Table of contents

Usage . . . 2
User interface . . . 2
Interfaces for accessing and querying data (API) . . . 8
Structure and terminology . . . 11
Workflow and access modes . . . 12
Roles and permissions . . . 14
Data model and view configuration . . . 14
Installation and configuration . . . 17
Local development . . . 17
Kubernetes and helm . . . 18
Design . . . 20
Storage . . . 20
License . . . 20

Fairspace is a secure place for managing research data. Research teams have their own workspaces in which they can manage research data collections. Researchers can upload directories and files to data collections. Data access is organised on data collection level. Collections can be shared with other teams or individual researchers. Also, collections can be published for all researchers in the organisation.

Collections and files can be annotated with descriptive metadata. The metadata is stored using the Resource Description Framework (RDF) in an Apache Jena database. For the metadata, a data model can be configured that suits the data management needs of the organisation. The data model is specified using the Shapes Constraint Language (SHACL), see the section on Data model and view configuration. Descriptive metadata entities (e.g., subjects, projects, samples) should be added to the database by a careful process, ensuring that duplicates and inconsistencies are avoided and all entities have proper unique identifiers. The application provides overviews of the available metadata entities. In the collection browser, researchers can link their collections and file to these entities or add textual descriptions and key words.

Key features

- Fairspace is a data repository that enables researchers to securely **store** and **organise** their research data sets, and **share** the data with collaborators.
- Fairspace lets researchers annotate their data collections with relevant metadata properties and link the data to associated metadata entities (subjects, samples, projects, etc.). This helps researchers find their own data and make it **findable** for others, contributing to implementation of the FAIR principles.
- Fairspace ensures that all metadata entities have a unique identifier and validates metadata consistency and validity upon data entry.
- Fairspace allows organisations to use **customise** the configured data model, by specifying custom entity types and constraints. This enables the adoption of community standards for metadata relevant for the research domain, which contributes to the **reusability** of the data.
- Fairspace uses the Resource Description Framework (RDF) and WebDAV standards for data exchange, and stimulates the uses of standard vocabularies, contributing to **interoperability** of data.

