EUROPEAN COMMISSION

Community Research

SEVENTH FRAMEWORK PROGRAMME

Private Public Partnership Project (PPP)

Large-scale Integrated Project (IP)

FIWARE-Metaware

Project full title: Future Internet Core
Project acronym: FI-Core
Contract No.: 632893
description: Metadata Store Management Platform Documentation

Contents

Metaware

Introduction

Welcome to the Metaware Installation and Administration Guide! Metaware is a Java-based web service that deals with metadata information, and it can run on Tomcat.

It is an implementation of the FIWARE Metadata Store Management Platform Generic Enabler, from Telecom Italia and Consoft Sistemi S.p.A.

Any feedback on this document is highly welcome, including bug reports, typos or stuff you think should be included but is not. Please send the feedback through GitHub. Thanks in advance.

User guide

The interaction with Metaware is designed to work only through RESTful APIs, and no user interface is currently implemented; for these reasons we assume that the audience is mainly composed by developers and engineers, so please refer to the following section of this manual.

Programming guide

As mentioned in the Installation and Administration Guide, Metaware is composed by three layers:

1) the Web application (`metaware`) is the upper layer which exposes the RESTful APIs, together with the related services; this layer is also in charge to start the connection to the database, which is then managed by the lower layer.
2) the middle-layer (`metaware-dao`) is composed by the DAO definitions of Metaware (including entities definition); this layer contains also some basic string definitions and custom exceptions.
3) the lower-layer contains the DAO implementation based on MongoDB, together with the basic entities definition; indeed, this is the layer that directly communicates with the MongoDB instance.

Please note that even if the entities definition is included in both middle and lower-layer, only the latter is really involved in the workflow of Metaware; the former can be considered only as a line-guide (and indeed the DAO are interface classes). This is mainly due to some technical restrictions imposed by the usage of MongoDB; in a future release, the middle-layer will be completely included in the lower-layer, in order to "clean-up" the code.

Entities

The following represents a list of entities, or objects, declared and used in Metaware.

Before proceeding with the analysis of each entity it is extremely important to mention that every Java entity class extends a BasicDBObject. This extension allows to deal with MongoDB objects and facilitate the data-flow from and to MongoDB itself. One of the most notable fact about this extension is that all the entity classes are managed like a `Map` and this means that the entity values are not "physically" stored in the internal variables, but stored as a "key-value" pair in a Map (which indeed comes from the BasicDBObject). By having a look to the various "getters" and "setters" it is possible to see that there is not direct variable assignation, but instead the methods `put(String key)`, `get(String key)`, `get<type>(Stringkey)` are used.

The `get<type>(Stringkey)` method is used to execute the automatic conversion (when possible) of the requested field to the specified type. Some examples are: getString(String key), getLong(String key), getObjectId(String key). In Metaware when the type of the requested field is known, such methods are used instead of the generic one, in order to include an additional layer in type-checking. Obviously, if the type of the selected field does not have the corresponding method, the generic one is called, together with the explicit type conversion (`get(String key)` returns an `Object`).

A possible alternative to the current usage of MongoDB Java driver consists in using a ORM, like Morphia, which can definitely speed-up the development, but for this version of Metaware we chose to use natively the MongoDB Java driver and deal with BasicDBObjects.

Another important note is that most of the following entities, in addition to the specified internal fields, they have the `_id` field, which is automatically added once the entity is stored in the database. The type of this field is ObjectId.

Company

This entity contains the basic information that can be associated to a common company.

This entity is most probably the "easiest" one, since it doesn't contain any reference to other entities; the schema is the following:

Key name	Value type
`name`	`String`
`description`	`String`
`email`	`String`
`phone`	`String`
`address`	`String`
`url`	`String`

The Company entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.Company`

Department

A department belongs to a company and such relation is made by the presence of the Id of the associated company. The type of this Id is `ObjectId`.

Key name	Value type	Note
`name`	`String`	
`description`	`String`	
`company_id`	`ObjectId`	At the web application level this field is represented as a String
`email`	`String`	
`phone`	`String`	

The Department entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.Department`.

User

A user can be related to a department and a company, and indeed the schema contains both references.

Key name	Value type	Note
`name`	`String`	
`surname`	`String`	
`email`	`String`	
`phone`	`String`	
`address`	`String`	
`company_id`	`ObjectId`	At the web application level this field is represented as a String
`department_id`	`ObjectId`	At the web application level this field is represented as a String
`username`	`String`	
`password`	`String`	The value is hashed

Key name	Value type	Note
`role`	`String`	Can be `basic` or `admin`

The hash of the password is computed before the storage by the UserService class (at the web application layer).

