A Documentation

As a successor to *Open Data Portal Watch (ODPW)* [26], the Data Portal Tracker aims to semi-automatically create, validate and regularly update a comprehensive list of Open Data portals, crawl the URLs of all datasets and the associated metadata on these portals and add them to the *Open Dataset Archiver (ODArchiver)* [40] for downloading, periodic crawling and version tracking. It consists of the core *Data Portal Tracker* functionality and an adapted version of Daniil Dobriy's search engine crawling tool *Crawley* [11] and connects to the *ODArchiver's* API and MongoDB.

The subsequent documentation chapters and the extensive comments in the code aim to provide the reader with a thorough understanding of the tool's functionality and will enable them to try out, deploy, improve and extend the code. A concise version of the information presented in this section can be found in the README files in the Git repository[17] - see README.md and crawley-lite/README.md.

Even though testing has shown no major issues, we recommend performing manual sanity checks whenever results seem odd. If you want to improve this tool and need a starting point, please see the limitations and future work chapter of the thesis. Requirements for extracting metadata from the ODArchiver as a first step towards a metadata rating system can be found in section 4.4 in the main part of the thesis. Additionally, chapter A.8 below outlines the technical steps to add support for other portal software options.

Chapter A.1 shows the necessary steps to deploy the system.

Chapter A.2 covers search engine portal discovery using Crawley.

Chapter A.3 describes the list creation and validation pipeline.

Chapter A.4 presents the four crawling scripts.

Chapter A.5 explains how the crawling scripts connect to the ODArchiver.

Chapter A.6 outlines the helper functions.

Chapter A.7 gives an overview of the experiments that we carried out.

Chapter A.8 provides details on how to extend the system.

[17] https://git.ai.wu.ac.at/h1613073/data_portal_tracker

A.1 Deploying the system

Before deploying the Data Portal Tracker, first install the Python libraries listed in the requirements.txt file in the project's root directory.

To use the full functionality of the system and be able to connect to the ODArchiver's production MongoDB and use the method of the ODArchiver API that posts resources, request the .env file from the institute's system administrators and save it in the project's root directory. While the repository includes an .env_example file, it does not contain the API secret and the MongoDB connection strings.

The ODArchiver's MongoDB uses a Kubernetes cluster and is currently running on three nodes, that's why there are three production connection strings in the .env file and multiple try/except blocks in the __init__ method of the ArchiverConnector class that try out these strings. A virtual machine provided by the institute is needed to connect to these nodes, each of which has a port that was opened for this purpose.

Whenever the ArchiverConnector class, which is defined in the data_portal_tracker/archiver_connector.py script, is instantiated, for example after importing it into the data_portal_tracker/portal_crawler.(ipynb|py) script, the argument "mode" must be set to "production" instead of "local".

Checklist:

- Install Python libraries
- Save .env file to project root
- Use "[...]ai.wu.ac.at" virtual machine
- Call ArchiverConnector(mode = "production")

For brevity, some of the subsequent Python function call examples will only be shown in the short form used within a script (see listing 1), but the functions can of course also be executed on the command line (see listing 2).

```
1 crawl_ckan(portal_list, "data/portal_statistics_ckan.csv")
```

Listing 1: Function call within a script

```
1 python3 -c 'from portal_crawler import *;
  ↪  crawl_ckan(portal_list, "data/portal_statistics_ckan.csv")'
```

Listing 2: Function call on the command line

A.2 Search engine portal discovery

Path: crawley-lite/.*

The tool Crawley, which we use for **portal discovery** via search engine APIs, was provided by Daniil Dobriy [11] and subsequently adapted and simplified for our use case. Also, parts of our documentation related to the tool were taken over and edited.

When using Crawley, make sure that there are SerpAPI keys in the crawley-lite/keys.txt file. If there are none, register for a free account and add keys to the file. In order to perform a Google search for "Open Data Portal", use the command line to navigate to the crawley-lite directory and execute the crawley-lite/crawley-lite.py script:

```
python3 crawley-lite.py --query "Open Data Portal" --engine
↪ Google --count 100  --offset 0
```

Listing 3: Portal discovery: basic search engine query

For Google searches, there are up to 100 results per query - the number of results to be returned is controlled by the parameter --count, while the number of results to be skipped is controlled by the parameter --offset. In pagination terms, when showing 100 results per page, you can skip to page 2 by specifying an offset of 100, to page 3 with an offset of 200 and so on.

```
python3 crawley-lite.py --query "Open Data Portal" --engine
↪ Google --count 100  --offset 100
```

Listing 4: Portal discovery: search engine query with offset

Just like on the web interface, quotes can be used for exact matches (and must be escaped when using the same type of quotes for the whole search query) and search operators can restrict the results to certain domains or exclude domains.

```
python3 crawley-lite.py --query "site:*.opendatasoft.com \"Open
↪ Government Data\" -site:data.opendatasoft.com" --engine
↪ Google --count 100  --offset 0
```

Listing 5: Portal discovery: search engine query with search operators

Search results returned by the API are saved to JSON files in the crawley-lite/results folder. These files are later used as input files in the portal_handler script in the data_portal_tracker project directory, which extracts all of the organic result URLs and adds them to the list creation and portal validation pipeline.