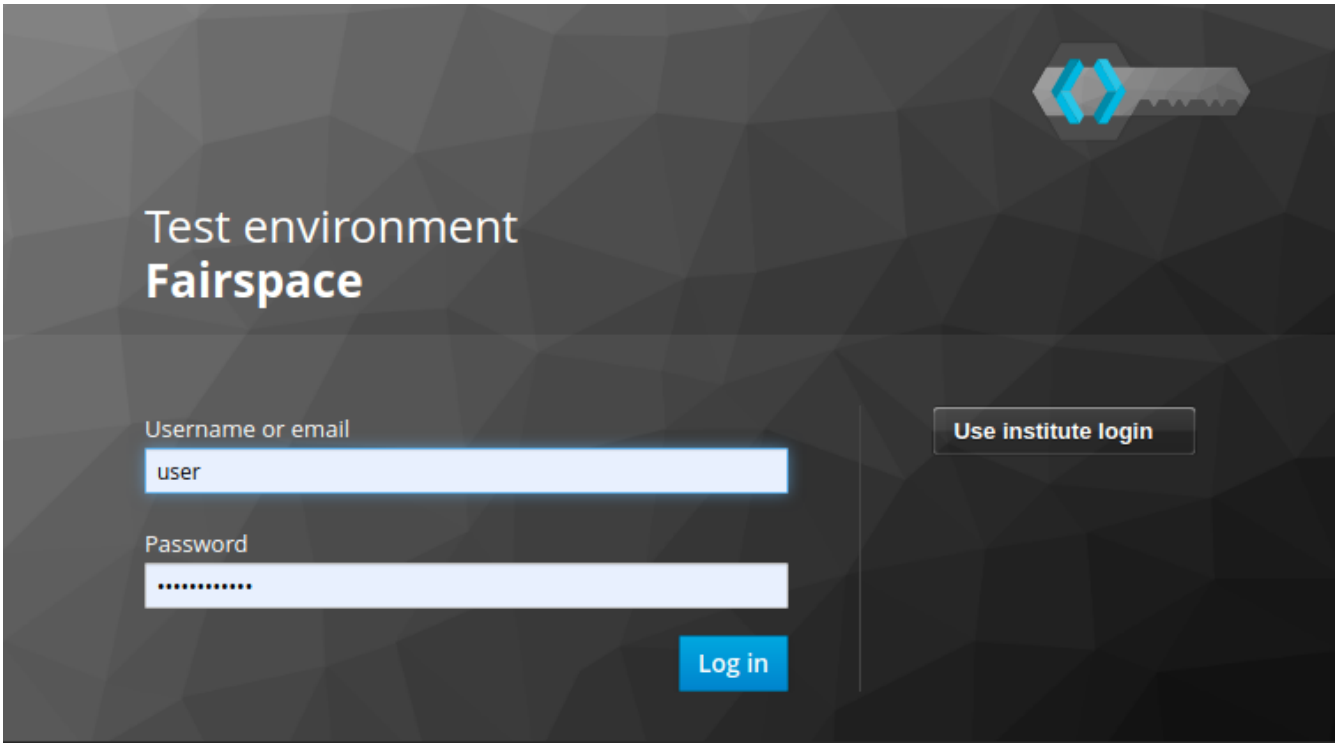
Usage

User interface

Login

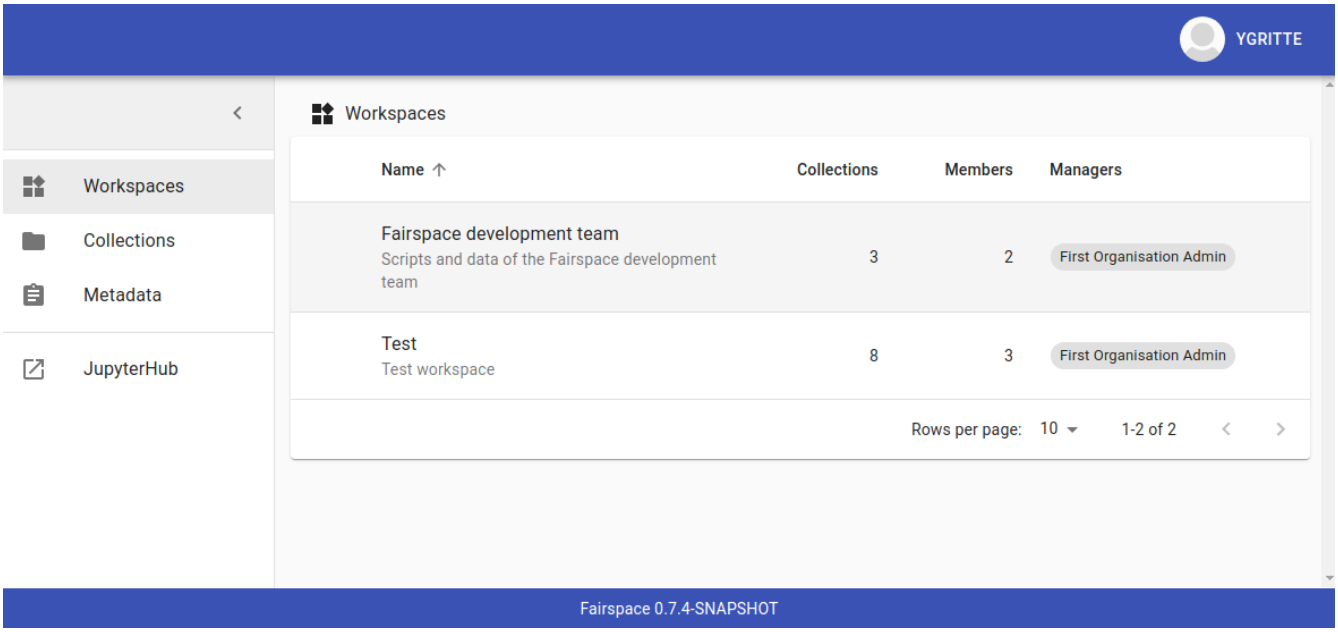
Users are authenticated using Keycloak, an open-source identity provider that provides secure authentication methods and can be configured to integrate with institutional identity providers using user federation or identity brokering, see the Keycloak server administration pages.

The user either logs in directly using Keycloak or is forwarded to a configured external login:

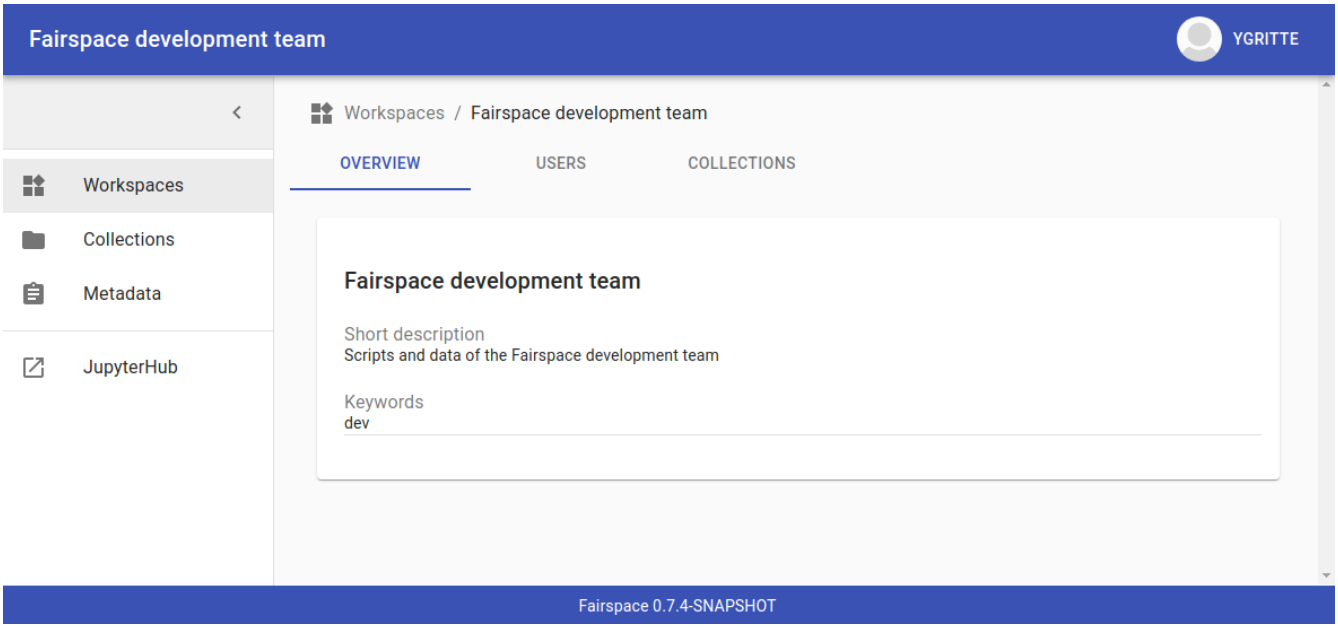


Workspaces

Users enter Fairspace on the workspaces page that lists all workspaces. A workspace represents a team in the organisation that collaborates on research data collections.



