



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 that 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 files 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 checks metadata consistency and validity upon data entry.
- FairSPACE allows organisations to **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 use of standard vocabularies, contributing to **interoperability** of data.

Fairspace

Usage	4
User interface	4
Login	4
Workspaces	4
Collections	5
Metadata	9
Interfaces for accessing and querying data (API)	9
Authentication	10
WebDAV	13
SPARQL	22
RDF metadata	24
Metadata views	28
Text search	30
Workspace management	31
Users and permissions	33
Configuration endpoints	35
External file system integration	36
Access policy	36
API specification and supported parameters	36
Text search in external storages	37
Configuration	37
Structure and terminology	39
Workflow and access modes	40
Roles and permissions	41
Data model and view configuration	43
Metadata	43
Types of metadata	43
Purpose	43
Data model	44
Data domains	44
Data model configuration	45
Limitations	49
Controlled vocabularies	49
View configuration	50
Installation and configuration	51
Local development	51
Kubernetes and helm	52
Instructions for deploying to Google Cloud	52

Logging	54
Configuration	54
Audit log	54
Design	57
Storage	57
License	58

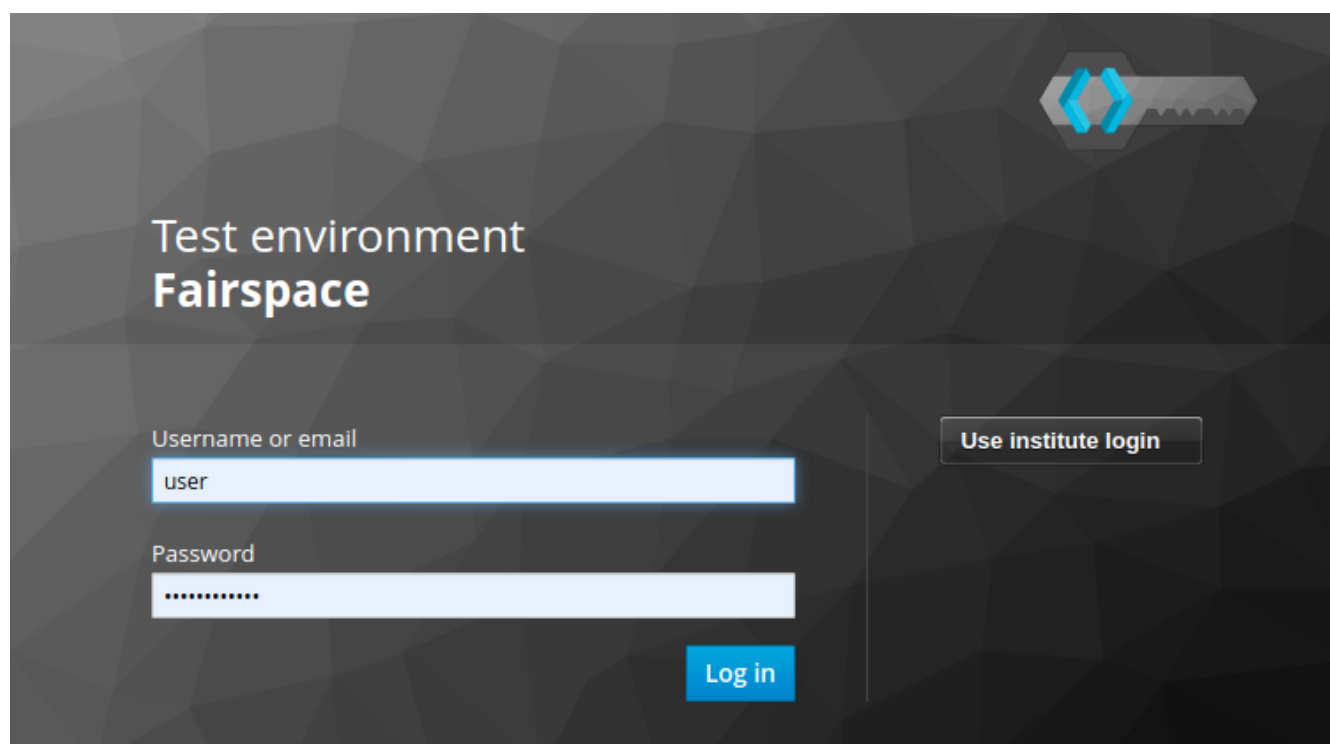
Usage

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:

The image shows a login page for 'Test environment Fairspace'. The background is dark with a geometric pattern. In the top right corner, there is a logo consisting of a blue double arrow pointing right, followed by a grey key icon. The main heading 'Test environment Fairspace' is in white. Below the heading, there are two input fields: 'Username or email' with the text 'user' and 'Password' with masked characters '.....'. To the right of these fields is a button labeled 'Use institute login'. Below the password field is a blue 'Log in' button.

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.

Name ↑	Collections	Members	Managers
Fairspace development team Scripts and data of the Fairspace development team	3	2	First Organisation Admin
Test Test workspace	8	3	First Organisation Admin

Rows per page: 10 ▾ 1-2 of 2 < >

Fairspace 0.7.4-SNAPSHOT

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

Fairspace development team

Short description
Scripts and data of the Fairspace development team

Keywords
dev

Workspaces / Fairspace development team

OVERVIEW USERS COLLECTIONS

Fairspace 0.7.4-SNAPSHOT

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.

Users with *manage* access can share collections with other users or workspaces, and change the default access mode for workspace members. Collection managers can also change the status of the collection (*Active*, *Archived* or *Closed*), change the view mode (*Restricted*, *Metadata published* or *Data published*—only available in the *Archived* status), delete the collection, or transfer ownership of the collection to another workspace.

The screenshot displays the FairSpace web interface. At the top, a blue header bar contains the 'YGRITTE' logo. Below the header, the breadcrumb navigation shows 'Collections / Performance tests / metrics / December 2020'. A search bar and a 'Show deleted' toggle are also present. The main content area is divided into two panels. The left panel shows a table with one file, 'performance_metrics.csv', which is 1.04 kB and was last modified at 19:03. The right panel displays the metadata for this file, including the collection name 'Performance tests', the owner 'FairSpace development team', and the status 'Active'. It also shows a list of metadata fields: 'Write access', 'Metadata for Performance tests', 'Metadata for metrics', 'Metadata for metrics/December 2020', and 'Metadata for metrics/December 2020/performance_metrics.csv'. The bottom of the interface features a blue footer bar with the text 'FairSpace 0.7.4-SNAPSHOT'.

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:

Metadata for dir_1/samples.csv



samples.csv

Description

Measurement data

Keywords

2021

lab



Is about biological sample

ab2b



Sample f862313b-ab2b-4d19-bd0b-6a7c2c07c759

Sample 4b83486d-fdfd-43d4-ab2b-cb2728ebdee7



Type of analysis



Content type

text/csv

Created

10:23

Created by

Ygritte

UPDATE CANCEL

The shared metadata entities and controlled vocabularies cannot be added via the user interface. The [RDF metadata](#) 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.

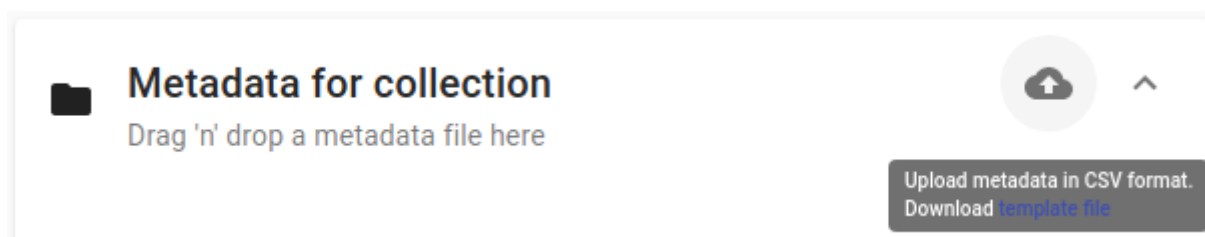
The file should have a header row containing the names of the columns. The mandatory `Path` column is used for the file path. For the property columns, the name should match exactly the name of the property in the database.

