PubFetcher

Release 1.1.2-SNAPSHOT

Feb 06, 2023

Contents:

A Java command-line tool and library to download and store publications with metadata by combining content from various online resources (Europe PMC, PubMed, PubMed Central, Unpaywall, journal web pages), plus extract content from general web pages.

CHAPTER 1

What is PubFetcher?

A Java command-line tool and library to download and store publications with metadata by combining content from various online resources (Europe PMC, PubMed, PubMed Central, Unpaywall, journal web pages), plus extract content from general web pages.

1.1 Overview

PubFetcher used to be part of EDAMmap until its functionality was determined to be potentially useful on its own, thus PubFetcher is now an independently usable application. However, its features and structure are still influenced by EDAMmap, for example the supported *publication resources* are mainly from the biomedical and life sciences fields and getting the list of authors of a publication is currently not supported (as it's not needed in EDAMmap). Also, the functionality of extracting content from *general web pages* is geared towards web pages containing software tools descriptions and documentation (GitHub, BioConductor, etc), as PubFetcher has built-in rules to extract from these pages and it has fields to store the *software license* and *programming language*.

Ideally, all scientific literature would be open and easily accessible through one interface for text mining and other purposes. One interface for getting publications is Europe PMC, which PubFetcher uses as its main resource. In the middle of 2018, Europe PMC was able to provide almost all of the titles, around 95% of abstracts, 50% of full texts and only 10% of user-assigned keywords for the publications present in the bio.tools registry at that time. While some articles don't have keywords and some full texts can't be obtained, many of the gaps can be filled by other *resources*. And sometimes we need the maximum amount of content about each publication for better results, thus the need for PubFetcher, that extracts and combines data from these different resources.

The speed of downloading, when *multithreading* is enabled, is roughly one publication per second. This limitation, along with the desire to not overburden the used APIs and publisher sites, means that PubFetcher is best used for medium-scale processing of publications, where the number of entries is in the thousands and not in the millions, but where the largest amount of completeness for these few thousand publications is desired. If millions of publications are required, then it is better to restrict oneself to the Open Access subset, which can be downloaded in bulk: https://europepmc.org/downloads.

In addition to the main content of a publication (*title*, *abstract* and *full text*), PubFetcher supports getting different keywords about the publication: the *user-assigned keywords*, the *MeSH terms* as assigned in PubMed and *EFO terms* and *GO terms* as mined from the full text by Europe PMC. Each publication has up to three identificators: a *PMID*, a

PMCID and a *DOI*. In addition, different metadata (found from the different *resources*) about a publication is saved, like whether the article is *Open Access*, the *journal* where it was published, the *publication date*, etc. The *source* of each *publication part* is remembered, with content from a higher confidence resource potentially overwriting the current content. It is possible to fetch only some *publication parts* (thus avoiding querying some *resources*) and there is *an algorithm* to determine if an already existing entry should be refetched or is it complete enough. Fetching and *extracting* of content is done using various Java libraries with support for *JavaScript* and *PDF* files. The downloaded publications can be persisted to disk to a *key-value store* for later analysis. A number of *built-in rules* are included (along with *tests*) for *scraping* publication parts from publisher sites, but additional rules can also be defined. Currently, there is support for around 50 publishers of journals and 25 repositories of tools and tools' metadata and documentation and around 750 test cases for the rules have been defined. If no rules are defined for a given site, then *automatic cleaning* is applied to get the main content of the page.

PubFetcher has an extensive *command-line tool* to use all of its functionality. It contains a few *helper operations*, but the main use is the construction of a simple *pipeline* for querying, fetching and outputting of publications and general and documentation web pages: first IDs of interest are specified/loaded and filtered, then corresponding content fetched/loaded and filtered, and last it is possible to output the results or store them to a database. Among other functionality, content and all the metadata can be output in *HTML or plain text*, but also *exported* to *JSON*. All fetching operations can be influenced by a few *general parameters*. Progress along with error messages is logged to the console and to a *log file*, if specified. The command-line tool can be *extended*, for example to add new ways of loading IDs.

1.2 Outline

- *Command-line interface manual* documents all parameters of the command-line interface, accompanied by many examples
- *Output* describes different outputs: the database, the log file and the JSON output, through which the structure of publications, webpages and docs is also explained
- *Fetching logic* deals with fetching logic, describing for example the content fetching methods and the resources and filling logic of publication parts
- *Scraping rules* is about scraping rules and how to define and test them
- *Programming reference* gives a short overview about the source code for those wanting to use the PubFetcher library
- *Ideas for future* contains ideas how to improve PubFetcher

1.3 Install

Installation instructions can be found in the project's GitHub repo at INSTALL.

1.4 Quickstart

```
# Create a new empty database
$ java -jar pubfetcher-cli-<version>.jar -db-init database.db
# Fetch two publications and store them to the database
$ java -jar pubfetcher-cli-<version>.jar -pub 10.1093/nar/gkz369 10.1101/692905 -db-
↪fetch-end database.db
# Print the fetched publications
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db -db database.db -out
```

For many more examples, see *Examples*.

1.5 Repo

PubFetcher is hosted at https://github.com/edamontology/pubfetcher.

1.6 Support

Should you need help installing or using PubFetcher, please get in touch with Erik Jaaniso (the lead developer) directly via the tracker.

1.7 License

PubFetcher is free and open-source software licensed under the GNU General Public License v3.0, as seen in COPYING.

CHAPTER 2

Command-line interface manual

The CLI of PubFetcher is provided by a Java executable packaged in a .jar file. If a Java Runtime Environment (JRE) capable of running version 8 of Java is installed on the system, then this .jar file can be executed using the `java` command. For example, executing PubFetcher-CLI with the parameter `-h` or `--help` outputs a list of all possible parameters:

```
$ java -jar path/to/pubfetcher-cli-<version>.jar --help
```

Parsing of command line parameters is provided by JCommander.

2.1 Logging

Parameter	Description
`-l` or `--log`	The path of the log file

PubFetcher-CLI will output its log to the console (to stderr). With the `--log` parameter we can specify a text file location where this same log will be output. It will not be coloured as the console output, but will include a few DEBUG level messages omitted in the console (this includes the very first line listing all parameters the program was run with).

If the specified file already exists, then new log messages will be appended to its end. In case of a new log file creation, any missing parent directories will be created as necessary.

2.2 General parameters

Parameters affecting many other operations specified below. These parameters can be supplied to PubFetcher externally through programmatic means. When supplied on the command line (of PubFetcher-CLI), then two dashes (`--`) have to be added in front of the parameter names specified in the two following tables.

2.2.1 Fetching

Parameters that affect many of the operations specified further below, for example `--timeout` changes the timeouts of all attempted network connections. The cooldown and *retryLimit* parameters affect if we *can fetch* (or rather, refetch) a *publication* or *webpage*. The minimum length and size parameters affect whether an entry is usable and final.

Parameter	Default	Min	Description
emptyCooldown	`720`		If that many minutes have passed since last fetching attempt of an *empty publication* or *empty webpage*, then fetching can be attempted again, resetting the *retryCounter*. Setting to `0` means fetching of empty *database* entries will always be attempted again. Setting to a negative value means refetching will never be done (and retryCounter never reset) only because the entry is empty.
nonFinalCooldown	`10080`		If that many minutes have passed since last fetching attempt of a non-*final publication* or non-*final webpage* (which are not empty), then fetching can be attempted again, resetting the *retryCounter*. Setting to `0` means fetching of non-final database entries will always be attempted again. Setting to a negative value means refetching will never be done (and retryCounter never reset) only because the entry is non-final.
fetchExceptionCooldown	`1440`		If that many minutes have passed since last fetching attempt of a *publication* or *webpage* with a *fetchException*, then fetching can be attempted again, resetting the *retryCounter*. Setting to `0` means fetching of database entries with fetchException will always be attempted again. Setting to a negative value means refetching will never be done (and retryCounter never reset) only because the fetchException of the entry is `true`.
retryLimit	`3`		How many times can fetching be retried for an entry that is still empty, non-final or has a *fetchException* after the initial attempt. Setting to `0` will disable retrying, unless the *retryCounter* is reset by a cooldown in which case one initial attempt is allowed again. Setting to a negative value will disable this upper limit.
titleMinLength	`4`	`0`	Minimum length of a *usable publication title*
keywordsMinSize	`2`	`0`	Minimum size of a *usable publication keywords*/*MeSH* list
minedTermsMinSize	`1`	`0`	Minimum size of a *usable publication EFO*/*GO* terms list
abstractMinLength	`200`	`0`	Minimum length of a *usable publication abstract*
fulltextMinLength	`2000`	`0`	Minimum length of a *usable publication fulltext*
webpageMinLength	`50`	`0`	Minimum length of a *usable webpage* combined *title* and *content*
webpageMinLengthJavascript	`200`	`0`	If the length of a the whole web page text fetched without JavaScript is below the specified limit and no *scraping rules* are found for the corresponding URL, then refetching using JavaScript support will be attempted
timeout	`15000`	`0`	Connect and read timeout of connections, in milliseconds

2.2.2 Fetching private

These are like *Fetching* parameters in that they have a general effect, e.g. setting `--userAgent` changes the HTTP User-Agent of all HTTP connections. However, *Fetching* parameters are such parameters that we might want to expose via a web API to be changeable by a client (when extending or using the PubFetcher library), but the parameters below should probably only be configured locally and as such are separated in code.

Parameter	Description
europepmcEmail	E-mail to send to the *Europe PMC* API
oadoiEmail	E-mail to send to the oaDOI (*Unpaywall*) API
userAgent	HTTP User-Agent
journalsYaml	YAML file containing custom *journals scrape rules* to add to default ones
webpagesYaml	YAML file containing custom *webpages scrape rules* to add to default ones

2.3 Simple one-off operations

Some simple operations (represented by the parameters with one dash (`-`) below), that mostly should by the sole parameter supplied to PubFetcher, when used.

2.3.1 Database

A collection of one-off database operations on a single *database* file.

Parameter	Parameter args	Description
`-db-init`	<*database file*>	Create an empty database file. This is the only way to make new databases.
`-db-commit`	<*database file*>	Commit all pending changes by merging all WAL files to the main database file. This has only an effect if WAL files are present beside the database file after an abrupt termination of the program, as normally committing is done in code where required.
`-db-compact`	<*database file*>	Compaction reclaims space by removing deprecated records (left over after database updates)
`-db-publications-size`	<*database file*>	Output the number of *publications* stored in the database to stdout
`-db-webpages-size`	<*database file*>	Output the number of *webpages* stored in the database to stdout
`-db-docs-size`	<*database file*>	Output the number of *docs* stored in the database to stdout
`-db-publications-map`	<*database file*>	Output all *PMID* to primary ID, *PMCID* to primary ID and *DOI* to primary ID mapping pairs stored in the database to stdout
`-db-publications-map-reverse`	<*database file*>	Output all mappings from primary ID to the triple [*PMID*, *PMCID*, *DOI*] stored in the database to stdout

2.3.2 Print a web page

Methods for fetching and outputting a web page. Affected by *timeout* and *userAgent* parameters, `-fetch-webpage-selector` also by *webpageMinLength* and *webpageMinLengthJavascript*.

Parameter	Parameter args	Description
`-fetch-document`	*<url>*	Fetch a web page (without JavaScript support, i.e. using jsoup) and output its raw HTML to stdout
`-fetch-document-javascript`	*<url>*	Fetch a web page (with JavaScript support, i.e. using HtmlUnit) and output its raw HTML to stdout
`-post-document`	*<url>* *<param name>* *<param value>* *<param name>* *<param value>* …	Fetch a web resource using HTTP POST. The first parameter specifies the resource URL and is followed by the request data in the form of name/value pairs, with names and values separated by spaces.
`-fetch-webpage-selector`	*<url>* *<title selector>* *<content selector>* *<javascript support>*	Fetch a *webpage* and output it to stdout in the format specified by the *Output modifiers* `--plain` and `--format`. Works also for PDF files. *Title* and *content* args are CSS selectors as supported by jsoup. If the *title selector* is an empty string, then the *page title* will be the text content of the document's `<title>` element. If the *content selector* is an empty string, then *content* will be the *automatically cleaned* whole text content parsed from the HTML/XML. If javascript arg is `true`, then fetching will be done using JavaScript support (HtmlUnit), if `false`, then without JavaScript (jsoup). If javascript arg is empty, then fetching will be done without JavaScript and if the text length of the returned document is less than *webpageMinLengthJavascript* or if a `<noscript>` tag is found in it, a second fetch will happen with JavaScript support.

2.3.3 Scrape rules

Print requested parts of currently effective *scraping rules* loaded from default or custom scrape rules *YAML files*.

Parameter	Parameter args	Description
`-scrape-site`	*<url>*	Output found journal site name for the given URL to stdout (or `null` if not found or URL invalid)
`-scrape-selector`	*<url>* *<ScrapeSiteKey>*	Output the CSS selector used for extracting the *publication part* represented by *ScrapeSiteKey* from the given URL
`-scrape-javascript`	*<url>*	Output `true` or `false` depending on whether JavaScript will be used or not for fetching the given publication URL
`-scrape-webpage`	*<url>*	Output all CSS selectors used for extracting webpage content and metadata from the given URL (or `null` if not found or URL invalid)

2.3.4 Publication IDs

Simple operations on *publication IDs*, with result output to stdout.

Parameter	Parameter args	Description
`-is-pmid`	*<string>*	Output `true` or `false` depending on whether the given string is a valid *PMID* or not
`-is-pmcid`	*<string>*	Output `true` or `false` depending on whether the given string is a valid *PMCID* or not
`-extract-pmcid`	*<pmcid>*	Remove the prefix "PMC" from a *PMCID* and output the rest. Output an empty string if the given string is not a valid PMCID.
`-is-doi`	*<string>*	Output `true` or `false` depending on whether the given string is a valid *DOI* or not
`-normalise-doi`	*<doi>*	Remove any valid prefix (e.g. "https://doi.org/", "doi:") from a *DOI* and output the rest, converting letters from the 7-bit ASCII set to uppercase. The validity of the input DOI is not checked.
`-extract-doi-registrant`	*<doi>*	Output the registrant ID of a *DOI* (the substring after "10." and before "/"). Output an empty string if the given string is not a valid DOI.

2.3.5 Miscellaneous

Methods to test the escaping of HTML entities as done by PubFetcher (necessary when outputting raw input to HTML format) and test the validity of *publication IDs* and *webpage URLs*.

Parameter	Parameter args	Description
`-escape-html`	*<string>*	Output the result of escaping necessary characters in the given string such that it can safely by used as text in a HTML document (without the string interacting with the document's markup)
`-escape-html-attribute`	*<string>*	Output the result of escaping necessary characters in the given string such that it can safely by used as an HTML attribute value (without the string interacting with the document's markup)
`-check-publication-id`	*<string>*	Given one publication ID, output it in publication IDs form (`<pmid>\t<pmcid>\t<doi>`) if it is a valid *PMID*, *PMCID* or *DOI*, or throw an exception if it is an invalid publication ID
`-check-publication-ids`	*<pmid>* *<pmcid>* *<doi>*	Given a *PMID*, a *PMCID* and a *DOI*, output them in publication IDs form (`<pmid>\t<pmcid>\t<doi>`) if given IDs are a valid PMID, PMCID and DOI, or throw an exception if at least one is invalid
`-check-url`	*<string>*	Given a webpage ID (i.e. a URL), output the parsed URL, or throw an exception if it is an invalid URL

2.4 Pipeline of operations

A simple pipeline that allows for more complex querying, fetching and outputting of *publications* **,** *webpages* **and** *docs* **: first IDs of interest are specified/loaded and filtered, then corresponding content fetched/loaded and filtered, and last it is possible to output or store the results.** Component operations of the pipeline are specified as command-line parameters with one dash (`-`). In addition, there are some parameters modifying some aspect of the pipeline, these will have two dashes (`--`). The *Fetching* and *Fetching private* parameters will also have an effect (on fetching and determining the finality of content).

2.4.1 Add IDs

publication IDs, *webpage URLs* and *doc URLs* can be specified on the command-line and can be loaded from text and *database* files. The resultant list of IDs is actually a set, meaning that if duplicate IDs are encountered, they'll be ignored and not added to the list.

Parameter	Parameter args	Description
`-pub`	*<string>* *<string>* ...	A space-separated list of *publication IDs* (either *PMID*, *PMCID* or *DOI*) to add
`-web`	*<string>* *<string>* ...	A space-separated list of *webpage URLs* to add
`-doc`	*<string>* *<string>* ...	A space-separated list of *doc URLs* to add
`-pub-file`	*<text file>* ...	Load all *publication IDs* from the specified list of text files containing publication IDs in the form `<pmid>\t<pmcid>\t<doi>`, one per line. Empty lines and lines beginning with # are ignored.
`-web-file`	*<text file>* ...	Load all *webpage URLs* from the specified list of text files containing webpage URLs, one per line. Empty lines and lines beginning with # are ignored.
`-doc-file`	*<text file>* ...	Load all *doc URLs* from the specified list of text files containing doc URLs, one per line. Empty lines and lines beginning with # are ignored.
`-pub-db`	*<database file>* ...	Load all *publication IDs* found in the specified *database* files
`-web-db`	*<database file>* ...	Load all *webpage URLs* found in the specified *database* files
`-doc-db`	*<database file>* ...	Load all *doc URLs* found in the specified *database* files

2.4.2 Filter IDs

Conditions that *publication IDs*, *webpage URLs* and *doc URLs* must meet to be retained in the *pipeline*.