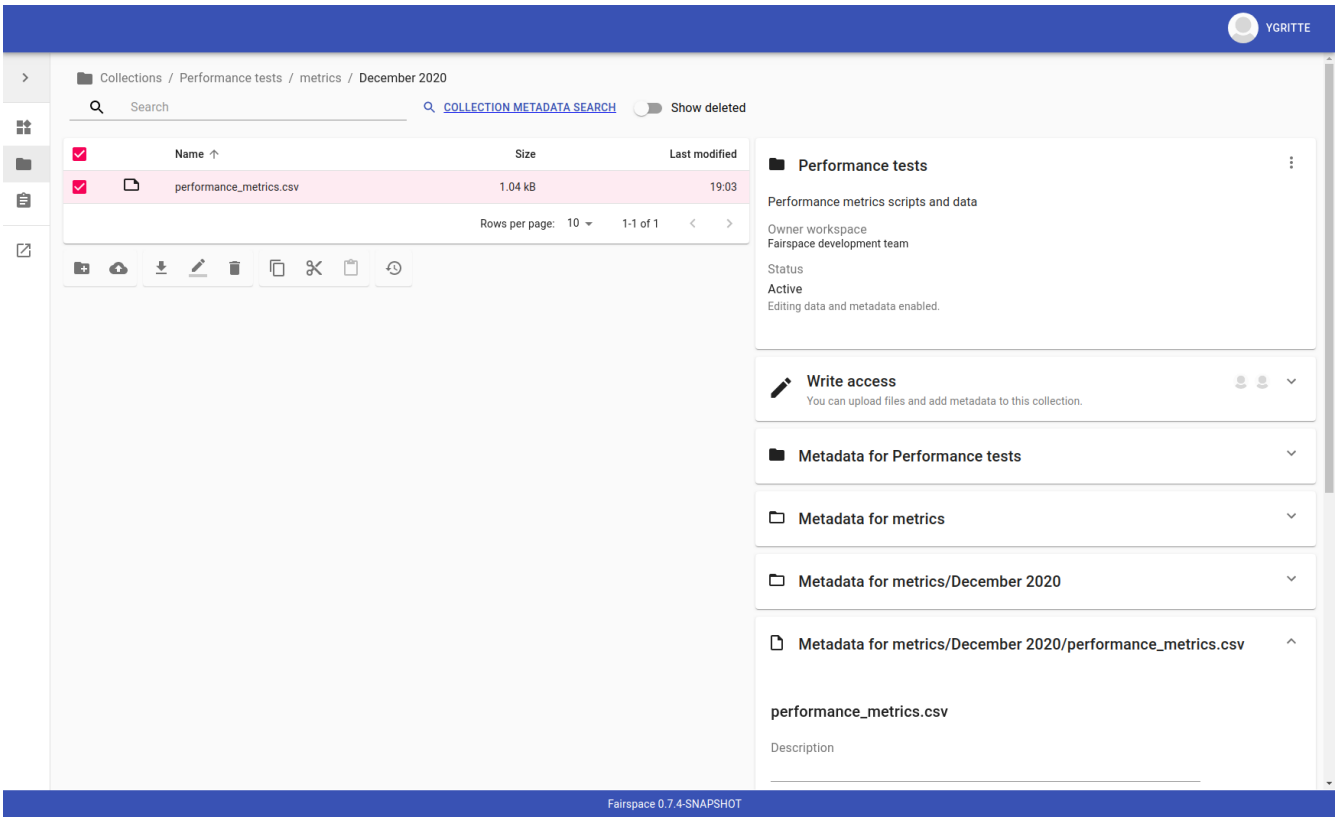
Workspace administrators can edit the workspace overview page and manage workspace membership. All workspace members can add collections to the workspace.



Collections

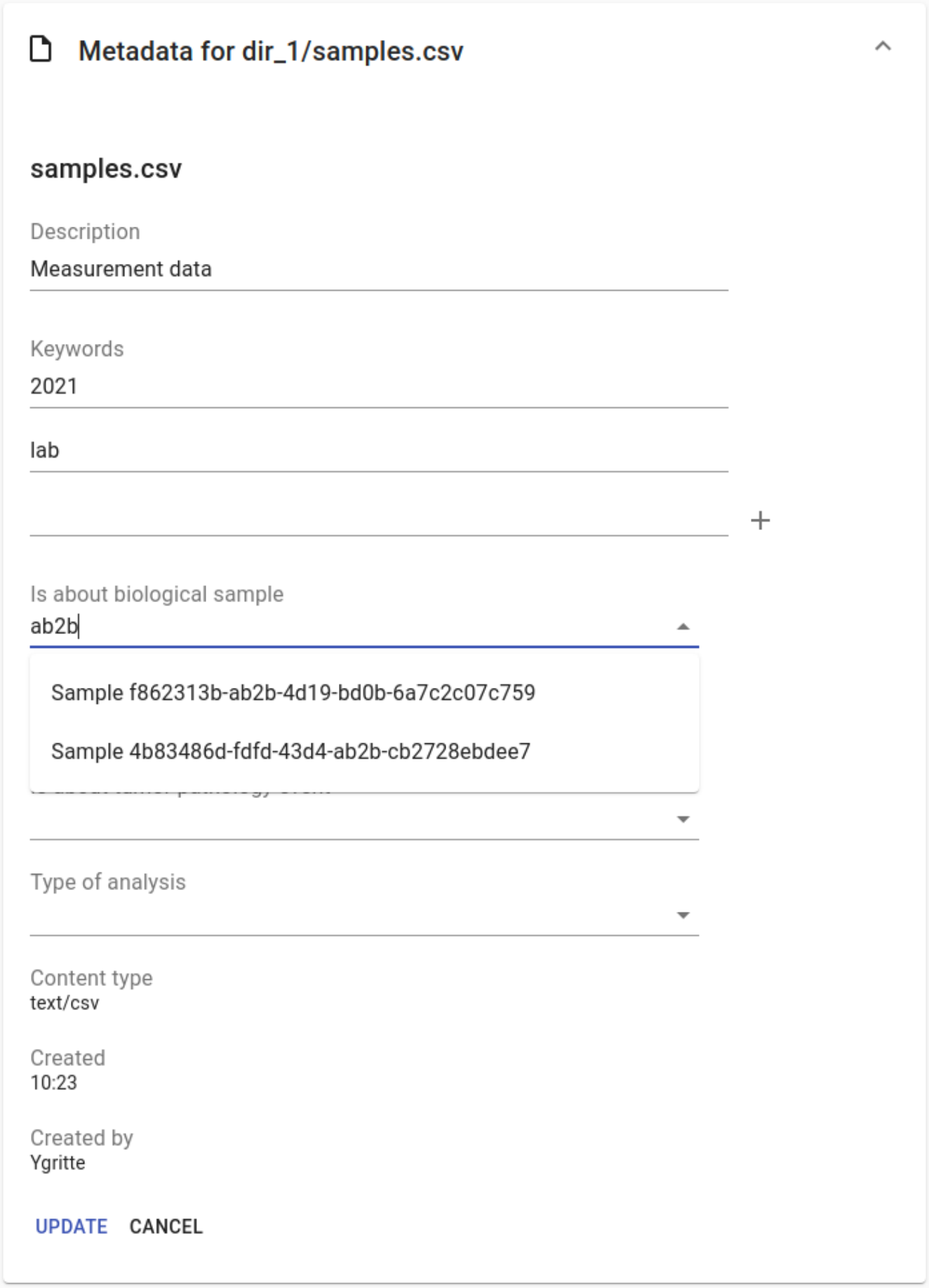
The contents of collections can be navigated in the collections browser. It behaves like a regular file browser. Click to select a directory or file and see its metadata, double click to navigate into directories or open a file.