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,,
```

This specifies the table:

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.

The screenshot displays the FairSpace 0.7.4-SNAPSHOT web interface. The main content area shows a table of samples under the 'SAMPLES' tab. The table columns are Sample, Tumor cellularity, Sample nature, and Gender. The selected sample is 'ff40f912-20e0-4c66-969e-f8cfd115287', which is a Frozen Specimen with a cellularity of 38. The left sidebar contains filters for Gender, Tumor topography (with 'Stomach, NOS' selected), Morphology, Laterality, Event type, and Age at diagnosis. The right sidebar shows detailed metadata for the selected sample, including diagnosis, sample nature, subject, and topography.

Interfaces for accessing and querying data (API)

The data in FairSpace can be accessed via Application Programming Interfaces (APIs). The user interfaces application uses those APIs, but also other programs can use them, e.g., for automated data uploading or for exporting data for further processing or for synchronisation with other systems.

Authentication

All API endpoints require authentication via an authorisation header. To enable WebDAV clients to connect to Fairspace, also so-called *Basic authentication* is supported.

For secure authentication, it is strongly advised to use the *OpenID Connect (OIDC) / OAuth2* workflow. The user interface application also uses this workflow.

When using the APIs in automated scripts, ensure that an account is used with only the required privileges (conform the *principle of least privilege*). I.e., when an admin account is not needed, use a non-admin account. For adding shared metadata, an account with *Add shared metadata* role is required, see [Uploading metadata](#).

When an action is done on behalf of a specific user, do not use a service account or system account for the action directly, but obtain a token for that user first, e.g., by using the [impersonation](#) feature of Keycloak. That way the audit logging still captures which user did what.

OpenID Connect (OIDC) / OAuth2 workflow

Fairspace supports OpenID Connect authentication via Keycloak. The workflow for API access is roughly as follows.

- The client authenticates with the token endpoint of the identity provider (Keycloak) and obtains a signed access token
- The client uses the access token in the request header when connecting to the Fairspace API
- Fairspace receives the request with the access token and validates if the token is valid, using the public key of the identity provider.

The token endpoint of Keycloak supports refreshing the token if it is close to expiry. However, checking the token expiration and refreshing make the authentication logic quite complex.

You can either obtain a fresh token before every API request or use an existing library that implements the authentication workflow. For finding available client-side libraries, check the [Securing applications and services guide](#) of Keycloak.

For use in scripts, it is advised to obtain a token for offline access, using the [Offline access](#) feature of OpenID Connect.

```
import logging
import os
import requests

log = logging.getLogger()

def fetch_access_token(keycloak_url: str = os.environ.get('KEYCLOAK_URL'),
                      realm: str = os.environ.get('KEYCLOAK_REALM'),
                      client_id: str = os.environ.get('KEYCLOAK_CLIENT_ID'),
                      client_secret: str = os.environ.get('KEYCLOAK_CLIENT_SECRET'),
                      username: str = os.environ.get('KEYCLOAK_USERNAME'),
                      password: str = os.environ.get('KEYCLOAK_PASSWORD')) -> 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())
        raise Exception('Error fetching token.')
    data = response.json()
    token = data['access_token']
    log.info(f"Token obtained successfully. It will expire in {data['expires_in']} seconds")
    return token

def auth():
    return f'Bearer {fetch_access_token()}'
```

Code to obtain the OpenID Connect authorisation header (bash, curl)

Requires the [jq](#) JSON parser.

```
fetch_access_token() {  
  curl -s \  
    -d "client_id=${KEYCLOAK_CLIENT_ID}" \  
    -d "client_secret=${KEYCLOAK_CLIENT_SECRET}" \  
    -d "username=${KEYCLOAK_USERNAME}" \  
    -d "password=${KEYCLOAK_PASSWORD}" \  
    -d 'grant_type=password' \  
    "${KEYCLOAK_SERVER_URL}/auth/realms/${KEYCLOAK_REALM}/protocol/openid-connect/token" | jq -r  
    '.access_token'  
}  
ACCESS_TOKEN=$(fetch_access_token)
```

Basic authentication

For WebDAV client access and for a simpler authentication method during testing, Fairspace also supports *Basic authentication*, which means that the [base64](#) encoded `username:password` string is sent in the `Authorization` header together with a prefix `Basic`.

This authentication method is considered to be less secure than token based authentication, because it requires scripts to have a plain text password stored somewhere. Also, users may have to retype their passwords when logging in, tempting them to choose less secure, easier to remember, passwords.

Code to generate the Basic authorisation header (Python)

```
import base64  
import os  
  
def auth():  
    username = os.environ.get('KEYCLOAK_USERNAME')  
    password = os.environ.get('KEYCLOAK_PASSWORD')  
    return f"Basic {base64.b64encode(f'{username}:{password}'.encode()).decode()}"
```

Code to generate the Basic authorisation header (bash)

```
AUTH_HEADER="Basic $(echo -n "${KEYCLOAK_USERNAME}:${KEYCLOAK_PASSWORD}" | base64)"
```

Examples

In the examples in this documentation, we assume one of both methods to be available.

This means for the Python examples that a function `auth()` should be implemented that returns the authorisation header value, see the examples above.

```
import os
from requests import Response, Session

def auth():
    """ Returns authorisation header
    Replace this with an implementation from one of the sections above.
    """
    pass

server_url = os.environ.get('FAIRSPACE_URL')
headers = {
    'Authorization': auth()
}
response = Session().get(f'{server_url}/api/users/current', headers=headers)
if not response.ok:
    raise Exception(f"Error fetching current user: {response.status_code} {response.reason}")
print(response.json())
```

For examples using curl, an authorisation header needs to be passed using the **-H** option.

For Basic authentication:

```
AUTH_HEADER="Basic $(echo -n "${KEYCLOAK_USERNAME}:${KEYCLOAK_PASSWORD}" | base64)"
curl -i -H "Authorization: ${AUTH_HEADER}" "${FAIRSPACE_URL}/api/users/current"
```

For OpenID Connect:

```
# ACCESS_TOKEN=...
AUTH_HEADER="Bearer ${ACCESS_TOKEN}"
curl -i -H "Authorization: ${AUTH_HEADER}" "${FAIRSPACE_URL}/api/users/current"
```

Automatic authentication in Jupyter Hub

In Jupyter Hub, users are automatically authenticated and can directly connect to the local API address without adding authentication headers.

WebDAV

A file storage API is exposed via the WebDAV protocol for accessing the file system via the web. It runs on **/api/webdav/**.

This endpoint can be used by many file explorers, including Windows Explorer, and by tools like [FileZilla](#) and [Cyberduck](#). Use <https://fairspace.example.com/api/webdav/> or <davs://fairspace.example.com/api/webdav/> as location, with [fairspace.example.com](#) replaced by the server name.

All visible collections in the system are exposed as top-level directories. Creating a top-level directory via WebDAV will result in an error message, see [Create collection or directory](#).

The [Web-based Distributed Authoring and Versioning \(WebDAV\)](#) protocol allows users to operate on collections and files. Fairspace exposes a WebDAV API for accessing the file systems, while

restricting access to only the files accessible by the user.

The WebDAV API allows to upload and download files and to perform standard file operations such as copying or moving, as well as custom operations, such as collection lifecycle management and advanced data loss prevention features such as versioning and undeletion.

Directory listing and path properties

PROPFIND /api/webdav/{path}	
Request headers:	
Depth	When 0 only the information about the path is returned, when 1 the contents of the directory is returned, if the path is a directory.
Show-Deleted	Include deleted paths when the value is on . (Optional)
Version	Specify a version number to request properties of a specific file version. The first version has number 1 . If not specific, the current version is returned.
With-Metadata-Links	Include list of metadata entities that are linked to the resource, when value true .
Request body:	
To include also custom FairSPACE attributes in the response, like the collection description, send the following request body: <propfind><allprop /></propfind>	

Code examples

Check if path exists (Python)

```
import logging
import os
from requests import Request, Response, Session

log = logging.getLogger()

server_url = os.environ.get('FAIRSPACE_URL')

def exists(path):
    """ Check if a path exists
    """
    headers = {
        'Depth': '0',
        'Authorization': auth()
    }
    session = Session()
    req = Request('PROPFIND', f'{server_url}/api/webdav/{path}/', headers=headers, cookies=session.cookies)
    response: Response = session.send(req.prepare())
    return response.ok
```

Fetch directory listing (Python)

```
import logging
import os
from requests import Request, Response, Session
from xml.etree.ElementTree import fromstring

log = logging.getLogger()

server_url = os.environ.get('FAIRSPACE_URL')

def ls(path: str):
    """ List contents of path
    """
    headers = {
        'Depth': '1',
        'Authorization': auth()
    }
    session = Session()
    req = Request('PROPFIND', f'{server_url}/api/webdav/{path}', headers=headers, cookies=session.cookies)
    response: Response = session.send(req.prepare())
    if not response.ok:
        raise Exception(f"Error fetching directory '{path}': {response.status_code} {response.reason}")
    tree = fromstring(response.content.decode())
    for item in tree.findall('{DAV:}response'):
        print(item.find('{DAV:}href').text)
```

Fetch directory listing (curl)

Requires the [xmlstarlet](#) tool.

```
curl -s -H "Authorization: ${AUTH_HEADER}" -X PROPFIND -H "Depth: 1" "${FAIRSPACE_URL}/api/webdav/
${path}" -d '<propfind><allprop /></propfind>' \
| xmlstarlet sel -T -t -m d:multistatus/d:response -v d:propstat/d:prop/d:displayname -n
```

Example response

Example PROPFIND response

Example response using PROPFIND on the root location <https://fairspace.ci.fairway.app/api/webdav> with Depth: 1 and request body `<propfind><allprop /></propfind>`. Adding the `<allprop />` in the request results in custom Fairspace properties, like the description (`ns1:comment`), to be included in the WebDAV response.