The User entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.User`.

Permission

A permission is associated to one entity like Algorithm, Dataset, Data-Source, or Process. It specifies if a user, a department, or a company, can or cannot perform a read, update, or delete action.

This entity does not include the `_id` field, since it is only stored as a sub-document of an outer entity.

Key name	Value type	Note
`referenceId`	`ObjectId`	This refers to a User, a Department, or a Company
`perm`	`String`	Can be a combination of `r`, `u`, and `d`

Some possible values of `perm` are: `r`, `ru`, `ud`, and `rud`, which respectively means "the user | department | company specified in the `referenceId` can read | read and update | update and delete | read, update and delete the related entity".

The Permission entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.Permission`.

Algorithm

The following is the schema of the Algorithm metadata entity.

Key name	Value type	Note
`name`	`String`	
`description`	`String`	
`type`	`String`	
`creationDate`	`Long`	Represented as Unix time
`lastModifiedDate`	`Long`	Represented as Unix time
`permissions`	`List<Permission>`	List of permission specifications
`owner`	`ObjectId`	This refers to a User, a Department, or a Company At the web application level this field is represented as a String
`status`	`String`	
`model`	`String`	
`subModel`	`String`	
`hiveQuery`	`String`	
`elapsedTime`	`Long`	Represented as Unix time
`runNumber`	`int`	
`logUrl`	`String`	

The Algorithm entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.Algorithm`.

Dataset

The following is the schema of the Dataset metadata entity; which is completed by the entity called `DatasetStructure` (sub-entity)

Key name	Value type	Note
`name`	`String`	

Key name	Value type	Note
`description`	`String`	
`type`	`String`	
`creationDate`	`Long`	Represented as Unix time
`lastModifiedDate`	`Long`	Represented as Unix time
`permissions`	`List<Permission>`	List of permission specifications
`owner`	`ObjectId`	This refers to a User, a Department, or a Company At the web application level this field is represented as a String
`status`	`String`	
`readOnly`	`boolean`	
`structure`	`DatasetStructure`	See the following entity

Thanks to the exposition of ImportOpenData and ExportOpenData APIs, Dataset metadata entity can also be represented in DCAT format (XML - RDF). The following is an example of a Dataset in DCAT representation.