Access is managed on collection level. Users with at least write access to a collection can upload files or directories, rename or delete files, restore old file versions, and edit the associated metadata.



Metadata forms

Users with write access to the collection can annotate collections, directories and files using *metadata forms*. Free text fields, like description and key words, can be entered freely, links to shared entities, like subjects, samples and projects, or values from a controlled vocabulary, like taxonomy or analysis type, can be selected from a list:



The shared metadata entities and controlled vocabularies cannot be added via the user interface. The Metadata upload API should be used for that instead.

Metadata upload

Another way to annotate directories and files is by uploading a comma-separated values (CSV) file with metadata. This section describes the CSV-based format used for bulk metadata uploads.

The file should be a valid CSV-file:

- Records are separated with a `,`-character.
- Values may be enclosed in double quotes: `"value"`.
- In values that contain a double, the double quotes need to be escaped by replacing them with double double quotes: `Example "quoted" text` becomes `"Example ""quoted"" text"`.

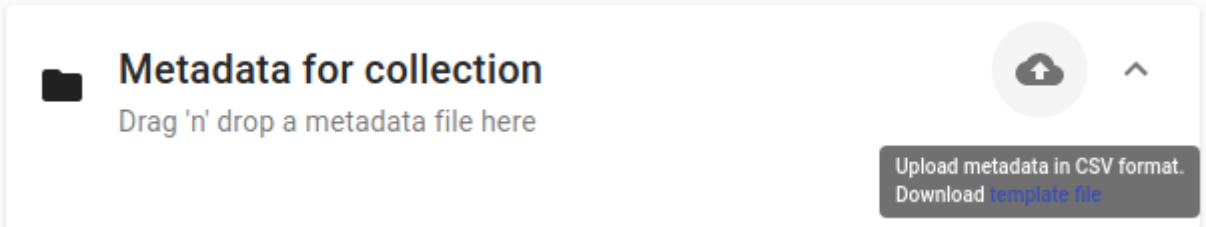
In the metadata upload, lines starting with `#` are ignored. These lines are considered to be comments.

The file should have a header row containing the names of the columns. The mandatory `Path` column is used for the file path. For the property columns, the name should match exactly the name of the property in the database.

The format of the values is as follows:

- *Path*: the relative path to a file or a directory (relative to the collection or directory where the file is uploaded). Use `./` for the current directory or collection.
- *Entity types* can be referenced by ID or unique label.
- Multiple values must be separated by the pipe symbol `|`, e.g., use `test|lab` to enter the values `test` and `lab`.

The file can be uploaded to the current directory by dropping the file in the metadata panel of the directory, or by selecting the metadata upload button.
By hovering over the metadata upload button, a link to a *metadata template file* becomes available:



The file describes the format in commented lines and contains the available properties in the header row.

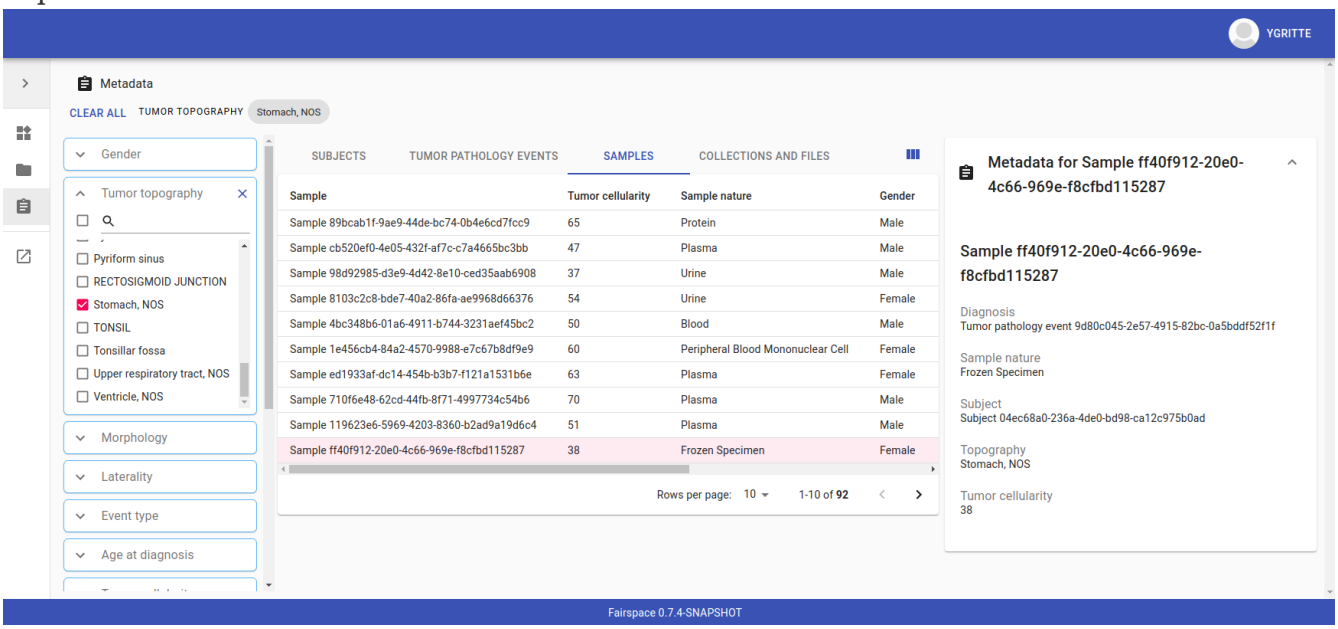
Example 1. Example metadata file

An example comma-separated values file with metadata about the current directory `./`, which is annotated with a description and two key words (`sample` and `lab`), and the file `test.txt` which is linked to Subject 1 by the unique subject label and to the RNA-seq analysis type by the analysis type identifier (`06-12`).

```
Path,Is about subject,Type of analysis,Description,Keywords
./,,,Directory with samples,sample|lab,
test.txt,Subject 1,https://institut-curie.org/analysis#06-12,,
```

Metadata

Explore metadata and find associated collections and files.



Interfaces for accessing and querying data (API)

Authentication

OpenID Connect (OICD) / OAuth2 workflow

Via header, via session.