```

<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:ns1="https://fairspace.nl/ontology#" xmlns:d="DAV:">
  <d:response>
    <d:href>/api/webdav/</d:href>
    <d:propstat>
      <d:prop>
        <d:getcontenttype></d:getcontenttype>
        <d:getetag>"https://fairspace.ci.fairway.app/api/webdav"</d:getetag>
        <d:iscollection>TRUE</d:iscollection>
        <d:displayname></d:displayname>
        <d:isreadonly>TRUE</d:isreadonly>
        <d:name></d:name>
        <d:supported-report-set></d:supported-report-set>
        <d:resourcetype>
          <d:collection/>
        </d:resourcetype>
      </d:prop>
      <d:status>HTTP/1.1 200 OK</d:status>
    </d:propstat>
  </d:response>
  <d:response>
    <d:href>/api/webdav/Demonstration/</d:href>
    <d:propstat>
      <d:prop>
        <ns1:access>Write</ns1:access>
        <ns1:canRead>TRUE</ns1:canRead>
        <ns1:userPermissions>http://fairspace.ci.fairway.app/iri/user-iri Manage
        </ns1:userPermissions>
        <ns1:accessMode>Restricted</ns1:accessMode>
        <ns1:availableStatuses>Active</ns1:availableStatuses>
        <ns1:canDelete>FALSE</ns1:canDelete>
        <ns1:iri>https://fairspace.ci.fairway.app/api/webdav/Demonstration</ns1:iri>
        <ns1:canWrite>TRUE</ns1:canWrite>
        <ns1:ownedByName>Demo</ns1:ownedByName>
        <ns1:canManage>FALSE</ns1:canManage>
        <ns1:canUndelete>FALSE</ns1:canUndelete>
        <ns1:workspacePermissions>http://fairspace.ci.fairway.app/iri/workspace-iri
          Write
        </ns1:workspacePermissions>
        <ns1:createdBy>http://fairspace.ci.fairway.app/iri/user-iri</ns1:createdBy>
        <ns1:comment>Demonstration collection</ns1:comment>
        <ns1:availableAccessModes>Restricted</ns1:availableAccessModes>
        <ns1:ownedBy>http://fairspace.ci.fairway.app/iri/workspace-iri</ns1:ownedBy>
        <ns1:status>Active</ns1:status>
        <d:getcreated>2021-02-02T12:12:33Z</d:getcreated>
        <d:creationdate>2021-02-02T12:12:33Z</d:creationdate>
        <d:getcontenttype>text/html</d:getcontenttype>
        <d:getetag>"https://fairspace.ci.fairway.app/api/webdav/Demonstration"</d:getetag>
        <d:iscollection>TRUE</d:iscollection>
        <d:displayname>Demonstration collection</d:displayname>
        <d:isreadonly>FALSE</d:isreadonly>
        <d:name>Demonstration collection</d:name>
        <d:supported-report-set></d:supported-report-set>
        <d:resourcetype>
          <d:collection/>
        </d:resourcetype>
      </d:prop>
      <d:status>HTTP/1.1 200 OK</d:status>
    </d:propstat>
  </d:response>
</d:multistatus>

```

Create collection or directory

MKCOL /api/webdav/{path}

Create collection or directory

Request headers:

Owner

Specify the identifier of the owner workspace when creating a collection.

Example create collection or directory (Python)

```
import logging
import os
from requests import Request, Response, Session

log = logging.getLogger()

server_url = os.environ.get('FAIRSPACE_URL')

def mkdir(path: str, workspace_iri: str=None):
    # Create directory
    headers = {
        'Authorization': auth()
    }
    if workspace_iri is not None:
        headers['Owner'] = workspace_iri
    req = Request('MKCOL', f'{server_url}/api/webdav/{path}/', headers=headers, cookies=self.session().cookies)
    response: Response = Session().send(req.prepare())
    if not response.ok:
        raise Exception(f"Error creating directory '{path}': {response.status_code} {response.reason}")
```

Example create collection or directory (curl)

```
# Create a new collection, owned by workspace WORKSPACE_IRI
NEW_COLLECTION=New collection
WORKSPACE_IRI=http://fairspace.ci.fairway.app/iri/workspace-iri
curl -i -H "Authorization: ${AUTH_HEADER}" -X MKCOL -H "Owner: ${WORKSPACE_IRI}" "${FAIRSPACE_URL}/api/webdav/${NEW_COLLECTION}"
# Create a new directory in the newly created collection
curl -i -H "Authorization: ${AUTH_HEADER}" -X MKCOL "${FAIRSPACE_URL}/api/webdav/${NEW_COLLECTION}/Test directory"
```

Upload files

POST /api/webdav/{path}

action=upload_files

Request data:

action

upload_files

files	Send files with the target file names as keys, see the examples below.
-------	--

Example uploading files (Python)

```
import logging
import os
from requests import Response, Session

log = logging.getLogger()

server_url = os.environ.get('FAIRSPACE_URL')

def upload_files(path: str, files: Dict[str, any]):
    # Upload files
    response: Response = Session().post(f'{server_url}/api/webdav/{path}/',
                                         headers={'Authorization': auth()},
                                         data={'action': 'upload_files'},
                                         files=files)
    if not response.ok:
        raise Exception(f"Error uploading files into '{path}': {response.status_code} {response.reason}")
```

Example uploading files (curl)

```
# Upload files 'coffee.jpg' and 'coffee 2.jpg' to a collection
path="new collection"
curl -i -H "Authorization: ${AUTH_HEADER}" -X POST -F 'action=upload_files' -F
'coffee.jpg=@coffee.jpg' -F 'coffee 2.jpg=@coffee 2.jpg' "${FAIRSPACE_URL}/api/webdav/${path}"
```

Copy and move a directory or file

COPY /api/webdav/{path}	
Copy a directory or file.	
Request headers:	
Destination	The destination path relative to the server, URL encoded, e.g., /api/webdav/collection%20abc/test.txt.

Example copy path (curl)

```
# Copy 'Examples/Test dir/test 1.txt' to 'Examples/Test dir/test 2.txt'
path="Examples/Test dir/test 1.txt"
target="/api/webdav/Examples/Test%20dir/test%202.txt"
curl -i -H "Authorization: ${AUTH_HEADER}" -X COPY -H "Destination: ${target}" "${FAIRSPACE_URL}/api/webdav/${path}"
```

MOVE /api/webdav/{path}

Move or rename a directory or file

Request headers:

Destination

The destination path relative to the server, URL encoded, e.g.,
/api/webdav/collection%20abc/test.txt.

Example move path (curl)

```
# Move 'Examples/Test dir/test 1.txt' to 'Examples/Test dir/test 2.txt'
path="Examples/Test dir/test 1.txt"
target="/api/webdav/Examples/Test%20dir/test%202.txt"
curl -i -H "Authorization: ${AUTH_HEADER}" -X MOVE -H "Destination: ${target}" "${FAIRSPACE_URL}
/api/webdav/${path}"
```

Undelete a directory or file

POST /api/webdav/{path}

action=undelete

Undelete a directory or file

Request headers:

Show-deleted

on

Request data:

action

undelete

Example undelete path (curl)

```
curl -i -H "Authorization: ${AUTH_HEADER}" -X POST -F "action=undelete" "${FAIRSPACE_URL}
/api/webdav/${path}"
```

Revert to a file version

POST /api/webdav/{path}

action=revert

Restore a previous file version

Request data:

action

revert

version

The version number to restore.

Example revert file version (curl)

```
curl -i -H "Authorization: ${AUTH_HEADER}" -X POST -F "action=revert" -F "version=${version}" "${FAIRSPACE_URL}/api/webdav/${path}"
```

Other collection actions

On collections, a number of actions is available. These are not documented here in detail, but can be used from the user interface instead.

Action	Description
set_access_mode	Change the access mode of a collection.
set_status	Change the status of a collection.
set_permission	Change the permission of the specified user or workspace on a collection.
set_owned_by	Transfer ownership of a collection to another workspace.
unpublish	Unpublish a published collection.

SPARQL

The [SPARQL](#) API is a standard API for querying RDF databases. This endpoint is read-only and can be used for advanced search, analytics, data extraction, etc.

POST /api/rdf/query
Execute SPARQL query
<i>Request body:</i>
The SPARQL query.

Example SPARQL query (Python)

Query for the first 500 samples.

```
import logging
import os
from requests import Response, Session

log = logging.getLogger()

server_url = os.environ.get('FAIRSPACE_URL')

def query_sparql(query: str):
    headers = {
        'Authorization': auth(),
        'Content-Type': 'application/sparql-query',
        'Accept': 'application/json'
    }
    response: Response = Session().post(f"{server_url}/api/rdf/query", data=query, headers=headers)
    if not response.ok:
        raise Exception(f'Error querying metadata: {response.status_code} {response.reason}')
    return response.json()

query_sparql("""
PREFIX example: <https://example.com/ontology#>
PREFIX fs: <https://fairspace.nl/ontology#>

SELECT DISTINCT ?sample
WHERE {
    ?sample a example:BiologicalSample .
    FILTER NOT EXISTS { ?sample fs:dateDeleted ?anyDateDeleted }
}
# ORDER BY ?sample
LIMIT 500
""")
```

Example SPARQL query (curl)

Query for the first 500 samples.

