asclepias-broker Documentation

Release 1.0.0

CERN

Feb 24, 2020

Contents

The Asclepias Broker is a web service that enables building and flexibly querying graphs of links between research outputs. It's aiming to address a couple of problems in the world of scholarly link communication, with a focus on Software citation:

Governance of the scholarly links data and metadata Storage and curation of scholarly links is a problem that cannot be easily solved in a centralized fashion. In the same manner that specialized repositories exist to facilitate research output of different scientific fields, scholarly link tracking is a task performed best by a service that specializes in a specific scientific field.

Meaningful counting of software citations Software projects (and other types of research) evolve over time, and these changes are tracked via the concept of versioning. The issue that rises is that citations to software projects end up being "diluted" throughout their versions, leading to inaccurate citation counting for the entire sotware project. Rolling-up these citations is critical to assess the impact a software project has in a scientific field.

Sharing of scholarly links across interested parties Keeping track of the incoming scholarly links for a research artifact is a difficult task that usually repositories have to individually tackle by tapping into a multitude of external services, that expose their data in different ways. Receiving "live" notifications and having a consistent format and source for these events is crucial in order to reduce complexity and provide a comprehensive view.

These problems are addressed by providing an easy to setup service that:

- Can receive and store scholarly links through a REST API
- Exposes these scholarly links through a versatile REST API
- Can connect to a network of similar services and exchange links with them

The code in this repository was funded by a grant from the Alfred P. Sloan Foundation to the American Astronomical Society (2016).

CHAPTER 1

User's Guide

This part of the documentation will show you how to get started using the Asclepias Broker.

1.1 Installation

First you need to install pipenv, it will handle the virtual environment creation for the project in order to sandbox our Python environment, as well as manage the dependencies installation, among other things.

For running dependent services, you'll need Docker (`>=18.06.1`) and Docker Compose (`>=1.22.0`), in order get things up and running quickly but also to make sure that the same versions and configuration for these services is used, independent of your OS.

Start all the dependent services using `docker-compose` (this will start PostgreSQL, Elasticsearch 6, RabbitMQ, Redis, Kibana and Flower):

```
$ docker-compose up -d
```

Note: Make sure you have enough virtual memory for Elasticsearch in Docker:

```
# Linux
$ sysctl -w vm.max_map_count=262144

# macOS
$ screen ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/tty
<enter>
linut00001:~# sysctl -w vm.max_map_count=262144
```

Next, bootstrap the instance (this will install all Python dependencies):

```
$ ./scripts/bootstrap
```

Next, create the database tables and search indices:

```
$ ./scripts/setup
```

1.1.1 Running

Start the webserver and the celery worker:

```
$ ./scripts/server
```

Start a Python shell:

```
$ ./scripts/console
```

1.1.2 Upgrading

In order to upgrade an existing instance simply run:

```
$ ./scripts/update
```

1.1.3 Testing

Run the test suite via the provided script:

```
$ ./run-tests.sh
```

1.1.4 Documentation

You can build the documentation with:

```
$ pipenv run build_sphinx
```

1.1.5 Production environment

You can simulate a full production environment using the `docker-compose.full.yml`. You can start it like this:

```
$ docker-compose -f docker-compose.full.yml up -d
```

In addition to the normal `docker-compose.yml`, this one will start:

- HAProxy (load balancer)
- Nginx (web frontend)
- uWSGI (application container)
- Celery (background task worker)

As done for local development, you will also have to run the initial setup script inside the running container:

```
$ docker-compose -f docker-compose.full.yml run --rm web ./scripts/setup
```

1.2 Configuration

Configuration for Asclepias Broker.

See also Invenio-Config for more details, like e.g. how to override configuration via environment variables or an `invenio.cfg` file.

`asclepias_broker.config.`**`SECRET_KEY = 'CHANGE_ME'`**
Flask's `SECRET_KEY`. This is an essential variable that has to be set before deploying the broker service to a production environment. It's used by Invenio/Flask in various places for encrypting/hashing/signing sessions, passwords, tokens, etc. You can generate one using:

```
python -c 'import os; print(os.urandom(32))'
```

`asclepias_broker.config.`**`SQLALCHEMY_DATABASE_URI = 'postgresql+psycopg2://asclepias:asclepia`**
SQLAlchemy database connection string.

See also SQLAlchemy's Database Urls docs.

`asclepias_broker.config.`**`SEARCH_ELASTIC_HOSTS = [{'host': 'localhost', 'port': 9200}]`**
Elasticsearch hosts configuration.

For a single-node cluster you can configure the connection via the following environment variables:

- `ELASTICSEARCH_HOST` and `ELASTICSEARCH_PORT`. `localhost` and `9200` by default respectively
- `ELASTICSEARCH_URL_PREFIX`. URL prefix for the Elasticsearch host, e.g. `es` would result in using `http://localhost:9200/es`
- `ELASTICSEARCH_USER` and `ELASTICSEARCH_PASSWORD`. Used for Basic HTTP authentication. By default not set
- `ELASTICSEARCH_USE_SSL` and `ELASTICSEARCH_VERIFY_CERTS`

For more complex multi-node cluster setups see Invenio-Search documentation.

`asclepias_broker.config.`**`REDIS_BASE_URL = 'redis://localhost:6379'`**
Redis base host URL.

Used for Celery results, rate-limiting and session storage. Can be set via the environment variable `REDIS_BASE_URL`.

`asclepias_broker.config.`**`BROKER_URL = 'amqp://guest:guest@localhost:5672/'`**
Celery broker URL.

See also Celery's documentation.

`asclepias_broker.config.`**`SENTRY_DSN = None`**
Sentry DSN for logging errors and warnings.

