Metadata Submitter

Release 0.9.0

Apr 28, 2021

Contents:

Metadata Submission service to handle submissions of metadata, either as JSON, XML files or via form submissions via the Single Page Application frontend.

Metadata Submitter is divided intro *Metadata Submitter Backend* and *Metadata Submitter Frontend*, both of them coming together in a Single Page Application that aims to streamline working with metadata and providing a submission process through which researchers can submit and publish metadata.

The application's inteded use is with NeIC SDA (Sensitive Data Archive) stand-alone version, and it consists out of the box includes the ENA (European Nucleotide Archive) metadata model, model which is used also by the European Genome-phenome Archive (EGA).

Out of the box the `metadata-submitter` offers:

- flexible REST API for working with metadata;
- validating metadata objects against ENA XSD metadata models and their respective JSON schema;
- asynchronous web server;
- OIDC authentication;
- dynamic forms based on JSON schemas;
- simple wizard for submitting metadata.

A command-line interface for validating any given XML file against a specific XSD Schema has also been implemented see *XML Validation CLI*.

CHAPTER 1

Metadata Submitter Backend

Note: Requirements:

- Python 3.8+
- MongoDB

1.1 Environment Setup

The application requires some environmental arguments in order to run properly, these are illustrated in the table below.

ENV	Default	Description	Mandatory
MONGO_HOST	localhost:27017	MongoDB server hostname, with port specified if needed.	Yes
MONGO_AUTHDB	-	MongoDB authentication database.	Yes
MONGO_DATABASE	default	MongoDB default database, will be used as authentication database if MONGO_AUTHDB is not set.	No
MONGO_USERNAME	admin	Admin username for MongoDB.	Yes
MONGO_PASSWORD	admin	Admin password for MongoDB.	Yes
MONGO_SSL	-	Set to True to enable MONGO TLS connection url.	No
MONGO_SSL_CA	-	Path to CA file, required if MONGO_SSL enabled.	No
MONGO_SSL_CLIENT_KEY	-	Path to contains client's TLS/SSL X.509 key,required if MONGO_SSL enabled.	No
MONGO_SSL_CLIENT_CERT	-	Path to contains client's TLS/SSL X.509 cert,required if MONGO_SSL enabled.	No
AAI_CLIENT_SECRET	`secret`	OIDC client secret.	Yes
AAI_CLIENT_ID	secret	OIDC client ID.	Yes
AUTH_REFERER	-	OIDC Provider url that redirects the request to the application.	Yes
BASE_URL	http://localhost:5430	base URL of the metadata submitter.	Yes
ISS_URL	-	OIDC claim issuer URL.	Yes
AUTH_URL	-	Set if a special OIDC authorize URL is required, otherwise use "OIDC_URL"/authorize.	No
OIDC_URL	-	OIDC base URL for constructing OIDC provider endpoint calls.	Yes
REDIRECT_URL	-	Required only for testing with front-end on localhost or change to http://frontend:3000 if started using docker-compose (see *Build and Deployment*).	No
JWK_URL	-	JWK OIDC URL for retrieving key for validating ID token.	Yes
LOG_LEVEL	INFO	Set logging level, uppercase.	No
SERVE_KEY	-	Keyfile used for TLS.	No
SERVE_CERT	-	Certificate used for TLS.	No
SERVE_CA	-	CA file used for TLS.	No
SERVE_SLLVERSION	-	Version used for TLS, see the gunicorn documentation for ssl_version for more information.	No
SERVE_CIPHERS	-	Ciphers used for TLS, see the gunicorn documentation for ciphers for more information.	No
SERVE_CERTREQS	-	Client certificate requirement used for TLS, see the gunicorn documentation for cert_reqs for more information.	No

Note: If just MONGO_DATABASE is specified it will autenticate the user against it. If just MONGO_AUTHDB is specified it will autenticate the user against it. If both MONGO_DATABASE and MONGO_AUTHDB are specified, the client will attempt to authenticate the specified user to the MONGO_AUTHDB database. If both MONGO_DATABASE and MONGO_AUTHDB are unspecified, the client will attempt to authenticate the specified user to the admin database.

1.2 Install and run

For installing metadata-submitter backend do the following:

```
$ git clone https://github.com/CSCfi/metadata-submitter
$ pip install .
```

Hint: Before running the application have MongoDB running.