Parameter	Parameter args	Description
`-has-pmid`		Only keep *publication IDs* whose *PMID* is present
`-not-has-pmid`		Only keep *publication IDs* whose *PMID* is empty
`-pmid`	<*regex*>	Only keep *publication IDs* whose *PMID* has a match with the given regular expression
`-not-pmid`	<*regex*>	Only keep *publication IDs* whose *PMID* does not have a match with the given regular expression
`-pmid-url`	<*regex*>	Only keep *publication IDs* whose *PMID provenance URL* has a match with the given regular expression
`-not-pmid-url`	<*regex*>	Only keep *publication IDs* whose *PMID provenance URL* does not have a match with the given regular expression
`-has-pmcid`		Only keep *publication IDs* whose *PMCID* is present
`-not-has-pmcid`		Only keep *publication IDs* whose *PMCID* is empty
`-pmcid`	<*regex*>	Only keep *publication IDs* whose *PMCID* has a match with the given regular expression
`-not-pmcid`	<*regex*>	Only keep *publication IDs* whose *PMCID* does not have a match with the given regular expression
`-pmcid-url`	<*regex*>	Only keep *publication IDs* whose *PMCID provenance URL* has a match with the given regular expression
`-not-pmcid-url`	<*regex*>	Only keep *publication IDs* whose *PMCID provenance URL* does not have a match with the given regular expression
`-has-doi`		Only keep *publication IDs* whose *DOI* is present
`-not-has-doi`		Only keep *publication IDs* whose *DOI* is empty
`-doi`	<*regex*>	Only keep *publication IDs* whose *DOI* has a match with the given regular expression
`-not-doi`	<*regex*>	Only keep *publication IDs* whose *DOI* does not have a match with the given regular expression
`-doi-url`	<*regex*>	Only keep *publication IDs* whose *DOI provenance URL* has a match with the given regular expression
`-not-doi-url`	<*regex*>	Only keep *publication IDs* whose *DOI provenance URL* does not have a match with the given regular expression
`-doi-registrant`	<*string*> <*string*> ...	Only keep *publication IDs* whose *DOI* registrant code (the bit after "10." and before "/") is present in the given list of strings
`-not-doi-registrant`	<*string*> <*string*> ...	Only keep *publication IDs* whose *DOI* registrant code (the bit after "10." and before "/") is not present in the given list of strings
`-url`	<*regex*>	Only keep *webpage URLs* and *doc URLs* that have a match with the given regular expression
`-not-url`	<*regex*>	Only keep *webpage URLs* and *doc URLs* that don't have a match with the given regular expression
`-url-host`	<*string*> <*string*> ...	Only keep *webpage URLs* and *doc URLs* whose host part is present in the given list of strings (comparison is done case-insensitively and "www." is removed)
`-not-url-host`	<*string*> <*string*> ...	Only keep *webpage URLs* and *doc URLs* whose host part is not present in the given list of strings (comparison is done case-insensitively and "www." is removed)
`-in-db`	<*database file*>	Only keep *publication IDs*, *webpage URLs* and *doc URLs* that are present in the given *database* file
`-not-in-db`	<*database file*>	Only keep *publication IDs*, *webpage URLs* and *doc URLs* that are not present in the given *database* file

2.4.3 Sort IDs

Sorting of added and filtered IDs. *publication IDs* are first sorted by *PMID*, then by *PMCID* (if PMID is absent), then by *DOI* (if PMID and PMCID are absent). Internally, the PMID, the PMCID and the DOI registrant are sorted numerically, DOIs within the same registrant alphabetically. *webpage URLs* and *doc URLs* are sorted alphabetically.

Parameter	Parameter args	Description
`-asc-ids`		Sort *publication IDs*, *webpage URLs* and *doc URLs* in ascending order
`-desc-ids`		Sort *publication IDs*, *webpage URLs* and *doc URLs* is descending order

2.4.4 Limit IDs

Added, filtered and sorted IDs can be limited to a given number of IDs either in the front or back.

Parameter	Parameter args	Description
`-head-ids`	*<positive integer>*	Only keep the first given number of *publication IDs*, *webpage URLs* and *doc URLs*
`-tail-ids`	*<positive integer>*	Only keep the last given number of *publication IDs*, *webpage URLs* and *doc URLs*

2.4.5 Remove from database by IDs

The resulting list of IDs can be used to remove corresponding entries from a *database*.

Parameter	Parameter args	Description
`-remove-ids`	*<database file>*	From the given *database*, remove content corresponding to *publication IDs*, *webpage URLs* and *doc URLs*

2.4.6 Output IDs

Outputs the final list of loaded IDs to stdout or the specified text files in the format specified by the *Output modifiers* `--plain` and `--format`. Without `--plain` *publication IDs* are output with their corresponding provenance URLs, with `--plain` these are omitted. *webpage URLs* and *doc URLs* are not affected by `--plain`. Specifying `--format` as `text` (the default) and using `--plain` will output *publication IDs* in the form `<pmid>\t<pmcid>\t<doi>`.

Parameter	Parameter args	Description
`-out-ids`		Output *publication IDs*, *webpage URLs* and *doc URLs* to stdout in the format specified by the *Output modifiers* `--plain` and `--format`
`-txt-ids-pub`	*<file>*	Output *publication IDs* to the given file in the format specified by the *Output modifiers* `--plain` and `--format`
`-txt-ids-web`	*<file>*	Output *webpage URLs* to the given file in the format specified by `--format`
`-txt-ids-doc`	*<file>*	Output *doc URLs* to the given file in the format specified by `--format`
`-count-ids`		Output count numbers for *publication IDs*, *webpage URLs* and *doc URLs* to stdout

2.4.7 Get content

Operations to get *publications*, *webpages* and *docs* corresponding to the final list of loaded *publication IDs*, *webpage URLs* and *doc URLs*. Content will be fetched from the Internet, loaded from a *database* file, or both, with updated content possibly saved back to the database. In case multiple content getting operations are used, first everything with `-db` is got, then `-fetch`, `-fetch-put`, `-db-fetch` and last `-db-fetch-end`. The list of entries will have the order in which entries were got, duplicates are allowed. When saved to a database file, duplicates will be merged, in other cases (e.g. when outputting content) duplicates will be present.

Parameter	Parameter args	Description
`-db`	*<database file>*	Get *publications*, *webpages* and *docs* from the given *database*
`-fetch`		Fetch *publications*, *webpages* and *docs* from the Internet. All entries for which some *fetchException* happens are fetched again in the end (this is done only once).
`-fetch-put`	*<database file>*	Fetch *publications*, *webpages* and *docs* from the Internet and put each entry in the given *database* right after it has been fetched, ignoring any filters and overwriting any existing entries with equal IDs/URLs. All entries for which some *fetchException* happens are fetched and put to the database again in the end (this is done only once).
`-db-fetch`	*<database file>*	First, get an entry from the given *database* (if found), then fetch the entry (if the entry *can be fetched*), then put the entry back to the database while ignoring any filters (if the entry was updated). All entries which have the *fetchException* set are got again in the end (this is done only once). This operation is multithreaded (in contrast to `-fetch` and `-fetch-put`), with `--threads` number of threads, thus it should be preferred for larger amounts of content.
`-db-fetch-end`	*<database file>*	Like `-db-fetch`, except no content is kept in memory (saving back to the given *database* still happens), thus no further processing down the *pipeline* is possible. This is useful for avoiding large memory usage if only fetching and saving of content to the database is to be done and no further operations on content (like outputting it) are required.

2.4.8 Get content modifiers

Some parameters to influence the behaviour of content getting operations.

Parameter	Parameter args	Default	Description
`--fetch-part`	*<PublicationPartName>* …		List of publication parts that will be fetched from the Internet. All other parts will be *empty* (except the publication IDs which will be filled whenever possible). Fetching of *resources* not containing any specified parts will be skipped. If used, then `--not-fetch-part` must not be used. If neither of `--fetch-part` and `--not-fetch-part` is used, then all parts will be fetched.
`--not-fetch-part`	*<PublicationPartName>* …		List of publication parts that will not be fetched from the Internet. All other parts will be fetched. Fetching of *resources* not containing any not specified parts will be skipped. If used, then `--fetch-part` must not be used.
`--pre-filter`			Normally, all content is loaded into memory before filtering specified in *Filter content* is applied. This option ties the filtering step to the loading/fetching step for each individual entry, discarding entries not passing the filter right away, thus reducing memory usage. As a tradeoff, in case multiple filters are used, it won't be possible to see in the log how many entries were discarded by each filter.
`--limit`	*<positive integer>*	0	Maximum number of *publications*, *webpages* and *docs* that can be loaded/fetched. In case the limit is applied, the concrete returned content depends on the order it is loaded/fetched, which depends on the order of content getting operations, then on whether there was a *fetchException* and last on the ordering of received IDs. If the multithreaded `-db-fetch` is used or a fetchException happen, then the concrete returned content can vary slightly between equal applications of limit. If `--pre-filter` is also used, then the filters of *Filter content* will be applied before the limit, otherwise the limit is applied beforehand and the filters can reduce the number of entries further. Set to `0` to disable.
`--threads`	*<positive integer>*	8	Number of threads used for getting content with `-db-fetch` and `-db-fetch-end`. Should not be bound by actual processor core count, as mostly threads sit idle, waiting for an answer from a remote host or waiting behind another thread to finish communicating with the same host.

2.4.9 Filter content

Conditions that *publications*, *webpages* and *docs* must meet to be retained in the *pipeline*. All filters will be ANDed together.

Parameter	Parameter args	Description
`-fetch-time-more`	*<ISO-8601 time>*	Only keep *publications*, *webpages* and *docs* whose *fetchTime* is more than or equal to the given time
`-fetch-time-less`	*<ISO-8601 time>*	Only keep *publications*, *webpages* and *docs* whose *fetchTime* is less than or equal to the given time
`-retry-counter`	*<positive integer>* ...	Only keep *publications*, *webpages* and *docs* whose *retryCounter* is equal to one of given counts
`-not-retry-counter`	*<positive integer>* ...	Only keep *publications*, *webpages* and *docs* whose *retryCounter* is not equal to any of given counts
`-retry-counter-more`	*<positive integer>*	Only keep *publications*, *webpages* and *docs* whose *retryCounter* is more than the given count
`-retry-counter-less`	*<positive integer>*	Only keep *publications*, *webpages* and *docs* whose *retryCounter* is less than the given count
`-fetch-exception`		Only keep *publications*, *webpages* and *docs* with a *fetchException*
`-not-fetch-exception`		Only keep *publications*, *webpages* and *docs* without a *fetchException*
`-empty`		Only keep *empty publications*, *empty webpages* and empty docs
`-not-empty`		Only keep non-*empty publications*, non-*empty webpages* and non-empty docs
`-usable`		Only keep *usable publications*, *usable webpages* and usable docs
`-not-usable`		Only keep non-*usable publications*, non-*usable webpages* and non-usable docs
`-final`		Only keep *final publications*, *final webpages* and final docs
`-not-final`		Only keep non-*final publications*, non-*final webpages* and non-final docs
`-grep`	*<regex>*	Only keep *publications*, *webpages* and *docs* whose whole content (as output using `--plain`) has a match with the given regular expression
`-not-grep`	*<regex>*	Only keep *publications*, *webpages* and *docs* whose whole content (as output using `--plain`) does not have a match with the given regular expression

2.4.10 Filter publications

Conditions that *publications* must meet to be retained in the *pipeline*.

Parameter	Parameter args	Description
`-totally-final`		Only keep *publications* whose content is *totally final*
`-not-totally-final`		Only keep *publications* whose content is not *totally fin*
`-oa`		Only keep *publications* that are *Open Access*
`-not-oa`		Only keep *publications* that are not *Open Access*
`-journal-title`	*<regex>*	Only keep *publications* whose *journal title* has a matc
`-not-journal-title`	*<regex>*	Only keep *publications* whose *journal title* does not ha
`-journal-title-empty`		Only keep *publications* whose *journal title* is empty
`-not-journal-title-empty`		Only keep *publications* whose *journal title* is not empt
`-pub-date-more`	*<ISO-8601 time>*	Only keep *publications* whose *publication date* is mor
`-pub-date-less`	*<ISO-8601 time>*	Only keep *publications* whose *publication date* is less
`-citations-count`	*<positive integer>* ...	Only keep *publications* whose *citations count* is equal
`-not-citations-count`	*<positive integer>* ...	Only keep *publications* whose *citations count* is not ec
`-citations-count-more`	*<positive integer>*	Only keep *publications* whose *citations count* is more
`-citations-count-less`	*<positive integer>*	Only keep *publications* whose *citations count* is less t
`-citations-timestamp-more`	*<ISO-8601 time>*	Only keep *publications* whose *citations count last upd*
`-citations-timestamp-less`	*<ISO-8601 time>*	Only keep *publications* whose *citations count last upd*

Table 1 – continued fron

Parameter	Parameter args	Description
`-corresp-author-name`	<*regex*>	Only keep *publications* with a *corresponding author* n
`-not-corresp-author-name`	<*regex*>	Only keep *publications* with no *corresponding author*
`-corresp-author-name-empty`		Only keep *publications* whose *corresponding author*s
`-not-corresp-author-name-empty`		Only keep *publications* with a *corresponding author* n
`-corresp-author-orcid`	<*regex*>	Only keep *publications* with a *corresponding author* O
`-not-corresp-author-orcid`	<*regex*>	Only keep *publications* with no *corresponding author*
`-corresp-author-orcid-empty`		Only keep *publications* whose *corresponding author*s
`-not-corresp-author-orcid-empty`		Only keep *publications* with a *corresponding author* O
`-corresp-author-email`	<*regex*>	Only keep *publications* with a *corresponding author* e
`-not-corresp-author-email`	<*regex*>	Only keep *publications* with no *corresponding author*
`-corresp-author-email-empty`		Only keep *publications* whose *corresponding author*s
`-not-corresp-author-email-empty`		Only keep *publications* with a *corresponding author* e
`-corresp-author-phone`	<*regex*>	Only keep *publications* with a *corresponding author* t
`-not-corresp-author-phone`	<*regex*>	Only keep *publications* with no *corresponding author*
`-corresp-author-phone-empty`		Only keep *publications* whose *corresponding author*s
`-not-corresp-author-phone-empty`		Only keep *publications* with a *corresponding author* t
`-corresp-author-uri`	<*regex*>	Only keep *publications* with a *corresponding author* v
`-not-corresp-author-uri`	<*regex*>	Only keep *publications* with no *corresponding author*
`-corresp-author-uri-empty`		Only keep *publications* whose *corresponding author*s
`-not-corresp-author-uri-empty`		Only keep *publications* with a *corresponding author* v
`-corresp-author-size`	<*positive integer*> . . .	Only keep *publications* whose *corresponding author*s
`-not-corresp-author-size`	<*positive integer*> . . .	Only keep *publications* whose *corresponding author*s
`-corresp-author-size-more`	<*positive integer*>	Only keep *publications* whose *corresponding author*s
`-corresp-author-size-less`	<*positive integer*>	Only keep *publications* whose *corresponding author*s
`-visited`	<*regex*>	Only keep *publications* with a *visited site* whose URL
`-not-visited`	<*regex*>	Only keep *publications* with no *visited site*s whose UR
`-visited-host`	<*string*> <*string*> . . .	Only keep *publications* with a *visited site* whose URL
`-not-visited-host`	<*string*> <*string*> . . .	Only keep *publications* with no *visited site*s whose UR
`-visited-type`	<*PublicationPartType*> . . .	Only keep *publications* with a *visited site* of type equa
`-not-visited-type`	<*PublicationPartType*> . . .	Only keep *publications* with no *visited site*s of type eq
`-visited-type-more`	<*PublicationPartType*>	Only keep *publications* with a *visited site* of better typ
`-visited-type-less`	<*PublicationPartType*>	Only keep *publications* with a *visited site* of lesser typ
`-visited-type-final`		Only keep *publications* with a *visited site* of final type
`-not-visited-type-final`		Only keep *publications* with no *visited site*s of final ty
`-visited-type-pdf`		Only keep *publications* with a *visited site* of PDF type
`-not-visited-type-pdf`		Only keep *publications* with no *visited site*s of PDF ty
`-visited-from`	<*regex*>	Only keep *publications* with a *visited site* whose prove
`-not-visited-from`	<*regex*>	Only keep *publications* with no *visited site*s whose pro
`-visited-from-host`	<*string*> <*string*> . . .	Only keep *publications* with a *visited site* whose prove
`-not-visited-from-host`	<*string*> <*string*> . . .	Only keep *publications* with no *visited site*s whose pro
`-visited-time-more`	<*ISO-8601 time*>	Only keep *publications* with a *visited site* whose visit t
`-visited-time-less`	<*ISO-8601 time*>	Only keep *publications* with a *visited site* whose visit t
`-visited-size`	<*positive integer*> . . .	Only keep *publications* whose *visited site*s size is equa
`-not-visited-size`	<*positive integer*> . . .	Only keep *publications* whose *visited site*s size is not e
`-visited-size-more`	<*positive integer*>	Only keep *publications* whose *visited site*s size is more
`-visited-size-less`	<*positive integer*>	Only keep *publications* whose *visited site*s size is less

2.4.11 Filter publication parts

Conditions that *publication part*s must meet for the publication to be retained in the *pipeline*.

Each parameter (except `-part-empty`, `-not-part-empty`, `-part-usable`, `-not-part-usable`, `-part-final`, `-not-part-final`) has a corresponding parameter specifying the publication parts that need to meet the condition given by the parameter. For example, `-part-content` gives a regular expression and `-part-content-part` lists all publication parts that must have a match with the given regular expression. If `-part-content` is specified, then `-part-content-part` must also be specified (and vice versa).

A publication part is any of: *the pmid*, *the pmcid*, *the doi*, *title*, *keywords*, *MeSH*, *EFO*, *GO*, *theAbstract*, *fulltext*.