`asclepias_broker.config.`**`ASCLEPIAS_SEARCH_INDEXING_ENABLED = True`**
Determines if the search index will be updated after ingesting an event

1.3 Usage

Once you have completed the *Installation* you can start interacting with the broker. You should have already run the `./scripts/server` script in a separate terminal, in order to have a view of what is happening in the web and worker application logs while we execute various commands.

1.3.1 Loading data

By default the broker is initialized without having any link data stored. There are two ways to import data at the moment:

- The `asclepias-broker events load` CLI command
- The `/events` REST API endpoint

Both methods accept JSON data based on the Scholix schema. In the root directory of this repository there is an `examples` folder that contains JSON files that we will use ot import data. Specifically we will use the `examples/cornerpy-*.json` files, which contain link data related to the various versions and papers of the corner.py astrophysics software package.

Loading events from the CLI

To load some of the events via the CLI you can run the following command:

```
$ pipenv run asclepias-broker events load examples/cornerpy-1.json
```

You should then see in the terminal that your web and celery applications are running, some messages being logged. What happened is that:

1. The CLI command loads the JSON file,
2. creates an *Event* object, and
3. sends it to be processed via the *process_event()* Celery task.
4. **The task:**
    a. stores the low-level information about the events (i.e. *Identifier* and *Relationship* objects),
    b. forms *Groups* based on their `IsIdenticalTo` and `hasVersion` relations and
    c. connects them using *GroupRelationships* to form a graph.
    d. It then indexes these relationships, objects and their metadata in Elasticsearch in order to make them searchable through the REST API.

Submitting events through the REST API

To use the REST API you will first have to create a user.

```
# Create admin role to restrict access
$ pipenv run asclepias-broker roles create admin
$ pipenv run asclepias-broker users create admin@cern.ch -a --password=123456
$ pipenv run asclepias-broker roles add admin@cern.ch admin
```

Requests to the `/events` endpoint require authentication. The standard way to authenticate is via using OAuth tokens, which can be generated through the `asclepias-broker tokens create` CLI command:

```
# We store the token in the "API_TOKEN" variable for future use
$ API_TOKEN=$(pipenv run asclepias-broker tokens create \
    --name api-token \
    --user admin@cern.ch \
    --internal)
$ echo $API_TOKEN
...<generated access token>...
```

Now that we have our token, we can submit events via `curl` (or any HTTP client of your preference):

```
$ curl -kX POST "https://localhost:5000/api/events" \
    --header "Content-Type: application/json" \
    --header "Authorization: Bearer $API_TOKEN" \
    -d @examples/cornerpy-2.json
{
    "event_id": "<some-event-id>",
    "message": "event accepted"
}
```

If you pay attention to your web/celery terminal you will see similar messages to the ones that appeared when you previously loaded the data via the CLI command.

Controlling indexing

For completeness let's also import the last file, `examples/cornerpy-3.json`, but this time we'll instruct the APIs to skip the indexing part:

```
# CLI method
$ pipenv run asclepias-broker events load examples/cornerpy-3.json --no-index

# ...or...

# REST API method
$ curl -k -X POST "https://localhost:5000/api/events?noindex=1" \
    --header "Content-Type: application/json" \
    --header "Authorization: Bearer $API_TOKEN" \
    -d @examples/cornerpy-3.json
{
  "event_id": "<some-event-id>",
  "message": "event accepted"
}
```

You might want to do this in case you want to import a lot of files/events and then reindex everything afterwards (since indexing takes times as well). To reindex everything you can run:

```
# We pass the "--destroy" flag to clean the index state
$ pipenv run asclepias-broker search reindex --destroy
```

1.3.2 Querying

Now that we have loaded data into the broker we can proceed with performing REST API queries to discover what kind of relationships corner.py has with other papers/software.

Basic relationships

The most usual question one might want to answer, is how many citations does **corner.py** have. The authors of **corner.py** recommend using the JOSS paper with DOI `10.21105/joss.00024` for citations, so lets construct a query to search for all relationships of type `isCitedBy` were this DOI is involved. You can think of the following query's results as the answer to somebody asking something like `10.21105/joss.00024 isCitedBy _____`:

```
# We can see that the paper has been cited 80 times...
$ curl -k -G "https://localhost:5000/api/relationships" \
    --header "Accept: application/json" \
    -d id=10.21105/joss.00024 \
    -d scheme=doi \
    -d relation=isCitedBy \
    -d prettyprint=1
{
  "hits": {
    "hits": [ ...<Scholix-formatted links>... ],
    "total": 80
  }
}
```

This is fine, but there is an issue here: the fact that the authors of **corner.py** wanted others to cite the software in a certain way, unfortunately doesn't mean that everybody did so. We can quickly verify this by querying for citations of specific versions of **corner.py**. Let's try citations for **corner.py v1.0.2** (DOI `10.5281/zenodo.45906`):

```
# The DOI of v1.0.2 has been cited 14 times...
$ curl -k -G "https://localhost:5000/api/relationships" \
    --header "Accept: application/json" \
    -d id=10.5281/zenodo.45906 \
    -d scheme=doi \
    -d relation=isCitedBy \
    -d prettyprint=1
{
  ...
  "total": 14
  ...
}
```

For those familiar though with the history of **corner.py**, the software used to be named **triangle.py**. Let's see how many citations exist for **triangle.py v0.1.1** (DOI `10.5281/zenodo.11020`):

```
# As we can see, there are 46 citations to "triangle.py v0.1.1"...
$ curl -k -G "https://localhost:5000/api/relationships" \
    --header "Accept: application/json" \
    -d id=10.5281/zenodo.11020 \
    -d scheme=doi \
    -d relation=isCitedBy \
    -d prettyprint=1
{
  ...
  "total": 46
  ...
}
```

Grouped relationships

At this point, we can see that there is a clear issue when it comes to counting citations for software that has been through multiple versions, name changes and published papers. Perceptually though, all of these objects are just different versions of the same thing, the software **corner.py**.