MongoDB Server expects to find MongoDB instance running, spesified with following environmental variables:

- `MONGO_INITDB_ROOT_USERNAME` (username for admin user to mondogdb instance)
- `MONGO_INITDB_ROOT_PASSWORD` (password for admin user to mondogdb instance)
- `MONGO_HOST` (host and port for MongoDB instance, e.g. *localhost:27017*)

To run the backend from command line use:

```
$ metadata_submitter
```

Hint: For a setup that requires also frontend follow the instructions in *Build and Deployment*.

1.3 Authentication

The Authentication follows the OIDC Specification.

We follow the steps of the OpenID Connect protocol.

- The RP (Client) sends a request to the OpenID Provider (OP), for this we require `AAI_CLIENT_SECRET`, `AAI_CLIENT_ID`, `OIDC_URL`, a callback url constructed from `BASE_URL` and `AUTH_URL` if required.
- The OP authenticates the End-User and obtains authorization.
- The OP responds with an ID Token and usually an Access Token, we validate the ID Token for which we need `JWK_URL` to get the key and `ISS_URL` to check the claims issuer is correct.
- The RP can send a request with the Access Token to the UserInfo Endpoint.
- The UserInfo Endpoint returns Claims about the End-User, use use some claims `sub` and `eppn` to identify the user and start a session.

Information related to the OpenID Provider (OP) that needs to be configured is displayed in the table below. Most of the information can be retrieved from OIDC Provider metadata endpoint `https://<provider_url>/.well-known/openid-configuration`.

ENV	Default	Description	Mandatory
AAI_CLIENT_SECRET	`generic`	OIDC client secret.	Yes
AAI_CLIENT_ID	secret	OIDC client ID.	Yes
AUTH_REFERER	–	OIDC Provider url that redirects the request to the application.	Yes
BASE_URL	http://localhost:5430	base URL of the metadata submitter.	Yes
ISS_URL	–	OIDC claim issuer URL.	Yes
AUTH_URL	–	Set if a special OIDC authorize URL is required, otherwise use "OIDC_URL"/authorize.	No
OIDC_URL	–	OIDC base URL for constructing OIDC provider endpoint calls.	Yes
JWK_URL	–	JWK OIDC URL for retrieving key for validating ID token.	Yes

1.4 REST API

View metadata submitter API in swagger editor.

The REST API is structured as follows:

- *Submission Endpoints* used in submitting data, mostly POST endpoints;
- *Query Endpoints* used for data retrieval (folders, objects, users) uses HTTP GET;
- *Management Endpoints* used for handling data updates and deletion, makes use of HTTP PUT, PATCH and DELETE.

Important: A logged in user can only perform operations on the data it has associated. The information for the current user can be retrieved at /users/current (the user ID is current), and any additional operations on other users are rejected.

1.5 Metadata Backend Modules

Backend for submitting and validating XML Files containing ENA metadata.

metadata_backend.api	API endpoints and other api-related classes.
metadata_backend.conf	App configurations.
metadata_backend.database	Database services, initialisation and other tools.
metadata_backend.helpers	Helper tools, such as app configurations, logging and data validators.
metadata_backend.server	Functions to launch backend server.

1.5.1 Metadata Backend API

API endpoints and other api-related classes.

metadata_backend.api.auth	Handle Access for request and OIDC workflow.
metadata_backend.api.handlers	Handle HTTP methods for server.
metadata_backend.api.health	Handle health check endpoint.
metadata_backend.api.middlewares	Middleware methods for server.
metadata_backend.api.operators	Operators for handling database-related operations.

1.5.2 Database Operations

Database services, initialisation and other tools.

metadata_backend.database.db_service	Services that handle database connections.

1.5.3 Utility Functions

Helper tools, such as app configurations, logging and data validators.

metadata_backend.helpers.logger	Logging formatting and functions for debugging.
metadata_backend.helpers.parser	Tool to parse XML files to JSON.
metadata_backend.helpers.schema_loader	Utility class to find XSD Schema that can be used to test XML files.
metadata_backend.helpers.validator	Utility classes for validating XML or JSON files.

1.5.4 Configuration

App configurations.

metadata_backend.conf.conf	Python-based app configurations.

1.5.5 Server

Functions to launch backend server.

metadata_backend.server.**init**() → aiohttp.web_app.Application
Initialise server and setup routes.

Routes should be setup by adding similar paths one after the another. (i.e. POST and GET for same path grouped together). Handler method names should be used for route names, so they're easy to use in other parts of the application.

Note: if using variable resources (such as {schema}), add specific ones on top of more generic ones.

metadata_backend.server.**kill_sess_on_shutdown**(*app: aiohttp.web_app.Application*) → None
Kill all open sessions and purge their data when killed.

metadata_backend.server.**main**() → None
Launch the server.

