BioThings SDK Documentation

Release 0.1

BioThings team

Jun 05, 2018

Tutorials

CHAPTER 1

Introduction

1.1 What's BioThings?

We use "**BioThings**" to refer to objects of any biomedical entity-type represented in the biological knowledge space, such as genes, genetic variants, drugs, chemicals, diseases, etc.

1.2 BioThings SDK

SDK represents "Software Development Kit". BioThings SDK provides a Python-based toolkit to build high-performance data APIs (or web services) from a single data source or multiple data sources. It has the particular focus on building data APIs for biomedical-related entities, a.k.a "*BioThings*", though it's not necessarily limited to the biomedical scope. For any given "*BioThings*" type, BioThings SDK helps developers to aggregate annotations from multiple data sources, and expose them as a clean and high-performance web API.

The BioThings SDK can be roughly divided into two main components: data hub (or just "hub") component and web component. The hub component allows developers to automate the process of monitoring, parsing and uploading your data source to an Elasticsearch backend. From here, the web component, built on the high-concurrency Tornado Web Server , allows you to easily setup a live high-performance API. The API endpoints expose simple-to-use yet powerful query features using Elasticsearch's full-text query capabilities and query language.

1.3 BioThings API

We also use "*BioThings API*" (or *BioThings APIs*) to refer to an API (or a collection of APIs) built with BioThings SDK. For example, both our popular MyGene.Info and MyVariant.Info APIs are built and maintained using this BioThings SDK.

CHAPTER 2

Installing BioThings SDK

You can install the latest stable BioThings SDK release with pip from PyPI, like:

```
pip install biothings
```

You can install the latest development version of BioThings SDK directly from our github repository like:

```
pip install git+https://github.com/biothings/biothings.api.git#egg=biothings
```

Alternatively, you can download the source code, or clone the BioThings SDK repository and run:

```
python setup.py install
```

2.1 Single Data Source, No Source Updating Tutorial

The following tutorial shows a minimal use case for the BioThings SDK: creating a high-performance, high-concurrency API from a single flat-file. The BioThings SDK is broadly divided into two components, the hub and the web. The hub component is a collection of tools to automate the downloading of source data files, the merging of different sources, and the updating of the Elasticsearch index. The web component is a Tornado-based API app that subsequently serves data from this Elasticsearch index.

2.1.1 Prerequisites

Before starting, there are a few requirements that need to be installed and configured.

Python

The BioThings SDK requires Python version 3.4 or higher for full functionality. We recommend installing all python dependencies into a virtualenv.

BioThings SDK

Either install from source, like:

```
git clone https://github.com/biothings/biothings.api.git
cd biothings.api
python setup.py install
```

or use pip, like:

```
pip install biothings
```

or directly from our repository, like:

```
pip install git+https://github.com/biothings/biothings.api.git#egg=biothings
```

Elasticsearch

BioThings APIs currently serve data from an Elasticsearch index, so Elasticsearch is a requirement. Install Elasticsearch 2.4 either directly, or as a docker container.

Configure Elasticsearch

To configure Elasticsearch, execute the following commands as su:

```
echo 'http.enabled: True' >> /etc/elasticsearch/elasticsearch.yml
echo 'network.host: "0.0.0.0"' >> /etc/elasticsearch/elasticsearch.yml
```

Note: This guide was created using Ubuntu 16.04, the exact location of elasticsearch.yml may vary in other platforms.

2.1.2 PharmGKB Gene

Once all prerequisites have been installed and Elasticsearch is running, the data loading step can begin. Consider the following script, which defines a "load_data" function that parses the PharmGKB gene flat file and then iterates through it, storing the results in an Elasticsearch index using biothings.utils.es.ESIndexer.

```
In [1]: import re

In [2]: from biothings.utils.es import ESIndexer

In [3]: def load_data(f):
   ...:     for (i, line) in enumerate(f):
   ...:         line = line.strip('\n')
   ...:         if i == 0:  # get the column header names in the first row
   ...:             header_dict = dict([(p, re.sub(r'\s', '_', h.lower())) for (p, h)␣
↪in enumerate(line.split('\t'))])
   ...:         else:
   ...:             _r = {}
   ...:             for (pos, val) in enumerate(line.split('\t')):
   ...:                 if val:
   ...:                     _r[header_dict[pos]] = val if '","' not in val else val.
↪strip('"').split('","')
```

(continues on next page)

(continued from previous page)

```
   ...:             yield _r
   ...:

In [4]: indexer = ESIndexer(index='pharmgkb_gene_current', doc_type='pharmgkb_gene',␣
↪es_host='localhost:9200')

In [5]: indexer.create_index(mapping={'pharmgkb_gene':{'dynamic': True}})

In [6]: with open('genes.tsv', 'r') as gene_file:
   ...:     indexer.index_bulk(load_data(gene_file))
```

2.1.3 Generate BioThings API

Now that we have an Elasticsearch index with our indexed gene data in it, we can create and start an API. Change to a directory you want to store the front-end code, and type:

```
biothings-admin.py pharmgkb_gene . -o src_package=pharmgkb_gene
```

Now you can start your API by typing:

```
cd pharmgkb_gene/src
pip install -r ../requirements_web.txt
python www/index.py --debug --port=8001
```

Your API is live. To use it, you can query it with a curl (or your local browser). For example, if you wanted to find the PharmGKB accession for an NCBI gene (or gene list) you have, you could do a query like:

```
curl "http://localhost:8001/v1/query?q=ncbi_gene_id:1017&fields=pharmgkb_accession_id"
{
  "max_score": 8.178926,
  "took": 9,
  "total": 1,
  "hits": [
    {
      "_id": "AVydiHIJYMgArMwkfE8R",
      "_score": 8.178926,
      "pharmgkb_accession_id": "PA101"
    }
  ]
}
```

Or, to find all PharmGKB genes that have a CDK* symbol, you can do this query:

```
curl "http://localhost:8001/v1/query?q=symbol:CDK*&fields=pharmgkb_accession_id,symbol
↪"
{
  "max_score": 1.0,
  "took": 11,
  "total": 50,
  "hits": [
    {
      "_id": "AVydiHIJYMgArMwkfE8F",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA99",
      "symbol": "CDK1"
```

(continues on next page)

(continued from previous page)


```
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8H",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA26263",
      "symbol": "CDK11A"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8M",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA165696414",
      "symbol": "CDK15"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8R",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA101",
      "symbol": "CDK2"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8n",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA26317",
      "symbol": "CDKL1"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8N",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA33095",
      "symbol": "CDK16"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8e",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA38632",
      "symbol": "CDK5RAP2"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8h",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA26314",
      "symbol": "CDK7"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8m",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA134871999",
      "symbol": "CDKAL1"
    },
    {
      "_id": "AVydiHIJYMgArMwkfE8v",
      "_score": 1.0,
      "pharmgkb_accession_id": "PA106",
      "symbol": "CDKN2A"
    }
  ]
}
```

2.2 Multiple Data Sources, Automated Source Updating Tutorial

In this tutorial, we will build the whole process, or "hub", which produces the data for Taxonomy BioThings API, accessible at <*t.biothings.io*>. This API serves information about species, lineage, etc... This "hub" is used to download, maintain up-to-date, process, merge data. At the end of this process, an Elasticsearch index is created containing all the data of interest, ready to be served as an API, using Biothings SDK Web component (covered in another tutorial). Taxonomy Biothings API code is avaiable at <*https://github.com/biothings/biothings.species*>

2.2.1 Prerequesites

BioThings SDK uses MongoDB as the "staging" storage backend for JSON objects before they are sent to Elasticsearch for indexing. You must a have working MongoDB instance you can connect to. We'll also perform some basic commands. You must also have BioThings SDK installed (`git clone https://github.com/biothings/biothings.api.git` is usually enough, followed by `pip install -r requirements.txt`). You may want to use `virtualenv` to isolate your installation. Finally, BioThings SDK is written in python, so you must know some basics.

2.2.2 Configuration file

Before starting to implement our hub, we first need to define a configuration file. A *config_common.py* <*https://github.com/biothings/biothings.species/blob/master/src/config_common.py*> file contains all the required configuration variables, some **have** to be defined in your own application, other **can** be overriden as needed.

Typically we will have to define the following:

- MongoDB connections parameters, `DATA_SRC_*` and `DATA_TARGET_*` parameters. They define connections to two different databases, one will contain individual collections for each datasource (SRC) and the other will contain merged collections (TARGET).
- `HUB_DB_BACKEND` defines a database connection for hub purpose (application specific data, like sources status, etc...). Default backend type is MongoDB. We will need to provide a valid mongodb:// URI. Other backend types are available, like sqlite3 and ElasticSearch, but since we'll use MongoDB to store and process our data, we'll stick to the default.
- `DATA_ARCHIVE_ROOT` contains the path of the root folder that will contain all the downloaded and processed data. Other parameters should be self-explanatory and probably don't need to be changed.
- `LOG_FOLDER` contains the log files produced by the hub

Create a **config.py** and add `from config_common import *` then define all required variables above. **config.py** will look something like this:

```
from config_common import *

DATA_SRC_SERVER = "myhost"
DATA_SRC_PORT = 27017
DATA_SRC_DATABASE = "tutorial_src"
DATA_SRC_SERVER_USERNAME = None
DATA_SRC_SERVER_PASSWORD = None

DATA_TARGET_SERVER = "myhost"
DATA_TARGET_PORT = 27017
DATA_TARGET_DATABASE = "tutorial"
DATA_TARGET_SERVER_USERNAME = None
DATA_TARGET_SERVER_PASSWORD = None
```

(continues on next page)

(continued from previous page)

```
HUB_DB_BACKEND = {
        "module" : "biothings.utils.mongo",
        "uri" : "mongodb://myhost:27017",
        }

DATA_ARCHIVE_ROOT = "/tmp/tutorial"
LOG_FOLDER = "/tmp/tutorial/logs"
```

Note: Log folder must be created manually

2.2.3 hub.py

This script represents the main hub executable. Each hub should define it, this is where the different hub commands are going to be defined and where tasks are actually running. It's also from this script that a SSH server will run so we can actually log into the hub and access those registered commands.

Along this tutorial, we will enrich that script. For now, we're just going to define a JobManager, the SSH server and make sure everything is running fine.

```
import asyncio, asyncssh, sys
import concurrent.futures
from functools import partial

import config, biothings
biothings.config_for_app(config)

from biothings.utils.manager import JobManager

loop = asyncio.get_event_loop()
process_queue = concurrent.futures.ProcessPoolExecutor(max_workers=2)
thread_queue = concurrent.futures.ThreadPoolExecutor()
loop.set_default_executor(process_queue)
jmanager = JobManager(loop,
                      process_queue, thread_queue,
                      max_memory_usage=None,
                      )
```

`jmanager` is our JobManager, it's going to be used everywhere in the hub, each time a parallelized job is created. Species hub is a small one, there's no need for many process workers, two should be fine.