The broker allows performing a query that can answer the follwing interesting question: *How many times has any version of corner.py been cited?*:

```
# Note the "group_by=version" parameter...
$ curl -k -G "https://localhost:5000/api/relationships" \
    --header "Accept: application/json" \
    -d id=10.21105/joss.00024 \
    -d scheme=doi \
    -d group_by=version \
    -d relation=isCitedBy \
    -d prettyprint=1
{
  ...
  "total": 144
  ...
}
```

Filtering

The broker's REST API also provides some basic filtering. E.g. one can find all of the citations that were performed in the year 2016:

```
# Note the "from" and "to" parameters...
$ curl -k -G "https://localhost:5000/api/relationships" \
    --header "Accept: application/json" \
    -d id=10.21105/joss.00024 \
    -d scheme=doi \
    -d group_by=version \
    -d relation=isCitedBy \
    -d from="2016-01-01" -d to="2016-12-31" \
    -d prettyprint=1
{
  ...
  "total": 50
  ...
}
```

CHAPTER 2

Architecture

This section describes the design principles of the Asclepias Broker.

2.1 Data Model

2.1.1 Events

Events are a *vessel* for data to arrive to the broker. The event data is ingested in order to be broken down to *Object Events* and eventually into *Identifiers*, *Relationships* and *Metadata*. These entities are then processed and integrated into the existing data to enhance the knowledge that the broker service possesses.

2.1.2 Graph

Identifiers represent references to scholarly entities and follow a specific identifier scheme (e.g. `DOI`, `arXiv`, `URL`, etc). *Relationships* have a type (e.g. `isIdenticalTo`, `hasVersion`, `cites`, etc.) and exist between two identifiers, the source and the target. These are the building blocks for the broker's graph model

To represent scholarly entities (software, articles, etc.), the concept of *Groups* is introduced. Groups define a set of **Identifiers** which are formed based on the **Relationships** between them. For example, one can define that all **Identifiers** that have Relationships of type `isIdenticalTo` form a Group of type `Identity` and can be considered as a single entity.

One can also define Groups of Groups. For example `Identity` Groups with Identifiers that have a `hasVersion` relationship with Identifiers from other `Identity` Groups, can form a `Version` Group.

One can then finally model relationships between scholarly entities (e.g. *Paper A cites Software X*), by abstracting the low-level Relationships between Identifiers to the Group level and thus form *Group Relationships*. For

example, one can define that `Identity` Groups of Identifiers that have Relationships of type `cites` to Identifiers of other `Identity` Groups, can form a `cites` Group Relationship.

2.1.3 Metadata

Identifiers, Relationships and Groups can form complex graphs. While this is important for discovering connections between them, it is also valuable to be able to retrieve information about the objects they hold references to. In order to facilitate this information, *`Group Metadata`* and *`Group Relationship Metadata`* is stored for **Groups** and **Group Relationships** respectively.

This metadata can be used for e.g. rendering a proper citation when needed, filtering.

2.1.4 Persistence

As described in the previous sections, the broker receives raw events that are then processed to produce a graph. The data goes through a transformation pipeline that at various stages requires persisting its inputs and outputs. This persistence takes place in an RDBMS, like PostgreSQL or SQLite.

We can divide the persisted information into three incremental levels:

A) Raw data The raw event payloads that arrive into the system

B) Ground truth Normalized form of the raw data, representing deduplicated facts about Identifiers and Relationships

C) Processed knowledge Information that is extracted from the ground truth and is transformed into structured knowledge

Each level depends on all of its predecessors. This means that if there is an issue on e.g. level C, levels A and B are enough to rebuild it. In the same fashion, level B depends only on level A.

Note: All of the above models map to actual database schema tables. For the sake of clarity though, intermediary tables that represent many-to-many relationships between these models (e.g. *`GroupM2M`* for `Group <-> Group` relationships) were not included.

2.1.5 Search

Now that we have the above information stored persistently in the system, we need an efficient way to perform queries over it. Doing this directly through the database would seem like a practical, although naive, solution for fetching this information. Our graph representation spreads over many tables, which means that fetching it would require multiple complex joins. On top of that our metadata is stored in JSON/blob-like columns, where filtering is slow and inefficient.

The way to tackle this issue, is to denormalize our data back into a rich document representation that clients of the service can consume with ease. This can be easily done via the use of a document-based store (aka *NoSQL*) system, like Elasticsearch.

We can create and index the documents using the following strategy:

1. **For each Group Relationship in our system:**

a. Fetch its **Group Relationships Metadata**

b. **For its source and target groups:**

   i. Fetch the **Group Metadata** and **Identifiers**

c. Create a document from the fetched information and index it

By performing the expensive database queries only once in order to index the denormalized documents we have managed to get the best of both worlds: a relationally consistent graph (backed by RDB constraints) which is easy to perform complex queries over (backed by Elasticsearch).

Consistency

A downside to this solution is that the state of our document store is not always in sync with what we have in our graph in the database. This issue originates from the fact that changes in the database are automatically protected via foreign-key and unique constraints that cannot be applied with the same ease in a document-based store.

A solution to this is to periodically rebuild the entire index from scratch. This guarantees that Elasticsearch starts from a blank state, with no "orphan" or stale information lying around. Also, using some of Elasticsearch's features this index rebuilding process can be achieved without affecting the responsiveness of the service.

2.2 Systems

Although the broker is considered a single service, to provide its functionality it is split into applications and dependend services.