Parameter	Parameter args	Description
`-part-empty`	<*PublicationPartName*> …	Only keep *publications* with specified parts being *empty*
`-not-part-empty`	<*PublicationPartName*> …	Only keep *publications* with specified parts not being *empty*
`-part-usable`	<*PublicationPartName*> …	Only keep *publications* with specified parts being *usable*
`-not-part-usable`	<*PublicationPartName*> …	Only keep *publications* with specified parts not being *usable*
`-part-final`	<*PublicationPartName*> …	Only keep *publications* with specified parts being *final*
`-not-part-final`	<*PublicationPartName*> …	Only keep *publications* with specified parts not being *final*
`-part-content`	<*regex*>	Only keep *publications* where the *contents* of all parts specified with `-part-content-part` have a match with the given regular expression
`-not-part-content`	<*regex*>	Only keep *publications* where the *contents* of all parts specified with `-not-part-content-part` do not have a match with the given regular expression
`-part-size`	<*positive integer*> …	Only keep *publications* where the *sizes* of all parts specified with `-part-size-part` are equal to any of given sizes
`-not-part-size`	<*positive integer*> …	Only keep *publications* where the *sizes* of all parts specified with `-not-part-size-part` are not equal to any of given sizes
`-part-size-more`	<*positive integer*>	Only keep *publications* where the *sizes* of all parts specified with `-part-size-more-part` are more than the given size
`-part-size-less`	<*positive integer*>	Only keep *publications* where the *sizes* of all parts specified with `-part-size-less-part` are less than the given size
`-part-type`	<*PublicationPartType*> …	Only keep *publications* where the *types* of all parts specified with `-part-type-part` are equal to any of given types
`-not-part-type`	<*PublicationPartType*> …	Only keep *publications* where the *types* of all parts specified with `-not-part-type-part` are not equal to any of given types
`-part-type-more`	<*PublicationPartType*>	Only keep *publications* where the *types* of all parts specified with `-part-type-more-type` are better than the given type
`-part-type-less`	<*PublicationPartType*>	Only keep *publications* where the *types* of all parts specified with `-part-type-less-type` are lesser than the given type
`-part-type-final`	<*PublicationPartType*>	Only keep *publications* where the *types* of all parts specified with `-part-type-final` are of final type
`-not-part-type-final`	<*Publica-*	Only keep *publications* where the *types* of all parts specified with

2.4.12 Filter webpages and docs

Conditions that *webpages* and *docs* must meet to be retained in the *pipeline*.

Parameter	Parameter args	Description
`-broken`		Only keep *webpages* and *docs* that are *broken*
`-not-broken`		Only keep *webpages* and *docs* that are not *broken*
`-start-url`	<*regex*>	Only keep *webpages* and *docs* whose *start URL* has a match with the
`-not-start-url`	<*regex*>	Only keep *webpages* and *docs* whose *start URL* does not have a mat
`-start-url-host`	<*string*> <*string*> ...	Only keep *webpages* and *docs* whose *start URL* host part is present i
`-not-start-url-host`	<*string*> <*string*> ...	Only keep *webpages* and *docs* whose *start URL* host part is not prese
`-final-url`	<*regex*>	Only keep *webpages* and *docs* whose *final URL* has a match with the
`-not-final-url`	<*regex*>	Only keep *webpages* and *docs* whose *final URL* does not have a mat
`-final-url-host`	<*string*> <*string*> ...	Only keep *webpages* and *docs* whose *final URL* host part is present i
`-not-final-url-host`	<*string*> <*string*> ...	Only keep *webpages* and *docs* whose *final URL* host part is not prese
`-final-url-empty`		Only keep *webpages* and *docs* whose *final URL* is empty
`-not-final-url-empty`		Only keep *webpages* and *docs* whose *final URL* is not empty
`-content-type`	<*regex*>	Only keep *webpages* and *docs* whose *HTTP Content-Type* has a mat
`-not-content-type`	<*regex*>	Only keep *webpages* and *docs* whose *HTTP Content-Type* does not h
`-content-type-empty`		Only keep *webpages* and *docs* whose *HTTP Content-Type* is empty
`-not-content-type-empty`		Only keep *webpages* and *docs* whose *HTTP Content-Type* is not emp
`-status-code`	<*integer*> <*integer*> ...	Only keep *webpages* and *docs* whose *HTTP status code* is equal to o
`-not-status-code`	<*integer*> <*integer*> ...	Only keep *webpages* and *docs* whose *HTTP status code* is not equal
`-status-code-more`	<*integer*>	Only keep *webpages* and *docs* whose *HTTP status code* is bigger tha
`-status-code-less`	<*integer*>	Only keep *webpages* and *docs* whose *HTTP status code* is smaller th
`-title`	<*regex*>	Only keep *webpages* and *docs* whose *page title* has a match with the
`-not-title`	<*regex*>	Only keep *webpages* and *docs* whose *page title* does not have a matc
`-title-size`	<*positive integer*> ...	Only keep *webpages* and *docs* whose *title length* is equal to one of g
`-not-title-size`	<*positive integer*> ...	Only keep *webpages* and *docs* whose *title length* is not equal to any
`-title-size-more`	<*positive integer*>	Only keep *webpages* and *docs* whose *title length* is more than the giv
`-title-size-less`	<*positive integer*>	Only keep *webpages* and *docs* whose *title length* is less than the give
`-content`	<*regex*>	Only keep *webpages* and *docs* whose *content* has a match with the g
`-not-content`	<*regex*>	Only keep *webpages* and *docs* whose *content* does not have a match
`-content-size`	<*positive integer*> ...	Only keep *webpages* and *docs* whose *content length* is equal to one
`-not-content-size`	<*positive integer*> ...	Only keep *webpages* and *docs* whose *content length* is not equal to a
`-content-size-more`	<*positive integer*>	Only keep *webpages* and *docs* whose *content length* is more than the
`-content-size-less`	<*positive integer*>	Only keep *webpages* and *docs* whose *content length* is less than the
`-content-time-more`	<*ISO-8601 time*>	Only keep *webpages* and *docs* whose *content time* is more than or eq
`-content-time-less`	<*ISO-8601 time*>	Only keep *webpages* and *docs* whose *content time* is less than or equ
`-license`	<*regex*>	Only keep *webpages* and *docs* whose *software license* has a match w
`-not-license`	<*regex*>	Only keep *webpages* and *docs* whose *software license* does not have
`-license-empty`		Only keep *webpages* and *docs* whose *software license* is empty
`-not-license-empty`		Only keep *webpages* and *docs* whose *software license* is not empty
`-language`	<*regex*>	Only keep *webpages* and *docs* whose *programming language* has a
`-not-language`	<*regex*>	Only keep *webpages* and *docs* whose *programming language* does n
`-language-empty`		Only keep *webpages* and *docs* whose *programming language* is emp
`-not-language-empty`		Only keep *webpages* and *docs* whose *programming language* is not
`-has-scrape`		Only keep *webpages* and *docs* that have *scraping rules* (based on *fin*
`-not-has-scrape`		Only keep *webpages* and *docs* that do not have *scraping rules* (based

2.4.13 Sort content

Sorting of fetched/loaded and filtered content. If sorted by their ID, then *publications* are first sorted by *the PMID*, then by *the PMCID* (if PMID is absent), then by *the DOI* (if PMID and PMCID are absent). Internally, the PMID, the PMCID and the DOI registrant are sorted numerically, DOIs within the same registrant alphabetically. If sorted by their URL, then *webpages* and *docs* are sorted alphabetically according to their *startUrl*.

Parameter	Parameter args	Description
`-asc`		Sort *publications*, *webpages* and *docs* by their ID/URL in ascending order
`-desc`		Sort *publications*, *webpages* and *docs* by their ID/URL in descending order
`-asc-time`		Sort *publications*, *webpages* and *docs* by their *fetchTime* in ascending order
`-desc-time`		Sort *publications*, *webpages* and *docs* by their *fetchTime* in descending order

2.4.14 Limit content

Fetched/loaded, filtered and sorted content can be limited to a given number of entries either in the front or back. The list of *top hosts* will also be limited.

Parameter	Parameter args	Description
`-head`	*<positive integer>*	Only keep the first given number of *publications*, *webpages* and *docs* (same for *top hosts* from *publications*, *webpages* and *docs*)
`-tail`	*<positive integer>*	Only keep the last given number of *publications*, *webpages* and *docs* (same for *top hosts* from *publications*, *webpages* and *docs*)

2.4.15 Update citations count

Parameter	Parameter args	Description
`-update-citat`	*<database file>*	Fetch and update the *citations count* and *citations count last update timestamp* of all *publications* resulting from the *pipeline* and put successfully updated *publications* to the given *database*

2.4.16 Put to database

Parameter	Parameter args	Description
`-put`	*<database file>*	Put all *publications*, *webpages* and *docs* resulting from the *pipeline* to the given *database*, overwriting any existing entries that have equal IDs/URLs

2.4.17 Remove from database

Parameter	Parameter args	Description
`-remove`	*<database file>*	From the given *database*, remove all *publications*, *webpages* and *docs* with IDs corresponding to IDs of *publications*, *webpages* and *docs* resulting from the *pipeline*

2.4.18 Output

Output final list of *publications* (or *publication part*s specified by `--out-part`), *webpages* and *docs* resulting from the *pipeline* to stdout or the specified text files in the format specified by the *Output modifiers* `--plain` and `--format`.

If `--format text` (the default) and `--plain` are specified and `--out-part` specifies only *publication IDs*, then publications will be output in the form `<pmid>\t<pmcid>\t<doi>`, one per line. Also in case of `--format text --plain`, if `--out-part` specifies only one publication part (that is not *theAbstract* or *fulltext*), then for each publication there will be only one line in the output, containing the plain text output of that publication part. Otherwise, there will be separator lines separating different publications in the output.

If `--format html` and `--plain` are specified and `--out-part` specifies only publication IDs, then the output will be a HTML table of publication IDs, with one row corresponding to one publication.

The full output format of `--format json` is specified later in *JSON format*. There is also a short description about the *HTML and plain text outputs*.

Additionally, there are operations to get the so-called top hosts: all host parts of URLs of *visited site*s of publications, of URLs of webpages and of URLs of docs, starting from the most common and including count numbers. This can be useful for example for finding hosts to write *scraping rules* for. When counting different hosts, comparison of hosts is done case-insensitively and "www." is removed. Parameter `-has-scrape` can be added to only output hosts for which scraping rules could be found and parameter `-not-has-scrape` added to only output hosts for which no scraping rules could be found. Parameters `-head` and `-tail` can be used to limit the size of top hosts output.

For analysing the different sources of publication part content, there is an option to print a *PublicationPartType* vs *PublicationPartName* table in CSV format.

Parameter	Parameter args	Description
`-out`		Output *publications* (or *publication part*s specified by `--out-part`), *webpages* and *docs* to stdout in the format specified by the *Output modifiers* `--plain` and `--format`
`-txt-pub`	*<file>*	Output *publications* (or *publication part*s specified by `--out-part`) to the given file in the format specified by the *Output modifiers* `--plain` and `--format`
`-txt-web`	*<file>*	Output *webpages* to the given file in the format specified by the *Output modifiers* `--plain` and `--format`
`-txt-doc`	*<file>*	Output *docs* to the given file in the format specified by the *Output modifiers* `--plain` and `--format`
`-count`		Output count numbers for *publications*, *webpages* and *docs* to stdout
`-out-top-hosts`		Output all host parts of URLs of visited sites of *publications*, of URLs of *webpages* and of URLs of *docs* to stdout, starting from most common and including count number
`-txt-top-hosts-pub`	*<file>*	Output all host parts of URLs of *visited site*s of *publications* to the given file, starting from the most common and including count numbers
`-txt-top-hosts-web`	*<file>*	Output all host parts of URLs of *webpages* to the given file, starting from the most common and including count numbers
`-txt-top-hosts-doc`	*<file>*	Output all host parts of URLs of *docs* to the given file, starting from the most common and including count numbers
`-count-top-hosts`		Output number of different host parts of URLs of *visited site*s of *publications*, of URLs of *webpages* and of URLs of *docs* to stdout
`-part-table`		Output a *PublicationPartType* vs *PublicationPartName* table in CSV format to stdout, i.e. how many *publications* have content for the given publication part fetched from the given resource type

2.4.19 Output modifiers

Some parameters to influence the behaviour of outputting operations.

Parameter	Parameter args	Default	Description
`--plain`			If specified, then any potential metadata will be omitted from the output
`--format`	*<Format>*	`text`	Can choose between plain text output format (`text`), HTML format (`html`) and *JSON format* (`json`)
`--out-part`	*<PublicationPartName>* ...		If specified, then only the specified publication parts will be output (*webpages* and *docs* are not affected). Independent from the `--fetch-part` parameter.

2.5 Test

Operations for testing built-in and configurable scraping rules (e.g., `-print-europepmc-xml` and `-test-europepmc-xml`; `-print-site` and `-test-site`) are described in the *scraping rules* section.

2.6 Examples

2.6.1 Operations with IDs

As a first step in the *pipeline of operations*, some *publication IDs*, *webpage URLs* or *doc URLs* must be loaded (and possibly filtered). How to create and populate the *database* files used in this section is explained in the next section.

```
$ java -jar pubfetcher-cli-<version>.jar \
-pub 12345678 10.1093/nar/gkw199 -pub-file pub1.txt pub2.txt \
-pub-db database.db new.db \
-has-pmcid -doi '(?i)nmeth' \
-doi-url '^https://www.ebi.ac.uk/europepmc/' -doi-registrant 1038 \
-out-ids --plain
```

First, add two *publication IDs* from the command-line: a publication ID where the *PMID* is `12345678` and a publication ID where the *DOI* is `10.1093/nar/gkw199`. Then add publication IDs from the text files `pub1.txt` and `pub2.txt`, where each line must be in the form `<pmid>\t<pmcid>\t<doi>` (except empty lines and lines beginning with `#` which are ignored). As last, add all publication IDs found in the *database* files `database.db` and `new.db`. The resulting list of publication IDs is actually a set, meaning duplicate IDs will be merged.

Then, the publication IDs will be filtered. Parameter `-has-pmcid` means that only publication IDs that have a non-*empty PMCID* (probably meaning that the fulltext is available in *PubMed Central*) will be kept. Specifying `-doi '(?i)nmeth'` means that, in addition, the DOI part of the ID must have a match with "nmeth" (Nature Methods) case-insensitively (we specify case-insensitivity with "(?i)" because we are converting the DOIs to upper-case). With `-doi-url` we specify that the DOI was found first from the *Europe PMC* API and with `-doi-registrant` we specify that the DOI registrant code must be `1038` (Nature).

The resultant list of filtered publication IDs will be output to standard output as plain text with the parameter `-out-ids`. Specifying the modifier `--plain` means that the ID provenance URLs will not be output and the output of IDs will be in the form `<pmid>\t<pmcid>\t<doi>`.

```
$ java -jar pubfetcher-cli-<version>.jar \
-pub-db new.db -web-db new.db -not-in-db database.db \
-url '^https' -not-url-host bioconductor.org github.com \
-txt-ids-pub pub.json -txt-ids-web web.json --format json
```

First, add all *publication IDs* and all *webpage URLs* from the *database* file `new.db`. With `-not-in-db` we remove all publication IDs and webpage URLs that are already present in the database file `database.db`. With the regex `^https` specified using the `-url` parameter only webpage URLs whose schema is HTTPS are kept. And with `-not-url-host` we remove all webpage URLs whose host part is `bioconductor.org` or `github.com` (or "www.bioconductor.org" or "www.github.com") case-insensitively. The resultant list of publication IDs will be output to the file `pub.json` and the resultant list of webpage URLs will be output to the file `web.json`. The output will be in *JSON format* because it was specified using the `--format` modifier. By using `--format html` or `--format html --plain` we would get a HTML file instead, which when opened in a web browser would list the IDs and URLs as clickable links.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-has-pmid -asc-ids -head-ids 10 -txt-ids-pub oldpmid.txt --plain
```

Add all *publication IDs* from the *database* file `database.db`, only keep publication IDs that have a non-*empty* *PMID* part, order the publication IDs (smallest PMID first) and only keep the `10` first IDs. The resultant 10 publication IDs will be output to the file `oldpmid.txt`, where each line is in the form `<pmid>\t<pmcid>\t<doi>`.

```
$ java -jar pubfetcher-cli-<version>.jar \
-pub-file oldpmid.txt -pub 12345678 -remove-ids database.db
```

publications that have a small *PMID* are in the *database* possibly by mistake. So we can review the file `oldpmid.txt` generated in the previous step and keep entries we want to remove from the database listed in that file. Then, with the last command, we add *publication IDs* from the file `oldpmid.txt`, manually add an extra publication ID with PMID `12345678` from the command-line and with `-remove-ids` remove all publications corresponding to the resultant list of publication IDs from the database file `database.db`.

2.6.2 Get content

Next, we'll see how content can be fetched/loaded and how *database* files (such as those used in the previous section) can be populated with content.

```
$ java -jar pubfetcher-cli-<version>.jar -db-init database.db
```

This creates a new empty *database* file called `database.db`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-fetch --timeout 30000 -usable -put database.db
```

Add all *publication IDs* from the file `pub.txt` (where each line is in the form `<pmid>\t<pmcid>\t<doi>`) and for each ID put together a *publication* with content fetched from different *resources*, thus getting a list of *publications*. The connect and read timeout is changed from the default value of `15` seconds to `30` seconds with the general *Fetching* parameter *timeout*. Filter out non-*usable publication*s from the list with parameter `-usable` and put all publications from the resultant list to the database file `database.db`. Any existing publication with an ID equal to an ID of a new publication will be overwritten.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-fetch-put database.db --timeout 30000

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -not-usable -remove database.db
```

If parameters `-fetch` and `-put` are used, then first all *publications* are fetched and loaded into memory, and only then all publications are saved to the *database* file at once. This is not optimal if there are a lot of publications to fetch, as if some severe error occurs, all content will be lost. Using the parameter `-fetch-put`, each publication will be put to the database right after it has been fetched. This has the downside of not being able to filter publications before they are put to the database. One way around this is to put all content to the database while fetching and then remove some of the entries from the database based on required filters, as illustrated by the second command.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-db-fetch database.db --threads 16 -usable -count

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -count
```

With parameter `-db-fetch` the following happens for each *publication*: first the publication is looked for in the *database*; if found, it will be updated with fetched content, if possible and required, and saved back to the database file; if not found, a new publication will be put together with fetched content and put to the database file. This potentially enables less fetching in the future and enables progressive betterment of some *publications* over time. Additionally, in contrast to `-fetch` and `-fetch-put`, operation `-db-fetch` is multithreaded (with the number of threads specified using `--threads`), thus much quicker.