Listing 6 shows the first organic search result in one of the JSON files mentioned above and gives an idea of the additional, currently unused data that is available for potential further refinement of the system in the future:


```
{
    "position": 1,
    "title": "Data.gov CKAN API - Catalog",
    "link":
    ↪ "https://catalog.data.gov/dataset/data-gov-ckan-api",
    "displayed_link": "https://catalog.data.gov > dataset >
    ↪ data-gov-ckan-api",
    "date": "Nov 10, 2020",
    "snippet": "The data.gov catalog is powered by CKAN, a
    ↪ powerful open source data platform that includes a robust
    ↪ API. Please be aware that data.gov and ...",
    "snippet_highlighted_words": [
        "powered by CKAN"
    ],
    "rich_snippet": {
        "bottom": {
            "detected_extensions": {
                "data_update_frequency_r_pm": 1,
                "metadata_created_date_november": 10
            },
            "extensions": [
                "Data Update Frequency: R/P1M",
                "Metadata Created Date: November 10, 2020"
            ]
        }
    },
    "about_this_result": {
        "keywords": [
            "powered",
            "by",
            "ckan"
        ],
```

```
        "languages": [
            "English"
        ],
        "regions": [
            "United States"
        ]
    },
    "cached_page_link":
    "https://webcache.googleusercontent.com/search?q=cache:
    L-lWAlt2ab8J:https://catalog.data.gov/dataset/
    data-gov-ckan-api&cd=10&hl=en&ct=clnk&gl=us",
    "related_pages_link":
    "https://www.google.com/search?q=related:
    https://catalog.data.gov/dataset/data-gov-ckan-api",
    "source": "data.gov"
}
```

Listing 6: Portal discovery: search result example

More details about our adapted version of Crawley can be found in the crawley-lite/README.md file. The original code is also available on GitHub[18].

[18] https://github.com/semantisch/crawley

A.3 Portal list creation and validation

Path: data_portal_tracker/portal_handler.(ipynb|py)

Our **portal handler** uses a multi-step pipeline to create a list of portals from various sources and validate the websites on the list. It allows manual additions at two different points.

Currently, files created by functions are saved to data_portal_tracker/data, but for all files whose paths are passed to functions as arguments, this can be changed. When rerunning the script's functions to create an updated list, we recommend creating a new (sub-)folder for every update and change the paths passed to the functions as arguments accordingly. In this way, no existing data is overwritten and the evolution of data portals can be analyzed.

Log files containing failure or success information that are generated during the execution of functions are stored in the data_portal_tracker/logs directory. To change this, the function definitions would have to be edited. Log files that are automatically created by functions are stored in CSV format and include the function name and a timestamp in their name - for example: crawl_socrata_2023-08-15_16_52_42_fail.csv. In addition, we manually saved printed output of functions to files that include the function name in their name and a have .log file extension - for example: validate_list.log.

A.3.1 Extract search results

Once the search engine portal discovery component has delivered some results, we can use the function extract_search_results to extract the URLs of organic search results from the saved JSON files. This function includes code for looping through the search results adapted from Crawley.

When calling the function, two arguments are required:

- search_results_folder (string): the path of the folder in the crawley-lite directory containing the search results
- output_file (string): the path of the CSV file to be exported

Here is how we called the function for the first run:

```
extract_search_results(
    search_results_folder = "../crawley-lite/results",
    output_file = "data/0_search_results.csv")
```

Listing 7: Portal handler: extract search results

A.3.2 Create a portal list

Next, the function create_list is used to create an initial list of portal URLs based on multiple sources. This function contains the first of two options for manually adding portals. All URLs added here will go through the entire deduplication and validation process without bypassing any steps. Currently, additions are made by extending the array additional_portals in the function definition, however, it might be useful to do this via a function parameter in the future (just like we already implemented in the add_api_endpoints function, the second option for manual additions).

When calling the function, two arguments are required:

- search_results_file (string): the path of the CSV input file containing search result URLs in a column "url"
- output_file (string): the path of the CSV file to be exported

Here is how we called the function for the first run:

```
create_list(
    search_results_file = "data/0_search_results.csv",
    output_file = "data/1_initial_portals.csv")
```

Listing 8: Portal handler: create list

A.3.3 Deduplicate the portal list

The function remove_duplicates deduplicates a list of URLs by removing unwanted characters, reducing each URL to its base URL, truncating the HTTP(S) protocol prefix and finally dropping all duplicates.

When calling the function, two arguments are required:

- initial_portals_file (string): the path of the CSV input file containing initial portal URLs in a column "url"
- output_file (string): the path of the CSV file to be exported

Here is how we called the function for the first run:

```
remove_duplicates(
    initial_portals_file = "data/1_initial_portals.csv",
    output_file =  "data/2_deduplicated_portals.csv")
```

Listing 9: Portal handler: remove duplicates

A.3.4 Add manually validated API endpoints

The function add_api_endpoints addresses two issues:

- Some portals have API endpoints that use a non-standard path (e.g. /catalog/api instead of /api).
- The second validation step of the validate_list function (see section A.3.6) only checks the API functionality of portals for which HTML markers were found in the first validation step.

Therefore, the function add_api_endpoints can be used to add portals which use a custom path or are known to support a certain API, but do not have any HTML markers. If they are added in this way, the first validation step of the validate_list function is bypassed and the portals will end up on the final portal list. Also note that portals added like this will be counted as "suspected" for their respective portal software in the validation statistics - thus, the number of "suspected" portals is not fully equivalent to the number of sites for which portal HTML markers were found.

When calling the function, three arguments are required:

- manual_api_additions_file (string): the path of the CSV input file containing API base URLs without "/api/..." in a column "url", e.g. "data.gv.at/katalog", and the API software name or "Unknown" in a column "manually_checked_api"
- deduplicated_portals_file (string): the path of the CSV input file containing deduplicated portal URLs in a column "url"
- output_file (string): the path of the CSV file to be exported

Here is how we called the function for the first run:

```
1 add_api_endpoints(
2     manual_api_additions_file =
      ↪ "data/manual_api_additions.csv",
3     deduplicated_portals_file =
      ↪ "data/2_deduplicated_portals.csv",
4     output_file = "data/3_extended_portals.csv")
```

Listing 10: Portal handler: add API endpoints

A.3.5 Add protocol prefixes and activity status

For each entry in the input list, the function requests the URL with HTTPS, then falls back to HTTP in case of an exception or a response code indicating failure. Information about the supported protocol and the website activity status is added to each row in the DataFrame and the enriched list is exported to a CSV file.

