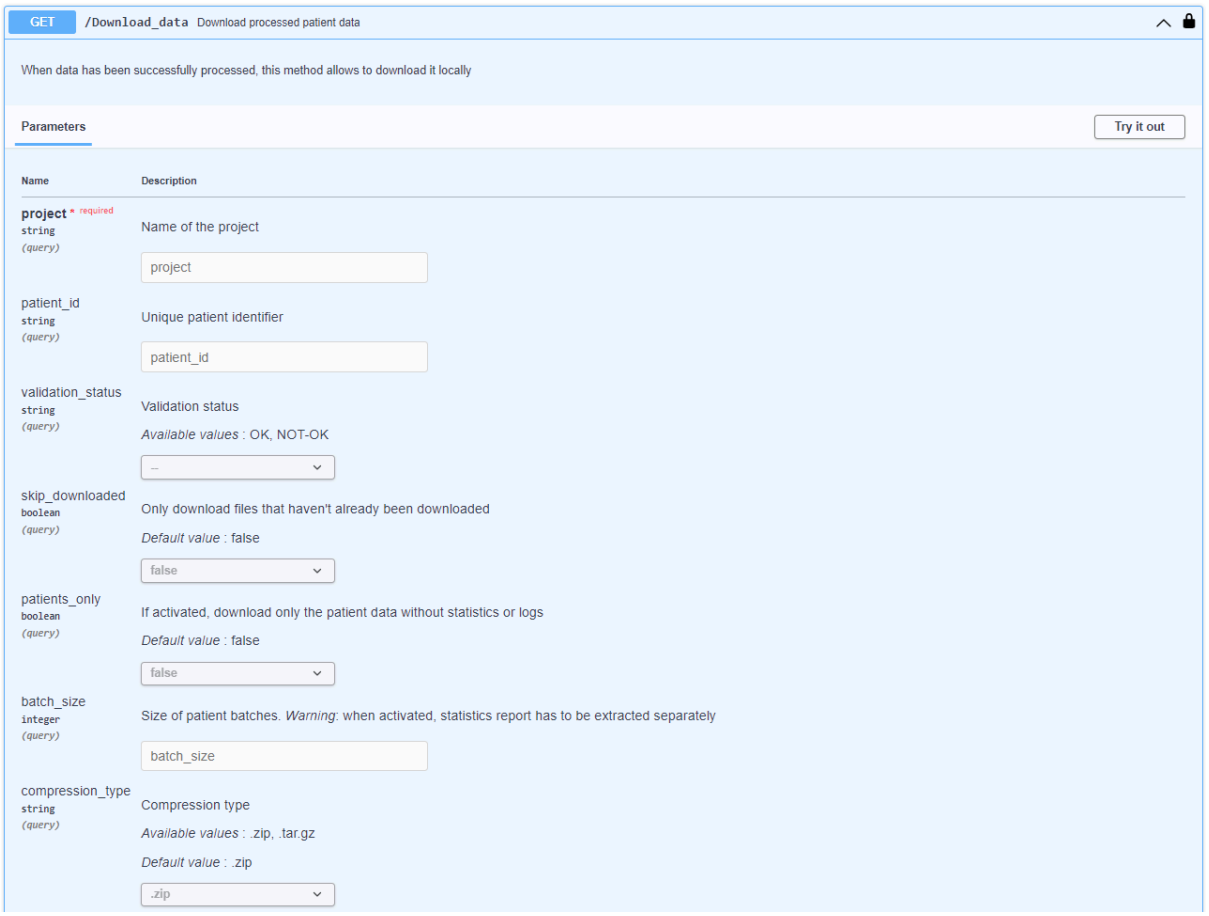
Download data

The user can download data via this endpoint, either per patient data which has been validated OK or NOT-OK, or both. By additionally specifying the patient id, the user can download the output files for a specific patient, otherwise all the patient data for the given validation status will be downloaded. Once the download of some patient files has been triggered via the endpoint, the downloaded files are marked internally with status 'downloaded'. Hence the parameter `skip_downloaded` can be enabled (`False` by default) to only download files that haven't already been downloaded.

`/Download_data (project: str, [validation_status: ["OK", "NOT-OK"]], [patient_id: str], [skip_downloaded: bool = False], [patients_only: bool = False], [batch_size: int], [compression_type: [".zip",".tar.gz"]])`

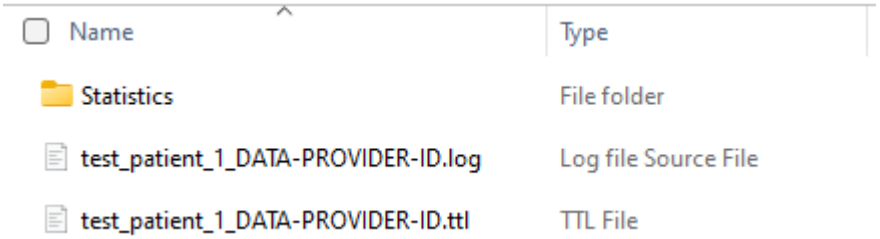
Note: *parameter* ***patients_only*** *gives the option to only download the patient data (.ttl files) without any validation log files and statistics report.*

Note*: if you omit the patient_id in the request the download will be a zip file containing all the RDF turtle files.* See also General Remarks.



The downloaded bundle contains the following files:

- Generated RDF files for the patients: `project_patientID_dataProviderID.ttl`
- Validation reports (if validation of the patient failed): `project_patientID_dataProviderID.log`
- Statistics folder with following report (only if Statistics step is run):
    - Excel report: `statistics_report.xlsx`

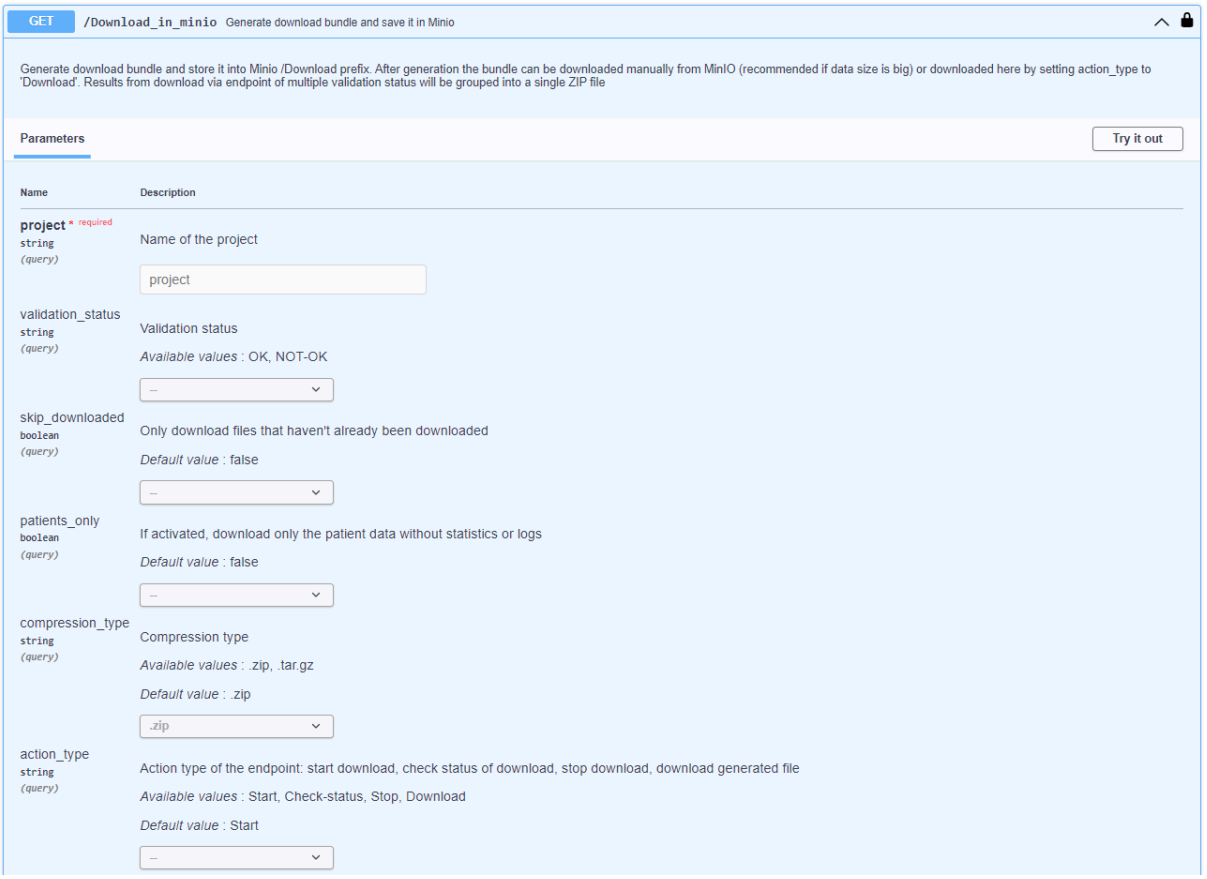


Download in batches: when the amount of data is considerable, the user could encounter issues when downloading the full bundle. We introduced the optional parameter `batch_size`, to give the user more flexibility on the size of the data downloaded. The batch size defines the number of patients that are downloaded by a single call to the `/Download_data` endpoint (it implicitly sets `skip_downloaded` to `True` such that only un-downloaded files are considered). The user can call the endpoint sequentially after setting batch size in order to mimic a batch download of the data.
Warning: when `batch_size` is set, the reports of the Statistics step are not downloaded. The user has to download them directly via `/Get_statistics_report`

Download data in Minio

When the data to download is consistent, this endpoint can be used to download it in MinIO. The process is triggered in the background, therefore the user doesn't have to wait and potentially hit a timeout.. The `/Download_in_minio` endpoint triggers in the background the generation of the download bundle and stores it in MinIO in the project's bucket under the prefix *Download/*. Depending on the validation status value passed it generates one of the following files or both: *Download/project-name_OK.ext, Download/project-name_NOT-OK.ext*, where the extension is defined by the compression type configured when triggering the endpoint. These files contain the data to download for patients with successful/unsuccessful validation status. The creation of the bundle is triggered by setting `action_type = Start`. Especially if there is a lot of data in the current project, it is recommended to download the bundle directly from the MinIO UI. Alternatively, it is also possible to download it via the endpoint by setting `action_type = Download` (if both statuses are requested, they will be zipped into a ZIP file). The other available action types can be configured to check if the download is still running or to stop a running download.

`/Download_in_minio (project: str, [validation_status: ["OK", "NOT-OK"]], [skip_downloaded: bool = False], [patients_only: bool = False], [compression_type: [".zip",".tar.gz"]], [action_type: ["Start", "Check-status", "Stop", "Download"]])`

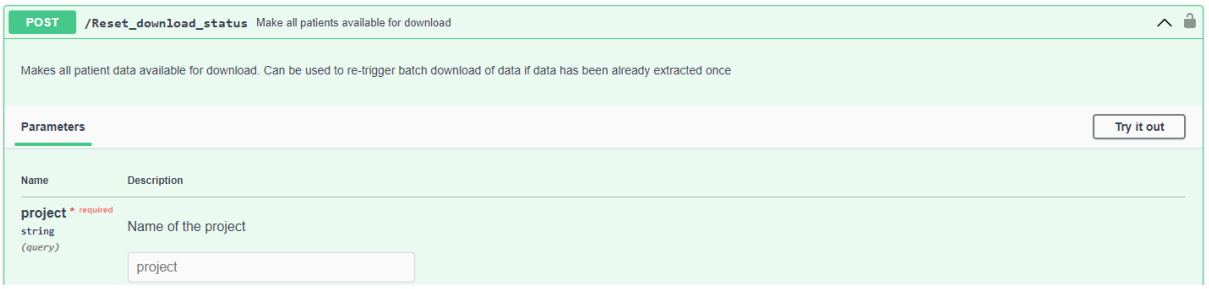


***Note:** parameter **patients_only** gives the option to only download the patient data (.ttl files) without any validation log files and statistics report.*

Reset Download Status

When triggering the `/Download_data` endpoint with `batch_size` defined or when setting the `skip_downloaded` flag, the patients that get downloaded are marked as downloaded and cannot be extracted anymore unless they are reprocessed. Especially for the batch download of data, it could be useful to retrigger the full download of the data in batches. The `/Reset_download_status` endpoint resets the download status of all the patients such that a batch download retrigger is possible. Example: after downloading all the patients in batches, no more data is available for download. Executing this endpoint once makes all the released patient data available again for download.

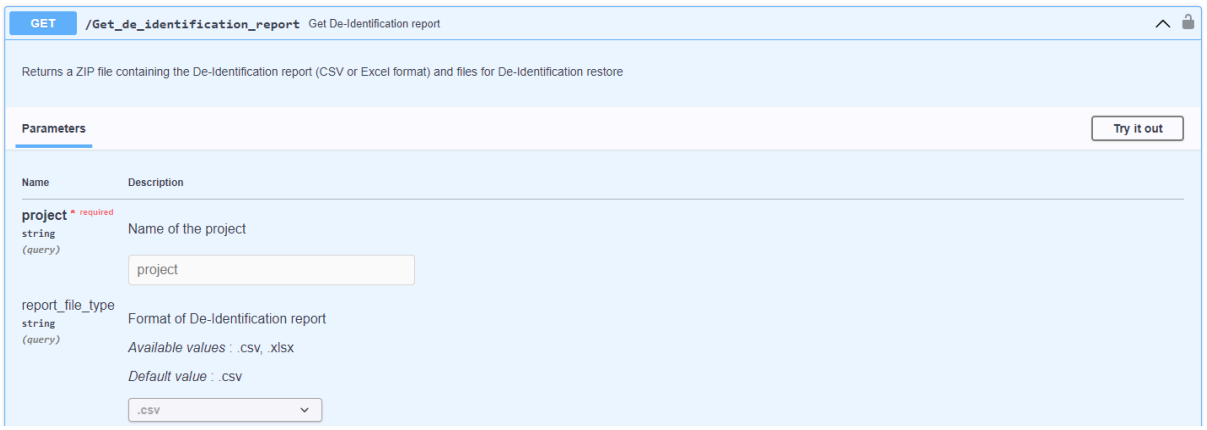
`/Reset_download_status (project: str)`



Get De-Identification Report

The user can extract the report of the De-Identification rules executed. The endpoint `/Get_de_identification_report` returns a csv/Excel file with the content of the table `de_identification` in the internal PostgreSQL database (De-Identification Report). This endpoint is accessible only by admin users. In addition to the report, the endpoint returns two files stored in the *Restore* folder. These files can be used to restore the de-identification status after re-installing the SPHN Connector or after re-creating a project. They contain database insert statements and the de-identification config file. For more information related to the de-identification restore check documentation of /Restore_de_identification endpoint.

`/Get_de_identification_report (project: str, report_file_type: ['.csv', '.xlsx'])`



Export and Import projects

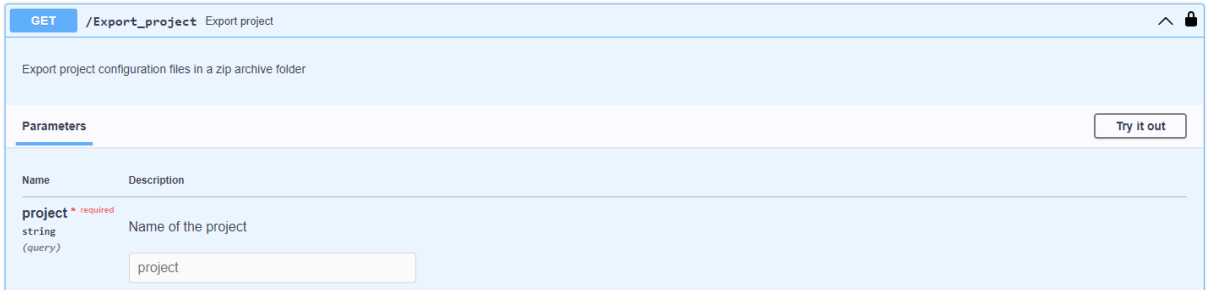
The endpoints are mainly meant for testing purposes and allow to export and re-import a project by exporting and importing its configuration data. This allows us to skip the time consuming step of uploading all the project configuration files one by one.

Note: the exported zip archive should not be modified by the user. The archive is constructed in a way that reflects the SPHN Connector object storage, therefore we can only guarantee the correct functioning of these endpoints if the uploaded zip archive comes untouched from the `/Export_project` endpoint.

Export project

The endpoint exports the project configuration files (schema, external files) into a zip archive. The main purpose of this endpoint is to save the project configuration such that it can be re-imported directly to create a new project. Only successfully initialized projects can be exported.

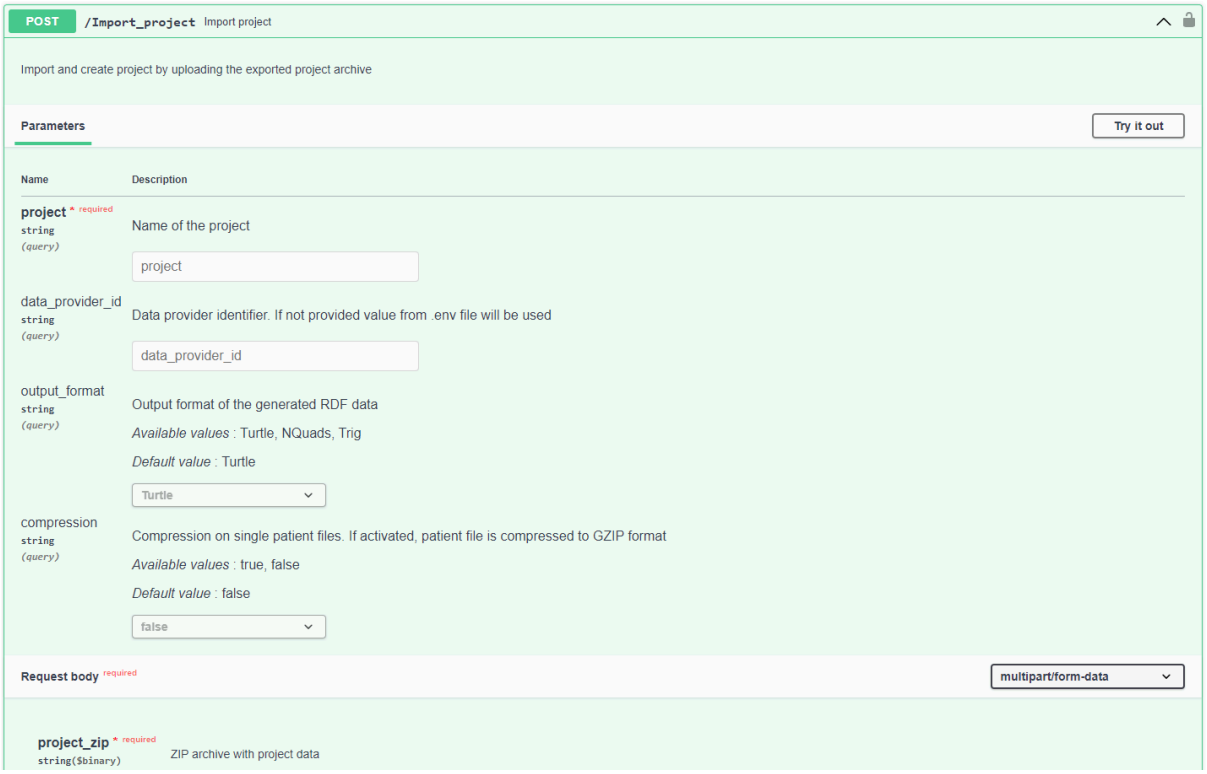
`/Export_project (project: str)`



Import project

The endpoint imports a previously exported zip archive containing the project configuration files, and creates a new project with the specified name. In addition the output format and compression of the generated data can be specified.

`/Import_project (project: str, project_zip: UploadFile, [data_provider_id: str], [output_format: [Turtle, NQuads, Trig] = Turtle] = Turtle], [compression: [True, False] = False])`



The optional output format parameter defines the format of the generated data (more details here).

Note about *data_provider_id*: when creating/importing a project the value of data_provider_id can be specified. If that's the case, the value defined in the `.env` file will be overwritten. All the loaded data should then be defined in relation to the provided value (e.g. database patient data). If no value is passed, the default value from the `.env` file is used.

Backup and Restore SPHN Connector

These endpoints can be used to save the content of the SPHN Connector (projects, data, monitoring) in case a hard rebuild of the Connector is needed (all volumes need to be dropped). The endpoints can be executed only by users with admin-level permissions.

Note: the exported zip archive should not be modified by the user. The archive is constructed in a way that reflects the SPHN Connector object storage and database schemas, therefore we can only guarantee the correct functioning of these endpoints if the uploaded zip archive comes untouched from the `/Backup_project` endpoint.

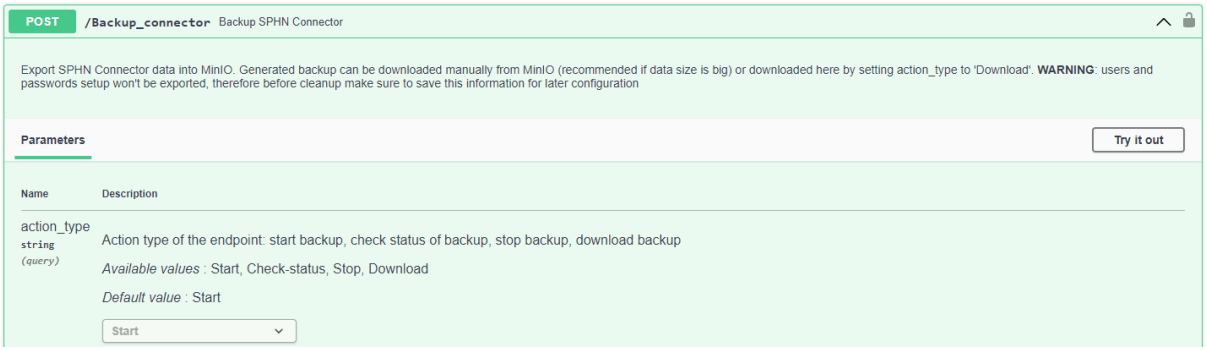
In the Installation and Requirements document you'll find a dedicated section with steps to take in order to switch version / backup the Connector called *Backup and Restore*.

Backup connector

The endpoint exports the SPHN Connector content (configuration files, generated files, patient data, database schema and content, internal tables content) into a zip archive. The main purpose of this endpoint is to save the content of the Connector if it needs to be cleaned up and rebuilt (for example due to new release). When triggered with `action_type = Start`, the backup is generated in the background and stored in a MinIO bucket named *sphn-connector-backup*. The process can be triggered only if there are no pipelines running.Before resetting the Connector, the user can download the backup directly from MinIO (recommended option especially if a lot of data is stored),

or they can set `action_type = Download` to download it via the API endpoint. The other available action types can be set to check if the backup is still running or to stop a running backup. When the backup is running, all the endpoints that change the content of the Connector are blocked and cannot be triggered.

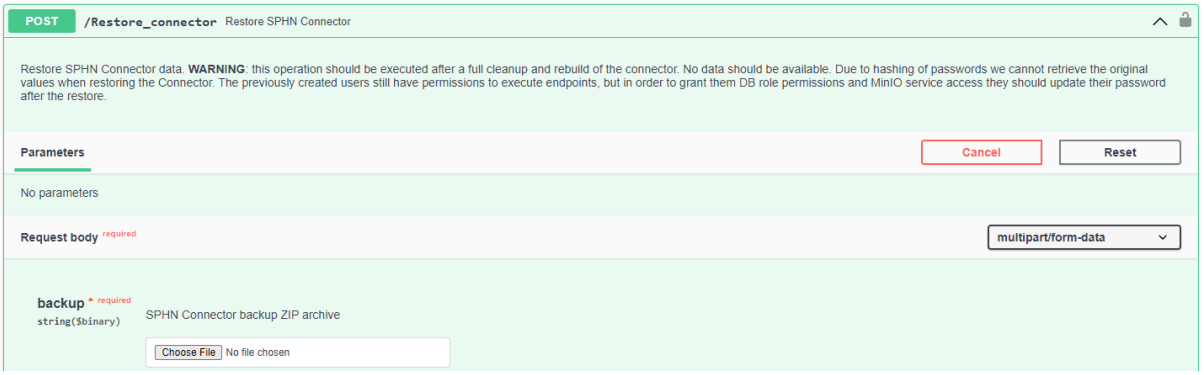
`/Backup_connector ([action_type: ["Start", "Check-status", "Stop", "Download"]])`



Restore connector

The endpoint restores the SPHN Connector content (configuration files, generated files, patient data, database schema and content, internal tables content) by loading a zip archive exported by the `/Backup_connector` endpoint. The main purpose of this endpoint is to reload the content of the Connector after a complete rebuild.

`/Restore_connector (backup: UploadFile)`



Note: the created users will also be reimported, but due to hashing of password, once the connector is restored, the hashed values will be used to create those accesses. The users will still be able to access the API endpoints with the original value, though it is recommended to reset the user password such that the DB roles and MinIO access will be set up properly with the original password value.

Backup and Restore de-identification

Especially when the SPHN Connector is re-installed after a new release or when a project needs to be re-created for some reasons, it could be necessary to restore the status of the de-identification. A restore will enforce that the de-identification rules are applied the same way as before the reset on the patients that have already been processed.

Backup De-Identification

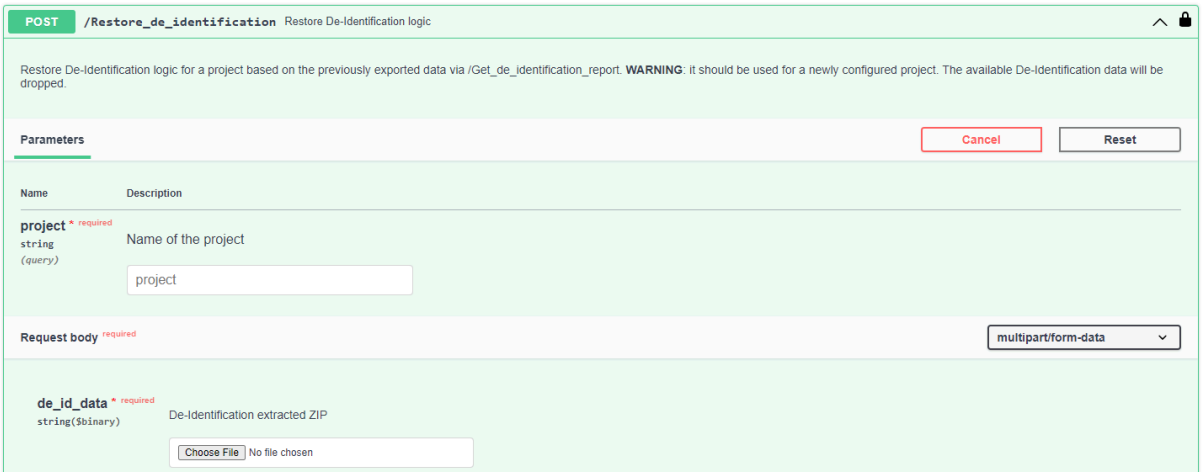
For a successful restore, it is essential that the admin user extracts the de-identification data before resetting the project/Connector via the endpoint /Get_de_identification_report. The folder *Restore* in the extracted archive contains the following files:

These files contain the SQL insert statements to re-ingest the de-identification tracked data of the patients already processed, and the de-identification configuration file.

Restore De-Identification

To restore the de-identification status of an existing project (best if newly created), the user needs to pass the extracted zip file to the endpoint `/Restore_de_identification`.

`/Restore_de_identification(project: str, de_id_data: zip)`



WARNING: the restore of de-identification is mainly meant for restoring the status on a brand new project before ingesting any data into it. Be aware that if the restored project already has some processed data, this data will be dropped together with the current de-identification results (previously processed patients will be considered as new patients, e.g. new date shift configured), logging, statistics, and so on. Basically we will have a fresh start with only the data in the landing zone and the new de-identification logic.

Testing and Implementation

For the productive implementation it is suggested to use an ETL Tool that supports posts to API endpoints and ingesting data into databases. Alternatively a dedicated Python program could also be developed.

The repository https://git.dcc.sib.swiss/hospfair/sphn-connector-integration will help you with the integration of the SPHN Connector. If you develop something different or use another programming language than Python you are welcome to upload your own solution.

For testing purposes you could either use one of the three options:

- Use the FastAPI manually, however, this is only recommended for limited test cases if you want to test something specific, maybe related to an issue you have.
- Use command-line curl statements as they are documented as examples in the FastAPI browser front-end.
- Postman or similar solution that allows you to set up REST requests and send it to the SPHN Connector. In this case keep in mind that you will have to first retrieve the JWT that you need to set in the request header.
- Python test framework which we developed for running automated test cases for each release. You can find this in the repository https://git.dcc.sib.swiss/hospfair/sphn-connector-user-testing

JSON data example

In our CI/CD pipeline we use some test JSON data to test the SPHN Connector pipeline. It contains some random data and represents two patients, one validating OK and another validating NOT-OK.

Successful validation

Mock data of a patient validating successfully against the Quality Assurance Framework


```
{
    "id": "https://biomedit.ch/rdf/sphn-schema/sphn/2024/2.json",
    "schema": "https://biomedit.ch/rdf/sphn-schema/sphn/2024/2",
    "title": "PatientRecord",
    "targetType": "RDF",
    "license": "",
    "sphn:DataRelease": {
        "id": "1666216800",
        "creationTime": "2022-10-20T12:00:00.000"
    },
    "sphn:DataProvider": {
        "id": "DATA-PROVIDER-ID",
        "sphn:hasInstitutionCode": {
            "id": "test_data_provider",
            "sphn:hasCodingSystemAndVersion": "UID",
            "sphn:hasIdentifier": "DGG_346_346_233"
        },
        "sphn:hasCategory": {
            "iri": "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Hospital"
        },
        "sphn:hasDepartment": {
            "id": "department01",
```

```
            "sphn:hasName": "department01Name"
        }
    },
    "sphn:SubjectPseudoIdentifier": {
        "id": "patient_1",
        "sphn:hasIdentifier": "patient_1"
    },
    "content": {
        "sphn:Allergy": [
            {
                "id": "allergy01",
                "sphn:hasFirstRecordDateTime": "2022-04-24T11:55:43.304Z",
                "sphn:hasLastReactionDateTime": "2022-03-31T11:55:43.304Z",
                "sphn:hasAllergen": {
                    "id": "allergen01",
                    "sphn:hasCode": [
                        {
                            "id": "TestInternal-test_pf4n3cke3",
                            "sphn:hasCodingSystemAndVersion": "TEST Internal-TEST",
                            "sphn:hasIdentifier": "P4dgneKf",
                            "sphn:hasName": "PF4N3Cke3"
                        }
                    ]
                },
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]
            }
        ],
        "sphn:SourceSystem": [
            {
                "id": "sourceSystem01",
                "sphn:hasName": "sourceSystem01Name"
            }
        ]
    },
    "supporting_concepts": {
        "sphn:Interpretation": [
            {
                "id": "allergyInterpretation",
                "sphn:hasOutput": [
                    {
                        "id": "allergy01",
                        "target_concept":
"https://biomedit.ch/rdf/sphn-schema/sphn#Allergy"
                    },
                    {
                        "id": "TestInternal-test_pf4n3cke3",
                        "sourceConceptType": "sphn-Allergen",
                        "sourceConceptID": "allergen01",
                        "target_concept":
"https://biomedit.ch/rdf/sphn-schema/sphn#Code"

                    }
```

```
                ],
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]

            },
            {
                "id": "sourceSystemInterpretation",
                "sphn:hasOutput": [
                    {
                        "id": "sourceSystem01",
                        "target_concept":
"https://biomedit.ch/rdf/sphn-schema/sphn#SourceSystem"
                    }
                ],
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]

            }
        ],
        "sphn:SourceData": [
            {
                "id": "sourceData01",
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]
            }
        ],
        "sphn:SemanticMapping": [
            {
                "id": "semanticMapping01",
                "sphn:hasOutputCode": {
                    "id": "TestInternal-test_pf4n3cke3",
                    "sourceConceptType": "sphn-Allergen",
                    "sourceConceptID": "allergen01"
                },
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]
            }
        ]
    }
}
```


Failed validation

Mock patient data with failing validation due to wrong date-time format

```
{
    "id": "https://biomedit.ch/rdf/sphn-schema/sphn/2024/2.json",
    "schema": "https://biomedit.ch/rdf/sphn-schema/sphn/2024/2",
    "title": "PatientRecord",
    "targetType": "RDF",
    "license": "",
    "sphn:DataRelease":{
        "id": "1666216800",
        "creationTime": "2022-10-20T12:00:00.000"
    },
    "sphn:DataProvider": {
        "id" : "DATA-PROVIDER-ID",
        "sphn:hasInstitutionCode" : {
            "id" : "test_data_provider",
            "sphn:hasCodingSystemAndVersion" : "UID",
            "sphn:hasIdentifier" : "DGG_346_346_233"
            },
        "sphn:hasCategory": {
            "iri": "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Hospital"
        },
        "sphn:hasDepartment": {
            "id": "department01",
            "sphn:hasName": "department01Name"
        }
    },
    "sphn:SubjectPseudoIdentifier": {
        "id" : "patient_2",
        "sphn:hasIdentifier" : "patient_2"
    },
    "content": {
            "sphn:Allergy": [
                {
                "id" : "allergy01",
                "sphn:hasFirstRecordDateTime": "2022-04-24T11:55:43.304Z",
                "sphn:hasLastReactionDateTime": "2022-03-31",
                "sphn:hasAllergen": {
                    "id": "allergen01",
                    "sphn:hasCode": [
                        {
                            "id": "TestInternal-test_pf4n3cke3",
                            "sphn:hasCodingSystemAndVersion": "TEST Internal-TEST",
                            "sphn:hasIdentifier": "P4dgneKf",
                            "sphn:hasName": "PF4N3Cke3"
                        }
                    ]
                },
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]
            }
        ],
        "sphn:SourceSystem": [
            {
                "id": "sourceSystem01",
```

```
                "sphn:hasName": "sourceSystem01Name"
            }
        ]
    },
    "supporting_concepts": {
        "sphn:Interpretation": [
            {
                "id": "allergyInterpretation",
                "sphn:hasOutput": [
                    {
                        "id": "allergy01",
                        "target_concept":
"https://biomedit.ch/rdf/sphn-schema/sphn#Allergy"
                    },
                    {
                        "id": "TestInternal-test_pf4n3cke3",
                        "sourceConceptType": "sphn-Allergen",
                        "sourceConceptID": "allergen01",
                        "target_concept":
"https://biomedit.ch/rdf/sphn-schema/sphn#Code"

                    }
                ],
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]

            },
            {
                "id": "sourceSystemInterpretation",
                "sphn:hasOutput": [
                    {
                        "id": "sourceSystem01",
                        "target_concept":
"https://biomedit.ch/rdf/sphn-schema/sphn#SourceSystem"
                    }
                ],
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]

            }
        ],
        "sphn:SourceData": [
            {
                "id": "sourceData01",
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]
            }
```

```
        ],
        "sphn:SemanticMapping": [
            {
                "id": "semanticMapping01",
                "sphn:hasOutputCode": {
                    "id": "TestInternal-test_pf4n3cke3",
                    "sourceConceptType": "sphn-Allergen",
                    "sourceConceptID": "allergen01"
                },
                "sphn:hasSourceSystem": [
                    {
                        "id": "sourceSystem01"
                    }
                ]
            }
        ]
    }

}
```

Database data example

In our CI/CD pipeline we use some test SQL inserts to ingest mock data into the database tables to test the SPHN Connector pipeline. It contains some random data and represents two patients, one validating OK and another validating NOT-OK.