```
curl -X POST -H "Authorization: ${AUTH_HEADER}" -H 'Content-Type: application/sparql-query' -H
'Accept: application/json' \
-d "
PREFIX example: <https://example.com/ontology#>
PREFIX fs: <https://fairspace.nl/ontology#>

SELECT DISTINCT ?sample
WHERE {
    ?sample a example:BiologicalSample .
    FILTER NOT EXISTS { ?sample fs:dateDeleted ?anyDateDeleted }
}
# ORDER BY ?sample
LIMIT 500
" \
"${FAIRSPACE_URL}/api/rdf/query"
```

RDF metadata

For reading and writing metadata to the database, the [/api/metadata](#) endpoint supports a number of operations:

- **GET**: Retrieve metadata for a specified subject, predicate or object.
- **PUT**: Add metadata
- **PATCH**: Update metadata
- **DELETE**: Delete specified triples or all metadata linked to a subject.

The metadata is stored as subject-predicate-object triples. The API supports several serialisation formats for sending :

- **Turtle** ([text/turtle](#))
- **JSON-LD** ([application/ld+json](#), [JSON schema](#))
- **N-Triples** ([application/n-triples](#))

After any update, the metadata must be consistent with the data model, see [Data model and view configuration](#). If an update would violate the data model constraints, the request is rejected with a status **400** response, with a message indicating the violation.

Uploading metadata

Shared metadata entities will in most cases come from other systems and will be added to FairSPACE exclusively by an ETL process which will extract data from the laboratory and clinical systems, perform pseudonymization of identifiers, convert the metadata to some RDF-native format conforming the data model and send them to FairSPACE.

FairSPACE will validate the uploaded metadata against the constraints defined in the data model and returns a detailed error message in case of violations. The validations include all the necessary type checks, referential consistency (validity of identifiers) checks, validation of mandatory fields, etc. If any entity violates the constraints, the entire bulk upload will be rejected.

The ETL process will use a special technical account with the *Add shared metadata* role. Regular users will not be able to add or modify shared metadata entities. Regular users can link files to shared metadata entities, see [Metadata forms](#) and [Metadata upload](#).

In addition to the main ETL workflow, data managers need a possibility to add or modify certain properties of top-level metadata entities. This can be done using the RDF-based metadata API.

A number of guidelines for uploading shared metadata:

- Entities must have a type, a globally unique identifier, and a unique label for the type.
It is advised to use a unique identifier from an existing reference system for this purpose.
- Because of the nature of linked data, it is advised to add shared metadata entities in an append-only fashion: only adding entities and avoid updating or deleting entities.
- By nature of RDF, metadata is typically added on the level of triples. E.g., when adding a property **dcat:keyword** to a file, this will add a key word to the (possibly) already existing list of

key words.

If you want to completely replace (or remove) a property from an entity, use the **PATCH** method instead of **PUT**.

Example metadata file in turtle format: **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 uploading metadata file using Python.

```
import logging
import os
from requests import Response, Session

log = logging.getLogger()

server_url = os.environ.get('FAIRSPACE_URL')

with open('testdata.ttl') as testdata:
    response: Response = Session().put(f"{server_url}/api/metadata/",
    data=testdata.read(),
    headers={
        'Authorization': auth(),
        'Content-type': 'text/turtle'
    })
    if not response.ok:
        raise Exception(f"Error uploading metadata: {response.status_code} {response.reason}")
```

Example uploading metadata file (curl).

```
curl -v -X PUT -H "Authorization: Basic $(echo -n "${KEYCLOAK_USERNAME}:${KEYCLOAK_PASSWORD}" |
base64)" \
-H "Content-type: text/turtle" --data @testdata.ttl "${FAIRSPACE_URL}/api/metadata/"
```

API specification

GET /api/metadata/

Retrieve metadata

Parameters:

subject	string	The subject to filter on.
predicate	string	The predicate to filter on, not required.
object	string	The object to filter on, not required.
includeObjectProperties	boolean	If set, the response will include several properties for the included objects. The properties to be included are marked with <code>fs:importantProperty</code> in the vocabulary.

Response:

Returns serialised triples matching the query parameters.

Example of retrieving metadata (curl)

Request metadata for the triple with subject 'a', predicate 'b' and object 'c'.

```
curl -G -H "Accept: application/ld+json" \  
--data-urlencode "subject=a" \  
--data-urlencode "predicate=b" \  
--data-urlencode "object=c" \  
--data-urlencode "withValueProperties=true" \  
"http://localhost:8080/api/metadata/"
```

PUT /api/metadata/

Add metadata. Existing metadata is left untouched. The data must be consistent with the data model after the update (see [Data model and view configuration](#)), otherwise **400** is returned. Only available for users with *Add shared metadata* role.

Request body:

Serialised RDF triples.

Example of adding metadata (curl)

```
curl -X PUT -H "Content-type: application/ld+json" -d \
'{
  "@graph" : [ {
    "@id" : "ws:7f0ed5bb-7a96-4715-8727-6b4b7f1facfd",
    "hasData" : "ws:4bcd1780-c9cd-4bdd-0000-1ce966ff9ac3"
  } ],
  "@context" : {
    "hasData" : {
      "@id" : "http://fairspace.ci.fairway.app/vocabulary/hasData",
      "@type" : "@id"
    },
    "ws" : "http://fairspace.ci.fairway.app/iri/"
  }
}' \
"http://localhost:8080/api/metadata/"
```

PATCH /api/metadata/

Update metadata. Any existing metadata for a given subject/predicate combination will be overwritten with the provided values. The data must be consistent with the data model after the update (see [Data model and view configuration](#)), otherwise 400 is returned. Only available for users with *Add shared metadata* role.

Request body:

Serialised RDF triples.

Example of updating metadata (curl)

```
curl -X PATCH -H "Content-type: application/ld+json" -d \
'{
  "@graph" : [ {
    "@id" : "ws:7f0ed5bb-7a96-4715-8727-6b4b7f1facfd",
    "hasData" : "ws:4bcd1780-c9cd-4bdd-0000-1ce966ff9ac3"
  } ],
  "@context" : {
    "hasData" : {
      "@id" : "http://fairspace.ci.fairway.app/vocabulary/hasData",
      "@type" : "@id"
    },
    "ws" : "http://fairspace.ci.fairway.app/iri/"
  }
}' \
"http://localhost:8080/api/metadata/"
```

DELETE /api/metadata/

Delete metadata. If a request body is provided, the triples specified in the body will be deleted. Otherwise, the subject specified in the subject parameter will be marked as deleted. Please note that the subject will still exist in the database. Only available for users with *Add shared metadata* role.

DELETE /api/metadata/

Parameters:

subject	string	The subject to filter on. (Optional)
---------	--------	---

Request body:

Serialised RDF triples. (Optional)

Example of deleting metadata (curl)

```
curl -X DELETE -H "Content-Type: application/ld+json" -d \
'{
  "@graph" : [ {
    "@id" : "ws:7f0ed5bb-7a96-4715-8727-6b4b7f1facfd",
    "hasData" : "ws:4bcd1780-c9cd-4bdd-0000-1ce966ff9ac3"
  } ],
  "@context" : {
    "hasData" : {
      "@id" : "http://fairspace.ci.fairway.app/vocabulary/hasData",
      "@type" : "@id"
    },
    "ws" : "http://fairspace.ci.fairway.app/iri/"
  }
}' \
"http://localhost:8080/api/metadata/"
```

Metadata views

Metadata views endpoint used for metadata-based search.

GET /api/views/

List all views with available columns per each view.

Example list view (curl)

```
curl -H "Accept: application/json" "http://localhost:8080/api/views/"
```

POST /api/views/

Fetch page of rows of a view matching the request filters.

Parameters:

view	string	Name of the view.
filters	List of filters, based on available facets and their values. Each filter has to contain a "field" property, matching the name of a facet, and list of values to filter on.	
page	integer	Requested page

POST /api/views/		
size	integer	Page size

Example fetching page of view rows (curl)

```
curl -X POST -H 'Content-type: application/json' -H 'Accept: application/json' -d \
'{
  "view": "Resource",
  "filters": [
    {
      "field": "Resource_type",
      "values": ["https://fairspace.nl/ontology#Collection"]
    }
  ],
  "page": 1,
  "size": 100
}' \
"http://localhost:8080/api/views/"
```

POST /api/views/count		
Count rows of a view matching request filters.		
<i>Parameters:</i>		
view	string	Name of the view.
filters	List of filters, based on available facets and their values. Each filter has to contain a "field" property, matching the name of a facet, and list of values to filter on.	

Example counting view rows (curl)