CHAPTER 2

Metadata Submitter Frontend

Note: Requirements:

- Node 14+

2.1 Environment Setup

The frontend can utilise the following env variables.

ENV	Default	Description
`NODE_ENV`	-	Set to `development`, if running in development mode.
`REACT_APP_BACKEND_PROXY`	`localhost:5430`	Proxy frontend requests to this backend, port must be specified.
Cypress.env `port`	`3000`	Port Cypress can use for integration tests. Can be set in `cypress.json`

2.2 Install and run

For installing `metadata-submitter` frontend do the following:

```
$ git clone https://github.com/CSCfi/metadata-submitter-frontend
$ npm install
```

To run the frontend from command line use:

```
$ npm start
```

After installing and running, frontend can be found from `http://localhost:3000`.

Hint: Some functionality in frontend requires a working backend. Follow the instructions in *Build and Deployment* for setting it up.

2.3 Internal structure

Reusable components are stored in `src/components` and views in `src/views`. View-components reflect page structure, such as `/`, `/newdraft`, `/login` etc. One should not define and export views to be rendered inside other views, but rather always build views using components.

React Router is used to render different views in App-component. All components are wrapped with *Nav* which provider app menu and navigation.

2.4 Form components

Form components are crucial part of the application:

- All submissions and folder creation are made with react-hook-form. Latter uses form as a reference so submission can be triggered outside the form. JSON schema based forms are created with custom JSON schema parser, which builds `react-hook-form` based forms from given schema. The forms are validated against the JSON schema with `Ajv`. React-hook-form is used for performance reasons: it uses uncontrolled components so adding a lot of fields to array doesn't slow rendering of the application.

2.5 Constants

Folder `src/constants` holds all the constants used in the application. The constants are uniquely defined and separated into different files according to its related context. For example, the file `constants/wizardObject.js` contains unique constants regarding to `wizardObject` such as: `ObjectTypes, ObjectStatus, etc.`

The purposes of using these *constants* are:

- to avoid hard coding the values of variables repeatedly
- to keep the consistency when defining the values of variables
- to reuse those predefined values across the application

Example of defining and using a constant:

- First, define the constant object `ObjectSubmissionTypes` in `constants/wizardObject.js`

```
export const ObjectSubmissionTypes = {
form: "Form",
xml: "XML",
existing: "Existing",
}
```

- Then, use this constant in *WizardComponents/WizardObjectIndex*:

```
import { ObjectSubmissionTypes } from "constants/wizardObject"

switch (currentSubmissionType) {
    case ObjectSubmissionTypes.form: {
    target = "form"
    break
    }
    case ObjectSubmissionTypes.xml: {
    target = "XML upload"
    break
    }
    case ObjectSubmissionTypes.existing: {
    target = "drafts"
    break
    }
}
```

2.6 Commonly used data types

All commonly used data types of variables are defined in the file `index.js` in folder `src/types`. The purposes are:

- to avoid hard coding the same data types frequently in different files
- to keep track and consistency of the data types across different files

For example:

- declare and export these data types in `src/types/index.js`

```
export type ObjectInsideFolder = {
accessionId: string,
schema: string,
}

export type ObjectTags = {
submissionType: string,
fileName?: string,
}

export type ObjectInsideFolderWithTags = ObjectInsideFolder & { tags: ObjectTags }
```

- import and reuse the data types in different files:
- Reuse type `ObjectInsideFolder` in `features/wizardSubmissionFolderSlice.js`:

```
import type { ObjectInsideFolder } from "types"

export const addObjectToFolder = (
folderID: string,
objectDetails: ObjectInsideFolder
) => {}

export const addObjectToDrafts = (
folderID: string,
objectDetails: ObjectInsideFolder
) => {}
```

- Reuse type `ObjectInsideFolderWithTags` consequently in both `WizardComponents/WizardSavedObjectsList.js` and `WizardSteps/WizardShowSummaryStep.js`:

```
import type { ObjectInsideFolderWithTags } from "types"

type WizardSavedObjectsListProps = { submissions: Array<ObjectInsideFolderWithTags> }
```

```
import type { ObjectInsideFolderWithTags } from "types"

type GroupedBySchema = {| [Schema]: Array<ObjectInsideFolderWithTags> |}
```

2.7 Redux store

Redux is handled with Redux Toolkit and app is using following redux toolkit features:

- Store, global app state, configured in `store.js`
- Root reducer, combining all reducers to one, configured in `rootReducer.js`
- Slices with `createSlice`-api, defining all reducer functions, state values and actions without extra boilerplate. - Slices are configured for different features in `features/` -folder. - Async reducer functions are also configured inside slices.