Successful validation

Mock data of a patient validating successfully against the Quality Assurance Framework

```
INSERT INTO "test-project"."sphn_DataRelease" (
    "sphn_hasSubjectPseudoIdentifier",
    "sphn_hasDataProvider",
    "id",
    "creationTime"
    )
    VALUES (
        'db_ok',
        'DATA-PROVIDER-ID',
        '1666216800',
        '2022-10-20T12:00:00.000'
        );

INSERT INTO "test-project"."sphn_SubjectPseudoIdentifier" (
        "id",
    "sphn_hasIdentifier"
    )
    VALUES (
        'db_ok',
        'db_ok'
        );

INSERT INTO "test-project"."sphn_DataProvider" (
        "id",
```

```
    "sphn_hasInstitutionCode__id",
    "sphn_hasInstitutionCode__sphn_hasIdentifier",
    "sphn_hasInstitutionCode__sphn_hasCodingSystemAndVersion",
    "sphn_hasCategory__iri",
    "sphn_hasDepartment__id",
    "sphn_hasDepartment__sphn_hasName"
    )
    VALUES (
        'DATA-PROVIDER-ID',
        'test_data_provider',
        'DGG_346_346_233',
        'UID',
        'https://biomedit.ch/rdf/sphn-schema/sphn/individual#Hospital',
        'department01',
        'department01Name'
        );

INSERT INTO "test-project"."sphn_Allergy" (
        "sphn_hasSubjectPseudoIdentifier",
    "sphn_hasDataProvider",
    "id",
    "sphn_hasLastReactionDateTime",
    "sphn_hasFirstRecordDateTime",
    "sphn_hasAllergen__id",
    "sphn_hasAllergen__sphn_hasCode__id",
    "sphn_hasAllergen__sphn_hasCode__sphn_hasCodingSystemAndVersion",
    "sphn_hasAllergen__sphn_hasCode__sphn_hasIdentifier",
    "sphn_hasAllergen__sphn_hasCode__sphn_hasName",
    "sphn_hasSourceSystem__id"
    )
        VALUES (
         'db_ok',
         'DATA-PROVIDER-ID',
         'allergy01',
         '2022-03-31T11:55:43.304Z',
         '2022-04-24T11:55:43.304Z',
         'allergen01',
         'TestInternal-test_pf4n3cke3',
         'TEST Internal-TEST',
         'P4dgneKf',
         'PF4N3Cke3',
         'sourceSystem01'
         );

INSERT INTO "test-project"."sphn_SourceSystem"(
        "id",
    "patient_id",
    "sphn_hasDataProvider",
    "sphn_hasName"
    )
        VALUES (
         'sourceSystem01',
         'db_ok',
         'DATA-PROVIDER-ID',
         'sourceSystem01Name'
    );
```

```
INSERT INTO "test-project"."supporting__sphn_Interpretation" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutput__Allergy__id",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'allergyInterpretation',
        'DATA-PROVIDER-ID',
        'db_ok',
        'allergy01',
        'sourceSystem01'
    );

INSERT INTO "test-project"."supporting__sphn_Interpretation" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutput__Code__id",
    "sphn_hasOutput__Code__sourceConceptType",
    "sphn_hasOutput__Code__sourceConceptID",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'allergyInterpretation',
        'DATA-PROVIDER-ID',
        'db_ok',
        'TestInternal-test_pf4n3cke3',
        'sphn-Allergen',
        'allergen01',
        'sourceSystem01'
    );

INSERT INTO "test-project"."supporting__sphn_Interpretation" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutput__SourceSystem__id",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'sourceSystemInterpretation',
        'DATA-PROVIDER-ID',
        'db_ok',
        'sourceSystem01',
        'sourceSystem01'
    );
INSERT INTO "test-project"."supporting__sphn_SemanticMapping" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutputCode__id",
    "sphn_hasOutputCode__sourceConceptType",
    "sphn_hasOutputCode__sourceConceptID",
```

```
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'semanticMapping01',
        'DATA-PROVIDER-ID',
        'db_ok',
        'TestInternal-test_pf4n3cke3',
        'sphn-Allergen',
        'allergen01',
        'sourceSystem01'
    );

INSERT INTO "test-project"."supporting__sphn_SourceData" (
    "id",
    "patient_id",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'sourceData01',
        'db_ok',
        'sourceSystem01'
    );
```

Failed validation

Mock patient data with failing validation due to missing required unit

```
INSERT INTO "test-project"."sphn_DataRelease" (
    "sphn_hasSubjectPseudoIdentifier",
    "sphn_hasDataProvider",
    "id",
    "creationTime"
    )
    VALUES (
        'db_not_ok',
        'DATA-PROVIDER-ID',
        '1666216800',
        '2022-10-20T12:00:00.000'
        );

INSERT INTO "test-project"."sphn_SubjectPseudoIdentifier" (
       "id",
    "sphn_hasIdentifier"
    )
    VALUES (
        'db_not_ok',
        'db_not_ok'
        );

INSERT INTO "test-project"."sphn_Allergy" (
       "sphn_hasSubjectPseudoIdentifier",
    "sphn_hasDataProvider",
    "id",
    "sphn_hasLastReactionDateTime",
    "sphn_hasFirstRecordDateTime",
    "sphn_hasAllergen__id",
```

```
    "sphn_hasAllergen__sphn_hasCode__id",
    "sphn_hasSourceSystem__id"
    )
        VALUES (
         'db_not_ok',
         'DATA-PROVIDER-ID',
         'allergy01',
         '2022-03-31T11:55:43.304Z',
         '2022-04-24T11:55:43.304Z',
         'allergen01',
         'TestInternal-test_pf4n3cke3',
         'sourceSystem01'
         );

INSERT INTO "test-project"."sphn_SourceSystem"(
        "id",
    "patient_id",
    "sphn_hasDataProvider",
    "sphn_hasName"
    )
        VALUES (
         'sourceSystem01',
         'db_not_ok',
         'DATA-PROVIDER-ID',
         'sourceSystem01Name'
    );

INSERT INTO "test-project"."supporting__sphn_Interpretation" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutput__Allergy__id",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
         'allergyInterpretation',
         'DATA-PROVIDER-ID',
         'db_not_ok',
         'allergy01',
         'sourceSystem01'
    );

INSERT INTO "test-project"."supporting__sphn_Interpretation" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutput__Code__id",
    "sphn_hasOutput__Code__sourceConceptType",
    "sphn_hasOutput__Code__sourceConceptID",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
         'allergyInterpretation',
         'DATA-PROVIDER-ID',
         'db_not_ok',
         'TestInternal-test_pf4n3cke3',
```

```
        'sphn-Allergen',
        'allergen01',
        'sourceSystem01'
    );

INSERT INTO "test-project"."supporting__sphn_Interpretation" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutput__SourceSystem__id",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'sourceSystemInterpretation',
        'DATA-PROVIDER-ID',
        'db_not_ok',
        'sourceSystem01',
        'sourceSystem01'
    );
INSERT INTO "test-project"."supporting__sphn_SemanticMapping" (
    "id",
    "sphn_hasDataProvider",
    "patient_id",
    "sphn_hasOutputCode__id",
    "sphn_hasOutputCode__sourceConceptType",
    "sphn_hasOutputCode__sourceConceptID",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'semanticMapping01',
        'DATA-PROVIDER-ID',
        'db_not_ok',
        'TestInternal-test_pf4n3cke3',
        'sphn-Allergen',
        'allergen01',
        'sourceSystem01'
    );

INSERT INTO "test-project"."supporting__sphn_SourceData" (
    "id",
    "patient_id",
    "sphn_hasSourceSystem__id"
    )
    VALUES (
        'sourceData01',
        'db_not_ok',
        'sourceSystem01'
    );
```

Troubleshooting

Introduction

When working with the SPHN-Connector there might be situations where the SPHN-Connector behaves differently than expected by the user. This section covers methods to retrieve further information on problems and the associated services you can access to further analyze what went wrong.

Logging information

The easiest way to identify issues is via the logs provided directly by the SPHN Connector. The API offers multiple endpoints to access logging information over different steps. For logging information for a single, multiple or all patients you can use the /Get_logs endpoint. For logging information about the entire project you'd use the /Get_global_logs and for information about the initialization you'd use the /Get_init_logs endpoint. Additionally you are able to retrieve the status of a project via the /Get_status endpoint. Endpoint /Get_patients_with_errors allows to get a list of patients with errors for a specific severity level and step. Endpoint /Get_patient_file allows the user to download the generated RDF file before it is moved to the release zone. This could be helpful to investigate issues during the validation step.

Access Services directly

There might be cases in which the logging information is not enough to meet your demands or you want to dig deeper into the underlying issues. Depending on the information needed, different services might be of interest to you. The accessible services that are open for (admin) users are: Minio, postgres via pgadmin and Airflow.

Minio

Minio is the object storage of the SPHN Connector. You'll find the different projects you created as buckets in Minio. Inside of the buckets projects you'll find files like schemas, external terminologies etc as well as patient files. This is dependent on the state of your project (Is it initialized yet? Have patient files been uploaded? Have files already been processed?). In case there is something wrong with your patient files or your schema files and you want to have a look at it you can do so via the Minio UI.

You can access Minio directly via port 1443 at location /minio/: https://{your-server}:1443/minio/. You'll find further information on how to access / use Minio in the Appendix: Access minio

Warning:
changes you introduce directly via Minio might lead to inconsistencies with the Connector and it is not advisable to do so.

Pgadmin

pgadmin offers a handy interface to access data stored in the postgres database. Currently many logging steps, airflow statistics as well as the patient data are stored in separate databases.
You can access pgadmin directly via port 1443 at location /pgadmin: https://{your-server}:1443/pgadmin. You'll find further information on how to access the postgres tables via pgadmin in the Appendix: Access pgadmin

Warning:
changes you introduce directly via pgadmin might lead to inconsistencies with the Connector and it is not advisable to do so.

Airflow

Airflow is the scheduling / pipeline service that is used by the Connector. If you want to debug where the error occurred you might want to check out the airflow UI.
You can access airflow directly via port 1443 at location /airflow: https://{your-server}:1443/airflow . You'll find further information on how to access the logs / process overview in Airflow in the Appendix: Access airflow or in the visual user guide: SPHN-Connector_airflow-ui.

Grafana

Airflow is an open source analytics and interactive visualization web application. For the SPHN Connector Grafana is shipped with predefined postgres connections and dashboards.
You can access grafana directly via port 1443 at location /grafana/: https://{your-server}:1443/grafana .
The Grafana service is shipped with five dashboards:

Folder	Name	Description
Overview	Connector instance overview	High level view on the running Connector instanced
Overview	Logging	Access to the logs created by SPHN

Folder	Name	Description
		Connector
Overview	Quality Checks Overview	Access to the output of the quality checks
performance	Performance	Visual analysis for the loadtesting tool (will be empty if loadtesting has not been executed)
statistics	SPHN Statistics	Access to the output for the Connector statistics (will be empty if statistics have not been executed)

You'll find further information on how to access & interact with Grafana in the Appendix: Access grafana or in the visual user guide: SPHN-Connector_grafana.

Docker

The Docker CLI is a useful tool to determine e. g. the status of the different services (check if they are up and running). The table below lists popular / basic commands to interact with a running installation. If you need more details on how to determine if everything is up and running please have a look at the section: *Troubleshooting and additional help for validation of installation* in the **installation guide**

Command	Output description
docker ps	Shows active containers
docker logs [**container_name**]	Shows logs of a specific container
docker logs -n 5 [**container_name**]	Prints last 5 lines of a log
docker compose down --rmi all	Only containers and image will be removed

docker compose down –rmi all –volumes	All containers, images and volumes will be deleted
docker volume ls	Lists all volumes
docker volume inspect **volume_name**	Provides additional information about a specific / multiple volumes

JSON schema, RML mapping and DDL generation

Introduction

The aim of this section is to discuss the RML mapping generation for the conversion of JSON files to RDF files, the JSON schema generation, and the DDL generation. We want to focus on special edge cases and how those are handled and reflected into the generated files.

The generation logic for JSON schema and RML is defined by `lib/rml_generator.py` and `lib/rml_generator_lib.py`, while the generation logic for DDL is defined in `lib/database.py` in SPHN Connector GitLab repository. The section is up to date with the RDF schema 2024. In the appendix section SPHN Connector for 2023.2 RDF schema we list special cases that are relevant for RDF schemas prior to 2024.

Starting from 2024, the new SPHN RDF schema introduces some new patterns in the generation of the JSON schema, RML mapping file, and database schema. Please checkout the section 2024 RDF schema in order to adapt to the new schema.

Generation logic

After loading the project specific schema and the external schemas into a `rdflib` graph, the Connector extracts all the relevant concepts which can be separated into several types:

- **Core concepts**: schema classes (`OWL.Class`) that define the object property `hasSubjectPseudoIdentifier`. In addition to those, `SourceSystem` is also considered as core concepts.
- **Special concepts**: `SubjectPseudoIdentifier` and `DataProvider`.
- **Supporting concepts**: concepts that are not core concepts and that are not referenced directly/indirectly inside core concepts.

Each one of those core concepts is then processed recursively by checking its datatype properties and object properties. Two different logics are used to construct the RML mapping file and the JSON schema, but both of them start from the core concept and recursively process it down to all the datatype properties. For example, a datatype property of a core concept is directly mapped, while an object property is processed to the next nesting level, meaning that all datatype properties and object properties of that core concept's object property are in turn processed.

The JSON schema does not allow additional properties than those defined. Moreover, fields `sphn:DataProvider`, `sphn:SubjectPseudoIdentifier`, `sphn:DataRelease`, and `content` are mandatory. The first describes the data provider institute, the second describes the patient, the third is used to define the `sphn:DataRelease` triple in the converted RDF file, and the last one is the list of SPHN core concepts. The field `supporting_concepts` is not required.

Once the JSON schema has been generated, it is used to create the database schema.

Naming convention

Most SPHN concepts define an `id` field in the schema. The purpose of it is to map, during the RML mapping, the quantities defined on different levels of nesting in the schema. Moreover, its value is used to construct the name of the unique resource. SPHN Connector logic is based on the Naming conventions, a unique resource is instantiated with name:

`resource:<provider_id>-<prefix>-<ClassName>-<unique_id>.` Here, the `unique_id` gets the value of the `id` field defined in the patient data. The `ClassName` and `prefix` value depends on the class of the SPHN concepts which is being instantiated, while the value of the `provider_id` is taken from the `.env` file of the SPHN Connector configuration.

Naming convention prior to 2024.1 RDF schema:

- Unique resource:
  `resource:<provider_id>-<prefix>-<ClassName>-<unique_id>`
- Shared resource: **`resource:sphn-Code-<unique_id>`**

Naming convention from 2024.1 RDF schema:

- Unique resource (not Code):
  `resource:<provider_id>-<prefix>-<ClassName>-<unique_id>`
- Code/Terminology:
  `resource:<data_provider_id>-<prefix-of-class-where-used>-<name-of-class-where-used>-<id-of-class-where-used>-sphn-Code-<id-of-the-code>` (for more details here).

Restrictions

We propagate the available restrictions to the JSON schema based on the RDF schema. The most efficient technique we are using to extract restrictions at the moment is to collect the full path of the transverse graph and store the associated values set, if there is any. Then when building the JSON schema we can check on the restriction map and extract the restricted values.

JSON schema

We take as example the `sphn:Allergy` core concept. From the fact, the RML mapping is constructed on the JSON data structure, we start by describing how the JSON schema generation works. The most important feature of the JSON schema is the nesting.The datatype properties of a core concept are directly reflected in the properties of the core concept schema.
Every object property has a mandatory additional field defined by the `id` key. This field is a unique identifier of the concept/property described. The top-level property `content` is a map which defines each SPHN concept as a list of objects with a single schema.

```
{
    "sphn:Allergy": {
        "type": "array",
        "description": "List of 'sphn:Allergy' concepts",
        "items": {
                "type": "object",
                "additionalProperties": false,
                "description": "SPHN Concept 'Allergy'",
                "properties": {
                    "id": {
                        "description": "ID of SPHN Concept 'Allergy'",
```

```
                                "type": "string"
                            },
                        "sphn:hasFirstRecordDateTime": {
                            "description": "Value for 'hasFirstRecordDateTime'
property",
                            "type": "string",
                            "format": "date-time"
                        },
                        ...
                    }
                }
    }
}
```

In this example, we see the `id` field defined, and the datatype property `sphn:hasFirstRecordDateTime` which is directly added at the top layer of the properties.
As said, the nesting is a main characteristic of the JSON schema, especially of the object properties. For example, the object property `sphn:hasVerificationStatusCode` is of type `Terminology`, and therefore the schema structure of that class is reported directly under the `sphn:hasVerificationStatusCode` property.

```
{
    "sphn:Allergy": {
        "type": "array",
        "description": "List of 'sphn:Allergy' concepts",
        "items": {
        "type": "object",
        "description": "SPHN Concept 'Allergy'",
        "properties": {
            "sphn:hasVerificationStatusCode": {
                "type": "object",
                "description": "SPHN Concept 'Terminology'",
                "properties": {
                    "termid": {
                        "type": "string",
                        "description": "Unique ID for the given IRI. Its format must
follow the convention: <coding_system>-<identifier>"
                    },
                    "iri": {
                        "type": "string",
                        "description": "IRI of SPHN Concept 'Terminology'"
                    },
                    ...
                },
                "required": [
                    "termid",
                    "iri"
                ],
                "additionalProperties": false
```

```
            }
        }
    }
    }
}
```

The nesting can go down to several levels, depending on how many object properties nesting exist until only datatype properties are found.

RML mapping

We take the result of the previous section as an example. We describe here the RML mapping generated for the `sphn:Allergy`. Every datatype property is referenced directly pointing to the value described in the JSON data. Every object property is defined by accessing the `id` field defined on the next level of nesting. Exceptions are properties `sphn:hasSubjectPseudoIdentifier` and `sphn:hasDataProvider` which are defined for the patient at the top level of the JSON schema. These properties are joined via the unique identifier `id` with the RML mapping defining these properties (`:sphnSubjectPseudoIdentifier, :sphnDataProvider`). Every concept (defined on every level) is defined by a specific RML mapping rule which maps it from the JSON structure to the RDF structure. The mapping rules names are built with the transverse path in the JSON schema.

```
:sphnAllergy a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.content.sphn:Allergy[*]" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:parentTriplesMap :sphnDataProvider ] ;
            rr:predicate sphn:hasDataProvider ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-SourceSystem-{sphn:hasSourc
eSystem[*].id}" ] ;
            rr:predicate sphn:hasSourceSystem ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Allergy-{id}-sphn-Code-{sph
n:hasVerificationStatusCode.termid}" ] ;
            rr:predicate sphn:hasVerificationStatusCode ],
        [ rr:objectMap [ rml:reference "sphn:hasLastReactionDateTime" ;
                    rr:datatype xsd:dateTime ] ;
            rr:predicate sphn:hasLastReactionDateTime ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Allergen-{sphn:hasAllergen.
id}" ] ;
            rr:predicate sphn:hasAllergen ],
        [ rr:objectMap [ rr:parentTriplesMap :sphnSubjectPseudoIdentifier ] ;
            rr:predicate sphn:hasSubjectPseudoIdentifier ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Allergy-{id}-sphn-Code-{sph
n:hasSeverityCode.termid}" ] ;
            rr:predicate sphn:hasSeverityCode ],
        [ rr:objectMap [ rml:reference "sphn:hasFirstRecordDateTime" ;
                    rr:datatype xsd:dateTime ] ;
            rr:predicate sphn:hasFirstRecordDateTime ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Allergy-{id}-sphn-Code-{sph
n:hasReactionTypeCode.termid}" ] ;
            rr:predicate sphn:hasReactionTypeCode ] ;
    rr:subjectMap [ rr:class sphn:Allergy ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Allergy-{id}" ] .
```

The property `sphn:hasFirstRecordDateTime` is reported directly with an object map referencing to the same key in the JSON data and defined with the expected type:

```
[ rr:objectMap [ rml:reference "sphn:hasFirstRecordDateTime" ;
                    rr:datatype xsd:dateTime ] ;
            rr:predicate sphn:hasFirstRecordDateTime ]
```

Object property `sphn:hasVerificationStatusCode` is defined by accessing the `termid` field on the lower level and the `id` field of the allergy:

```
[ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Allergy-{id}-sphn-Code-{sph
n:hasVerificationStatusCode.termid}" ] ;
            rr:predicate sphn:hasVerificationStatusCode ]
```

Property `sphn:hasSubjectPseudoIdentifier` is the defined with a join condition:

```
[ rr:objectMap [ rr:parentTriplesMap :sphnSubjectPseudoIdentifier ] ;
            rr:predicate sphn:hasSubjectPseudoIdentifier ]
```

Which is joined with the specific mapping for that property:

```
:sphnSubjectPseudoIdentifier a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.sphn:SubjectPseudoIdentifier" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rml:reference "sphn:hasSharedIdentifier" ;
                    rr:datatype xsd:anyURI ] ;
            rr:predicate sphn:hasSharedIdentifier ],
        [ rr:objectMap [ rr:parentTriplesMap :sphnDataProvider ] ;
            rr:predicate sphn:hasDataProvider ],
        [ rr:objectMap [ rml:reference "sphn:hasIdentifier" ;
                    rr:datatype xsd:string ] ;
            rr:predicate sphn:hasIdentifier ] ;
    rr:subjectMap [ rr:class sphn:SubjectPseudoIdentifier ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-SubjectPseudoIdentifier-{id
}" ] .
```

Database schema

The project specific database schema is constructed on the JSON schema. Each concept defined in the schema is mapped to a database table, where the colons are replaced by underscores. The names of the columns are constructed according to the nesting of objects in the schema, where each additional layer of nesting is mapped to the column name by separating the upper object with the nested object with a double underscore. For example

```
"sphn:Allergy": {
    "type": "array",
    "description": "List of 'sphn:Allergy' concepts",
    "items": {
            "type": "object",
            "description": "SPHN Concept 'Allergy'",
            "properties": {
                "id": {
                    "type": "string",
                    "description": "ID of SPHN Concept 'Allergy'"
                },
                "sphn:hasSeverityCode": {
                    "type": "object",
                    "description": "SPHN Concept 'Terminology'",
                    "properties": {
                        "termid": {
                            "type": "string",
                            "description": "Unique ID for the given IRI. String format
follows convention: <coding_system>-<identifier>"
                        },
                        "iri": {
                            "type": "string",
```

```
                        "description": "IRI of SPHN Concept 'Terminology'"
                    }
                },
                "required": [
                    "termid",
                    "iri"
                ],
                "additionalProperties": false
            }
        }, ...
}}
```

The `sphn:Allergy` concept is mapped to the database table `sphn_Allergy`. Field `id` corresponds to column `id` in the table, field `termid` for property `sphn:hasSeverityCode` corresponds to column `sphn_hasSeverityCode__termid` in the table, and field `iri` for property `sphn:hasSeverityCode` corresponds to column `sphn_hasSeverityCode__iri` in the table.

In addition to the properties described by the JSON schema, each core concept defined in the content field of the schema, has the fields `sphn_hasSubjectPseudoIdentifier` and `sphn_hasDataProvider` defined.
According to the schema, we define as non-nullable the columns that are marked as required in the JSON schema (field must be required from upper level down to the level of the property to be non-nullable).

Shared resources across data providers

Project specific schemas could potentially define a datatype property for representing shared resources across data providers and registers (Creating Identifiers across SPHN data). The corresponding property defined as `project_prefix:hasSharedIdentifier`, will be of

type `xsd:anyURI`. In the SPHN Connector, we expect this property to be defined as of type string. The JSON schema will look like this:

```
{
    "project_prefix:hasSharedIdentifier": {
        "description": "Value for 'hasSharedIdentifier' property",
        "type": "string"
    }
}
```

The provided string values will then be defined in the RDF generated file as `xsd:anyURI`. Since the bare minimum rules for valid URI references are fairly generic, we do not apply any checks on the values provided via JSON/DB ingestion into the SPHN Connector. Therefore, it is essential that the user provides values for the property project_prefix:hasSharedIdentifier that adheres to the guidelines for the type https://www.w3.org/TR/xmlschema-2/#anyURI.

Edge Cases

There are some concepts and cases where logic is special. In this section we summarize the special/edge cases to give the user a better understanding on how to prepare the data for the SPHN Connector. The cases described are based on RDF schema version 2024.1 (for 2023.2 edge cases check the appendix).

Core concepts references

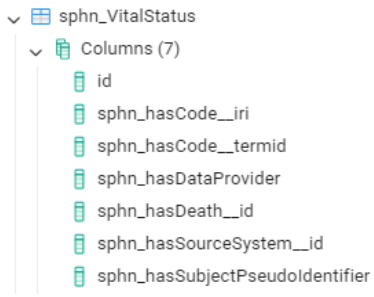
The core concepts are represented in the JSON schema under the key `content` and in the database schema as separate tables. The main difference we encounter in the new schema generated from 2024.1 is the reference to core concepts inside other core concepts. As you may remember, in SPHN RDF schema 2023.2 the property `sphn:hasAdministrativeCase` was considered a special case (see appendix).

From 2024, this pattern is applied to all the object properties that are pointing towards other core concepts. This means that no core concepts will be defined at a nested level but all will be defined at the top level under the `content` key. For example the core concept `sphn:VitalStatus` has an object property `sphn:hasDeath` which points to the core concept `sphn:Death`. In this case, instead of defining in the JSON schema the whole `sphn:Death` concept again, we only reference the `id` field of the `sphn:Death` concept which should be defined at the `content` level:

JSON schema:

```
"sphn:VitalStatus": {
    "description": "List of 'sphn:VitalStatus' concepts",
    "items": {
        "additionalProperties": false,
        "description": "SPHN Concept 'VitalStatus'",
        "properties": {
            ...,
            "sphn:hasDeath": {
                "additionalProperties": false,
                "description": "SPHN Concept 'Death'",
                "properties": {
                    "id": {
                        "description": "ID of SPHN Concept 'Death'",
                        "type": "string"
                    }
                },
                "required": [
                    "id"
                ],
                "type": "object"
            }
```

Database schema:



UCUM class

From SPHN RDF schema 2024.1, UCUM codes are treated as classes instead of named individuals (see appendix for 2023.2 structure). This affects the SPHN Connector with a small change on the UCUM concepts. UCUM codes are defined as classes, therefore the object will have an additional property `termid`. It will basically be described the same as a `sphn:Terminology` concept.

```
"sphn:hasCode": {
    "additionalProperties": false,
    "description": "Schema for property 'hasCode'",
    "properties": {
        "iri": {
            "description": "UCUM IRI for 'hasCode' property",
            "enum": [
                "https://biomedit.ch/rdf/sphn-resource/ucum/d"
            ],
            "type": "string"
        },
        "termid": {
            "description": "Unique ID for the given IRI. String format follows
convention: <coding_system>-<identifier>",
            "type": "string"
        }
    },
    "required": [
        "iri",
        "termid"
    ],
    "type": "object"
}
```

After RML mapping this results in the following structure (example):

```
resource:DATA-PROVIDER-ID-sphn-Unit-unitID a sphn:Unit;
  sphn:hasCode resource:DATA-PROVIDER-ID-sphn-Unit-unitID-sphn-Code-UCUM-CM .

resource:DATA-PROVIDER-ID-sphn-Unit-unitID-sphn-Code-UCUM-CM a ucum:cm .
```

Another addition, compared to older RDF schemas, is that for some UCUM IRIs (when it is defined by the restrictions) we add the allowed values via the `enum` field. In the above example, the value of the IRI field can only be representing UCUM code for the day. In the database schema this additional information generated an ad-hoc type for the column (same behavior as for other value sets).

Supporting concepts

In the 2024 SPHN RDF schema, there are some new concepts that differ from all the concepts we saw so far in previous versions. In previous versions we either had core concepts, special concepts like `sphn:DataProvider`, `sphn:SubjectPseudoIdentifier`, and `sphn:DataRelease`, and SPHN concepts that were referenced inside core concepts at an arbitrary nested level. In the new schema we find some concepts that do not fall in any of these cases. There are concepts that are not core concepts and are not referenced inside the latter at any point. We call them **supporting concepts**. According to the pre-release of 2024.1 schema these are the existing supporting concept which may change with future versions:

- `sphn:Interpretation`
- `sphn:ReferenceInterpretation`
- `sphn:SemanticMapping`
- `sphn:SourceData`.

The supporting concepts are defined in the JSON schema under a new section at the same level of the `content` key. They are defined under the key `supporting_concepts`.

```
"supporting_concepts": {
    "additionalProperties": false,
    "description": "List of SPHN supporting concepts",
    "properties": {
        "sphn:Interpretation": {...
        },
        "sphn:ReferenceInterpretation": {...
        },
        "sphn:SemanticMapping": {...
        },
        "sphn:SourceData": {...
        }
    },
    "type": "object"
}
```

The concepts defined inside the `supporting_concepts` object are similar to the concepts defined under the `content` object. Nevertheless, there are some relevant differences. All the object properties inside supporting concepts that point to a class that is not of type `Code` or `Terminology`, define only the `id` field for reference. The referenced concepts should be already defined somewhere else as a core concept or inside a core concept.
For example the supporting concept `sphn:SemanticMapping` has an object property `sphn:hasSourceSystem` which points to the concept `sphn:SourceSystem`. Every referenced object which is not `Code` or `Terminology` is not directly defined under the property but it is only referenced via the `id` field. The referred concept, e.g. `sphn:SourceSystem` in this case, is defined somewhere else inside the content object of the schema.

```
"sphn:hasSourceSystem": {
    "description": "List of 'hasSourceSystem' properties",
```

```
    "items": {
        "additionalProperties": false,
        "description": "SPHN Concept 'SourceSystem'",
        "properties": {
            "id": {
                "description": "ID of SPHN Concept 'SourceSystem'",
                "type": "string"
            }
        },
        "required": [
            "id"
        ],
        "type": "object"
    },
    "type": "array"
}
```

On the other hand, the object properties that point to `Code` or `Terminology` are similar to the properties defined inside core concepts. They have lighter restrictions on the mandatory fields. The `Terminology` codes require only the `termid` field and not the `iri` field, while the `Code` codes require only the `id` field. That's because we cannot infer from the RDF schema if the defined code should point to an existing code, and therefore only have the identifying field defined, or if it is a new code that needs to be fully defined. Is it up to the user to fill out the concepts as required. Moreover, there are two additional **mandatory** fields:

- `sourceConceptID`: The ID of the SPHN/Project concept type where this code has been used, e.g. 123.
- `sourceConceptType`: The SPHN/Project concept type where this code has been used, e.g. sphn-Consent. Naming convention: <prefix>-<conceptName>.

These additional concepts are used to correctly map the generated codes according to the 2024 naming convention (more details here).