When calling the function, two arguments are required:

- extended_portals_file (string): the path of the CSV input file containing portal URLs in a column "url"
- output_file (string): the path of the CSV file to be exported

Here is how we called the function for the first run:

```
add_prefixes(
    extended_portals_file = "data/3_extended_portals.csv",
    output_file = "data/4_prefixed_portals.csv")
```

Listing 11: Portal handler: add prefixes

A.3.6 Validate the list

To identify portals that use one of our supported API software solutions (CKAN, Opendatasoft, Socrata), the function validate_list iterates over the input portal list, validates that the portals use a relevant catalog software and exports the validation results. Portal software validation is a two-step process: First, the function searches a portal's HTML code for specific validation markers that indicate the use of a certain software. If HTML markers are found or if a portal was previously added to the list via the function add_api_endpoints, the function checks whether the suspected API is working. This function includes code for HTML validation and related JSON exporting adapted from Crawley. The rules for the marker-based validation are located in the crawley-lite/config.json file.

Since there is always at least a small number of websites that are unavailable at a given time, we implemented an optional mode that allows you to retry the failed portals without starting from scratch. To do this, the input list/markers must be the output list/markers of the previous validation run and the retry_failed_portals argument must be set to True. The function will then rerun the validation, but will retry only the portals for which the

validation failed or the suspected API did not work previously.

When calling the function, three arguments are required and two are optional:

- input_list (string): the path of the CSV input file containing URLs - must be a file created previously by add_prefixes or validate_list - if retry_failed_portals is True, must be a file created previously by validate_list
- output_list (string): the path of the CSV output file to be exported, containing validated URLs
- output_markers (string): the path of the JSON output file to be exported, containing portals and their detected validation markers
- input_markers (string, optional): the path of the JSON input file containing portals and their detected validation markers - must be a file created previously by validate_list
- retry_failed_portals (Boolean, optional): whether or not to retry the portals for which the validation failed or the suspected API did not work in a previous run - defaults to False

Here is how we called the function for the first run:

```
validate_list(
    input_list = "data/4_prefixed_portals.csv",
    output_list = "data/5_validated_portals.csv",
    output_markers = "data/5_validated_sites.json")
```

Listing 12: Portal handler: validate list

And here is how we called the function to retry the portals that failed in the first run:

```
validate_list(
    input_list = "data/5_validated_portals.csv",
    output_list = "data/5_validated_portals_retry.csv",
    output_markers = "data/5_validated_sites_retry.json",
    input_markers = "data/5_validated_sites.json",
    retry_failed_portals = True)
```

Listing 13: Portal handler: validate list, retry failed portals

A.3.7 Analyze the validated list

Once the validation is done, the function analyze_list allows analyzing, presenting and saving the most important information about a validated portal list. Results are currently stored in the data/validation_statistics.csv file, which is hard-coded in the function definition so that the statistics of all validation runs are collected in one file.

In the validation statistics, the term "suspected" (e.g. in the column "ckan_suspected") is deliberately used instead of "markers". This is because portals added via the add_api_endpoints function will be counted as "suspected" for their respective portal software and thus, the number of "suspected" portals is not equivalent to the number of sites for which HTML markers were found.

When calling the function, one argument is required and two are optional:

- validated_portals_file (string): the path of the CSV input file containing validated portal URLs - must be a file created previously by validate_list
- show (Boolean, optional): whether or not to display the relevant DataFrames and results - defaults to True
- export (Boolean, optional): whether or not to append the results to the statistics CSV file - defaults to False

Here is how we called the function for the first run:

```
analyze_list(
    validated_portals_file =
    ↪ "data/5_validated_portals_retry.csv",
    show = True,
    export = True)
```

Listing 14: Portal handler: analyze list

A.3.8 Extract portals with working APIs

In the last step of the portal handler pipeline, the function extract_working_apis extracts the essential data from the validated portal list, keeps only the portals with working APIs, performs a final deduplication and exports the final list that will be used for portal crawling later.

When calling the function, two arguments are required:

- validated_portals_file (string): the path of the CSV input file containing validated portal URLs - must be a file created previously by validate_list
- output_file (string): the path of the CSV file to be exported, containing the final list of portal APIs

Here is how we called the function for the first run:

```
extract_working_apis(
    validated_portals_file =
    ↪  "data/5_validated_portals_retry.csv",
    output_file = "data/portals.csv")
```

Listing 15: Portal handler: extract working APIs

A.3.9 Check custom URL lists

Just like we already deployed our portal handler to check URLs submitted by colleagues, any arbitrary list of URLs can be run through and validated. If you want to combine your custom URLs with our 4 sources, edit the create_list function and add code that loads the URLs and appends them to the initial_portals DataFrame, then proceed with the other functions as usual. If you only want to check the custom URLs, save them in a CSV file that has a single column named "url" and skip the create_list function, instead pass the CSV file to the remove_duplicates function as input, then continue. When doing such a custom check, all presented rules and guidelines still apply and all functionality, for example adding manually validated endpoints that skip the first validation step, is available.

A.4 Portal crawling

Path: data_portal_tracker/portal_crawler.(ipynb|py)

Our **portal crawler** is the next major component of the Data Portal Tracker. It contains four crawling functions, each for a specific portal API software:

- **Opendatasoft API v1.0** (deprecated, only for completeness)
- **Opendatasoft API v2.1** (latest version)
- **CKAN API v2.x** (wide range, including latest version)
- **Socrata API v1.0** (only version of metadata API)

Which specific APIs the stated numbers refer to is explained in the thesis, however, please note that some of the crawling functions might partially use APIs that are different to the API to which the respective number belongs. For example, within the same portal API software, downloading datasets might sometimes be handled by a different API / service than displaying metadata.

Every function of the portal crawler takes the final portal list from the portal handler, loops through the corpus of every data portal and adds each dataset to the ODArchiver along with its metadata and a dataset/metadata mapping. In case of an exception, the last activity is retried up to three times before skipping the current loop iteration. Errors and, where applicable, also the dataset for which they occurred are saved for troubleshooting. Some crawling ideas and logic were taken from the unfinished Portal Watch API.