Examples for storing and dispatching with async folder function:

```
import { useSelector, useDispatch } from "react-redux"
import { createNewDraftFolder } from "features/submissionFolderSlice"

// Create base folder (normally from form)
const folder = {
name: "Test",
description: "Test description for very best folder."
}

// Initialize dispatch with hook
const dispatch = useDispatch()

// Dispatch the action with folder
dispatch(createNewDraftFolder(folder))

// Folder is now submitted to backend and added to redux store

// Take folder from redux state, destructure and log values
const folder = useSelector(state => state.submissionFolder)
const { id, name, description, metadataObjects } = folder
console.log(id) // Should be id generated in backend
console.log(name) // Should be name we set earlier
console.log(description) // Should be description we set earlier
console.log(metadataObjects) // Should be an empty array
```

2.8 Communicating with backend REST API

API/backend modules are defined in `services/` -folder with help from `apisauce` library. Modules should be only responsible for API-related things, so one shouldn't modify data inside them.

Example:

```
import { create } from "apisauce"

const api = create({ baseURL: "/objects" })

const createFromXML = async (objectType: string, XMLFile: string) => {
let formData = new FormData()
formData.append(objectType, XMLFile)
return await api.post(`/${objectType}`, formData)
}

const createFromJSON = async (objectType: string, JSONContent: any) => {
return await api.post(`/${objectType}`, JSONContent)
}
```

2.9 Styles

App uses Material UI components.

Global styles are defined with `style.css` and Material UI theme, customized for CSC. Material UI theme is set `theme.js`, and added to `index.js` for use.

Styles are also used inside components, either with `withStyles` (modifies Material UI components) or `makeStyles` (creates css for component and its children). See customizing components for more info.

CHAPTER 3

Metadata Model

3.1 ENA Metadata Model

The object schemas that are used for rendering the forms and validating the information submitted to the application are based on the ENA (European Nucleotide Archive) Metadata Model.

The source XML schemas are from ENA Sequence Github repository. The XML schemas are converted to JSON Schemas so that they can be both validate the submitted data as well as be rendered as forms in the User Interface. For this reason the translation from XML Schema to JSON schema is not a 1-1 mapping, but an interpretation.

The ENA model consists of the following objects:

- `Study`: A study groups together data submitted to the archive. A study accession is typically used when citing data submitted to ENA. Note that all associated data and other objects are made public when the study is released.
- `Project`: A project groups together data submitted to the archive. A project accession is typically used when citing data submitted to ENA. Note that all associated data and other objects are made public when the project is released.
- `Sample`: A sample contains information about the sequenced source material. Samples are typically associated with checklists, which define the fields used to annotate the samples.
- `Experiment`: An experiment contain information about a sequencing experiment including library and instrument details.
- `Run`: A run is part of an experiment and refers to data files containing sequence reads.
- `Analysis`: An analysis contains secondary analysis results derived from sequence reads (e.g. a genome assembly).
- `DAC`: An European Genome-phenome Archive (EGA) data access committee (DAC) is required for authorized access submissions.
- `Policy`: An European Genome-phenome Archive (EGA) data access policy is required for authorized access submissions.

- `Dataset`: An European Genome-phenome Archive (EGA) data set is required for authorized access submissions.

3.1.1 Relationships between objects

Each of the objects are connected between each other by references, usually in the form of an `accessionId`. Some of the relationships are illustrated in the Metdata ENA Model figure, however in more detail they are connected as follows:

- `Study` - usually other objects point to it, as it represents one of the main objects of a `Submission`;
- `Analysis` - contains references to:
  - parent `Study` (not mandatory);
  - zero or more references to objects of type: `Sample`, `Experiment`, `Run`;
- `Experiment` - contains references to exactly one parent `Study`. It can also contain a reference to `Sample` as an individual or a Pool;
- `Run` - contains reference to exactly one parent `Experiment`;
- `Policy` - contains reference to exactly one parent `DAC`;
- `Dataset` - contains references to:
  - exactly one `Policy`;
  - zero or more references to objects of type: `Analysis` and `Run`.

