10

Interfaces

10.1 Data Access API

The data access API is the interface via which data consumers gain access to the data offered by a data provider or data space. Since this open interface enables direct interactions among stakeholders of different data spaces/marketplaces, we need not only an open interface specification that can be implemented by all but also a high level of security, as the data exchange might involve sensitive data, e.g., personal data or commercial data.

The endpoints documented below were grouped by modules.

Batch controller:

Get /batch/listDataSourceFiles/{offeringId}
(**batchController.listDataSourceFiles**)
Returns a list of datasets that are available for consumption.

post /batch/{data}/{agreementId}
(**batchController.getBatch**)
Requests data from a provider in the form of batch.

- **Request – First Block**
- **Request – Intermediate Block**
- **Request – Last Block**

Get /batch/listDataSourceFiles/{offeringId}
(**batchController.listDataSourceFiles**)
Returns a list of data sets that are available for consumption.

post /batch/{data}/{agreementId}
(**batchController.getBatch**)
Requests data from a provider in the form of batch.

- **Request – First Block**
- **Request – Intermediate Block**
- **Request – Last Block**

post /batch/pop

Stream controller:

post /newdata/{offeringId}
(**streamController.newData**)
Endpoint for the data soure to send data to the mqtt broker.

Agreement controller:

post /agreement/payMarketFee/{agreementID}
(**agreementController.payMarketFee**)
Endpoint the consumer can use to pay the market fee.

post /agreement/deployRawPaymentTransaction/{agreementId}
(**agreementController.deployRawPaymentTransaction**)
Endpoint for the consumer to deploy a transaction obtained by signing a transaction object that resulted by paying the market fee.

get /agreement/getAgreementId /{exchangeId}
(**agreementController.getAgreementId**)
Endpoint to retrieve the agreementId.

post /agreement/dataExchangeAgreementInfo
(**agreementController.dataExchangeAgreementInfo**)
Endpoint for the provider to post information relevant for the data transfer.

Connector registration controller:

post /regds
(connectorRegistrationController.regds)
Endpoint used by the provider to register the batch or stream data connectors.

Stream auth controller:

post /stream/auth/user

(**streamAuthController.authStreamUser**)
Endpoint to authenticate data transfer stream user.

post /stream/auth/acl
(**streamAuthController.authStreamAcl**)
Endpoint to check if the topic subscribed by the consumer matches a pre-set description standard.

OIDC auth controller:

get /oidc/login/provider
(**oidcAuthController.oidcLoginProvider**)
Endpoint to retrieve a bearer token as provider.

get /oidc/login/consumer
(**oidcAuthController.oidcLoginConsumer**)
Endpoint to retrieve a bearer token as consumer.

get /oidc/cb
(**oidcAuthController.oidcCb**)
Endpoint that will be called after a successful authentication as either a consumer or a provider.

Data transfer report controller:

post /report/nrpCompletnessCheck
(**dataTransferReportController.nrpCompletenessCheck**)
Endpoint to check if the Non-repudiable Protocol was completed for a block of data.

get /report/getListOfVerificationRequests/{agreementId}
(**dataTransferReportController.getListOfVerificationRequests**)
Endpoint to get all the verification requests for an agreement.

get /report/getSubId/{consumerDid}/{offeringId}
(**dataTransferReportController.getSubId**)
Endpoint to get the subscription ID.

get /report/streamingAccountReport/{subId}
(**dataTransferReportController.streamingAccountReport**)
Endpoint for a consumer to get information about a subscription.

get /report/getAccountSummary /{consumerDid}

(**dataTransferReportController.getAccountSummary**)
Endpoint to get information about the amount of data transferred for a consumer.

10.2 Background Technologies

Loopback:
Loopback is an open-source solution developed by StrongLoop, an IBM company. It is a framework that enables you to create dynamic end-to-end APIs (RESTful and GraphQL). It is for Node.js and developed in TypeScript, a typed superset of JavaScript. Due to its modular connectors, it can (indeed does) support any DB as well as custom data integrations like blockchains.

This is the technology that was selected for the implementation of the data access API.

10.3 Notifications Manager

The notification manager is the service responsible for allowing the creation and emission of notifications both for users and between services; it also integrates the functionality to allow users to subscribe to topics (categories) of the offers, with the objective of receiving notifications when new offers are created that coincide with those subscribed by the user.

On the other hand, it is possible to create notifications for a certain user and store them, access the stored notifications, mark a notification as read, delete it, etc.