Fetch token (Python code example)

```
import logging
import requests
import sys
import time

log = logging.getLogger()

def fetch_access_token(keycloak_url: str,
                       realm: str,
                       client_id: str,
                       client_secret: str,
                       username: str,
                       password: str) -> str:
    """
    Obtain access token from Keycloak
    :return: the access token as string.
    """
    params = {
        'client_id': client_id,
        'client_secret': client_secret,
        'username': username,
        'password': password,
        'grant_type': 'password'
    }
    headers = {
        'Content-type': 'application/x-www-form-urlencoded',
        'Accept': 'application/json'
    }
    response = requests.post(f'{keycloak_url}/auth/realms/{realm}/protocol/openid-
connect/token',
                             data=params,
                             headers=headers)
    if not response.ok:
        log.error('Error fetching token!', response.json())
        sys.exit(1)
    data = response.json()
    token = data['access_token']
    log.info(f"Token obtained successfully. It will expire in {data['expires_in']}
seconds")
    return token
```

Basic authentication

Use the `base64` encoded `username:password` in the `Authorization` header.

Authentication (Curl code example)

```
curl -v -H "Authorization: Basic $(echo -n "${USERNAME}:${PASSWORD}" | base64)"
http://localhost:8080/api/users/current
```

Automatic authentication in Jupyter Hub

Metadata upload API

Metadata can be specified using:

- turtle
- json-ld

Example file: `testdata.ttl`:

```
@prefix example: <https://example.com/ontology#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix subject: <http://example.com/subjects#> .
@prefix file: <http://example.com/api/webdav/> .
@prefix gender: <http://hl7.org/fhir/administrative-gender#> .
@prefix ncbitaxon: <https://bioportal.bioontology.org/ontologies/NCBITAXON/> .
@prefix dcat: <http://www.w3.org/ns/dcat#> .

subject:s1 a example:Subject ;
           rdfs:label "Subject 1" ;
           example:isOfSpecies ncbitaxon:9606 .

file:coll1\/coffee.jpg
    dcat:keyword "fairspace", "java" ;
    example:aboutSubject example:s1 .
```

Example with Python.

```
import logging
from requests import Session
import sys

log = logging.getLogger()

session = Session()
with open('testdata.ttl') as testdata:
    response: Response = session.put(f"{server_url}/api/metadata/",
                           data=testdata.read(),
                           headers={'Content-type': 'text/turtle'})
    if not response.ok:
        log.error('Error uploading metadata!')
        log.error(f'{response.status_code} {response.reason}')
        sys.exit(1)
```

```
curl -v
```

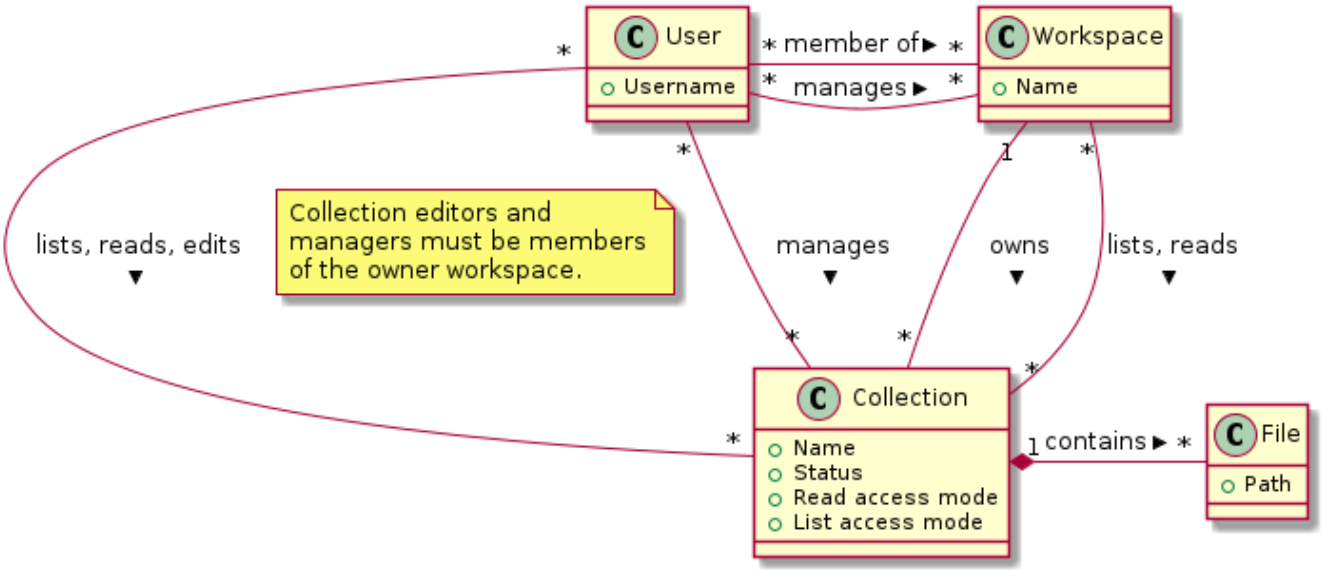
WebDAV

Structure and terminology

In this section we describe in detail the main concepts and components of the Fairspace data repository and how they relate to each other.

The core entities of the data repository are:

- *Users*: individual users in the organisation, looking for data, contributing to data collections or managing data.
- *Workspaces* (for projects, teams): entities in the system linked, representing a group of users, to organise data collections and data access.
- *Collections*: entities in the system to group data files. These are the minimal units of data for data access and data modification rules.
- *Files*: The smallest units of data that the system processes. Files always belong to a single collection. Files can be added, changed and deleted, but not in all collection states. Changing a file creates a new version. Access to a file is based on access to the collection the file belongs to. Files can be organised in *Directories*, which we will leave out of most descriptions for brevity.



The diagram above sketches the relevant entities and actors. The basic structure consists of users, workspaces, collections and files as represented in the system. Collections are the basic units of data access management. A collection is owned by a workspace. The responsibility for a collection is organised via the owner workspace: members of the owner workspace can be assigned as editors or managers of the collection. This reflects the situation where in an organisation, a data collection belongs to a project or a research team. This way the workspace represents the organisational unit that is responsible for a number of data collections (e.g., a research team or project). Data can be shared with other workspaces or individual users (for reading) and ownership may be transferred to another workspace (e.g., in the case the workspace is temporary, or when the organisation changes).