It is important to know that conversion from “normal” entity representation to DCAT is not lossless, and indeed future release of Metaware will improve this functionality.

```xml
<?xml version="1.0"?>
<rdf:RDF
    xmlns:dct="http://purl.org/dc/terms/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:adms="http://www.w3.org/ns/adms#"
    xmlns:vcard="http://www.w3.org/2006/vcard/ns#"
    xmlns:dcat="http://www.w3.org/ns/dcat#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/" >
    <rdf:Description rdf:about="http://metaware.org/dataset#559503cd11f61c70f31dbbff"\>
        <rdf:type>http://www.w3.org/ns/dcat#Dataset\</rdf:type\>
        <dct:description>Bike Sharing of Turin from AperTO</dct:description>
        <dct:publisher rdf:resource="http://metaware.org/organization#550812fab2d1f89cb41208c2" /\>
        <dct:title>Bike Sharing</dct:title>
        <adms:contactPoint rdf:resource="http://metaware.org/contactpoint#550812fab2d1f89cb41208c2"/\>
        <dcat:keyword>bike</dcat:keyword>
        <dct:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"\>2016-09-18T03:01:01\</dct:modified\>
        <dct:identifier>559503cd11f61c70f31dbbff</dct:identifier>
        <dct:issued rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"\>2014-09-18T03:01:01\</dct:issued\>
        <dcat:distribution rdf:resource="http://metaware.org/distribution#558d759f8192682c10631285"/\>
    </rdf:Description>
    <rdf:Description rdf:about="http://metaware.org/organization#550812fab2d1f89cb41208c2"\>
        <foaf:mbox>info@tilab.com</foaf:mbox>
        <foaf:homepage>http://www.telecomitalia.com/\</foaf:homepage\>
        <foaf:name>Telecom Italia</foaf:name>
        <dct:type>http://purl.org/adms/publishertype/Company\</dct:type\>
        <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Agent"/\>
    </rdf:Description>
    <rdf:Description rdf:about="http://metaware.org/address#"\>
        <vcard:country-name>tmp</vcard:country-name>
        <vcard:locality>tmp</vcard:locality>
        <vcard:postal-code>tmp</vcard:postal-code>
        <vcard:street-address>Via Reiss Romoli 274 Torino</vcard:street-address>
        <rdf:type>http://www.w3.org/2006/vcard/ns#Address\</rdf:type\>
    </rdf:Description>
    <rdf:Description rdf:about="http://metaware.org/publisher#558d761b8192682c10631286"\>
        <foaf:name>Telecom Italia</foaf:name>
        <dct:type>http://purl.org/adms/publishertype/Company\</dct:type\>
        <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Agent"/\>
    </rdf:Description>
    <rdf:Description rdf:about="http://metaware.org/telephone#"\>
        <rdf:type>http://www.w3.org/2006/vcard/ns#Work\</rdf:type\>
        <rdf:type>http://www.w3.org/2006/vcard/ns#Voice\</rdf:type\>
        <vcard:hasValue>+390112285111</vcard:hasValue>
    </rdf:Description>
    <rdf:Description rdf:about="http://metaware.org/distribution#558d759f8192682c10631285"\>
        <dct:license>http://www.dati.gov.it/iodl/2.0/\</dct:license\>
        <dct:format>cvs</dct:format>
        <dct:description>Comma separated value file</dct:description>
        <dcat:accessURL>http://aperto.comune.torino.it/sites/default/files/bike_sharing.csv\</dcat:accessURL\>
        <rdf:type>http://www.w3.org/ns/dcat#Distribution\</rdf:type\>
    </rdf:Description>
    <rdf:Description rdf:about="http://metaware.org/catalog#"\>
        <dcat:dataset rdf:resource="http://metaware.org/dataset#559503cd11f61c70f31dbbff"/\>
        <dct:license rdf:resource="http://www.dati.gov.it/iodl/2.0/"/\>
        <foaf:homepage>http://metaware.org/\</foaf:homepage\>
        <dct:title>Default Catalog Title</dct:title>
        <dct:publisher rdf:resource="http://metaware.org/publisher#558d761b8192682c10631286"/\>
        <rdf:type>http://www.w3.org/ns/dcat#Catalog\</rdf:type\>
        <dct:description>This is the default catalog for MetaWare.</dct:description>
        <dct:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"\>2015-09-04T10:40:18\</dct:modified\>
        <dct:language>http://id.loc.gov/vocabulary/iso639-1/en\</dct:language\>
        <dct:issued rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"\>2015-09-04T10:40:18\</dct:issued\>
    </rdf:Description>
    <rdf:Description rdf:about="http://metaware.org/contactpoint#550812fab2d1f89cb41208c2"\>
        <vcard:hasAddress rdf:resource="http://metaware.org/address#"/\>
        <vcard:hasTelephone rdf:resource="http://metaware.org/telephone#"/\>
        <vcard:hasEmail rdf:resource="mailto:info@tilab.com"/>
        <vcard:fn>Telecom Italia</vcard:fn>
        <rdf:type>http://www.w3.org/2006/vcard/ns#Organization\</rdf:type\>
    </rdf:Description>
</rdf:RDF>
```

The Dataset entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.Dataset`.

DatasetStructure

The `DatasetStructure` is a sub-entity (or sub-document) of `Dataset`, and for this reason it does not have the `_id` field. This sub-entity does not have a well-defined schema neither, because each dataset can have a different structure; the class `DatasetStructure` should be considered as a simple `Map`.

The following is an example of `Dataset` and the `DatasetStructure` (field `structure`) contains a certain number of keys that specify the various fields contained in the metadata object.

```
{
    "_id": "55b2415196f0bc192c79cc31",
    "name": "Skate Sharing",
    "description": "Skate Sharing of Turin from AperTO",
    "type": "Default type",
    "creationDate": 1411002061,
    "lastModifiedDate": 1474160461,
    "permissions": [],
    "owner": "550812fab2d1f89cb41208c2",
    "status": "public",
    "readOnly": false,
    "structure": {
        "ID_SKATE_SHARING": "String",
        "NAME_HUB": "String",
        "UBICAZIONE": "String",
        "URL": "String",
        "COORD_X": "Float",
        "COORD_Y": "Float"
    }
}
```

So the previous example is the metadata of the Dataset, instead the final Dataset will be something like this:

ID_SKATE_SHARING	NAME_HUB	UBICAZIONE	URL	COORD_X	COORD_Y
11	hub1	Street 1	http://example/skate/11/	45.1	45.2
22	hub1	Street 1	http://example/skate/22/	45.1	45.2
33	hub2	Street 4	http://example/skate/33/	47.1	45.9

As you can see, the titles of the various columns are exactly the same fields of the `structure` sub-document.