Like with `-fetch-put`, publications can't be filtered before they are put to the database. Any specified filter parameters will only have an effect on which content is retained in memory for further processing (like outputting) down the *pipeline*. For example, with `-usable -count`, the number of *usable publication*s is output to stdout after fetching is done, but both usable and non-usable publications were saved to the database file, as can be seen with the `-count` of the seconds command.

```
$ java -jar pubfetcher-cli-<version>.jar -db-init new.db

$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-db-fetch new.db --threads 16 -usable -count

$ java -jar pubfetcher-cli-<version>.jar -pub-db new.db \
-db new.db -not-usable -remove new.db

$ java -jar pubfetcher-cli-<version>.jar -pub-db new.db \
-db new.db -put database.db
```

Sometimes, we may want only "fresh" entries (fetched only once and not updated), like `-fetch` and `-fetch-put` provide, but with multithreading support, like `-db-fetch` provides, and with filtering support, like `-fetch` provides. Then, the above sequence of commands can be used: make a new *database* file called `new.db`; fetch entries to `new.db` using `16` threads; filter out non-usable entries from `new.db`; and put content from `new.db` to our main database file, overwriting any existing entries there.

Another similar option would be to disable updating of entries by setting the *retryLimit* to `0` and *emptyCooldown*, *nonFinalCooldown*, *fetchExceptionCooldown* to a negative number.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-db-fetch-end database.db --threads 16

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -usable -count
```

Parameter `-db-fetch` will, in addition to saving entries to the *database* file, load all entries into memory while fetching for further processing (like outputting) down the *pipeline*. This might cause excessive memory usage if a lot of entries are fetched. Thus, parameter `-db-fetch-end` is provided, which is like `-db-fetch` except it does not retain any of the entries in memory. Any further filtering, outputting, etc can be done on the database file after fetching with `-db-fetch-end` is done, as shown with the provided second command.

```
$ java -jar pubfetcher-cli-<version>.jar \
-pub-file pub.txt -web-file web.txt -doc-file doc.txt \
-db-fetch-end database.db --threads 16 --log database.log
```

An example of a comprehensive and quick fetching command: add all provided *publication IDs*, *webpage URLs* and *doc URLs*, fetch all corresponding *publications*, *webpages* and *docs*, using `16` threads for this process and saving the content to the *database* file `database.db`, and append all log messages to the file `database.log` for possible future reference and analysis.

2.6.3 Loading content

After content has been fetched, e.g. using one of commands in the previous section, it can be loaded and explored.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db --pre-filter -oa -journal-title 'Nature' \
-not-part-empty fulltext -out | less
```

From the *database* file `database.db`, load all *publications* that are Open Access, that are from a journal whose title has a match with the regular expression `Nature` and whose *fulltext* part is not *empty*, and output these publications with metadata and in plain text to stdout, from where output is piped to the pager `less`. Specifying `--pre-filter` means that content is filtered while being loaded from the database, meaning that entries not passing the filter will not be retained in memory. If `--pre-filter` would not be specified, then first all entries corresponding to the added *publication IDs* would be loaded to memory at once and only then would the entries start to be removed with the specified filters. This has the advantage of being able to see in log messages how many entries pass each filter, however, if the number of added and filtered publication IDs is very big, it could be better to use `--pre-filter` to not cause excessive memory usage.

2.6.4 Limit fetching/loading

For testing or memory reduction purposes the number of fetched/loaded entries can be limited with `--limit`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-fetch --limit 3 -out | less
```

Only fetch and output the first `3` *publications* listed in `pub.txt`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-fetch --limit 3 --pre-filter -oa -out | less
```

Only fetch and output the first `3` Open Access *publications* listed in `pub.txt`. Using `--pre-filter` means that filtering is done before limiting the entries, meaning that more than 3 entries might be fetched, because fetching happens until a third Open Access publication is encountered, but exactly 3 entries are output (if there are enough publications listed in `pub.txt`). If `--pre-filter` was not used, then exactly 3 entries would be fetched (if there are enough publications listed in `pub.txt`), meaning that less than 3 entries might be output, because not all of the publications might be Open Access.

2.6.5 Fetch only some publication parts

If we are only interested in some *publication part*s, it might be advantageous to list them explicitly. This might make fetching faster, because we can skip Internet *resources* that can't provide us with any missing parts we are interested in or we can stop fetching of new resources altogether if all parts we are interested in are *final*.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-fetch --fetch-part title theAbstract -out | less
```

Only fetch the *title* and *theAbstract* for the added publication IDs, all other *publication part*s (except IDs) will be *empty* in the output.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-fetch --fetch-part title theAbstract \
-out --out-part title theAbstract --plain | less
```

If only *title* and *theAbstract* are fetched, then all other *publication part*s (except IDs) will be *empty*, thus we might not want to output these empty parts. This can be done be specifying the `title` and `theAbstract` parts with `--out-part`. Additionally specifying `--plain` means no metadata is output either, thus the output will consist of only plain text *publication* titles and abstracts with separating characters between different publications.

2.6.6 Converting IDs

As a special case of the ability to only fetch some *publication part*s, PubFetcher can be used as an ID converter between *PMID*/*PMCID*/*DOI*.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-fetch --fetch-part pmid pmcid doi --out-part pmid pmcid doi \
-txt-pub newpub.txt --plain
```

Take all *publication IDs* from `pub.txt` (where each line is in the form `<pmid>\t<pmcid>\t<doi>`) and for each ID fetch only *publication part*s *the PMID*, *the PMCID* and *the DOI* and output only these parts to the file `newpub.txt`. In the output file each line will be in the form `<pmid>\t<pmcid>\t<doi>`, because ID provenance URLs are excluded with `--plain` and no other publication parts are output. If the goal is to convert only DOI to PMID and PMCID, for example, then each line in `pub.txt` could be in the form `\t\t<doi>` and parameters specified as `--fetch-part pmid pmcid --out-part pmid pmcid`.

```
$ java -jar pubfetcher-cli-<version>.jar -db-init newpub.db

$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-db-fetch-end newpub.db --threads 16 --fetch-part pmid pmcid doi

$ java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt \
-db newpub.db --out-part pmid pmcid doi -txt-pub newpub.txt --plain
```

If a lot of *publication IDs* are to be converted, it would be better to first fetch all *publications* to a resumable temporary *database* file, using the multithreaded `-db-fetch-end`, and only then output the parts *the PMID*, *the PMCID* and *the DOI* to the file `newpub.txt`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db newpub.db \
-db newpub.db -part-table
```

We can output a *PublicationPartType* vs *PublicationPartName* table in CSV format to see from which *resources* the converted IDs were got from. Most likely the large majority will be from *Europe PMC* (e.g., https://www.ebi.ac.uk/europepmc/webservices/rest/search?resulttype=core&format=xml&query=ext_id:17478515%20src:med). *DOI*s with types other than the "europepmc", "pubmed" or "pmc" types were not converted to DOI by the corresponding resource but just confirmed by it (as fetching that resource required the knowledge of a DOI in the first place). Type "external" means that the supplied ID was not found and confirmed in any resource.

In one instance of around 10000 *publications*, the usefulness of PubFetcher for only ID conversion manifested itself mostly in the case of finding *PMCID*s. But even then, around 97% of PMCIDs were present in *Europe PMC*. As to the rest, around 2% were of type "link_oadoi" (i.e., found using *Unpaywall*) and around 1% were of type "pubmed_xml" (i.e., present in *PubMed*, but not Europe PMC, although it was mostly articles which had been assigned a PMCID but were actually not yet available due to delayed release (embargo)). In the case of *PMID*s the usefulness is even less and mostly in finding a corresponding PMID (if missing) to the PMCID found using a source other than Europe PMC. And in the case of DOIs, only a couple (out of 10000) were found from *resources* other than Europe PMC (mostly because initially only a PMCID was supplied and that PMCID was not present in Europe PMC).

So in conclusion, PubFetcher gives an advantage of a few percent over simply using an XML returned by the Europe PMC API when finding PMCIDs for articles (but also when converting from DOI to PMID), but gives almost no advantage when converting from PMID to DOI.

2.6.7 Filtering content

The are many possible filters, all of which are defined above in the section *Filter content*.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db -web-db \
database.db -doc-db database.db -db database.db -usable -grep 'DNA' \
-oa -pub-date-more 2018-08-15T00:00:00Z -citations-count-more 9 \
-corresp-author-size 1 2 -part-size-more 2 -part-size-more-part \
keywords mesh -part-type europepmc_xml pmc_xml doi -part-type-part \
fulltext -part-time-more 2018-08-15T12:00:00Z -part-time-more-part \
fulltext -title '(?i)software|database' -status-code-more 199 \
-status-code-less 300 -not-license-empty -has-scrape -asc -out | less
```

This example will load all content (*publications*, *webpages* and *docs*) from the *database* file `database.db` and apply the following filters (ANDed together) to remove content before it is sorted in ascending order and output:

Parameter	Description
`-usable`	Only *usable publication*s, *usable webpage*s and usable docs will be kept
`-grep 'DNA'`	Only *publications*, *webpages* and *docs* whose whole content (excluding metadata) has a match with the regular expression "DNA" (i.e., contains the string "DNA")
`-oa`	Only keep *publications* that are Open Access
`-pub-date-more 2018-08-15T00:00:00Z`	Only keep *publications* whose *publication date* is `2018-08-15` or later
`-citations-count-more 9`	Only keep *publications* that are cited more than `9` times
`-corresp-author-size 1 2`	Only keep *publications* for whose `1` or `2` *corresponding author*s were found (i.e., publications with no found corresponding authors or more that 2 corresponding authors are discarded)
`-part-size-more 2 -part-size-more-part keywords mesh`	Only keep *publications* that have more than `2` *keywords* and more than `2` *MeSH terms*
`-part-type europepmc_xml pmc_xml doi -part-type-part fulltext`	Only keep *publications* whose *fulltext* part is of type "europepmc_xml", "pmc_xml" or "doi"
`-part-time-more 2018-08-15T12:00:00Z -part-time-more-part fulltext`	Only keep *publications* whose *fulltext* part has been obtained at `2018-08-15 noon (UTC)` or later
`-title '(?i)software|database'`	Only keep *webpages* and *docs* whose *page title* has a match with the regular expression `(?i)software|database` (i.e., contains case-insensitively "software" or "database")
`-status-code-more 199 -status-code-less 300`	Only keep *webpages* and *docs* whose *status code* is `2xx`
`-not-license-empty`	Only keep *webpages* and *docs* that have a non-empty *software license* name present
`-has-scrape`	Only keep *webpages* and *docs* for which *scraping rules* are present

2.6.8 Terminal operations

Operations that are done on the final list of entries. If multiple such operations are specified in one command, then they will be performed in the order they are defined in this reference.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -oa -update-citations-count database.db
```

Load all *publications* from the *database* file `database.db`, update the *citations count* of all Open Access publications and save successfully updated publications back to the database file `database.db`.

```
$ java -jar pubfetcher-cli-<version>.jar -db-init oapub.db

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -oa -put oapub.db
```

Copy all Open Access *publications* from the *database* file `database.db` to the new database file `oapub.db`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db new.db -db new.db \
-put database.db
```

Copy all *publications* from the *database* file `new.db` to the database file `database.db`, overwriting any existing entries in `database.db`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -not-oa -remove database.db
```

Remove all not Open Access *publications* from the *database* file `database.db`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db other.db \
-remove-ids database.db
```

Remove all *publications* that are also present in the *database* file `other.db` from the database file `database.db`. As removal is done based on all IDs found in `other.db` and no filtering based on the content of entries needs to be done, then loading of content from the database file `other.db` is not done and `-remove-ids` must be used instead of `-remove` for removal from the database file `database.db`.

2.6.9 Output

Output can happen to stdout or text files in plain text, HTML or JSON, with or without metadata.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -out | less
```

Output all *publications* from the *database* file `database.db` to stdout in plain text and with metadata and pipe stdout to the pager `less`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-web-db database.db -db database.db \
-txt-pub pub.html -txt-web web.html --format html
```

Output all *publications* and *webpages* from the *database* file `database.db` in HTML format and with metadata to the files `pub.html` and `web.html` respectively.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db \
-txt-pub pubids.html --out-part pmid pmcid doi --format html --plain

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-txt-ids-pub pubids.html --format html --plain
```

Both commands will output all *publication IDs* from the *database* file `database.db` as an HTML table to the file `pubids.html`.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -out --out-part mesh --format text --plain
```

Output the *MeSH terms* of all *publications* from the *database* file `database.db` to stdout in plain text and without metadata. As only one *publication part* (that is not *theAbstract* or *fulltext*) is output without metadata, then there will be one line of output (a list of MeSH terms) for each publication.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-web-db database.db -db database.db \
-out-top-hosts -head 10 -not-has-scrape
```

From the *database* file `database.db`, output host parts of URLs of *visited sites* of *publications* and of URLs of *webpages* for which no *scraping rules* could be found, starting from the most common and including count numbers and limiting output to the `10` first hosts for both cases. This could be useful for finding hosts to add scraping rules for.

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -part-table > part-table.csv
```

From all *publications* in the *database* file `database.db`, generate a *PublicationPartType* vs *PublicationPartName* table in CSV format and output it to the file `part-table.csv`.

2.6.9.1 Export to JSON

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-web-db database.db -doc-db database.db -db database.db \
-txt-pub pub.json -txt-web web.json -txt-doc doc.json --format json
```

Output all *publications*, *webpages* and *docs* from the *database* file `database.db` in *JSON format* and with metadata to the files `pub.json`, `web.json` and `doc.json` respectively. That is, export all content in JSON, so that the database file and PubFetcher itself would not be needed again for further work with the data.

2.7 Notes

The syntax of regular expressions is as defined in Java, see documentation of the Pattern class: https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

The ISO-8601 times must be specified like “2018-08-31T13:37:51Z” or “2018-08-31T13:37:51.123Z”.

All *publication DOIs* are normalised, this effect can be tested with the `-normalise-doi` method.

webpages and *docs* have the same structure, equivalent methods and common *scraping rules*, they just provide separate stores for saving general web pages and documentation web pages respectively.

If an entry is final (and without a fetching exception) in a *database*, then it can never be refetched again (only the *citations count* can be updated). If that entry needs to be refreshed for some reason, then `-fetch` or `-fetch-put` must be used to fetch a completely new entry and overwrite the old one in the database.

On the other hand, `-db-fetch` or `-db-fetch-end` could be used multiple times after some interval to try to complete non-final entries, e.g. web servers that were offline might be up again, some resources have been updated with extra content or we have updated some scraping rules. For example, the command `java -jar pubfetcher-cli-<version>.jar -pub-file pub.txt -db-fetch-end database.db` could be run a week after the same command was initially run.

2.7.1 Limitations

The querying capabilities of PubFetcher are rather rudimentary (unlike SQL), but hopefully enough for most use cases.

For example, different filters are ANDed together and there is no support for OR. As a workaround, different conditions can be output to temporary files of IDs/URLs that can then be put together. For example, output all *publications* from the *database* file `database.db` that are cited more than `9` times or that have been published on `2018-01-01` or later to stdout:

```
$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -citations-count-more 9 \
-txt-pub pub_citations.txt --out-part pmid pmcid doi --plain

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -pub-date-more 2018-01-01T00:00:00Z \
-txt-pub pub_pubdate.txt --out-part pmid pmcid doi --plain

$ java -jar pubfetcher-cli-<version>.jar -pub-file pub_citations.txt \
pub_pubdate.txt -db database.db -out | less
```

Some advanced filtering might not be possible, because some command-line switches can't be specified twice. For example, the filter `-part-size-more 2 -part-size-more-part keywords -part-size-more 999 -part-size-more-part theAbstract` will not filter out entries that have more than `2` *keywords* and whose *theAbstract* length is more than `999`, but instead result in an error. As a workaround, the filter might be broken down and the result of the different conditions saved in temporary database files that can then be ANDed together:

```
$ java -jar pubfetcher-cli-<version>.jar -db-init pub_keywords.db

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -part-size-more 2 -part-size-more-part keywords \
-put pub_keywords.db

$ java -jar pubfetcher-cli-<version>.jar -db-init pub_abstract.db

$ java -jar pubfetcher-cli-<version>.jar -pub-db database.db \
-db database.db -part-size-more 999 -part-size-more-part theAbstract \
-put pub_abstract.db

$ java -jar pubfetcher-cli-<version>.jar -pub-db pub_keywords.db \
-in-db pub_abstract.db -db database.db -out | less
```

In the *pipeline* the operations are done in the order they are defined in this reference and with one command the pipeline is run only once. Which means, for example, that it is not possible to filter some content and then refetch the filtered entries using only one command, because content loading/fetching happens before content filtering. In such cases, intermediate results can be saved to temporary files, which can be used by the next command to get the desired outcome. For example, get all *publications* from the *database* file `database.db` that have a *visited site* whose URL has a match with the regular expression `academic\.oup\.com|[a-zA-Z0-9.-]*sciencemag\.org` and refetch those publications from scratch, overwriting the corresponding old publications in `database.db`:

```
$ java -jar pubfetcher-cli-<version>.jar \
-pub-db database.db -db database.db \
-visited 'academic\.oup\.com|[a-zA-Z0-9.-]*sciencemag\.org' \
-txt-pub oup_science.txt --out-part pmid pmcid doi --plain

$ java -jar pubfetcher-cli-<version>.jar -pub-file oup_science.txt \
-fetch-put database.db
```

CHAPTER 3

Output

3.1 Database

The database file that is used by PubFetcher to save *publications*, *webpages* and *docs* on disk is a simple key-value store generated by the MapDB library.