Fairspace provides a *data catalogue*, containing all the metadata, which is visible for all users with catalogue access (*View public metadata*). Users with metadata write access (*Add shared metadata*) can add metadata to the catalogue. Preferably this is done by an automated process that ensures the consistency of the metadata and uniqueness of metadata entities. Metadata on collection and file level is protected by the access policy of the collections.

User administration is organised in an external component ([Keycloak]), but user permissions are stored in Fairspace. A back end application is responsible for storing the data and metadata, and for providing APIs for securely retrieving and adding data and metadata using standard data formats and protocols. A user interface application provides an interactive file manager and (meta)data browser and data entry forms based on the back end APIs. Besides the data storage and data management, Fairspace offers *analysis environments* using Jupyter Hub. In Jupyter Hub, the data repository is accessible. Every user has a private working directory. We do no assumptions on the structure of the data or on the permissions of the external file systems that are connected to the data repository and referenced in the data catalogue. The organisation structure may be replicated in the different systems in incompatible ways, and the permissions may not be aligned.

Workflow and access modes

During the lifetime of a collection, different rules may be applicable for data modification and data access. In Fairspace, collections follow a workflow with the following statuses:

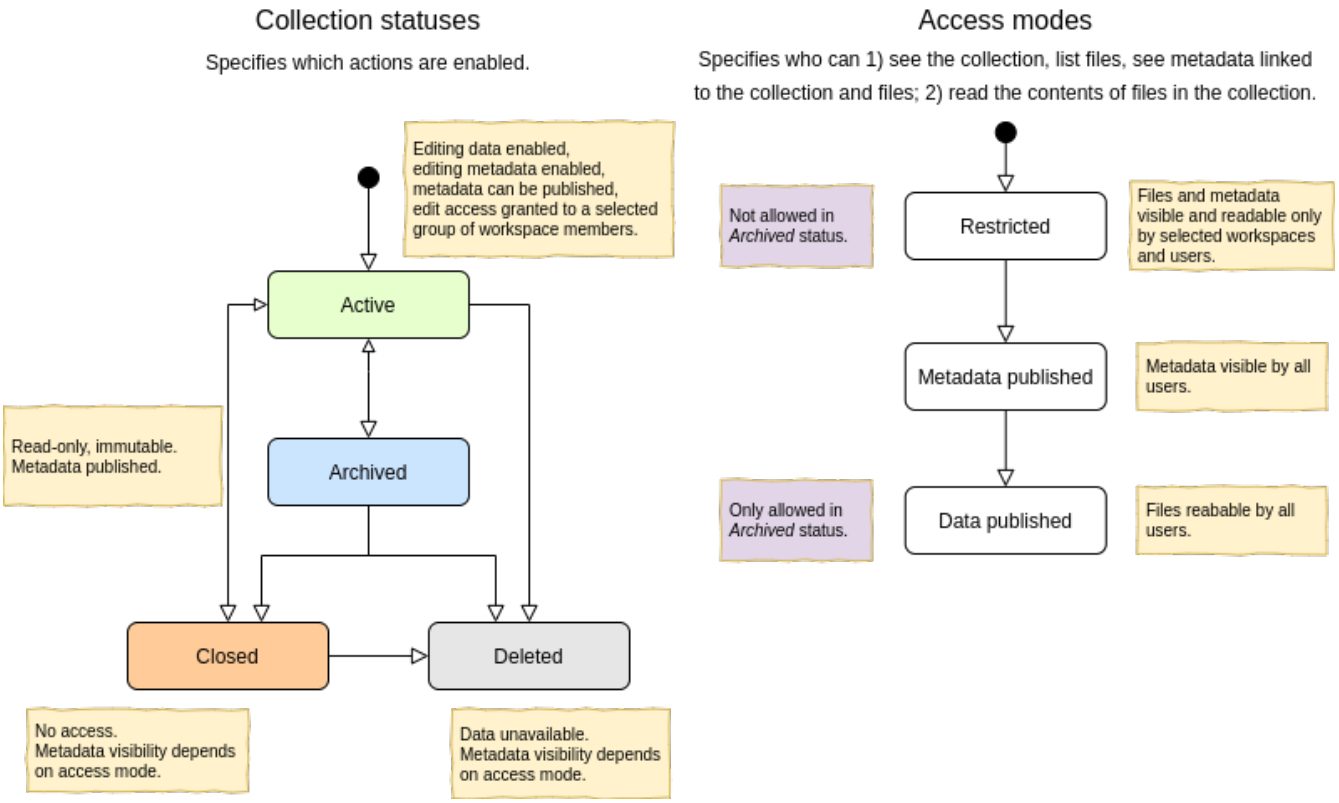
- *Active*: for the phase of data collection, data production and data processing;
- *Archived*: for when the data set is complete and is available for reuse;
- *Closed*: for when the data set should not be available for reading, but still needs to be preserved;
- *Deleted*: for when the data set needs to be permanently made unavailable. This status is irreversible. There is one exception to this rule – for the sake of data loss prevention, in special cases, administrators can still undelete a collection that was already deleted.

In these different statuses, different actions on the data are enabled or disabled. Also, visibility of the data and linked metadata depends partly on the collection status. We also distinguish three access modes for reading and listing files in a collection (where listing also includes seeing the metadata):

- *Restricted*: only access to explicitly selected workspaces and users;
- *Metadata published*: the collection and its files are visible, metadata linked to them is visible for all users;
- *Data published*: the files in the collection are readable for all users. This mode is irreversible. There is one exception to this rule – there might be a special situation, resulting from, e.g., a legal reason, when a collection has to be unpublished. This action is available to administrators, but it is highly discouraged, since the collection (meta)data may already be referenced in other systems.

The statuses and access modes, and the transitions between them are shown in the following diagram.

Collection editing and publication workflow



Roles and permissions

We distinguish the following roles in the solution:

- *User*: regular users can only view their own workspaces and collections.
- *View public metadata*: the user can view public metadata, workspaces, collections and files;
- *View public data*: the user can read public files;
- *Admin*: can create workspaces, assign roles and permissions;
- *Add shared metadata*: can add, modify and delete shared metadata entities.

Most users should have the *View public data* role. Only when the shared metadata may contain sensitive information that should not be visible for some users, the public data and public metadata roles should be discarded for those users.