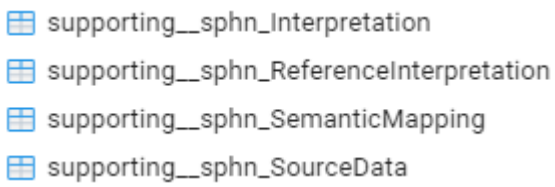
Example for property of the supporting concept `sphn:SemanticMapping`:

```
"sphn:hasOutputCode": {
            "description": "SPHN Concept 'Code'/SPHN Concept 'Terminology'",
            "oneOf": [
                {
                    "additionalProperties": false,
                    "description": "SPHN Concept 'Code'",
                    "properties": {
                        "id": {
                            "description": "ID of SPHN Concept 'Code'",
                            "type": "string"
                        },
                        "sourceConceptID": {
                            "description": "The ID of the SPHN/Project concept type
where this code has been used, e.g. 123",
                            "type": "string"
                        },
                        "sourceConceptType": {
```

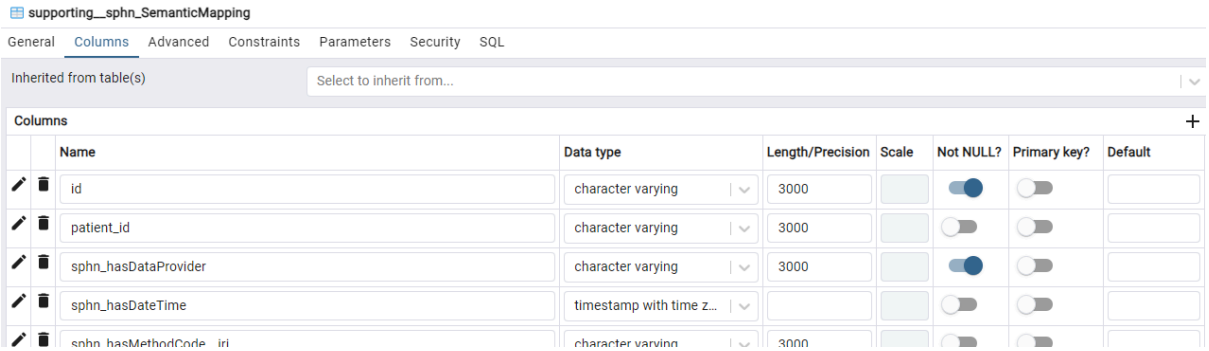
```
                            "description": "The SPHN/Project concept type where this
code has been used, e.g. sphn-Consent. Naming convention: <prefix>-<conceptName>",
                            "type": "string"
                        },
                        "sphn:hasCodingSystemAndVersion": {
                            "description": "Value for 'hasCodingSystemAndVersion'
property",
                            "type": "string"
                        },
                        "sphn:hasIdentifier": {
                            "description": "Value for 'hasIdentifier' property",
                            "type": "string"
                        },
                        "sphn:hasName": {
                            "description": "Value for 'hasName' property",
                            "type": "string"
                        }
                    },
                    "required": [
                        "id",
                        "sourceConceptID",
                        "sourceConceptType"
                    ],
                    "type": "object"
                },
                {
                    "additionalProperties": false,
                    "description": "SPHN Concept 'Terminology'",
                    "properties": {
                        "iri": {
                            "description": "IRI of SPHN Concept 'Terminology'",
                            "type": "string"
                        },
                        "sourceConceptID": {
                            "description": "The ID of the SPHN/Project concept type
where this code has been used, e.g. 123",
                            "type": "string"
                        },
                        "sourceConceptType": {
                            "description": "The SPHN/Project concept type where this
code has been used, e.g. sphn-Consent. Naming convention: <prefix>-<conceptName>",
                            "type": "string"
                        },
                        "termid": {
                            "description": "Unique ID for the given IRI. String format
follows convention: <coding_system>-<identifier>",
                            "type": "string"
                        }
                    },
                    "required": [
                        "sourceConceptID",
                        "sourceConceptType",
                        "termid"
                    ],
                    "type": "object"
                }
```

```
        ],
        "type": "object"
    }
```

Regarding the database schema, the supporting concepts are represented as separate tables with the suffix `supporting__` in their table names.



As explained, these concepts do not have a direct relationship with `sphn:SubjectPesudoIdentifier`. Nevertheless, they could be valid for all patients or only be relevant for some of them. Hence, the supporting tables in the database define an artificial column `patient_id`. Example:



If the supporting concept should be extracted only for one specific patient, then the column `patient_id` should be filled with the SPHN Connector patient identifier (same value defined in the `id` field of table `sphn_SubjectPseudoIdentifier`). In this case the data of the supporting concept will be extracted in the patient's JSON file. On the other hand, if the supporting concept is relevant for all the patients, then the column `patient_id` is left empty and the concept is extracted for all the available patients.

Code/Terminology naming convention

As described in Naming convention, Code and Terminology concepts contain information about the concept type where the code has been instantiated such as the concept unique identifier. The format is the following:
`resource:<data_provider_id>-<prefix-of-class-where-used>-<name-of-class-where-used>-<id-of-class-where-used>-sphn-Code-<id-of-the-code>`.
For example from the following JSON data for a aphn:Allergy:

```
{
```

```
    "id" : "allergy_id",
    "sphn:hasFirstRecordDateTime": "2022-04-24T11:55:43.304Z",
    "sphn:hasLastReactionDateTime": "2022-03-31T11:55:43.304Z",
    "sphn:hasAllergen": {
        "id": "allergen_id",
        "sphn:hasCode": [
            {
                "id": "code_id",
                "sphn:hasCodingSystemAndVersion": "test",
                "sphn:hasIdentifier": "test",
                "sphn:hasName": "test"
            }
        ]
    }
```

We extract the following instantiated code for sphn:Allergen:

```
resource:DATA-PROVIDER-ID-sphn-Allergen-allergen_id-sphn-Code-code_id a sphn:Code;
    sphn:hasCodingSystemAndVersion "test";
    sphn:hasIdentifier "test";
    sphn:hasName "test" .
```

To automatically map the data we added a default pre-check (see here) responsible for pushing down the upper concept unique identifier to an artificial field `sourceConceptID`. The user should not worry about defining source concept details as long as they are defining non-supporting concepts. For supporting concepts the fields `sourceConceptID` and `sourceConceptType` are mandatory in the schema and need to be specified as the mapping logic has no information about the source concept where that the code is pointing to.

SPHNConcept in property range

In RDF schema 2024.1 we encounter two object properties

- `sphn:Interpretation/sphn:hasInput`
- `sphn:Interpretation/sphn:hasInput`

that defines their range as `sphn:SPHNConcept`. This means that they can define any kind of concept described in the RDF schema. Their description is something like: *SPHN Concept 'AccessDevice'/SPHN Concept 'AccessDevicePresence'/SPHN Concept 'AdministrativeCase'/SPHN Concept 'AdministrativeSex' / …* From the fact we can have different resulting concepts we use the `target_concept` field (see Multi-classes ranges) to define which concept we are referring to, for example:

```
{
    "additionalProperties": false,
    "description": "SPHN Concept 'AccessDevice'",
    "properties": {
        "id": {
            "description": "ID of SPHN Concept 'AccessDevice'",
            "type": "string"
        },
        "target_concept": {
            "description": "IRI for Concept 'AccessDevice'",
            "enum": [
```

```
                "https://biomedit.ch/rdf/sphn-schema/sphn#AccessDevice"
            ],
            "type": "string"
        }
    },
    "required": [
        "id",
        "target_concept"
    ],
    "type": "object"
}
```

This is handled the same as for other multi-range properties of core concepts. Obviously, as this is a supporting concept, we only report the `id` field. There are some exceptions though. The first one is when defining `Code` and `Terminology` concepts. For those cases, differently as done before for core concepts, we also need to define the `target_concept` field. We have:

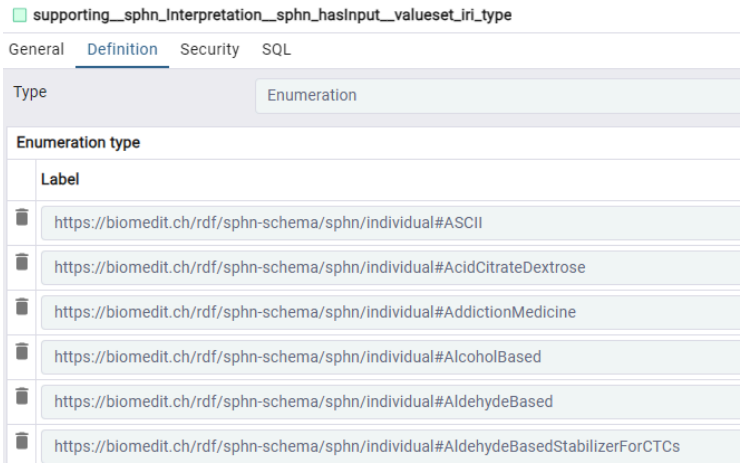
```
{
    "additionalProperties": false,
    "description": "SPHN Concept 'Code'",
    "properties": {
        "id": {
            "description": "ID of SPHN Concept 'Code'",
            "type": "string"
        },
        "sourceConceptID": {
            "description": "The ID of the SPHN/Project concept type where this code has
been used, e.g. 123",
            "type": "string"
        },
        "sourceConceptType": {
            "description": "The SPHN/Project concept type where this code has been used,
e.g. sphn-Consent. Naming convention: <prefix>-<conceptName>",
            "type": "string"
        },
        "sphn:hasCodingSystemAndVersion": {
            "description": "Value for 'hasCodingSystemAndVersion' property",
            "type": "string"
        },
        "sphn:hasIdentifier": {
            "description": "Value for 'hasIdentifier' property",
            "type": "string"
        },
        "sphn:hasName": {
            "description": "Value for 'hasName' property",
            "type": "string"
        },
        "target_concept": {
            "description": "IRI for Concept 'Code'",
            "enum": [
                "https://biomedit.ch/rdf/sphn-schema/sphn#Code"
            ],
            "type": "string"
        }
```

```
    },
    "required": [
        "id",
        "sourceConceptID",
        "sourceConceptType",
        "target_concept"
    ],
    "type": "object"
},
{
    "additionalProperties": false,
    "description": "SPHN Concept 'Terminology'",
    "properties": {
        "iri": {
            "description": "IRI of SPHN Concept 'Terminology'",
            "type": "string"
        },
        "sourceConceptID": {
            "description": "The ID of the SPHN/Project concept type where this code has
been used, e.g. 123",
            "type": "string"
        },
        "sourceConceptType": {
            "description": "The SPHN/Project concept type where this code has been used,
e.g. sphn-Consent. Naming convention: <prefix>-<conceptName>",
            "type": "string"
        },
        "target_concept": {
            "description": "IRI for Concept 'Terminology'",
            "enum": [
                "https://biomedit.ch/rdf/sphn-schema/sphn#Terminology"
            ],
            "type": "string"
        },
        "termid": {
            "description": "Unique ID for the given IRI. String format follows
convention: <coding_system>-<identifier>",
            "type": "string"
        }
    },
    "required": [
        "sourceConceptID",
        "sourceConceptType",
        "target_concept",
        "termid"
    ],
    "type": "object"
}
```

In these cases we are also allowed to specify the additional fields. The second exception is related to the subclasses of `sphn:ValueSet`. We group all the possible values of a value set inside a single block and define a new field called **`valueset_iri`**. The value should be a full IRI listed in the allowed values. Example:

```
{
    "additionalProperties": false,
    "description": "SPHN Concept 'ValueSet'",
    "properties": {
        "valueset_iri": {
            "description": "IRI for 'hasInput' property",
            "enum": [
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#ASCII",
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#AcidCitrateDextrose",
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#AddictionMedicine",
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#AlcoholBased",
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#AldehydeBased",
                ...
```

Similarly as for other IRI fields, the column in the database will have its own defined type:



Data provider concept

The structure of the data provider institute concept changes from 2024.1. The concept is renamed from `sphn:DataProviderInstitute` to `sphn:DataProvider` (see appendix for 2023.2 structure). In the SPHN Connector each patient file must be related to the same DataProvider. Based on this assumption, we moved the `sphn:DataProvider` schema to the top level of the JSON schema and adapted the RML mapping accordingly.

JSON schema

This object is defined on the top level of the JSON schema. In that way the user can define the data provider once for the entire file. There are no other references in the JSON schema to that field, meaning that all the object properties `sphn:hasDataProvider` have been removed from the corresponding objects.

```
"sphn:DataProvider": {
    "additionalProperties": false,
    "description": "SPHN Concept 'DataProvider'",
    "properties": {
        "id": {
            "description": "ID of SPHN Concept 'DataProvider'",
            "type": "string"
        },
```

```
        "sphn:hasCategory": {
            "additionalProperties": false,
            "description": "Value Set for property 'hasCategory'",
            "properties": {
                "iri": {
                    "description": "IRI for 'hasCategory' property",
                    "enum": [
                        "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Company",

"https://biomedit.ch/rdf/sphn-schema/sphn/individual#ExternalLaboratory",

"https://biomedit.ch/rdf/sphn-schema/sphn/individual#FederalOffice",

"https://biomedit.ch/rdf/sphn-schema/sphn/individual#HealthInsurance",
                        "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Hospital",
                        "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Pharmacy",

"https://biomedit.ch/rdf/sphn-schema/sphn/individual#PrivatePractice",

"https://biomedit.ch/rdf/sphn-schema/sphn/individual#ResearchOrganization",

"https://biomedit.ch/rdf/sphn-schema/sphn/individual#ServiceProvider",
                        "https://biomedit.ch/rdf/sphn-schema/sphn/individual#University"
                    ],
                    "type": "string"
                }
            },
            "required": [
                "iri"
            ],
            "type": "object"
        },
        "sphn:hasDepartment": {
            "additionalProperties": false,
            "description": "SPHN Concept 'Department'",
            "properties": {
                "id": {
                    "description": "ID of SPHN Concept 'Department'",
                    "type": "string"
                },
                "sphn:hasName": {
                    "description": "Value for 'hasName' property",
                    "type": "string"
                }
            },
            "required": [
                "id",
                "sphn:hasName"
            ],
            "type": "object"
        },
        "sphn:hasInstitutionCode": {
            "additionalProperties": false,
            "description": "SPHN Concept 'Code'",
            "properties": {
                "id": {
```

```
                    "description": "ID of SPHN Concept 'Code'",
                    "type": "string"
                },
                "sphn:hasCodingSystemAndVersion": {
                    "description": "Value for 'hasCodingSystemAndVersion' property",
                    "type": "string"
                },
                "sphn:hasIdentifier": {
                    "description": "Value for 'hasIdentifier' property",
                    "type": "string"
                },
                "sphn:hasName": {
                    "description": "Value for 'hasName' property",
                    "type": "string"
                }
            },
            "required": [
                "id",
                "sphn:hasCodingSystemAndVersion",
                "sphn:hasIdentifier"
            ],
            "type": "object"
        }
    },
    "required": [
        "id",
        "sphn:hasInstitutionCode"
    ],
    "type": "object"
}
```

RML mapping

The correlation between the object properties depending on the data provider institute and the institute defined at the top level of the JSON is handled directly in the mapping.

There is a single TriplesMap for that object that iterates over the JSON path "$.sphn:DataProvider":

```
:sphnDataProvider a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.sphn:DataProvider" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rml:reference "sphn:hasCategory.iri" ;
                    rr:termType rr:IRI ] ;
            rr:predicate sphn:hasCategory ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-DataProvider-{id}-sphn-Code
-{sphn:hasInstitutionCode.id}" ] ;
            rr:predicate sphn:hasInstitutionCode ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Department-{sphn:hasDepartm
ent.id}" ] ;
            rr:predicate sphn:hasDepartment ] ;
    rr:subjectMap [ rr:class sphn:DataProvider ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-DataProvider-{id}" ] .
```

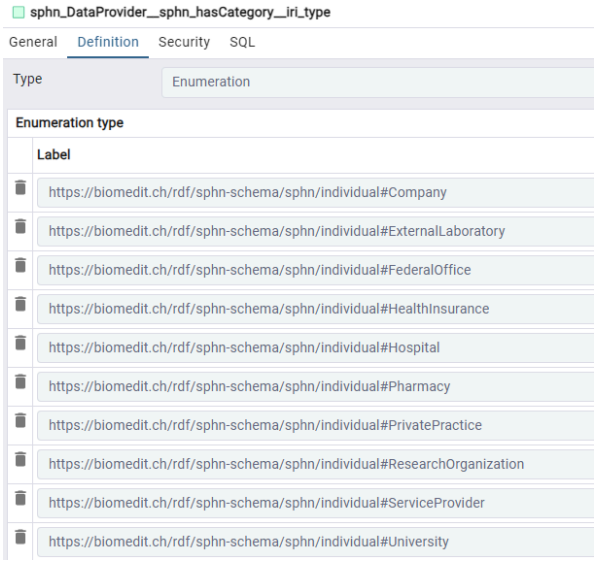
Other concepts can define this concepts by defining a join condition defined in the properties `sphn:hasDataProvider` with the map:

```
[ rr:objectMap [ rr:parentTriplesMap :sphnDataProvider ] ;
            rr:predicate sphn:hasDataProvider ]
```

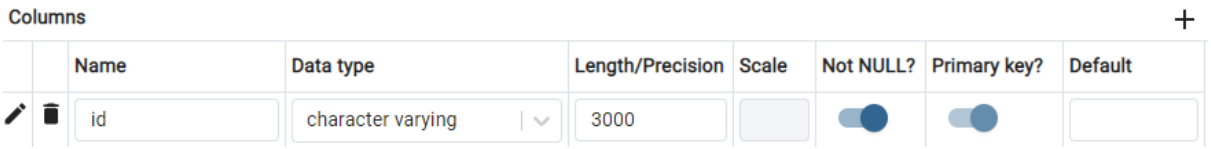
Database schema

Concept `sphn:DataProvider` has its own table in the database schema. In principle, from the fact that a project should be related to a single data provider, that table has only a single record defining the provider.

The property `sphn_hasCategory__iri` can only take the following values:

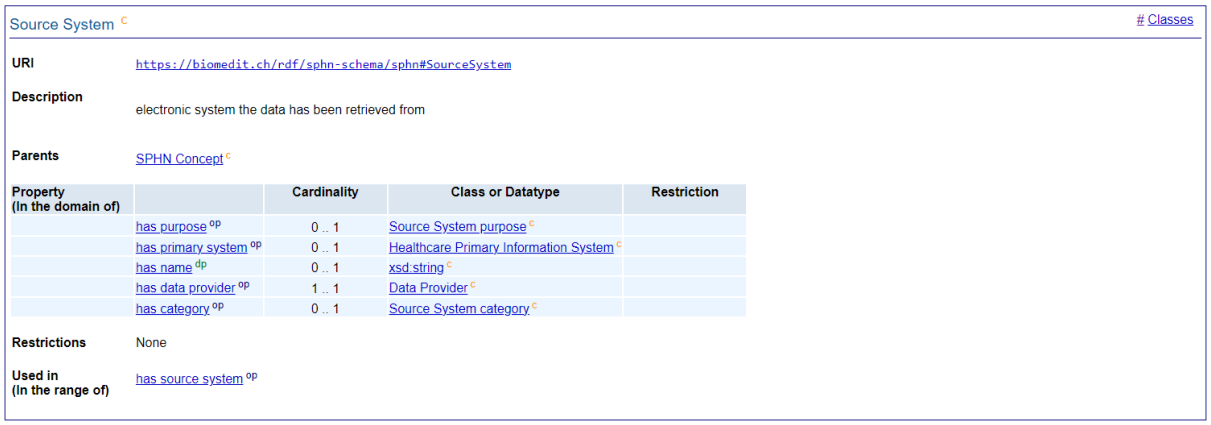


This table has primary key defined by the `id` field



Source system concept

The concept `sphn:SourceSystem` is not a core concept.



Source System [C] # Classes

URI	https://biomedit.ch/rdf/sphn-schema/sphn#SourceSystem
Description	electronic system the data has been retrieved from
Parents	SPHN Concept [C]

Property (In the domain of)		Cardinality	Class or Datatype	Restriction
	has purpose [OP]	0 .. 1	Source System purpose [C]	
	has primary system [OP]	0 .. 1	Healthcare Primary Information System [C]	
	has name [DP]	0 .. 1	xsd:string [C]	
	has data provider [OP]	1 .. 1	Data Provider [C]	
	has category [OP]	0 .. 1	Source System category [C]	

Restrictions	None
Used in (In the range of)	has source system [OP]

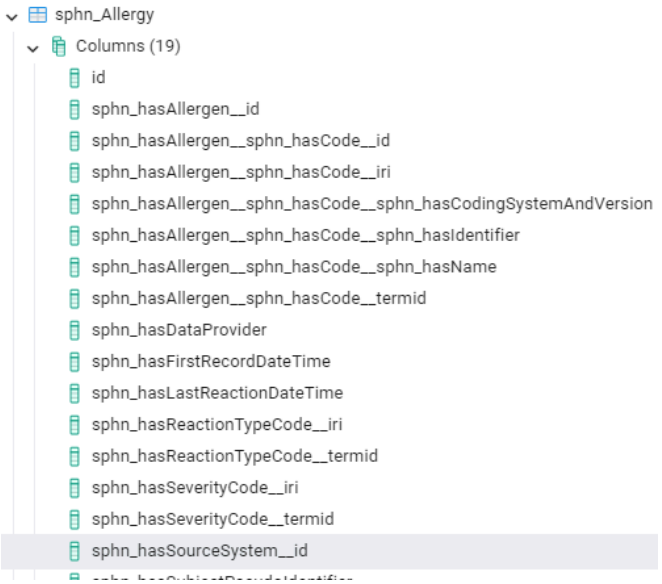
Nevertheless, it is referenced many times in other concepts via the object property `sphn:hasSourceSystem`.

To avoid the creation of extra properties/columns in the JSON/database schema we decided to treat this concept as a special concept. It will be treated as a standard core concept and inside other concepts it will be referenced solely by the `id` field.

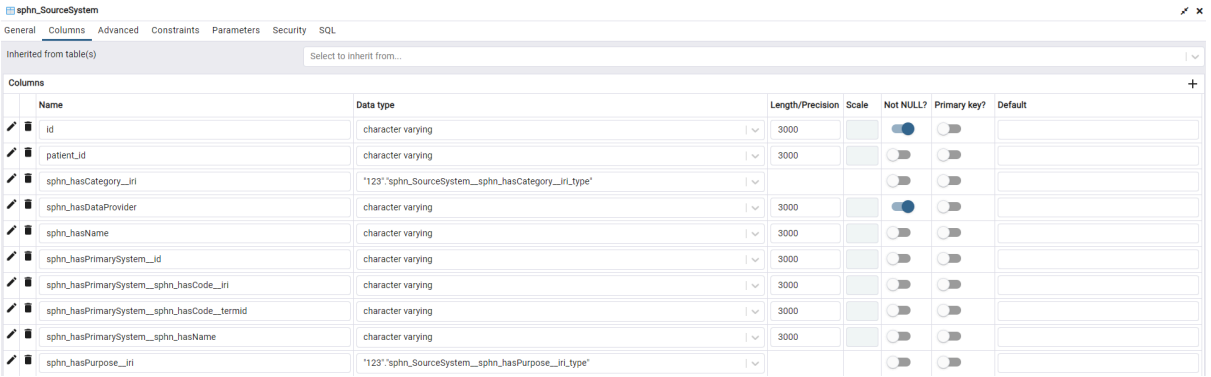
JSON schema:

```
"sphn:hasSourceSystem": {
    "description": "List of 'hasSourceSystem' properties",
    "items": {
        "additionalProperties": false,
        "description": "SPHN Concept 'SourceSystem'",
        "properties": {
            "id": {
                "description": "ID of SPHN Concept 'SourceSystem'",
                "type": "string"
            }
        },
        "required": [
            "id"
        ],
        "type": "object"
    },
    "type": "array"
}
```

Database schema:



The table defining the source system concept is represented as a standard core concept. The difference is that there is no direct relation to the `sphn:SubjectPseudoIdentifier` and therefore we define an artificial column `patient_id` to extract the source system data only for the relevant patients. If the column is not filled the data is extracted for all the patients, otherwise it is extracted only for the specified patient. The behavior is similar as for the supporting concepts.



Subject pseudo identifier concept

In the SPHN Connector each patient file must be related to the same SubjectPseudoIdentifier. Based on this assumption, we moved the `sphn:SubjectPseudoIdentifier` schema to the top level of the JSON schema and adapted the RML mapping accordingly (see appendix for 2023.2 structure).

JSON schema

This object is defined on the top level of the JSON schema. In that way the user can define the patient information once for the entire file. There are no other references in the JSON schema to that field, meaning that all the object properties `sphn:hasSubjectPseudoIdentifier` have been removed from the corresponding objects.

```
"sphn:SubjectPseudoIdentifier": {
    "additionalProperties": false,
    "description": "SPHN Concept 'SubjectPseudoIdentifier'",
    "properties": {
        "id": {
            "description": "ID of SPHN Concept 'SubjectPseudoIdentifier'",
            "type": "string"
        },
        "sphn:hasIdentifier": {
            "description": "Value for 'hasIdentifier' property",
            "type": "string"
        },
        "sphn:hasSharedIdentifier": {
            "description": "Value for 'hasSharedIdentifier' property",
            "type": "string"
        }
    },
    "required": [
        "id",
        "sphn:hasIdentifier"
    ],
    "type": "object"
}
```

RML mapping

The correlation between the object properties depending on the subject pseudo identifier and the identifier defined at the top level of the JSON is handled directly in the mapping.

There is a single TriplesMap for that object that iterates over the JSON path `"$.sphn:SubjectPseudoIdentifier"`:

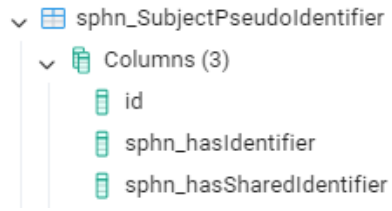
```
:sphnSubjectPseudoIdentifier a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.sphn:SubjectPseudoIdentifier" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rml:reference "sphn:hasSharedIdentifier" ;
                    rr:datatype xsd:anyURI ] ;
            rr:predicate sphn:hasSharedIdentifier ],
        [ rr:objectMap [ rr:parentTriplesMap :sphnDataProvider ] ;
            rr:predicate sphn:hasDataProvider ],
        [ rr:objectMap [ rml:reference "sphn:hasIdentifier" ;
                    rr:datatype xsd:string ] ;
            rr:predicate sphn:hasIdentifier ] ;
    rr:subjectMap [ rr:class sphn:SubjectPseudoIdentifier ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-SubjectPseudoIdentifier-{id
}" ] .
```

Other concepts can define this concepts by defining a join condition defined in the properties `sphn:hasSubjectPseudoIdentifier` with the map:

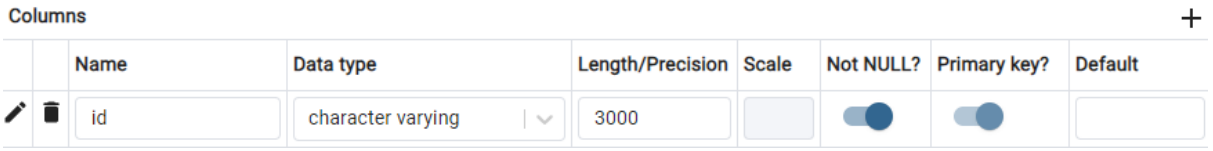
```
[ rr:objectMap [ rr:parentTriplesMap :sphnSubjectPseudoIdentifier ] ;
            rr:predicate sphn:hasSubjectPseudoIdentifier ]
```

Database schema

Reflecting the JSON schema, the resulting table for this concept is very simple



The table has primary key defined by the `id` field



Data release concept

This is an additional concept which should be added to all RDF files after JSON to RDF conversion. It gives information about the extraction date of the data and the version of the RDF schema (see appendix for 2023.2 structure).

JSON schema

In the JSON schema, this quantity is described by the `sphn:DataRelease` property at the top level:

```
"sphn:DataRelease": {
    "description": "sphn:DataRelease properties",
    "properties": {
        "creationTime": {
            "description": "Value for the sphn:hasExtractionDateTime of the data
release",
            "format": "date-time",
            "type": "string"
        },
        "id": {
            "description": "ID of 'sphn:DataRelease'. To ensure the uniqueness of a
DataRelease instance ID (i.e. the dataset identifier), a UNIX Epoch timestamp should
ideally be concatenated to it as a suffix",
            "type": "string"
        }
    },
    "required": [
        "creationTime",
        "id"
    ],
    "type": "object"
```

```
}
```

The uniqueness of the DataRelease object is given by its `id`. Usually, its value should represent the UNIX timestamp of the `creationTime` date-time.

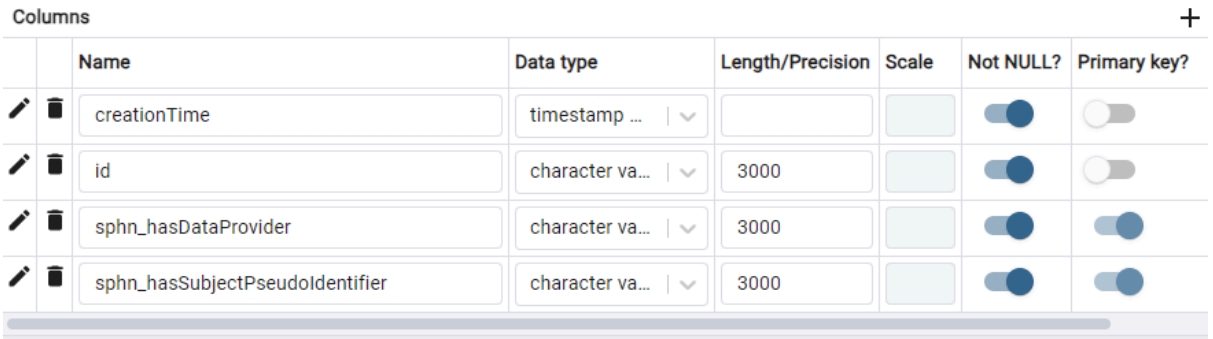
RML mapping

The RDF schema version is extracted internally from the schema and reported to the mapping. The extraction date and extraction timestamp are read from the JSON data.

```
:sphnDataRelease a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.sphn:DataRelease" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-schema/sphn/2024/2" ] ;
            rr:predicate dct:conformsTo ],
        [ rr:objectMap [ rml:reference "creationTime" ;
                rr:datatype xsd:dateTime ] ;
            rr:predicate sphn:hasExtractionDateTime ],
        [ rr:objectMap [ rr:parentTriplesMap :sphnDataProvider ] ;
            rr:predicate sphn:hasDataProvider ] ;
    rr:subjectMap [ rr:class sphn:DataRelease ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-DataRelease-{id}" ] .
```

Database schema

The table `sphn_DataRelease` is also constructed on the JSON schema. It has only non-nullable columns and the primary key defined by `sphn_hasSubjectPseudoIdentifier` and `sphn_hasDataProvider`, such that only one patient for a specific data provider can be defined in the table.



This table is used by the user once all the data for a specific patient has been uploaded. Therefore, the user marks the patient as released and the data will be pulled at the next triggered ingestion into the SPHN Connector.

Code and Terminology

Some object properties have a range that includes both the `Code` and the `Terminology` concept. This is reflected in a duality also for the JSON schema and the RML mapping.

JSON schema

The `Code` schema reflects its defined properties, while the `Terminology` schema has predefined structure:

```
{
    "type": "object",
    "description": "SPHN Concept 'Terminology'",
    "properties": {
        "termid": {
            "type": "string",
            "description": "Unique ID for the given IRI. Its format must follow the
convention: <coding_system>-<identifier>"
        },
        "iri": {
            "type": "string",
            "description": "IRI of SPHN Concept 'Terminology'"
        }
    },
    "required": [
        "termid",
        "iri"
    ],
    "additionalProperties": false
}
```

The mandatory field `termid` works similarly as the `id` field for the non-terminology properties. It is a unique identifier which is unique for every defined IRI value. The `termid` format must match the following convention: `<coding_system>-<identifier>`. For example

```
"sphn:hasSeverityCode": {
    "termid": "SNOMED_CT-723505004",
    "iri" : "http://snomed.info/id/723505004"
}
```

Moreover, it's possible that an object property could be defined either as `Code` concept or as `Terminology` concept. This is defined in the schema by means of the `oneOf` attribute:

```
"sphn:hasCode": {
    "type": "object",
    "description": "SPHN Concept 'Code'/SPHN Concept 'Terminology'",
    "oneOf": [
        {
            "type": "object",
            "description": "SPHN Concept 'Code'",
            "properties": {
                "id": {
```

```
                    "type": "string",
                    "description": "ID of SPHN Concept 'Code'"
                },
                "sphn:hasIdentifier": {
                    "description": "Value for 'hasIdentifier' property",
                    "type": "string"
                },
                "sphn:hasCodingSystemAndVersion": {
                    "description": "Value for 'hasCodingSystemAndVersion' property",
                    "type": "string"
                },
                "sphn:hasName": {
                    "description": "Value for 'hasName' property",
                    "type": "string"
                }
            },
            "required": [
                "id",
                "sphn:hasIdentifier",
                "sphn:hasCodingSystemAndVersion"
            ],
            "additionalProperties": false
        },
        {
            "type": "object",
            "description": "SPHN Concept 'Terminology'",
            "properties": {
                "termid": {
                    "type": "string",
                    "description": "Unique ID for the given IRI. String format follows
convention: <coding_system>-<identifier>"
                },
                "iri": {
                    "type": "string",
                    "description": "IRI of SPHN Concept 'Terminology'"
                }
            },
            "required": [
                "termid",
                "iri"
            ],
            "additionalProperties": false
        }
    ]
}
```

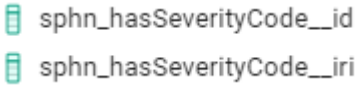
RML mapping

The RML maps are constructed based on the JSON schema. The duality is represented simply by reporting both possibilities, then depending on which schema is used in the JSON data the correct one is automatically picked up and used to generate the corresponding RDF data. In the following example both possibilities are represented in the mapping (both `id` and `termid` referenced) for the property `sphn:hasSeverityCode`:

```
:sphnAdverseEvent a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.content.sphn:AdverseEvent[*]" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:parentTriplesMap :sphnDataProvider ] ;
            rr:predicate sphn:hasDataProvider ],
        [ rr:objectMap [ rml:reference "sphn:hasConsequences.iri" ;
                rr:termType rr:IRI ] ;
            rr:predicate sphn:hasConsequences ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-AdverseEvent-{id}-sphn-Code
-{sphn:hasSeverityCode.termid}" ] ;
            rr:predicate sphn:hasSeverityCode ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-AdverseEvent-{id}-sphn-Code
-{sphn:hasSeverityCode.id}" ] ;
            rr:predicate sphn:hasSeverityCode ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-AdverseEvent-{id}-sphn-Code
-{sphn:hasCode.id}" ] ;
            rr:predicate sphn:hasCode ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-SourceSystem-{sphn:hasSourc
eSystem[*].id}" ] ;
            rr:predicate sphn:hasSourceSystem ],
        [ rr:objectMap [ rml:reference "sphn:hasOnsetDateTime" ;
                rr:datatype xsd:dateTime ] ;
            rr:predicate sphn:hasOnsetDateTime ],
        [ rr:objectMap [ rr:parentTriplesMap :sphnSubjectPseudoIdentifier ] ;
            rr:predicate sphn:hasSubjectPseudoIdentifier ],
        [ rr:objectMap [ rml:reference "sphn:hasOutcome.iri" ;
                rr:termType rr:IRI ] ;
            rr:predicate sphn:hasOutcome ],
        [ rr:objectMap [ rml:reference "sphn:hasIntervention" ;
                rr:datatype xsd:string ] ;
            rr:predicate sphn:hasIntervention ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-AdministrativeCase-{sphn:ha
sAdministrativeCase.id}" ] ;
            rr:predicate sphn:hasAdministrativeCase ] ;
    rr:subjectMap [ rr:class sphn:AdverseEvent ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-AdverseEvent-{id}" ]
```

Database schema

In the case of `Code/Terminology` duality defined in the schema by the `oneOf` keyword, the corresponding columns for both the cases are reported into the table. Therefore, the user will update only the columns corresponding to either a `Code` concept or a `Terminology` concept. For example in the `sphn:AdverseEvent` concept we have the following:



Value Sets

With ValueSets we define those concepts which have a number of well-defined values. Some restrictions are applied in the schema on the properties by setting the field `owl:someValuesFrom`. We consider this a special case, because the idea was to report those values on the JSON schema.