In case of the webpages and docs stores, a key is simply the string representing the *startUrl*, i.e. the URL given to PubFetcher for fetching content for. The resolved *finalUrl* might be different than the *startUrl* (for example a redirection from HTTP to HTTPS might happen), meaning there might be webpages and docs with equal final URLs (that had different start URLs) stored in the database. Also to note, that webpages and docs have the same structure, they just provide two entirely separate stores for saving general web pages and documentation web pages respectively.

Publications can be identified by 3 separate IDs: *a PMID*, *a PMCID* or *a DOI*. Therefore, the following is done. A key – which can be called the primary ID of the publication – in the publications store is either a PMID, a PMCID or a DOI, depending on which of them was non-empty when the publication was first saved to the database. If more than one of them was available, then the PMID is preferred over the PMCID and the PMCID is preferred over the DOI. Then, there is an extra store called "publicationsMap", where a key is an ID (PMID/PMCID/DOI) of a publication and the corresponding value is the primary ID (PMID/PMCID/DOI) of that publication. So, for example, if a publication is to be loaded from the database, first publicationsMap is consulted to find the primary ID and then the found primary ID used to find the publication from the publications store. All the mappings in publicationsMap can be dumped to stdout with `-db-publications-map`. There is also a store called "publicationsMapReverse", which has mappings that are the reverse of the publicationsMap mappings, that is, from primary ID to the triplet PMID, PMCID, DOI. In addition, publicationsMapReverse stores the URLs where these PMID, PMCID and DOI were found. This reverse mapping can be useful, for example, for quickly listing all publication IDs (as the triplet PMID, PMCID, DOI) found in a database file. All the mappings in publicationsMapReverse can be dumped to stdout with `-db-publications-map-reverse`. The stores publicationsMapReverse and publicationsMap and the publications store are all kept coherent and in sync with each other. Also to note, that all stored DOIs are normalised, i.e. any valid prefix is removed (e.g. "https://doi.org/", "doi:") and letters from the 7-bit ASCII set are converted to uppercase.

The structure of the values in the publications, webpages and docs stores, i.e. the actual *contents* stored in the database, is best described by the next section *JSON output*, as the entire content of the database can be exported to an equivalently structured JSON file. To note, all the "empty", "usable", "final", "totallyFinal" and "broken" fields present in the JSON output are not stored in the database, but these values are inferred from actual database values and depend on some *fetching* parameters. Additionally, the fields "version" and "argv" are only specific to JSON.

With a new release of PubFetcher, the structure of the database content might change (this involves code in the package org.edammap.pubfetcher.core.db). Currently, there is no database migration support, which means that the content of existing database files will be become unreadable in case of structure updates. If that content is still required, it would need to be refetched to a new database file (created with the new version of PubFetcher).

3.2 JSON output

The output of PubFetcher will be in JSON format if the option `--format json` is specified. If the option `--plain` is additionally specified, then fields about metadata will be omitted from the output. JSON support is implemented using libraries from the Jackson project.

3.2.1 Common

All JSON output will contain the fields "version" and "argv".

version Information about the application that generated this JSON file

- **name** Name of the application
- **url** Homepage of the application
- **version** Version of the application

argv Array of all command-line parameters that were supplied to the application that generated this JSON file

3.2.2 IDs

JSON output of IDs/URLs, output using `-out-ids`, `-txt-ids-pub`, `-txt-ids-web` or `-txt-ids-doc`.

3.2.2.1 IDs of publications

Publications are identified by the triplet PMID, PMCID and DOI.

publicationIds Array of publication IDs

- **pmid** The PubMed ID of the publication. Only articles available in PubMed can have this.
- **pmcid** The PubMed Central ID of the publication. Only articles available in PMC can have this.
- **doi** The Digital Object Identifier of the publication
- **pmidUrl** Provenance URL of the PMID
- **pmcidUrl** Provenance URL of the PMCID
- **doiUrl** Provenance URL of the DOI

If `--plain` is specified, then the provenance URLs are not output.

3.2.2.2 URLs of webpages

Webpages are identified by a URL.

webpageUrls Array of webpage URLs

3.2.2.3 URLs of docs

Docs are identified by a URL.

docUrls Array of doc URLs

3.2.3 Contents

JSON output of the entire content of publications, webpages and docs, output using `-out`, `-txt-pub`, `-txt-web` and `-txt-doc`.

3.2.3.1 Content of publications

A publication represents one publication (most often a research paper) and contains its ID (*a PMID*, *a PMCID* and/or *a DOI*), content (title, abstract, full text), keywords (user-assigned, MeSH and mined EFO and GO terms) and various metadata (Open Access flag, journal title, publication date, etc).

publications Array of publications

- **fetchTime** Time of initial fetch or last *retryCounter* reset as UNIX time (in milliseconds)
- **fetchTimeHuman** Time of initial fetch or last *retryCounter* reset as ISO 8601 combined date and time
- **retryCounter** A refetch can occur if the value of *retryCounter* is less than *retryLimit*; or if any of the cooldown times (in *fetching* parameters) of a currently `true` condition have passed since *fetchTime*, in which case *retryCounter* is also reset
- **fetchException** `true` if there was a fetching exception during the last fetch; `false` otherwise
- **oa** `true` if the article is Open Access; `false` otherwise
- **journalTitle** Title of the journal the article was published in
- **pubDate** Publication date of the article as UNIX time (in milliseconds); negative, if unknown
- **pubDateHuman** Publication date of the article as ISO 8601 date; before `1970-01-01`, if unknown
- **citationsCount** Number of times the article has been cited (according to *Europe PMC*); negative, if unknown
- **citationsTimestamp** Time when *citationsCount* was last updated as UNIX time (in milliseconds); negative, if *citationsCount* has not yet been updated
- **citationsTimestampHuman** Time when *citationsCount* was last updated as ISO 8601 combined date and time; before `1970-01-01T00:00:00.000Z`, if *citationsCount* has not yet been updated
- **correspAuthor** Array of objects representing corresponding authors of the article
  - **name** Name of the corresponding author
  - **orcid** ORCID iD of the corresponding author

- **email** E-mail of the corresponding author
- **phone** Telephone number of the corresponding author
- **uri** Web page of the corresponding author

visitedSites Array of objects representing sites visited for getting content (outside of standard Europe PMC, PubMed and oaDOI *resources* and also excluding PDFs)

- **url** URL of the visited site
- **type** *The type* of the site (as resource)
- **from** URL where the link of the site was picked up
- **timestamp** Time when the link of the site was picked up as UNIX time (in milliseconds)
- **timestampHuman** Time when the link of the site was picked up as ISO 8601 combined date and time

empty `true`, if all *publication part*s (except IDs) are *empty*; `false` otherwise

usable `true`, if at least one *publication part* (apart from IDs) is *usable*; `false` otherwise

final `true`, if *title*, *abstract* and *fulltext* are *final*; `false` otherwise

totallyFinal `true`, if all *publication part*s are *final*; `false` otherwise

pmid A *publication part* (like the following *pmcid*, *doi*, *title*, etc), in this case representing the publication PMID

- **content** Content of the publication part (in this case, the publication PMID as a string)
- **type** *The type* of the publication part content source
- **url** URL of the publication part content source
- **timestamp** Time when the publication part content was set as UNIX time (in milliseconds)
- **timestampHuman** Time when the publication part content was set as ISO 8601 combined date and time
- **size** Number of characters in the content
- **empty** `true`, if the content is empty (size is `0`); `false` otherwise
- **usable** `true`, if the content is long enough (the threshold can be influenced by *fetching* parameters), in other words, if the publication part content can be used as input for other applications; `false` otherwise
- **final** `true`, if the content is from a reliable source and is long enough, in other words, if there is no need to try fetching the publication part content from another source; `false` otherwise

pmcid Publication part representing the publication PMCID. Structure same as in *pmid*.

doi Publication part representing the publication DOI. Structure same as in *pmid*.

title Publication part representing the publication title. Structure same as in *pmid*.

keywords Publication part representing publication keywords. Structure same as in *pmid*, except *content* is replaced with "list" and *size* is number of elements in "list".

- **list** Array of string representing publication keywords

mesh Publication part representing publication MeSH terms. Structure same as in *pmid*, except *content* is replaced with "list" and *size* is number of elements in "list".

- **list** Array of objects representing publication MeSH terms

term Term name

majorTopic `true`, if the term is a major topic of the article

uniqueId MeSH Unique Identifier

efo Publication part representing publication EFO and other experimental methods terms. Structure same as in *pmid*, except *content* is replaced with "list" and *size* is number of elements in "list".

list Array of objects representing publication EFO terms

term Term name

count Number of times the term was mined from full text by *Europe PMC*

uri Unique URI to the ontology term

go Publication part representing publication GO terms. Structure same as in *efo*.

abstract Publication part representing the publication abstract. Structure same as in *pmid*.

fulltext Publication part representing the publication fulltext. Structure same as in *pmid*.

If `--plain` is specified, then metadata is omitted from the output (everything from *fetchTime* to *totallyFinal*) and the value corresponding to *publication part* keys (*pmid* to *fulltext*) will be the value of *content* (for *pmid*, *pmcid*, *doi*, *title*, *abstract*, *fulltext*) or the value of "list" (for *keywords*, *mesh*, *efo*, *go*) as specified above for each corresponding part.

If `--out-part` is specified, then everything from *fetchTime* to *totallyFinal* will be omitted from the output and only *publication part*s specified by `--out-part` will be output (with structure as specified above). If `--plain` is specified along with `--out-part`, then output parts will only have as value the value of *content* (for *pmid*, *pmcid*, *doi*, *title*, *abstract*, *fulltext*) or the value of "list" (for *keywords*, *mesh*, *efo*, *go*).

3.2.3.2 Content of webpages

A webpage represents a general web page from where relevant content has been extracted, along with some metadata. If the web page is about a software tool, then the software license and programming language can be stored separately, if found (this feature has been added to support EDAMmap).

webpages Array of webpages

fetchTime Same as *fetchTime* of publications

fetchTimeHuman Same as *fetchTimeHuman* of publications

retryCounter Same as *retryCounter* of publications

fetchException Same as *fetchException* of publications

startUrl URL given as webpage identifier, same as listed by *webpageUrls*

finalUrl Final URL after potential redirections

contentType HTTP Content-Type header

statusCode HTTP status code

contentTime Time when current webpage content was last set as UNIX time (in milliseconds)

contentTimeHuman Time when current webpage content was last set as ISO 8601 combined date and time

license Software license of the tool the webpage is about (empty if not found or missing corresponding *scraping rule*)

language Programming language of the tool the webpage is about (empty if not found or missing corresponding *scraping rule*)

titleLength Number of characters in the *webpage title*

contentLength Number of characters in the *webpage content*

title The webpage title (as extracted by the corresponding *scraping rule*; or text from the HTML `<title>` element if scraping rules were not found)

empty `true`, if *webpage title* and *webpage content* are empty; `false` otherwise

usable `true`, if the length of *webpage title* plus the length of *webpage content* is large enough (at least *webpageMinLength* characters), that is, the webpage can be used as input for other applications; `false` otherwise

final `true`, if the webpage is not *broken* and the webpage is *usable* and the length on the *webpage content* is larger than 0; `false` otherwise

broken `true`, if the webpage with the given URL could not be fetched (based on the values of *statusCode* and *finalUrl*); `false` otherwise

content The webpage content (as extracted by the corresponding *scraping rule*; or the *automatically cleaned* content from the entire HTML of the page if scraping rules were not found)

If `--plain` is specified, then only *startUrl*, *webpage title* and *webpage content* will be present.

3.2.3.3 Content of docs

Like *Content of webpages*, except it allows for a separate store for documentation web pages.

docs Array of docs

Structure is same as in *webpages*

3.3 HTML and plain text output

Output will be in HTML format, if `--format html` is specified, and in plain text, if `--format text` is specified or `--format` is omitted (as `text` is the default).

The HTML output is meant to be formatted and viewed in a web browser. Links to external resources (such as the different URL fields) are clickable in the browser.

The plain text output is formatted for viewing in the console or in a text editor.

Both the HTML output and the plain text output will contain the same information as the *JSON output* specified above and will behave analogously in respect to the `--plain` and `--out-part` parameters. There are however a few fields that are missing in HTML and plain text compared to JSON: "empty", "usable", "final", "totallyFinal", "broken" (these values are inferred from the values of some other fields and depend on some *fetching* parameters) and the JSON specific "version" and "argv".

3.4 Log file

PubFetcher-CLI will log to stderr using the Apache Log4j 2 library. With the `--log` parameter (described in *Logging*), a text file where the same log will be output to can be specified.

Each log line will consist of the following: the data and time, log level, log message, the name of the logger that published the logging event and the name of the thread that generated the logging event. The date and time will be the local time in the format "2018-08-24 11:37:20,187". Log level can be DEBUG, INFO, WARN and ERROR. DEBUG level messages are only output to the log file (and not to the console). Currently, there are only few DEBUG messages, including the very first message listing all parameters the program was run with. Any line breaks in the log message will be escaped, so that each log message can fit on exactly one line. The name of the logger is just the fully qualified Java class (with the prefix "org.edamontology" removed) the logging event is called from (prepended with "@" in the log file), e.g. "@pubfetcher.cli.Cli". The name of the thread will be "main" if the logging event was generated by the main thread, any subsequent thread will be named "Thread-2", "Thread-3", etc. In the log file the thread name will be in square brackets, e.g. "[Thread-2]". Some Java exceptions can also be logged, these will be output with the stack trace on subsequent lines after the logged exception message.

3.4.1 Analysing logs

Log level ERROR is set to erroneous conditions which mostly occur on the side of the PubFetcher user (like problems in provided input), so searching for "ERROR" in log files can potentially help in finding problems that can be fixed by the user. Some problems might be caused by issues in the used *resources*, like *Europe PMC* and PubMed, and some reported problems are not problems at all, like failing to find a *publication part* which is actually supposed to be missing, but these messages will usually have the log level WARN. One example of WARN level messages that can indicate inconsistencies in used resource are the messages beginning with "Old ID".

Some examples of issues found by analysing logs:

- https://github.com/bio-tools/biotoolsRegistry/issues/281
- https://github.com/bio-tools/biotoolsRegistry/issues/331
- https://github.com/bio-tools/biotoolsRegistry/issues/332

If multiple threads are writing to a log file, then the messages of different threads will be interwoven. To get the sequence of messages of only one thread, `grep` could be used:

```
$ grep Thread-2 database.log
```

In addition to analysing logs, the output of `-part-table` (described in *Output*) could be checked for possible problems. For example, *title* being "na" is a good indicator of an invalid ID. To list all such *publications* the filter `-part-type na -part-type-part title` could be used. Other things of interest might be for example parts which are from other sources than the main ones (the europepmc, pubmed, pmc types and doi) or parts missing in *Europe PMC*, but present in PubMed or PMC.

CHAPTER 4

Fetching logic

The functionality explained in this section is mostly implemented in the source code file Fetcher.java.

4.1 Low-level methods

4.1.1 Getting a HTML document

Fetching HTML (or XML) resources for both *publications* and *webpages*/*docs* is done in the same method, where either the jsoup or HtmlUnit libraries are used for getting the document. The HtmlUnit library has the advantage of supporting JavaScript, which needs to be executed to get the proper output for many sites, and it also works for some sites with problematic SSL certificates. As a disadvantage, it is a lot slower than jsoup, which is why using jsoup is the default and HtmlUnit is used only if JavaScript support is requested (or switched to automatically in case of some SSL exceptions). Also, fetching with JavaScript can get stuck for a few rare sites, in which case the misbehaving HtmlUnit code is terminated.

Supplied *fetching* parameters *timeout* and *userAgent* are used for setting the connect timeout and the read timeout and the User-Agent HTTP header of connections. If getting the HTML document for a publication is successful and a list of already fetched links is supplied, then the current URL will be added to that list so that it is not tried again for the current publication. The successfully fetched document is returned to the caller for further processing.

A number of exceptions can occur, in which case getting the HTML document has failed and the following is done:

MalformedURLException The protocol is not HTTP or HTTPS or the URL is malformed. Getting the URL is tried again as a PDF document, as a few of these exception are caused by URLs that point to PDFs accessible through the FTP protocol.

HttpStatusException or FailingHttpStatusCodeException The *fetchException* of the publication, webpage or doc is set to `true` in case the HTTP status code in the response is `503 Service Unavailable`. Setting *fetchException* to `true` means the URL can be tried again in the future (depending on *retryCounter* or *fetchExceptionCooldown*) as the `503` code is usually a temporary condition. Additionally, in case of publications, *fetchException* is set to `true` for all failing HTTP status codes if the URL is not from "doi.org" and it is not a URL pointing to a PDF, PS or GZIP file.

ConnectException or NoRouteToHostException An error occurred while attempting to connect a socket to a remote address and port. Set *fetchException* to `true`.

UnsupportedMimeTypeException The response MIME type is not supported. If the MIME type is determined to be a PDF type, then getting the URL is tried again, but as a PDF document.

SocketTimeoutException A timeout has occurred on a socket read or accept. A new attempt is made right away and if that also fails with a timeout, then *fetchException* is set to `true`.

SSLHandshakeException or SSLProtocolException Problem with SSL. If fetching was attempted with jsoup, then it is attempted once more, but with HtmlUnit.