Workspaces are used to organise collections in a hierarchy. On workspace level there are two access levels:

- *Manager*: can edit workspace details, manage workspace access and manage access to all collections that belong to the workspace;
- *Member*: can create a collection in the workspace.

Access to collections and files is managed on collection level. We distinguish the following access levels on collections:

- *List*: see collection, directory and file names and metadata properties/relations (only applicable for collections shared via the *Metadata published* access mode);
- *Read*: read file contents;
- *Write*: add files, add new file versions, mark files as deleted;
- *Manage*: grant, revoke access to the collection, change collection status and modes.

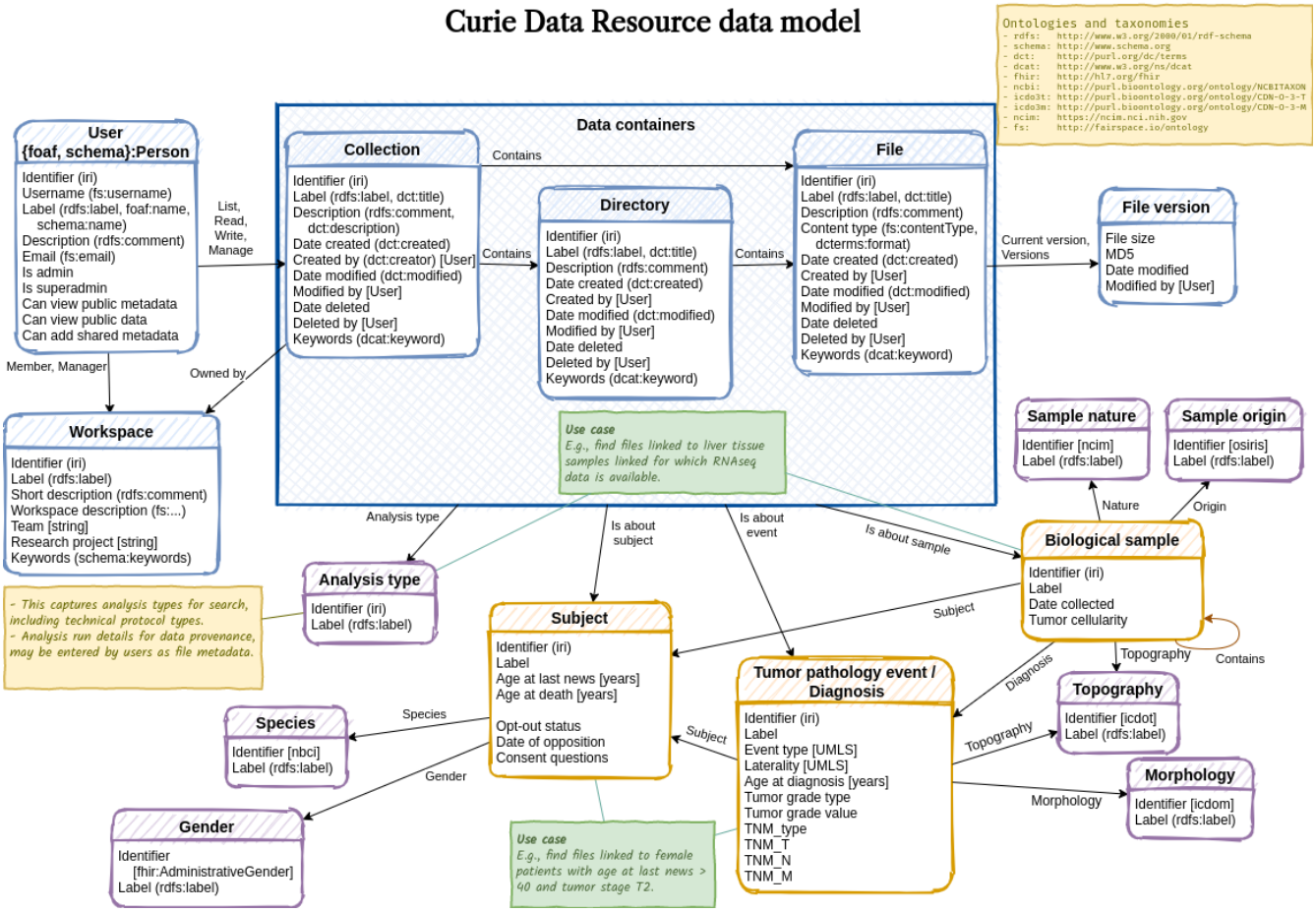
Access levels are hierarchical: the *Read* level includes the *List* level; the *Edit* level includes *Read* level; the *Manage* level includes *Edit* and *Read* level access. The user that creates the collection gets *Manage* access.

Data model and view configuration

Data model defined using the Shapes Constraint Language (SHACL).

- System data model: system-vocabulary.ttl
- Customisable data model: vocabulary.ttl
- Taxonomies: taxonomies.ttl

A schematic overview of the default data model in vocabulary.ttl:



Terminologies as types, entities as types.

Example taxonomy types and entity type:

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dash: <http://datashapes.org/dash#> .
@prefix fs: <https://fairspace.nl/ontology#> .
@prefix example: <https://example.com/ontology#> .

example:Gender a rdfs:Class, sh:NodeShape ;
    sh:closed false ;
    sh:description "The gender of the subject." ;
    sh:name "Gender" ;
    sh:ignoredProperties ( rdf:type owl:sameAs ) ;
    sh:property
    [
        sh:name "Label" ;
        sh:description "Unique gender label." ;
        sh:datatype xsd:string ;
        sh:maxCount 1 ;
        dash:singleLine true ;
        fs:importantProperty true ;
```

```
        sh:path rdfs:label
    ] .

example:Species a rdfs:Class, sh:NodeShape ;
    sh:closed false ;
    sh:description "The species of the subject." ;
    sh:name "Species" ;
    sh:ignoredProperties ( rdf:type owl:sameAs ) ;
    sh:property
    [
        sh:name "Label" ;
        sh:description "Unique species label." ;
        sh:datatype xsd:string ;
        sh:maxCount 1 ;
        dash:singleLine true ;
        fs:importantProperty true ;
        sh:path rdfs:label
    ] .

example:isOfGender a rdf:Property .
example:isOfSpecies a rdf:Property .

example:Subject a rdfs:Class, sh:NodeShape ;
    sh:closed false ;
    sh:description "A subject of research." ;
    sh:name "Subject" ;
    sh:ignoredProperties ( rdf:type owl:sameAs ) ;
    sh:property
    [
        sh:name "Gender" ;
        sh:description "The gender of the subject." ;
        sh:maxCount 1 ;
        sh:class example:Gender ;
        sh:path example:isOfGender
    ],
    [
        sh:name "Species" ;
        sh:description "The species of the subject." ;
        sh:maxCount 1 ;
        sh:class example:Species ;
        sh:path example:isOfSpecies
    ] .

example:aboutSubject a rdf:Property .