Next, let's define some basic commands for our new hub:

```
from biothings.utils.hub import schedule, top, pending, done
COMMANDS = {
        "sch" : partial(schedule,loop),
        "top" : partial(top,process_queue,thread_queue),
        "pending" : pending,
        "done" : done,
        }
```

These commands are then registered in the SSH server, which is linked to a python interpreter. Commands will be part of the interpreter's namespace and be available from a SSH connection.

```
passwords = {
        'guest': '', # guest account with no password
        }

from biothings.utils.hub import start_server
server = start_server(loop, "Taxonomy hub",passwords=passwords,port=7022,
↪commands=COMMANDS)

try:
    loop.run_until_complete(server)
except (OSError, asyncssh.Error) as exc:
    sys.exit('Error starting server: ' + str(exc))

loop.run_forever()
```

Let's try to run that script ! The first run, it will complain about some missing SSH key:

```
AssertionError: Missing key 'bin/ssh_host_key' (use: 'ssh-keygen -f bin/ssh_host_key'␣
↪to generate it
```

Let's generate it, following instruction. Now we can run it again and try to connect:

```
$ ssh guest@localhost -p 7022
The authenticity of host '[localhost]:7022 ([127.0.0.1]:7022)' can't be established.
RSA key fingerprint is SHA256:USgdr9nlFVryr475+kQWlLyPxwzIUREcnOCyctU1y1Q.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[localhost]:7022' (RSA) to the list of known hosts.

Welcome to Taxonomy hub, guest!
hub>
```

Let's try a command:

```
hub> top()
0 running job(s)
0 pending job(s), type 'top(pending)' for more
```

Nothing fancy here, we don't have much in our hub yet, but everything is running fine.

2.2.4 Dumpers

BioThings species API gathers data from different datasources. We will need to define different dumpers to make this data available locally for further processing.

Taxonomy dumper

This dumper will download taxonomy data from NCBI FTP server. There's one file to download, available at this location: ftp://ftp.ncbi.nih.gov/pub/taxonomy/taxdump.tar.gz.

When defining a dumper, we'll need to choose a base class to derive our dumper class from. There are different base dumper classes available in BioThings SDK, depending on the protocol we want to use to download data. In this case, we'll derive our class from `biothings.dataload.dumper.FTPDumper`. In addition to defining some specific class attributes, we will need to implement a method called `create_todump_list()`. This method fills `self.to_dump` list, which is later going to be used to download data. One element in that list is a dictionary with the following structure:

```
{"remote": "<path to file on remote server", "local": "<local path to file>"}
```

Remote information are relative to the working directory specified as class attribute. Local information is an absolute path, containing filename used to save data.

Let's start coding. We'll save that python module in dataload/sources/taxonomy/dumper.py.

```
import biothings, config
biothings.config_for_app(config)
```

Those lines are used to configure BioThings SDK according to our own configuration information.

```
from config import DATA_ARCHIVE_ROOT
from biothings.dataload.dumper import FTPDumper
```

We then import a configuration constant, and the FTPDumper base class.

```
class TaxonomyDumper(FTPDumper):

    SRC_NAME = "taxonomy"
    SRC_ROOT_FOLDER = os.path.join(DATA_ARCHIVE_ROOT, SRC_NAME)
    FTP_HOST = 'ftp.ncbi.nih.gov'
    CWD_DIR = '/pub/taxonomy'
    SUFFIX_ATTR = "timestamp"
    SCHEDULE = "0 9 * * *"
```

- `SRC_NAME` will used as the registered name for this datasource (more on this later).
- `SRC_ROOT_FOLDER` is the folder path for this resource, without any version information (dumper will create different sub-folders for each version).
- `FTP_HOST` and `CWD_DIR` gives information to connect to the remove FTP server and move to appropriate remote directory (`FTP_USER` and `FTP_PASSWD` constants can also be used for authentication).
- `SUFFIX_ATTR` defines the attributes that's going to be used to create folder for each downloaded version. It's basically either "release" or "timestamp", depending on whether the resource we're trying to dump has an actual version. Here, for taxdump file, there's no version, so we're going to use "timestamp". This attribute is automatically set to current date, so folders will look like that: **.../taxonomy/20170120**, **.../taxonomy/20170121**, etc...
- Finally `SCHEDULE`, if defined, will allow that dumper to regularly run within the hub. This is a cron-like notation (see aiocron documentation for more).

We now need to tell the dumper what to download, that is, create that self.to_dump list:

```
def create_todump_list(self, force=False):
    file_to_dump = "taxdump.tar.gz"
    new_localfile = os.path.join(self.new_data_folder,file_to_dump)
    try:
        current_localfile = os.path.join(self.current_data_folder, file_to_dump)
    except TypeError:
        # current data folder doesn't even exist
        current_localfile = new_localfile
    if force or not os.path.exists(current_localfile) or self.remote_is_better(file_
↪to_dump, current_localfile):
        # register new release (will be stored in backend)
        self.to_dump.append({"remote": file_to_dump, "local":new_localfile})
```

That method tries to get the latest downloaded file and then compare that file with the remote file using `self.remote_is_better(file_to_dump, current_localfile)`, which compares the dates and return True if the remote is more recent. A dict is then created with required elements and appened to `self.to_dump` list.

When the dump is running, each element from that self.to_dump list will be submitted to a job and be downloaded in parallel. Let's try our new dumper. We need to update `hub.py` script to add a DumperManager and then register this dumper:

In hub.py:

```
import dataload
import biothings.dataload.dumper as dumper

dmanager = dumper.DumperManager(job_manager=jmanager)
dmanager.register_sources(dataload.__sources__)
dmanager.schedule_all()
```

Let's also register new commands in the hub:

```
COMMANDS = {
     # dump commands
    "dm" : dmanager,
    "dump" : dmanager.dump_src,
...
```

`dm` will a shortcut for the dumper manager object, and `dump` will actually call manager's `dump_src()` method.

Manager is auto-registering dumpers from list defines in dataload package. Let's define that list:

In dataload/__init__.py:

```
__sources__ = [
        "dataload.sources.taxonomy",
]
```

That's it, it's just a string pointing to our taxonomy package. We'll expose our dumper class in that package so the manager can inspect it and find our dumper (note: we could use give the full path to our dumper module, `dataload.sources.taxonomy.dumper`, but we'll add uploaders later, it's better to have one single line per resource).

In dataload/sources/taxonomy/__init__.py

```
from .dumper import TaxonomyDumper
```

Let's run the hub again. We can on the logs that our dumper has been found:

```
Found a class based on BaseDumper: '<class 'dataload.sources.taxonomy.dumper.
↪TaxonomyDumper'>'
```

Also, manager has found scheduling information and created a task for this:

```
Scheduling task functools.partial(<bound method DumperManager.create_and_dump of
↪<DumperManager [1 registered]: ['taxonomy']>>, <class 'dataload.sources.taxonomy.
↪dumper.TaxonomyDumper'>, job_manager=<biothings.utils.manager.JobManager object at␣
↪0x7f88fc5346d8>, force=False): 0 9 * * *
```

We can double-check this by connecting to the hub, and type some commands:

```
Welcome to Taxonomy hub, guest!
hub> dm
<DumperManager [1 registered]: ['taxonomy']>
```

When printing the manager, we can check our taxonomy resource has been registered properly.

```
hub> sch()
DumperManager.create_and_dump(<class 'dataload.sources.taxonomy.dumper.TaxonomyDumper
↪'>,) [0 9 * * * ] {run in 00h:39m:09s}
```

Dumper is going to run in 39 minutes ! We can trigger a manual upload too:

```
hub> dump("taxonomy")
[1] RUN {0.0s} dump("taxonomy")
```

OK, dumper is running, we can follow task status from the console. At some point, task will be done:

```
hub>
[1] OK  dump("taxonomy"): finished, [None]
```

It successfully run (OK), nothing was returned by the task ([None]). Logs show some more details:

```
DEBUG:taxonomy.hub:Creating new TaxonomyDumper instance
INFO:taxonomy_dump:1 file(s) to download
DEBUG:taxonomy_dump:Downloading 'taxdump.tar.gz'
INFO:taxonomy_dump:taxonomy successfully downloaded
INFO:taxonomy_dump:success
```

Alright, now if we try to run the dumper again, nothing should be downloaded since we got the latest file available. Let's try that, here are the logs:

```
DEBUG:taxonomy.hub:Creating new TaxonomyDumper instance
DEBUG:taxonomy_dump:'taxdump.tar.gz' is up-to-date, no need to download
INFO:taxonomy_dump:Nothing to dump
```

So far so good! The actual file, depending on the configuration settings, it's located in **./data/taxonomy/20170125/taxdump.tar.gz**. We can notice the timestamp used to create the folder. Let's also have a look at in the internal database to see the resource status. Connect to MongoDB:

```
> use hub_config
switched to db hub_config
> db.src_dump.find()
{
        "_id" : "taxonomy",
        "release" : "20170125",
        "data_folder" : "./data/taxonomy/20170125",
        "pending_to_upload" : true,
        "download" : {
                "logfile" : "./data/taxonomy/taxonomy_20170125_dump.log",
                "time" : "4.52s",
                "status" : "success",
                "started_at" : ISODate("2017-01-25T08:32:28.448Z")
        }
}
>
```

We have some information about the download process, how long it took to download files, etc... We have the path to the `data_folder` containing the latest version, the `release` number (here, it's a timestamp), and a flag named `pending_to_upload`. That will be used later to automatically trigger an upload after a dumper has run.

So the actual file is currently compressed, we need to uncompress it before going further. We can add a post-dump step to our dumper. There are two options there, by overriding one of those methods:

```
def post_download(self, remotefile, localfile): triggered for each downloaded file
def post_dump(self): triggered once all files have been downloaded
```

We could use either, but there's a utility function available in BioThings SDK that uncompress everything in a directory, let's use it in a global post-dump step:

```
from biothings.utils.common import untargzall
...

    def post_dump(self):
        untargzall(self.new_data_folder)
```

`self.new_data_folder` is the path to the folder freshly created by the dumper (in our case, **./data/taxonomy/20170125**)

Let's try this in the console (restart the hub to make those changes alive). Because file is up-to-date, dumper will not run. We need to force it:

```
hub> dump("taxonomy",force=True)
```

Or, instead of downloading the file again, we can directly trigger the post-dump step:

```
hub> dump("taxonomy",steps="post")
```

There are 2 steps steps available in a dumper:

1. **dump** : will actually download files
2. **post** : will post-process downloaded files (post_dump)

By default, both run sequentially.

After typing either of these commands, logs will show some information about the uncompressing step:

```
DEBUG:taxonomy.hub:Creating new TaxonomyDumper instance
INFO:taxonomy_dump:success
INFO:root:untargz '/opt/slelong/Documents/Projects/biothings.species/src/data/
↪taxonomy/20170125/taxdump.tar.gz'
```

Folder contains all uncompressed files, ready to be process by an uploader.

UniProt species dumper

Following guideline from previous taxonomy dumper, we're now implementing a new dumper used to download species list. There's just one file to be downloaded from ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/complete/docs/speclist.txt. Same as before, dumper will inherits FTPDumper base class. File is not compressed, so except this, this dumper will look the same.

Code is available on github for further details: ee674c55bad849b43c8514fcc6b7139423c70074 for the whole commit changes, and dataload/sources/uniprot/dumper.py for the actual dumper.