10.4 Notifications as a Service

It allows you to send notifications to other services that are not directly connected to our service, and we do not necessarily know who they are. This happens when, for example, a new offer is created, and a request is sent to create a notification to the service and notify the rest of the services/marketplaces that have subscribed to the queue.

This section can be understood from two points of view:

- The first is that of a service that wishes to send a notification to others for a certain event.
- The second is that of a service that expects to receive notifications to perform some action, such as indexing this event within the service.

For the first case, we only need to follow the section "Create Service Notification".

For the second case, we must follow the section "Services and Queues" in order to register our service within the notification manager and need to have an endpoint in our service to receive the notifications.

Services and queues:

In this section, we explain the concepts of service and queue and indicate the methods to work with them.

A service is determined by a name, to identify our service within the system, a generic endpoint where to receive notifications and a list of queues to which it is subscribed.

A queue has a name that indicates what type of event it handles, and it is possible to indicate a specific endpoint where it will send the notifications; this specific endpoint can be null and, in that case, it will use the generic endpoint of the service.

Types of queues:

The following queues have been implemented within the notification manager:

- **offering.new**
- **offering.update**
- **agreement.accepted**
- **agreement.rejected**
- **agreement.update**
- **agreement.pending**
- **agreement.termination**
- **agreement.claim**

Service management:

This section provides methods to perform the following actions:

- list all registered services;
- get the information of a service through its identifier;
- register a service;
- delete a service.

Listing registered services:
GET /services

Get the information of a service through its identifier:
GET /services/{service_id}

Registering a new service:
POST /services

Deleting a service:
DELETE /services/{service_id}

Queue management:

This section indicates the methods to carry out the following actions: * Register a queue * Get the service queues by identifier * Get the information of a specific queue by identifier * Activate-or-deactivate-a-queue * Delete a queue.

Register a queue:
Can also be called subscribe service to queue.
POST /services/{service_id}/queues

Get the service queues by identifier:
GET /services/{service_id}/queues

Obtain the information of a specific queue by identifier:
GET /services/{service_id}/queues/{queue_id}

Activate or deactivate a queue:
PATCH /services/{service_id}/queues/{queue_id}/activate
PATCH /services/{service_id}/queues/{queue_id}/deactivate

Deleting a queue:
DELETE /services/{service_id}/queues/{queue_id}

Create a service notification:
POST /notification/service
The system will search among all the registered services, those that are subscribed to the queue indicated by the receiver_id, then it will create and send a notification to its registered endpoint.

Create a service notification for a single marketplace
POST /notification/service

The system will search among all the registered services, those that are subscribed to the queue indicated by the receiver_id, and then it will create and send a notification to its registered endpoint.

10.5 Notifications to Users

User notifications are messages that are created and stored to be read by the target users. The purpose of these messages is to notify users that an event relevant to them has occurred.

This section indicates the methods to perform the following actions:

- create a user notification.
- access to notifications.
- modification of the notifications.

Create a user notification:
User notifications are created using a POST method.
POST /notification

Access to notifications:

Once a user notification has been created, it can be accessed using one of the following access methods.

Getting all stored notifications:
This method allows access to all notifications stored in the system, including their identifiers and the information contained within the *message* field.
GET /notification

Get all users unread notifications
GET /notification/unread

Get the notifications of a user (by user ID):
GET /notification/user/{user_id}

Get an unread user notification:
GET /notification/user/{user_id}/unread

Get a notification by ID:
GET /notification/{notification_id}

Modification of the notifications:
The following methods are used to make changes to notifications, such as marking them as read/unread and deleting notifications.

Mark notification as read:
PATCH /notification/{notification_id}/read

Mark notification as unread:
PATCH /notification/{notification_id}/unread

Delete notifications:
The following method is used to delete a notification:
DELETE /notification/{notification_id}

10.6 User Subscriptions

A user can subscribe to categories; these categories are the ones to which the offers registered in the semantic engine belong in order to be notified when a new offer related to the category to which the user is subscribed appears.

This indicates the methods to perform the following actions:

- Create a user-subscription.
    - To create a subscription, it is only necessary to have the identifier of the user who wants to subscribe and the category to which he/she wants to subscribe.
- Access the subscriptions.
    - Once a subscription has been created, it is possible to access it by the following methods.
- Modify subscriptions.
    - It is possible to activate/deactivate a subscription and delete it from the system.