JSON schema

Properties that have a predefined set of values have a specific schema. They are defined only by the `iri` attribute which describes the full IRI representing one of the allowed values. The allowed values are then listed by means of the `enum` keyword, therefore enforcing the `iri` field to only take one of those values in order for the schema to be valid. For example the property `sphn:hasRank` for the core concept `sphn:BilledDiagnosis` has only few values allowed:

```
"sphn:hasRank": {
    "additionalProperties": false,
    "description": "Value Set for property 'hasRank'",
    "properties": {
        "iri": {
            "description": "IRI for 'hasRank' property",
            "enum": [
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Complementary",
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Principal",
                "https://biomedit.ch/rdf/sphn-schema/sphn/individual#Secondary"
            ],
            "type": "string"
        }
    },
    "required": [
        "iri"
    ],
    "type": "object"
}
```

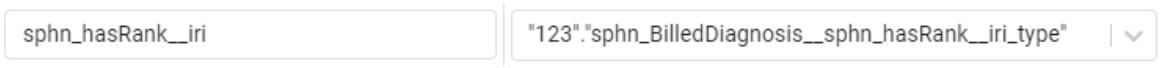
RML mapping

The structure of the JSON schema is reflected in the RML mapping where only the `iri` field is referenced when generating the RDF data. Its value is defined as a `rr.IRI`:

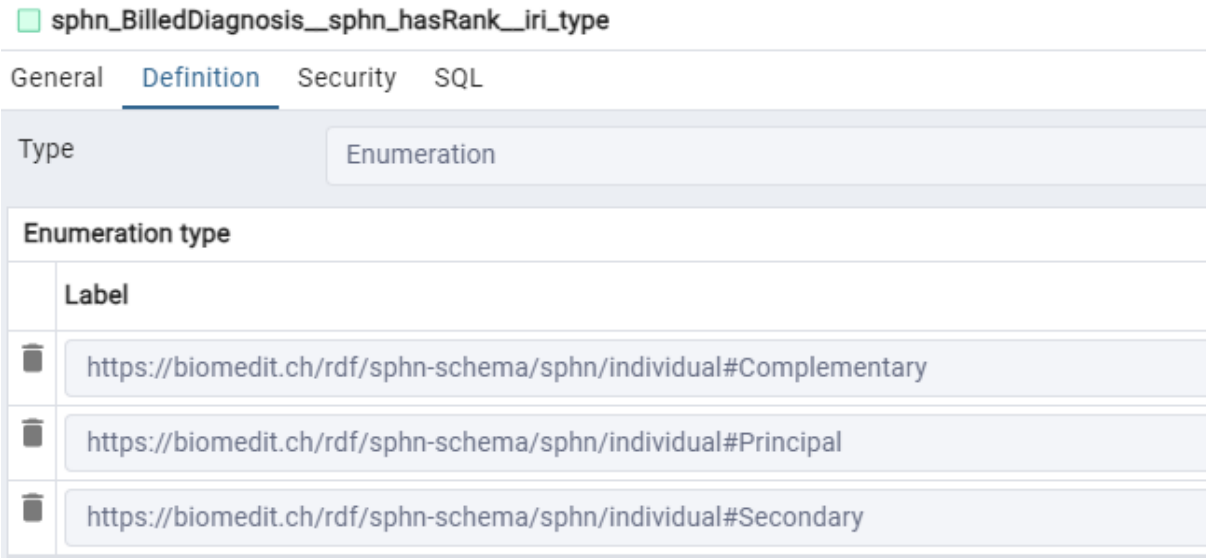
```
[ rr:objectMap [ rml:reference "sphn:hasRank.iri" ;
                 rr:termType rr:IRI ] ;
          rr:predicate sphn:hasRank ]
```

Database schema

The properties representing a ValueSet for which Enum values are listed in the JSON schema, are mapped to table's columns with an ad-hoc predefined enum type. For example, the property `sphn:hasRank` in the `sphn:BilledDiagnosis` concept has the following column name and column type



Where "123" is the test project schema and the type `sphn_BilledDiagnosis_sphn_hasRank_iri_type` is defined with the enum values taken from the JSON schema



This means that only those four values can be inserted into that column.

Properties with higher cardinality

In general, each property defines a single object. In some cases though, some properties are allowed to have multiple definitions. For example, the SPHN concept `sphn:DrugPrescription` allows the object property `sphn:hasActiveIngredient` under `sphn:hasDrug` property to have multiple `Substance` objects defined. In this case, we define the property as an array of the same concept.

JSON schema

To describe the schema for this edge case we use an example. As described above, the object property `sphn:hasActiveIngredient` can have multiple `Substance` concepts defined. In the schema, we therefore define this property as an **array** of `Substance` concepts:

```
"sphn:hasActiveIngredient": {
    "type": "array",
    "description": "List of 'hasActiveIngredient' properties",
    "items": {
            "type": "object",
            "description": "SPHN Concept 'Substance'",
            "properties": {
                "id": {
                    "type": "string",
```

```
                    "description": "ID of SPHN Concept 'Substance'"
                },
                ...
            },
            "required": [
                "id"
            ],
            "additionalProperties": false
        }
}
```

RML mapping

We generate a RML mapping to map all the concepts listed recursively. This is done by iterating over the JSON path `"$.content..sphn:hasDrug.sphn:hasActiveIngredient[*]"`:

```
:sphnArrayhasDrugsphnhasActiveIngredientMapping a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator
"$.content..sphn:hasDrug.sphn:hasActiveIngredient[*]" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Substance-{id}-sphn-Code-{s
phn:hasCode.id}" ] ;
            rr:predicate sphn:hasCode ],
        [ rr:objectMap [ rml:reference "sphn:hasGenericName" ;
                    rr:datatype xsd:string ] ;
            rr:predicate sphn:hasGenericName ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Quantity-{sphn:hasQuantity.
id}" ] ;
            rr:predicate sphn:hasQuantity ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Substance-{id}-sphn-Code-{s
phn:hasCode.termid}" ] ;
            rr:predicate sphn:hasCode ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-SourceSystem-{sphn:hasSourc
eSystem[*].id}" ] ;
            rr:predicate sphn:hasSourceSystem ] ;
    rr:subjectMap [ rr:class sphn:Substance ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Substance-{id}" ]
```

To map the corresponding object listed in the same place for the `Drug` concept we are considering, we use a logic with `rr:template`. We do not introduce a join condition because we will have multiple matches on the object list:

```
[ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Substance-{sphn:hasActiveIn
gredient[*].id}" ] ;
    rr:predicate sphn:hasActiveIngredient ]
```

Database schema

The schema of the tables with columns with higher cardinality does not change. To construct a quantity which is supposed to have a multi-value field, we just need to define multiple records in the table. The records will have the same values for all the fields that are not part of the array property, and just differ in those values. For example (based on 2023.2, same logic from 2024.1):

	id character varying (3000)	sphn_hassubjectpseudoidentifier character varying (3000)	sphn_hasdataproviderinsitute character varying (3000)	sphn_hasdrug__id character varying (3000)	sphn_hasdrug__sphn_hasactiveingredient__id character varying (3000)	sphn_hasdrug__sphn_hasactiveingredient__sphn_hasgenericname character varying (3000)
1	drugPres1	subject1	DATA-PROVIDER-ID	mydrug1	myingredient1	name1
2	drugPres1	subject1	DATA-PROVIDER-ID	mydrug1	myingredient2	name2

Multi-classes ranges

Similarly as in previous RDF schema versions (see appendix for 2023.2 structure), the target_concept field is necessary in the schema to distinguish the range of object properties that have multiple classes in their range. For example the `sphn:DrugPrescription` core concept has a property called `sphn:hasIndicationToStart` which links directly to the concept Diagnosis . Diagnosis on the other hand has multiple children: BilledDiagnosis, NursingDiagnosis, OncologyDiagnosis. So the `sphn:hasIndicationToStart` can link to Diagnosis, BilledDiagnosis, NursingDiagnosis, OncologyDiagnosis.

This is not a special case for the concept `sphn:DrugPrescription`, this pattern is frequent in the RDF schema. For the documentation we will limit our explanation to the relationship between the property `sphn:hasIndicationToStart` and the concept Diagnosis.

JSON schema

This one to many relationship is resolved in JSON with a “oneOf” so the “hasIndicationToStart” can be one of Diagnosis, BilledDiagnosis, NursingDiagnosis, OncologyDiagnosis. Without a proper identifier we can not be certain which concept the data is referring to. To distinguish between those concepts we propose a new property that is added to these special cases. The “**target_concept**” property. The target concept property is defined as an enum and it only holds one value: The IRI of the concept it is referring to. So for example for the Diagnosis concept inside of `sphn:hasIndicationToStart` is defined in the JSON like this:

```
{
   "target_concept": {
      "type": "string",
     "description": "IRI for Concept 'Diagnosis'",
     "enum": [
        "https://biomedit.ch/rdf/sphn-schema/sphn#Diagnosis"
     ]
   }
}
```

If we combine these information the JSON schema for `sphn:hasIndicationToStart` will look like this:

```
 "sphn:hasIndicationToStart": {
    "description": "List of 'hasIndicationToStart' properties",
    "items": {
        "description": "SPHN Concept 'BilledDiagnosis'/SPHN Concept 'Diagnosis'/SPHN
Concept 'NursingDiagnosis'/SPHN Concept 'OncologyDiagnosis'",
        "oneOf": [
                {
                    "additionalProperties": false,
                    "description": "SPHN Concept 'BilledDiagnosis'",
                    "properties": {
                        "id": {
                            "description": "ID of SPHN Concept 'BilledDiagnosis'",
                            "type": "string"
                        },
                        "target_concept": {
                            "description": "IRI for Concept 'BilledDiagnosis'",
                            "enum": [

"https://biomedit.ch/rdf/sphn-schema/sphn#BilledDiagnosis"
                            ],
                            "type": "string"
                        }
                    },
                    "required": [
                        "id",
                        "target_concept"
                    ],
                    "type": "object"
                },
                {
                    "additionalProperties": false,
                    "description": "SPHN Concept 'Diagnosis'",
                    "properties": {
                        "id": {
                            "description": "ID of SPHN Concept 'Diagnosis'",
                            "type": "string"
                        },
                        "target_concept": {
                            "description": "IRI for Concept 'Diagnosis'",
                            "enum": [
                                "https://biomedit.ch/rdf/sphn-schema/sphn#Diagnosis"
                            ],
                            "type": "string"
                        }
                    },
                    "required": [
                        "id",
                        "target_concept"
                    ],
                    "type": "object"
                },
                {
                    "additionalProperties": false,
                    "description": "SPHN Concept 'NursingDiagnosis'",
                    "properties": {
                        "id": {
```

```
                            "description": "ID of SPHN Concept 'NursingDiagnosis'",
                            "type": "string"
                        },
                        "target_concept": {
                            "description": "IRI for Concept 'NursingDiagnosis'",
                            "enum": [
"https://biomedit.ch/rdf/sphn-schema/sphn#NursingDiagnosis"
                            ],
                            "type": "string"
                        }
                    },
                    "required": [
                        "id",
                        "target_concept"
                    ],
                    "type": "object"
                },
                {
                    "additionalProperties": false,
                    "description": "SPHN Concept 'OncologyDiagnosis'",
                    "properties": {
                        "id": {
                            "description": "ID of SPHN Concept 'OncologyDiagnosis'",
                            "type": "string"
                        },
                        "target_concept": {
                            "description": "IRI for Concept 'OncologyDiagnosis'",
                            "enum": [
"https://biomedit.ch/rdf/sphn-schema/sphn#OncologyDiagnosis"
                            ],
                            "type": "string"
                        }
                    },
                    "required": [
                        "id",
                        "target_concept"
                    ],
                    "type": "object"
                }
    ],
    "type": "object"
}
```


RML Mapping

This new pattern has also an effect on the created RML mappings. We use a conditional mapping to map the properties according to the selected class for the target_concept. These are the mappings for sphn:hasIndicationToStart inside the drug prescription mapping:

```
rr:predicateObjectMap [ rr:objectMap [ rr:template
```

```
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-BilledDiagnosis-{sphn:hasIn
dicationToStart[?(@.target_concept=='https://biomedit.ch/rdf/sphn-schema/sphn#BilledDiag
nosis')].id}" ] ;
        rr:predicate sphn:hasIndicationToStart ],
    [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Diagnosis-{sphn:hasIndicati
onToStart[?(@.target_concept=='https://biomedit.ch/rdf/sphn-schema/sphn#Diagnosis')].id}
" ] ;
        rr:predicate sphn:hasIndicationToStart ],
    [ rr:objectMap [ rr:parentTriplesMap :sphnSubjectPseudoIdentifier ] ;
        rr:predicate sphn:hasSubjectPseudoIdentifier ],
    [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-OncologyDiagnosis-{sphn:has
IndicationToStart[?(@.target_concept=='https://biomedit.ch/rdf/sphn-schema/sphn#Oncology
Diagnosis')].id}" ] ;
        rr:predicate sphn:hasIndicationToStart ],
    [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-NursingDiagnosis-{sphn:hasI
ndicationToStart[?(@.target_concept=='https://biomedit.ch/rdf/sphn-schema/sphn#NursingDi
agnosis')].id}" ] ;
        rr:predicate sphn:hasIndicationToStart ]
```

As you can see, the class name used for the resource instantiation matches the one defined by the target concept. For the above shown example, the potential classes in the range are all core concepts. This means that they are directly referenced by the id field and there is no additional mapping rule needed. Nevertheless, there are cases where the classes defined by the target concept are not core concepts and therefore need an additional mapping. Such a case is the property `sphn:hasMedicalDevice` of concept `sphn:HeartRateMeasurement` which can be either a `sphn:MedicalDevice` or a `sphn:Implant`. These are not core concepts and the properties defined need an additional mapping to be generated successfully. For example, the second case is mapped still by the target concept property in the following way:

```
:sphnHeartRateMeasurement_sphnhasMedicalDevice_rangesphnImplant a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator
"$.content.sphn:HeartRateMeasurement[*].sphn:hasMedicalDevice[?(@.target_concept=='https
://biomedit.ch/rdf/sphn-schema/sphn#Implant')]" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Software-{sphn:hasSoftware[
*].id}" ] ;
            rr:predicate sphn:hasSoftware ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Implant-{id}-sphn-Code-{sph
n:hasProductCode.termid}" ] ;
            rr:predicate sphn:hasProductCode ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Implant-{id}-sphn-Code-{sph
n:hasProductCode.id}" ] ;
            rr:predicate sphn:hasProductCode ],
        [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Implant-{id}-sphn-Code-{sph
```

```
n:hasTypeCode.termid}" ] ;
            rr:predicate sphn:hasTypeCode ] ;
    rr:subjectMap [ rr:class sphn:Implant ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Implant-{id}" ] .
```

Here the condition on the target concept is defined directly in the `rml:iterator`.

Database schema

In the Database schema the changes are a bit more subtle. An additional layer of nesting has been introduced meaning that the name of the concept (for example “Diagnosis”) is brought into the column names. The further nesting is done as before. For example for table sphn_DrugPrescription we get

- sphn_hasIndicationToStart__BilledDiagnosis__id
- sphn_hasIndicationToStart__Diagnosis__id
- sphn_hasIndicationToStart__NursingDiagnosis__id
- sphn_hasIndicationToStart__OncologyDiagnosis__id

while for the second example, i.e. table sphn_HeartRateMeasurement we get

- sphn_hasMedicalDevice__Implant__id
- sphn_hasMedicalDevice__Implant__sphn_hasProductCode__id
- sphn_hasMedicalDevice__Implant__sphn_hasProductCode__iri
- sphn_hasMedicalDevice__Implant__sphn_hasProductCode__sphn_hasCodingSystemAndVersion
- sphn_hasMedicalDevice__Implant__sphn_hasProductCode__sphn_hasIdentifier
- sphn_hasMedicalDevice__Implant__sphn_hasProductCode__sphn_hasName
- sphn_hasMedicalDevice__Implant__sphn_hasProductCode__termid
- sphn_hasMedicalDevice__Implant__sphn_hasSoftware__id
- sphn_hasMedicalDevice__Implant__sphn_hasSoftware__sphn_hasDescription
- sphn_hasMedicalDevice__Implant__sphn_hasSoftware__sphn_hasName
- sphn_hasMedicalDevice__Implant__sphn_hasSoftware__sphn_hasUniformResourceLocator
- sphn_hasMedicalDevice__Implant__sphn_hasSoftware__sphn_hasVersion
- sphn_hasMedicalDevice__Implant__sphn_hasTypeCode__iri
- sphn_hasMedicalDevice__Implant__sphn_hasTypeCode__termid
- sphn_hasMedicalDevice__MedicalDevice__id
- sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__id
- sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__iri
- sphn_hasMedicalDevice__MedicalDevice__sphn_hasProductCode__sphn_hasCodingSystemAndVersion

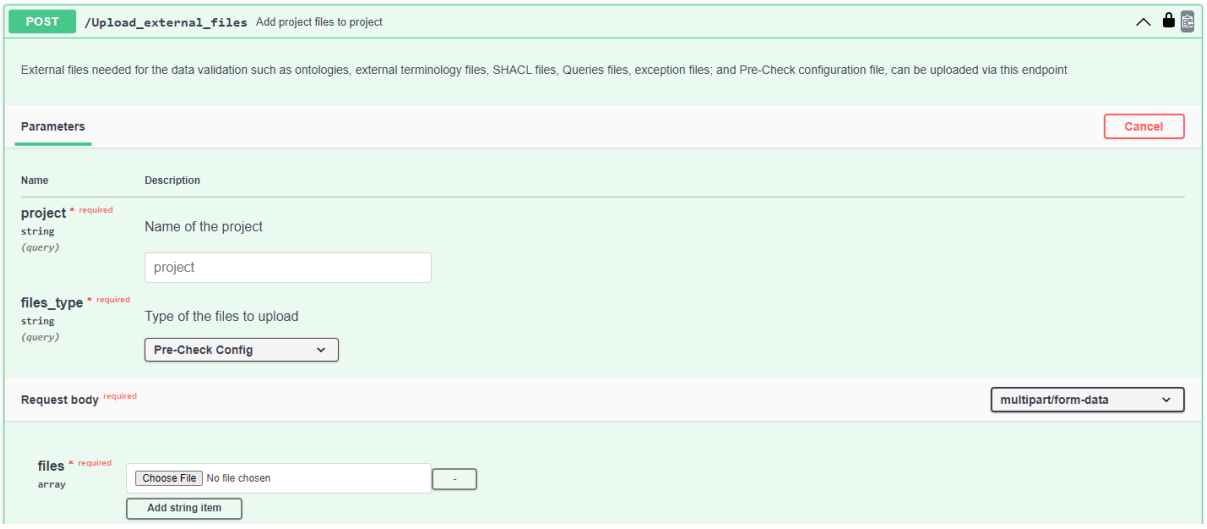
Pre Checks

Introduction

According to the new architecture structure (SPHN_Connector_vision.pptx), Pre-Checks are executed together with De-Identification as the first step after patient data ingestion. The execution takes place between the landing zone and the refined zone. It is mainly meant for JSON files, but it is possible that also some pre-checks for RDF files will be introduced in the future. The pre-checks are configured by the user and should match the predefined structure in order to be executed correctly.

Pre-Checks configuration

The pre-checks for a project are configured via a JSON configuration file. This file needs to be uploaded before project initialization via the `/Upload_external_files` endpoint by setting file type to `Pre-Checks Config`.



The uploaded file is a JSON file which contains the definition of the checks that should be implemented. Checks definitions are divided by check type, meaning that all checks of the same type are grouped under the same key (which is the check type name) and the in the below map, the single checks are defined by check name:

```
{
    "checkType1": {
        "checkName1": {
            "param1": "value1"
        },
        "checkName2": {
            "param1": "value2"
        }
    },
    "checkType2": {
        "checkName3": {
            "param1": "value3",
            "param2": "value4"
```

```
        },
        "checkName4": {
            "param1": "value5",
            "param2": "value6"
        }
    }
}
```

This reflects the main structure of the configuration file. In the next sections, we describe the different types of checks and how to configure them properly.

Note: the configuration data is validated before upload, if the configuration does not match the required structure for the defined pre-checks, then the file is rejected and the user needs to update it before re-uploading.

Severity level

Pre-Checks configuration has a common parameter, the `level` parameter. This is a severity level field, which defines the effect that the check result has on the pipeline execution. The field takes one of the following values:

- **`FAIL`**: if a check fails, the patient's processing is stopped and the error message is reported in the execution logs. The patient data is not moved to the next data persistence layer until the check is removed or the input data is fixed by the user according to the check results.
- **`WARN`**: if a check fails, the patient data is still processed and moved to the next data persistence layer. A warning message with the check failure details will be added to the execution logs.

This attribute defaults to `FAIL`. For the checks which act on the data by replacing some values, the value of the severity level is not important and can be neglected because those checks won't affect the patient processing and the data won't be blocked.

JSON Checks

In this section, we defined all the available check types for JSON input data, meaning JSON ingestion or database ingestion.

regexCheck

The `regexCheck`, checks that a specific attribute matches a predefined regex expression. It is applied to a field which is specified by the user, that field represents the lowest level attribute according to the JSON schema structure.

Structure

JSON structure to use to define a `regexCheck` type pre-check. It defines all the available fields.

```
{
    "regexCheck": {
        "regexCheckName": {
            "level": "",
```

```
            "applies_to_field": "",
            "regex": ""
        }
       }
}
```

Parameters

The following table describes the parameters defined for the `regexCheck`:

Name	Description	Type	Optional	Example
level	Severity level of the check.	str [`FAIL`, `WARN`] Default: `FAIL`	Y	`WARN`
applies_to_field	Field suffix to which the check is applied.	str	N	`id`
regex	Regex expression to check against the defined field	str	N	`^[0-9][a-z][A-Z]$`

applies_to_field: each element of this list represents a path suffix of the JSON data. Each level of nesting is separated by a forward slash to the next one. For example, the path `'id'` applies the rule to all `id` fields, while the path `sphn:Allergy/id` only applies it to the `id` fields of the `sphn:Allergy` concepts

Example

```
{
    "regexCheck": {
        "regexCheck1": {
            "level": "fail",
            "applies_to_field": "sphn:Allergy/id",
            "regex": "^[0-9][a-z][A-Z]$"
        },
        "regexCheck2": {
            "level": "fail",
            "applies_to_field": "termid",
            "regex": "^[0-9][a-z][A-Z]$"
        }
    }
}
```

validationCheck

The `validationCheck`, checks that the input patient data is valid against the JSON schema generated based on the schemas uploaded.

Structure

JSON structure to use to define a `validationCheck` type pre-check. It defines all the available fields.

```
{
    "validationCheck": {
        "validationCheckName": {
            "level": "",
            "check_formats": ""
         }
        }
}
```

Parameters

The following table describes the parameters defined for the `validationCheck`. Ideally, one single validation check should be defined.

Name	Description	Type	Optional	Example
level	Severity level of the check.	str [`FAIL`, `WARN`] Default: `FAIL`	Y	`WARN`
check_formats	Enforces the validation of string fields with defined format in the schema, e.g. 'date-time', 'date', 'time'.	boolean Default: False	Y	`true`

Example

```
{
    "validationCheck": {
        "validationCheck1": {
            "level": "fail",
            "check_formats": false
        }
    }
}
```

replaceCharsCheck

The `replaceCharsCheck`, can't really be defined as a check, but more as a preliminary step which acts on the data before executing the "normal" checks. If checks the values of the predefined field, and in case the defined characters to replace are found, it replaces them directly in the patient data with the provided replacement value.

Structure

JSON structure to use to define a `replaceCharsCheck` type pre-check. It defines all the available fields.

```
{
```

```
    "replaceCharsCheck": {
        "replaceCharsCheckName": {
            "applies_to_field": "",
            "replacement_map": {
                "char_to_replace1": "replacement_value1",
                "char_to_replace2": "replacement_value2",
            }
         }
        }
}
```

Parameters

The following table describes the parameters defined for the `replaceCharsCheck`.

Name	Description	Type	Optional	Example
applies_to_field	Field suffix to which the check is applied.	str	N	`id`
replacement_map	Map of characters to replace	dict	N	`{"/": "_"}`

Example

```
{
    "replaceCharsCheck": {
        "replaceCharsCheck1": {
            "applies_to_field": "id",
            "replacement_map": {
                ":": "_",
                "ü": "ue"
            }
        },
        "replaceCharsCheck2": {
            "applies_to_field": "termid",
            "replacement_map": {
                "/": "_",
                "ä": "ae"
            }
        }
    }
}
```

replaceRegexCheck

The `replaceRegexCheck`, can't really be defined as a check, but more as a preliminary step which acts on the data before executing the "normal" checks. It works similarly as the `replaceCharsCheck`, the difference is that in this case we define a regex expression which defines all the characters that should be replaced with the replacement value.

Structure

JSON structure to use to define a `replaceRegexCheck` type Pre-Check. It defines all the available fields.

```
{
    "replaceRegexCheck": {
        "replaceRegexCheckName": {
            "applies_to_field": "",
            "regex": "",
            "replacement": ""
         }
        }
}
```

Parameters

The following table describes the parameters defined for the `replaceRegexCheck`.

Name	Description	Type	Optional	Example
applies_to_field	Field suffix to which the check is applied.	str	N	`id`
regex	Regex expression defining the characters to replace.	str	N	`[\\s]`
replacement	Replacement value.	Str Default: "_"	Y	`-`

Example

```
{
    "replaceRegexCheck": {
        "replaceRegexCheck1":{
            "applies_to_field": "id",
            "regex": "[\\s]"
        }
    }
}
```

IRIValidationCheck

The `IRIValidationCheck` checks the IRI values provided in JSON or database for the field `id` and `termid`. We expect valid IRIs for these fields otherwise the RDF conversion will fail. This is a default check with severity level `FAIL` which blocks the processing of the patient files in case non-valid IRI values are provided.

Structure

JSON structure to use to define a `replaceRegexCheck` type Pre-Check. It defines all the available fields.

```
{
    "IRIValidationCheck": {
```

```
        "defaultIRIValidationCheck": {
            "level": ""
         }
        }
}
```

Parameters

The following table describes the parameters defined for the `replaceRegexCheck`.

Name	Description	Type	Optional	Example
level	Severity level of the check.	str [`FAIL`, `WARN`] Default: `FAIL`	Y	`WARN`

Example

```
{
    "IRIValidationCheck": {
        "defaultIRIValidationCheck":{
            "level": "warn"
        }
    }
}
```

RDF Checks

In this section, we defined all the available check types for RDF input data.

dataTypeCheck

The `dataTypeCheck` checks values related to a specific type are defined correctly in RDF data. The validation is implemented for the following RDF types: `xsd:dateTime`, `xsd:date`, `xsd:time`, `xsd:double`, `xsd:gDay`, `xsd:gMonth`, `xsd:gYear` where `xsd` is the prefix representing `http://www.w3.org/2001/XMLSchema#`. The data in RDF is checked if the property type is explicitly defined. For example, for property . `sphn:hasAdmissionDateTime "2022-06-02T09:57:00.000Z"^^xsd:dateTime` the value `"2022-06-02T09:57:00.000Z"` is explicitly tested against the `xsd:dateTime` regex pattern.

Structure

JSON structure to use to define a `dataTypeCheck` type pre-check. It defines all the available fields.

```
{
    "dataTypeCheck": {
        "dataTypeCheck1": {
            "level": ""
        }
```

```
    }
}
```

Parameters

The following table describes the parameters defined for the `replaceRegexCheck`.

Name	Description	Type	Optional	Example
level	Severity level of the check.	str [`FAIL`, `WARN`] Default: `FAIL`	Y	`WARN`

Example

```
{
    "dataTypeCheck": {
        "dataTypeCheck1": {
            "level": "warn"
        }
    }
}
```

rdfValidationCheck

The `rdfValidationCheck` checks that the ingested RDF patient files are valid RDF files. This is checked by parsing the turtle file data into a graph. Keep in mind that this operation requires some computational time and memory usage to be performed, especially for large amounts of files and large RDF input files. This check, as are the others, is only applied to input patient data. Configuration files of type RDF are by default validated.

Structure

JSON structure to use to define a `rdfValidationCheck` type Pre-Check. It defines all the available fields.

```
{
    "rdfValidationCheck": {
        "rdfValidationCheckName": {
            "level": ""
        }
    }
}
```

Parameters

The following table describes the parameters defined for the `rdfValidationCheck:`.

Name	Description	Type	Optional	Example
level	Severity level of the check.	str [FAIL, WARN] Default: FAIL	Y	WARN

Example

```
{
    "rdfValidationCheck": {
        "rdfValidationCheckName": {
            "level": "FAIL"
        }
    }
}
```

Default checks

The SPHN Connector is equipped with some default checks that are executed when no external check configuration file is uploaded. It defines a `validationCheck` at `WARN` level and a `replaceCharsCheck` to update the data on-the-fly. To ensure that the IRIs are not corrupted by special characters replacements for German & French special chars are included. For the `termid` field the default checks replace forward slashes and empty spaces.
he default checks configuration is the following:

```
{
    "validationCheck": {
        "defaultValidationCheck": {
            "level": "warn"
            }
    },
     "replaceCharsCheck": {
         "defaultReplaceCharsCheck": {
             "applies_to_field": "id",
            "_comment" : "Replace spaces and forwards slashes with '_'",
            "chars_to_replace": [" ", "/"],
            "replacement": "_"
            },
       "defaultReplaceCharsCheckTermID":{
          "applies_to_field": "termid",
          "_comment" : "Replace spaces and forwards slashes with '_'",
          "chars_to_replace": [" ", "/"],
          "replacement": "_"
       },
            "germanReplaceWithae": {
            "_comment" : "Turn German 'ä' to 'ae'",
            "applies_to_field": "id",
            "chars_to_replace": ["ä"],
            "replacement": "ae"
            },
            "germanReplaceWithAe": {
            "_comment" : "Turn German 'Ä' to 'Ae'",
```

```
"applies_to_field": "id",
"chars_to_replace": ["Ä"],
"replacement": "Ae"
},
"germanReplaceWithue": {
"_comment" : "replace German 'ü' with 'ue'",
"applies_to_field": "id",
"chars_to_replace": ["ü"],
"replacement": "ue"
},
"germanReplaceWithUe": {
"_comment" : "replace German 'Ü' with 'Ue'",
"applies_to_field": "id",
"chars_to_replace": ["Ü"],
"replacement": "Ue"
},
"germanReplaceWithoe": {
"_comment" : "replace German 'ö' with 'oe'",
"applies_to_field": "id",
"chars_to_replace": ["ö"],
"replacement": "oe"
},
"germanReplaceWithOe": {
"_comment" : "replace German 'Ö' with 'oe'",
"applies_to_field": "id",
"chars_to_replace": ["Ö"],
"replacement": "Oe"
},
"germanReplaceWithss": {
"_comment" : " # replace German 'ß' with double 's'",
"applies_to_field": "id",
"chars_to_replace": ["ß"],
"replacement": "ss"
},
"frenchReplacewitha":{
"_comment" : "replace french 'à', 'á' and 'â' with 'a'",
"applies_to_field": "id",
"chars_to_replace": ["à", "á", "â"],
"replacement": "a"
},
"frenchReplaceWithcapitalA":{
"_comment" : "replace french 'À', 'Á' and 'Â' with 'A'",
"applies_to_field": "id",
"chars_to_replace": ["À","Á","Â"],
"replacement": "A"
},
"frenchReplacewithe":{
"_comment" : "replace french 'è' , 'é' and 'ê' with 'e'",
"applies_to_field": "id",
"chars_to_replace": ["è","é", "ê" ],
"replacement": "e"
},
"frenchReplaceWithcapitalE":{
"_comment" : "replace french 'É', 'È' and 'Ê' with 'E'",
"applies_to_field": "id",
"chars_to_replace": ["É","È", "Ê"],
```

```
            "replacement": "E"
            },
            "frenchReplacewithi":{
            "_comment" : "replace french 'ì', 'í' and 'î' with 'i'",
            "applies_to_field": "id",
            "chars_to_replace": ["ì", "í", "î"],
            "replacement": "i"
      },
      "frenchReplaceWithcapitalI":{
      "_comment" : "replace french 'Í', 'Ì' and 'Î' with 'I'",
      "applies_to_field": "id",
      "chars_to_replace": ["Í","Ì", "Î"],
      "replacement": "I"
      },
      "frenchReplacewitho":{
      "_comment" : "replace french 'ò', 'ó' and 'ô' with the letter 'o'",
      "applies_to_field": "id",
      "chars_to_replace": ["ò", "ó", "ô"],
      "replacement": "o"
      },
      "frenchReplaceWithcapitalO":{
 "_comment":"replace french 'Ó', 'Ò' and 'Ô' with the capital letter 'O'",
       "applies_to_field": "id",
        "chars_to_replace": ["Ó", "Ò", "Ô"],
        "replacement": "O"
        },
        "frenchReplacewithu":{
         "_comment" : "replace french 'ú', 'ù' and 'û' with 'u'",
         "applies_to_field": "id",
         "chars_to_replace": ["ú", "ù", "û" ],
         "replacement": "u"
         },
         "frenchReplaceWithcapitalU":{
         "_comment" : "replace french 'Ú', 'Ù' and 'Û' with 'U'",
         "applies_to_field": "id",
         "chars_to_replace": ["Ú", "Ù", "Û"],
         "replacement": "U"
         }
      }
  }
```

Data pre-processing

From RDF schema 2024.1 the Pre-Checks introduces a pre-processing logic which is necessary to correctly map Code and Terminology concepts, and supporting concepts. The pre-processing cannot be disabled and will be executed for every patient based on 2024.1 schema or later. The pre-processing can be divided into two distinct processes: source concepts IDs pushdown and supporting concepts array fields expansion.