2.2.1 Applications

Web (Flask/Invenio)

The web service is a Flask application using the Invenio framework. It handles the REST API endpoints that receive and serve the data that the broker processes and stores. It exposes two endpoints:

`/events` Used for receiving Scholix link data in the form of events in JSON format

`/relationships` Used for serving user queries for information regarding the graph, its objects and their metadata

Worker (Celery/Invenio)

The worker service is responsible for asynchronously running background tasks that are initiated either via the web application or periodically (in a UNIX `cron`-like fashion). These tasks usually involve processing events, updating the graph state and indexing documents on Elasticsearch.

2.2.2 Services

The broker's applications depend on some other services used mainly for sotrage and operational purposes.

PostgreSQL

This is the database of our choice that persistently stores the raw events and graph state we described in the Data model section.

Elasticsearch

This is the searchable document store that is used to store denormalized information from the graph.

RabbitMQ

This is the message queue used for sending tasks to Celery from the web application.

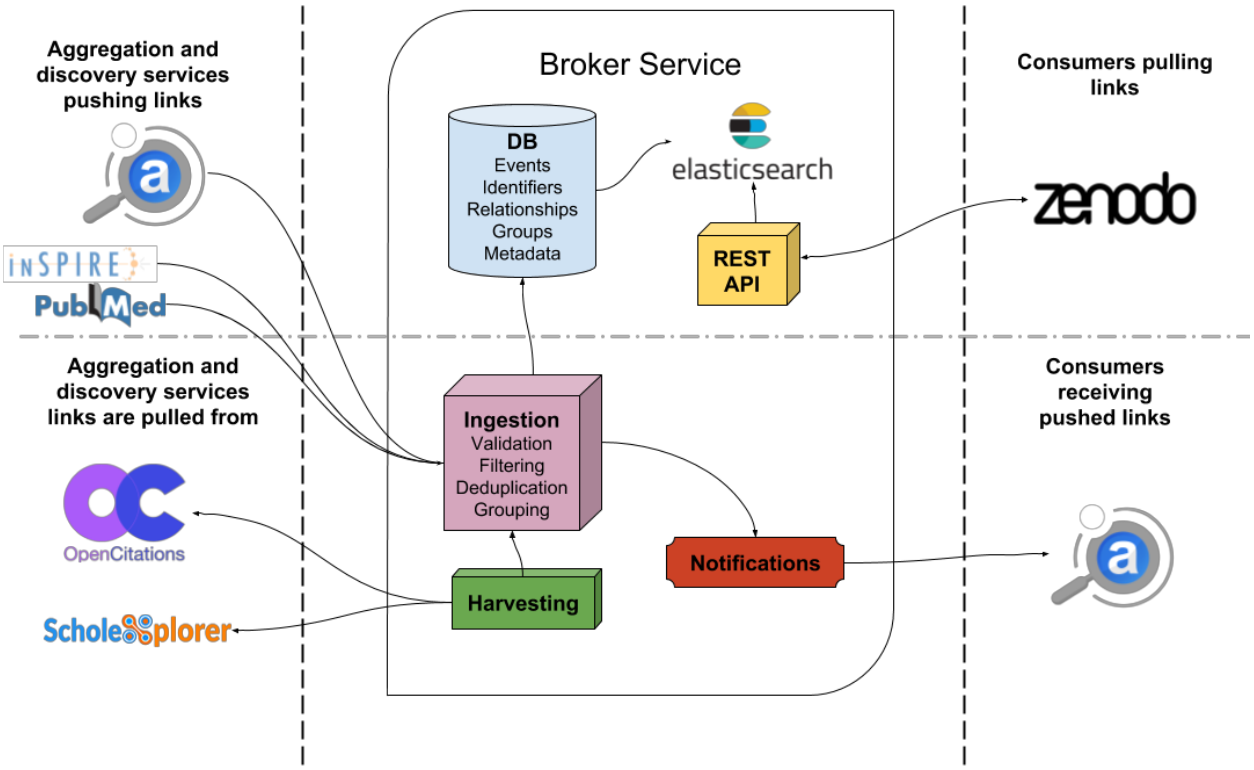
Redis

This is a key-value store used for various short-lived storage purposes. The web application uses its for storing rate-limiting information. The worker uses it for storing task state, task results and as a temporary cache that.