Gene information dumper

The last dumper we have to implement will download some gene information from NCBI (ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/gene_info.gz). It's very similar to the first one (we could even have merged them together).

Code is available on github: d3b3486f71e865235efd673d2f371b53eaa0bc5b for whole changes and dataload/sources/geneinfo/dumper.py for the dumper.

2.2.5 Uploaders

Now that we have local data available, we can process them. We're going to create 3 different uploaders, one for each datasource. Each uploader will load data into MongoDB, into individual/single collections. Those will then be used in the last merging step.

Before going further, we'll first create an UploaderManager instance and register some of its commands in the hub:

```
import biothings.dataload.uploader as uploader
# will check every 10 seconds for sources to upload
umanager = uploader.UploaderManager(poll_schedule = '* * * * * */10', job_
↪manager=jmanager)
umanager.register_sources(dataload.__sources__)
umanager.poll()

COMMANDS = {
...
        # upload commands
        "um" : umanager,
        "upload" : umanager.upload_src,
...
```

Running the hub, we'll see the kind of log statements:

```
INFO:taxonomy.hub:Found 2 resources to upload (['species', 'geneinfo'])
INFO:taxonomy.hub:Launch upload for 'species'
ERROR:taxonomy.hub:Resource 'species' needs upload but is not registered in manager
INFO:taxonomy.hub:Launch upload for 'geneinfo'
ERROR:taxonomy.hub:Resource 'geneinfo' needs upload but is not registered in manager
...
```

Indeed, datasources have been dumped, and a `pending_to_upload` flag has been to True in `src_dump`. UploadManager polls this `src_dump` internal collection, looking for this flag. If set, it runs automatically the corresponding uploader(s). Since we didn't implement any uploaders yet, manager complains... Let's fix that.

Taxonomy uploader

The taxonomy files we downloaded need to be parsed and stored into a MongoDB collection. We won't go in too much details regarding the actual parsing, there are two parsers, one for **nodes.dmp** and another for **names.dmp** files. They yield dictionaries as the result of this parsing step. We just need to "connect" those parsers to uploaders.

Following the same approach as for dumpers, we're going to implement our first uploaders by inheriting one the base classes available in BioThings SDK. We have two files to parse, data will stored in two different MongoDB collections, so we're going to have two uploaders. Each inherits from `biothings.dataload.uploader.BaseSourceUploader`, `load_data` method has to be implemented, this is where we "connect" parsers.

Beside this method, another important point relates to the storage engine. `load_data` will, through the parser, yield documents (dictionaries). This data is processed internally by the base uploader class (`BaseSourceUploader`) using a storage engine. `BaseSourceUploader` uses `biothings.dataload.storage.BasicStorage` as its engine. This storage inserts data in MongoDB collection using bulk operations for better performances. There are other storages available, depending on how data should be inserted (eg. IgnoreDuplicatedStorage will ignore any duplicated data error). While choosing a base uploader class, we need to consider which storage class it's actually using behind-the-scene (an alternative way to do this is using `BaseSourceUploader` and set the class attribute storage_class, such as in this uploader: biothings/dataload/uploader.py#L447).

The first uploader will take care of nodes.dmp parsing and storage.

```
import biothings.dataload.uploader as uploader
from .parser import parse_refseq_names, parse_refseq_nodes

class TaxonomyNodesUploader(uploader.BaseSourceUploader):

    main_source = "taxonomy"
    name = "nodes"

    def load_data(self,data_folder):
        nodes_file = os.path.join(data_folder,"nodes.dmp")
        self.logger.info("Load data from file '%s'" % nodes_file)
        return parse_refseq_nodes(open(nodes_file))
```

- `TaxonomyNodesUploader` derives from `BaseSourceUploader`
- `name` gives the name of the collection used to store the data. If `main_source` is *not* defined, it must match `SRC_NAME` in dumper's attributes
- `main_source` is optional and allows to define main sources and sub-sources. Since we have 2 parsers here, we're going to have 2 collections created. For this one, we want the collection named "nodes". But this parser relates to *taxonomy* datasource, so we define a `main source` called **taxonomy**, which matches `SRC_NAME` in dumper's attributes.
- `load_data()` has `data_folder` as parameter. It will be set accordingly, to the path of the last version dumped. Also, that method gets data from parsing function `parse_refseq_nodes`. It's where we "connect" the parser. We just need to return parser's result so the storage can actually store the data.

The other parser, for names.dmp, is almost the same:

```
class TaxonomyNamesUploader(uploader.BaseSourceUploader):

    main_source = "taxonomy"
    name = "names"

    def load_data(self,data_folder):
        names_file = os.path.join(data_folder,"names.dmp")
        self.logger.info("Load data from file '%s'" % names_file)
        return parse_refseq_names(open(names_file))
```

We then need to "expose" those parsers in taxonomy package, in dataload/sources/taxonomy/__init__.py:

```
from .uploader import TaxonomyNodesUploader, TaxonomyNamesUploader
```

Now let's try to run the hub again. We should see uploader manager has automatically triggered some uploads:

```
INFO:taxonomy.hub:Launch upload for 'taxonomy'
...
...
INFO:taxonomy.names_upload:Uploading 'names' (collection: names)
INFO:taxonomy.nodes_upload:Uploading 'nodes' (collection: nodes)
INFO:taxonomy.nodes_upload:Load data from file './data/taxonomy/20170125/nodes.dmp'
INFO:taxonomy.names_upload:Load data from file './data/taxonomy/20170125/names.dmp'
INFO:root:Uploading to the DB...
INFO:root:Uploading to the DB...
```

While running, we can check what jobs are running, using top() command:

```
hub> top()
   PID     |               SOURCE                | CATEGORY |         STEP         |
↪  DESCRIPTION           |    MEM     | CPU  |      STARTED_AT      | DURATION
5795       | taxonomy.nodes                      | uploader | update_data          |
↪                        | 49.7MiB    | 0.0% | 2017/01/25 14:58:40|15.49s
5796       | taxonomy.names                      | uploader | update_data          |
↪                        | 54.6MiB    | 0.0% | 2017/01/25 14:58:40|15.49s
2 running job(s)
0 pending job(s), type 'top(pending)' for more
16 finished job(s), type 'top(done)' for more
```

We can see two uploaders running at the same time, one for each file. `top(done)` can also display jobs that are done and finally `top(pending)` can give an overview of jobs that are going to be launched when a worker is available (it happens when there are more jobs created than the available number of workers overtime).

In `src_dump` collection, we can see some more information about the resource and its upload processes. Two jobs were created, we have information about the duration, log files, etc...

```
> db.src_dump.find({_id:"taxonomy"})
{
        "_id" : "taxonomy",
        "download" : {
                "started_at" : ISODate("2017-01-25T13:09:26.423Z"),
                "status" : "success",
                "time" : "3.31s",
                "logfile" : "./data/taxonomy/taxonomy_20170125_dump.log"
        },
        "data_folder" : "./data/taxonomy/20170125",
        "release" : "20170125",
        "upload" : {
                "status" : "success",
                "jobs" : {
                        "names" : {
                                "started_at" : ISODate("2017-01-25T14:58:40.034Z"),
                                "pid" : 5784,
                                "logfile" : "./data/taxonomy/taxonomy.names_20170125_
↪upload.log",
                                "step" : "names",
                                "temp_collection" : "names_temp_eJUdh1te",
                                "status" : "success",
                                "time" : "26.61s",
                                "count" : 1552809,
                                "time_in_s" : 27
                        },
                        "nodes" : {
                                "started_at" : ISODate("2017-01-25T14:58:40.043Z"),
                                "pid" : 5784,
                                "logfile" : "./data/taxonomy/taxonomy.nodes_20170125_
↪upload.log",
                                "step" : "nodes",
                                "temp_collection" : "nodes_temp_T5VnzRQC",
                                "status" : "success",
                                "time" : "22.4s",
                                "time_in_s" : 22,
                                "count" : 1552809
                        }
                }
```

(continues on next page)

(continued from previous page)

```
        }
}
```

In the end, two collections were created, containing parsed data:

```
> db.names.count()
1552809
> db.nodes.count()
1552809

> db.names.find().limit(2)
{
        "_id" : "1",
        "taxid" : 1,
        "other_names" : [
                "all"
        ],
        "scientific_name" : "root"
}
{
        "_id" : "2",
        "other_names" : [
                "bacteria",
                "not bacteria haeckel 1894"
        ],
        "genbank_common_name" : "eubacteria",
        "in-part" : [
                "monera",
                "procaryotae",
                "prokaryota",
                "prokaryotae",
                "prokaryote",
                "prokaryotes"
        ],
        "taxid" : 2,
        "scientific_name" : "bacteria"
}

> db.nodes.find().limit(2)
{ "_id" : "1", "rank" : "no rank", "parent_taxid" : 1, "taxid" : 1 }
{
        "_id" : "2",
        "rank" : "superkingdom",
        "parent_taxid" : 131567,
        "taxid" : 2
}
```

UniProt species uploader

Following the same guideline, we're going to create another uploader for species file.

```
import biothings.dataload.uploader as uploader
from .parser import parse_uniprot_speclist

class UniprotSpeciesUploader(uploader.BaseSourceUploader):
```

(continues on next page)

(continued from previous page)

```
    name = "uniprot_species"

    def load_data(self,data_folder):
        nodes_file = os.path.join(data_folder,"speclist.txt")
        self.logger.info("Load data from file '%s'" % nodes_file)
        return parse_uniprot_speclist(open(nodes_file))
```

In that case, we need only one uploader, so we just define "name" (no need to define main_source here).

We need to expose that uploader from the package, in dataload/sources/uniprot/__init__.py:

```
from .uploader import UniprotSpeciesUploader
```

Let's run this through the hub. We can use the "upload" command there (though manager should trigger the upload itself):

```
hub> upload("uniprot_species")
[1] RUN {0.0s} upload("uniprot_species")
```

Similar to dumpers, there are different steps we can individually call for an uploader:

- **data**: will take care of storing data
- **post**: calls post_update() method, once data has been inserted. Useful to post-process data or create an index for instance
- **master**: will register the source in src_master collection, which is used during the merge step. Uploader method `get_mapping()` can optionally returns an ElasticSearch mapping, it will be stored in src_master during that step. We'll see more about this later.
- **clean**: will clean temporary collections and other leftovers...

Within the hub, we can specify these steps manually (they're all executed by default).

```
hub> upload("uniprot_species",steps="clean")
```

Or using a list:

```
hub> upload("uniprot_species",steps=["data","clean"])
```

Gene information uploader

Let's move forward and implement the last uploader. The goal for this uploader is to identify whether, for a taxonomy ID, there are existing/known genes. File contains information about genes, first column is the `taxid`. We want to know all taxonomy IDs present in the file, and the merged document, we want to add key such as `{'has_gene' : True/False}`.