If you only want to count the numbers of datasets/resources per portal without crawling and indexing all datasets/resources, run the functions crawl_opendatasoft_v2, crawl_ckan and crawl_socrata after commenting out the code that handles the datasets and adds them to the Archiver:

```
1 # Calling the Archiver connector to insert data into the
  ↪ Archiver
2 archiver.handle_dataset(dataset_url, metadata_url, source_url,
  ↪ log_file_success, log_file_fail)
```

Listing 16: Portal crawler: code to comment out for counting datasets

A.4.1 Crawl Opendatasoft API v1.0

To crawl all portals on an input list that support the Opendatasoft API v1.0, you can use the function crawl_opendatasoft_v1. However, it was only created for completeness and to perform some comparisons with the Opendatasoft API v2.0 / v2.1 - Opendatasoft API v1.0 is now **deprecated** and we recommend using the function for v2.1 below!

As the Opendatasoft API v1.0 supports pagination, the function iterates over the metadata and datasets in batches of 800 until there are none left. For each set of dataset URL, metadata URL and source URL, it calls the ODArchiver connector to add the dataset, metadata and their mapping to the ODArchiver. Dataset URLs are built by assuming the availability of the CSV export format for all datasets, as explained in chapter 4.3 of the thesis. Optionally, if the portal is the Opendatasoft data hub[19], the source URL can be built differently than for all other portals by taking the URL of the original data source instead of the URL of the page that presents the dataset on the Opendatasoft data hub.

The two required arguments are the same as in all crawling functions:

- portal_list (string): the path of the CSV input file containing the final portal list - must be a file created previously by extract_working_apis in the portal handler
- statistics_file (string): the path of the CSV file to be created or extended, containing the statistics for the crawled portals

Here is how we called the function (during testing only):

```
crawl_opendatasoft_v1(
    portal_list = portal_list,
    statistics_file =
    ↪ "data/portal_statistics_opendatasoft_test.csv")
```

Listing 17: Portal crawler: crawl Opendatasoft v1

A.4.2 Crawl Opendatasoft API v2.1

All portals on an input list that support the Opendatasoft API v2.1 can be crawled with the function crawl_opendatasoft_v2. Even though API v2.0

[19] https://data.opendatasoft.com

also exists and the API responses of v2.1 differ slightly from v2.0, there is no need for a separate v2.0 function since every portal in our list that supports v2.0 also supports v2.1.

The logic of the function is similar to that of the v1.0 function, with two main differences. Firstly, since portals using API v2.1 offer a JSON metadata catalog of all datasets, the catalog is requested for each portal instead of using pagination. Secondly, the optional code from the v1.0 function is not present here, but there is another optional code section that checks the available export formats of each dataset and that was used to determine that CSV is generally available - which is why CSV is now assumed for building the dataset URL and the code is commented out. Per dataset, the function takes roughly 2 seconds, which means it can work through 30 datasets per minute or 1800 datasets per hour.

The two required arguments are the same as in all crawling functions:

- portal_list (string): the path of the CSV input file containing the final portal list - must be a file created previously by extract_working_apis in the portal handler
- statistics_file (string): the path of the CSV file to be created or extended, containing the statistics for the crawled portals

Here is how we called the function for the first run:

```
crawl_opendatasoft_v2(
    portal_list = portal_list,
    statistics_file =
     ↪ "data/portal_statistics_opendatasoft.csv")
```

Listing 18: Portal crawler: crawl Opendatasoft v2

A.4.3 Crawl CKAN API v2.x

The function crawl_ckan crawls all portals on the input list that support the CKAN API v2.x and has been tested with portals using a wide range of CKAN versions from v2.0 to v2.10.

Similar to the Opendatasoft API v1.0 function, pagination is used to loop through all datasets. However, on CKAN portals, one dataset/package can contain multiple resources. Therefore, unlike all other crawling functions, there is a nested loop for each dataset that iterates over all resources

of a dataset. For each set of resource URL, (dataset) metadata URL and (dataset) source URL, the function calls the ODArchiver connector to add the resource, metadata and their mapping to the ODArchiver.

The two required arguments are the same as in all crawling functions:

- portal_list (string): the path of the CSV input file containing the final portal list - must be a file created previously by extract_working_apis in the portal handler
- statistics_file (string): the path of the CSV file to be created or extended, containing the statistics for the crawled portals

Here is how we called the function for the first run:

```
crawl_ckan(
    portal_list = portal_list,
    statistics_file = "data/portal_statistics_ckan.csv")
```

Listing 19: Portal crawler: crawl CKAN

A.4.4 Crawl Socrata API v1.0

The function crawl_socrata crawls all portals on the input list that support the Socrata API v1.0.

Using pagination, the function loops through all datasets of each portal and checks whether the dataset type is one of two supported asset types for which our function can build dataset URLs: "dataset" or "file". If one of these types is found, the dataset and metadata are sent to the ODArchiver for indexing and mapping, otherwise the dataset is skipped. The asset types "chart", "datalens" and "filter" might work with the same method as the "dataset" type, but further testing is required to ensure that this approach is valid. We added a comment to the function that contains this information and a replacement for the current if-statement which can be used if testing shows positive results. For the type "map", future support is not likely because maps are purely front-end visualizations that use data from entities of the type "dataset", which are already supported and crawled.

The two required arguments are the same as in all crawling functions:

- portal_list (string): the path of the CSV input file containing the final portal list - must be a file created previously by extract_working_apis in the portal handler

- statistics_file (string): the path of the CSV file to be created or extended, containing the statistics for the crawled portals

Here is how we called the function for the first run:

```
crawl_socrata(
    portal_list = portal_list,
    statistics_file = "data/portal_statistics_socrata.csv")
```

Listing 20: Portal crawler: crawl Socrata

A.5 ODArchiver connection

Path: data_portal_tracker/archiver_connector.(ipynb|py)

Our ArchiverConnector contains methods which connect to the ODArchiver API (using HTTP requests) and the ODArchiver database (via MongoDB queries). In the subsections below, we describe how to interact with this class and its methods. Currently, the crawling script calls the ArchiverConnector's class constructor and all crawling functions call its handle_dataset method which then calls all other class methods.