Source Concepts IDs pushdown

According to the new Code/Terminology instances naming convention described here, the RML mapping needs to be aware of the source concept ID to generate a correct mapping. This process

creates an artificial field inside the code definition which defines the source concept ID of the upper concept. For example the input data

```
"sphn:hasAllergen": {
    "id": "allergen_id",
    "sphn:hasCode": [
        {
            "id": "test",
            "sphn:hasCodingSystemAndVersion": "test",
            "sphn:hasIdentifier": "test",
            "sphn:hasName": "test"
        }
    ]
}
```

will add the value of `sphn:hasAllergen/id` to the new field `sourceConceptID` inside the `sphn:hasCode` definition. The data stored in the refined zone will look like this

```
"sphn:hasAllergen": {
    "id": "allergen_id",
    "sphn:hasCode": [
        {
            "id": "test",
            "sourceConceptID": "allergen_id",
            "sphn:hasCodingSystemAndVersion": "test",
            "sphn:hasIdentifier": "test",
            "sphn:hasName": "test"
        }
    ]
}
```

The RML mapping will then use the defined value to construct the matching code instance:

```
rr:subjectMap [ rr:class sphn:Code ; rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-sphn-Allergen-{sourceConceptID}-
sphn-Code-{id}" ]
```

This preprocessing won't be applied to supporting concepts codes because there the fields are mandatory.

Array fields expansion

Inside supporting concepts, there are some array properties that were causing issues when defining a proper RML mapping rule. To simplify the mapping, we add a preprocessing step responsible for expanding the array lists of multiple elements to single lists defined in multiple objects. For example the input data for `sphn:Interpretation`

```
[
    {
        "id": "allergyInterpretation",
        "sphn:hasOutput": [
            {
                "id": "allergy01",
                "target_concept": "https://biomedit.ch/rdf/sphn-schema/sphn#Allergy"
            },
            {
                "id": "allergenCode01",
```

```
                "sourceConceptType": "sphn-Allergen",
                "sourceConceptID": "allergen01",
                "target_concept": "https://biomedit.ch/rdf/sphn-schema/sphn#Code"

            }
        ]
    }
]
```

is expanded to

```
[
    {
        "id": "allergyInterpretation",
        "sphn:hasOutput": [
            {
                "id": "allergy01",
                "target_concept": "https://biomedit.ch/rdf/sphn-schema/sphn#Allergy"
            }
        ]
    },
    {
        "id": "allergyInterpretation",
        "sphn:hasOutput": [
            {
                "id": "allergenCode01",
                "sourceConceptType": "sphn-Allergen",
                "sourceConceptID": "allergen01",
                "target_concept": "https://biomedit.ch/rdf/sphn-schema/sphn#Code"

            }
        ]
    }
]
```

such that the list of `sphn:hasOutput` is a single-element list. This preprocessing allows a successful RML mapping.

Pre-Checks deactivation

It is possible to disable the Pre-Checks phase. As described in the previous section, some pre-checks are executed by default. In order to completely deactivate the pre-checks on the input data, the user needs to upload a configuration file with empty JSON data inside, i.e. a file with content `{}`. This user configuration will then overwrite the default checks configuration and no checks will be executed. Preprocessing logic for 2024 RDF schema cannot be disabled.

Multiple checks example

The following configurations define examples of configuration files setting up multiple checks for a project:

JSON Checks

```
{
```

```
    "regexCheck": {
        "regexCheck1": {
            "level": "fail",
            "applies_to_field": "id",
            "regex": "^[0-9][a-z][A-Z]$"
        },
        "regexCheck2": {
            "level": "fail",
            "applies_to_field": "termid",
            "regex": "^[0-9][a-z][A-Z]$"
        }
    },
    "validationCheck": {
        "validationCheck1": {
            "level": "fail",
            "check_formats": false
        }
    },
    "replaceCharsCheck": {
        "replaceCharsCheck1": {
            "applies_to_field": "id",
            "chars_to_replace": [" ", "/"]
        },
        "replaceCharsCheck2": {
            "applies_to_field": "id",
            "chars_to_replace": ["ü"],
            "replacement": "u"
        }
    },
    "replaceRegexCheck": {
        "replaceRegexCheck1":{
            "applies_to_field": "id",
            "regex": "[\\s/]"
        }
    }
}
```

RDF Checks

```
{
    "dataTypeCheck": {
        "dataTypeCheck1": {
            "level": "warn"
        }
    }
}
```

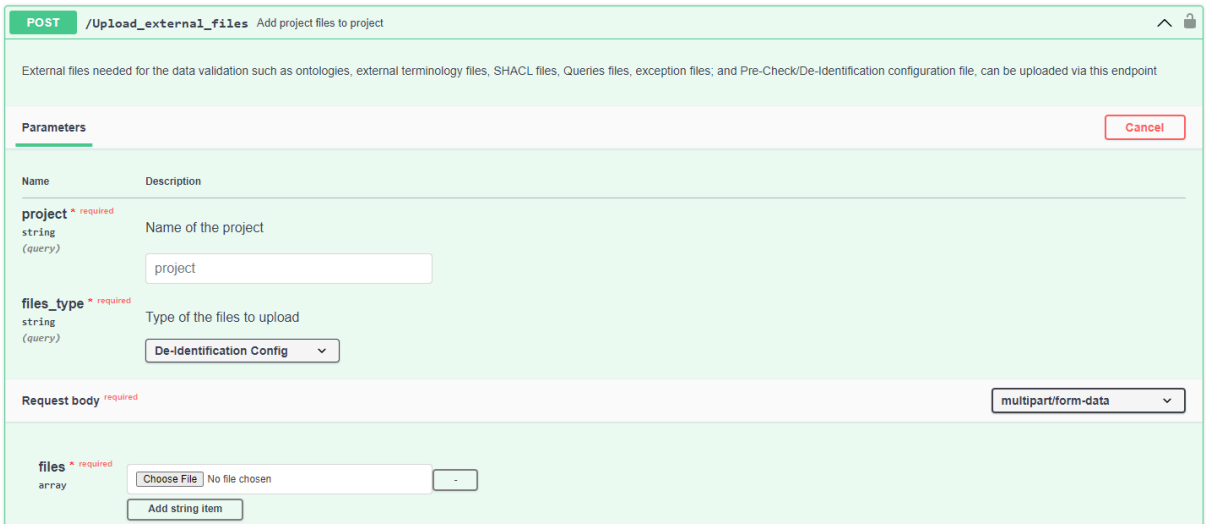
De-Identification

Introduction

According to the new architecture structure (SPHN_Connector_vision.pptx), De-Identification is executed together with Pre-Checks as the first step after patient data ingestion. The execution takes place between the landing zone and the refined zone. De-Identification is applied **only** on JSON input files (JSON ingestion and database ingestion). There are **NO default** De-Identification rules configured. The rules and the fields to which they are applied must be configured manually by the user and should match the predefined structure to be executed correctly.

De-Identification configuration

The de-identification rules for a project are configured via a JSON configuration file. This file needs to be uploaded before project initialization via the `/Upload_external_files` endpoint by setting file type to `De-Identification Config`.



The uploaded file is a JSON file which contains the definition of the de-identification rules to be implemented. Rules are grouped by rule type, meaning that all rules of the same type are defined under the same key representing the rule type. Each single rule is then defined by a rule name.

```
{
    "ruleType1": {
        "ruleName1": {
            "param1": "value1"
        },
        "ruleName2": {
            "param1": "value2"
        }
    },
    "ruleType2": {
        "ruleName3": {
            "param1": "value3",
            "param2": "value4"
```

```
        },
        "ruleName4": {
            "param1": "value5",
            "param2": "value6"
        }
    }
}
```

Note: the configuration data is validated before upload, if the configuration does not match the required structure for the defined de-identification rules, then the file is rejected and the user needs to update it before re-uploading.

Minimum configuration: we recommend applying de-identification to the `sphn:SubjectPseudoIdentifier`, `sphn:AdministrativeCase`, and `sphn:Sample` concepts and a standard date shift. This can be achieved by uploading the following configuration file:

```
{
    "scrambleField": {
        "defaultScrambling": {
            "applies_to_fields": ["sphn:SubjectPseudoIdentifier/id",
"sphn:SubjectPseudoIdentifier/sphn:hasIdentifier", "sphn:AdministrativeCase/id",
"sphn:AdministrativeCase/sphn:hasIdentifier", "sphn:Sample/id",
"sphn:Sample/sphn:hasIdentifier", "sphn:hasSample/id",
"sphn:hasSample/sphn:hasIdentifier"]
        }
    },
    "dateShift": {
        "defaultDateShift": {
            "low_range": -30,
            "high_range": 30
        }
    }
}
```

Value match: with RDF schema 2024.1 many ID references are introduced inside the schema. This means that if we de-identify an ID of a concept, we want to de-identify the referenced IDs as well. To achieve that we check in the data for values matching. For example if the ID of an administrative case "adminCase1" is de-identified, we search for all the occurrences of the value "adminCase1" and de-identify it with the same logic (e.g. it could be present in the value of a `sphn:hasAdministrativeCase` property). The rule is applied only if the match is found at a field with the same name of the original one (e.g. `id` field), or if the matched field is equal to "sourceConceptID" but only in the case when the original field was "id".

De-Identification Rules

In this section, we define the available De-Identification rules.

scrambleField Rule

The `scrambleField` rule generated unique identifiers for the specified fields. The user can pass a list of field path suffixes that needs to be de-identified. The generated de-identified values do not depend on the property value. The SPHN Connector by default keeps track of the de-identified patients, and in case of a re-ingestion, the same unique identifier will be used for the specified field (if it has been de-identified before).

Note: we keep track of the JSON path to which the de-identification has been applied to, e.g. `content/sphn:Allergy/0/id`, this means that even if the property value has been changed the previously generated unique identifier will be used. Assume we have a patient with a list of allergies. We apply a scrambling rule to the id properties of each `sphn:Allergy` concept and we store the generated unique value. If the patient is re-ingested with completely new data (even more or less allergies), then the ids will take the scrambled values previously generated in the same order. When we do not have a list of properties, e.g. sphn:SubjectPseudoIdentifier/sphn:hasIdentifier then we only have a single value that will be replaced. Keep in mind though that if the value in the original data is changed, the de-identified value will still pick the value generated during the first de-identification. Therefore, make sure to have immutable data when de-identifying in case you need to use the de-identified value for identification of the concept.

Note: 2024.1 RDF schema changes bring new logic into the SPHN Connector where all the references to core concepts are defined solely by the `id` field. In older versions, the same logic was applied to the `sphn:hasAdministrativeCase` property. Now it is applied to all properties pointing towards a core concept. Due to this behavior we decided to implement some additional logic in the scrambling de-identification rule such that the matching IDs are de-identified in the same way. When a configured `id` field is de-identified, we look for other `id` fields with the same value in the patient data. If we find a match, we'll replace that field value with the same de-identified value such that the link is maintained.

Anonymization:
The rule can also be configured to generate new sets of values at every patient re-processing. This can be achieved by setting the optional field `anonymize` to *true*. This "forgetting" feature makes sure to remove any connection between the downloaded data and the ingested data. To anonymize the internal patient ID used by the Connector the user **must** add to the list of fields to de-identify the field *sphn:SubjectPseudoIdentifier/id*. When configured, the internal patient ID is changed to the scrambled value and it is used in the filename and in the monitoring tables from refined zone onwards. The link with the original value is removed after moving the patient to the refined zone. Warnings related to this feature

- Let's say we process a patient with ID *p1*. The anonymization creates a new ID *abc123* that is moved through the zones and can be downloaded at the end. If we re-ingest patient *p1*, the data will overlap in the landing zone but in all other zones it will create a new patient with ID *def456* as we do not maintain the link between the anonymized patient and the original patient ID. This could lead to duplicated patient data in the release zone which have different patient identifiers, therefore in case of re-ingestion it could be beneficial to execute a full project reset to cleanup all the already downloaded data.

- The logs extracted for an anonymized patient are split between two log files. The older one is represented by the original patient ID and describes steps up to the refined zone. The newer one will use the anonymized patient ID and describe steps from refined zone onwards.

Anonymization cannot be applied only to the patient ID itself, but to all the string fields and basically allows the generation of new de-identified values at every patient processing.

Structure

JSON structure to use to configure a `scrambleField` de-identification rule:

```
{
    "scrambleField": {
        "scrambleRuleName": {
            "applies_to_fields": [...],
            "anonymized": false
        }
    }
}
```

Parameters

The following table describes the parameters defined for the `scrambleField` rule:

Name	Description	Type	Optional	Example
applies_to_fields	List of field path suffixes to which the rule is applied	List of str	N	`['id', 'sphn:hasIdentifier']`
anonymize	Enable/disable anonymization feature	boolean	Y (Default: false)	`true`

applies_to_field: each element of this list represents a path suffix of the JSON data. Each level of nesting is separated by a forward slash to the next one. For example, the path `'id'` applies the rule to all `id` fields, while the path `sphn:Allergy/id` only applies it to the `id` fields of the `sphn:Allergy` concepts

anonymize: if activated, re-processing of a patient generates new values of the applied fields every time it is triggered.

Example

```
{
    "scrambleField": {
        "scrambleAllergyIDs": {
            "applies_to_fields":
["content/sphn:Allergy/sphn:hasSubstance/sphn:hasCode/id", "content/sphn:Allergy/id"]
        }
    }
}
```

The information about the already executed de-identifications is store for later use in the database internal table `de_identification_scrambling`:

project_name [PK] character varying (200)	data_provider_id [PK] character varying (200)	patient_id [PK] character varying (200)	key_path [PK] character varying (1000)	new_value character varying (50)
123	DATA-PROVIDER-ID	1	content/sphn:Allergy/0/sphn:hasAllergen/sphn:hasCode/0/...	58865622-be37-4663-a616-cbfcf6c00472
123	DATA-PROVIDER-ID	1	content/sphn:Allergy/0/id	a5dc6469-40d5-4c80-9a42-648f64b0e988

dateShift Rule

The `dateShift` rule shifts date-related entities by a random amount of days given a shift range. The rule allows customization in terms of shift range. A day shift is generated for each patient when it is first processed for de-identification. An integer value is picked randomly between the specified range; value 0 is excluded. For each patient, the SPHN Connector keeps track of the day shift applied and in case of a re-ingestion of the same patient it will reuse the existing shift and apply it to the properties' values.

Structure

JSON structure to use to configure a `dateShift` de-identification rule:

```
{
    "dateShift": {
        "dateShiftRuleName": {
            "low_range": "",
            "high_range": ""
        }
    }
}
```

Parameters

The following table describes the parameters defined for the `dateShift` rule:

Name	Description	Type	Optional	Example
low_range	Low range of the day's shift.	integer Default: -30	Y	`-30`
high_range	High range of the day's shift.	integer Default: 30	Y	`30`

low_range: the rule picks randomly a number for the day's shift in between the defined low_range and high_range values. This parameter defines the lowest possible day's shift value.

high_range: the rule picks randomly a number for the day's shift in between the defined low_range and high_range values. This parameter defines the highest possible day's shift value.

shift_g_types: SPHN concepts sometimes contain date information which is defined by means of types `xsd:gYear`, `xsd:gMonth`, and `xsd:gDay`. An example is the `sphn:DeathDate` concept or the `sphn:BirthDate` concept. These properties are treated with some special logic.

Note that the values are shifted only if all the three components (`hasDay`, `hasMonth`, `hasYear`) are specified. If one of them is missing, then no shift is applied to the date.

The information about the applied days shift to patient dates is stored in an internal database table for later use. The table is named `de_identification_shifts` and looks like this:

project_name [PK] character varying (200)	data_provider_id [PK] character varying (200)	patient_id [PK] character varying (200)	shift integer
123	DATA-PROVIDER-ID	1	-11
123	DATA-PROVIDER-ID	2	14

The content of this table is cleaned up when a project is deleted. When the shift rule is applied to a patient, if it has been previously applied on that patient we take the previous shift value from the table, otherwise we use the new shift value at runtime and store it in the table for further usage.

Example

```
{
    "dateShift": {
        "shiftDateTimes": {
            "low_range": -30,
            "high_range": 30
        }
    }
}
```

substituteFieldList Rule

The `substituteFieldList` rule replaces, for a given input field, the field with a replacement string if that value is part of the substitution list provided.

Structure

JSON structure to use to configure a `substituteFieldList` de-identification rule:

```
{
    "substituteFieldList": {
        "substituteFieldListRuleName": {
            "applies_to_field": "",
            "substitution_list": [],
            "replacement": ""
        }
    }
}
```

Parameters

The following table describes the parameters defined for the `substituteFieldList` rule:

Name	Description	Type	Optional	Example

applies_to_field	Field suffix to which the de-identification rule is applied.	str	N	`sphn:Allergy/id`
substitution_list	List of field values that should be replaced.	list of strings	N	`["allergy1", "allergy2"]`
replacement	Replacement string substituting the matched value.	str	N	`ALLERGY_ID`

If the value of the field does not match one of the specified values in the substitution list, then it is not replaced, but still reported into the database `de_identification` table with `substituted=False`. The user can extract this information and update the substitution list.

Example

```
{
    "substituteFieldList": {
        "subIDlist": {
            "applies_to_field": "sphn:Allergy/id",
            "substitution_list": ["allergy1", "allergy2", "allergy3"],
            "replacement": "ALLERGY_ID"
        }
    }
}
```

substituteFieldRegex Rule

The `substituteFieldRegex` rule replaces, for a given input field, the field with a replacement string if that value is matched by the regex expression provided.

Structure

JSON structure to use to configure a `substituteFieldRegex` de-identification rule:

```
{
    "substituteFieldRegex": {
        "substituteFieldRegexRuleName": {
            "applies_to_field": "",
            "regex": "",
            "replacement": ""
        }
    }
}
```

Parameters

The following table describes the parameters defined for the `substituteFieldRegex` rule:

Name	Description	Type	Optional	Example

applies_to_field	Field suffix to which the de-identification rule is applied.	str	N	`sphn:Allergy/id`
regex	Regex expression representing the values that should be replaced.	str	N	`^allergy[1-3]$`
replacement	Replacement string substituting the matched value.	str	N	`ALLERGY_ID`

If the value of the field does not match the regex expression, then it is not replaced, but still reported into the database `de_identification` table with `substituted=False`. The user can extract this information and update the regex expression.

Example

```
{
    "substituteFieldRegex": {
        "substituteFieldRegex": {
            "applies_to_field": "sphn:Allergy/id",
            "regex": "^allergy[1-3]$",
            "replacement": "ALLERGY_ID"
        }
    }
}
```

De-Identification Report

De-Identification changes are tracked on the PostgreSQL database. This allows the user to keep track of the changes and eventually revert them once the patient data has been fully processed and validated. Table `de_identification` on the internal database can be queried by admin users to extract this data. Alternatively, the endpoint `/Get_de_identification_report` can be triggered by the admin users to download a csv/Excel report of the de-identification changes (See /Get_de_identification_report endpoint for more details).

Table de_identification

This table is cleaned up only in the case when a project is deleted. In all the other cases, it is overwritten only if the same primary key is found. The table is defined with the following columns:

de_identification

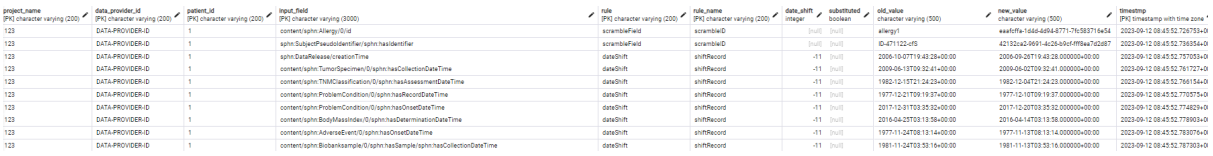
General | Columns | Advanced | Constraints | Parameters | Security | SQL

Inherited from table(s) — Select to inherit from...

Columns

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
project_name	character varying	200		✓	✓
data_provider_id	character varying	200		✓	✓
patient_id	character varying	200		✓	✓
input_field	character varying	3000		✓	✓
rule	character varying	200		✓	✓
rule_name	character varying	200		✓	✓
date_shift	integer				
substituted	boolean				
old_value	character varying	500			
new_value	character varying	500			
timestmp	timestamp with time zone			✓	✓

The following is an example of the data present in the `de_identification` table after applying `scrambleField`, `dateShift`, and `substituteFieldList` rules on an input JSON file:



SPHN Connector utility scripts

In the /utils folder of the SPHN Connector, the users can find some utility script that could help them with specific tasks. The README file describes in detail the utility scripts and how to use them. Please take a look at it if you want to know more.

Brief summary of the available scripts:

- **connector_setup.py**: can be used to create a defined number of projects, upload the external terminologies, and create a defined number of users automatically.
- **validate_json.py**: validate patient JSON file against provided JSON schema.
- **ingest_into_database.py**: import CSV files extracted from data provider database and ingest them into SPHN Connector internal database by means of a tables/columns mapping file.
- **generate_config_file.py** (internal): script used internally by the SPHN Connector to generate the JSON configuration file based on parameters defined in the `.env` file.

Appendix

More detailed table ingestion example

This section provides a more detailed description of how to insert tabular data into the SPHN-Connector. The example is based on RDF schema version **2023.2**. The following is an example of a more detailed / completed patient that can be ingested into the SPHN Connector to showcase the inner workings of the Connector. It is advisable to insert the below sql / rdf statements into an IDE or text editor of your choice for a better visualization. The examples and description were kindly provided by USZ. There is also a graphical view of the mapping available. You will find the file here: DrugAdministrationEvent . Keep in mind that this Diagram is quite large. It is beneficial to download the file and load it into drawIO which you can find here: drawIO. The section below will be helpful to understand the mappings in the Diagram better: explanation for graphical representation

Introduction

In order to correctly populate the columns in the SPHN Connector in the Postgres DB some rules apply how they have to be formatted. In order to understand the different parts we have to understand the following definitions:

RDF consists of triplets that are called subject, predicate, object or s,p,o for short.

- the subject defines the thing/resource that is described by giving it a unique resource id
- the predicate is defining the property for which the object states the value
- the object can either be a data type property (i.e. a value like a string, datetime, double) or an object property (i.e. it contains the resource id of another subject it links to)
- the term resource will be used for everything that will have an instantiation to denote the id of this subject

Resources

in the SPHN data set we also have to different types of resources/concepts:

Shared resources/concepts

Shared resources correspond to data that can be provided by different data providers with the same meaning and therefore the same identifier. These resources are quite generic type of information and generally do not depend on any date time.

The list of concepts shared across data providers in the SPHN schema are the ones that only have a Code or value from a SPHN valueset

For instances of SPHN Concepts that only have a Code information, it is recommended to generate IRIs following this convention:

resource:<ClassName>-<coding_system>-<identifier>

where coding_system should be provided following conventions written here and unique_id corresponds to the unique identifier coming from the coding system.

Unique (non-shared) resources/concepts

Resource that must be unique for each data provider must follow these conventions:

resource:<provider_id>-<ClassName>-<identifier>

where unique_id is a unique identifier defined by the data provider. In the USZ we usually use newid() to generate it.

Different types of columns

In the SPHN Connector there are different kind of columns that have their own formating rules:

type	description	example
xxx_iri	All the columns ending in _iri have to contain a full IRI of the code/resource..	https://biomedit.ch/rdf/sphn-ontology/sphn#LessThan
xxx_termid	Termid is used for codes that are expressed as an terminology entry and are in RDF Term represented as an ObjectProperty	SNOMED-CT-395009001
xxx_id	The ID is the unique ID of the subject. For unique resources it is usually a newid() For shared concepts that are NOT expressed a terminology it could be coding_system + code or any combination of values that make the entry unique.	C04920AD-B2C8-493C-B1B9-58198F7A30E0 or GTIN-7680362030131 50-mg

Properties/Concepts with two types of Code

Instantiate as Code	Instantiate as Terminology (ATC, CHOP, ICD-10, SNOMED, LOINC etc.)
The following columns need to be filled	
sphn_hasCode__id sphn_hasCodingSystemAndVersion sphn_hasCode__sphn_hasIdentifier sphn_hasCode__sphn_hasName	sphn_hasCode__termid sphn_hasCode__iri

Type definitions in SPHN Connector/Postgres

In the SPHN Connector Postgres database there are some special types generated for properties that have an SPHN defined valueset. These are set for each corresponding column in the database and are in most of the cases related to the comparator column. As a consequence the values in these columns have to comply with these types.

Example:

the following column has defined the below type:

column: **sphn_hasDrugQuantity__sphn_hasComparator__iri**

```
"Hospfair".sphn_DrugAdministrationEvent__sphn_hasDrugQuantity__sphn_hasComparator__iri_type, sphn_hasDrugQuantity__sphn_hasUnit__id varchar(3000),
```

type:

```
"Hospfair".sphn_DrugAdministrationEvent__sphn_hasDuration__sphn_hasComparator__iri_type AS ENUM ('https://biomedit.ch/rdf/sphn-ontology/sphn#Equal',
'https://biomedit.ch/rdf/sphn-ontology/sphn#GreaterThan',
'https://biomedit.ch/rdf/sphn-ontology/sphn#GreaterThanOrEqual',
'https://biomedit.ch/rdf/sphn-ontology/sphn#LessThan',
'https://biomedit.ch/rdf/sphn-ontology/sphn#LessThanOrEqual');
```

Id definitions

To define a unique ID is a balance between creating as few instances of a class as possible and making sure they have a unique key. If two different instances have the same unique_ID but the have different properties the properties will be merged into one instance which will result in an incorrect representation of the data. Usually a unique_ID can be created by including all the properties of an instance into the unique_ID. However, this can be cumbersome and error prone which is why we use newid() to generate a unique_ID where a manually defined unique_ID can not easily be achieved like with code instances.

Following list proposes how to populate the ID in order to generate a unique instance ID

Concept	ID (Unique_ID)	Example	Remark
Access Device Presence	newid()	6D4E31E4-AD56-45E7-BAE3-98F390656FCF	
Administrative Case	newid()		
Administrative Gender	{subjectPseudoId}		only one entry per patient.
Adverse Event	newid()		

Concept	ID (Unique_ID)	Example	Remark
Age	newid()		
Allergen	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Allergy	newid()		
Allergy Episode	newid()		
Biobanksample	newid()		
Birth Date	{year}-{month}-{day}-{comperator}	1943-05-01-lessThan 1995-10-13	born before 1.5.1943 born on 13.10.1995
Blood Pressure	newid()		

Concept	ID (Unique_ID)	Example	Remark
Body Height	newid()		
Body Mass Index	newid()		
Body Position	newid()		
Body Site	{codingSystemVersion}-{BodySeiteCode}-{LateralityCode}	SNOMED-CT-253452-1232124	
Body Surface Area	newid()		
Body Temperature	newid()		
Body Weight	newid()		

Concept	ID (Unique_ID)	Example	Remark
Cardiac Index	newid()		
Cardiac Output	newid()		
Care Handling	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Chromosomal Location	newid()		
Chromosome	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Circumference Measure	newid()		
Civil Status	newid()		

Concept	ID (Unique_ID)	Example	Remark
Code	{codingSystemVersion}-{Code}	icd-10-gm-2021-A06.6 GTIN-7680362030131	
Consent	newid()		
Data Determination	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Data File	newid()		
Data Provider Institute	`CHE_108_904_325`		fixed value (e. g. `CHE_108_904_325` for USZ)
Death Date	{year}-{month}-{day}-{time}		

Concept	ID (Unique_ID)	Example	Remark
Death Status	419099009-{year}-{month}-{day}-{time}		
Diagnosis	n/a		Not instantiated-Parent Class
Diagnostic Radiologic Examination	newid()		
Drug	newid()		
Drug Administration Event	newid()		
Drug Prescription	newid()		

Concept	ID (Unique_ID)	Example	Remark
Electrocardiogram	newid()		
Electrocardiographic Procedure	newid()		
FOPH Diagnosis	newid()		
FOPH Procedure	newid()		
Gene	newid()		
Genetic Variation	newid()		
Genomic Position	newid()		

Concept	ID (Unique_ID)	Example	Remark
Gestational Age At Birth	newid()		
Healthcare Encounter	newid()		
Heart Rate	newid()		
ICD-O Diagnosis	newid()		
Inhaled Oxygen Concentration	newid()		
Intent	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Lab Analyzer	{TypeCode}-{ProductCode}	AD32034-039234	

Concept	ID (Unique_ID)	Example	Remark
Lab Result	newid()		
Lab Test	newid()		
Laterality	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Location	{TypeCode}-{Exact}	12342342-ORL 3342342-Home 9242342-CareHome	Due to de-identification reason the exact is never exact but an abstraction of the exact location like OE.
Measurement	n/a		Not instantiated-Parent Class
Measurement Method	{codingSystemVersion}-{Code}	SNOMED-CT-253452	

Concept	ID (Unique_ID)	Example	Remark
Medical Device	{TypeCode}-{ProductCode}	AD32034-039234	Usually not instantiated-Parent Class
Nursing Diagnosis	newid()		
Oncology Treatment Assessment	newid()		
Organism	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Oxygen Saturation	newid()		
Pharmaceutical Dose Form	{codingSystemVersion}-{Code}	SNOMED-CT-253452	

Concept	ID (Unique_ID)	Example	Remark
Physiologic State	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Problem Condition	newid()		
Procedure	n/a		Not instantiated-Parent Class
Protein	{codingSystemVersion}-{Code}-{OrganismCode}		
Quantity	{Amount}-{Unit}-{comparator}	50-mg-LessThan 50-mg	when exact value (equal) the comparator can be omitted.
Radiotherapy Procedure	newid()		

Concept	ID (Unique_ID)	Example	Remark
Reference	NCBI-{Code}	NCBI-2343wer	
Reference Range	{LowerAmount}-{UpperAmount}-{Unit}	10-54-permin	
Respiratory Rate	newid()		
Sample	newid()		
Simple Score	newid()		
Single Nucleotide Variation	newid()		

Concept	ID (Unique_ID)	Example	Remark
Subject Pseudo Identifier	newid()		
Substance	newid()		
Therapeutic Area	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
Time Pattern	{codingSystemVersion}-{Code}	SNOMED-CT-253452	
TNM Classification	newid()		
Transcript	newid()		
Tumor Grade	newid()		

Concept	ID (Unique_ID)	Example	Remark
Tumor Specimen	newid()		
Tumor Stage	newid()		
Unit	{UCUMUnit}	mg	
Variant Descriptor	newid()		
Variant Notation	{codingSystemVersion}-{Value}		

Bringing it all together

Last but not least you'll find in this section a script that will insert data into the SPHN-Connector including the expected output (that the SPHN Connector will produce) in an rdf format.

NOTE: Before you can use the provided example and compare the output you need / might need to adapt the project_name ("Hospfair") and maybe the value for the DataProviderInstitute ('CHE_108_904_325').