2.3 Interoperability

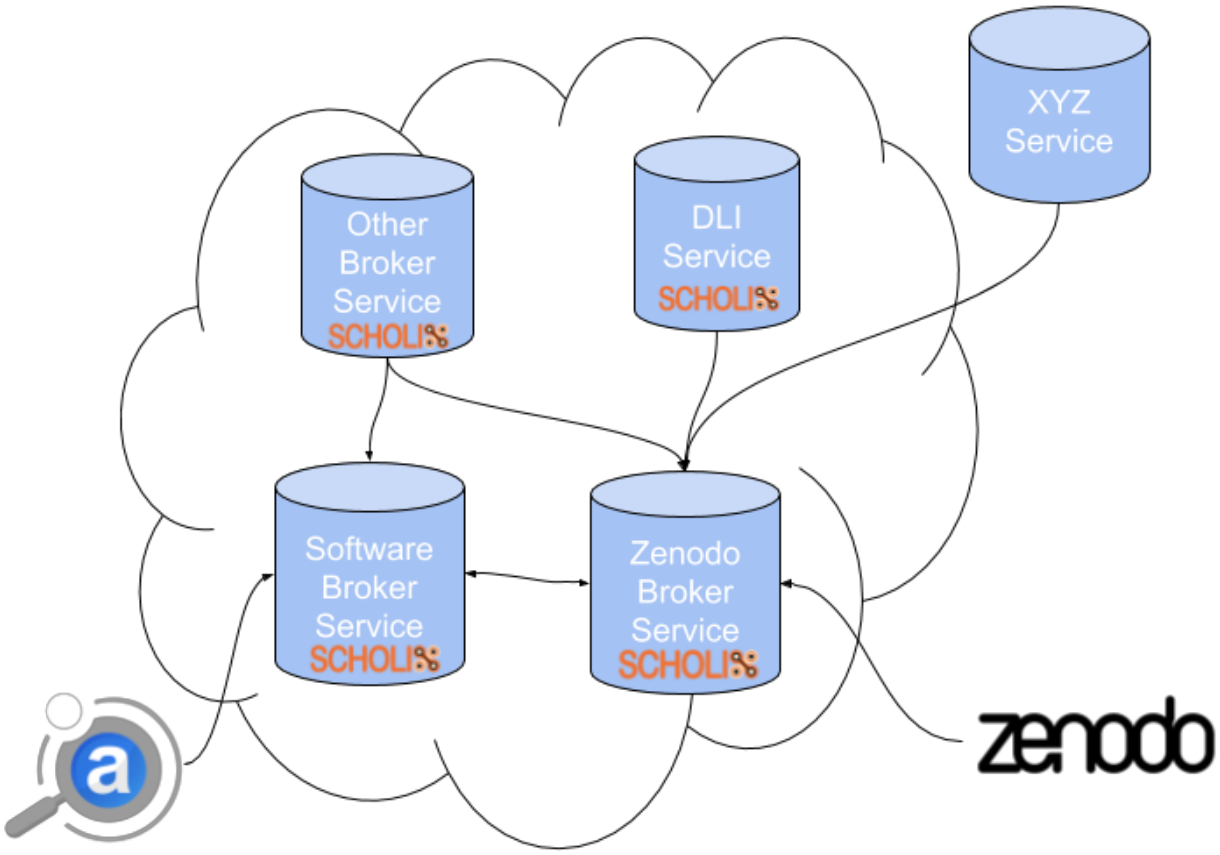
The Broker service is populating its links store through a variety of different methods. Some of these are accepting data that is pushed from trusted external sources while others actively harvest data from publicly available sources.

The common denominator between these methods is the format of the link data which is based on Scholix. Any data input that arrives to the Broker is either pre-processed to comply or already complies with the Scholix schema. This allows for a robust ingestion pipeline which can keep track of the underlying scholarly objects, their identifiers and the relationships between them.



On a higher level, Broker services aim at achieving interoperability with other brokers or services which at this point were not able to formally communicate with each other and exchange information. The goal is to build a Broker Network where each participant focuses on maintaining the information that he has the best domain knowledge on and ability to extract the most quiality information from. This way the quality of the information is kept high and concentrated in separate parts of the network, avoiding redundant information that is unused, but at the same time allowing individual participants to decide about what kind of information they want to store depending on their needs. In its entirety, the network can then hold vast amounts of high quality knowledge in a distributed manner.

For example, ADS has a specialized ingestion workflow for extracting various forms of citations and references to software packages related to astrophysics. For that reason it would make more sense for ADS to push this knowledge to a *Software Broker* and not a *Humanities & Social Sciences Broker*.


XYZ
Service
Other
Broker
Service
SCHOLIX
DLI
Service
SCHOLIX
Software
Broker
Service
SCHOLIX
Zenodo
Broker
Service
SCHOLIX
zenodo

CHAPTER 3

REST API

This section documents the REST APIs exposed by the service.

3.1 REST API

The broker service currently exposes two REST API endpoints, one for the ingestion of the

3.1.1 Events

`POST /events`

Submit an array of Scholix relationships to be ingested by the broker. This endpoint requires API token authentication in order to identify the user/source that submitted the events and to protect the service from spam.

Example request:

Submit Scholix relationships describing:

- `(2017ascl.soft02002F) IsIdenticalTo (10.21105/joss.00024)`
- `(2017JOSS.2017..188X) References (10.21105/joss.00024)`

```
POST /events HTTP/1.1
Authorization: Bearer <...API Token...>
Content-Type: application/x-scholix-v3+json

[
  {
    "Source": {
      "Identifier": {"ID": "2017ascl.soft02002F", "IDScheme": "ads"},
      "Type": {"Name": "software"}
    },
    "RelationshipType": {
      "Name": "IsRelatedTo",
```

(continues on next page)

(continued from previous page)


```
      "SubType": "IsIdenticalTo",
      "SubTypeSchema": "DataCite"
    },
    "Target": {
      "Identifier": {"ID": "10.21105/joss.00024", "IDScheme": "doi"},
      "Type": {"Name": "software"}
    },
    "LinkProvider": [{"Name": "Zenodo"}],
    "LinkPublicationDate": "2018-01-01"
  },
  {
    "Source": {
      "Identifier": {"ID": "2017JOSS.2017..188X", "IDScheme": "ads"},
      "Type": {"Name": "unknown"}
    },
    "RelationshipType": {"Name": "References"},
    "Target": {
      "Identifier": {"ID": "10.21105/joss.00024", "IDScheme": "doi"},
      "Type": {"Name": "software"}
    },
    "LinkProvider": [{"Name": "SAO/NASA Astrophysics Data System"}],
    "LinkPublicationDate": "2017-04-01"
  }
]
```


Example response:

```
HTTP/1.1 202 OK
Content-Type: application/json

{
  "message": "event accepted",
  "event_id": "69270574-7cf4-477b-9b20-84554bb7032b"
}
```

Request Headers

- Authorization – API token to authenticate.

Status Codes

- 202 Accepted – Event received successfully

3.1.2 Relationships

`POST /relationships`

Search for relationships of a specific identifier.