For manual interactions with the ODArchiver's MongoDB, here is some information about the database schema:

- datasets: essential information for crawler to work - two unique identifiers, _id and id, an array of versions and the three objects meta, url and crawl_info
- datasets.files: information about the individual versions of a dataset, referenced by their IDs from the versions array of the respective datasets document
- datasets.chunks: actual data of the files, stored as chunked, Base64-encoded binaries - the files_id field references the _id field in the files collection.
- datasets.mappings: newly added as part of the Data Portal Tracker, contains the mappings of datasets and metadata - the dataset_id, metadata_id and added fields store the IDs of the dataset and metadata as well as the timestamp of the mapping creation
- hosts: necessary information for the locking mechanism and host politeness to work properly, for example the currentlyCrawled field
- sources: referenced by the source array field in the meta object of the datasets collection - one dataset can have multiple sources and one source can be referenced by multiple datasets

When performing queries in the ODArchiver MongoDB without using the methods we provided, keep in mind that some fields, for example _id in the datasets collection, use the type ObjectID rather than string.

A.5.1 Instantiate the class

ArchiverConnector is instantiated by its __init__ method which attempts to establish a connection to the MongoDB and prints information on the success or failure. By passing an argument, the database to connect to can be chosen: If "production" is passed, the three nodes of the Kubernetes cluster that the ODArchiver's MongoDB runs on are tried out one after another. For testing, a local database can be used by passing "local" and ensuring that the local connection string and database name in the .env file match a local MongoDB that has the same database structure and a collection named datasets.mappings.

One argument is required:

- mode (string): which MongoDB to connect to - must be "local" or "production"

Here are the two ways of calling the class constructor:

```
1 archiver = ArchiverConnector(mode = "production")
2 # archiver = ArchiverConnector(mode = "local")
```

Listing 21: ODArchiver connector: call class constructor

A.5.2 Get dataset information via API

The method api_get_dataset takes a dataset URL, encodes it and performs an API request to check if the dataset is already indexed by the ODArchiver. Keep in mind that all ODArchiver API methods refer to "datasets", but now also apply to metadata because of the extensions we made to the system.

One argument is required:

- dataset_url (string): the URL of the dataset

A dictionary containing four items is returned:

- request_success: whether the request was successful
- dataset_found: whether the dataset was found
- dataset_id: the ID of the dataset in the ODArchiver or None

- message: success message or failure message with details about the error

Here is how we called the method for testing:

```
archiver = ArchiverConnector(mode = "production")
dataset_url = "http://data.cookcountyil.gov/download/ikxe-tdm7"

archiver.api_get_dataset(dataset_url = dataset_url)
```

Listing 22: ODArchiver connector: get dataset information from API

A.5.3 Add dataset via API

The method api_add_dataset takes a dataset URL and a source URL and performs an API request to add the dataset to the ODArchiver. Keep in mind that all ODArchiver API methods refer to "datasets", but now also apply to metadata because of the extensions we made to the system.

Two arguments are required:

- dataset_url (string): the URL of the dataset
- source_url (string): the URL of the dataset's source

A dictionary containing three items is returned:

- request_success: whether the request was successful
- dataset_inserted: whether the dataset was inserted
- message: success message or failure message with details about the error

Here is how we called the method for testing:

```
archiver = ArchiverConnector(mode = "production")
dataset_url = "http://data.cookcountyil.gov/download/ikxe-tdm7"
source_url = "http://data.cookcountyil.gov/d/ikxe-tdm7"

archiver.api_add_dataset(
    dataset_url = dataset_url,
    source_url = source_url)
```

Listing 23: ODArchiver connector: add dataset via API

A.5.4 Get mapping via database

The method mongodb_get_mapping checks if there is an existing mapping between a dataset and its metadata in the "datasets.mappings" collection of the MongoDB.

Two arguments are required:

- dataset_id (string): the ID of the dataset in the ODArchiver
- metadata_id (string): the ID of the metadata in the ODArchiver

A dictionary containing five items is returned:

- query_success: whether the query was successful
- dataset_found: whether the dataset was found in any mapping
- metadata_found: whether the metadata was found in any mapping
- mapping_found: whether a mapping between the dataset and the metadata was found
- message: success message or failure message with details about the error

Here is how we called the method for testing:

```
archiver = ArchiverConnector(mode = "production")
dataset_id = "64db96b381165e001229e325"
metadata_id = "64db96b382d36f00137ff647"

archiver.mongodb_get_mapping(
    dataset_id = dataset_id,
    metadata_id = metadata_id)
```

Listing 24: ODArchiver connector: get mapping via database

A.5.5 Add mapping via database

The method mongodb_add_mapping adds a mapping entry for a given dataset ID and metadata ID to the "datasets.mappings" collection of the MongoDB.

Two arguments are required:

- dataset_id (string): the ID of the dataset in the ODArchiver
- metadata_id (string): the ID of the metadata in the ODArchiver

A dictionary containing three items is returned:

- inserted: whether the mapping was inserted
- mapping_id: the ID of the mapping document or None
- message: success message or failure message with details about the error

Here is how we called the method for testing:

```
archiver = ArchiverConnector(mode = "production")
dataset_id = "64db96b381165e001229e325"
metadata_id = "64db96b382d36f00137ff647"

archiver.mongodb_add_mapping(
    dataset_id = dataset_id,
    metadata_id = metadata_id)
```

Listing 25: ODArchiver connector: add mapping via database

A.5.6 Handle dataset

The method handle_dataset checks if a dataset and its metadata are both already indexed by the ODArchiver and have a mapping that describes their relation. Any missing indexing or mapping is added.