SQL Insert Statement

```
INSERT INTO "Hospfair"."sphn_DataProviderInstitute" (
        id,
        "sphn_hasCode__id",
        "sphn_hasCode__sphn_hasIdentifier",
        "sphn_hasCode__sphn_hasName",
  "sphn_hasCode__sphn_hasCodingSystemAndVersion"
        )
        VALUES (
        'CHE_108_904_325',
     'CHE_108_904_325',
    'CHE_108_904_325',
     'University Hospital Zürich',
    'UID');

INSERT INTO "Hospfair"."sphn_SubjectPseudoIdentifier" (
        id,
        "sphn_hasIdentifier"
        )
        VALUES (
        '6D4E31E4-AD56-45E7-BAE3-98F390656FCF',
        '6D4E31E4-AD56-45E7-BAE3-98F390656FCF'
        );

INSERT INTO "Hospfair"."sphn_AdministrativeCase"(
        id,
        "sphn_hasAdmissionDateTime",
        "sphn_hasCareHandling__id",
        "sphn_hasCareHandling__sphn_hasTypeCode__iri",
        "sphn_hasCareHandling__sphn_hasTypeCode__termid",
        "sphn_hasDataProviderInstitute",
        "sphn_hasDischargeDateTime",
        "sphn_hasDischargeLocation__id",
```

```
        "sphn_hasDischargeLocation__sphn_hasExact",
        "sphn_hasDischargeLocation__sphn_hasTypeCode__iri",
        "sphn_hasDischargeLocation__sphn_hasTypeCode__termid",
        "sphn_hasIdentifier",
        "sphn_hasOriginLocation__id",
        "sphn_hasOriginLocation__sphn_hasExact",
        "sphn_hasOriginLocation__sphn_hasTypeCode__iri",
        "sphn_hasOriginLocation__sphn_hasTypeCode__termid",
        "sphn_hasSubjectPseudoIdentifier"
        )
        VALUES (
        'CAE9ECC4-BE4C-4BFC-BCD8-B1EC4E69FFE8',
        '2020-04-29T19:00:00+00:00',
        '394656005',
        'http://snomed.info/id/394656005',
        'SNOMED-CT-394656005',
        'CHE_108_904_325',
        '2020-04-29T19:00:00+00:00',
        '264362003',
        null,
        'http://snomed.info/id/264362003',
        'SNOMED-CT-264362003',
        'CAE9ECC4-BE4C-4BFC-BCD8-B1EC4E69FFE8',
        '264362003',
        null,
        'http://snomed.info/id/264362003',
        'SNOMED-CT-264362003',
        '6D4E31E4-AD56-45E7-BAE3-98F390656FCF'
        );

INSERT INTO "Hospfair"."sphn_DrugAdministrationEvent" (
        "sphn_hasSubjectPseudoIdentifier",
        "sphn_hasDataProviderInstitute",
    "id",
        "sphn_hasAdministrativeCase__id",
        "sphn_hasEndDateTime",
        "sphn_hasStartDateTime",
        "sphn_hasReasonToStopCode__termid",
        "sphn_hasReasonToStopCode__iri",
        "sphn_hasTimePattern__id",
        "sphn_hasTimePattern__sphn_hasTypeCode__termid",
        "sphn_hasTimePattern__sphn_hasTypeCode__iri",
        "sphn_hasDrug__id",
        "sphn_hasDrug__sphn_hasActiveIngredient__id",
        "sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasGenericName",
```

```
        "sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasQuantity__id",
        "sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasQuantity__sphn_hasValue",
        "sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasQuantity__sphn_hasComparator__iri",
        "sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasQuantity__sphn_hasUnit__id",
"sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasQuantity__sphn_hasUnit__sphn_hasCode__iri”
,
 "sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasCode__id",
"sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasCode__sphn_hasCodingSystemAndVersion",
"sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasCode__sphn_hasIdentifier",
"sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasCode__sphn_hasName",
"sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasCode__termid",
"sphn_hasDrug__sphn_hasActiveIngredient__sphn_hasCode__iri",
"sphn_hasDrug__sphn_hasInactiveIngredient__id",
"sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasGenericName",
"sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasQuantity__id",
"sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasQuantity__sphn_hasValue",
"sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasQuantity__sphn_hasComparator__iri",
"sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasQuantity__sphn_hasUnit__id",
"sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasQuantity__sphn_hasUnit__sphn_hasCode__ir
i",
  "sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasCode__id",
"sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasCode__sphn_hasCodingSystemAndVersion",
        "sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasCode__sphn_hasIdentifier",
        "sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasCode__sphn_hasName",
        "sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasCode__termid",
        "sphn_hasDrug__sphn_hasInactiveIngredient__sphn_hasCode__iri",
        "sphn_hasDrug__sphn_hasProductCode__id",
        "sphn_hasDrug__sphn_hasProductCode__sphn_hasCodingSystemAndVersion",
        "sphn_hasDrug__sphn_hasProductCode__sphn_hasIdentifier",
        "sphn_hasDrug__sphn_hasProductCode__sphn_hasName",
        "sphn_hasDrug__sphn_hasManufacturedDoseForm__id",
        "sphn_hasDrug__sphn_hasManufacturedDoseForm__sphn_hasCode__termid",
        "sphn_hasDrug__sphn_hasManufacturedDoseForm__sphn_hasCode__iri",
        "sphn_hasDrug__sphn_hasManufacturedDoseForm__sphn_hasCode__id",
"sphn_hasDrug__sphn_hasManufacturedDoseForm__sphn_hasCode__sphn_hasCodingSystemAndVersion",
        "sphn_hasDrug__sphn_hasManufacturedDoseForm__sphn_hasCode__sphn_hasIdentifier",
        "sphn_hasDrug__sphn_hasManufacturedDoseForm__sphn_hasCode__sphn_hasName",
        "sphn_hasDrugQuantity__id",
        "sphn_hasDrugQuantity__sphn_hasValue",
        "sphn_hasDrugQuantity__sphn_hasComparator__iri",
        "sphn_hasDrugQuantity__sphn_hasUnit__id",
        "sphn_hasDrugQuantity__sphn_hasUnit__sphn_hasCode__iri",
        "sphn_hasAdministrationRouteCode__termid",
        "sphn_hasAdministrationRouteCode__iri",
        "sphn_hasDuration__id",
```

```
"sphn_hasDuration__sphn_hasValue",
"sphn_hasDuration__sphn_hasComparator__iri",
"sphn_hasDuration__sphn_hasUnit__id",
"sphn_hasDuration__sphn_hasUnit__sphn_hasCode__iri"
)
VALUES (
'6D4E31E4-AD56-45E7-BAE3-98F390656FCF',
'CHE_108_904_325',
'0002F9B1-4C89-4E93-BF37-6EC02EB5E4E0',
'CAE9ECC4-BE4C-4BFC-BCD8-B1EC4E69FFE8',
'2020-04-29T19:00:00+00:00',
'2020-04-29T19:00:00+00:00',
'SNOMED-CT-395009001',
'http://snomed.info/id/395009001',
'7087005',
'SNOMED-CT-7087005',
'http://snomed.info/id/7087005',
'0002F9B1-4C89-4E93-BF37-6EC02EB5E4E0',
'N05BA06',
'Lorazepam',
'50-mg',
'50',
null,
'mg',
'https://biomedit.ch/rdf/sphn-resource/ucum/mg',
null,
null,
null,
null,
'ATC-N05BA06',
'https://www.whocc.no/atc_ddd_index/?code=N05BA06',
'N05BA06',
'Lorazepam',
'50-mg',
'50',
null,
'mg',
'https://biomedit.ch/rdf/sphn-resource/ucum/mg',
null,
null,
null,
null,
'ATC-N05BA06',
'https://www.whocc.no/atc_ddd_index/?code=N05BA06',
'GTIN-7680362030131',
```

```
        'GTIN',
        '7680362030131',
        '{Product Name if available}',
        '385055001',
        'SNOMED-CT-385055001',
        'http://snomed.info/id/385055001',
        null,
        null,
        null,
        null,
        '1.000cblpiececbreq',
        '1.00',
        null,
        'cblpiececbr',
        'https://biomedit.ch/rdf/sphn-resource/ucum/cblpiececbr',
        'SNOMED-CT-26643006',
        'http://snomed.info/id/26643006',
        '1.000min',
        '1.000',
        null,
        'min',
        'https://biomedit.ch/rdf/sphn-resource/ucum/min'
        );

INSERT INTO "Hospfair"."sphn_DataRelease" (
        "sphn_hasSubjectPseudoIdentifier",
  "sphn_hasDataProviderInstitute",
  "id",
  "creationTime"
        )
  VALUES (
        '6D4E31E4-AD56-45E7-BAE3-98F390656FCF',
     'CHE_108_904_325',
    '6D4E31E4-AD56-45E7-BAE3-98F390656FCF',
    '2023-04-27T13:00:00.000'
        );
```

RDF Output

```
@prefix dct: <http://purl.org/dc/terms/> .

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix resource: <https://biomedit.ch/rdf/sphn-resource/> .

@prefix snomed: <http://snomed.info/id/> .

@prefix sphn: <https://biomedit.ch/rdf/sphn-ontology/sphn#> .

@prefix ucum: <https://biomedit.ch/rdf/sphn-resource/ucum/> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

 resource:CHE_108_904_325-AdministrativeCase-CAE9ECC4-BE4C-4BFC-BCD8-B1EC4E69FFE8 a

        sphn:AdministrativeCase;

  sphn:hasAdmissionDateTime "2020-04-29T19:00:00+00:00"^^xsd:dateTime;

 sphn:hasCareHandling resource:CHE_108_904_325-CareHandling-394656005;

  sphn:hasDataProviderInstitute
resource:CHE_108_904_325-DataProviderInstitute-CHE_108_904_325;

   sphn:hasDischargeDateTime "2020-04-29T19:00:00+00:00"^^xsd:dateTime;

  sphn:hasDischargeLocation resource:CHE_108_904_325-Location-264362003;

 sphn:hasIdentifier "CAE9ECC4-BE4C-4BFC-BCD8-B1EC4E69FFE8";

 sphn:hasOriginLocation resource:CHE_108_904_325-Location-264362003;

 sphn:hasSubjectPseudoIdentifier
resource:CHE_108_904_325-SubjectPseudoIdentifier-6D4E31E4-AD56-45E7-BAE3-98F390656FCF .

resource:CHE_108_904_325-CareHandling-394656005 a sphn:CareHandling;

   sphn:hasTypeCode resource:Code-SNOMED-CT-394656005 .

resource:CHE_108_904_325-Code-CHE_108_904_325 a sphn:Code;

 sphn:hasCodingSystemAndVersion "UID";

 sphn:hasIdentifier "CHE_108_904_325";

 sphn:hasName "University Hospital Zürich" .

resource:CHE_108_904_325-Code-GTIN-7680362030131 a sphn:Code;
```

```
 sphn:hasCodingSystemAndVersion "GTIN";

 sphn:hasIdentifier "7680362030131";

 sphn:hasName "{Product Name if available}" .

resource:CHE_108_904_325-DataProviderInstitute-CHE_108_904_325 a sphn:DataProviderInstitute;

 sphn:hasCode resource:CHE_108_904_325-Code-CHE_108_904_325 .

resource:CHE_108_904_325-DataRelease-6D4E31E4-AD56-45E7-BAE3-98F390656FCF a
sphn:DataRelease;

  dct:conformsTo <https://biomedit.ch/rdf/sphn-ontology/sphn/2023/2>;

 sphn:hasDataProviderInstitute
resource:CHE_108_904_325-DataProviderInstitute-CHE_108_904_325;

 sphn:hasExtractionDateTime "2023-04-27T13:00:00+00:00"^^xsd:dateTime .

resource:CHE_108_904_325-Drug-0002F9B1-4C89-4E93-BF37-6EC02EB5E4E0 a sphn:Drug;

 sphn:hasActiveIngredient resource:CHE_108_904_325-Substance-N05BA06;

 sphn:hasInactiveIngredient resource:CHE_108_904_325-Substance-ATC-N05BA06;

 sphn:hasManufacturedDoseForm resource:CHE_108_904_325-PharmaceuticalDoseForm-385055001;

 sphn:hasProductCode resource:CHE_108_904_325-Code-GTIN-7680362030131 .

resource:CHE_108_904_325-DrugAdministrationEvent-0002F9B1-4C89-4E93-BF37-6EC02EB5E4E0

  a sphn:DrugAdministrationEvent;

 sphn:hasAdministrationRouteCode resource:Code-SNOMED-CT-26643006;

 sphn:hasAdministrativeCase
resource:CHE_108_904_325-AdministrativeCase-CAE9ECC4-BE4C-4BFC-BCD8-B1EC4E69FFE8;

 sphn:hasDataProviderInstitute
resource:CHE_108_904_325-DataProviderInstitute-CHE_108_904_325;

 sphn:hasDrug resource:CHE_108_904_325-Drug-0002F9B1-4C89-4E93-BF37-6EC02EB5E4E0;

 sphn:hasDrugQuantity resource:CHE_108_904_325-Quantity-1.000cblpiececbreq;

 sphn:hasDuration resource:CHE_108_904_325-Quantity-1.000min;

 sphn:hasEndDateTime "2020-04-29T19:00:00+00:00"^^xsd:dateTime;
```

```
 sphn:hasReasonToStopCode resource:Code-SNOMED-CT-395009001;

 sphn:hasStartDateTime "2020-04-29T19:00:00+00:00"^^xsd:dateTime;

 sphn:hasSubjectPseudoIdentifier
resource:CHE_108_904_325-SubjectPseudoIdentifier-6D4E31E4-AD56-45E7-BAE3-98F390656FCF;

 sphn:hasTimePattern resource:CHE_108_904_325-TimePattern-7087005 .

resource:CHE_108_904_325-Location-264362003 a sphn:Location;

 sphn:hasDataProviderInstitute
resource:CHE_108_904_325-DataProviderInstitute-CHE_108_904_325;

 sphn:hasTypeCode resource:Code-SNOMED-CT-264362003 .

resource:CHE_108_904_325-PharmaceuticalDoseForm-385055001 a sphn:PharmaceuticalDoseForm;

 sphn:hasCode resource:Code-SNOMED-CT-385055001 .

 resource:CHE_108_904_325-Quantity-1.000cblpiececbreq a sphn:Quantity;

   sphn:hasUnit resource:CHE_108_904_325-Unit-cblpiececbr;

  sphn:hasValue 1.0E0 .

 resource:CHE_108_904_325-Quantity-1.000min a sphn:Quantity;

 sphn:hasUnit resource:CHE_108_904_325-Unit-min;

 sphn:hasValue 1.0E0 .

resource:CHE_108_904_325-Quantity-50-mg a sphn:Quantity;

   sphn:hasUnit resource:CHE_108_904_325-Unit-mg;

 sphn:hasValue 5.0E1 .

 resource:CHE_108_904_325-SubjectPseudoIdentifier-6D4E31E4-AD56-45E7-BAE3-98F390656FCF

 a sphn:SubjectPseudoIdentifier;

 sphn:hasDataProviderInstitute
resource:CHE_108_904_325-DataProviderInstitute-CHE_108_904_325;

 sphn:hasIdentifier "6D4E31E4-AD56-45E7-BAE3-98F390656FCF" .

resource:CHE_108_904_325-Substance-ATC-N05BA06 a sphn:Substance;

 sphn:hasCode resource:Code-ATC-N05BA06;
```

```
 sphn:hasGenericName "Lorazepam";

 sphn:hasQuantity resource:CHE_108_904_325-Quantity-50-mg .

resource:CHE_108_904_325-Substance-N05BA06 a sphn:Substance;

 sphn:hasCode resource:Code-ATC-N05BA06;

 sphn:hasGenericName "Lorazepam";

 sphn:hasQuantity resource:CHE_108_904_325-Quantity-50-mg .

resource:CHE_108_904_325-TimePattern-7087005 a sphn:TimePattern;

 sphn:hasTypeCode resource:Code-SNOMED-CT-7087005 .

resource:CHE_108_904_325-Unit-cblpiececbr a sphn:Unit;

 sphn:hasCode ucum:cblpiececbr .

resource:CHE_108_904_325-Unit-mg a sphn:Unit;

 sphn:hasCode ucum:mg .

resource:CHE_108_904_325-Unit-min a sphn:Unit;

   sphn:hasCode ucum:min .

 resource:Code-ATC-N05BA06 a <https://www.whocc.no/atc_ddd_index/?code=N05BA06> .

 resource:Code-SNOMED-CT-264362003 a snomed:264362003 .

 resource:Code-SNOMED-CT-26643006 a snomed:26643006 .


resource:Code-SNOMED-CT-385055001 a snomed:385055001 .

resource:Code-SNOMED-CT-394656005 a snomed:394656005 .

resource:Code-SNOMED-CT-395009001 a snomed:395009001 .

resource:Code-SNOMED-CT-7087005 a snomed:7087005 .
```

Graphical representation of property mapping to column names or vice versa

It might also be helpful to have a graphical representation for the necessary mappings. For this purpose we provide an example mapping for the DrugAdministrationEvent. You will find the file here: DrugAdministrationEvent . Keep in mind that this Diagram is quite large. It is beneficial to download the file and load it into drawIO which you can find here: drawIO. The section below will be helpful to understand the mappings in the Diagram better.

How to read the diagram:

- The rounded boxes (in blue) are concepts that will be instantiated. The light blue ones have a special treatment which is why there are no additional properties left from them.
- Every Concept instance has an Identifier column. This does not show up in the Excel of the SPHN Dataset as these are the identifier values added to the subject resource after the concept name (e.g. resource:CHE-108_904_325-Drug-**0002F9B1-4C89-4E93-BF37-6EC02EB5E4E0)**
- The grey boxes represent data type properties, i.e. datetime, string and numbers.
- The red boxes represent IRIs for SPHN defined valuesets (e.g. Comperator) and for UCUM units. The UCUM terms are not instantiated as classes and they have to be represented differently than the other terminologies like SNOMEDCT, ICD-10, CHOP; ATC etc.
- Uncolored boxes are not represented in the Postgres table. This will be generated from the concept they link to.

Codes

- The green boxes are used to represent the codes that link to a terminology.
- The yellow ones are used to represent the code with data type properties and they do NOT link to a terminology directly, i.e. the Terminology is not provided as an RDF Taxonomy.
- As you can see sometimes the Code has only green boxes as the value set defined must be a code from a terminology provided, mostly SNOMEDCT.
- At some other occasions there are only grey boxes as there is not official terminology available like GTIN or TNM
- Some concepts allow to code the property either as a term from a terminology (green) or as a code (grey) but you should only use one of the options and not both. If a term from a terminology is available it is the preferred version.

Special cases

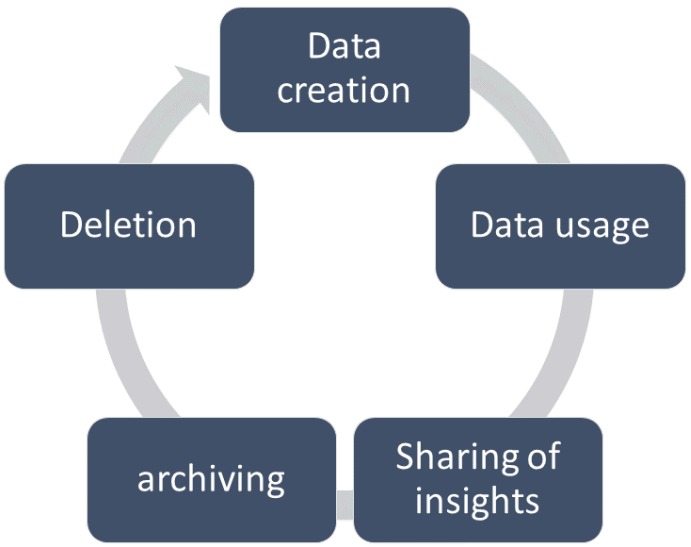
- sometimes you see in the columns an _id, _termId and _iri column like for example with TimePattern. However, if you check closely you can see that these are two different concepts: TimePattern and Code. Some Concepts only have one property like the code from TimePattern and this can be confusing.

- Cardinalities are not only given per property but also per class. This means whenever you instantiate an optional Concept you have to comply with the cardinality of the properties for this concept you are instantiating.
- Cardinalities sometimes look like everything is optional. This is due to technical restraint reasons and you should instantiate all you can even if it is optional. For example a Substance only with the identifier doesn't make sense and code and quantity should always provided where possible.

Data Lifecycle

The SPHN Connector is not meant as a permanent storage solution for projects. Therefore it is highly recommended to implement a continuous data assessment and cleanup for the projects configured and used inside of the SPHN connector. The practice of doing so is generally referred to as Data lifecycle / data lifecycle management. While there are many terms and steps coined by different vendors and providers they all share the same principles:

1. Data should be created in a sensible fashion (according to existing law and in consideration of data protection) and serve a purpose
2. Data should serve an information need and should be used accordingly
3. Insights generated from the data should be shared with the appropriate stakeholders / audience
4. Information or data that is important should be archived and filed appropriately (also in harmony with legal requirements)
5. Data that is not needed anymore should be destroyed



The key takeaways for the user of the SPHN connector should be the following:

- created projects should always have a “purpose” and should not create data to simply create data (testing purposes are obviously excluded from this recommendation)
- projects that are not considered as “hot” anymore (not frequently used / accessed anymore) should be reviewed for archiving or deletion
- projects that can be archived should be archived and moved out of the connector (via e. g. `/Export_project` endpoint) and stored appropriately
- projects that are needed again can be re-import via the `/Import_project` endpoint
- data that is not used anymore should be reviewed for archiving or deletion

Access pgadmin

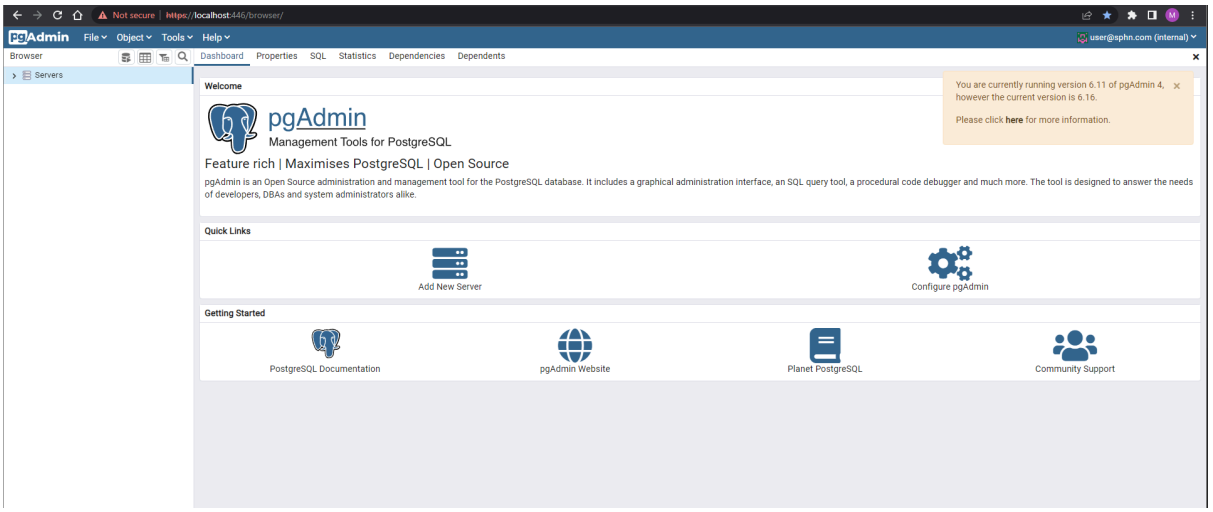
This section will discuss basic interactions and the navigation inside the pgadmin service. Unfortunately the pictures are very small. For more details you can either review the visual user guide for pgadmin in this presentation: SPHN-Connector_pgadmin. For even more details please consult the official pgadmin documentation: pgadmin documentation.

1) Navigate to your pgadmin instance

You find access to pgadmin at https://{your-server}:1443/pgadmin (screenshots url from previous version). Enter your credentials to get to the “welcome screen”

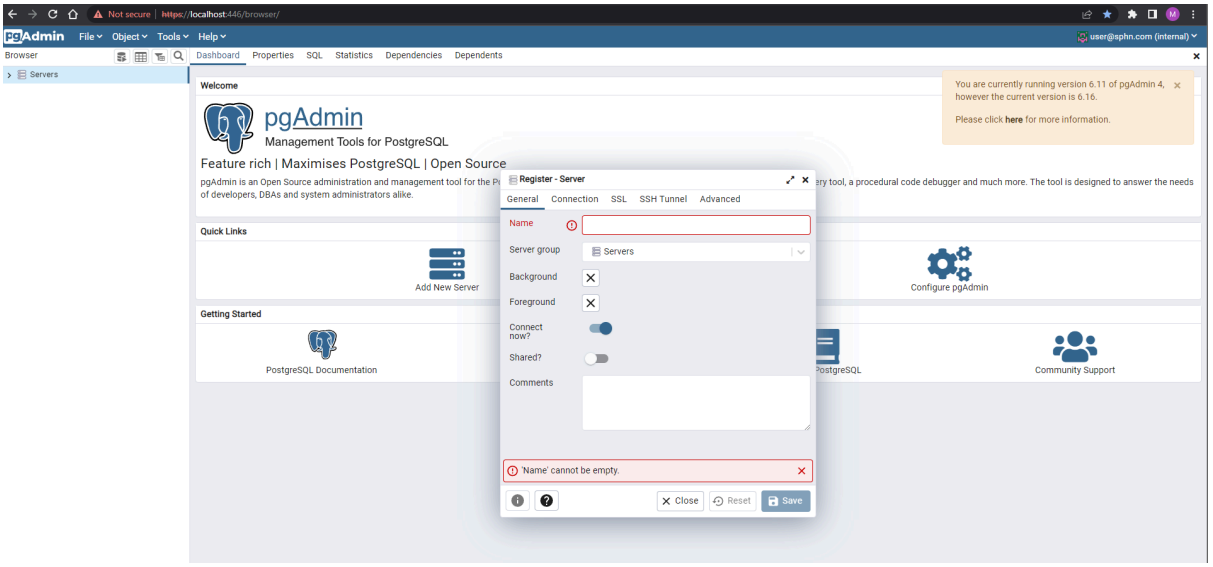
2) Welcome screen

This is the default screen you’ll see when you log into pgadmin. If this is your first login the *“Servers”* view on the left will be empty. To add the local postgres instance click on *“Add new Server”*



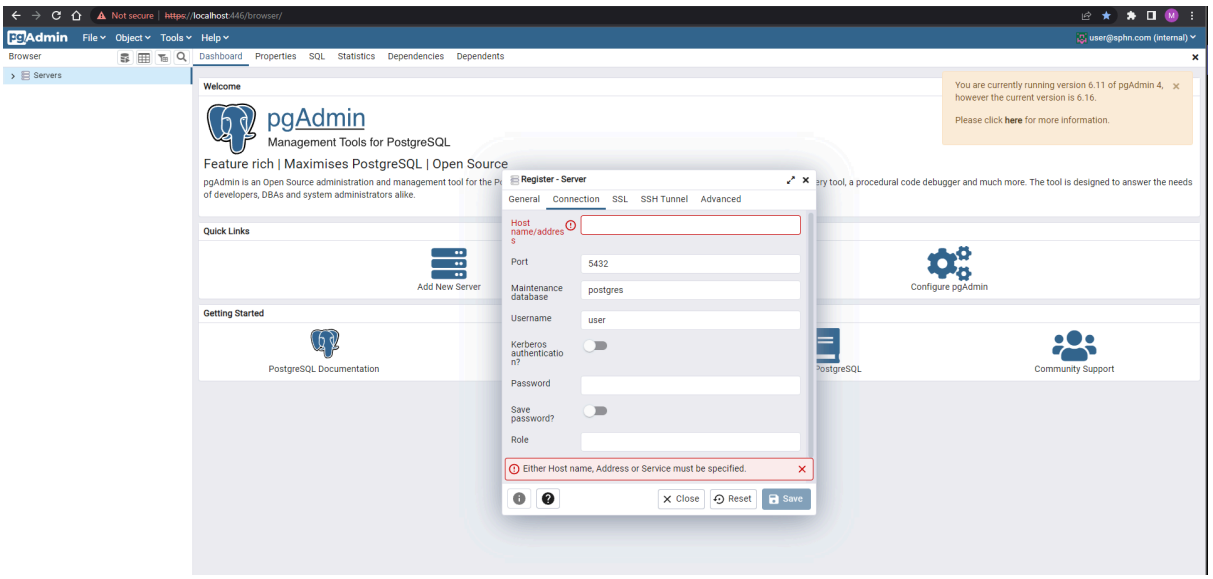
3) Add new Server

By clicking on *“Add new Server”* a new window will pop-up and ask you about information concerning the server you want to connect. In the tab “General” you need to provide a name for the server …

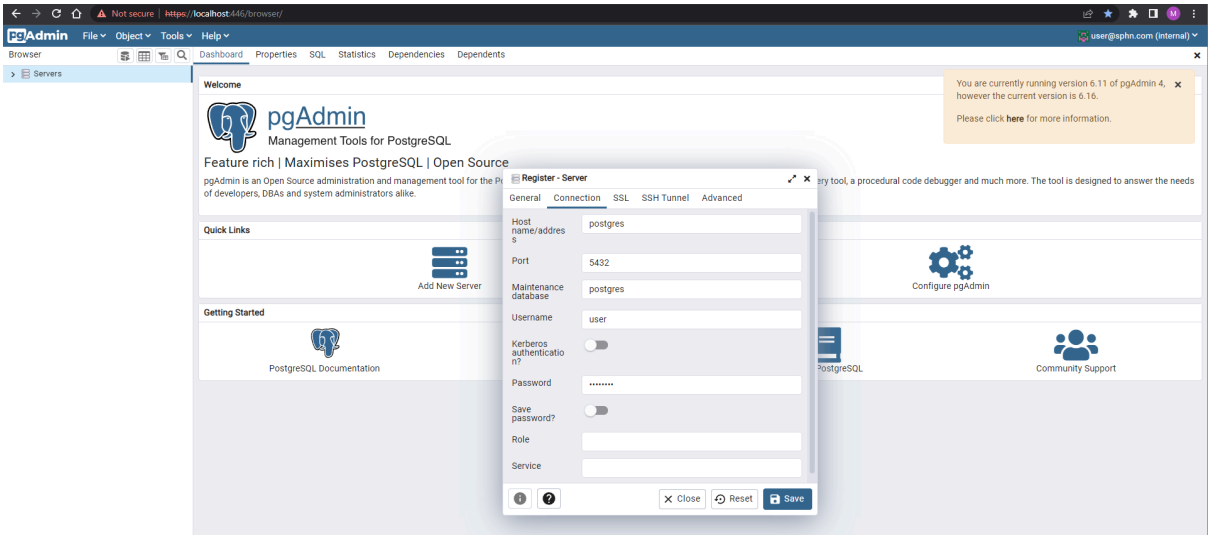


4) Enter Credentials

… and in the tab “Connection” you need to provide the host name & password.

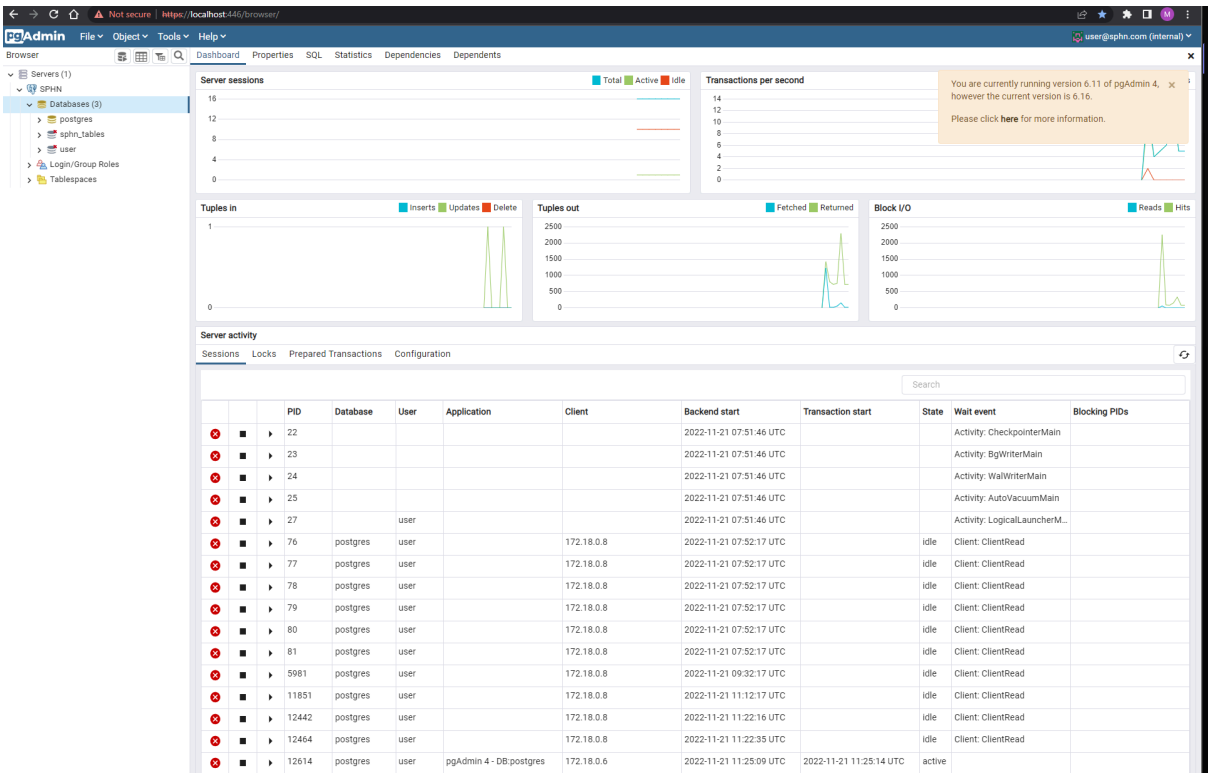


The default option here is “postgres” & “password” as password. You need to adapt it to match your configuration. **WARNING DO NOT USE** “password” as a password in production!!!