Example request:

Find `isCitedBy` relationships towards `10.5281/zenodo.53155`:

```
GET /relationships?id=10.5281/zenodo.53155&scheme=doi&relation=isCitedBy HTTP/1.1
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/x-scholix-v3+json

{
  "Source": {
    "Title": "corner.py v2.0.0",
    "Identifiers": [
      {"ID": "10.5281/zenodo.53155", "IDScheme": "doi"},
      {"ID": "https://zenodo.org/record/53155", "IDScheme": "url"},
      {"ID": "https://github.com/dfm/corner.py/tree/v2.0.0", "IDScheme": "url"}
    ],
    "Creator": [{"Name": "Dan Foreman-Mackey"}, {"Name": "Will Vousden"}],
    "Type": {"Name": "software"},
    "PublicationDate": "2016-05-26"
  },
  "Relation": {"Name": "isCitedBy"},
  "GroupBy": "identity",
  "Relationships": [
    {
      "Target": {
        "Title": "The mass distribution and gravitational...",
        "Type": {"Name": "literature"},
        "Identifiers": [
          {"ID": "10.1093/mnras/stw2759", "IDScheme": "doi"},
          {"ID": "https://doi.org/10.1093/mnras/stw2759", "IDScheme": "url"},
        ],
        "Creator": [{"Name": "Paul J. McMillan"}],
        "PublicationDate": "2016-10-26"
      },
      "LinkHistory": [
        {
          "LinkPublicationDate": "2016-12-01",
          "LinkProvider": {"Name": "Zenodo"}
        },
        {
          "LinkPublicationDate": "2016-10-28",
          "LinkProvider": {"Name": "ADS"}
        }
      ]
    },
    {
      "Target": {
        "Title": "PROBABILISTIC FORECASTING OF THE MASSES...",
        "Identifiers": [
          {"ID": "10.3847/1538-4357/834/1/17", "IDScheme": "doi"},
          {"ID": "https://doi.org/10.3847/1538-4357/834/1/17", "IDScheme": "url"}
        ],
        "Creator": [{"Name": "Jingjing Chen"}, {"Name": "David Kipping"}],
        "PublicationDate": "2016-12-27"
      },
      "LinkHistory": [
        {
          "LinkPublicationDate": "2016-12-30",
          "LinkProvider": {"Name": "ADS"}
        }
      ]
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

Query Parameters

- **id** – Value of source identifer **(required)**. Example: `10.5281/zenodo.1120265`
- **scheme** – Identifier scheme of the source identifier. Example: `doi`, `arxiv`, `url`
- **relation** – Filter by type of the relation between source and target identifiers. Accepted values: `cites`, `isCitedBy`, `isSupplementTo`, `isSupplementedBy`, `isRelatedTo`
- **type** – Filter by type of the target objects. Accepted values: `literature`, `software`, `dataset`, `unknown`
- **publication_year** – Filter by the publication year of the target objects. Examples: `2015--<2018` (from incl. 2015 until excl. 2018), `>2005--` (from excl. 2005), `2005--2005` (all from 2005).
- **from** – Filter by start date of publication/discovery of the relationships. Example: `2018-01-02T13:30:00`
- **to** – Filter by end date of publication/discovery of the relationships. Example: `2018-01-31`
- **group_by** – Expand the scope of the relationships to source identifier (default: `identity`). Accepted values: `identity`, `version`
- **q** – Filter result using free-text search or Elasticsearch Query string syntax.
- **sort** – Sorting order of the results. At the moment `mostrecent` is the only acceptable value. To reverse the sorting order you can add a – in front (i.e. `-mostrecent` will return older relationships first).

CHAPTER 4

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

4.1 Core

Core module.

4.1.1 Models

Core database models.

class asclepias_broker.core.models.**Identifier**(***kwargs*)
Identifier model.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data
Get the metadata of the identity group the identifier belongs to.

fetch_or_create_id()
Fetches from the database or creates an id for the identifier.

classmethod get(*value=None*, *scheme=None*, ***kwargs*)
Get the identifier from the database.

get_children(*rel_type*, *as_relation=False*)
Get all children of given Identifier for given relation.

get_identities()
Get the fully-expanded list of 'Identical' Identifiers.

get_parents(*rel_type*, *as_relation=False*)
Get all parents of given Identifier for given relation.

identity_group
Get the identity group the identifier belongs to.

class asclepias_broker.core.models.**Relation**
Relation type.

class asclepias_broker.core.models.**Relationship**(***kwargs*)
Relationship between two identifiers.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

data
Get the relationship's identity group metadata.

fetch_or_create_id()
Fetches from the database or creates an id for the relationship.

classmethod get(*source*, *target*, *relation*, ***kwargs*)
Get the relationship from the database.

identity_group
Get the relationship's identity group.

4.2 Events

Events module.

4.2.1 Models

Event database models.

class asclepias_broker.events.models.**Event**(***kwargs*)
Event model.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

classmethod get(*id=None*, ***kwargs*)
Get the event from the database.

class asclepias_broker.events.models.**EventStatus**
Event status.

class asclepias_broker.events.models.**ObjectEvent**(***kwargs*)
Event related to an Identifier or Relationship.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

object
Get the associated Identifier or Relationship.

Return type Union[*Identifier*, *Relationship*]

class asclepias_broker.events.models.**PayloadType**
Payload type.

4.2.2 API

Events API.

class asclepias_broker.events.api.**EventAPI**
Event API.

classmethod handle_event(*event*, *no_index=False*, *user_id=None*, *eager=False*)
Handle an event payload.

Return type *Event*

4.2.3 Views

Event views.

class asclepias_broker.events.views.**EventResource**
Event resource.

post()
Submit an event.

4.2.4 CLI

Events CLI.

asclepias-broker events

Event CLI commands.

```
asclepias-broker events [OPTIONS] COMMAND [ARGS]...
```

load

Load events from a directory.