Five arguments are required:

- dataset_url (string): the URL of the dataset
- metadata_url (string): the URL of the dataset's metadata

- source_url (string): the URL of the dataset's source
- log_file_success (string): the path of a CSV file logging successfully handled datasets
- log_file_fail (string): the path of a CSV file logging datasets for which an exception occurred

A dictionary containing seven items is returned:

- success: whether the process was successfully completed
- dataset_added: whether the dataset was inserted via the API
- metadata_added: whether the metadata was inserted via the API
- mapping_added: whether a mapping between the dataset and the metadata was added via the MongoDB
- dataset_id: the ID of the dataset in the ODArchiver or None
- metadata_id: the ID of the metadata in the ODArchiver or None
- message: success message or failure message with details about the error

Here is how we called the method for testing:

```
archiver = ArchiverConnector(mode = "production")
dataset_url = "http://data.cookcountyil.gov/download/ikxe-tdm7"
metadata_url =
↪  "http://data.cookcountyil.gov/api/views/metadata/v1/ikxe-tdm7"
source_url = "http://data.cookcountyil.gov/d/ikxe-tdm7"
log_file_success = "logs/handle_dataset_TEST_success.csv"
log_file_fail = "logs/handle_dataset_TEST_fail.csv"

archiver.handle_dataset(
    dataset_url = dataset_url,
    metadata_url = metadata_url,
    source_url = source_url,
    log_file_success = log_file_success,
    log_file_fail = log_file_fail)
```

Listing 26: ODArchiver connector: handle dataset

A.6 Helper functions

Path: data_portal_tracker/helpers.py

We have created multiple functions to support URL processing tasks in the portal handler and portal crawler scripts. These utilities were designed to be reused in different components of our system, thus we collected them in a separate script only containing helpers, which enables intuitive and simple importing of the functions wherever they are needed.

A.6.1 Check website activity

The first helper function check_url requests a well-formed input URL that has a protocol prefix and returns information about the response. This function is currently called by check_protocol, another helper function.

One argument is required:

- url (string): the URL to be requested - must include a protocol prefix (http:// or https://)

A dictionary containing three items is returned:

- request_success: whether the request was successful
- response_code: the HTTP response code
- message: success message or failure message with details about the error

A.6.2 Check website protocol

Next, check_protocol takes an input URL with or without protocol prefix and finds out which protocol is working for the URL. It requests a URL with HTTPS and, if required, falls back to HTTP and finally returns the "best" working variant of the URL in this order: HTTPS, HTTP, URL without prefix. This function is currently called by the function add_prefixes in the portal handler.

One argument is required, one is optional:

- url (string): the URL for which the protocol should be checked - can be with or without HTTP(S) prefix

- show_details (Boolean, optional): whether or not to print details about the requests - defaults to True

One string is returned:

- str: the working URL with protocol prefix (HTTPS > HTTP) or non-working URL without protocol prefix

A.6.3 Remove double slashes

Finally, remove_double_slashes removes redundant forward slashes by replacing a double forward slash with a single forward slash in any part of a HTTP(S) URL except for the protocol prefix. In the context of the portal_crawler script, only apply this function to the substring of a URL related to the API, so before adding a dataset-specific ID or something similar! The function is currently called in all crawling functions of the portal crawler.

One argument is required:

- url (string): the URL to be modified which may contain double forward slashes

One string is returned:

- str: the modified URL with only single forward slashes

A.7 Experiments

Path: data_portal_tracker/experiments.ipynb

In this section, we will briefly describe experiments that we carried out to justify implementation decisions, to use our portal validation on other data and to prepare future work. More details, the code and all results can be found in the data_portal_tracker/experiments.ipynb notebook.

A.7.1 Validating "www." URLs with and without "www."

This experiment supported the decision-making process regarding the deduplication of URLs in the portal handler. Since a minority of URLs were appearing in the list twice, once with and once without the "www." prefix, the question was whether this prefix could be removed in the early deduplication step in the portal handler or if this would cause problems, e.g. a large number of sites not responding to requests anymore.

Based on the results of the experiment, which showed that 3 CKAN portals, 10 Opendatasoft portals and 4 Socrata portals would be lost when removing "www." early on, we have decided not to remove the "www." prefix in the early deduplication step. Taking into account the small share of the described duplicates among all URLs, the benefits derived from this deduplication (reducing the number of HTTP requests during validation, avoiding any duplicate sites in the list) are not outweighing the disadvantages ("losing" 10-20 Open Data portals with working APIs), especially since one main goal is to collect as many Open Data portals as possible and there are some interesting portals in the list of portals broken by the prefix removal, like the Open Data portals of Bahrain, Wallonia, Corsica and the City of Dallas.

Instead, any remaining duplicates that appear with and without "www." are removed from the list of portals with working APIs near the end of the pipeline, after the validation step. See the portal handler for details.

An alternative solution would have been to request every URL with and without the "www." prefix, similar to the already implemented function that performs an HTTPS request and, if necessary, an HTTP request to determine the best available protocol. However, this would have lead to multiple additional requests for many of the approximately 5000 sites as multiple combinations of HTTP or HTTPS and WWW or no WWW would have had to be tried out and would have vastly exceeded the reduction in requests from removing just over 100 duplicates.

A.7.2 Validating Opendatasoft file export formats

This experiment, which informed our decision on always assuming the availability of CSV, was already described in detail in section 4.3 of the thesis.

A.7.3 Validating railway and university portals

We deployed our portal handler to validate different Open Data portals of railway companies and universities at the request of interested colleagues.

Our validation of the given railway portals showed 2 working CKAN portals and 3 working Opendatasoft portals which are now also included in our main portal list. The portal of Deutsche Bahn (DB) is supposed to be based on CKAN, but the endpoints do not work, while the portal of the Austrian Federal Railways (ÖBB) is not using CKAN, Opendatasoft or Socrata. For Prorail, there is a portal that is based on ArcGIS and thus out of scope currently, but there is also some Prorail data in the Dutch government's Open Data portal[20] which has a working CKAN API but contains much more than just railway data.