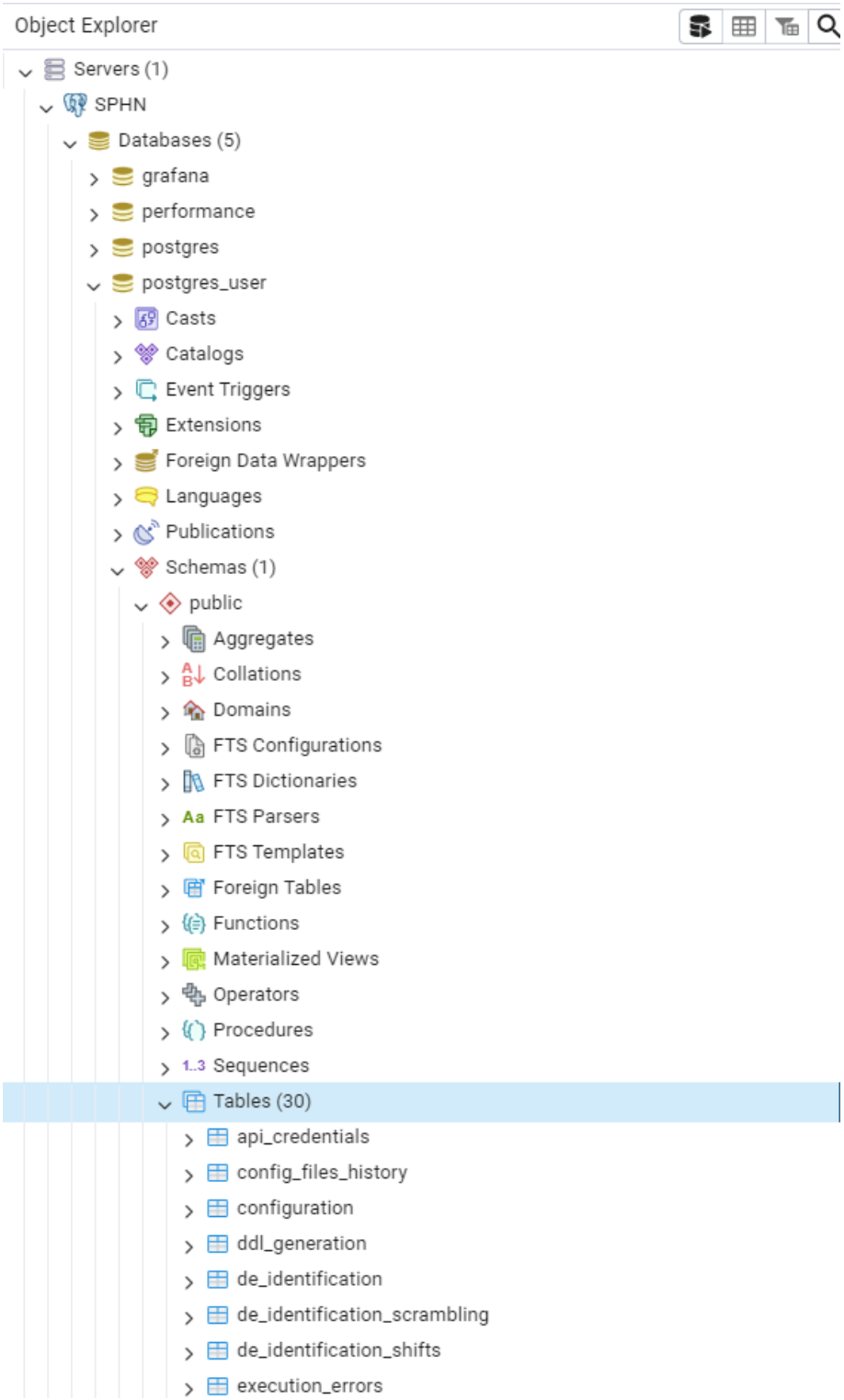
5) Access the the databases / tables of interest

After you successfully connect to your postgres instance you get access to at least three databases: *“postgres”*, *“sphn_tables”* and *“user”*.



6) Example “user” table

If you want to access a specific table e. g. in the user database you need to navigate: *Schemas>>public>>Tables* and then select the table of interest. In General the direct way to access any table is *database>>schemas>>schema-of-interest>>table-of-interest.* Keep in mind that you might encounter multiple schemas. E. g. for each initialized project a project specific schema is created.

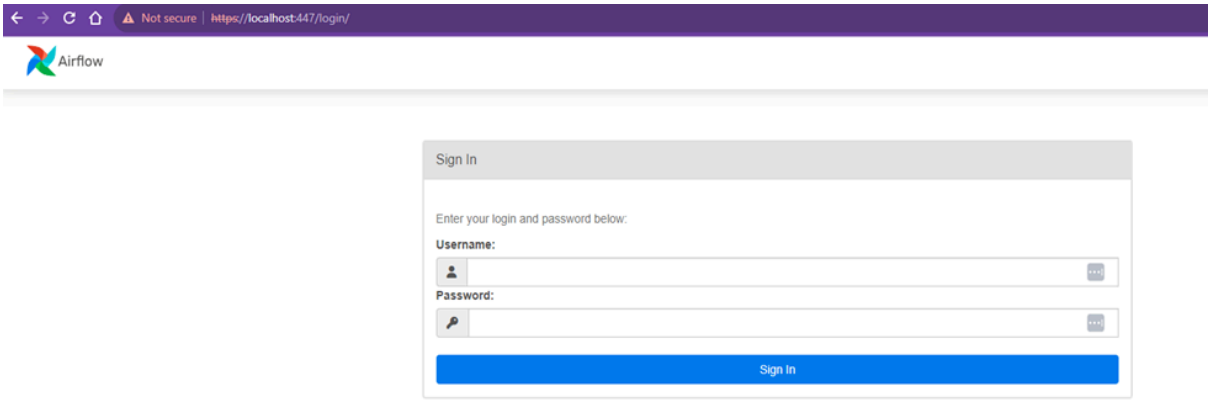

Object Explorer
Servers (1)
SPHN
Databases (5)
grafana
performance
postgres
postgres_user
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (30)
api_credentials
config_files_history
configuration
ddl_generation
de_identification
de_identification_scrambling
de_identification_shifts
execution_errors

Access Airflow

This section will discuss basic interactions and the navigation inside the airflow service. Unfortunately the pictures are very small. For more details you can either review the visual user guide for airflow in this presentation: SPHN-Connector_airflow-ui. For even more details please consult the official airflow documentation: UI / Screenshots.

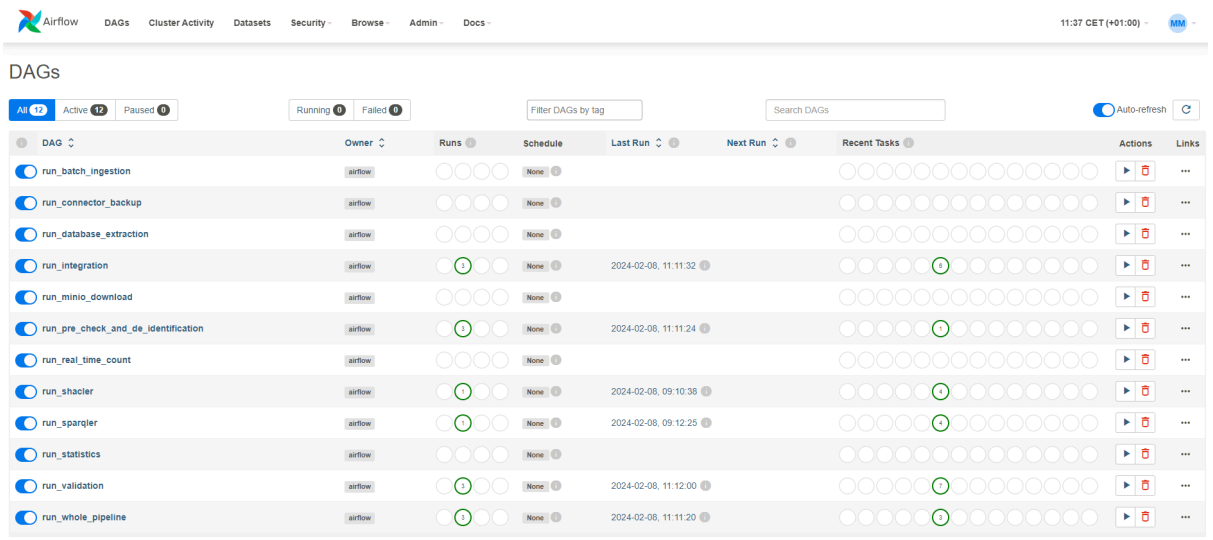
1) Navigate to your airflow instance

You find access to airflow at https://{your-server}:1443/airflow (screenshots url based on previous version). Enter your credentials to get to the pipeline overview.



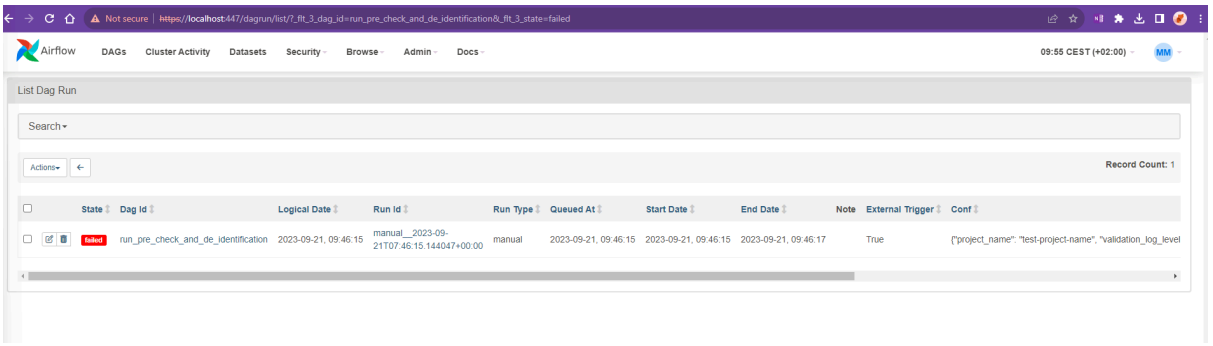
2) Select failed Pipeline

To get to the logs of the process that has been failing select *"failed runs"* (red circle in category *Runs)*. If you are uncertain which process failed and you ran the whole pipeline you can select the red circle in *run_whole_pipeline* .



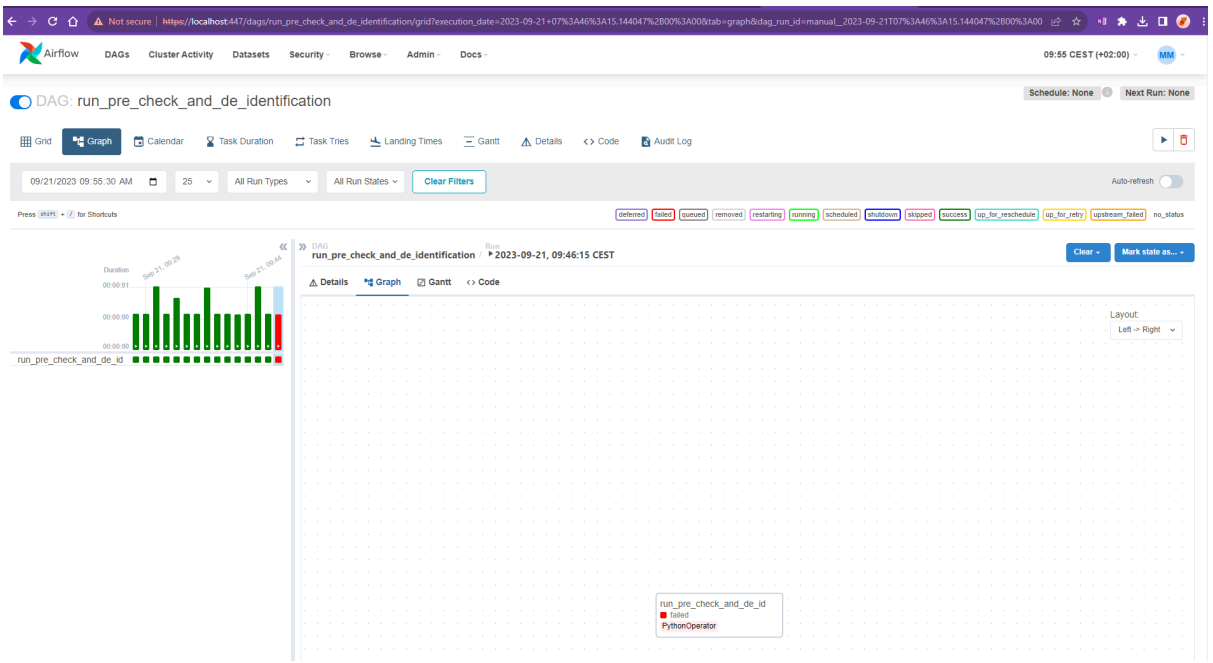
3) Select failed run

After you click on the red circle you'll get an overview of all runs that (in this case) failed. Determine which run you an interested in via the project name in the *"conf"* column and then click on the link in *"Run id"*



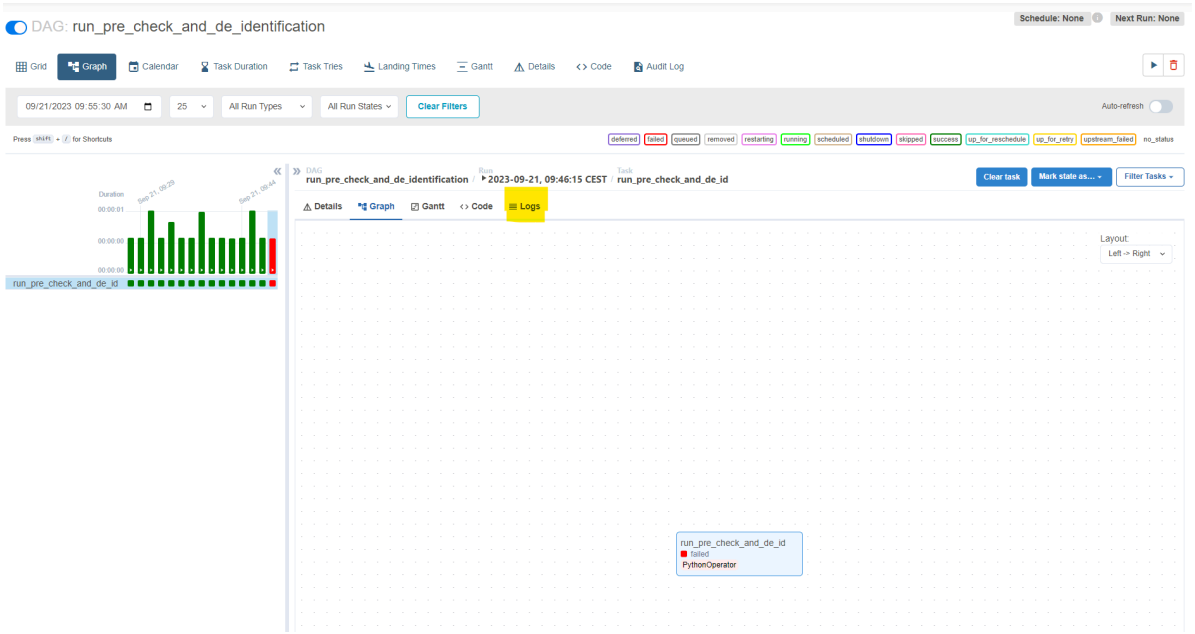
4) Select failed pipeline Step

After you click on the run you are interested in you'll get an overview of the different pipeline steps. Steps that were executed successfully are marked in green, failed steps are marked in red. Not executed steps are marked in orange. Click on the step that failed



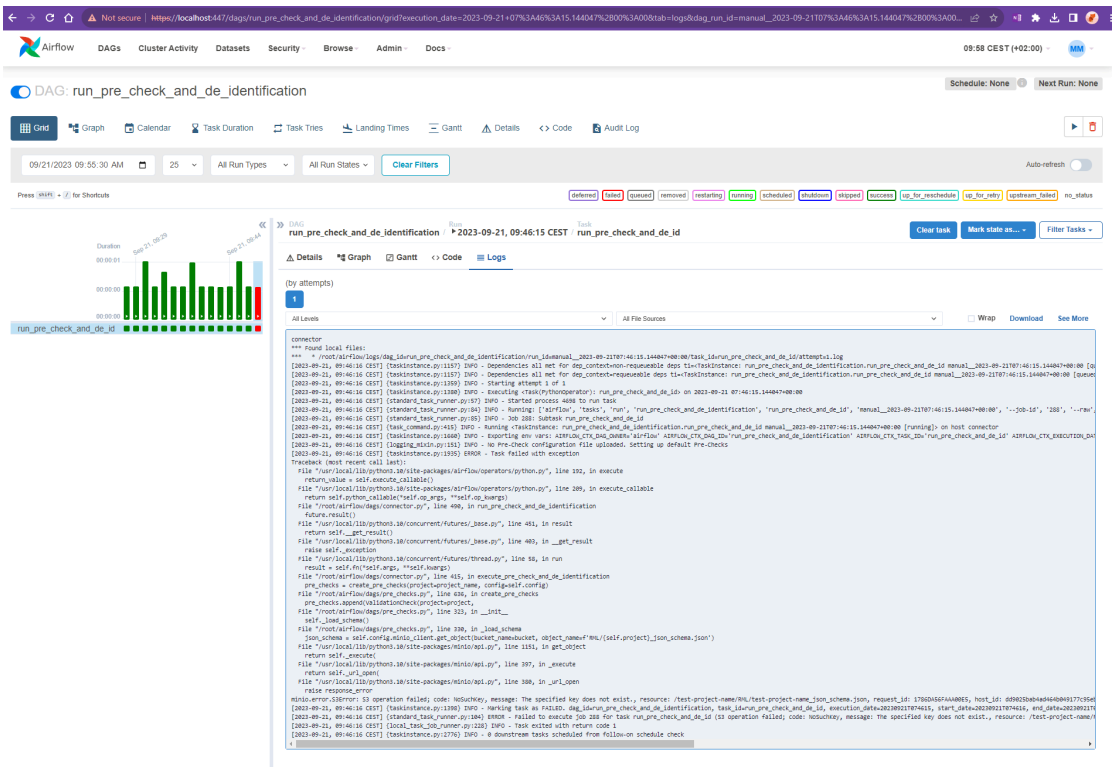
5) Navigate to Process logs

After you click on the failed step and new options will be available. Click on “Log” at the top to go to the log file. [Marked in yellow box].



6) Read Airflow log

Afterwards the logs will be displayed to you



7) Airflow DAGs overview

There are several active DAGs in our Airflow environment. The aim of this section is to give an overview of the available DAGs, their purposes, and when they are invoked.

- **run_batch_ingestion**: executed when endpoint `/Trigger_batch_ingestion` is triggered. It uploads the data stored inside Input/batch_ingestion into the landing zone.
- **run_connector_backup**: executed when endpoint `/Backup_connector` is triggered. It builds the backup and stores it into MinIO bucket sphn-connector-backup.
- **run_database_extraction**: executed when the endpoint `/Ingest_from_database` is triggered. It is used to extract the patient data from the database into JSON files.
- **run_integration**: executed for the Integration step in the pipeline (triggered via `/Start_pipeline` when step set to `Integration`). Responsible for conversion of JSON data to RDF data.
- **run_minio_download**: executed when endpoint `/Download_in_minio` with action type 'Start' is triggered. It builds the download bundle and stores it in MinIO project's bucket under prefix /Download.
- **run_pre_check_and_de_identification**: executed for the Pre-Check/De-Identification step in the pipeline (triggered via `/Start_pipeline` when step set to `Pre-Check/De-Id`). Responsible for running pre-checks and de-identification.
- **run_real_time_count**: executed as a consequence of `/Get_status` and used to count real-time value of processed patients.
- **run_shacler**: triggered at project creation (`/Create_project`) and used to auto-generate the SHACL (via the Shacler) file based on the provided schema and exception files.

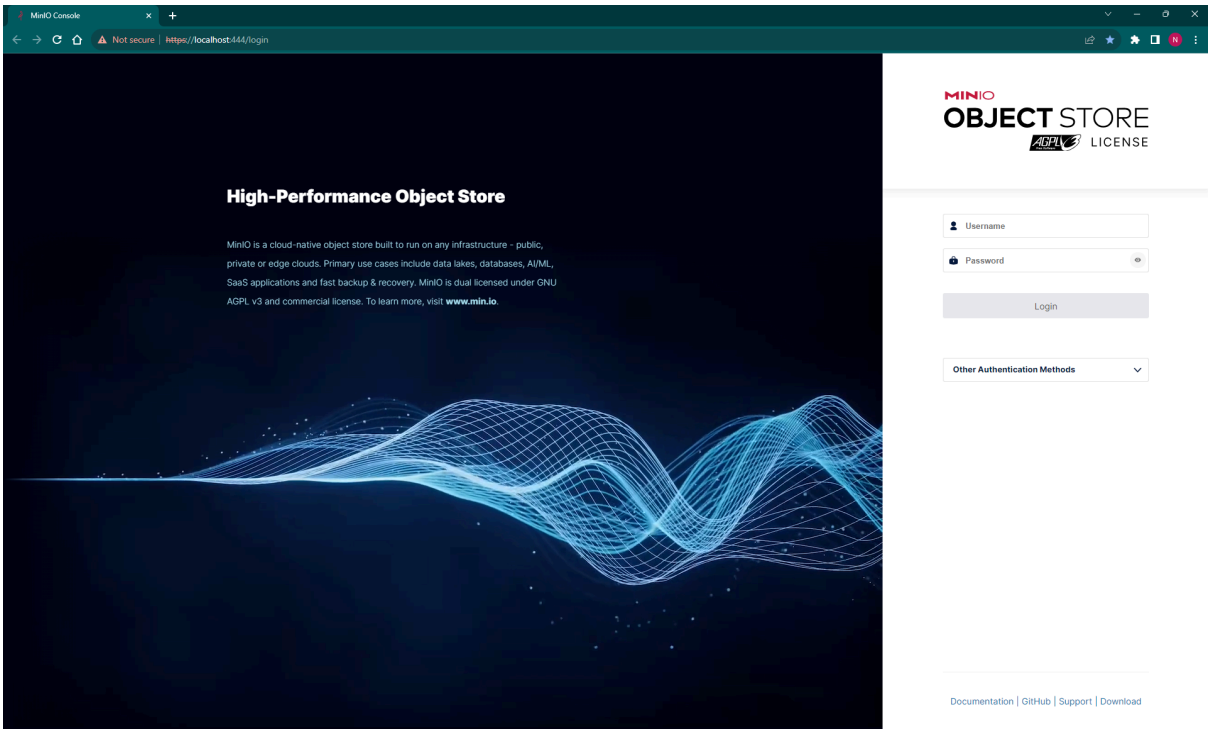
- **run_sparqler**: triggered after project creation. It generates (via the SPARQLer) the SPARQL queries used during the `Statistics` step.
- **run_statistics**: triggered by the endpoint `/Start_statistics` and responsible for running patients' statistics.
- **run_validation**: executed for the Validation step in the pipeline (triggered via `/Start_pipeline` when step set to `Validation`). Responsible for validating patient data (via QAF).
- **run_whole_pipeline**: triggered when endpoint `/Start_pipeline` is executed. It will trigger in sequence run_pre_check_and_de_identification, run_integration, and run_validation.

Access Minio

This section will discuss basic interactions and the navigation inside the mino service. Unfortunately the pictures are very small. For more details you can either review the visual user guide for Minio in this presentation: SPHN-Connector_minio. For even more details please consult the official Minio documentation: minio_docker.

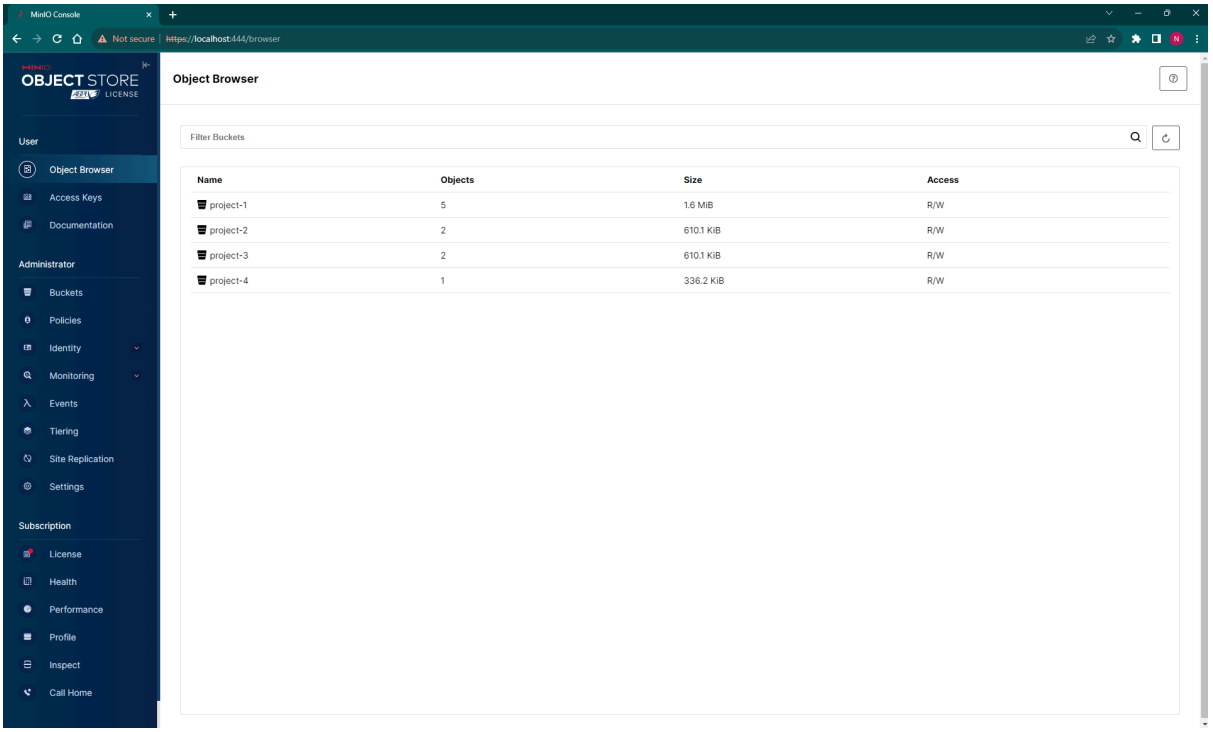
1) Navigate to minio

You find access to mino at https://{your-server}:1443/minio/ (screenshots url from previous versions). Enter your credentials to get an overview of all the buckets that are available.



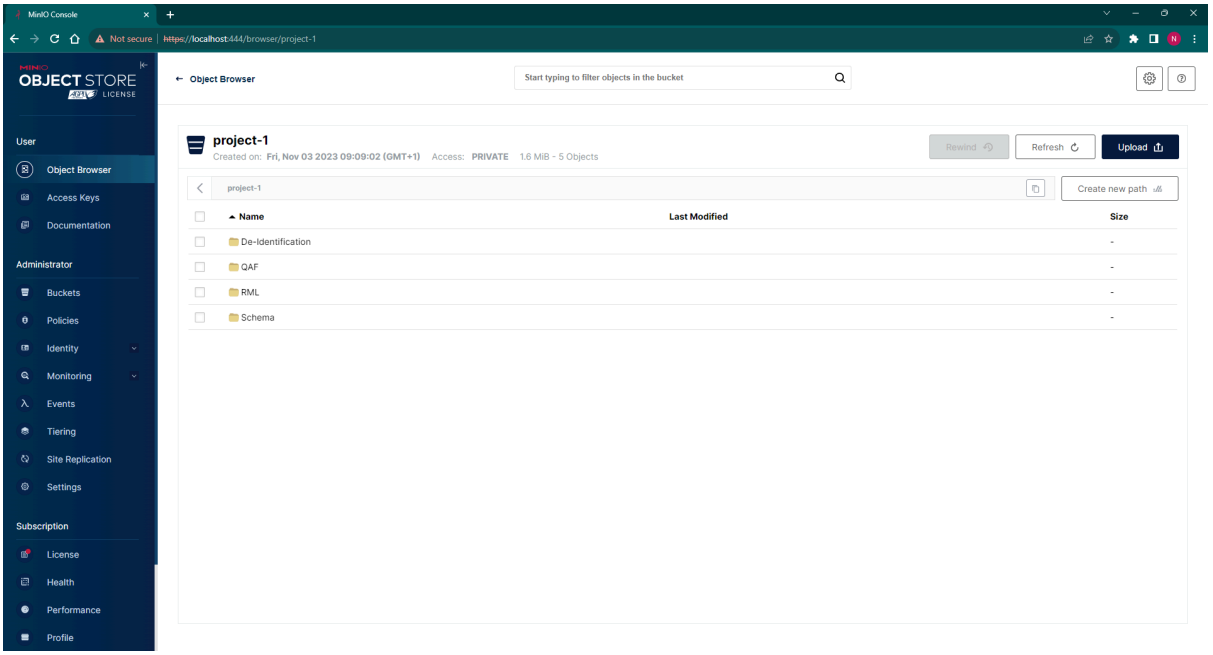
2) Overview of all Buckets

After you log in you get an overview of all available buckets / projects in your Minio instance.

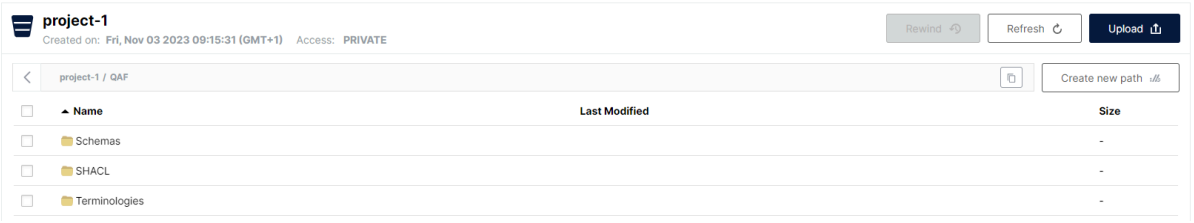


3) Deep dive into project

Click on “Browse” on the bucket / project you are interested in to get an overview of the files / folders available. Inside the Input folder you’ll find the files you uploaded depending on their status / file format in landing-, refined- or graph zone. in the Schema folder you’ll find the project specific schema. The QAF folder holds all files needed for the validation with the Quality assurance framework. The RML folder contains the rml-mapping file and the json schema for the project.

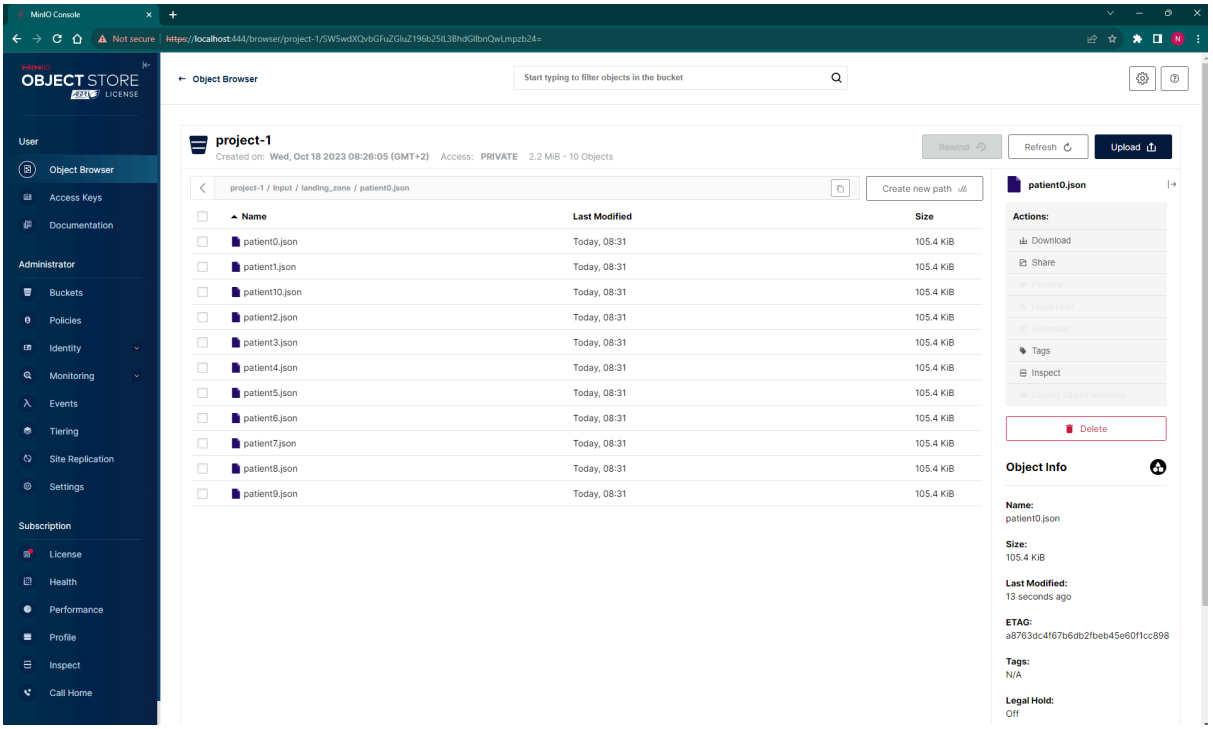


4) Example QAF folder



5) Download file

If you want to take a closer look at a file you can download it. Select the file and afterwards pick Download in the Actions box on the right size.