The DatasetStructure sub-entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.DatasetStructure`.

Data-Source

The following is the schema of the Data-Source metadata entity.

Key name	Value type	Note
`name`	`String`	
`description`	`String`	
`type`	`String`	
`creationDate`	`Long`	Represented as Unix time
`lastModifiedDate`	`Long`	Represented as Unix time
`permissions`	`List<Permission>`	List of permission specifications
`owner`	`ObjectId`	This refers to a User, a Department, or a Company At the web application level this field is represented as a String
`status`	`String`	
`subtype`	`String`	
`url`	`String`	
`username`	`String`	
`password`	`String`	
`resourceType`	`String`	Can be `table`, `query`, or `file`
`resource`	`String`	Can be `table name`, `query`, `file name`

The Data-Source entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.DataSource`.

Process

The following is the schema of the Process metadata entity.

Key name	Value type	Note
`name`	`String`	
`description`	`String`	

Key name	Value type	Note
`type`	`String`	
`creationDate`	`Long`	Represented as Unix time
`lastModifiedDate`	`Long`	Represented as Unix time
`permissions`	`List<Permission>`	List of permission specifications
`owner`	`ObjectId`	This refers to a User, a Department, or a Company At the web application level this field is represented as a String
`status`	`String`	
`frequence`	`String`	
`startingTime`	`Long`	Represented as Unix time
`lastRunTime`	`Long`	Represented as Unix time
`runNumber`	`Long`	
`processingBlock`	`List<ProcessingBlock>`	See the following entity

The Process entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.Process`.

ProcessingBlock

The `ProcessingBlock` is a sub-entity (or sub-document) of `Process`, and for this reason it does not have the `_id` field; instead, it has a precise schema, which is the following.

Key name	Value type
`order`	`int`
`block`	`String`

The ProcessingBlock sub-entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.ProcessingBlock`.

Template

For each entity type (i.e., Algorithm, Dataset, Data-Source, Process), there is a Template that describes the various key-value pairs contained in each schema (actually, the real schema is described in the field `details`, described below).

Contrary to the other entities, the index of Template is represented by the field `name` and indeed it does not have the `_id` field (which is not necessary).

The following is the basic schema of a Template.

Key name	Value type	Note
`name`	`String`	
`details`	`TemplateDetails`	See the following entity

The Template entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.Template`.

TemplateDetails

Similarly to `DatasetStructure`, `TemplateDetails` is a sub-entity (or sub-document) of `Template`, and for this reason it does not have the `_id` field. This sub-entity does not have a well-defined schema and indeed the class `TemplateDetails` is treated like a `Map`.

The following is an example of `Template` and the `TemplateDetails` (field `details`) contains a certain number of keys that specify the various fields contained in the metadata object.

```
{
    "_id": "550b0b10264ed792cace7a3c",
    "name": "user",
    "details": {
        "_id": "id - string",
        "name": "string",
        "surname": "string",
        "email": "string",
        "phone": "string",
        "company_id": "id - string",
        "department_id": "id - string",
        "username": "string",
        "password": "hex - string",
        "role": "string"
    }
}
```

It is easy to understand that this is a template for User entity and the `details` field contains a sub-document that describes the various fields contained in the User metadata entity.

The TemplateDetails sub-entity is represented by the class `com.tilab.fiware.metaware.dao.impls.mongodb.domain.TemplateDetails`.

Authentication

For this first version of Metaware, we decided to use the Basic Access Authentication method, which consists in providing a username and password when making a HTTP request (REST). Every REST API accepts an authorization header parameter (`String authorization`) that is used to identify the user who is requesting the current resource. Here's an example:

```
public Response getAlgorithmsList(@HeaderParam("Authorization") String authorization) {
    /*...*/
}
```

The steps to produce this authorization string are well-known and at this address you can find a brief explanation.