Obviously, we're going to have a lot of duplicates, because for one taxid we can have many genes present in the files. We have options here 1) remove duplicates before inserting data in database, or 2) let the database handle the duplicates (rejecting them). Though we could process data in memory – processed data is rather small in the end –, for demo purpose, we'll go for the second option.

```
import biothings.dataload.uploader as uploader
import biothings.dataload.storage as storage
from .parser import parse_geneinfo_taxid
```

(continues on next page)

(continued from previous page)

```
class GeneInfoUploader(uploader.BaseSourceUploader):

    storage_class = storage.IgnoreDuplicatedStorage

    name = "geneinfo"

    def load_data(self,data_folder):
        gene_file = os.path.join(data_folder,"gene_info")
        self.logger.info("Load data from file '%s'" % gene_file)
        return parse_geneinfo_taxid(open(gene_file))
```

- `storage_class`: this is the most important setting in this case, we want to use a storage that will ignore any duplicated records.
- `parse_geneinfo_taxid` : is the parsing function, yield documents as `{"_id" : "taxid"}`

The rest is closed to what we already encountered. Code is available on github in dataload/sources/geneinfo/uploader.py

When running the uploader, logs show statements like these:

```
INFO:taxonomy.hub:Found 1 resources to upload (['geneinfo'])
INFO:taxonomy.hub:Launch upload for 'geneinfo'
INFO:taxonomy.hub:Building task: functools.partial(<bound method UploaderManager.
↪create_and_load of <UploaderManager [3 registered]: ['geneinfo', 'species',
↪'taxonomy']>>, <class 'dataload.sources.gen
einfo.uploader.GeneInfoUploader'>, job_manager=<biothings.utils.manager.JobManager␣
↪object at 0x7fbf5f8c69b0>)
INFO:geneinfo_upload:Uploading 'geneinfo' (collection: geneinfo)
INFO:geneinfo_upload:Load data from file './data/geneinfo/20170125/gene_info'
INFO:root:Uploading to the DB...
INFO:root:Inserted 62 records, ignoring 9938 [0.3s]
INFO:root:Inserted 15 records, ignoring 9985 [0.28s]
INFO:root:Inserted 0 records, ignoring 10000 [0.23s]
INFO:root:Inserted 31 records, ignoring 9969 [0.25s]
INFO:root:Inserted 16 records, ignoring 9984 [0.26s]
INFO:root:Inserted 4 records, ignoring 9996 [0.21s]
INFO:root:Inserted 4 records, ignoring 9996 [0.25s]
INFO:root:Inserted 1 records, ignoring 9999 [0.25s]
INFO:root:Inserted 26 records, ignoring 9974 [0.23s]
INFO:root:Inserted 61 records, ignoring 9939 [0.26s]
INFO:root:Inserted 77 records, ignoring 9923 [0.24s]
```

While processing data in batch, some are inserted, others (duplicates) are ignored and discarded. The file is quite big, so the process can be long...

Note: should we want to implement the first option, the parsing function would build a dictionary indexed by taxid and would read the whole, extracting taxid. The whole dict would then be returned, and then processed by storage engine.

So far, we've defined dumpers and uploaders, made them working together through some managers defined in the hub. We're now ready to move the last step: merging data.

2.2.6 Mergers

Merging will the last step in our hub definition. So far we have data about species, taxonomy and whether a taxonomy ID has known genes in NCBI. In the end, we want to have a collection where documents look like this:

```
{
    _id: "9606",
    authority: ["homo sapiens linnaeus, 1758"],
    common_name: "man",
    genbank_common_name: "human",
    has_gene: true,
    lineage: [9606,9605,207598,9604,314295,9526,...],
    other_names: ["humans"],
    parent_taxid: 9605,
    rank: "species",
    scientific_name: "homo sapiens",
    taxid: 9606,
    uniprot_name: "homo sapiens"
}
```

- _id: the taxid, the ID used in all of our invidual collection, so the key will be used to collect documents and merge them together (it's actually a requirement, documents are merged using _id as the common key).
- authority, common_name, genbank_common_name, other_names, scientific_name and taxid come from taxonomy.names collection.
- uniprot_name comes from species collection.
- has_gene is a flag set to true, because taxid 9606 has been found in collection geneinfo.
- parent_taxid and rank come from taxonomy.nodes collection.
- (there can be other fields available, but basically the idea here is to merge all our individual collections...)
- finally, lineage... it's a little tricky as we need to query nodes in order to compute that field from _id and parent_taxid.

A first step would be to merge **names**, **nodes** and **species** collections together. Other keys need some post-merge processing, they will handled in a second part.

Let's first define a BuilderManager in the hub.

```
import biothings.databuild.builder as builder
bmanager = builder.BuilderManager(poll_schedule='* * * * * */10', job_
↪manager=jmanager)
bmanager.configure()
bmanager.poll()

COMMANDS = {
...
    # building/merging
    "bm" : bmanager,
    "merge" : bmanager.merge,
...
```

Merging configuration

BuilderManager uses a builder class for merging. While there are many different dumpers and uploaders classes, there's only one merge class (for now). The merging process is defined in a configuration collection named src_build. Usually, we have as many configurations as merged collections, in our case, we'll just define one configuration.

When running the hub with a builder manager registered, manager will automatically create this src_build collection and create configuration placeholder.

```
> db.src_build.find()
{
        "_id" : "placeholder",
        "name" : "placeholder",
        "sources" : [ ],
        "root" : [ ]
}
```

We're going to use that template to create our own configuration:

- **_id** and name are the name of the configuration (they can be different but really, _id is the one used here)... We'll set these as: `{"_id":"mytaxonomy", "name":"mytaxonomy" }`.
- **sources** is a list of collection names used for the merge. A element is this can also be a regular expression matching collection names. If we have data spread across different collection, like one collection per chromosome data, we could use a regex such as `data_chr.*`. We'll set this as: `{"sources":["names","species", "nodes", "geneinfo"]}`
- **root** defines root datasources, that is, datasources that can be used to initiate document creation. Sometimes, we want data to be merged only if an existing document previously exists in the merged collection. If root sources are defined, they will be merged first, then the other remaining in sources will be merged with existing documents. If root doesn't exist (or list is empty), all sources can initiate documents creation. root can be a list of collection names, or a negation (not a mix of both). So, for instance, if we want all datasources to be root, except source10, we can specify: `"root" : ["!source10"]`. Finally, all root sources must all be declared in sources (root is a subset of sources). That said, it's interesting in our case because we have taxonomy information coming from NCBI and UniProt, but we want to make sure a document built from UniProt only doesn't exist (it's because we need parent_taxid field which only exists in NCBI data, so we give priority to those sources first). So root sources are going to be `names` and `nodes`, but because we're lazy typist, we're going to set this to: `{"root" : ["!species"]}`

The resulting document should look like this. Let's save this in src_build (and also remove the placeholder, not useful anymore):

```
> conf
{
        "_id" : "mytaxonomy",
        "name" : "mytaxonomy",
        "sources" : [
                "names",
                "uniprot_species",
                "nodes",
                "geneinfo"
        ],
        "root" : ["!uniprot_species"]
}
> db.src_build.save(conf)
> db.src_build.remove({_id:"placeholder"})
```

Note: **geneinfo** contains only IDs, we could ignore it while merging but we'll need it to be declared as a source when we'll create the index later.

Restarting the hub, we can then check that configuration has properly been registered in the manager, ready to be used. We can list the sources specified in configuration.

```
hub> bm
<BuilderManager [1 registered]: ['mytaxonomy']>
hub> bm.list_sources("mytaxonomy")
['names', 'species', 'nodes']
```

OK, let's try to merge !

```
hub> merge("mytaxonomy")
[1] RUN {0.0s} merge("mytaxonomy")
```

Looking at the logs, we can see builder will first root sources:

```
INFO:mytaxonomy_build:Merging into target collection 'mytaxonomy_20170127_pn1ygtqp'
...
INFO:mytaxonomy_build:Sources to be merged: ['names', 'nodes', 'species', 'geneinfo']
INFO:mytaxonomy_build:Root sources: ['names', 'nodes', 'geneinfo']
INFO:mytaxonomy_build:Other sources: ['species']
INFO:mytaxonomy_build:Merging root document sources: ['names', 'nodes', 'geneinfo']
```

Then once root sources are processed, **species** collection merged on top on existing documents:

```
INFO:mytaxonomy_build:Merging other resources: ['species']
DEBUG:mytaxonomy_build:Documents from source 'species' will be stored only if a␣
↪previous document exists with same _id
```

After a while, task is done, merge has returned information about the amount of data that have been merge: 1552809 records from collections **names**, **nodes** and **geneinfo**, 25394 from **species**. Note: the figures show the number fetched from collections, not necessarily the data merged. For instance, merged data from **species** may be less since it's not a root datasource).

```
hub>
[1] OK  merge("mytaxonomy"): finished, [{'total_species': 25394, 'total_nodes':␣
↪1552809, 'total_names': 1552809}]
```

Builder creates multiple merger jobs per collection. The merged collection name is, by default, generating from the build name (**mytaxonomy**), and contains also a timestamp and some random chars. We can specify the merged collection name from the hub. By default, all sources defined in the configuration are merged., and we can also select one or more specific sources to merge:

```
hub> merge("mytaxonomy",sources="uniprot_species",target_name="test_merge")
```

Note: `sources` parameter can also be a list of string.

If we go back to `src_build`, we can have information about the different merges (or builds) we ran:

```
> db.src_build.find({_id:"mytaxonomy"},{build:1})
{
        "_id" : "mytaxonomy",
        "build" : [
             ...
{
                "src_versions" : {
                        "geneinfo" : "20170125",
                        "taxonomy" : "20170125",
                        "uniprot_species" : "20170125"
                },
                "time_in_s" : 386,
                "logfile" : "./data/logs/mytaxonomy_20170127_build.log",
                "pid" : 57702,
                "target_backend" : "mongo",
                "time" : "6m26.29s",
                "step_started_at" : ISODate("2017-01-27T11:36:47.401Z"),
                "stats" : {
```

(continues on next page)

(continued from previous page)

```
                        "total_uniprot_species" : 25394,
                        "total_nodes" : 1552809,
                        "total_names" : 1552809
                },
                "started_at" : ISODate("2017-01-27T11:30:21.114Z"),
                "status" : "success",
                "target_name" : "mytaxonomy_20170127_pn1ygtqp",
                "step" : "post-merge",
                "sources" : [
                        "uniprot_species"
                ]
        }
```

We can see the merged collection (auto-generated) is **mytaxonomy_20170127_pn1ygtqp**. Let's have a look at the content (remember, collection is in target database, not in src):

```
> use tutorial
switched to db tutorial
> db.mytaxonomy_20170127_pn1ygtqp.count()
1552809
> db.mytaxonomy_20170127_pn1ygtqp.find({_id:9606})
{
        "_id" : 9606,
        "rank" : "species",
        "parent_taxid" : 9605,
        "taxid" : 9606,
        "common_name" : "man",
        "other_names" : [
                "humans"
        ],
        "scientific_name" : "homo sapiens",
        "authority" : [
                "homo sapiens linnaeus, 1758"
        ],
        "genbank_common_name" : "human",
        "uniprot_name" : "homo sapiens"
}
```

Both collections have properly been merged. We now have to deal with the other data.