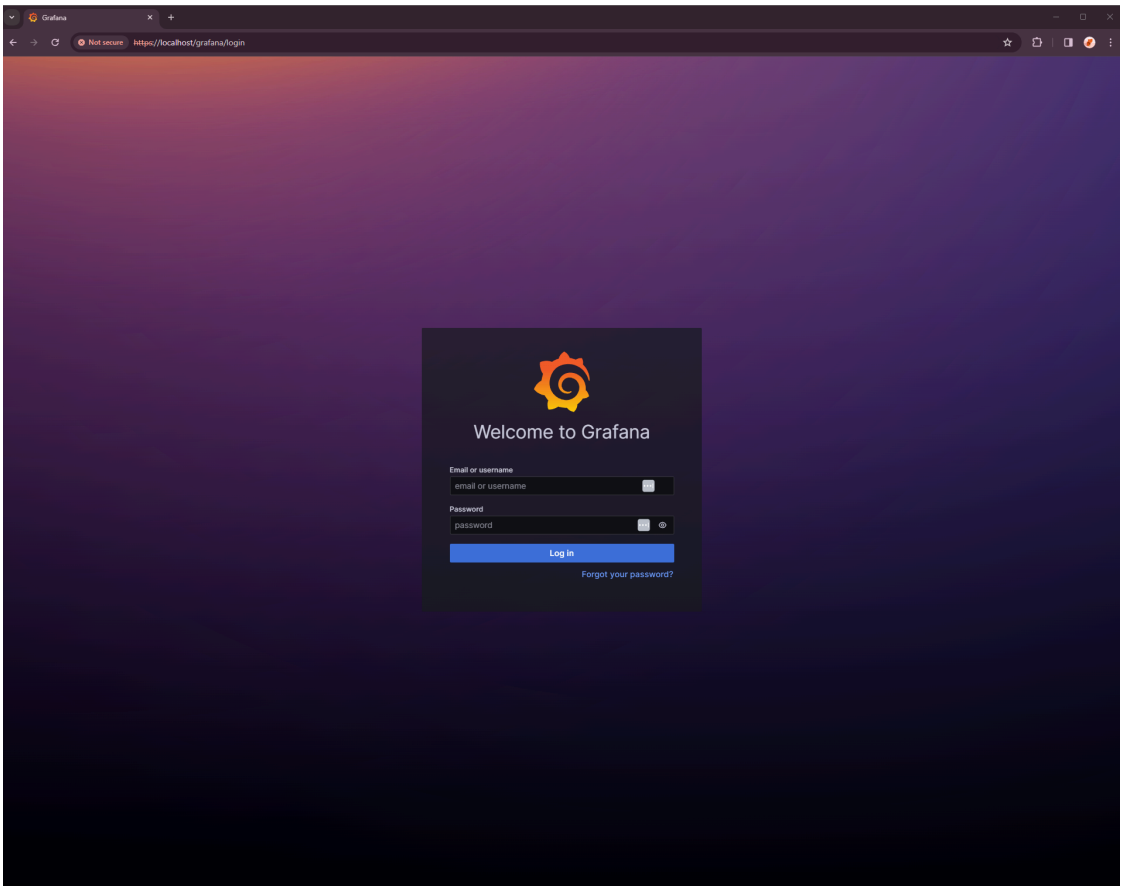

MinIO Console
Not secure
https://localhost:444/browser/project-1/SW5wdXQvbGFuZGluZ196b25fL38hdGlibnQwLmpzb24=
OBJECT STORE
LICENSE
User
Object Browser
Access Keys
Documentation
Administrator
Buckets
Policies
Identity
Monitoring
Events
Tiering
Site Replication
Settings
Subscription
License
Health
Performance
Profile
Inspect
Call Home
Object Browser
Start typing to filter objects in the bucket
project-1
Created on: Wed, Oct 18 2023 08:26:05 (GMT+2) Access: PRIVATE 2.2 MiB - 10 Objects
Rewind
Refresh
Upload
project-1 / input / landing_zone / patient0.json
Create new path
Name
Last Modified
Size
patient0.json Today, 08:31 105.4 KiB
patient1.json Today, 08:31 105.4 KiB
patient10.json Today, 08:31 105.4 KiB
patient2.json Today, 08:31 105.4 KiB
patient3.json Today, 08:31 105.4 KiB
patient4.json Today, 08:31 105.4 KiB
patient5.json Today, 08:31 105.4 KiB
patient6.json Today, 08:31 105.4 KiB
patient7.json Today, 08:31 105.4 KiB
patient8.json Today, 08:31 105.4 KiB
patient9.json Today, 08:31 105.4 KiB
patient0.json
Actions:
Download
Share
Tags
Inspect
Delete
Object Info
Name:
patient0.json
Size:
105.4 KiB
Last Modified:
13 seconds ago
ETAG:
a8763dc4f67b6db2fbeb45e60f1cc898
Tags:
N/A
Legal Hold:
Off

Access Grafana

This section will discuss basic interactions and the navigation inside the grafana service. Unfortunately the pictures are very small. For more details you can review the visual user guide for grafana in this presentation: SPHN-Connector_grafana. For even more details please consult the official grafana documentation: grafana. Please note there are different versions of Grafana available (Enterprise, Open source, Grafana cloud). For the SPHN Connector the Grafana open source version (grafana-oss) is used.

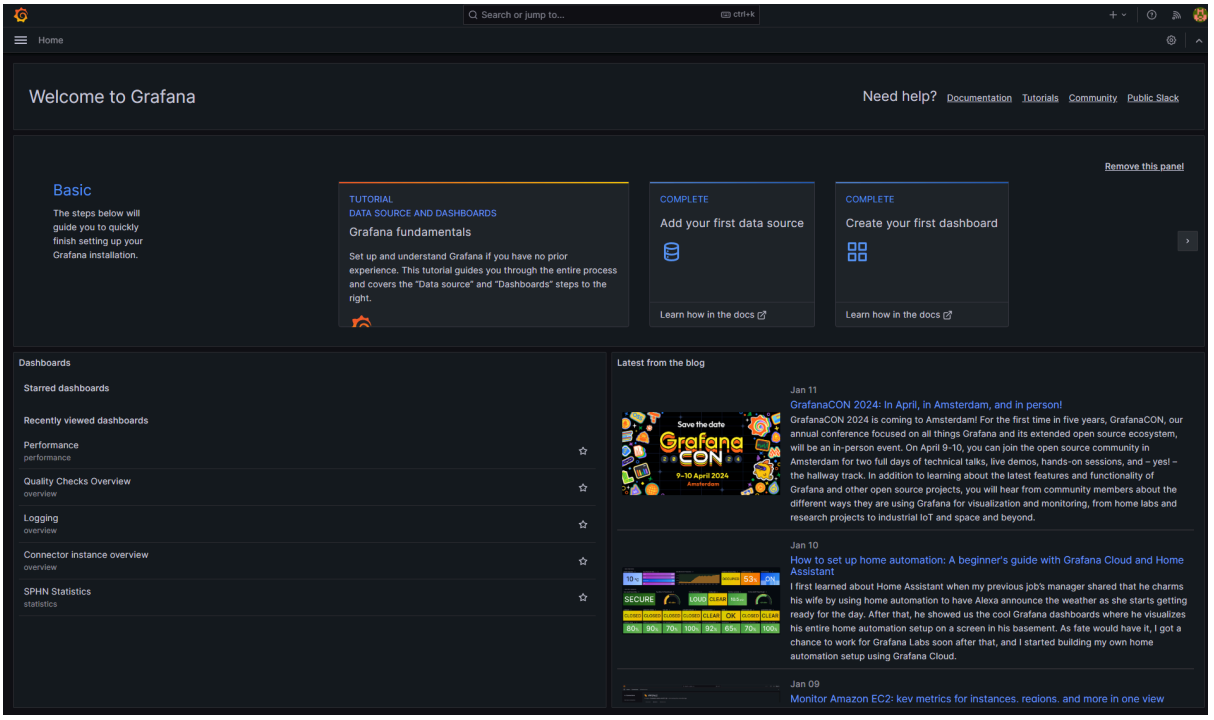
1) Navigate to grafana

You find access to mino at https://{your-server}:1443/grafana. Enter your credentials to get an overview of all the buckets that are available.



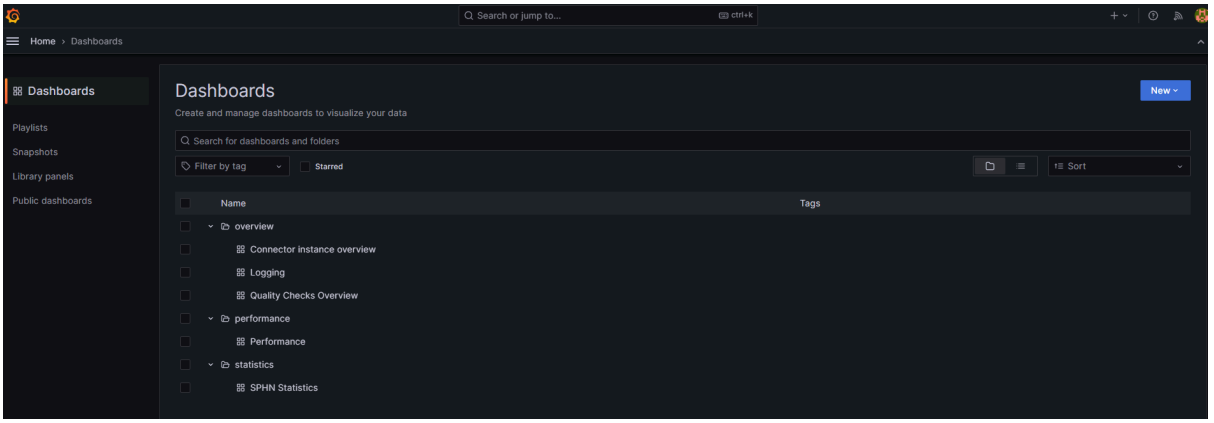
2) Welcome screen

Once you successfully logged in you see the Grafana home screen. In case you Connector instance is not connected to the Internet (which is the recommended way) some links presented in the home screen will not work (e. g. Latest from the blog). This will not limit the functionality of grafana. it is just something to keep in mind



3) Dashboards overview

Clicking on the burger icon in the top left corner (under the Grafana logo) will open the navigation panel. If you select Dashboards you will get an overview of all available dashboards in your grafana instance. Per default three folders with 5 Dashboards are shipped per default:

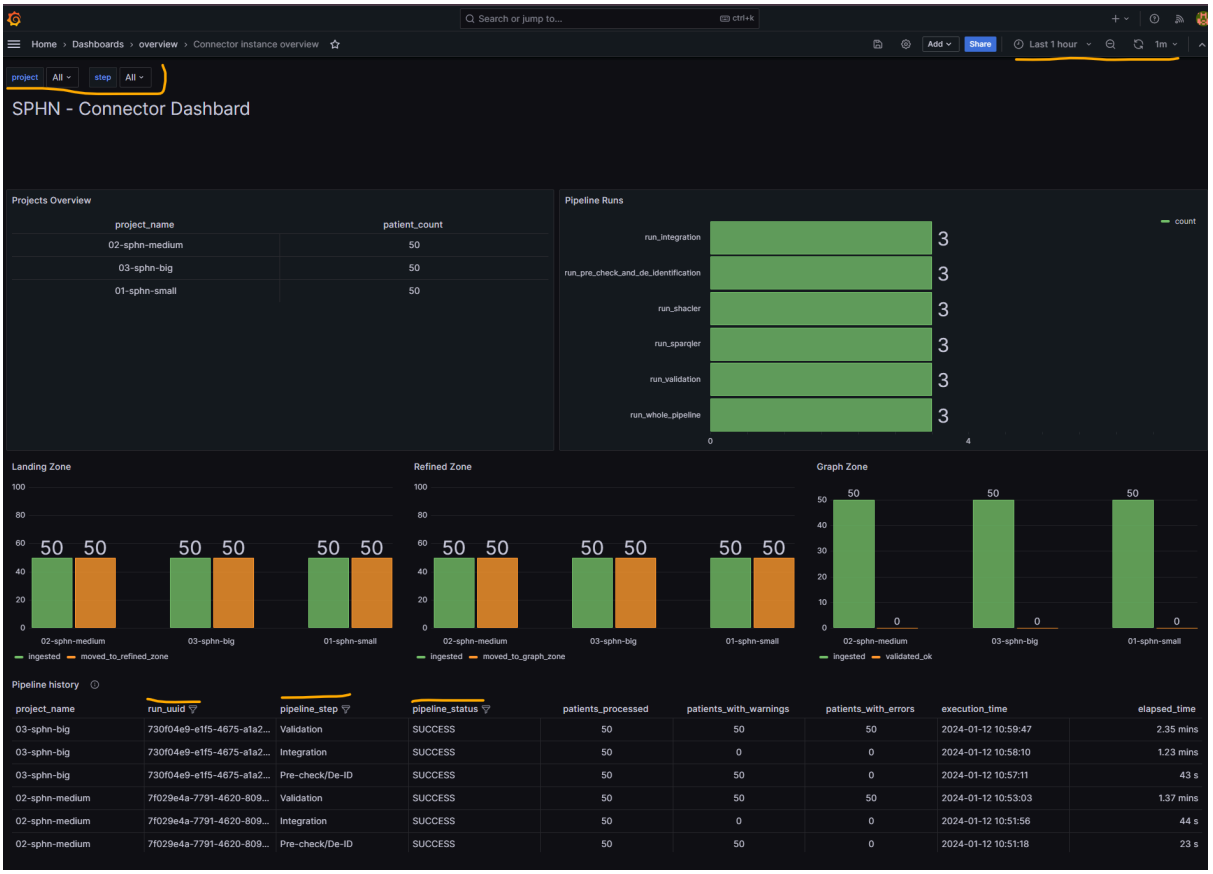


Folder	Dashboard name	short description
overview	Connector Instance Overview	high level overview about the Connector (created projects, ingested patients, status

		of the patient files).
overview	Logging	Access to the logs created by the SPHN Connector
overview	Quality Checks Overview	Overview of the results of the Quality checks
performance	Performance	Visualizes the results of the performance test
statistics	SPHN Statistics	Access to the results of the standard startics

4) Interact with a dashboard

Once you select the Dashboard of interest you can interact with it. E. g use the predefined filters or filter table visuals directly. For time series visuals you can furthermore limit the displayed range of the data. For a better view please have a look at the visual user guides: [SPHN-Connector_grafana]



Release Plan

Semantic Versioning

`<Major>.<Workpackage-Release>.<Bugfix-Version>`

For example, the first release will be 0.1.0. Then all bug fixes for WP1 will increase the last digit, until WP2 (0.2.0) is released, then typically no further bugfix releases for WP1 branch.

Planning

In last Sprint of the corresponding WP-X → make release-candidate branch WP-X
In the release-candidate branch only planned features will be merged (e.g. bug fixing).
If WP-X+1 is already under development, bug fixes will be picked to WP-X bugfix release-candidates.
The Release-candidates will be continuous.

A release-candidate will be released when it is tested-ok by at least USZ.

Sprint	**Y-2**	**Y-1**	**Y**
Description	**Release Candidate assembly**	**Test at USZ** / potential bug fixing	**Release of Version** if no blocking bugs, otherwise delay release.

There will be at most 2 release-candidates around in on Sprint: a planned Workpackage-Release and a Bugfix-Version for the currently released Workpackage-Release under Test or Release.

Issue Severities

Severity Level	**Description**	**Consequences**
Low	A minor bug is existing, but main functionality is not impacted (e.g. wrong display of a number)	Decide on further actions (e.g. plan for bugfix release or next WP-Release).
Normal	A bug is existing, but does not block the main functionality (e.g. workaround is possible or exists)	Try fix/workaround in the Test Spring (Y-1). Decide on further actions.
Blocking	Important functionality of the SPHN Connector is not given.	The release is blocked until the issue is resolved.

Edge cases for 2023.2 RDF schema

Introduction

In this section, we summarize concepts and logics of the SPHN Connector that are relevant up to RDF schema 2023.2. If the user needs to produce data according to 2023.2 RDF schema, they can read here to get some useful insights.

AdministrativeCase

The AdministrativeCase concept is used very often inside nested fields of the JSON data. For that reason, we decided to reference it directly and not report it entirely everytime. That means that inside a core concept, maybe a few levels deep into the nesting, we have the property `sphn:hasAdministrativeCase` there is no need to define the entire `AdministrativeCase` schema structure again, but the id to the previously defined object will suffice. Administrative cases are defined at the corresponding key in the `content` properties map.

JSON schema

The sphn:AdministrativeCase property is defined as a list of administrative cases at the top level of the JSON schema:

```
"sphn:AdministrativeCase": {
    "type": "array",
    "description": "List of 'sphn:AdministrativeCase' concepts",
    "items": {
            "type": "object",
            "description": "SPHN Concept 'AdministrativeCase'",
            "properties": {
                "id": {
                    "type": "string",
                    "description": "ID of SPHN Concept 'AdministrativeCase'"
                },
                ...
            },
            "required": [
                "id",
                "sphn:hasIdentifier",
                "sphn:hasAdmissionDateTime"
            ],
            "additionalProperties": false
        }
}
```

All the administrative cases used for the patient can be defined there with a unique identifier `id`, and then referenced back via the object property `sphn:hasAdministrativeCase`:

```
"sphn:hasAdministrativeCase": {
    "type": "object",
```

```
    "description": "SPHN Concept 'AdministrativeCase'",
    "properties": {
        "id": {
            "type": "string",
            "description": "ID of SPHN Concept 'AdministrativeCase'"
        }
    },
    "required": [
        "id"
    ],
    "additionalProperties": false
}
```

RML mapping

The RML mapping of the administrative cases is constructed as all the other SPHN objects. The difference is in the join condition used to link the correct administrative case to its property.

The join condition maps the administrative case defined by the `id` in the object property `sphn:hasAministrativeCase` to the corresponding object in the cases list:

```
[ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-AdministrativeCase-{sphn:hasAdmi
nistrativeCase.id}" ] ;
            rr:predicate sphn:hasAdministrativeCase ]
```

Database schema

The table schema for the `sphn:AdministrativeCase` concept is constructed in the same way as the other tables. The peculiarity of this case is found when defining the `sphn:hasAdministrativeCase` property. The behavior is the same as the one for the JSON schema, meaning that we won't report the entire definition of the concept, but we just need to define the referencing id which is defined by the column `sphn_hasAdministrativeCase`.

UCUM codes

The `Unit` concepts code is defined solely by UCUM codes.That means that the codes IRI should be directly referenced in the RDF output. This can be considered as a special case for the Terminology class and it behaves similarly as the value sets.

JSON schema

The JSON schema for the property `sphn:hasCode` under `sphn:hasUnit` is the following:

```
"sphn:hasUnit": {
    "type": "object",
    "description": "SPHN Concept 'Unit'",
    "properties": {
        "id": {
            "type": "string",
            "description": "ID of SPHN Concept 'Unit'"
        },
```

```
        "sphn:hasCode": {
            "type": "object",
            "description": "Schema for property 'hasCode'",
            "properties": {
                "iri": {
                    "type": "string",
                    "description": "UCUM IRI for 'hasCode' property"
                }
            },
            "required": [
                "iri"
            ],
            "additionalProperties": false
        }
    },
    "required": [
        "id",
        "sphn:hasCode"
    ],
    "additionalProperties": false
}
```

RML mapping

The RML mapping directly reference the IRI provided in the JSON data:

```
:hasUnitMapping a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$..sphn:hasUnit" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient-data-input.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rml:reference "sphn:hasCode.iri" ;
                    rr:termType rr:IRI ] ;
            rr:predicate sphn:hasCode ] ;
    rr:subjectMap [ rr:class sphn:Unit ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-Unit-{id}" ] .
```

Database schema

Nothing special here, the property is mapped as other properties by considering the nesting of the JSON schema. For example:
`sphn_hasSubjectAge__sphn_hasQuantity__sphn_hasUnit__sphn_hasCode__iri`

DataProviderInstitute

In the SPHN Connector each patient file must be related to the same DataProviderInstitute. Based on this assumption, we moved the `sphn:DataProviderInstitute` schema to the top level of the JSON schema and adapted the RML mapping accordingly.

JSON schema

This object is defined on the top level of the JSON schema. In that way the user can define the data provider once for the entire file. There are no other references in the JSON schema to that field, meaning that all the object properties `sphn:hasDataProviderInstitute` have been removed from the corresponding objects.

```
  "sphn:DataProviderInstitute": {
            "type": "object",
            "description": "SPHN Concept 'DataProviderInstitute'",
            "properties": {
                "id": {
                    "type": "string",
                    "description": "ID of SPHN Concept 'DataProviderInstitute'"
                },
                "sphn:hasCode": {
                    "type": "object",
                    "description": "SPHN Concept 'Code'",
                    "properties": {
                        "id": {
                            "type": "string",
                            "description": "ID of SPHN Concept 'Code'"
                        },
                        "sphn:hasIdentifier": {
                            "description": "Value for 'hasIdentifier' property",
                            "type": "string"
                        },
                        "sphn:hasCodingSystemAndVersion": {
                            "description": "Value for 'hasCodingSystemAndVersion'
property",
                            "type": "string"
                        },
                        "sphn:hasName": {
                            "description": "Value for 'hasName' property",
                            "type": "string"
                        }
                    },
                    "required": [
                        "id",
                        "sphn:hasIdentifier",
                        "sphn:hasCodingSystemAndVersion"
                    ],
                    "additionalProperties": false
                }
            },
            "required": [
                "id",
                "sphn:hasCode"
            ],
            "additionalProperties": false
        }
```

RML mapping

The correlation between the object properties depending on the data provider institute and the institute defined at the top level of the JSON is handled directly in the mapping.

There is a single TriplesMap for that object that iterates over the JSON path "$.sphn:DataProviderInstitute":

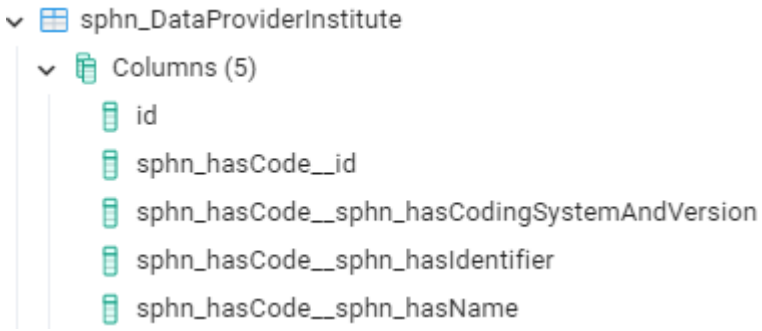
```
:DataProviderInstituteMapping a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.sphn:DataProviderInstitute" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "allergy_input.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-Code-{sphn:hasCode.id}" ] ;
            rr:predicate sphn:hasCode ] ;
    rr:subjectMap [ rr:class sphn:DataProviderInstitute ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-DataProviderInstitute-{id}" ] .
```

Other concepts can define this concepts by defining a join condition defined in the properties `sphn:hasDataProviderInstitute` with the map:

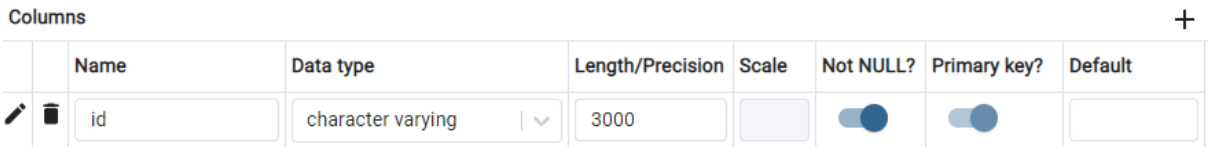
```
[ rr:objectMap [ rr:parentTriplesMap :DataProviderInstituteMapping ] ;
            rr:predicate sphn:hasDataProviderInstitute ]
```

Database schema

Concept sphn:DataProviderInstitute has its own table in the database schema. In principle, from the fact that a project should be related to a single data provider, that table has only a single record defining the provider.



This table has primary key defined by the `id` field



SubjectPseudoIdentifier

In the SPHN Connector each patient file must be related to the same SubjectPseudoIdentifier. Based on this assumption, we moved the `sphn:SubjectPseudoIdentifier` schema to the top level of the JSON schema and adapted the RML mapping accordingly.

JSON schema

This object is defined on the top level of the JSON schema. In that way the user can define the patient information once for the entire file. There are no other references in the JSON schema to that

field, meaning that all the object properties `sphn:hasSubjectPseudoIdentifier` have been removed from the corresponding objects.

```
"sphn:SubjectPseudoIdentifier": {
    "type": "object",
    "description": "SPHN Concept 'SubjectPseudoIdentifier'",
    "properties": {
        "id": {
            "type": "string",
            "description": "ID of SPHN Concept 'SubjectPseudoIdentifier'"
        },
        "sphn:hasIdentifier": {
            "description": "Value for 'hasIdentifier' property",
            "type": "string"
        }
    },
    "required": [
        "id",
        "sphn:hasIdentifier"
    ],
    "additionalProperties": false
}
```

RML mapping

The correlation between the object properties depending on the subject pseudo identifier and the identifier defined at the top level of the JSON is handled directly in the mapping.

There is a single TriplesMap for that object that iterates over the JSON path `"$.sphn:SubjectPseudoIdentifier"`:

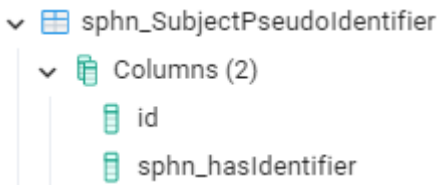
```
:SubjectPseudoIdentifierMapping a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.sphn:SubjectPseudoIdentifier" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient-data-input.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rml:reference "sphn:hasIdentifier" ;
                    rr:datatype xsd:string ] ;
            rr:predicate sphn:hasIdentifier ],
        [ rr:objectMap [ rr:parentTriplesMap :DataProviderInstituteMapping ] ;
            rr:predicate sphn:hasDataProviderInstitute ] ;
    rr:subjectMap [ rr:class sphn:SubjectPseudoIdentifier ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-SubjectPseudoIdentifier-{id}" ]
.
```

Other concepts can define this concepts by defining a join condition defined in the properties `sphn:hasSubjectPseudoIdentifier` with the map:

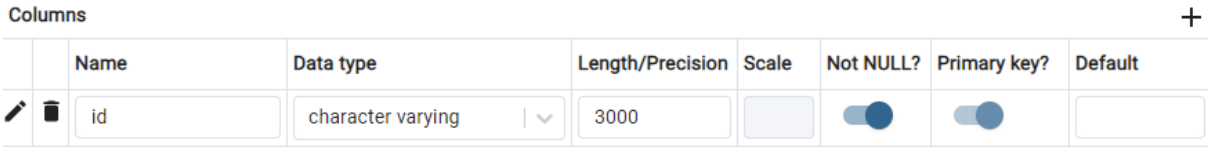
```
[ rr:objectMap [ rr:parentTriplesMap :SubjectPseudoIdentifierMapping ] ;
            rr:predicate sphn:hasSubjectPseudoIdentifier ]
```

Database schema

Reflecting the JSON schema, the resulting table for this concept is very simple



The table has primary key defined by the `id` field



DataRelease

This is an additional concept which should be added to all RDF files after JSON to RDF conversion. It gives information about the extraction date of the data and the version of the RDF schema.

JSON schema

In the JSON schema, this quantity is described by the `sphn:DataRelease` property at the top level:

```
"sphn:DataRelease": {
    "type": "object",
    "description": "sphn:DataRelease properties",
    "properties": {
        "id": {
            "type": "string",
            "description": "ID of 'sphn:DataRelease'. To ensure the uniqueness of a
DataRelease instance ID (i.e. the dataset identifier), a UNIX Epoch timestamp should
ideally be concatenated to it as a suffix"
        },
        "creationTime": {
            "type": "string",
            "description": "Value for the sphn:hasExtractionDateTime of the data
release",
            "format": "date-time"
        }
    },
    "required": [
        "id",
        "creationTime"
    ]
}
```

The uniqueness of the DataRelease object is given by its `id`. Usually, its value should represent the UNIX timestamp of the `creationTime` date-time.

RML mapping

The RDF schema version is extracted internally from the schema and reported to the mapping. The extraction date and extraction timestamp are read from the JSON data.

```
:DataRelease a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator "$.sphn:DataRelease" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "allergy_input.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-ontology/sphn/2023/2" ] ;
            rr:predicate dct:conformsTo ],
        [ rr:objectMap [ rml:reference "creationTime" ;
                    rr:datatype xsd:dateTime ] ;
            rr:predicate sphn:hasExtractionDateTime ],
        [ rr:objectMap [ rr:parentTriplesMap :DataProviderInstituteMapping ] ;
            rr:predicate sphn:hasDataProviderInstitute ] ;
    rr:subjectMap [ rr:class sphn:DataRelease ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-DataRelease-{id}" ] .
```

Database schema

The table `sphn_DataRelease` is also constructed on the JSON schema. It has only non-nullable columns and the primary key defined by `sphn_hasSubjectPseudoIdentifier` and `sphn_hasDataProviderInstitute`, such that only one patient for a specific data provider can be defined in the table.

Columns +

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
	creationTime	timestamp with time zone			●	○	
	id	character varying	3000		●	○	
	sphn_hasDataProviderInstitute	character varying	3000		●	●	
	sphn_hasSubjectPseudoIdentifie	character varying	3000		●	●	

This table is used by the user once all the data for a specific patient has been uploaded. Therefore, the user marks the patient as released and the data will be pulled at the next triggered ingestion into the SPHN Connector.

Multi-classes ranges

To be compliant with the latest version of the schema (2023-2) we introduce a new property called *target_concept*.

The version 2023-1 has introduced a new pattern. Now properties that are not related to a Code can be one of multiple things. We see that for example in the *'Drug Prescription'* core concept. This core concept has a property called *'has Indication to Start'* which links directly to the concept *'Intent'* and *'Diagnosis'. 'Diagnosis'* on the other hand has multiple children*: 'FOPH Diagnosis', 'ICD-O DIagnosis'*

and '*Nursing Diagnosis'*. So the *'has Indication to Start'* can link to *'Intent', 'Diagnosis', 'FOPH Diagnosis', 'ICD-O Diagnosis'* and '*Nursing Diagnosis'*.
This is not only limited to the *'Drug Prescription'* we have a similar situation for the '*Variant Descriptor'* core concept and its property '*has Genetic Variation'* as well as for the 'Inhaled *Oxygen Concentration'* core concept and its '*has Oxygen Equipment'* property. For the documentation we will limit our explanation to the relationship between the property '*has Indication to Start'* and the concept '*Diagnosis'* .

JSON schema

This one to many relationship is resolved in JSON with a "oneOf" so the "hasIndicationToStart" can be one of "Diagnosis", "Intent", "FOPHDiagnosis", "ICDODiagnosis" or "NursingDiagnosis" for example. This works for the JSON Schema responsible well unfortunately if you instance the data problems arise. Without a proper identifier we can not be certain which concept is referred to / is instantiated. To distinguish between those concepts we propose a new property that is added to these special cases. The "target_concept" property. The target concept property is defined as an enum and it only holds one value: The IRI of the concept it is referring to. So for example for the Diagnosis concept inside of the hasIndicationToStart is defined in the JSON like this:

```
{
   "target_concept": {
     "type": "string",
     "description": "IRI for Concept 'Diagnosis'",
     "enum": [
       "https://biomedit.ch/rdf/sphn-ontology/sphn#Diagnosis"
     ]
   }
}
```

If we combine these information the JSON schema for the hasIndicationToStart will look like this:

```
 "sphn:hasIndicationToStart": {
  "type": "object",
  "description": "SPHN Concept 'Diagnosis'/SPHN Concept 'FOPHDiagnosis'/SPHN Concept
'ICDODiagnosis'/SPHN Concept 'Intent'/SPHN Concept 'NursingDiagnosis'",
     "oneOf": [
                    {
                      "type": "object",
                       "description": "SPHN Concept 'Diagnosis'",
                       "properties": {
                         "id": {
                           "type": "string",
                           "description": "ID of SPHN Concept 'Diagnosis'"
                         },
                          "sphn:hasCodingDateTime": {
                                        …
```

```
        },
        "sphn:hasRecordDateTime": {...

        },
        "sphn:hasAdministrativeCase": {...

        },
        "sphn:hasCode": {...

        },
        "sphn:hasSubjectAge": {...

        },
        "target_concept": {
          "type": "string",
          "description": "IRI for Concept 'Diagnosis'",
          "enum": [
            "https://biomedit.ch/rdf/sphn-ontology/sphn#Diagnosis"
          ]
        }
      },
      "required": [
        "id",
        "sphn:hasCode",
        "target_concept"
      ],
      "additionalProperties": false
    },
```

Please note: **This is NOT the complete schema** for the hasIndicationTostart. For simplicity we are showing here only the relationship: hasIndicationToStart & Diagnosis. While the properties for the Diagnosis concept *"sphn:hasCodingDateTime", "sphn:hasRecordDateTime", "sphn:hasAdmnistrativeCase", sphn:hasCode", "sphn:hasSubjectAge"* have been reduced to their titles.

RML Mapping

This new pattern has also an effect on the created rml mappings. The JSON Path expression used inside of the *'rml:iterator'* searches now explicitly for the target_concept property and the value it is holding. So the hasIndicationToStartMapping for the Diagnosis concept looks explicits for the "target_concept" which should have the IRI of the Diagnosis concept as a value.

```
:hasIndicationToStartMappingDiagnosis a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator
"$..sphn:hasIndicationToStart[?(@.target_concept=='https://biomedit.ch/rdf/sphn-ontology/sphn#Diagnosis')]" ;
```

```
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data_input.json" ] ;
```

An additional mapping has been added to ensure that the hasIndicationToStart property is only created for the DrugPrescription concept if there is a hasIndicationTostart Diagnosis relationship present in the JSON file.

```
:hasIndicationToStartMappingDiagnosisAvailability a rr:TriplesMap ;
    rml:logicalSource [ rml:iterator
"$..sphn:DrugPrescription[?(@.sphn:hasIndicationToStart.target_concept=='https://biomedit.ch/rdf/sphn-ontology/sphn#Diagnosis')]" ;
            rml:referenceFormulation ql:JSONPath ;
            rml:source "patient_data_input.json" ] ;
    rr:predicateObjectMap [ rr:objectMap [ rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-Diagnosis-{sphn:hasIndicationToStart.id}" ] ;
            rr:predicate sphn:hasIndicationToStart ] ;
    rr:subjectMap [ rr:class sphn:DrugPrescription ;
            rr:template
"https://biomedit.ch/rdf/sphn-resource/DATA-PROVIDER-ID-DrugPrescription-{id}" ] .
```

Database schema

In the Database schema the changes are a bit more subtle. An additional layer of nesting has been introduced meaning that the name of the concept (for example “Diagnosis”) is brought into the column names. The further nesting is done as before. Please note that this will cause an increase in the database schema: so you will find more columns in the concept “DrugPrescription” compared to version 2022.

For the DDL the relationship hasIndicationToStart and Diagnosis would look like this:

sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__sphn_hasQuantity__id varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__sphn_hasQuantity__sphn_hasUnit__sphn_hasCode__iri varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasCode__iri varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasCode__sphn_hasIdentifier varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__sphn_hasQuantity__sphn_hasComparator__iri "sphn-2023".sphn_DrugPrescription__sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__sphn_hasQuantity__sphn_hasComparator__iri_type,
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasCode__id varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__sphn_hasQuantity__sphn_hasUnit__id varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasCode__termid varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__sphn_hasDeterminationDateTime TIMESTAMPTZ,
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__id varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasSubjectAge__sphn_hasQuantity__sphn_hasValue numeric,
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasCode__sphn_hasName varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasCodingDateTime TIMESTAMPTZ,
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasRecordDateTime TIMESTAMPTZ,
sphn_hasIndicationToStart__**Diagnosis**__id varchar(3000),
sphn_hasIndicationToStart__**Diagnosis**__sphn_hasCode__sphn_hasCodingSystemAndVersion varchar(3000),

sphn_hasIndicationToStart__**Diagnosis**__sphn_hasAdministrativeCase__id `varchar(3000)`