Once the HTTP request comes to Metaware REST API, the method `filter(ContainerRequestContextcrc)` from the class `com.tilab.fiware.metaware.rest.AuthRequestFiler` (which implements `ContainerRequestFilter` class) is executed automatically; this method first extracts the authorization string from the header of the request (`crc`), which is passed to the method `decodeBasicAuth` (class `om.tilab.fiware.metaware.core.Util`) that returns an array of strings composed by username and password. The `filter` method starts from the credentials (just retrieved) and search for the related user (`getUserByCredentials(Stringusername, Stringpassword)`). If the user exists and is valid, then the authorization check is performed.

Authorization

The authorization in Metaware strongly relies on `securityRoleFiler(Useruser, String verb, String path)` method and on `Permission` entities (and of course the related method `checkPerm(List<Permission> permissionsList, StringuserId, CharSequenceperm)`).

When a new request comes and the related user is correctly deduced from the credentials, the `securityRoleFilter` method is called, by passing all the necessary parameters. The very first check is about the user, and in particular if the user is an administrator (`user.isAdmin()`), the authorization is immediately granted; this behavior might not be the best and probably this process will be enhanced during the next releases.

If the user is not an administrator (or *admin*) then we can identify several branches of execution, each branch is related to a specific type of resources and in particular we have:

- `algorithms`
- `api-docs` (this is for Swagger-UI)
- `companies`
- `datasets`
- `datasources`

- `departments`
- `discoverObjects`
- `users`
- `templates`

The behavior for each branch is quite similar and consists in:

1) retrieve the specified resource;
2) check if it is public or not;
3) check if the requestor is the owner of the resource;
4) considering the HTTP verb of the request (i.e., GET, POST, PUT, DELETE), check the permissions array associated with the resource.

Basically, if the resource is public or the requestor is its owner, the `securityRoleFilter` method returns true, otherwise the method `checkPerm` is executed. The latter takes the array of permissions associated with the resources, the Id of the requestor, and the action to be checked (i.e., `r` for read, `u` for update, `d` for delete), then checks if there is a permission in the array that corresponds to the requestor. If there is a match, the allowed actions are checked and the method returns true or false based on this last check.

If at the end of this process the user is authorized, the request can proceed, and the related REST API serves the request.

API summarize

Metaware exposes a rich set of RESTful API as Web Services, so all of them are accessible through HTTP calls. Most of them requires a basic authentication while calling (as explained before), this means that the header of the HTTP request must contain the authorization parameter with the hash of the user's credentials; in addition, when requested, another header parameter specifying the Content-Type must be inserted (`application/json` is valid in most of the cases).

The following is an example of a header parameters:

```
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
```

Where the `YWRtaW46YWRtaW4=` string is the hash of the user credentials.

The payload of the HTTP requests, when requested, must respect the schema of the related resource you are going to call; this is important in order to maintain the functionalities of Metaware. Unspecified schema in resource's call is accepted by Metaware, but then some functionalities will not work properly.

The following is the list of available RESTful APIs from Metaware.

Please assume the root of the Metaware as the following: http://localhost:8080/metaware/

Companies

Name	Verb	URL
Get Company	GET	/v1/companies/{companyId}
Get Companies List	GET	/v1/companies/
Create Company	POST	/v1/companies
Upsert Company	PUT	/v1/companies/{companyId}
Delete Company	DELETE	/v1/companies/{companyId}

The APIs that require a payload with the information related to the current Company metadata are:

- "Create Company"
- "Upsert Company"

Both of them have to respect the Company schema.

The “Upser Company” will replace the metadata of the selected company (specified by the `companyId`) with the content of the payload of the HTTP request.

Departments

Name	Verb	URL
Get Department	GET	/v1/departments/{departmentId}
Get Departments List	GET	/v1/departments
Create Department	POST	/v1/departments
Upsert Department	PUT	/v1/departments/{departmentId}
Delete Department	DELETE	/v1/departments/{departmentId}

The APIs that require a payload with the information related to the current Department metadata are:

- “Create Department”
- “Upsert Department”

Both of them have to respect the Department schema.

The “Upser Department” will replace the metadata of the selected department (specified by the `departmentId`) with the content of the payload of the HTTP request.

Users

Name	Verb	URL
Get User	GET	/v1/users/{userId}
Get Users List	GET	/v1/users
Create User	POST	/v1/users
Upsert User	PUT	/v1/users/{userId}
Delete User	DELETE	/v1/users/{userId}

The APIs that require a payload with the information related to the current User metadata are:

- “Create User”
- “Upsert User”

Both of them have to respect the User schema.