# Augmented system class shapes
fs:File sh:property
    [
        sh:name "Is about subject" ;
        sh:description "Subjects that are featured in this collection." ;
        sh:class example:Subject ;
```

```
        sh:path example:aboutSubject
    ] .
```

Example taxonomy:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example: <https://example.com/ontology#> .
@prefix gender: <http://hl7.org/fhir/administrative-gender#> .
@prefix ncbitaxon: <https://bioportal.bioontology.org/ontologies/NCBITAXON/> .

gender:male a example:Gender ;
  rdfs:label "Male" .
gender:female a example:Gender ;
  rdfs:label "Female" .

ncbitaxon:562 a example:Species ;
  rdfs:label "Escherichia coli" .
ncbitaxon:1423 a example:Species ;
  rdfs:label "Bacillus subtilis" .
ncbitaxon:4896 a example:Species ;
  rdfs:label "Schizosaccharomyces pombe" .
ncbitaxon:4932 a example:Species ;
  rdfs:label "Saccharomyces cerevisiae" .
ncbitaxon:6239 a example:Species ;
  rdfs:label "Caenorhabditis elegans" .
ncbitaxon:7227 a example:Species ;
  rdfs:label "Drosophila melanogaster" .
ncbitaxon:7955 a example:Species ;
  rdfs:label "Zebrafish" .
ncbitaxon:8355 a example:Species ;
  rdfs:label "Xenopus laevis" .
ncbitaxon:9606 a example:Species ;
  rdfs:label "Homo sapiens" .
ncbitaxon:10090 a example:Species ;
  rdfs:label "Mus musculus" .
```

Installation and configuration

Local development

Requires:

- yarn
- docker
- Java 15

To run the development version, checkout this repository, navigate to `projects/mercury` and run

```
yarn dev
```

If on MacOS, configure docker logging.... TODO As env variable, or in `.env` file: `DOCKER_LOGGING_DRIVER=json-file`.

This will start a Keycloak instance for authentication at port `5100`, the backend application named Saturn at port `8080` and the user interface at port `3000`.

At first run, you need to configure the service account in Keycloak.

- Navigate to http://localhost:5100
- Login with credentials `keycloak`, `keycloak`
- Grant `realm-management` roles in the Fairspace realm: `view-realm`, `manage-realm`, `manage-authorization`, `manage-users`.

Now everything should be ready to start using Fairspace:

- Navigate to http://localhost:3000 to open the application.
- Login with one of the following credentials:

Username	Password
organisation-admin	fairspace123
user	fairspace123

Kubernetes and helm

You can deploy Fairspace on a Kubernetes cluster using Helm. Helm charts for Fairspace are published to the public helm repository at https://storage.googleapis.com/fairspace-helm.

Instructions for deploying to Google Cloud

Download and install helm and gcloud

- Download `helm 2.14.3` from from https://github.com/helm/helm/releases/tag/v2.14.3
- Extract the downloaded archive to `~/bin/helm` and check with:

```
~/bin/helm/helm version
```

- Install kubectl.
- Download and install the Google Cloud SDK (requires Python).
- Obtain credentials for Kubernetes:

```
gcloud container clusters get-credentials <cluster id> --zone europe-west1-b
```

Use `fairspacecicluster` as cluster id for the CI environment. Ensure that your Google account has access to the `fairspace-207108` GCP project and log in using

```
gcloud auth login
```

- Check if all tools are correctly installed:

```
# List available clusters
gcloud container clusters list
# List Kubernetes namespaces
kubectl get ns
# List helm releases (deployments)
~/bin/helm/helm list
```

Initialise helm and add fairspace repository

```
# Initialise helm
~/bin/helm/helm init --client-only --stable-repo-url https://charts.helm.sh/stable
# Add the fairspace repo for reading
~/bin/helm/helm repo add fairspace https://storage.googleapis.com/fairspace-helm
# (Optional) Add the fairspace via the GCS plugin for writing
~/bin/helm/helm plugin install https://github.com/hayorov/helm-gcs.git --version 0.2.2
gcloud iam service-accounts keys create credentials.json --iam-account fairspace-
207108@appspot.gserviceaccount.com
export GOOGLE_APPLICATION_CREDENTIALS=/path/to/credentials.json
~/bin/helm/helm repo add fairspace-gcs gs://fairspace-helm
```

Fetch chart

```
# Update repo
~/bin/helm/helm repo update
# Fetch the fairspace chart
~/bin/helm/helm fetch fairspace/fairspace --version 0.7.5
```

Deploy Fairspace

Create a new Kubernetes namespace:

```
kubectl create namespace fairspace-new
```

Create a new deployment (called *release* in helm terminology) and install the Fairspace chart:

```
~/bin/helm/helm install fairspace/fairspace --version 0.7.5 --name fairspace-new
--namespace=fairspace-new \
-f /path/to/values.yaml --set-file saturn.vocabulary=/path/to/vocabulary.ttl --set
-file saturn.views=/path/to/views.yaml
```

You can pass values files with `-f` and provide a file for a specified value with `--set-file`.

Update an existing deployment

To update a deployment using a new chart:

```
~/bin/helm/helm upgrade fairspace-new fairspace-0.7.5.tgz
```

With `helm upgrade` you can also pass new values files with `-f` and pass files with `--set-file` as for `helm install`.

Clean up deployment

To clean up an environment or completely reinstall an environment, you can use `helm del`. :warning: Be careful, you may lose data!

```
~/bin/helm/helm del --purge fairspace-test
```

Design

Storage

RDF database using Apache Jena for:

- File metadata
- Permissions
- User metadata

File system data stored as blocks on the file system in append-only fashion.

License

...