IOException A connection or read error occurred, just issue a warning to the log.

Exception Some other checked exception has occurred, set *fetchException* to `true`.

The HTML document fetching method can be tested with the *CLI commands* `-fetch-document` or `-fetch-document-javascript` (but without publications, webpages, docs and PDF support).

4.1.2 Getting a PDF document

Analogous to *getting a HTML document*. The Apache PDFBox library is used for extracting content and metadata from the PDF. The method for getting a PDF document is called upon if the URL is known in advance to point to a PDF file or if this fact is found out during the fetching of the URL as a HTML document.

Nothing is returned to the caller, as the supplied *publication*, *webpage* or *doc* is filled directly. For webpages and docs, all the text extracted from the PDF is set as their content, and if a title is found among the PDF metadata, it is set as their title. For publications, the text extracted from the PDF is set to be the *fulltext*. Also, *title*, *keywords* or *theAbstract* are filled with content found among the PDF metadata, but as this happens very rarely, fetching of the PDF is not done at all if the *fulltext* is already *final*.

4.1.3 Selecting from the returned HTML document

The fetched HTML is parsed to a jsoup Document and returned to the caller.

Then, parts of the document can be selected to fill the corresponding fields of *publications*, *webpages* and *docs* using the jsoup CSS-like element Selector. This is explained in more detail in the *Scraping rules* section.

Testing fetching of HTML (and PDF) documents and selecting from them can be done with the *CLI operation* `-fetch-webpage-selector`.

4.1.4 Cleaning the returned HTML document

If no *selectors* are specified for the given HTML document, then automatic cleaning and formatting of the document will be done instead.

The purpose of cleaning is to only extract the main content, while discarding auxiliary content, like menus and other navigational elements, footers, search and login forms, social links, contents of `<noscript>`, publication references, etc. We clean the document by deleting such elements and their children. The elements are found by tag names (for example `<nav>` or `<footer>`), but also their IDs, class names and ARIA roles are matched with combinations of keywords. Some words (like “menu” or “navbar”) are good enough to outright delete the matched element, either matching it by itself or with a specifier (like “left” or “main”) or in combination with another word (like “tab” or “links”). Other words (like the mentioned “tab” and “links”, but also “bar”, “search”, “hidden”, etc), either by themselves or combined with specifiers, are not specific enough to delete the matched element without some extra confidence. So, for these words and combinations there is the extra condition that no children or parents of the matched element can be an element that we determine to be about the main content (`<main>`, `<article>`, `<h1>`, "content", "heading", etc).

After this cleaning has been done, the remaining text will be extracted from the document and formatted. Paragraphs and other blocks of text will be separated by empty lines in the output. If any text is found in the description `<meta>` tag, then it will be prepended to the output.

4.1.5 Multithreaded fetching

Only one thread should be filling one publication or one webpage or one doc. But many threads can be filling different *publications*, *webpages* and *docs* in parallel. If many of these threads depend on the same resources, then what can happen is many parallel connections to the same host. To avoid such hammering, locking is implemented around each connection such that only one connection to one host is allowed at once (comparison of hosts is done case-insensitively and "www." is removed). Other threads wanting to connect to the same host will have to wait until the resource is free again.

4.2 Fetching publications

4.2.1 Resources

Unfortunately, all content pertaining to a *publication* is not available from one sole Internet resource. Therefore, a number of resources are consulted and the final publication might contain content from different resources, for example an abstract from one place and the full text from another.

What follows is a list of these resources. They are defined in the order they are tried: if after fetching a given resource all required *publication parts* become *final*, or none of the subsequent resources can fill the missing parts, then the resources below the given resource are not fetched from.

But, if after going through all the resources below (as necessary) more IDs about the publication are known than before consulting the resources, then another run through all the resources is done, starting from the first (as knowing a new ID might enable us to query a resource that couldn't be queried before). In doing this we are keeping track of resources that have successfully been fetched to not fetch these a second time and of course, for each resource, we are still evaluating if the resource can provide us with anything useful before fetching is attempted.

Sometimes, publication IDs can change, e.g., when we find from a resource with better type (see *Publication types*) that the DOI of the publication is different than what we currently have. In such cases all publication content (except IDs) is emptied and fetching restarted from scratch.

4.2.1.1 Europe PMC

Europe PubMed Central is a repository containing, among other things, abstracts, full text and preprints of biomedical and life sciences articles. It is the primary resource used by PubFetcher and a majority of content can be obtained from there.

The endpoint of the API is https://www.ebi.ac.uk/europepmc/webservices/rest/search, documentation is at https://europepmc.org/RestfulWebService. The API accepts any of the publication IDs: either a *PMID*, a *PMCID* or a *DOI*. With parameter *europepmcEmail* an e-mail address can be supplied to the API.

We can possibly get all *publication parts* from the Europe PMC API, except for *fulltext*, *efo* and *go* for which we get a `Y` or `N` indicating if the corresponding part is available at the *Europe PMC fulltext* or *Europe PMC mined* resource. In addition, we can possibly get values for the publication fields *oa*, *journalTitle*, *pubDate* and *citationsCount*. Europe PMC is currently the only resource we can get the *citationsCount* value from.

Europe PMC itself has content from multiple sources (see https://europepmc.org/Help#contentsources) and in some cases multiple results are returned for a query (each from a different source). In that case the MED (MEDLINE) source is preferred, then PMC (PubMed Central), then PPR (preprints) and then whichever source is first in the list of results.

4.2.1.2 Europe PMC fulltext

Full text from the Europe PMC API is obtained from a separate endpoint: https://www.ebi.ac.uk/europepmc/webservices/rest/{PMCID}/fullTextXML. The *PMCID* of the publication must be known to query the API.

The API is primarily meant for getting the *fulltext*, but it can also be used to get the parts *pmid*, *pmcid*, *doi*, *title*, *keywords*, *theAbstract* if these were requested and are still non-*final* (for some reason not obtained from the main resource of Europe PMC). In addition, *journalTitle* and *correspAuthor* can be obtained.

4.2.1.3 Europe PMC mined

Europe PMC has text-mined terms from publication full texts. These can be fetched from the API endpoint https://www.ebi.ac.uk/europepmc/annotations_api/annotationsByArticleIds, documentation of the Annotations API is at https://europepmc.org/AnnotationsApi. These resources are the only way to fill the *publication parts efo* and *go* and only those publication parts can be obtained from these resources (type "Gene Ontology" is used for GO and type "Experimental Methods" for EFO). Either a *PMID* or a *PMCID* is required to query these resources.

4.2.1.4 PubMed XML

The PubMed resource is used to access abstracts of biomedical and life sciences literature from the MEDLINE database.

The following URL is used for retrieving data in XML format for an article: https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?retmode=xml&db=pubmed&id={PMID}. As seen, a *PMID* is required to query the resource. Documentation is at https://www.ncbi.nlm.nih.gov/books/NBK25500/.

In addition to *theAbstract*, the *publication parts pmid*, *pmcid*, *doi*, *title* and *mesh* can possibly be obtained from PubMed. Also, the publication part *keywords* can seldom be obtained, but if *keywords* is the only still missing publication part, then the resource is not fetched (instead, PubMed Central is relied upon for *keywords*). In addition, we can possibly get values for the publication fields *journalTitle* and *pubDate*.

4.2.1.5 PubMed HTML

Information from PubMed can be ouput in different formats, including in HTML (to be viewed in the browser) from the URL: https://www.ncbi.nlm.nih.gov/pubmed/?term={PMID}. By scraping the resultant page we can get the same *publication parts* as from the XML obtained through PubMed E-utilities, however the HTML version of PubMed is only fetched if by that point *title* or *theAbstract* are still non-*final* (i.e., *PubMed XML*, but also Europe PMC, failed to fetch these for some reason). So this is more of a redundant resource, that is rarely used and even more rarely useful.

4.2.1.6 PubMed Central

PubMed Central contains full text articles, which can be obtained in XML format from the URL: https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?retmode=xml&db=pmc&id={PMCID}, where {PMCID} is the *PMCID* of the publication with the "PMC" prefix removed.

It is analogous to *Europe PMC fulltext* and used as a backup to that resource for getting content for articles available in the PMC system.

4.2.1.7 DOI resource

Sometimes, some *publication parts* must be fetched directly from the publisher. A *DOI* (Digital Object Identifier) of a publication is a persistent identifier which, when resolved, should point to the correct URL of the journal article.

First, the DOI is resolved to the URL it redirects to and this URL is fed to the *Getting a HTML document* method. If the URL has a match in the JavaScript section of the *Journals YAML* scraping configuration, then the HTML document will be fetched using JavaScript support. The publication parts that can possibly be scraped from the article's page are *doi*, *title*, *keywords*, *theAbstract*, *fulltext* and possibly (but very rarely) *pmid* and *pmcid*. These publication parts are extracted from the web page using corresponding *scraping rules*. If no *scraping rules* are found, then the content of the HTML `<title>` element will be set as the value of the publication part *title* (if *title* is still non-*final*) and the whole text of the HTML set as the value of *fulltext* (if *fulltext* is still non-*final*). Additionally, a link to the web page containing the full text of the article and a link pointing to the article PDF might be added to *Links*, if specified by the *scraping rules*, and in addition names and e-mails for *correspAuthor* can be found.

In contrast to the other resources, `<meta>` elements are looked for in the HTML as these might contain the publication parts *pmid*, *pmcid*, *doi*, *title*, *keywords* and also *theAbstract*, plus *Links* to additional web pages or PDFs containing the article and sometimes also e-mail addresses for *correspAuthor*. More about these meta tags is described in *Meta*.

Also, in contrast to other resources, the final URL resolved from the DOI is added to *visitedSites*.

4.2.1.8 Unpaywall

The Unpaywall service helps to find Open Access content. It us mainly useful for finding PDFs for some articles for which no full text content was found using the above resources, but it can help in filling a few other *publication parts* and fields also, such as *oa*. The service was recently called oaDOI.

The API is queried as follows: https://api.unpaywall.org/v2/{DOI}?email={*oadoiEmail*}, documentation is at https://unpaywall.org/products/api. As seen, the *DOI* of the publication must be known to query the service.

The response will be in JSON format, which is why the method of *Getting a HTML document* is not used (but the process of obtaining the resource is analogous). Unpaywall will be called if *title*, *theAbstract* of *fulltext* are non-*final* (or *pmid*, *pmcid*, *doi* are non-*final*, but only if these are the only publication parts requested). From the response we can possibly directly fill the publication part *title* and the fields *oa* and *journalTitle*. But in addition we can find *Links* to web pages containing the article or to PDFs of the article.

4.2.1.9 Meta

The web pages of journal articles can have metadata embedded in the HTML in `<meta>` elements. Sometimes this can be used to fill *publication parts* which have not been found elsewhere.

There are a few standard meta tag formats, those supported by PubFetcher are: HighWire, EPrints, bepress, Dublin Core, Open Graph, Twitter and generic tag (without any prefix). An example of a HighWire tag: `<meta name="citation_keyword" content="foo">`. An example of a Open Graph tag: `<meta property="og:title" content="bar" />`.

Publication parts potentially found in `<meta>` elements (depending on format) are: *pmid*, *pmcid*, *doi*, *title*, *keywords*, *theAbstract*. Additionally, *Links* to web pages containing the article or to PDFs of the article can be found in some meta tags.

In web pages of articles of some journals the standard `<meta>` tags are filled with content that is not entirely correct (for our purposes), so some exceptions to not use these tags for these journals have been defined.

`<meta>` elements are only searched for in all web pages resolved from DOI and also in all web pages added to *Links*.

4.2.1.10 Links

Links to web pages containing an article or to PDFs of the article can be found in the *Unpaywall* resource, in some *Meta* tags and in web pages (resolved from *DOI* or from *Links*) that have *scraping rules* specifying how to extract links. In addition to its URL, a publication type (see *Publication types*) corresponding to the resource the link was found from, the URL of the web page the link was found from and a timestamp, are saved for each link.

These links are collected in a list that will be looked through only after all other resources above have been exhausted. *DOI* links (with host "doi.org" or "dx.doi.org") and links to web pages of articles in the PMC system (either Europe PMC or PubMed Central) are not added to this list. But, in case of PMC links, a missing *PMCID* (or *PMID*) of the publication can sometimes be extracted from the URL string itself. In addition, links that have already been tried or links already present in the list are not added to the list a second time.

Links are sorted according to *publication types* in the list they are collected to, with links of *final* type on top. Which means, that once fetching of resources has reached this list of links then links of higher types are visited first. If *publication parts title*, *keywords*, *theAbstract* and *fulltext* are *final* or with types that are better or equal to types of any of the remaining links in the list, then the remaining links are discarded.

In case of links to web pages the content is fetched and the publication is filled the same way as in the *DOI resource* (including the addition of the link to *visitedSites*), except the resolving of the DOI to URL step is not done (the supplied URL of the link is treated the same as a URL resolved from a DOI). In case of links to PDFs the content is fetched and the publication is filled as described in *Getting a PDF document*.

4.2.2 Publication types

Publication part types are the following, ordered from better to lower type:

europepmc Type given to parts got from Europe PMC and *Europe PMC mined* resources

europepmc_xml From *Europe PMC fulltext* resource

europepmc_html Currently disabled

pubmed_xml From *PubMed XML* resource

pubmed_html From *PubMed HTML* resource

pmc_xml From PubMed Central resource

pmc_html Currently disabled

doi From *DOI resource* (excluding PDF links)

link Link to publication. Not used in PubFetcher itself. Meant as an option in applications extending or using PubFetcher.

link_oadoi Given to *Links* found in *Unpaywall* resource (excluding PDF links)

citation From HighWire *Meta* tags (excluding links)

eprints From EPrints *Meta* tags (excluding links)

bepress From bepress *Meta* tags (excluding PDF links)

link_citation *Links* from Highwire *Meta* tags (excluding PDF links)

link_eprints *Links* from EPrints *Meta* tags (excluding PDF links)

dc From Dublin Core *Meta* tags

og From Open Graph *Meta* tags

twitter From Twitter *Meta* tags

meta From generic *Meta* tags (excluding links)

link_meta *Links* from generic *Meta* tags (excluding PDF links)

external Type given to externally supplied *pmid*, *pmcid* or *doi*

oadoi From *Unpaywall* resource (excluding links, currently only *title*)

pdf_europepmc Currently disabled

pdf_pmc Currently disabled

pdf_doi Type given to PDF *Links* extracted from a *DOI resource* or if the DOI itself resolves to a PDF file (which is fetched as described in *Getting a PDF document*)

pdf_link PDF from link to publication. Not used in PubFetcher itself. Meant as an option in applications extending or using PubFetcher.

pdf_oadoi PDF *Links* from *Unpaywall* resource

pdf_citation PDF *Links* from HighWire *Meta* tags

pdf_eprints PDF *Links* from EPrints *Meta* tags

pdf_bepress PDF *Links* from bepress *Meta* tags

pdf_meta PDF *Links* from generic *Meta* tags

webpage Type given to *title* and *fulltext* set from an article web page with no *scraping rules*

na Initial type of a publication part

Types "europepmc", "europepmc_xml", "europepmc_html", "pubmed_xml", "pubmed_html", "pmc_xml", "pmc_html", "doi", "link" and "link_oadoi" are final types. Final types are the best type and they are equivalent with each other (meaning that one final type is not better than some other final type and their ordering does not matter).

The type of the publication part being final is a necessary condition for the publication part to be *final*. The other condition is for the publication part to be large enough (as specified by *titleMinLength*, *keywordsMinSize*, *minedTermsMinSize*, *abstractMinLength* or *fulltextMinLength* in *fetching* parameters). The *fulltext* part has the additional requirement of being better than "webpage" type to be considered *final*.

When filling a publication part then the type of the new content must be better than the type of the old content. Or, if both types are final but the publication part itself is not yet *final* (because the content is not large enough), then new content will override old content if new content is larger. Publication parts which are *final* can't be overwritten. Also, the publication fields (these are not publication parts) *journalTitle*, *pubDate* and *correspAuthor* can only be set once with non-empty content, after which they can't be overwritten anymore.

4.2.3 Publication parts

publication parts have *content* and contain the fields *type*, *url* and *timestamp* as described in the *JSON output* of the publication part *pmid*. The publication fields *oa*, *journalTitle*, *pubDate*, etc do not contain extra information besides content and are not publication parts.

The publication parts are as follows:

pmid The PubMed ID of the publication. Only articles available in PubMed can have this. Only a valid PMID can be set to the part. The *pmid structure*.

pmcid The PubMed Central ID of the publication. Only articles available in PMC can have this. Only a valid PMCID can be set to the part. The *pmcid structure*.

doi The Digital Object Identifier of the publication. Only a valid DOI can be set to the part. The DOI will be normalised in the process, i.e. any valid prefix (e.g. "https://doi.org/", "doi:") is removed and letters from the 7-bit ASCII set are converted to uppercase. The *doi structure*.

title The title of the publication. The *title structure*.

keywords Author-assigned keywords of the publication. Often missing or not found. Empty and duplicate keywords are removed. The *keywords structure*.

mesh Medical Subject Headings terms of the publication. Assigned to articles in PubMed (with some delay after publication). The *mesh structure*.

efo Experimental factor ontology terms of the publication (but also experimental methods terms from other ontologies like Molecular Interactions Controlled Vocabulary and Ontology for Biomedical Investigations). Text-mined by the Europe PMC project from the full text of the article. The *efo structure*.

go Gene ontology terms of the publication. Text-mined by the Europe PMC project from the full text of the article. The *go structure*.

theAbstract The abstract of the publication. The part is called "theAbstract" instead of just "abstract", because "abstract" is a reserved keyword in the Java programming language. The *abstract structure*.

fulltext The full text of the publication. The part includes the title and abstract of the publication in the beginning of the content string. All the main content of the article's full text is included, from introduction to conclusions. Captions of figures and tables and descriptions of supplementary materials are also included. From back matter, the glossary, notes and misc sections are usually included. But acknowledgements, appendices, biographies, footnotes, copyrights and, most importantly, references are excluded, whenever possible. If fulltext is obtained from a PDF, then everything is included. In the future, it could be useful to include all these parts of full text, like references, but in a structured way. The *fulltext structure*.