```
curl -X POST -H 'Content-type: application/json' -H 'Accept: application/json' -d \
'{
  "view": "Resource",
  "filters": [
    {
      "field": "Resource_type",
      "values": ["https://fairspace.nl/ontology#Collection"]
    }
  ]
}' \
'http://localhost:8080/api/views/count'
```

GET /api/views/facets		
List all facets with available values per each facet.		

Example retrieving facets with values (curl)

```
curl -H "Accept: application/json" "http://localhost:8080/api/views/facets"
```

Text search

Search endpoint used for text search on labels or comments.

POST /api/search/files

Find files, directories or collections based on a label or a comment.

Parameters:

query	string	Text fragment to search on.
parentIRI	string	IRI of the parent directory or collection to limit the search area.
Response		
Object in JSON format, with query and results properties. Results contain a list of files (and/or directories, collections) with the following properties:		
id	string	File (or directory) identifier (IRI).
label	string	File (or directory) name.
type	string	Type of the resource as defined in the vocabulary, e.g. "https://fairspace.nl/ontology#File", "https://fairspace.nl/ontology#Directory"
comment	string	File (or directory) description. Optional.

Example text search (curl)

```
curl -X POST -H 'Content-type: application/json' -H 'Accept: application/json' -d '{
  "query": "test folder",
  "parentIRI": "http://localhost:8080/api/webdav/dir1"
}' \
'http://localhost:8080/api/search/files'
```

Example text search response

```
{
  "results": [
    {
      "id": "https://fairspace.example.com/api/webdav/col1/test",
      "label": "test",
      "type": "https://fairspace.nl/ontology#File",
      "comment": "Description of the test file from col1."
    },
    {
      "id": "https://fairspace.example.com/api/webdav/col2/new_test_folder",
      "label": "new_test_folder",
      "type": "https://fairspace.nl/ontology#Directory",
      "comment": null
    }
  ],
  "query": "test"
}
```

POST /api/search/lookup

Metadata entities lookup search by entity labels or description.

Parameters:

query	string	Text fragment to search on.
resourceType	string	Type of the entity in request.

Example lookup search (curl)

```
curl -X POST -H 'Content-type: application/json' -H 'Accept: application/json' -d \
'{
  "query": "test",
  "resourceType": "https://example.com/ontology#TumorPathologyEvent"
}' \
'http://localhost:8080/api/search/lookup'
```

Workspace management

Operations on workspace entities.

GET /api/workspaces/

List all available workspaces.

Response contains the following data:

iri	Unique workspace IRI.
name	Unique workspace name.
comment	Workspace description in markdown format.

GET /api/workspaces/	
managers	List of workspace managers.
summary	Short summary on the workspace - how many collections and how many users it has.
canCollaborate	If a current user is added to the workspace as a collaborator.
canManage	If a current user is a workspace manager.

Example of listing available workspaces (curl)

```
curl -H "Accept: application/json" "http://localhost:8080/api/workspaces/"
```

PUT /api/workspaces/		
Add a workspace. Available only to administrators.		
<i>Parameters:</i>		
name	string	Unique workspace name.
<i>Response:</i>		
Response contains the workspace name and newly assigned IRI.		

Example of adding a workspace (curl)

```
curl -X PUT -H "Accept: application/json" -d '{"name": "test workspace"}'
"http://localhost:8080/api/workspaces/"
```

DELETE /api/workspaces/		
Delete a workspace. Available only to administrators.		
<i>Parameters:</i>		
workspace	string	Workspace IRI (URL-encoded).

Example of deleting a workspace (curl)

```
curl -X DELETE --data-urlencode "workspace=http://fairspace.com/iri/123"
"http://localhost:8080/api/workspaces/"
```

Workspace users

GET /api/workspaces/users	
List all workspace users with workspace roles.	

GET /api/workspaces/users

Parameters:

workspace	string	Workspace IRI (URL-encoded).
-----------	--------	------------------------------

Response:

Response contains list of workspace users with their workspace roles.

Example of listing workspace users (curl)

```
curl -H 'Accept: application/json'
'http://localhost:8080/api/workspaces/users?workspace=http://fairspace.com/iri/123'
```

PATCH /api/workspaces/users

Update workspace users and their workspace roles.

Parameters:

workspace	string	Workspace IRI (URL-encoded).
-----------	--------	------------------------------

Response:

Response contains list of workspace users with their updated workspace roles.

Example of updating workspace users (curl)

```
curl -H 'Accept: application/json' --data-urlencode "workspace=http://fairspace.com/iri/123"
"http://localhost:8080/api/workspaces/users"
```

Users and permissions

GET /api/users/

List all organisation users.

Response:

Returns list of users with user's unique ID, name, email, username and user's organisation-level permissions: if a user is an administrator, super-administrator or can view public metadata, view public data or add shared metadata.

Example listing users (curl)

```
curl -H 'Accept: application/json' 'http://localhost:8080/api/users/'
```

PATCH /api/users/

Update user roles.

PATCH /api/users/

Parameters:

id	string	Id of the user for which roles will be updated.
"role name"	boolean	Role name is any of <code>isAdmin</code> , <code>canViewPublicData</code> , <code>canViewPublicMetadata</code> or <code>canAddSharedMetadata</code> . The value determines whether the user has the role or not.

Example updating user roles (curl)

```
curl -X PATCH -H "Accept: application/json" -d \
'{
  "id": "xyz-0000",
  "canViewPublicData": false,
  "canViewPublicMetadata": true
}' \
"http://localhost:8080/api/users/"
```

GET /api/users/current

Get current user.

Response:

Returns current user's unique ID, name, email, username and user's organisation-level permissions: if the user is an administrator, super-administrator or can view public metadata, view public data or add shared metadata.

Example getting current user (curl)

```
curl -H "Accept: application/json" "http://localhost:8080/api/users/current"
```

POST /api/users/current/logout

logout the current user.

Example logging out (curl)

```
curl -X POST "http://localhost:8080/api/users/current/logout"
```

Configuration endpoints

Vocabulary

The vocabulary contains a description of the structure of the metadata. It contains the types of entities that can be created, along with the data types for the fields. It is stored in [SHACL](#) format.

GET /api/vocabulary/

Retrieve a representation of the vocabulary.

Example fetching the vocabulary in turtle format (curl)

```
curl -H 'Accept: text/turtle' 'http://localhost:8080/api/vocabulary/'
```

Example fetching the vocabulary in json-ld format (curl)

```
curl -H 'Accept: application/json+ld' 'http://localhost:8080/api/vocabulary/'
```

Features

GET /api/features/

List available application features.

Response contains list of additional features that are currently available in the application.

Example listing features (curl)

```
curl -H 'Accept: application/json' 'http://localhost:8080/api/features/'
```

Services

GET /api/services/

List linked services.

Response contains list of external services linked to FairSpace, e.g. JupyterHub, cBioPortal, etc.

Example listing services (curl)

```
curl -H 'Accept: application/json' 'http://localhost:8080/api/services/'
```

External storages

GET /api/storages/

List linked data storages.

Response contains list of external data storages linked to Fairspace.

Example listing external storages using curl