```
asclepias-broker events load [OPTIONS] JSONDIR_OR_FILE
```

Options

--no-index

-e, --eager

Arguments

JSONDIR_OR_FILE
Required argument

4.2.5 Errors

Errors and exceptions.

exception asclepias_broker.events.errors.**PayloadValidationRESTError**(*error_message*, *code=None*, ***kwargs*)

Invalid payload error.

Initialize the PayloadValidation REST exception.

4.3 Graph

Graph module.

4.3.1 Models

Graph database models.

class asclepias_broker.graph.models.**Group**(***kwargs*)
Group model.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class asclepias_broker.graph.models.**GroupM2M**(***kwargs*)
Many-to-many model for Groups.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class asclepias_broker.graph.models.**GroupRelationship**(***kwargs*)
Group relationship model.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class asclepias_broker.graph.models.**GroupRelationshipM2M**(***kwargs*)
Many-to-many model for Group Relationships.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class asclepias_broker.graph.models.**GroupType**
Group type.

class asclepias_broker.graph.models.**Identifier2Group**(***kwargs*)
Many-to-many model for Identifier and Group.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class asclepias_broker.graph.models.**Relationship2GroupRelationship**(***kwargs*)
Many-to-many model for Relationship to GroupRelationship.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

4.3.2 API

Graph functions.

asclepias_broker.graph.api.**add_group_relationship**(*relationship*, *src_id_grp*, *tar_id_grp*, *src_ver_grp*, *tar_ver_grp*)
Add a group relationship between corresponding groups.

asclepias_broker.graph.api.**delete_duplicate_group_m2m**(*group_a*, *group_b*)
Delete any duplicate GroupM2M objects.

Removes one of each pair of GroupM2M objects for groups A and B.

asclepias_broker.graph.api.**delete_duplicate_relationship_m2m**(*group_a*, *group_b*, *cls=<class 'asclepias_broker.graph.models.GroupRelationshi*
Delete any duplicate relationship M2M objects.

Deletes any duplicate (unique-constraint violating) M2M objects between relationships and group relationships. This step is required before merging of two groups.

asclepias_broker.graph.api.**get_group_from_id**(*identifier_value*, *id_type='doi'*, *group_type=<GroupType.Identity: 1>*)
Resolve from 'A' to Identity Group of A or to a Version Group of A.

Return type *Group*

asclepias_broker.graph.api.**get_or_create_groups**(*identifier*)
Given an Identifier, fetch or create its Identity and Version groups.

Return type Tuple[*Group*, *Group*]

asclepias_broker.graph.api.**merge_group_relationships**(*group_a*, *group_b*, *merged_group*)
Merge the relationships of merged groups A and B to avoid collisions.

Parameters

- **group_a** (*Group*) – some things.
- **group_b** (*Group*) – some other things.

Groups 'group_a' and 'group_b' will be merged as 'merged_group'. This function takes care of moving any duplicate group relations, e.g.:

If we have 4 relations:

- A Cites X
- B Cites X
- Y Cites A
- Y Cites B

and we merge groups A and B, we also need to squash the first two and last two relations together:

- {AB} Cites X
- Y Cites {AB}

before we can perform the actual marging of A and B. Otherwise we will violate the unique constraint. We do that by removing the duplicate relationships (only one of each duplicate pair), so that we can later execute and UPDATE.

asclepias_broker.graph.api.**merge_identity_groups**(*group_a*, *group_b*)
Merge two groups of type "Identity".

Merges the groups together into one group, taking care of migrating all group relationships and M2M objects.

Return type Tuple[Optional[*Group*], Optional[*Group*]]

asclepias_broker.graph.api.**merge_version_groups**(*group_a*, *group_b*)
Merge two Version groups into one.

Return type Optional[*Group*]

asclepias_broker.graph.api.**update_groups**(*relationship*, *delete=False*)
Update groups and related M2M objects for given relationship.

Return type Tuple[Tuple[*Group*, *Group*, *Group*], Tuple[*Group*, *Group*, *Group*]]

4.3.3 Tasks

Asynchronous tasks.

asclepias_broker.graph.tasks.**compact_indexing_groups**(*groups_ids*)
Compact the collected group IDs into minimal set of UUIDs.

Return type Tuple[Set[str], Set[str], Set[str], Set[str], Dict[str, str]]

asclepias_broker.graph.tasks.**create_relation_object_events**(*event*, *relationship*, *payload_idx*)
Create the object event models.

asclepias_broker.graph.tasks.**get_or_create**(*model*, ***kwargs*)
Get or a create a database model.

(task) asclepias_broker.graph.tasks.**process_event**(*event_uuid: str*, *indexing_enabled: bool=True*)
Process the event.

4.4 Metadata

Metadata module.

4.4.1 Models

Metadata database models.

class asclepias_broker.metadata.models.**GroupMetadata**(***kwargs*)
Metadata for a group.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

update(*payload*, *validate=True*)
Update the metadata of a group.

class asclepias_broker.metadata.models.**GroupRelationshipMetadata**(***kwargs*)
Metadata for a group relationship.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

update(*payload*, *validate=True*, *multi=False*)
Updates the metadata of a group relationship.

4.4.2 API

Metadata functions.

asclepias_broker.metadata.api.**update_metadata**(*identifier*, *scheme*, *data*, *create_identity_events=True*, *create_missing_groups=True*, *providers=None*, *link_publication_date=None*)
.

asclepias_broker.metadata.api.**update_metadata_from_event**(*relationship*, *payload*)
Updates the metadata of the source, target and relationship groups.

4.4.3 CLI

CLI for Asclepias broker.

asclepias-broker metadata

Metadata CLI commands.

```
asclepias-broker metadata [OPTIONS] COMMAND [ARGS]...
```

load

Load events from a directory.