3.1.2 EGA/ENA Metadata submission Guides

Related guides for metadata submission:

- EGA Metadata guides:
  - Submitting array based metadata
  - Submitting sequence and phenotype data
- ENA Data Submission general Guide

CHAPTER 4

Build and Deployment

4.1 Development Deployment

For integration testing and local development we recommend `docker-compose`, which can be installed using `pip install docker-compose`.

4.1.1 Deploying Backend

Check out backend repository.

For quick testing, launch both server and database with Docker by running `docker-compose up --build` (add `-d` flag to run containers in background). Server can then be found from `http://localhost:5430`.

This will launch a version without the frontend.

4.1.2 Deploying Frontend

Check out frontend repository.

For quick testing run `docker-compose up --build` (add `-d` flag to run container in the background). By default, frontend tries to connect to docker container running the backend. Feel free to modify `docker-compose.yml` if you want to use some other setup.

Integrating Frontend and Backend

With backend running as container and frontend with `npm`:

1. check out metadata submitter backend repository
2. un-commented line 24 from `docker-compose.yml`
3. `docker-compose up -d --build` backend repository root directory

4. check out metadata submitter frontend repository
5. `npm start` frontend repository root directory

With backend and frontend running in containers:

1. check out metadata submitter backend repository
2. un-commented line 24 from `docker-compose.yml` and modify to `http://frontend:3000`
3. `docker-compose up -d --build` backend repository root directory
4. check out metadata submitter frontend repository
5. `docker-compose up -d --build` frontend repository root directory

4.2 Production Deployment

To ease production deployment Frontend is built and added as static files to backend while building the Docker image. The production image can be built and run with following docker commands:

```
docker build --no-cache . -t cscfi/metadata-submitter
docker run -p 5430:5430 cscfi/metadata-submitter
```

Important: Requires running MongoDB and consider setting the environment variables as pointed out in *Metadata Submitter Backend*.

4.2.1 Kubernetes Deployment

For deploying the application as part of Kubernetes us the helm charts from: https://github.com/CSCfi/metadata-submitter-helm/

CHAPTER 5

Testing

Note: Unit tests and integration tests are automatically executed with every PR to for both frontend and backend in their respective repositories.

5.1 Backend Testing

Tests can be run with tox automation: just run `tox` on project root (remember to install it first with `pip install tox`).

5.1.1 Unit Testing

In order to run the unit tests, security checks with bandit, Sphinx documentation check for links consistency and HTML output and flake8 (coding style guide) tox. To run the unit tests in parallel use:

```
$ tox -p auto
```

To run environments seprately use:

```
$ # list environments
$ tox -l
$ # run flake8
$ tox -e flake8
$ # run bandit
$ tox -e bandit
$ # run docs
$ tox -e docs
```

5.1.2 Integration Testing

Integration tests required a running backend, follow the instructions in *Build and Deployment* for development setup of backend. After the backend has been successfully setup run in backend repository root directory `python tests/integration/run_tests.py`. This command will run a series of integration tests.

5.2 Frontend Testing

Run Jest-based tests with `npm test`. Check code formatting and style errors with `npm run lint:check` and fix them with `npm run lint`. Respectively for formatting errors in `json/yaml/css/md` -files, use `npm run format:check` or `npm run format`. Possible type errors can be checked with `npm run flow`.

We're following recommended settings from `eslint`, `react` and `prettier` - packages witha a couple of exceptions, which can be found in `.eslintrc` and `.prettierrc`. Linting, formatting and testing are also configured for you as a git pre-commit, which is recommended to use to avoid fails on CI pipeline.

5.2.1 End to End testing

End-to-end tests can be run on local host with `npx cypress open` in frontend repository. These tests required a running backend, follow the instructions in *Build and Deployment* for development setup of backend.

If the frontend is started with `npm start` no changes required in the setup.

CHAPTER 6

XML Validation CLI

A command-line interface for validating any given XML file against a specific XSD Schema has also been implemented. The tool can be found and installed from metadata-submitter-tools repository.

6.1 Usage

After the package has been installed, the validation tool is used by by executing `xml-validate` in a terminal with specified options/arguments followingly:

```
$ xml-validate <option> <xml-file> <schema-file>
```

The `<xml-file>` and `<schema-file>` arguments need to be the correct filenames (including path) of a local XML file and the corresponding XSD file. The `<option>` can be `--help` for showing help and `-v` or `--verbose` for delivering a detailed validation error message.

Below is a terminal demonstration of the usage of this tool, which displays the different outputs the CLI will produce:

CHAPTER 7

Indices and tables

- genindex
- modindex
- search