Mappers

The next bit of data we need to merge is **geneinfo**. As a reminder, this collection only contains taxonomy ID (as _id key) which have known NCBI genes. We'll create a mapper, containing this information. A mapper basically acts as an object that can pre-process documents while they are merged.

Let's define that mapper in databuild/mapper.py

```
import biothings, config
biothings.config_for_app(config)
from biothings.utils.common import loadobj
import biothings.utils.mongo as mongo
import biothings.databuild.mapper as mapper
# just to get the collection name
from dataload.sources.geneinfo.uploader import GeneInfoUploader
```

(continues on next page)

(continued from previous page)

```
class HasGeneMapper(mapper.BaseMapper):

    def __init__(self, *args, **kwargs):
        super(HasGeneMapper,self).__init__(*args,**kwargs)
        self.cache = None

    def load(self):
        if self.cache is None:
            # this is a whole dict containing all taxonomy _ids
            col = mongo.get_src_db()[GeneInfoUploader.name]
            self.cache = [d["_id"] for d in col.find({},{"_id":1})]

    def process(self,docs):
        for doc in docs:
            if doc["_id"] in self.cache:
                doc["has_gene"] = True
            else:
                doc["has_gene"] = False
            yield doc
```

We derive our mapper from `biothings.databuild.mapper.BaseMapper`, which expects `load` and `process` methods to be defined. `load` is automatically called when the mapper is used by the builder, and `process` contains the main logic, iterating over documents, optionally enrich them (it can also be used to filter documents, by not yielding them). The implementation is pretty straightforward. We get and cache the data from geneinfo collection (the whole collection is very small, less than 20'000 IDs, so it can fit nicely and efficiently in memory). If a document has its _id found in the cache, we enrich it.

Once defined, we register that mapper into the builder. In bin/hub.py, we modify the way we define the builder manager:

```
import biothings.databuild.builder as builder
from databuild.mapper import HasGeneMapper
hasgene = HasGeneMapper(name="has_gene")
pbuilder = partial(builder.DataBuilder,mappers=[hasgene])
bmanager = builder.BuilderManager(
        poll_schedule='* * * * * */10',
        job_manager=jmanager,
        builder_class=pbuilder)
bmanager.configure()
bmanager.poll()
```

First we instantiate a mapper object and give it a name (more on this later). While creating the manager, we need to pass a builder class. The problem here is we also have to give our mapper to that class while it's instantiated. We're using `partial` (from `functools`), which allows to partially define the class instantiation. In the end, builder_class parameter is expected to a callable, which is the case with partial.

Let's try if our mapper works (restart the hub). Inside the hub, we're going to manually get a builder instance. Remember through the SSH connection, we can access python interpreter's namespace, which is very handy when it comes to develop and debug as we can directly access and "play" with objects and their states:

First we get a builder instance from the manager:

```
hub> builder = bm["mytaxonomy"]
hub> builder
<biothings.databuild.builder.DataBuilder object at 0x7f278aecf400>
```

Let's check the mappers and get ours:

```
hub> builder.mappers
{None: <biothings.databuild.mapper.TransparentMapper object at 0x7f278aecf4e0>, 'has_
↪gene': <databuild.mapper.HasGeneMapper object at 0x7f27ac6c0a90>}
```

We have our `has_gene` mapper (it's the name we gave). We also have a `TransparentMapper`. This mapper is automatically added and is used as the default mapper for any document (there has to be one...).

```
hub> hasgene = builder.mappers["has_gene"]
hub> len(hasgene.cache)
Error: TypeError("object of type 'NoneType' has no len()",)
```

Oops, cache isn't loaded yet, we have to do it manually here (but it's done automatically during normal execution).

```
hub> hasgene.load()
hub> len(hasgene.cache)
19201
```

OK, it's ready. Let's now talk more about the mapper's name. A mapper can applied to different sources, and we have to define which sources' data should go through that mapper. In our case, we want **names** and **species** collection's data to go through. In order to do that, we have to instruct the uploader with a special attribute. Let's modify dataload.sources.species.uploader.UniprotSpeciesUploader class

```
class UniprotSpeciesUploader(uploader.BaseSourceUploader):

    name = "uniprot_species"
    __metadata__ = {"mapper" : 'has_gene'}
```

`__metadata__` dictionary is going to be used to create a master document. That document is stored in src_master collection (we talked about it earlier). Let's add this metadata to dataload.sources.taxonomy.uploader.TaxonomyNamesUploader

```
class TaxonomyNamesUploader(uploader.BaseSourceUploader):

    main_source = "taxonomy"
    name = "names"
    __metadata__ = {"mapper" : 'has_gene'}
```

Before using the builder, we need to refresh master documents so these metadata are stored in **src_master**. We could trigger a new upload, or directly tell the hub to only process master steps:

```
hub> upload("uniprot_species",steps="master")
[1] RUN {0.0s} upload("uniprot_species",steps="master")
hub> upload("taxonomy.names",steps="master")
[1] OK  upload("uniprot_species",steps="master"): finished, [None]
[2] RUN {0.0s} upload("taxonomy.names",steps="master")
```

(you'll notice for taxonomy, we only trigger upload for sub-source **names**, using "dot-notation", corresponding to "main_source.name". Let's now have a look at those master documents:

```
> db.src_master.find({_id:{$in:["uniprot_species","names"]}})
{
        "_id" : "names",
        "name" : "names",
        "timestamp" : ISODate("2017-01-26T16:21:32.546Z"),
        "mapper" : "has_gene",
```

(continues on next page)

(continued from previous page)

```
        "mapping" : {

        }
}
{
        "_id" : "uniprot_species",
        "name" : "uniprot_species",
        "timestamp" : ISODate("2017-01-26T16:21:19.414Z"),
        "mapper" : "has_gene",
        "mapping" : {

        }
}
```

We have our `mapper` key stored. We can now trigger a new merge (we specify the target collection name):

```
hub> merge("mytaxonomy",target_name="mytaxonomy_test")
[3] RUN {0.0s} merge("mytaxonomy",target_name="mytaxonomy_test")
```

In the logs, we can see our mapper has been detected and is used:

```
INFO:mytaxonomy_build:Creating merger job #1/16, to process 'names' 100000/1552809 (6.
↪4%)
INFO:mytaxonomy_build:Found mapper '<databuild.mapper.HasGeneMapper object at␣
↪0x7f47ef3bbac8>' for source 'names'
INFO:mytaxonomy_build:Creating merger job #1/1, to process 'species' 25394/25394 (100.
↪0%)
INFO:mytaxonomy_build:Found mapper '<databuild.mapper.HasGeneMapper object at␣
↪0x7f47ef3bbac8>' for source 'species'
```

Once done, we can query the merged collection to check the data:

```
> use tutorial
switched to db tutorial
> db.mytaxonomy_test.find({_id:9606})
{
        "_id" : "9606",
        "has_gene" : true,
        "taxid" : 9606,
        "uniprot_name" : "homo sapiens",
        "other_names" : [
                "humans"
        ],
        "scientific_name" : "homo sapiens",
        "authority" : [
                "homo sapiens linnaeus, 1758"
        ],
        "genbank_common_name" : "human",
        "common_name" : "man"
}
```

OK, there's a `has_gene` flag that's been set. So far so good !

Post-merge process

We need to add lineage and parent taxid information for each of these documents. We'll implement that last part as a post-merge step, iterating over each of them. In order to do so, we need to define our own builder class to override proper methodes there. Let's define it in databuild/builder.py..

```
import biothings.databuild.builder as builder
import config

class TaxonomyDataBuilder(builder.DataBuilder):

    def post_merge(self, source_names, batch_size, job_manager):
        pass
```

The method we have to implement in post_merge, as seen above. We also need to change hub.py to use that builder class:

```
from databuild.builder import TaxonomyDataBuilder
pbuilder = partial(TaxonomyDataBuilder,mappers=[hasgene])
```

For now, we just added a class level in the hierarchy, everything runs the same as before. Let's have a closer look to that post-merge process. For each document, we want to build the lineage. Information is stored in **nodes** collection. For instance, taxid 9606 (homo sapiens) has a parent_taxid 9605 (homo), which has a parent_taxid 207598 (hominidae), etc... In the end, the homo sapiens lineage is:

```
9606, 9605, 207598, 9604, 314295, 9526, 314293, 376913, 9443, 314146, 1437010,
9347, 32525, 40674, 32524, 32523, 1338369, 8287, 117571, 117570, 7776, 7742,
89593, 7711, 33511, 33213, 6072, 33208, 33154, 2759, 131567 and 1
```

We could recursively query **nodes** collections until we reach the top the tree, but that would be a lot of queries. We just need `taxid` and `parent_taxid` information to build the lineage, maybe it's possible to build a dictionary that could fit in memory. **nodes** has 1552809 records. A dictionary would use 2 * 1552809 * sizeof(integer) + index overhead. That's probably few megabytes, let's assume that ok... (note: using pympler lib, we can actually know that dictionary size will be closed to 200MB...)

We're going to use another mapper here, but no sources will use it.We'll just instantiate it from post_merge method. In databuild/mapper.py, let's add another class:

from dataload.sources.taxonomy.uploader import TaxonomyNodesUploader

```
class LineageMapper(mapper.BaseMapper):

    def __init__(self, *args, **kwargs):
        super(LineageMapper,self).__init__(*args,**kwargs)
        self.cache = None

    def load(self):
        if self.cache is None:
            col = mongo.get_src_db()[TaxonomyNodesUploader.name]
            self.cache = {}
            [self.cache.setdefault(d["_id"],d["parent_taxid"]) for d in col.find({},{
↪"parent_taxid":1})]

    def get_lineage(self,doc):
        if doc['taxid'] == doc['parent_taxid']: #take care of node #1
            # we reached the top of the taxonomy tree
            doc['lineage'] = [doc['taxid']]
            return doc
```

(continues on next page)

(continued from previous page)

```
        # initiate lineage with information we have in the current doc
        lineage = [doc['taxid'], doc['parent_taxid']]
        while lineage[-1] != 1:
            parent = self.cache[lineage[-1]]
            lineage.append(parent)
        doc['lineage'] = lineage
        return doc

    def process(self,docs):
        for doc in docs:
            doc = self.get_lineage(doc)
            yield doc
```

Let's use that mapper in TaxonomyDataBuider's `post_merge` method. The signature is the same as merge() method (what's actually called from the hub) but we just need the batch_size one: we're going to grab documents from the merged collection in batch, process them and update them in batch as well. It's going to be much faster than dealing one document at a time. To do so, we'll use doc_feeder utility function:

```
from biothings.utils.mongo import doc_feeder, get_target_db
from biothings.databuild.builder import DataBuilder
from biothings.dataload.storage import UpsertStorage

from databuild.mapper import LineageMapper
import config
import logging

class TaxonomyDataBuilder(DataBuilder):

    def post_merge(self, source_names, batch_size, job_manager):
        # get the lineage mapper
        mapper = LineageMapper(name="lineage")
        # load cache (it's being loaded automatically
        # as it's not part of an upload process
        mapper.load()

        # create a storage to save docs back to merged collection
        db = get_target_db()
        col_name = self.target_backend.target_collection.name
        storage = UpsertStorage(db,col_name)

        for docs in doc_feeder(self.target_backend.target_collection, step=batch_size,
↪ inbatch=True):
            docs = mapper.process(docs)
            storage.process(docs,batch_size)
```

Since we're using the mapper manually, we need to load the cache

- **db** and **col_name** are used to create our storage engine. Builder has an attribute called `target_backend` (a `biothings.dataload.backend.TargetDocMongoBackend` object) which can be used to reach the collection we want to work with.
- **doc_feeder** iterates over all the collection, fetching documents in batch. `inbatch=True` tells the function to return data as a list (default is a dict indexed by `_id`).
- those documents are processed by our mapper, setting the lineage information and then are stored using our UpsertStorage object.