Only two of the given educational organizations have a working API that is based on one of the portal software options we support (CKAN, Opendatasoft, Socrata). Of those two, one is the US Department of Education, the other is California State University and both of them use CKAN.

A.7.4 Validating the ODPW list

This experiment, in which we validated the portals on the ODPW list, provided new data for our analysis of the list's evolution from 2016 to 2023. The full methodology can be found in section 4.4 of the thesis and the results are presented in section 5.2.

A.7.5 Checking false positives of marker validation

On some sites, validation markers can be found in the first part of the validation step, but no working API is located subsequently. Given the chosen approach of only testing the API functionality on sites for which the validation marker search has been successful, these cases, which could be described as false positives, are worth investigating. We wrote some very short and simple code to display such cases that can be used as a starting point for future work in this area and is also located in the experiments notebook.

[20] https://data.overheid.nl/data

A.7.6 Checking DCAT extension on CKAN portals

CKAN offers an extension that enables the retrieval of metadata using the Data Catalog Vocabulary (DCAT). The code provided in the experiments notebook shows how to check the availability of this extension as well as the TTL / RDF catalog for all CKAN portals on the list. More details can be found in section 4.4 of the thesis and the results are shown in section 5.2.

A.8 Extending the system

Since users of the Data Portal Tracker may want to add support for further portal software other than CKAN, Opendatasoft and Socrata, this section details the required steps to make the relevant modifications.

1. (Optional:) Firstly, when including a different portal solution, it may make sense to consider this change already in the very first part of the pipeline, the search engine portal discovery. Searches are currently being carried out via the Crawley command line tool, see crawley-lite/README.md and the relevant sections above in the thesis and documentation. The search terms used are manually entered - some suggestions and previously used examples can be found in the values of the "search" keys in crawley-lite/config.json. To improve the end results and get more portals with a validated, working API, search terms should include relevant keywords like "Open Data Portal + [Name of API software]". However, the current list of potential portals is already very long (more than 5500 websites), so that most large portals on the web should already be included, they just couldn't be validated successfully yet as they do not use CKAN, Opendatasoft or Socrata. Therefore, this step is optional.

2. To adapt the first validation step, edit the crawley-lite/config.json file and add a top-level JSON key with the name of the software (this will be used in multiple locations throughout Data Portal Tracker, thus being consistent is recommended) whose value is an object containing the keys "search" and "validate", each with an array of strings as their value. Identify validation markers (HTML elements that can be frequently found in the HTML code of portals using the relevant software) by using the developer tools of any web browser and searching for keywords like the name or an abbreviation of the software. These markers must go in the array belonging to "validate", while "search" does not need any values. Listing 27 below shows an extract of config.json with these mentioned additions at the end. Note: values for the three "search" keys were omitted in the listing because they are currently not used by any script.

```
{
  "CKAN": {
    "search": [
    ],
    "validate": [
```

```
      "img/ckan.svg",
      "<a href=\"http://docs.ckan.org/en/2.8/api/\">CKAN API</a>",
      "<a href=\"http://docs.ckan.org/en/2.9/api/\">CKAN API</a>",
      ">CKAN API</a>",
      "<form id=\"ckan-dataset-search\"",
      "<a id=\"ckan_de\"",
      "<a id=\"ckan_en\"",
      "header_od_ckan.en",
      "/ckan/organization",
      "ckan-footer-logo",
      "wpckan_dataset",
      "id='ckan_base-js'",
      "<meta name=\"generator\" content=\"ckan",
      "ckan.ico",
      "href=\"/ckan/dataset",
      "<span>Powered by</span>",
      "alt=\"CKAN\"",
      "alt=\"CKAN logo\"",
      "href=\"/fanstatic/ckanext-harvest",
      "href=\"/data/fanstatic/ckanext-scheming",
      "href=\"/ckan/fanstatic/ckanext-geoview",
      "href=\"/ckan/fanstatic/ckanext-harvest"
    ]
  },
  "OpenDataSoft": {
    "search": [
    ],
    "validate": [
      "BRAND_HOSTNAME: \"opendatasoft.com\"",
      "ods.core.config",
      "ods.minimal",
      "ods.core.config",
      "ods.core",
      "ods.core.form.directives"
    ]
  },
  "Socrata": {
    "search": [
    ],
```

```
    "validate": [
      "<!-- Start of socrata Zendesk Widget script -->",
      "var socrata",
      "window.socrata",
      "<!-- \n            Powered by Socrata",
      "<!-- Powered by Socrata",
      "<!--Powered by Socrata",
      "<!--\n  Powered by Socrata",
      "www.socrata.com\n  -->"
    ]
  },
  "Name of new API software": {
    "search": [
      "No need to add anything here - 'search' values are
↪   currently unused!"
    ],
    "validate": [
      "Some HTML element",
      "Another HTML element",
      "A third HTML element"
    ]
  }
}
```

Listing 27: How to extend the crawley-lite/config.json file

3. In the second validation step, add code that validates the functionality of the new software's API to the validate_list function of the data_portal_tracker/portal_handler.(ipynb|py) script. The relevant section of the function is shown in listing 28 below (code shortened, see the script for all details). On the same indentation level as the three comments "# Checking portals with (CKAN|Socrata|Opendatasoft) markers" and the subsequent code blocks started by if-statements matching the "suspected_api" field of the prefixed_portals DataFrame, add another comment and if-statement for the new software. Research the new API to find a method that returns general information about the API like the supported methods, any extensions, the status, the current API version and any older supported versions. Save the version information to the "api_version" field and the information whether the API could be validated and is working (as determined by the request giving the expected result) to the "api_working" field of the prefixed_portals