4.3 Fetching webpages and docs

A *webpage* or *doc* is also got using the method described in *Getting a HTML document* (or *Getting a PDF document* if the webpage or doc URL turns out to be a link to a PDF file). Webpage and doc fields that can be filled from the fetched content using *scraping rules* are the *webpage title*, the *webpage content*, *license* and *language*. Other fields are filled with metadata during the fetching process, the whole structure can be seen in *webpages* section of the output documentation. If no *scraping rules* are present for the webpage or doc then the *webpage content* will be the entire string parsed from the fetched HTML and the *webpage title* will be the content inside the `<title>` tag. Whether the webpage or doc is fetched with JavaScript support or not can also be influenced with *scraping rules*. A webpage or doc can also be fetch using rules specified on the command line with the command `-fetch-webpage-selector` (see *Print a web page*).

The same publication can be fetched multiple times, with each fetching potentially adding some missing content to the existing publication. In contrast, a webpage or doc is always fetched from scratch. If the resulting *webpage or doc is final* and a corresponding webpage or doc already exists, then this existing entry will be overwritten. An existing webpage or doc will also be overwritten, if the new entry is non-final (but not empty) and the old entry is non-final (and potentially empty) and if both new and old entries are empty.

4.4 Can fetch

The methods for fetching *publications*, *webpages* and *docs* are always given a publication, webpage or doc as parameter. If a publication, webpage or doc is fetched from scratch, then an initial empty entry is supplied. Each time, these methods have to determine if a publication, webpage or doc can be fetched or should the fetching be skipped this time. The fetching will happen if any of the following conditions is met:

- *fetchTime* is `0`, this is only true for initial empty entries;
- the *publication is empty* or the *webpage or doc is empty* and *emptyCooldown* is not negative and at least *emptyCooldown* minutes have passed since *fetchTime*;
- the *publication is final* or the *webpage or doc is final* (and they are not empty) and *nonFinalCooldown* is not negative and at least *nonFinalCooldown* minutes have passed since *fetchTime*;
- the entry has a *fetchException* and *fetchExceptionCooldown* is not negative and at least *fetchExceptionCooldown* minutes have passed since *fetchTime*;
- the entry is empty or non-final or has a *fetchException* and either *retryCounter* is less than *retryLimit* or *retryLimit* is negative.

If it was determined that fetching happens, then *fetchTime* is set to the current time and *retryCounter* is reset to `0` if any condition except the last is met. If only the last condition (about *retryCounter* and *retryLimit*) is met, then *retryCounter* is incremented by `1` (and *fetchTime* is left as is, meaning that *fetchTime* does not necessarily show the time of the last fetching, but only the time of the initial fetching or the time when fetching happened because one of the cooldown timers expired).

The *fetchException* is set to `false` in the beginning of each fetching and it is set to `true` if some certain types of errors happen during fetching, some such error conditions are described in *Getting a HTML document*. *fetchException* can be set to `true` also by the method described in *Getting a PDF document* and the custom method getting the *Unpaywall* resource.

CHAPTER 5

Scraping rules

5.1 Scraping

After *Getting a HTML document* is done, we receive a jsoup Document from it. Then, the jsoup CSS-like element Selector can be used to select parts of the HTML or XML document for filling corresponding fields of a *publication*, *webpage* or *doc*. See also documentation at https://jsoup.org/cookbook/extracting-data/selector-syntax. Note that, extracting content from XMLs received from an API is not really scraping, just the same interface and jsoup methods are used for both website HTMLs and API XMLs for simplicity.

In case of *publications*, only *publication parts* specified by the CLI *Get content modifiers* `--fetch-part`, or not specified by `--not-fetch-part`, will be extracted (except IDs, which are always extracted). If neither `--fetch-part` nor `--not-fetch-part` is specified then all publication parts will be extracted. All other publication fields (that are not publication parts) are always extracted when present.

In case of publication IDs (*pmid*, *pmcid*, *doi*), only the first ID found using the selector is extracted. Any "pmid:", "pmcid:", "doi:" prefix, case-insensitively and ignoring whitespace, is removed from the extracted ID and the ID set to the corresponding publication part, if valid. Problems, like not finding any content for the specified selector or an ID being invalid, are logged. In case of a publication *title*, also only the first found content using the selector is extracted. A few rare publications have also a subtitle – this is extracted separately and appended to the title with separator " : ". If the publication title, or any other publication part besides the IDs, is already *final* or the supplied selector is empty, then no extraction is attempted. For publication *keywords*, all elements found by the selector are used, each element being a separate keyword. But sometimes, the extracted string can be in the form "keywords: word1, word2", in which case the prefix is removed, case-insensitively and ignoring whitespace, and the keywords split on ",". The keyword separator could also be ";" or "|". For *abstract*s, all elements found by the selector are also extracted and concatenated to a final string, with `\n\n` (two line breaks) separating the elements. The same is done for *fulltext*s, but in addition the title, subtitle and abstract selectors are supplied to the fulltext selection method, because the fulltext part must also contain the title and abstract before the full text.

In case of publication fields *journalTitle*, *pubDate* and *citationsCount*, the first element found by the supplied selector is extracted. The date in *pubDate* may be broken down to subelements "Year", "Month" and "Day" in the XML. For *correspAuthor*, complex built-in selectors and extraction logic are needed, as the corresponding authors can be marked and additional data about them supplied in a variety of non-trivial ways.

For the resources *Europe PMC*, *Europe PMC fulltext*, *Europe PMC mined*, *PubMed XML*, *PubMed HTML* and

PubMed Central, the selector strings specifying how to extract the publication title, keywords etc are currently hard-coded, as the format of these resources is hopefully fairly static. But for the multitude of *DOI resources* and sites found in *Links*, the selectors are put in a configuration file (described in *Journals YAML*), as fairly often the format of a few sites can change.

For *webpages* and *docs*, there are no hardcoded selectors and all of them must be specified in a configuration file (described in *Webpages YAML*).

5.2 Rules in YAML

Scraping rules for journal web pages (resolved from *DOI resource* or in *Links*) and *webpages/docs* are specified in YAML configuration files. YAML parsing support is implemented using SnakeYAML.

There are built-in rules for both journal sites (in journals.yaml) and webpages/docs (in webpages.yaml). For adding scraping rules to non-supported sites, the location of a custom configuration file can be specified by the user: for journals using the parameter *journalsYaml* and for webpages/docs using the parameter *webpagesYaml*. In addition to adding rules, the default rules can be overridden. To do that, the top-level keys of the rules to be overridden must be repeated in the custom configuration file and the new desired values specified under those keys.

In case of problems in the configuration file – either errors in the YAML syntax itself or mistakes in adhering to the configuration format specified below – the starting of PubFetcher is aborted and a hopefully helpful error message output to the log.

The syntax of regular expressions used in the configuration file is as defined in Java, see documentation of the Pattern class: https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html. If the regex is meant to match a URL, then the string "(?i)^https?://(www.)?" is automatically put in front of the specified regular expression, except when the regular expression already begins with "^".

5.2.1 Journals YAML

Scraping rules for journal web pages (resolved from *DOI resource* or in *Links*), used for filling corresponding *publications*. To see an example of a journals YAML, the built-in rules file journals.yaml can be consulted.

One way to find journal web pages that are worth writing rules for is to use the *top hosts* functionality of the CLI on an existing collection of publications.

A journals YAML configuration file must have three sections (separated by `---`): *regex*, *site* and *javascript*.

5.2.1.1 regex

Entries in this section are in the form "regex: site". If the specified regular expression "regex" has a match with the URL resolved from *DOI resource* or the URL of the link taken from *Links*, then the corresponding value "site" will be the name of the rules used for scraping the web page at the URL. The mentioned URL is the final URL, i.e. the URL obtained after following all potential redirections. The rules for "site" must be present in the next section *site*. If multiple regular expressions have a match with the URL, then the "site" will be taken from the last such "regex".

If none of the regular expressions have a match with the URL, then no rules could be found for the site. In that case, the extracted *title* will have type "webpage" and as content the text value (until the first "|") of the document's `<title>` element and the extracted *fulltext* will have type "webpage" and as content the entire text parsed from the document. The extracted *title* and *fulltext* will fill corresponding publication parts if these parts were requested (as determined by `--fetch-part` or `--not-fetch-part`) and conditions described at the end of *Publication types* are met. No other publication parts besides *title* and *fulltext* can potentially be filled if no scraping rules are found for a site.

Different publishers might use a common platform, for example HighWire. In such cases the different keys "regex" for matching article web pages of these different publishers might point to a common "site" with rules for that common platform.

The `-scrape-site` command of the *CLI* can be used to test which "site" name is found from loaded configuration files for the supplied URL.

5.2.1.2 site

Entries in this section are in the form "site: rulesMap". Each rule name "site" in this section must be specified at least once in the previous section *regex*. In case of duplicate rule names "site", the last one will be in effect. The custom configuration file specified using *journalsYaml* is parsed after the built-in rules in journals.yaml, which means that by using the same rule name in the custom configuration file the corresponding rules of the built-in file can be overridden.

The "rulesMap" must be in the form "ScrapeSiteKey: selector". The "selector" is the jsoup CSS-like element Selector for selecting one or multiple elements in the document, with what will be done with the extracted content depending on "ScrapeSiteKey". In case of duplicate "ScrapeSiteKey", the "selector" from the last one will be in effect.

When writing the "selector", then care should be taken to not select an element (or parts of it) multiple times. For example, the selector "p" will select a `<p>` element, but also any potential sub-paragraphs `<p>` of that element, thus resulting in duplicate extracted content.

Extracted content will have the type "doi" (from *Publication types*), if the site's URL was resolved from *DOI resource*, or it will have the type attached to the link taken from *Links*. The content could be used to fill the corresponding publication part, but only if the part is requested as determined by `--fetch-part` or `--not-fetch-part` and certain other condition are met (as described at the end of *Publication types*).

The key "ScrapeSiteKey" must be one of the following:

pmid Its value is the selector string for the *pmid* of the publication. Only the first element found using the selector is extracted and any prefix in the form "pmid:", case-insensitively and ignoring whitespace, is removed from the element's text. A PMID is rarely found in journal article web pages.

pmcid Analogous to the pmid selector, except that meant for the *pmcid* of the publication.

doi Analogous to the pmid selector, except that meant for the *doi* of the publication. A DOI is quite often found in journal article web pages. Usually a DOI was used to arrive at the site, so the doi selector usually does not provide new information, but it can upgrade the type of the *doi* part (from "external" to "doi" or "link_oadoi" for example).

title Selector string for the *title* of the publication. Only the first element found using the selector is extracted. If the *title* is already *final* or the selector is empty, then no extraction is attempted (the same is also true for keywords, abstract and fulltext).

subtitle Selector string for a rarely occurring subtitle. Text from the first element found using this selector is appended to the text found using the title selector with separator " : ".

keywords Selector string for *keywords*. All elements found using the selector are extracted, each element being a separate keyword.

keywords_split Selector string for *keywords*. All elements found using the selector are extracted, but differently from the "keywords" selector above, the text in each element can be in the form "keywords: word1, word2". In that case, the potentially existing prefix "keywords:" is removed, case-insensitively and ignoring whitespace, and the following string split to separate keywords at the separator "," (and ";" and "|"). If both "keywords" and "keywords_split" are specified, then the "keywords" selector is attempted before.

abstract Selector string for *theAbstract*. All elements found using the selector are extracted and concatenated to a final string, with `\n\n` (two line breaks) separating the elements.

fulltext Selector string for *fulltext*. All elements found using the selector are extracted and concatenated to a final string, with `\n\n` (two line breaks) separating the elements. The selector must not extract the title and abstract from the site, as when putting together the *fulltext* of the publication, content extracted using the "title", "subtitle" and "abstract" selectors are used for the title and abstract that must be present in the beginning of the *fulltext* part and the "fulltext" selector is used for extracting the remaining full text from the site. Of the remaining full text, everything from introduction to conclusions should be extracted, however most following back matter and metadata, like acknowledgments, author information, author contributions and, most importantly, references, should be excluded. This is currently vaguely defined, but some content should still be included, like descriptions of supplementary materials and glossaries.

fulltext_src Sometimes, the full text of the publication is on a separate web page. So the URL of that separate page should be found out to later visit that page and extract the full text (and possibly other content) from it, using a different set of rules (mapped to by a different "regex"). In some cases, finding the URL of this separate page can be done by some simple transformations of the current URL. The transformation is done by replacing the first substring of the URL that matches the regular expression given in "fulltext_src" with the replacement string given in "fulltext_dst". If this replacement occurs and results in a new valid URL, then this URL is added to *Links* (with type equal to the current type) for later visiting.

fulltext_dst The replacement string for the URL substring matched using "fulltext_src". Must be specified if "fulltext_src" is specified (and vice versa).

fulltext_a Sometimes, the separate web page of the publication's full text can be linked to somewhere on the current page. This key enables specifying a selector string to extract those links: all elements (usually `<a>`) found using the selector are extracted and the value of their `href` attribute added to *Links* with type equal to the current type, if the value of `href` is a valid URL.

pdf_src Sometimes, the full text of the publication can be found in a PDF file. The URL of that PDF could be constructed analogously to the "fulltext_src" and "fulltext_dst" system: the first substring of the current URL that matches the regular expression given in "pdf_src" is replaced with the replacement string given in "pdf_dst" and if the result is a new valid URL, it is added to *Links*. The type (from *Publication types*) of the link will be the corresponding PDF type of the current type (e.g., type "pdf_doi" corresponds to type "doi").

pdf_dst The replacement string for the URL substring matched using "pdf_src". Must be specified if "pdf_src" is specified (and vice versa).

pdf_a Selector string to extract all full text PDF links on the current page. All elements (usually `<a>`) found using the selector are extracted and the value of their `href` attribute added to *Links*, if the value of `href` is a valid URL. The type (from *Publication types*) of the link will be the corresponding PDF type of the current type (e.g., type "pdf_doi" corresponds to type "doi"). If possible, the "pdf_a" selector should probably be preferred over "pdf_src" and "pdf_dst", as sometimes the PDF file can be missing or inaccessible and then the "pdf_a" selector will correctly fail to add any links, but "pdf_src" and "pdf_dst" will add a manually constructed, but non-existing link to *Links*.

corresp_author_names Selector string for the names of *correspAuthor*. All elements found using the selector are extracted, each name added as a separate corresponding author.

corresp_author_emails Selector string for the e-mails of *correspAuthor*. All elements found using the selector are extracted, with e-mail addresses found in `href` attributes (after the prefix `mailto:` which is removed). E-mail addresses are added to the names extracted with "corresp_author_names" (in the same order), which means the number of names must match the number of e-mail addresses – if they don't match, then names are discarded and corresponding authors are only created using the extracted e-mails.

The `-scrape-selector` command of the *CLI* can be used to test which selector string from loaded configuration files will be in effect for the supplied URL and "ScrapeSiteKey".

5.2.1.3 javascript

As mentioned in *Getting a HTML document*, either the jsoup or HtmlUnit library can be used for fetching a HTML document, with one difference being that HtmlUnit supports executing JavaScript, which jsoup does not. But as running JavaScript is very slow with HtmlUnit, then jsoup is the default and JavaScript is turned on only for sites from which content can't be extracted otherwise. This section enables the specification of such sites.

The section is made up of a list of regular expression. If the current URL has a match with any of the regexes, then HtmlUnit and JavaScript support is used for fetching the corresponding site, otherwise jsoup (without JavaScript support) is used. The current URL in this case is either the first URL resolved from *DOI resource* (there might be additional redirection while fetching the site) or the URL of a link from *Links* (again, this URL might change during fetching, so a different regex might be needed to apply scraping rules to the site at the final URL).

The `-scrape-javascript` command of the *CLI* can be used to test if JavaScript will be enabled for the supplied URL.

5.2.2 Webpages YAML

Scraping rules for *webpages/docs*. To see an example of a webpages YAML, the built-in rules file webpages.yaml can be consulted.

In contrast to the journals YAML, there is only one section in the webpages YAML.

The keys in the webpages YAML must be valid Java regular expressions. If a regex has a match with the *finalUrl* of a webpage or doc, then rules corresponding to that key are applied to extract content from the corresponding fetched document. If multiple regular expressions have a match with the URL, then the rules will be taken from the last such regex key (this enables overriding of built-in rules using the custom configuration file specified by *webpagesYaml*). If no regular expressions have a match with the URL, then scraping rules for the webpage or doc are no found and the *webpage title* will be the text value of the document's `<title>` element and *webpage content* will be the entire text parsed from the document.

Each rule under the regex key must be in the form "ScrapeWebpageKey: selector". The "selector" is the jsoup CSS-like element Selector for selecting one or multiple elements in the document, with what will be done with the extracted content depending on "ScrapeWebpageKey". In case of duplicate "ScrapeWebpageKey", the "selector" from the last one will be in effect. When writing the "selector", then care should be taken to not select an element (or parts of it) multiple times. For example, the selector "p" will select a `<p>` element, but also any potential sub-paragraphs `<p>` of that element, thus resulting in duplicate extracted content.