Note: `post_merge` actually runs within a thread, so any calls here won't block the execution (ie. won't block the

asyncio event loop execution)

Let's run this on our merged collection. We don't want to merge everything again, so we specify the step we're interested in and the actual merged collection (`target_name`)

hub> merge("mytaxonomy",steps="post",target_name="mytaxonomy_test") [1] RUN {0.0s} merge("mytaxonomy",steps="post",target_name="mytaxonomy_test")

After a while, process is done. We can test our updated data:

```
> use tutorial
switched to db tutorial
> db.mytaxonomy_test.find({_id:9606})
{
        "_id" : 9606,
        "taxid" : 9606,
        "common_name" : "man",
        "other_names" : [
                "humans"
        ],
        "uniprot_name" : "homo sapiens",
        "rank" : "species",
        "lineage" : [9606,9605,207598,9604,...,131567,1],
        "genbank_common_name" : "human",
        "scientific_name" : "homo sapiens",
        "has_gene" : true,
        "parent_taxid" : 9605,
        "authority" : [
                "homo sapiens linnaeus, 1758"
        ]
}
```

OK, we have new lineage information (truncated for sanity purpose). Merged collection is ready to be used. It can be used for instance to create and send documents to an ElasticSearch database. This is what's actually occuring when creating a BioThings web-servuce API. That step will be covered in another tutorial.

2.2.7 Indexers

Coming soon!

Full updated and maintained code for this hub is available here: https://github.com/biothings/biothings.species

Also, taxonomy BioThings API can be queried as this URL: http://t.biothings.io

2.3 Hub component

The purpose of the BioThings hub component is to allow you to easily automate the parsing and uploading of your data to an Elasticsearch backend.

2.3.1 dumper

BaseDumper

class biothings.dataload.dumper.**BaseDumper**(*src_name=None*, *src_root_folder=None*, *log_folder=None*, *no_confirm=True*, *archive=None*)

create_todump_list(*force=False*, ***kwargs*)
Fill self.to_dump list with dict("remote":remote_path,"local":local_path) elements. This is the todo list for the dumper. It's a good place to check whether needs to be downloaded. If 'force' is True though, all files will be considered for download

download(*remotefile*, *localfile*)
Download "remotefile' to local location defined by 'localfile' Return relevant information about remotefile (depends on the actual client)

dump(*steps=None*, *force=False*, *job_manager=None*, ***kwargs*)
Dump (ie. download) resource as needed this should be called after instance creation 'force' argument will force dump, passing this to create_todump_list() method.

get_pinfo()
Return dict containing information about the current process (used to report in the hub)

need_prepare()
check whether some prepare step should executed before running dump

new_data_folder
Generate a new data folder path using src_root_folder and specified suffix attribute. Also sync current (aka previous) data folder previously registeted in database. This method typically has to be called in create_todump_list() when the dumper actually knows some information about the resource, like the actual release.

post_download(*remotefile*, *localfile*)
Placeholder to add a custom process once a file is downloaded. This is a good place to check file's integrity. Optional

post_dump()
Placeholder to add a custom process once the whole resource has been dumped. Optional.

prepare_client()
do initialization to make the client ready to dump files

release_client()
Do whatever necessary (like closing network connection) to "release" the client

remote_is_better(*remotefile*, *localfile*)
Compared to local file, check if remote file is worth downloading. (like either bigger or newer for instance)

unprepare()
reset anything that's not pickable (so self can be pickled) return what's been reset as a dict, so self can be restored once pickled

FTPDumper

class biothings.dataload.dumper.**FTPDumper**(*src_name=None*, *src_root_folder=None*, *log_folder=None*, *no_confirm=True*, *archive=None*)

download(*remotefile*, *localfile*)
Download "remotefile' to local location defined by 'localfile' Return relevant information about remotefile (depends on the actual client)

need_prepare()
check whether some prepare step should executed before running dump

prepare_client()
do initialization to make the client ready to dump files

release_client()
Do whatever necessary (like closing network connection) to "release" the client

remote_is_better(*remotefile*, *localfile*)
'remotefile' is relative path from current working dir (CWD_DIR), 'localfile' is absolute path

HTTPDumper

class biothings.dataload.dumper.**HTTPDumper**(*src_name=None*, *src_root_folder=None*, *log_folder=None*, *no_confirm=True*, *archive=None*)
Dumper using HTTP protocol and "requests" library

download(*remoteurl*, *localfile*, *headers={}*)
kwargs will be passed to requests.Session.get()

need_prepare()
check whether some prepare step should executed before running dump

prepare_client()
do initialization to make the client ready to dump files

release_client()
Do whatever necessary (like closing network connection) to "release" the client

remote_is_better(*remotefile*, *localfile*)
Compared to local file, check if remote file is worth downloading. (like either bigger or newer for instance)

GoogleDriveDumper

class biothings.dataload.dumper.**GoogleDriveDumper**(*src_name=None*, *src_root_folder=None*, *log_folder=None*, *no_confirm=True*, *archive=None*)

download(*remoteurl*, *localfile*)

remoteurl is a google drive link containing a document ID, such as:

- https://drive.google.com/open?id=<1234567890ABCDEF>
- https://drive.google.com/file/d/<1234567890ABCDEF>/view

It can also be just a document ID

prepare_client()
do initialization to make the client ready to dump files

remote_is_better(*remotefile*, *localfile*)
Compared to local file, check if remote file is worth downloading. (like either bigger or newer for instance)

WgetDumper

class biothings.dataload.dumper.**WgetDumper**(*src_name=None*, *src_root_folder=None*, *log_folder=None*, *no_confirm=True*, *archive=None*)

create_todump_list(*force=False*, ***kwargs*)
Fill self.to_dump list with dict("remote":remote_path,"local":local_path) elements. This is the todo list for the dumper. It's a good place to check whether needs to be downloaded. If 'force' is True though, all files will be considered for download

download(*remoteurl*, *localfile*)
Download "remotefile' to local location defined by 'localfile' Return relevant information about remotefile (depends on the actual client)

need_prepare()
check whether some prepare step should executed before running dump

prepare_client()
Check if 'wget' executable exists

release_client()
Do whatever necessary (like closing network connection) to "release" the client

remote_is_better(*remotefile*, *localfile*)
Compared to local file, check if remote file is worth downloading. (like either bigger or newer for instance)

DummyDumper

class biothings.dataload.dumper.**DummyDumper**(**args*, ***kwargs*)
DummyDumper will do nothing... (useful for datasources that can't be downloaded anymore but still need to be integrated, ie. fill src_dump, etc...)

dump(*force=False*, *job_manager=None*)
Dump (ie. download) resource as needed this should be called after instance creation 'force' argument will force dump, passing this to create_todump_list() method.

prepare_client()
do initialization to make the client ready to dump files

ManualDumper

class biothings.dataload.dumper.**ManualDumper**(**args*, ***kwargs*)
This dumper will assist user to dump a resource. It will usually expect the files to be downloaded first (sometimes there's no easy way to automate this process). Once downloaded, a call to dump() will make sure everything is fine in terms of files and metadata

dump(*path*, *release=None*, *force=False*, *job_manager=None*)
Dump (ie. download) resource as needed this should be called after instance creation 'force' argument will force dump, passing this to create_todump_list() method.

new_data_folder
Generate a new data folder path using src_root_folder and specified suffix attribute. Also sync current (aka previous) data folder previously registeted in database. This method typically has to be called in create_todump_list() when the dumper actually knows some information about the resource, like the actual release.

prepare_client()
do initialization to make the client ready to dump files

2.3.2 uploader

BaseSourceUploader

class biothings.dataload.uploader.**BaseSourceUploader**(*db_conn_info*, *data_root*, *collection_name=None*, *log_folder=None*, **args*, ***kwargs*)

Default datasource uploader. Database storage can be done in batch or line by line. Duplicated records aren't not allowed

db_conn_info is a database connection info tuple (host,port) to fetch/store information about the datasource's state data_root is the root folder containing all resources. It will generate its own data folder from this point

classmethod create(*db_conn_info*, *data_root*, **args*, ***kwargs*)
Factory-like method, just return an instance of this uploader (used by SourceManager, may be overridden in sub-class to generate more than one instance per class, like a true factory. This is usefull when a resource is splitted in different collection but the data structure doesn't change (it's really just data splitted accros multiple collections, usually for parallelization purposes). Instead of having actual class for each split collection, factory will generate them on-the-fly.

classmethod get_mapping()
Return ES mapping

get_pinfo()
Return dict containing information about the current process (used to report in the hub)

load(*steps=['data', 'post', 'master', 'clean'], force=False, batch_size=10000, job_manager=None, **kwargs*)
Main resource load process, reads data from doc_c using chunk sized as batch_size. steps defines the different processes used to laod the resource: - "data" : will store actual data into single collections - "post" : will perform post data load operations - "master" : will register the master document in src_master

load_data(*data_folder*)
Parse data inside data_folder and return structure ready to be inserted in database

make_temp_collection()
Create a temp collection for dataloading, e.g., entrez_geneinfo_INEMO.

post_update_data(*steps*, *force*, *batch_size*, *job_manager*, ***kwargs*)
Override as needed to perform operations after data has been uploaded

prepare(*state={}*)
Sync uploader information with database (or given state dict)

prepare_src_dump()
Sync with src_dump collection, collection information (src_doc) Return src_dump collection

register_status(*status*, ***extra*)
Register step status, ie. status for a sub-resource

setup_log()
Setup and return a logger instance

switch_collection()
after a successful loading, rename temp_collection to regular collection name, and renaming existing collection to a temp name for archiving purpose.

unprepare()
reset anything that's not pickable (so self can be pickled) return what's been reset as a dict, so self can be restored once pickled

update_data(*batch_size*, *job_manager*)
Iterate over load_data() to pull data and store it

NoBatchIgnoreDuplicatedSourceUploader

class biothings.dataload.uploader.**NoBatchIgnoreDuplicatedSourceUploader**(*db_conn_info*, *data_root*, *collection_name=None*, *log_folder=None*, **args*, ***kwargs*)
Same as default uploader, but will store records and ignore if any duplicated error occuring (use with caution...). Storage is done line by line (slow, not using a batch) but preserve order of data in input file.