DataFrame. If there is no suitable API method, perform any general request to the API (for example a "help" method) and, using a try/except block, try to access a known field of the response - if there is no error, mark the API as working, otherwise as not working. The exact code will vary depending on the API and will require some experimentation.

```
# Verifying that the detected API is available and working
if prefixed_portals.loc[index, "suspected_api"] is not None and
 ↪  prefixed_portals.loc[index, "suspected_api"] != "Unknown":

    # Checking portals with CKAN markers
    if prefixed_portals.loc[index, "suspected_api"] == "CKAN":
        # Resetting version variable
        ckan_version = None

        try:
            api_url = base_url + "/api/3/action/package_search"
            response = requests.get(api_url, timeout = 15)
            if json.loads(response.text)["success"] == True:
                print("CKAN API working")
                prefixed_portals.loc[index, "api_working"] =
                 ↪  True
                # Checking the API version
                try:
                    api_version_url = base_url +
                     ↪  "/api/3/action/status_show"
                    response = requests.get(api_version_url,
                     ↪  timeout = 15)
                    ckan_version =
                     ↪  json.loads(response.text)["result"]["ckan_version"]
                    prefixed_portals.loc[index, "api_version"]
                     ↪  = ckan_version
                except Exception as e:
                    prefixed_portals.loc[index, "api_version"]
                     ↪  = "Unknown"
                    log(e)
        except Exception as e:
            print("CKAN API not working")
            prefixed_portals.loc[index, "api_working"] = False
            log(e)
```

```
28
29      # Checking portals with Socrata markers
30      elif prefixed_portals.loc[index, "suspected_api"] ==
        ↪  "Socrata":
31          [...]
32
33      # Checking portals with Opendatasoft markers
34      elif prefixed_portals.loc[index, "suspected_api"] ==
        ↪  "OpenDataSoft":
35          [...]
36
37      # Checking portals with [Name of new software] markers
38      elif prefixed_portals.loc[index, "suspected_api"] == "[Name
        ↪  of new software]":
39          [...]
```

Listing 28: How to extend the validate_list function

4. For the analysis step in which statistics about each validated portal list are displayed and saved, edit and extend the function analyze_list of the data_portal_tracker/portal_handler.(ipynb|py) script. Add two new columns to the code creating the "statistics" DataFrame at the beginning: one for the portals suspected to be using the new software (based on the results of marker-based validation) and one for the portals confirmed to be using the new software (based on the results of the API validation). For both of these subsets of portals, add code that counts, displays and saves them. As a reference and template, take the code for CKAN (shown in listing 29 below) or Opendatasoft or Socrata (omitted below), just with "CKAN/Opendatasoft/Socrata" replaced by the name of the new software in every string and variable name.

```
1  # Creating a DataFrame for the statistics
2  statistics = pd.DataFrame(columns = ["file", "total", "active",
   ↪  "inactive", "validated", "unvalidated",
   ↪  "subpage_endpoints", "no_markers", "ckan_suspected",
   ↪  "ckan_working", "opendatasoft_suspected",
   ↪  "opendatasoft_working", "socrata_suspected",
   ↪  "socrata_working", "name_of_new_software_suspected",
   ↪  "name_of_new_software_working", "timestamp"])
3  [...]
```

```
4
5  # Suspected CKAN portals (validation markers found or API
   ↪  manually checked)
6  ckan_markers_portals =
   ↪  validated_portals[validated_portals["suspected_api"] ==
   ↪  "CKAN"]
7  ckan_markers_portals.columns.name = "Suspected CKAN portals"
8  statistics.loc[0, "ckan_suspected"] = len(ckan_markers_portals)
9  if show is True:
10     display(ckan_markers_portals)
11
12 # Portals with working CKAN API
13 ckan_working_api_portals =
   ↪  ckan_markers_portals[ckan_markers_portals["api_working"] ==
   ↪  True]
14 ckan_working_api_portals.columns.name = "Portals with working
   ↪  CKAN API"
15 statistics.loc[0, "ckan_working"] =
   ↪  len(ckan_working_api_portals)
16 if show is True:
17     display(ckan_working_api_portals)
18
19 # Suspected Opendatasoft portals (validation markers found or
   ↪  API manually checked)
20 [...]
21
22 # Portals with working Opendatasoft API
23 [...]
24
25 # Suspected Socrata portals (validation markers found or API
   ↪  manually checked)
26 [...]
27
28 # Portals with working Socrata API
29 [...]
30
31 # Suspected [Name of new software] portals (validation markers
   ↪  found or API manually checked)
32 [...]
33
34 # Portals with working [Name of new software] API
```

```
35 [...]
```

Listing 29: How to extend the analyze_list function

5. Extend the data_portal_tracker/portal_crawler.(ipynb|py) script by adding a function that crawls all dataset URLs, metadata URLs and source URLs of portals based on the new software and model it after the existing functions. This is likely the most time-intensive step, as the API structure may deviate from those of the CKAN, Opendatasoft and Socrata APIs. Try to find out as much as possible about the new API and take the most suitable of the four finished functions as a template:

   - If datasets contain resources and the metadata API supports or requires pagination: crawl_ckan function.
   - If datasets are resources and the metadata API supports or requires pagination: crawl_socrata or crawl_opendatasoft_v1 function.
   - If datasets are resources and the metadata API offers a JSON metadata catalog of all datasets: crawl_opendatasoft_v2 function.
   - In any other case, a combination of the relevant code sections might be helpful.

   Then, when adapting the chosen code for the new platform, it is advisable to first comment out most lines except for the very start and work your way forward line by line. In any case, when modifying the code and possibly altering loops or removing sleep calls, definitely initially comment out any HTTP requests and calls of the handle_dataset function imported from data_portal_tracker/archiver_connector.py to prevent accidental denial-of-service attacks on WU's or external servers - see listing 30.

```
1 response = json.loads(requests.get(api_request_url).text)
2 [...]
3             archiver.handle_dataset(resource_url, metadata_url,
              ↪ source_url, log_file_success, log_file_fail)
4             [...]
```

Listing 30: Code to comment out when testing an adapted crawling script