The key "ScrapeWebpageKey" must be one of the following:

title Its value is the selector string for the *webpage title* or doc title. Only the first element found using the selector is extracted. If the selector is empty, then the title will be empty. If the selector is missing, then the title will be the text content of the `<title>` element.

content Selector string for the *webpage content* or doc content. All elements found using the selector are extracted and concatenated to a final string, with `\n\n` (two line breaks) separating the elements. If the selector is empty, then all content of the fetched document will be discarded and the content will be empty. If the selector is missing, then the fetched document will be *automatically cleaned* and the resulting formatted text set as the content.

javascript This key enables turning on JavaScript support, similarly to the *javascript* section in the journals YAML. If its value is `true` (case-insensitively), then fetching will be done using HtmlUnit and JavaScript support is enabled, in case of any other value fetching will be done using jsoup and executing JavaScript is not supported. In contrast to other "ScrapeWebpageKey" keys, the value of this key is taken from the rule found using matching to the *startUrl* (and not the *finalUrl*) of the webpage or doc. If the javascript key is missing (and not set explicitly to `false`), then JavaScript support is not enabled, but if after fetching the document without JavaScript support there are no scraping rules corresponding to the found *finalUrl* and the entire text content of the fetched document is smaller than *webpageMinLengthJavascript* or a `<noscript>` tag is found in it, or alternatively, scraping rules are present for the found *finalUrl* and the javascript key has a `true` value in those rules, then fetching of the document will be repeated, but this time with JavaScript support. If the javascript key is explicitly set to `false`, then fetching with JavaScript support will not be done in any case.

license Selector string for *license*. Only the first element found using the selector is extracted.

language Selector string for *language*. Only the first element found using the selector is extracted.

The `-scrape-webpage` command of the *CLI* can be used to print the rules that would be used for the supplied URL.

5.3 Testing of rules

Currently, PubFetcher has no tests or any framework for testing its functionality, except for the scraping rule testing described here. Scraping rules should definitely be tested from time to time, because they depend on external factors, like publishers changing the coding of their web pages.

Tests for journals.yaml are at journals.csv and tests for webpages.yaml are at webpages.csv. If new rules are added to a YAML, then tests covering them should be added to the corresponding CSV. In addition, tests for hardcoded rules of some other resources can be found in the resources/test directory. All *Resources* except *Meta* are covered.

The test files are in a simplified CSV format. The very first line is always skipped and should contain a header explaining the columns. Empty lines, lines containing only whitespace and lines starting with `#` are also ignored. Otherwise, each line describes a test and columns are separated using “,”. Any quoting of fields is not possible and not necessary, as fields are assumed to not contain the “,” symbol. Or actually, the number of columns for a given CSV file is fixed in advance, meaning that the last field can contain the “,” symbol as its value is taken to be everything from the separating “,” to the end of the line.

One field must be the publication ID (pmid, pmcid or doi), or URL in case of webpages.csv, defining the entry to be fetched. The other fields are mostly numbers specifying the lengths and sizes that the different entry parts must have. Only comparing the sizes of contents (instead of the content itself or instead of using checksums) is rather simplistic, but easy to specify and probably enough for detecting changes in resources that need correcting. What fields (besides the ID) are present in a concrete test depend on what can be obtained from the corresponding resource.

Possible fields for publications are the following: length of publication parts *pmid*, *pmcid*, *doi*, *title*, *theAbstract* and *fulltext*; size (i.e., number of keywords) of publication parts *keywords*, *mesh*, *efo* and *go*; length of the entire *correspAuthor* string (containing all corresponding authors separated by “;”) and length of the *journalTitle*; number of *visitedSites*; value of the string *pubDate*; value of the Boolean *oa* (`1` for `true` and `0` for `false`). Every field is a number, except *pubDate* where the actual date string must be specified (e.g., `2018-08-24`). Also, in the tests, the number of *visitedSites* is not the actual number of sites visited, but the number of links that were found on the tested page and added manually to the publication by the test routine. For webpages.csv, the fields (beside the ID/URL) are the following: length of the *webpage title*, the *webpage content*, the *software license* name and length of the *programming language* name.

The progress of running tests of a CSV is logged. If all tests pass, then the very last log message will be “OK”. Otherwise, the last message will be the number of mismatches, i.e. number of times an actual value was not equal to the value in the corresponding field of a test. The concrete failed tests can be found by searching for “ERROR” level messages in the log.

Tests can be run using PubFetcher-CLI by supplying a parameter specified in the following table. In addition to the `-test` parameters there are `-print` parameters that will fetch the publication or webpage and output it to stdout in plain text and with metadata. This enables seeing the exact content that will be used for testing the entry. Publications are filled using only the specified resource (e.g., usage of the *Meta* resource is also disabled). Additionally, *visitedSites* will be filled manually by the `-test` and `-print` methods with all links found from the one specified resource (when applicable).

Parameter	Parameter args	Description
`-print-europepmc-xml`	*<pmcid>*	Fetch the publication with the given PMCID from the *Europe PMC fulltext* resource and output it to stdout
`-test-europepmc-xml`		Run all tests for the *Europe PMC fulltext* resource (from europepmc-xml.csv)
`-print-europepmc-html`	*<pmcid>*	Fetch the publication with the given PMCID from the Europe PMC HTML resource and output it to stdout
`-test-europepmc-html`		Run all tests for the Europe PMC HTML resource (from europepmc-html.csv)
`-print-pmc-xml`	*<pmcid>*	Fetch the publication with the given PMCID from the *PubMed Central* resource and output it to stdout
`-test-pmc-xml`		Run all tests for the *PubMed Central* resource (from pmc-xml.csv)
`-print-pmc-html`	*<pmcid>*	Fetch the publication with the given PMCID from the PubMed Central HTML resource and output it to stdout
`-test-pmc-html`		Run all tests for the PubMed Central HTML resource (from pmc-html.csv)
`-print-pubmed-xml`	*<pmid>*	Fetch the publication with the given PMID from the *PubMed XML* resource and output it to stdout
`-test-pubmed-xml`		Run all tests for the *PubMed XML* resource (from pubmed-xml.csv)
`-print-pubmed-html`	*<pmid>*	Fetch the publication with the given PMID from the *PubMed HTML* resource and output it to stdout
`-test-pubmed-html`		Run all tests for the *PubMed HTML* resource (from pubmed-html.csv)
`-print-europepmc`	*<pmid>*	Fetch the publication with the given PMID from the *Europe PMC* resource and output it to stdout
`-test-europepmc`		Run all tests for the *Europe PMC* resource (from europepmc.csv)
`-print-europepmc-mined`	*<pmid>*	Fetch the publication with the given PMID from the *Europe PMC mined* resource and output it to stdout
`-test-europepmc-mined`		Run all tests for the *Europe PMC mined* resource (from europepmc-mined.csv)
`-print-oadoi`	*<doi>*	Fetch the publication with the given DOI from the *Unpaywall* resource and output it to stdout
`-test-oadoi`		Run all tests for the *Unpaywall* resource (from oadoi.csv)
`-print-site`	*<url>*	Fetch the publication from the given article web page URL (which can be a DOI link) and output it to stdout. Fetching happens like described in the *DOI resource* using the built-in rules in journals.yaml and custom rules specified using *journalsYaml*.
`-test-site`		Run all tests written for the built-in rules journals.yaml (from journals.csv)
`-test-site-regex`	*<regex>*	From all tests written for the built-in rules journals.yaml (from journals.csv), run only those whose site URL has a match with the given regular expression
`-print-webpage`	*<url>*	Fetch the webpage from the given URL, using the built-in rules in webpages.yaml and custom rules specified using *webpagesYaml*, and output it to stdout
`-test-webpage`		Run all tests written for the built-in rules webpages.yaml (from webpages.csv)
`-test-webpage-regex`	*<regex>*	From all tests written for the built-in rules webpages.yaml (from webpages.csv), run only those whose URL has a match with the given regular expression

If `--fetch-part` or `--not-fetch-part` are specified then only the selected *publication parts* are filled and printed using the `-print` methods or tested using the `-test` methods. Publication fields like *correspAuthor* are always included in the printout or testing. The printing and testing operations are also affected by the *Fetching* parameters. If one of the `-test` methods is used, then the `--log` parameter should also be used to specify a log file which can later be checked for testing results.

If any larger fetching of content is planned and tests have not been run recently, then tests should be repeated (especially `-test-site` and `-test-webpage`) to find outdated rules that need updating. If testing in a different network environment then some tests might fail because of different access rights to journal content.

For testing the effect of custom selectors, the `-fetch-webpage-selector` operation can be used to specify the desired selectors on the command line. This operation ignores all rules loaded from YAML configuration files.

CHAPTER 6

Programming reference

In addition to command line usage, documented in the section *Command-line interface manual*, PubFetcher can be used as a library. This section is a short overview of the public interface of the source code that constitutes PubFetcher. Documentation in the code itself is currently sparse.

6.1 Package pubfetcher.core.common

BasicArgs is the abstract class used as base class for FetcherArgs and FetcherPrivateArgs and other command line argument classes in "org.edamontology" packages that use JCommander for command line argument parsing and Log4J2 for logging. It provides the `-h`/`--help` and `-l`/`--log` keys and functionality.

FetcherArgs and FetcherPrivateArgs are classes encapsulating the parameters described in *Fetching* and *Fetching private*. Arg and Args are used to store properties of each parameter, like the default value or description string (this comes in useful in EDAMmap, where parameters, including fetching parameters, are displayed and controllable by the user).

IllegalRequestException is a custom Java runtime exception thrown if there are problems with the user's request. The exception message can be output back to the user, for example over a web API.

Version contains the name, URL and version of the program. These are read from the project's properties file, found at the absolute resource /project.properties.

The main class of interest for a potential library user is however PubFetcher. This class contains most of the public methods making up the PubFetcher API. Currently, it is also the only class documented using Javadoc. Some of the methods (those described in *Publication IDs* and *Miscellaneous*) can be called from PubFetcher-CLI.

6.2 Package pubfetcher.core.db (and subpackages)

The Database class can be used to initialise a database file, put content to or get or remove content from the database file, get IDs contained or ask if an ID is contained in the database file or compact a database file. The class abstracts away the currently used underlying database system (MapDB). The structure of the database is described in the

Database section of the output documentation. Some methods can be called from PubFetcher-CLI, these are described in the corresponding *Database section*.

DatabaseEntry is the base class for Publication and Webpage. It contains the methods "canFetch" and "updateCounters" whose logic is explained in *Can fetch*. DatabaseEntryType specifies whether a given DatabaseEntry is a *publication*, *webpage* or *doc*.

Publication, Webpage and most other classes in the "pubfetcher.core.db" packages are the entities stored in the database. These classes contain methods to get and set the value of their fields and methods to output content fields in plain text, HTML or JSON, with or without metadata fields. Their structure is explained in *Contents*.

The PublicationIds class encapsulates publication IDs that can be stored in the database. Its structure is explained in *IDs of publications*.

The PublicationPartType enumeration of possible publication types is explained in *Publication types*.

6.3 Package pubfetcher.core.fetching

Fetcher is the main class dealing with fetching. Its logic is explained in *Fetching logic*.

Fetcher contains the public method "getDoc", which is described in *Getting a HTML document*. The "getDoc" method, but also the "getWebpage" method and the "updateCitationsCount" method can be called from PubFetcher-CLI as seen in *Print a web page* and *Update citations count*.

The Fetcher methods "initPublication" and "initWebpage" must be used to construct a Publication and Webpage. Then, the methods "getPublication" and "getWebpage" can be used to fetch the Publication and Webpage. But instead of these "init" and "get" methods, the "getPublication", "getWebpage" and "getDoc" methods of class PubFetcher should be used, when possible.

Because executing JavaScript is prone to serious bugs in the used HtmlUnit library, fetching a HTML document with JavaScript support turned on is done in a separate JavaScriptThread, that can be killed if it gets stuck.

The HtmlMeta class is explained in *Meta* and the Links class in *Links*.

Automatic *cleaning and formatting* of web pages without *scraping rules* has been implemented in the CleanWebpage class.

The "pubfetcher.core.fetching" package also contains the classes related to testing: FetcherTest and FetcherTestArgs. These are explained in *Testing of rules*.

6.4 Package pubfetcher.core.scrape

Classes in this package deal with scraping, as explained in the *Scraping rules* section.

The public methods of the Scrape class can be called from PubFetcher-CLI using the parameters shown in *Scrape rules*.

6.5 Package pubfetcher.cli

The command line interface of PubFetcher, that is PubFetcher-CLI, is implemented in package "pubfetcher.cli". Its usage is the topic of the first section *Command-line interface manual*.

The functionality of PubFetcher-CLI can be **extended** by implementing new operations in a new command line tool, where the public "run" method of the PubFetcherMethods class can then be called to pull in all the functionality of PubFetcher-CLI. One of the main reasons to do this is to implement some new way of getting publication IDs and webpage/doc URLs. These IDs and URLs can then be passed to the "run" method of PubFetcherMethods as the lists "externalPublicationIds", "externalWebpageUrls" and "externalDocUrls". One example of such functionality extension is the EDAMmap-Util tool (see its UtilMain class).

6.6 Configuration resources/log4j2.xml

The PubFetcher-CLI *Logging* configuration file log4j2.xml specifies how logging is done and how the *Log file* will look like.

CHAPTER 7

Ideas for future

Sometimes ideas are emerging. These are written down here for future reference. A written down idea is not necessarily a good idea, thus not all points here should be implemented.

7.1 Structure changes

- Make publication *fulltext* more structured (currently it is just one big string). For example, “Introduction”, “Methods”, etc could all be separate parts. Also, references could be added as a separate part (currently references are excluded altogether). It should be investigated, how feasible this is. For PDFs it is not, probably.
- Treat publication fields (like *oa* or *journalTitle*) more akin to *publication parts*. Like, *journalTitle* could also be *final* or not, etc.
- Additional metadata about a publication could be supported. Most importantly - authors (with first name, last name, orcid, affiliation, email, etc).
- *Webpages* could also have an extra field about tags, for example those that occur in standard registries or code repositories. Analogous to the *keywords* of publications.

7.2 Logic changes

- Currently, a publication is considered final when its title, abstract and fulltext are final. Keywords are not required for this, as they are often missing. But this means that, if we for some reason fail to fetch author-assigned keywords, or those keywords are added to the publication at some later date, then we will not try to fetch these keywords at some later date if the publication is already final. Note that, adding keywords to the finality requirement is probably still not a good idea.
- Currently, content found in meta tags of article web pages can be extracted by the Meta class (as described in *Meta*). However, all content extracted this way will have a non-final publication part type (see *Publication types*). As these tags are simply part of the HTML source, then for some of these tags (where we are certain of the quality of its content for the given site) the possibility to use explicit scraping rules (e.g. using a ScrapeSiteKey

called "keywords_meta") in journals.yaml could be added. This way, content extracted from these tags (using a scraping rule) can have the final publication part type of "doi".

7.3 Extra sources

- The ouput of the Europe PMC search API has `<fullTextUrlList>` and PubMed has the LinkOut service with links to resources with full text available. But it should be determined if these provide extra value, i.e. find links not found with current resources.
- DOI Content Negotiation could be used as extra source of metadata. But again, it should be determined if this would provide extra value.
- Same for Open Access Button.

7.4 Extra extraction

- Some article web pages state if the article is Open Source somewhere in the HTML (e.g., https://gsejournal.biomedcentral.com/articles/10.1186/1297-9686-44-9). So the ScrapeSiteKey "oa" could be added to extract this information using rules in journals.yaml.
- The publication field *pubDate* is currently extracted from the *Europe PMC* and *PubMed XML* resources. But it could also potentially be found at *Europe PMC fulltext* and *PubMed Central* (documentation at https://www.ncbi.nlm.nih.gov/pmc/pmcdoc/tagging-guidelines/article/tags.html#el-pubdate).
- Meta tags are currently not used to fill publication fields. E.g., the `<meta>` tag "citation_journal_title" could be used for *journalTitle*.

7.5 Database

- With a new release of PubFetcher, the structure of the database content might change (in classes of org.edammap.pubfetcher.core.db). Currently, no database migration is supported, which means that content of existing database files will be become unreadable in such case. If that content is still required, it would need to be refetched to a new database file (created with the new version of PubFetcher). So implement support for migration of database content. Maybe through JSON.
- Is the separation of functionally equivalent webpages and docs really necessary?
- If performance or reliability of MapDB should become and issue, then alternative key-value stores, like LMDB or Chronicle-Map could be investigated.

7.6 Scraping

- The current quick and dirty and uniform approach for article web page scraping could be replaced with APIs for some publishers that provide one (there's a sample list at https://libraries.mit.edu/scholarly/publishing/apis-for-scholarly-resources/).
- Reuse of scraping rules from the Zotero reference management software could be attempted, either by using the JavaScript translators directly or through the translation server.
- Currently the CSS-like jsoup selector is used for extraction. But it has its limitations and sometimes the use of XPath could be better, for example when selecting parents is required.

- There is an extension to XPath called OXPath which highlights another problem: more web pages might start to require some JavaScript interactions before any content can be obtained.
- The entire original web page should also be saved when scraping. Then, the web page would not need re-fetching if some scraping rules are changed or the actual source web page examined at some later date when debugging.
- The robots.txt should be respected.

7.7 Meta

- Currently, only scraping rules are tested. But proper unit testing (with JUnit for example) should also be implemented.
- Do comment more in code.
- Also, the whole API should be documented with Javadoc (currently only PubFetcher is covered).
- Make code more robust and secure.
- Deploy the PubFetcher library to a remote repository, possibly Maven Central, mention this in INSTALL.md.

7.8 Misc new stuff

- Configurable proxy support for network code could be added.
- The querying capabilities of PubFetcher are rather rudimentary. Investigate if it can be improved using some existing library, like JXPath or CQEngine. Maybe a change in the database system would also be required.
- Maybe an interactive shell to type PubFetcher commands in could be implemented.
- A web app and API could be implemented. Look at EDAMmap-Server as an example. If done, then the full text of non OA articles should probably not be exposed.