db_conn_info is a database connection info tuple (host,port) to fetch/store information about the datasource's state data_root is the root folder containing all resources. It will generate its own data folder from this point

storage_class
alias of biothings.dataload.storage.NoBatchIgnoreDuplicatedStorage

IgnoreDuplicatedSourceUploader

class biothings.dataload.uploader.**IgnoreDuplicatedSourceUploader**(*db_conn_info*, *data_root*, *collection_name=None*, *log_folder=None*, **args*, ***kwargs*)
Same as default uploader, but will store records and ignore if any duplicated error occuring (use with caution...). Storage is done using batch and unordered bulk operations.

db_conn_info is a database connection info tuple (host,port) to fetch/store information about the datasource's state data_root is the root folder containing all resources. It will generate its own data folder from this point

MergerSourceUploader

class biothings.dataload.uploader.**MergerSourceUploader**(*db_conn_info*, *data_root*, *collection_name=None*, *log_folder=None*, **args*, ***kwargs*)
db_conn_info is a database connection info tuple (host,port) to fetch/store information about the datasource's state data_root is the root folder containing all resources. It will generate its own data folder from this point

storage_class
alias of biothings.dataload.storage.MergerStorage

DummySourceUploader

class biothings.dataload.uploader.**DummySourceUploader**(*db_conn_info*, *data_root*, *collection_name=None*, *log_folder=None*, **args*, ***kwargs*)

Dummy uploader, won't upload any data, assuming data is already there but make sure every other bit of information is there for the overall process (usefull when online data isn't available anymore)

db_conn_info is a database connection info tuple (host,port) to fetch/store information about the datasource's state data_root is the root folder containing all resources. It will generate its own data folder from this point

prepare_src_dump()

Sync with src_dump collection, collection information (src_doc) Return src_dump collection

update_data(*batch_size*, *job_manager=None*, *release=None*)

Iterate over load_data() to pull data and store it

ParallelizedSourceUploader

class biothings.dataload.uploader.**ParallelizedSourceUploader**(*db_conn_info*, *data_root*, *collection_name=None*, *log_folder=None*, **args*, ***kwargs*)

db_conn_info is a database connection info tuple (host,port) to fetch/store information about the datasource's state data_root is the root folder containing all resources. It will generate its own data folder from this point

jobs()

Return list of (*arguments) passed to self.load_data, in order. for each parallelized jobs. Ex: [(x,1),(y,2),(z,3)] If only one argument is required, it still must be passed as a 1-element tuple

update_data(*batch_size*, *job_manager=None*)

Iterate over load_data() to pull data and store it

NoDataSourceUploader

class biothings.dataload.uploader.**NoDataSourceUploader**(*db_conn_info*, *data_root*, *collection_name=None*, *log_folder=None*, **args*, ***kwargs*)

This uploader won't upload any data and won't even assume there's actual data (different from DummySourceUploader on this point). It's usefull for instance when mapping need to be stored (get_mapping()) but data doesn't comes from an actual upload (ie. generated)

db_conn_info is a database connection info tuple (host,port) to fetch/store information about the datasource's state data_root is the root folder containing all resources. It will generate its own data folder from this point

storage_class

alias of biothings.dataload.storage.NoStorage

update_data(*batch_size*, *job_manager=None*)

Iterate over load_data() to pull data and store it

2.3.3 builder

DataBuilder

class biothings.databuild.builder.**DataBuilder**(*build_name*, *source_backend*, *target_backend*, *log_folder*, *doc_root_key='root'*, *mappers=[]*, *default_mapper_class=<class 'biothings.databuild.mapper.TransparentMapper'>*, *sources=None*, *target_name=None*, ***kwargs*)

get_build_version()
Generate an arbitrary major build version. Default is using a timestamp (YYMMDD) '.' char isn't allowed in build version as it's reserved for minor versions

get_mapping(*sources*)
Merge mappings from src_master

get_metadata(*sources*, *job_manager*)
Return a dictionnary of metadata for this build. It's usually app-specific and this method may be overridden as needed. Default: return "merge_stats" (#docs involved in each single collections used in that build)

Return dictionary will potentially be merged with existing metadata in src_build collection. This behavior can be changed by setting a special key within metadata dict: {"__REPLACE__" : True} will... replace existing metadata with the one returned here.

"job_manager" is passed in case parallelization is needed. Be aware that this method is already running in a dedicated thread, in order to use job_manager, the following code must be used at the very beginning of its implementation: asyncio.set_event_loop(job_manager.loop)

get_pinfo()
Return dict containing information about the current process (used to report in the hub)

merge_sources(*source_names, steps=['merge', 'post'], batch_size=100000, ids=None, job_manager=None*)
Merge resources from given source_names or from build config. Identify root document sources from the list to first process them. ids can a be list of documents to be merged in particular.

register_status(*status*, *transient=False*, *init=False*, ***extra*)
Register current build status. A build status is a record in src_build The key used in this dict the target_name. Then, any operation acting on this target_name is registered in a "jobs" list.

resolve_sources(*sources*)
Source can be a string that may contain regex chars. It's usefull when you have plenty of sub-collections prefixed with a source name. For instance, given a source named 'blah' stored in as many collections as chromosomes, insteand of passing each name as 'blah_1', 'blah_2', etc... '**blah_**.*' can be specified in build_config. This method resolves potential regexed source name into real, existing collection names

unprepare()
reset anything that's not pickable (so self can be pickled) return what's been reset as a dict, so self can be restored once pickled

2.3.4 indexer

Indexer

class biothings.dataindex.indexer.**Indexer**(*es_host*, *target_name=None*)

get_index_creation_settings()
Override to return a dict containing some extra settings for index creation. Dict will be merged with mandatory settings, see biothings.utils.es.ESIndexer.create_index for more.

get_mapping(*enable_timestamp=True*)
collect mapping data from data sources. This is for GeneDocESBackend only.

get_pinfo()
Return dict containing information about the current process (used to report in the hub)

index(*target_name, index_name, job_manager, steps=['index', 'post'], batch_size=10000, ids=None, mode='index'*)
Build an index named "index_name" with data from collection "target_collection". "ids" can be passed to selectively index documents. "mode" can have the following values: - 'purge': will delete index if it exists - 'resume': will use existing index and add documents. "ids" can be passed as a list of missing IDs,

or, if not pass, ES will be queried to identify which IDs are missing for each batch in order to complete the index.

- None (default): will create a new index, assuming it doesn't already exist

load_build(*target_name=None*)
Load build info from src_build collection.

post_index(*target_name, index_name, job_manager, steps=['index', 'post'], batch_size=10000, ids=None, mode=None*)
Override in sub-class to add a post-index process. Method's signature is the same as index() to get the full context. This method will run in a thread (using job_manager.defer_to_thread())

2.3.5 differ

BaseDiffer

class biothings.databuild.differ.**BaseDiffer**(*diff_func*, *job_manager*, *log_folder*)

diff(*old_db_col_names, new_db_col_names, batch_size=100000, steps=['content', 'mapping'], mode=None, exclude=[]*)
wrapper over diff_cols() coroutine, return a task

diff_cols(*old_db_col_names, new_db_col_names, batch_size=100000, steps=['count', 'content', 'mapping'], mode=None, exclude=[]*)
Compare new with old collections and produce diff files. Root keys can be excluded from comparison with "exclude" parameter. *_db_col_names can be:

1. a colleciton name (as a string) asusming they are in the target database.
2. tuple with 2 elements, the first one is then either "source" or "target" to respectively specify src or target database, and the second element is the collection name.

3. tuple with 3 elements (URI,db,collection), looking like: ("mongodb://user:pass@host","dbname","collection"), allowing to specify any connection on any server

steps: 'count' will count the root keys for every documents in new collection (to check number of docs from datasources). 'content' will perform diff on actual content. 'mapping' will perform diff on ES mappings (if target collection involved)

mode: 'purge' will remove any existing files for this comparison.

JsonDiffer

class biothings.databuild.differ.**JsonDiffer**(*diff_func=<function diff_docs_jsonpatch>*, **args*, ***kwargs*)

SelfContainedJsonDiffer

class biothings.databuild.differ.**SelfContainedJsonDiffer**(*diff_func=<function diff_docs_jsonpatch>*, **args*, ***kwargs*)

DiffReportRendererBase

class biothings.databuild.differ.**DiffReportRendererBase**(*max_reported_ids=None*, *max_randomly_picked=None*, *detailed=False*)

save(*report*, *filename*)
Save report output (rendered) into filename

DiffReportTxt

class biothings.databuild.differ.**DiffReportTxt**(*max_reported_ids=None*, *max_randomly_picked=None*, *detailed=False*)

save(*report*, *filename='report.txt'*)
Save report output (rendered) into filename

2.3.6 syncer

BaseSyncer

class biothings.databuild.syncer.**BaseSyncer**(*job_manager*, *log_folder*)

sync(*diff_folder=None*, *batch_size=10000*, *mode=None*, *target_backend=None*, *steps=['mapping', 'content', 'meta']*)
wrapper over sync_cols() coroutine, return a task

sync_cols(*diff_folder, batch_size=10000, mode=None, force=False, target_backend=None, steps=['mapping', 'content', 'meta']*)
Sync a collection with diff files located in diff_folder. This folder contains a metadata.json file which describes the different involved collection: "old" is the collection/index to be synced, "new" is the collecion that should be obtained once all diff files are applied (not used, just informative). If target_backend (bt.databbuild.backend.create_backend() notation), then it will replace "old" (that is, the one being synced)

MongoJsonDiffSyncer

class biothings.databuild.syncer.**MongoJsonDiffSyncer**(*job_manager*, *log_folder*)

MongoJsonDiffSelfContainedSyncer

class biothings.databuild.syncer.**MongoJsonDiffSelfContainedSyncer**(*job_manager*, *log_folder*)

ESJsonDiffSyncer

class biothings.databuild.syncer.**ESJsonDiffSyncer**(*job_manager*, *log_folder*)

ESJsonDiffSelfContainedSyncer

class biothings.databuild.syncer.**ESJsonDiffSelfContainedSyncer**(*job_manager*, *log_folder*)

2.4 Web component

The BioThings SDK web component contains tools used to generate and customize an API, given an Elasticsearch index with data. The web component uses the Tornado Web Server to respond to incoming API requests.

2.4.1 Server boot script

BioThings API ioloop start utilities.

This module contains functions to configure and start the base event loop from command line args. Command line processing is done using tornado.options, with the following arguments defined:

- `port`: the port to start the API on, **default** 8000
- `address`: the address to start the API on, **default** 127.0.0.1
- `debug`: start the API in debug mode, **default** False
- `appdir`: path to API configuration directory, **default**: current working directory

The **main** function is the boot script for all BioThings API webservers.

index_base.**main**(*app_settings={}*, *debug_settings={}*, *sentry_client_key=None*)
Main ioloop configuration and start

Parameters

- **APP_LIST** – a list of URLSpec objects or (regex, handler_class) tuples
- **app_settings** – Tornado application settings
- **debug_settings** – Additional application settings for API debug mode
- **sentry_client_key** – Application-specific key for attaching Sentry monitor to the application

2.4.2 Settings

Settings objects used to configure the www API These settings get passed into the handler.initialize() function, of each request, and configure the web API endpoint. They are mostly a container for the *Config module*, and any other settings that are the same across all handler types, e.g. the Elasticsearch client.