```
asclepias-broker metadata load [OPTIONS] JSONDIR
```

Arguments

JSONDIR
Required argument

4.5 Search

Search module.

4.5.1 CLI

CLI for Asclepias broker search module.

asclepias-broker search

Utility CLI commands.

```
asclepias-broker search [OPTIONS] COMMAND [ARGS]...
```

reindex

Reindex all relationships.

```
asclepias-broker search reindex [OPTIONS]
```

Options

--destroy

--split, --no-split

-e, --eager

--yes
Confirm the action without prompting.

4.5.2 API

Relationships search API.

class asclepias_broker.search.api.**RelationshipAPI**
Relationship API.

classmethod get_citations(*identifier*, *with_parents=False*, *with_siblings=False*, *expand_target=False*)
Get citations of an identfier from the database.

classmethod get_citations2(*identifier*, *relation*, *grouping_type=<GroupType.Identity: 1>*)
Get citations of an identfier from the database.

classmethod print_citations(*pid_value*)
Print citations of an identifier.

4.5.3 Indexer

Elasticsearch indexing module.

asclepias_broker.search.indexer.**build_doc**(*rel*, *src_grp=None*, *trg_grp=None*, *grouping=None*)
Build the ES document for a relationship.

Return type dict

asclepias_broker.search.indexer.**build_group_metadata**(*group*)
Build the metadata for a group object.

Return type dict

asclepias_broker.search.indexer.**build_id_info**(*id_*)
Build information for the Identifier.

Return type dict

asclepias_broker.search.indexer.**build_relationship_metadata**(*rel*)
Build the metadata for a relationship.

Return type dict

asclepias_broker.search.indexer.**delete_group_relations**(*group_ids*)
Delete all relations for given group IDs from ES.

asclepias_broker.search.indexer.**index_documents**(*docs*, *bulk=False*)
Index a list of documents into ES.

asclepias_broker.search.indexer.**index_identity_group_relationships**(*ig_id*, *vg_id*, *exclude_group_ids=None*)

Build the relationship docs for Identity relations.

asclepias_broker.search.indexer.**index_version_group_relationships**(*group_id*, *exclude_group_id=None*)

Build the relationship docs for Version relations.

asclepias_broker.search.indexer.**update_indices**(*idx_ig*, *del_ig*, *idx_vg*, *del_vg*, *ig_to_vg_map*)

Updates Elasticsearch indices with the updated groups.

4.5.4 Query

Search utilities.

asclepias_broker.search.query.**enum_term_filter**(*label*, *field*, *choices*)

Term filter with controlled vocabulary.

asclepias_broker.search.query.**nested_range_filter**(*label*, *field*, *path=None*, *op=None*)

Nested range filter.

asclepias_broker.search.query.**nested_terms_filter**(*field*, *path=None*)

Nested terms filter.

asclepias_broker.search.query.**search_factory**(*self*, *search*, *query_parser=None*)

Parse query using elasticsearch DSL query.

Parameters

- **self** – REST view.
- **search** – Elastic search DSL search instance.

Returns Tuple with search instance and URL arguments.

4.5.5 Tasks

Asynchronous tasks.

(task) asclepias_broker.search.tasks.**reindex_all_relationships**(*destroy: bool=False*, *split: bool=True*)

Reindex all relationship documents.

4.5.6 Views

Search views.

asclepias_broker.search.views.**citations**(*pid_value*)

Renders all citations for an identifier.

asclepias_broker.search.views.**relationships**()

Renders relationships for an identifiers from DB.

CHAPTER 5

Additional Notes

Notes on how to contribute, legal information and changes are here for the interested.

5.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/asclepias/asclepias-broker/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Asclepias Broker could always use more documentation, whether as part of the official Asclepias Broker docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/asclepias/asclepias-broker/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.1.2 Get Started!

Ready to contribute? Here's how to set up *asclepias-broker* for local development.

1. Fork the *asclepias/asclepias-broker* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/asclepias-broker.git
```

3. Assuming you have `pipenv`, `docker` and `docker-compose` installed, this is how you set up your fork for local development:

```
$ ./scripts/bootstrap
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass tests:

```
$ ./run-tests.sh
```

   The tests will provide you with test coverage and also check PEP8 (code style), PEP257 (documentation), flake8 as well as build the Sphinx documentation and run doctests.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "component: summarize changes in 50 chars or less

* More detailed explanatory text, if necessary. Formatted using
  bullet points, preferably `*`. Wrapped to 72 characters.
* Explain the problem that this commit is solving. Focus on why you
  are making this change as opposed to how (the code explains that).
  Are there side effects or other unintuitive consequences of this
  change? Here's the place to explain them.
* The blank line separating the summary from the body is critical
  (unless you omit the body entirely); various tools like `log`,
```

(continues on next page)

(continued from previous page)

```
  `shortlog` and `rebase` can get confused if you run the two
  together.
* Use words like "Adds", "Fixes" or "Breaks" in the listed bullets to
  help others understand what you did.
* If your commit closes or addresses an issue, you can mention
  it in any of the bullets after the dot. (closes #XXX) (addresses #YYY)"

$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.1.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests and must not decrease test coverage.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring.
3. Check https://travis-ci.org/asclepias/asclepias-broker/pull_requests and make sure that all tests pass.

5.2 Changes

5.3 License

Note: In applying this license, CERN does not waive the privileges and immunities granted to it by virtue of its status as an Intergovernmental Organization or submit itself to any jurisdiction.

5.4 Authors

- Alexander Ioannidis
- Chiara Bigarella
- Krzysztof Nowak
- Thomas P. Robitaille

Python Module Index

a

HTTP Routing Table

/events

/relationships

Index

Symbols

A

B

C

D

E

F

G

H

I

J