It is possible to create or update a user without specifying the `department_id` nor the `company_id`, but it is important to include the key fields in the JSON payload request. The following is a JSON example of a User creation without Department and Company.

```
{
    "name" : "Marco",
    "surname" : "Terrinoni",
    "email" : "marco@terrinoni.it",
    "phone" : "123456789",
    "address" : "Via Terrinoni",
    "company_id" : "",
    "department_id" : "",
    "username" : "marco",
    "password" : "marco",
    "role" : "basic"
}
```

The “Upser User” will replace the metadata of the selected user (specified by the `userId`) with the content of the payload of the HTTP request.

While updating a User, it is important to mention that only an admin can change the role of a User (independently from his/her role), but in every case it is not possible to upgrade the role from “basic” to “admin” (only “admin” to “basic”). If you want to upgrade a basic user to an admin, you have to create a new admin user from scratch.

DiscoverObjects

Name	Verb	URL
Discover usable objects	GET	/v1/discoverObjects/usable/{requestedId}
Discover owned objects	GET	/v1/discoverObjects/owner/{userId}

Algorithms

Name	Verb	URL
Get Algorithm	GET	/v1/algorithms/{algorithmId}
Get Algorithms List	GET	/v1/algorithms
Create Algorithm	POST	/v1/algorithms
Upsert Algorithm	PUT	/v1/algorithms/{algorithmId}
Delete Algorithm	DELETE	/v1/algorithms/{algorithmId}

The APIs that require a payload with the information related to the current Algorithm metadata are:

- “Create Algorithm”
- “Upsert Algorithm”

Both of them have to respect the Algorithm schema.

The “Upser Algorithm” will replace the metadata of the selected algorithm (specified by the `algorithmId`) with the content of the payload of the HTTP request.

Datasets

Name	Verb	URL
Get Dataset	GET	/v1/datasets/{datasetId}
Get Datasets List	GET	/v1/datasets
Create Dataset	POST	/v1/datasets
Upsert Dataset	PUT	/v1/datasets/{datasetId}
Delete Dataset	DELETE	/v1/datasets/{datasetId}
Import Open Data	POST	/v1/datasets/importOpenData
Export Open Data	GET	/v1/datasets/exportOpenData/{datasetId}

The APIs that require a payload with the information related to the current Dataset metadata are:

- “Create Dataset”
- “Upsert Dataset”
- “Import Open Data”

All of them have to respect the Algorithm schema. In particular the “Import Open Data” must respect the XML DCAT format representation of the Dataset, previously described in the Dataset schema. Still regarding the “Import Open Data” request, it is important to mention that the import process is not lossless.

The “Upser Dataset” will replace the metadata of the selected dataset (specified by the `datasetId`) with the content of the payload of the HTTP request.

Datasources

Name	Verb	URL
Get Data-Source	GET	/v1/datasources/{datasourceId}
Get Data-Sources List	GET	/v1/datasources
Create Data-Source	POST	/v1/datasources
Upsert Data-Source	PUT	/v1/datasources/{datasourceId}
Delete Data-Source	DELETE	/v1/datasources/{datasourceId}

The APIs that require a payload with the information related to the current Data-Source metadata are:

- “Create Data-Source”
- “Upsert Data-Source”

Both of them have to respect the Data-Source schema.

The “Upsert Data-Source” will replace the metadata of the selected data-source (specified by the `datasourceId`) with the content of the payload of the HTTP request.

Processes

Name	Verb	URL
Get Process	GET	/v1/processes/{processId}
Get Processes List	GET	/v1/processes
Create Process	POST	/v1/processes
Upsert Process	PUT	/v1/processes/{processId}
Delete Process	DELETE	/v1/processes/{processId}

The APIs that require a payload with the information related to the current Process metadata are:

- “Create Process”
- “Upsert Process”

Both of them have to respect the Process schema.

The “Upsert Process” will replace the metadata of the selected process (specified by the `processId`) with the content of the payload of the HTTP request.

Templates

Name	Verb	URL
Get Template	GET	/v1/templates/{templateName}
Get Templates List	GET	/v1/templates
Create Template	POST	/v1/templates
Delete Template	DELETE	/v1/templates

The API that requires a payload with the information related to the current Template is “Create Data-Source”; the payload has to respect the Template schema.

Swagger-UI

Metaware also comes with a Swagger-UI interface, which contains the whole API list. Please, follow these instructions to interact with Metaware through Swagger-UI (http:///:8080/swagger-ui/).