```
curl -H 'Accept: application/json' 'http://localhost:8080/api/storages/'
```

External file system integration

As Fairspace supports the [WebDAV](#) protocol, it can be configured to connect to external data storages that implement a WebDAV interface. An overview of external files is integrated into Fairspace user interface. Currently, a read-only interaction is supported. Users can browse through the external file system, read the data and metadata (e.g. creation date, description). Files from the external storage will be also made available for analysis in Jupyter Hub.

Access policy

Access policies differ between systems. To avoid inconsistencies, permissions validation and management are expected to be under control of the external storage system. Each storage component is responsible for its own policy and needs to perform the required checks to ensure that users only get to see the data they are supposed to see.

It is assumed that a user requesting files from a storage using WebDAV has at least "read" access to all the files included in the WebDAV response. Access can be further limited by using a custom [access](#) property. If a value of this property on a resource is set to "List", the resource's metadata will be readable, but it will not be possible to read the resource's content.

Another assumption is that the Fairspace client can authenticate in the external storage via the same Keycloak and the same realm as configured for Fairspace, so that the same bearer token can be used for all storages. See the [Authentication](#) section for more information.

API specification and supported parameters

A subset of default WebDAV properties is used and displayed as a resource metadata in the Fairspace user interface. These properties are presented in the table below.

WebDAV property	Description
DAV:creationdate	Creation date
DAV:iscollection	Flag determining whether a resource is a file or directory
DAV:getlastmodified	Last modification date

WebDAV property	Description
<code>DAV:getcontentlength</code>	Size of the file (0 for directories)

There is also a set of custom Fairspace properties, some of which are required to be returned from the WebDAV request.

WebDAV property	Description
<code>iri</code>	IRI of the resource. Required.
<code>createdBy</code>	Id of a user that created the resource.
<code>comment</code>	Resource description.
<code>access</code>	By default, users are granted <code>Read</code> access to the resource returned from WebDAV endpoint. Other supported value is <code>List</code> , which means that users can see the resource and its metadata, but cannot read its content.
<code>metadataEntities</code>	List of IRIs in a form of comma-separated string. IRIs represent all metadata entities linked to the resource. If the IRI matches a metadata entity stored in Fairspace, such an entity will be displayed in the user interface.

It is also supported to specify any other custom property in the WebDAV response body, as WebDAV responses are easily extendable. All these properties (if not specifically marked as excluded in Fairspace), will be displayed in the user interface in a form of key-value pairs.

Text search in external storages

Text based search on external file system can be enabled in the Fairpsace user interface, if the external system exposes a search endpoint, following the specification from the [Text search](#) section. To enable finding files based on name or description, `searchUrl` has to be specified in the storage configuration.

Configuration

Multiple external storages can be configured simultaneously. A list of configuration parameters is presented below.

Parameter	Description
<code>name</code>	Unique name of the storage.
<code>label</code>	String to be used as a display name of the storage.
<code>url</code>	WebDAV endpoint to connect to.

Parameter	Description
<code>searchUrl</code>	Optional search endpoint URL. If specified, a text based search on file name or description will be enabled in the user interface.
<code>rootDirectoryIri</code>	Optional IRI of the root directory. If not specified, <code>url</code> will be used as a default root directory.

Sample configuration of storages in YAML format:

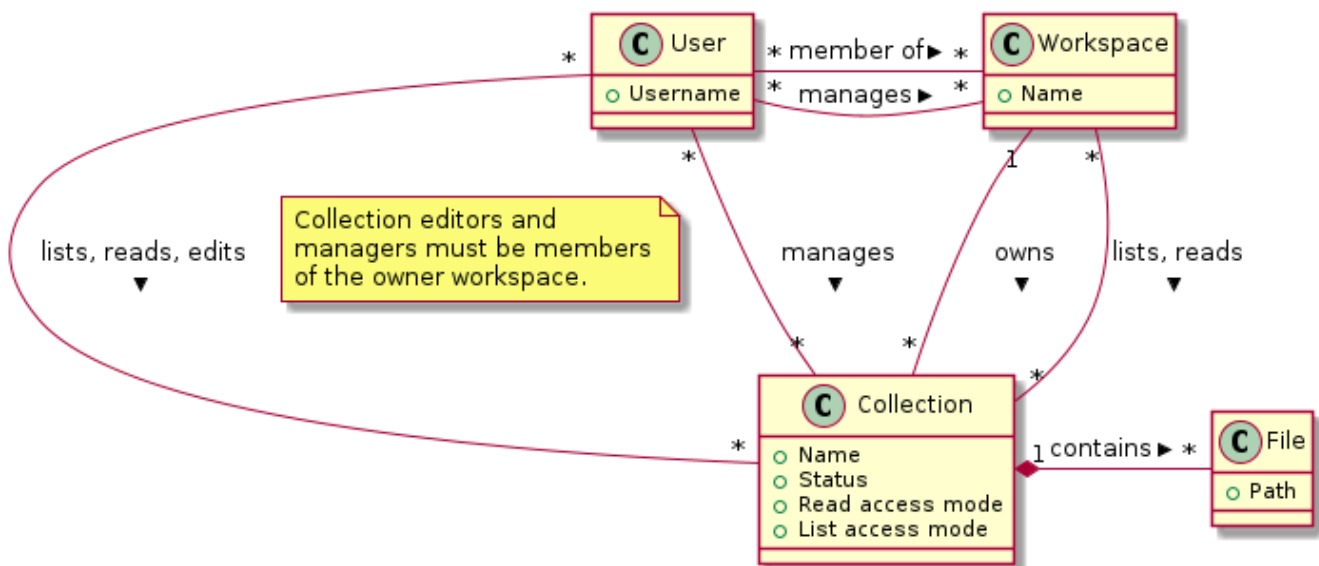
```
storages:
  exStorage1:
    name: exStorage1
    label: "External storage 1"
    url: https://exstorage1/api/webdav
    searchUrl: https://exstorage1/api/search/
    rootDirectoryIri: http://ex1/api/webdav/
  exStorage2:
    name: exStorage2
    label: "External storage 2"
    url: https://exstorage2/api/webdav
```

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

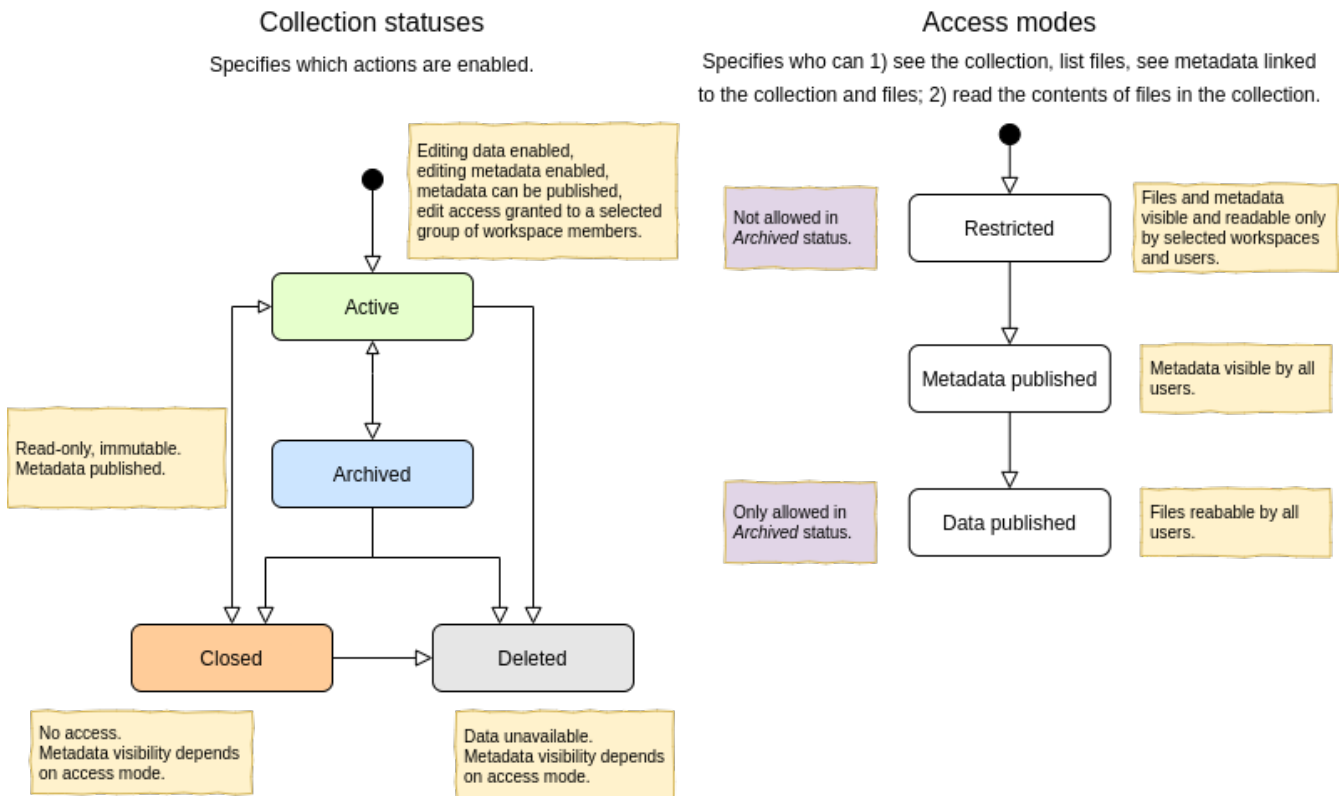
During the lifetime of a collection, different rules may be applicable for data modification and data access. In Fairspace, collections follow a workflow with the following statuses:

- *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.



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

Metadata

Metadata is data about data. Metadata is used to describe data assets, e.g., for making it easier to find or use certain data. Because metadata is data itself, it can be difficult to make a proper distinction between data and metadata in a system.

Types of metadata

In a digital archive, *technical metadata* is linked to data assets, like file type, location, size, creation or modification dates, checksums for checking data integrity, ownership. Such metadata is essential for a system to store and retrieve data files. Technical metadata can also include data format specific properties, like encoding, data layout, resolution, etc., required to correctly read the data. With most publications, *bibliographic metadata* is associated, such as author, title, abstract, publication details, keywords and subject categories. Such metadata makes it possible to find relevant publications. This is the kind of metadata used by libraries and archives and numerous standards exist for such data, such as [Dublin Core](#) and [METS](#).

More detailed *descriptive metadata* provides information about the contents of the data, e.g., description of rows and columns, summary statistics, project information, geographical information, results, study design, methods, materials or equipment. In the extreme case, the entire content of the file is captured in descriptive metadata.