Config module

BiothingWebSettings

class biothings.www.settings.**BiothingWebSettings**(*config='biothings.www.settings.default'*)
A container for the settings that configure the web API

The `config` init parameter specifies a module that configures this biothing. For more information see *config module* documentation.

generate_app_list()
Generates the tornado.web.Application (regex, handler_class, options) tuples for this project.

set_debug_level(*debug=False*)
Set if running API in debug mode. Should be called before passing `self` to handler initialization.

validate()
Validate these settings

BiothingESWebSettings

class biothings.www.settings.**BiothingESWebSettings**(*config='biothings.www.settings.default'*)
BiothingWebSettings subclass with functions specific to an elasticsearch backend

The `config` init parameter specifies a module that configures this biothing. For more information see *config module* documentation.

doc_url(*bid*)
Function to return a url on this biothing API to the biothing object specified by bid.

get_es_client()
Get the Elasticsearch client for this app, only called once on invocation of server.

source_metadata()
Function to cache return of the source metadata stored in _meta of index mappings

2.4.3 Handlers

BaseHandler

class biothings.www.api.helper.**BaseHandler**(*application*, *request*, ***kwargs*)

Parent class of all biothings handlers, only direct descendant of tornado.web.RequestHandler, contains the common functions in the biothings handler universe:

- return *self* as JSON
- set CORS and caching headers
- typify the URL keyword arguments
- optionally send tracking data to google analytics and integrate with sentry monitor

ga_event_object(*data={}*)

Create the data object for google analytics tracking.

get_query_params()

Extract, typify, and sanitize the parameters from the URL query string.

initialize(*web_settings*)

Tornado handler initialize(), Override to add settings for *this* biothing API. Assumes that the `web_settings` kwarg exists in APP_LIST

log_exceptions(*exception_msg=''*)

Logs the current exception in tornado logs and in hipchat room if available. This must be called in an exception handler

return_json(*data*, *encode=True*, *indent=None*)

Return passed data object as JSON response. If **jsonp** parameter is set in the request, return a valid JSONP response.

Parameters

- **data** – object to return as JSON
- **encode** – if encode is False, assumes input data is already a JSON encoded string.
- **indent** – number of indents per level in JSON string

set_cacheable(*etag=None*)

set proper header to make the response cacheable. set etag if provided.

support_cors(**args*, ***kwargs*)

Provide server side support for CORS request.

BaseESRequestHandler

class biothings.www.api.es.handlers.base_handler.**BaseESRequestHandler**(*application*, *request*, ***kwargs*)

Parent class of all Elasticsearch-based Request handlers, subclass of *BaseHandler*. Contains handling for Elasticsearch-specific query params (like `fields`, `size`, etc)

get_cleaned_options(*kwargs*)

Separate URL keyword arguments into their functional category. Incoming kwargs are separated into one of 4 categories, depending on how the argument controls the pipeline:

- `control_kwargs` - These are arguments that control the pipeline execution (**raw**, **rawquery**, etc)

- `es_kwargs` - These are arguments that get passed directly to the Elasticsearch client during query
- `esqb_kwargs` - These are arguments that go to the Elasticsearch query builder (**fields**, **size**, etc)
- `transform_kwargs` - These are arguments that go to the Elasticsearch result transformer (**jsonld**, **dotfield**, etc)

initialize(*web_settings*)
Tornado reqeust handler initialization. Initializations common to all Elasticsearch-specific request handlers go here.

return_raw_query_json(*query*)
Return valid JSON if *rawquery* option is selected. This is necessary as queries can span multiple lines (POST)

BiothingHandler

class biothings.www.api.es.handlers.biothing_handler.**BiothingHandler**(*application*, *request*, ***kwargs*)

Request handlers for requests to the annotation lookup endpoint

get(*bid=None*)
Handle a GET to the annotation lookup endpoint.

initialize(*web_settings*)
Tornado handler .initialize() function for all requests to the annotation lookup endpoint. Here, the allowed arguments are set (depending on the request method) for each kwarg category.

post(*ids=None*)
Handle a POST to the annotation lookup endpoint

QueryHandler

class biothings.www.api.es.handlers.query_handler.**QueryHandler**(*application*, *request*, ***kwargs*)

Request handlers for requests to the query endpoint

get()
Handle a GET to the query endpoint.

initialize(*web_settings*)
Tornado handler .initialize() function for all requests to the query endpoint. Here, the allowed arguments are set (depending on the request method) for each kwarg category.

post()
Handle a POST to the query endpoint.

MetadataHandler

class biothings.www.api.es.handlers.metadata_handler.**MetadataHandler**(*application*, *request*, ***kwargs*)

Request handlers for requests to the metadata endpoint.

get()
Handle a GET to the metadata endpoint. Also handles /metadata/fields.

initialize(*web_settings*)
Tornado handler .initialize() function for all requests to the metadata endpoint. Here, the allowed arguments are set (depending on the request method) for each kwarg category.

2.4.4 Elasticsearch Query Builder

class biothings.www.api.es.query_builder.**ESQueries**(*es_kwargs={}*)
A very simple class to object-ize Elasticsearch Query DSL. This should be replaced by the official Elasticsearch equivalent. Also contains a simple JSON validator after generating all queries (dump to string and re-read)

bool(*query_kwargs*)
Given `query_kwargs`, validate and return a **bool** query.

match(*query_kwargs*)
Given `query_kwargs`, validate and return a **match** query.

match_all(*query_kwargs*)
Given `query_kwargs`, validate and return a **match_all** query.

multi_match(*query_kwargs*)
Given `query_kwargs`, validate and return a **multi_match** query.

query_string(*query_kwargs*)
Given `query_kwargs`, validate and return a **query_string** query.

raw_query(*raw_query*)
Given `query_kwargs`, validate and return a *raw* query (queries that don't fit the same query_template).

class biothings.www.api.es.query_builder.**ESQueryBuilder**(*index, doc_type, options, es_options, scroll_options={}, userquery_dir='', regex_list=[], default_scopes=['_id']*)
Class to return the correct query given the request endpoint, method, and URL params.

Parameters

- **index** – The Elasticsearch index to run the query on
- **doc_type** – The Elasticsearch document type of the query
- **options** – Options from the URL string relevant to query building
- **es_options** – Options for Elasticsearch query stage
- **scroll_options** – Options for scroll requests
- **regex_list** – A list of (regex, scope) tuples for annotation lookup
- **userquery_dir** – The directory containing user queries for this app
- **default_scopes** – A list representing the default Elasticsearch query scope(s) for this query

add_query_filters(*_query*)
Given a query, add any other filters

annotation_GET_query(*bid*)
Return an annotation lookup GET query for this `bid`.

Parameters **bid** – Biothing ID, used to lookup the annotation

annotation_POST_query(*bids*)
Return an annotation lookup POST query for these `bids`.

Parameters **bids** – Biothing IDs, used to lookup the annotations

get_query_filters()
Override me to add more query filters

metadata_query()
Return a metadata query

query_GET_query(*q*)
Return a query endpoint GET query for this query string `q`.

Parameters **q** – query string specifying the query

query_POST_query(*qs*, *scopes*)
Return query endpoint POST queries for these query strings `qs`.

Parameters

- **qs** – Query strings to query
- **scopes** – Scope of query strings `qs`

scroll(*scroll_id*)
Return the next batch of results from a *scroll* query.

Parameters **scroll_id** – ID of the batch to yield hits from

2.4.5 Elasticsearch Query

class biothings.www.api.es.query.**ESQuery**(*client*, *options={}*)
This class contains functions to execute the *query* section of all handler pipelines. The inputs to it are an Elasticsearch client (from *BiothingESWebSettings*), and any options from the URL string. Each handler calls a different query function, though they all do essentially the same thing: get the query generated in the ESQueryBuilder stage of the pipeline (`query_kwargs`), and run it using the supplied Elasticsearch client.

annotation_GET_query(*query_kwargs*)
Given `query_kwargs` from ESQueryBuilder, return results of annotation lookup GET query on ES client.

annotation_POST_query(*query_kwargs*)
Given `query_kwargs` from ESQueryBuilder, return results of annotation lookup POST query on ES client.

get_biothing(*query_kwargs*)
Return a biothing using the Elasticsearch client.get function

metadata_query(*query_kwargs*)
Given `query_kwargs` from ESQueryBuilder, return results of metadata query on ES client.

query_GET_query(*query_kwargs*)
Given `query_kwargs` from ESQueryBuilder, return results of query GET on ES client.

query_POST_query(*query_kwargs*)
Given `query_kwargs` from ESQueryBuilder, return results of query POST on ES client.

scroll(*query_kwargs*)
Given `query_kwargs` from ESQueryBuilder, return results of a scroll on ES client - returns next batch of results.

2.4.6 Elasticsearch Result Transformer

class biothings.www.api.es.transform.**ESResultTransformer**(*options*, *host*, *doc_url_function=<function ESResultTransformer.<lambda>>*, *jsonld_context={}*, *data_sources={}*, *output_aliases={}*, *app_dir=''*, *source_metadata={}*)

Class to transform the results of the Elasticsearch query generated prior in the pipeline. This contains the functions to extract the final document from the elasticsearch query result in *Elasticsearch Query*. This also contains the code to flatten a document (if **dotfield** is True), or to add JSON-LD context to the document (if **jsonld** is True).

Parameters

- **options** – Options from the URL string controlling result transformer
- **host** – Host name (extracted from request), used for JSON-LD address generation
- **doc_url_function** – a function that takes one argument (a biothing id) and returns a URL to that biothing
- **jsonld_context** – JSON-LD context for this app (optional)
- **data_sources** – unused currently (optional)
- **output_aliases** – list of output key names to alias, unused currently (optional)
- **app_dir** – Application directory for this app (used for getting app information in /metadata)
- **source_metadata** – Metadata object containing source information for _license keys

clean_annotation_GET_response(*res*)

Transform the results of a GET to the annotation lookup endpoint.

Parameters **res** – Results from *Elasticsearch Query*.

clean_annotation_POST_response(*bid_list*, *res*, *single_hit=True*)

Transform the results of a POST to the annotation lookup endpoint.

Parameters

- **bid_list** – List of biothing id inputs
- **res** – Results from *Elasticsearch Query*
- **single_hit** – If `True`, render queries with 1 result as a dictionary, else as a 1-element list containing a dictionary

clean_metadata_response(*res*, *fields=False*)

Transform the results of a GET to the metadata endpoint.

Parameters **res** – Results from *Elasticsearch Query*.

clean_query_GET_response(*res*)

Transform the results of a GET to the query endpoint.

Parameters **res** – Results from *Elasticsearch Query*.

clean_query_POST_response(*qlist*, *res*, *single_hit=True*)

Transform the results of a POST to the query endpoint.

Parameters

- **qlist** – List of query inputs
- **res** – Results from *Elasticsearch Query*
- **single_hit** – If `True`, render queries with 1 result as a dictionary, else as a 1-element list containing a dictionary

clean_scroll_response(*res*)

Transform the results of a GET to the scroll endpoint

Parameters **res** – Results from *Elasticsearch Query*.

Python Module Index

b

Index

E

F

G

H

I

J

V

W