We can distinguish different kinds of descriptive metadata, such as:

- Description of the *contents* (rows, columns, values, summary statistics)
- Description of the *subject*, what the data is about (subject, topic, project, study design, object of study, time, location)
- Description of *data sources* (for derived or processed data)
- Description of the *methods* or technology used to produce or capture the data, such as scripts and versions.

In the context of health research data, it is essential to link data to research subjects, i.e., patients and samples.

The values of the metadata can be of any type, numerical, free text, date, conform to a controlled vocabulary (e.g., ICD or SNOMED codes, units, file types) or a reference to a typed entity within the database, or external entities.

Likewise, the data the metadata is about can be of any type, a file system, a tabular file, image, genomic data, a relational database, etc.

Purpose

Metadata is used for several purposes:

- Descriptors to enable use of the data (file type, file format, encoding, how it was created/generated). The metadata may be used by users or scripts to read or interpret a particular file or data set.
- Finding relevant data for analysis:
 - Metadata may be used to organise data within a data set that a researcher is working on, by using (study specific) categories linked to individual files.
 - Metadata may be used in search queries or navigation to find out if data is available that meets certain selection criteria (e.g., data types, categories, cohort characteristics), for inclusion in a new analysis.
 - Metadata may be used to identify data that is linked to a specific entity, such as a patient or a sample, to determine if such data has already been analysed, in order to avoid duplicate analysis.

It is important to identify for which purpose metadata is collected and used, as it may affect which types of metadata are collected, how they are navigated and if access control on metadata is desired or required.

Data model

To enable validation of (meta)data, and to enable intuitive navigation and search within the metadata, it is essential to have a good data model.

The data model consists of the entity types (classes), their properties (with types) and relationships between entities that can be represented in the system.

The data model needs to be broad (expressive) enough to allow users to express all relevant facts about data sets conveniently and accurately, but it needs to be specific enough to allow validation and the generation of useful overviews and information pages. International data standards should be used as much as possible to enable interoperability between systems.

E.g., it is probably better to use a specific field ‘disease’ where the value must be a valid ICD-10 code, than using a generic ‘description’ field where a disease is described in a free text field.

Data domains

We distinguish different data domains in order to clearly separate the data that is system specific and the metadata that is more flexible.

Workspaces and collection-level data

Users, workspaces, collections, directories and files are system-level entities, representing the file system of the system. Access to these entities is restricted by the workspace-level and collection-level access control. These entities cannot be changed on demand, but are inherent to the system. However, custom properties and relations may be added, e.g., to link files to patients.

Metadata

The data model for the other (non system-level) entities, the shared metadata, can be configured, in

order to make the metadata suitable for the environment where it is used. These metadata are used to link entities in the file system to entities in the research domain, such as samples, patients, diseases, diagnoses, or to entities in the organisation domain, such as projects. These entities may be displayed and navigated in the application and can be explored through the API (for technical users).

Controlled vocabularies

The data model may contain controlled vocabularies (e.g., disease codes, file types, project phases) that can be used as values in the metadata. Every value in a controlled vocabulary has a unique identifier and a label. Using such vocabularies enables standardisation and validation of metadata values.

Reference data

The data model may support domain specific entity types (patients, samples, genes, treatments, studies, etc.) or generic entity types (project, organisation, person, etc), defining the metadata objects that collection-level data assets can refer to. The reference data can also be linked.

Every entity has a unique identifier, a type, a label, and the properties and relations as specified by the type. These entities do not belong to a particular space that is owned by a specific group or user.

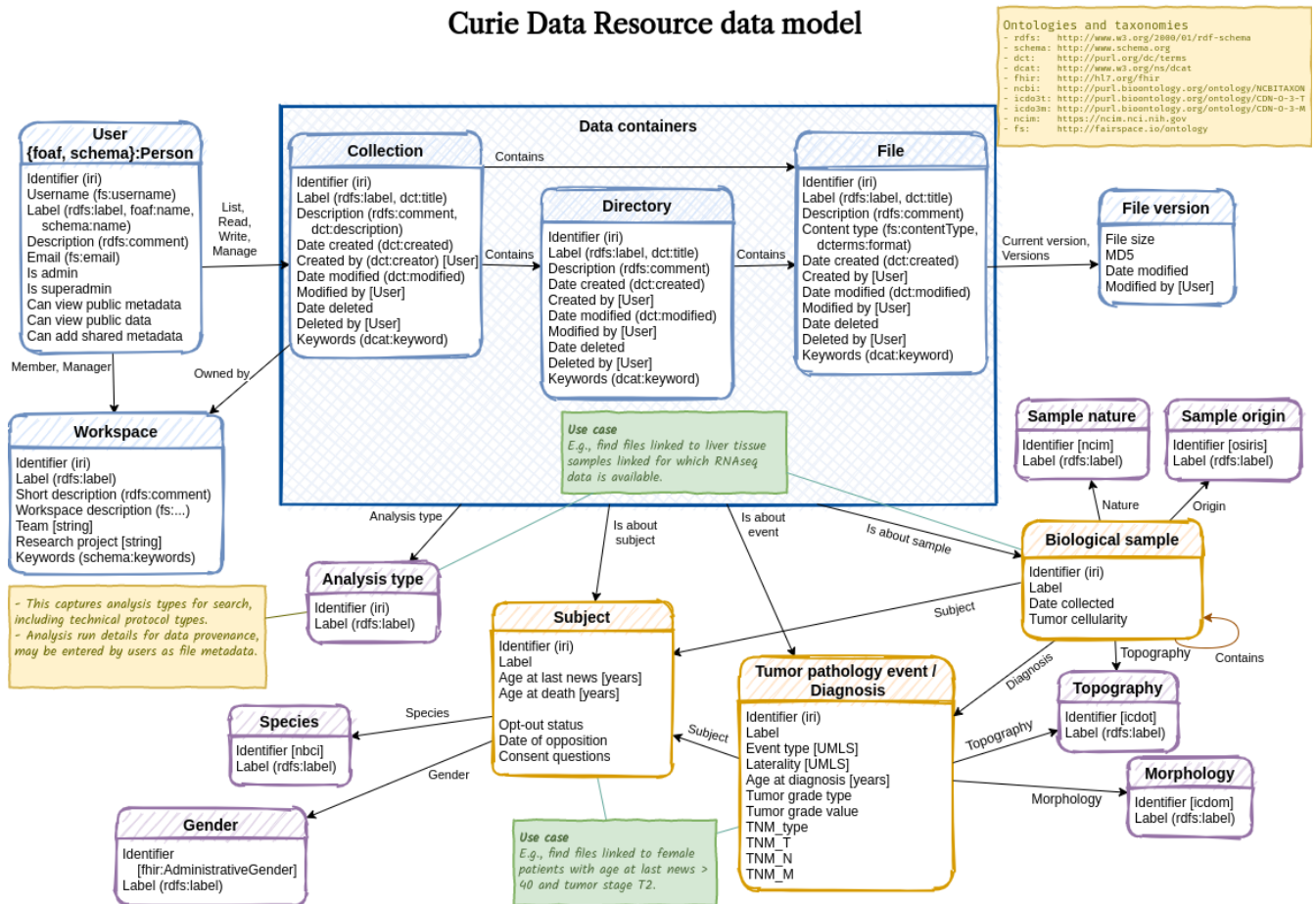
Data model configuration

Fairspace uses an [Apache Jena](#) database to store system metadata and the custom domain specific metadata. The data models for these metadata are defined using the [Shapes Constraint Language \(SHACL\)](#).

- The system metadata includes workspaces, collections, directories, files, file versions, users and access rights. The system data model is defined in [system-vocabulary.ttl](#)
- The customisable data model includes the custom (shared) metadata entities, custom controlled vocabulary types, and custom properties of the system entities. The default custom data model is defined in [vocabulary.ttl](#). This data model can be overridden by a data more suitable for your organisation.

A schematic overview of the default data model in [vocabulary.ttl](#):

Curie Data Resource data model



The data model defines an entity-relationship model, specifying the entity types that are relevant to describe your data assets, the properties of the entities, and the relationships between entities.

Example 2. Example data model

In this example data model, the following custom entity types are defined:

- `example:Gender` with property *Label*;
- `example:Species` with property *Label*;
- `example:Subject` with properties *Gender*, *Species*, *Age at last news* and *Files*.

The system class `fs:File` is extended with the *Is about subject* property.

```
@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:ageAtLastNews 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 "Label" ;
  sh:description "Unique subject label." ;
  sh:datatype xsd:string ;
  sh:maxCount 1 ;
  dash:singleLine true ;
  fs:importantProperty true ;
  sh:path rdfs:label;
  sh:order 0
],
[
  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
],
[
  sh:name "Age at last news" ;
  sh:description "The age at last news." ;
  sh:datatype xsd:integer ;
  sh:maxCount 1 ;
  sh:path example:ageAtLastNews
],
[
  sh:name "Files" ;
  sh:description "Linked files" ;
  sh:path [sh:inversePath example:aboutSubject];
] .

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
] .

```

All entity types have a unique label, specified using the `rdfs:label` predicate. The *Gender* and *Species* properties link the subject to an entity from the respective controlled vocabularies. The *Age at last news* property is a numerical (integer) value property.

The *Files* property of the *Subject* entity type is an example of an inverse relation. The link is defined on the file, but the link will be visible on the subject as well, because of this inverse relation.

The following guidelines should be followed when creating a custom data model.

- Define a namespace for your custom entities and properties, like `@prefix example: https://example.com/ontology# .` in the example.
- Each custom entity type must have types `rdfs:Class` and `sh:NodeShape`, the properties `sh:closed false` and `sh:ignoredProperties (rdf:type owl:sameAs)`, and a valid value for `sh:name`. The `sh:description` property is optional.
- Controlled vocabulary or terminology types are modelled as entity types as well, having only the *Label* (`rdfs:label`) property, see `example:Gender` and `example:Species`.
- Properties are specified using the `sh:property` property.
 - Every entity type must have a property *Label* (`sh:path rdfs:label`) of data type `xsd:string`. The label of an entity must be unique for that type. The label property should be singleton and marked `fs:importantProperty true`. If there are multiple properties, the label should have `sh:order: 0`.
 - Properties must have a valid value for `sh:name`. The `sh:description` property is optional.
 - A property must either have a `sh:datatype` property, specifying one of `xsd:string`, `xsd:integer` or `xsd:date`, or a property `sh:class` specifying an entity type as the target of a relationship.
 - The predicate used for the property (the middle part of the RDF triple) is specified with the `sh:path` property, e.g., `example:aboutSubject` for the *Is about subject* relation.
 - If a relationship is bidirectional, the path of the inverse relation is specified using `sh:inversePath`, see the *Files* property on the *Subject* entity type.
 - A property can be marked *mandatory* by specifying `sh:minCount 1`. A property can be marked *singleton* by specifying `sh:maxCount 1`.
 - A text property (with `sh:datatype xsd:string`) can be limited to a single line text field using `dash:singleLine true`.

Limitations

Although assigning multiple types to an entity is easy in RDF, Fairspace assumes entities to have a single type.

Inheritance is possible in SHACL, but not supported by Fairspace. Instead of specifying an entity type as a subtype of another, a single type can be specified with a *type* property, indicating the sub type of the entity.

E.g., instead of defining entity types *DNaseqAssay* and *RNAseqAssay* as sub types of *Assay*, a property type *assayType* can be defined on *Assay*, using a controlled vocabulary type *AssayType* with the assay types as values.

Controlled vocabularies

For controlled vocabulary types, e.g., *Gender* and *Species* in the example, you should insert the allowed values in the database by uploading a taxonomies file using the [RDF metadata](#) API. An example taxonomy is in [taxonomies.ttl](#).

It is preferred to use existing standard taxonomies and labels. If that is not possible, please define your own namespaces for your custom taxonomies.

Example 3. Example taxonomy

In this example we use existing standard ontologies for the *Gender* and *Species* controlled vocabulary types.

- The [HL7 FHIR AdministrativeGender code system](http://hl7.org/fhir/administrative-gender#) for *Gender*.
- The [NCBI Organismal Classification](https://bioportal.bioontology.org/ontologies/NCBITAXON/) for *Species*.

```
@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" .
```

View configuration

For the metadata pages in the user interface, a view configuration needs to be created that specifies the tables and columns. An example can be found in [views.yaml](#)

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
```

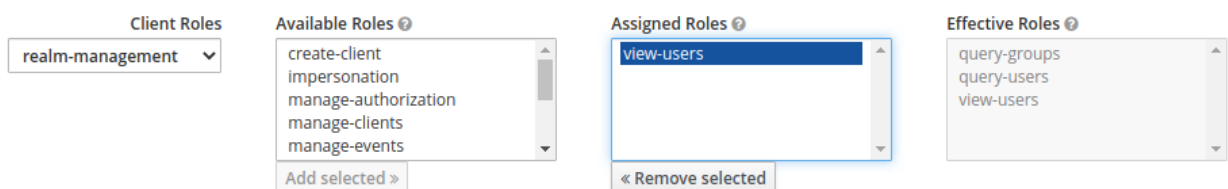
On MacOS the docker logging driver needs to be configured, because the default is not available (`journald`). Override the logging driver by setting the `DOCKER_LOGGING_DRIVER` environment variable or adding a line the `.env` file in `local-development`:

```
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 the `view-users` role to the client service account:
 - In the realm, select `Clients` → Select client
 - Choose tab `Service Account Roles`
 - Select `realm-management` from the `Client Roles` dropdown
 - Select `view-users` from the available roles, click `Add selected`.



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 <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 **fairspaceicluster** as cluster id for the CI environment. Ensure that your Google account has access to the **fairspace-207108** GCP project and set the project property

```
gcloud config set project fairspace-207108
```

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 repo 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
# List available fairspace chart versions
~/bin/helm/helm fetch fairspace/fairspace --versions
# Fetch the fairspace chart
~/bin/helm/helm fetch fairspace/fairspace --version 0.7.11
```

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 0.7.11 --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-0.7.11.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**.



Be careful, you may lose data!

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

Logging

For several purposes, three types of logs are generated:

- *Application log*: Informative messages about the system state and application errors. This enables system administrators to diagnose problems.
- *Audit log*: Records all user actions that add, change or delete data and access to files. This enables system administrators to audit important changes and access to sensitive data.
- *Transaction log*: Detailed log of all database changes. This enables the system to restore the database if it is corrupted.



Do not change or remove this log!

As default, application and audit logs are written to standard output. Additionally, the audit log is also written to log files in `data/audit`. Location can be overwritten by setting the `AUDIT_LOG_ROOT` environment variable. The log files are automatically ‘rolled over’: today’s records are in `audit.log`, previous records are stored in daily log files with file name pattern `data/audit/audit.yyyy-MM-dd.log`, and are retained for 50 days. If the audit log needs to be kept for a longer period, the log configuration can be replaced, or log files can be transferred elsewhere, e.g., using [filebeat](#). The audit log is encoded in a JSON format, that can be processed by, e.g., [logstash](#).

The transaction log is stored in `data/log` by default.

Configuration

The default log configuration for application and audit logs is in [log4j2.properties](#). The default can be overridden by placing a file `log4j2.properties` in the working directory where the application is run.

Audit log

The audit log is generated using [log4j 2](#) and *Mapped Diagnostic Context (MDC)*. The basic idea of Mapped Diagnostic Context is to provide a way to enrich log messages with pieces of information that could be not available in the scope where the logging actually occurs, but that can be indeed useful to better track the execution of the program. Basically it groups log data from a single event (`MDC.put`) into a one log message.

Audit log entries contain several fields, including `event`, `user_name`, `user_email`, `user_id` and request specific additional parameters. Below we list the actions that are logged and which information is captured in the audit log.

WebDAV

Request: `/api/webdav`

Request method	Event	Additional params	Description
GET	FS_READ	path, version, success	Read file
PROPPATCH	FS_PROPPATCH	path, success	Not used by the UI (metadata endpoint used instead)
MKCOL	FS_MKDIR	path, success	Create a new collection or directory.
COPY	FS_COPY	path, destination, success	Copy a file or directory (executed on “paste” action)
MOVE	FS_MOVE	path, destination, success	Rename a resource, move to a different location
DELETE	FS_MARK_AS_DELETED, FS_DELETE	path, success	Mark a resource as deleted or delete permanently
PUT	FS_WRITE	path, success	Not used by the UI (metadata endpoint used instead)
POST	FS_ACTION	path, action, parameters, success	Upload a file(s) or folder, undelete a resource, change collection status, access mode, owner or permissions.

Events that are not logged: accessing collections, listing collection and directory contents, listing previous versions.

Metadata

Request: `/api/metadata/`

Request method	Event	Additional params	Description
PUT	METADATA_UPDATED	iri	Add all the statements in the given model to the database
(Soft) DELETE	METADATA_MARKED_AS_DELETED	iri	Mark an entity as deleted
DELETE	METADATA_DELETED	iri	Delete metadata

Request method	Event	Additional params	Description
PATCH	METADATA_UPDATED	iri	Overwrite metadata in the database

Fetching metadata is not included in audit log.

Workspace operations

Request: `/api/workspace/`

Request method	Event	Additional params	Description
PUT	WS_CREATE	workspace	Create workspace
DELETE	WS_DELETE	workspace	Delete workspace
PATCH <code>users/</code>	WS_SET_USER_ROLE	workspace, affected_user, role	Add user to the workspace, change workspace user role, delete a user from a workspace

Listing workspaces and workspace users is not included in the audit log.

User roles operations

Request: `/api/users/`

Request method	Event	Additional params	Description
PATCH	USER_UPDATE	affected_user	Change organisation user role

Fetching user information is not included in the audit log.

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

...