Harvester tasks

This document tries to explain the mechanism behind tasks executed by the harvester after the “seeding” (initial load of data from an external source) has completed. I’ll use pdf_preview and check_set_integrity as examples. At the end there is a table with all tasks currently in use by Edusources and/or Publinova, together with some characteristics to understand them better at a glance. Apart from documenting the harvester I hope this gives a clear idea what functionality should be migrated or re-used in new systems.

Dispatch loop

After seeding the dispatch_document_tasks Celery task gets executed for every Document. These Documents can either be a ProductDocument (more generic name for a MaterialDocument to support Publinova) or FileDocument. In the not so far future we expect ProjectDocument and PersonDocument as well, that all follow the same logic as described below.

The dispatch_document_tasks calls get_pending_tasks which returns a list of Celery tasks that should be executed. The dispatch_document_tasks dispatches all these tasks to the queue as well as itself. After tasks complete it’s possible that new tasks get returned by get_pending_tasks, because after some initial processing the Document has new tasks waiting for execution based on the earlier processing. These new tasks need to get dispatched so the subsequent calls to dispatch_document_tasks will dispatch new tasks as returned by get_pending_tasks until it returns an empty list. A similar process occurs with dispatch_set_tasks, which dispatches tasks on the Set level. A Set is a collection of Documents that come from the same source and its tasks only run after all member Document tasks are completed. With dispatch_dataset_version_tasks tasks run that span the entire dataset, but only after all related Set tasks have completed. Note that a task may result in success, failure or some fallback behaviour, but the result is never “nothing” to prevent endless dispatch loops or forever waits of certain tasks.

pdf_preview a Resource based Document task (using HttpPipelineProcessor)

A very common pattern inside the Document task execution is that some kind of webservice, shell command or library function gets called with data coming from Document, which generates some output, that should become part of the Document. This enrichment of data is delegated to the ResourcePipelineProcessor which receives the following input:

- A Resource class that understands how to extract the data from a webservice, shell command or library call
- The args and kwargs input that should be used when calling the Resource class. This input is formatted like a “$.json.path“ string, which points to data inside the Documents.
- A specification of what output data is needed and where it should be stored.

To make this more concrete I’ll explain in detail how pdf_preview works. There is a PdfThumbnailResource which downloads a PDF located at a given URL and stores it inside the (Postgres) database. This storage is convenient when debugging why certain PDF’s have failed thumbnail generation as well as for caching to prevent creating tens of thousands of PDF thumbnails everytime we “reset” the harvester for whatever reason. After downloading and storing the full PDF the Resource will use the pdf2image library to create an image of the first page. Then it will create thumbnails of that image and store these on S3 (using Django’s file storage abstraction). The output of the Resource is some JSON with the (S3) file paths to both the first page image as well as its thumbnails.

In the case of pdf_preview the HttpPipelineResource is used, because we’re downloading a file from the web. This processor will receive an iterator with all Documents it needs to process as well as a JSON path pointing to where the PDF URL is located in the Document data ($.url in this case). The HttpPipelineResource splits the iterated Documents into batches (to minimize queue dispatch overhead) and creates one ProcessResult for each document. The ProcessResult model is a Django model able to store schemaless data as well as a Document foreignkey of in this case a FileDocument. The HttpPipelineResource then executes the PdfThumbnailResource logic using a specific URL from the Document being processed and stores the output on the related ProcessResult. After that it merges the data from ProcessResult onto the Document object in such a way that multiple processes editing the same Document will not overwrite each others data.

Now it might also be interesting to hear what pdf_preview isn’t doing to better understand the system as a whole as well as some complexity in the ResourcePipelineProcessor class family.

- Because we’re not invoking a shell command we’re not using ShellPipelineProcessor, which uses subprocess under the hood to execute shell commands based on Document input.
- Because we’re using the (relatively new) dispatch loop described above, which already batches Documents for us. We’re using asynchronous=False for the processor and instead of iterating over tens of thousands of Documents we pass in the small batch of Documents that the pdf_preview task is given. We also set the batch_size to the length of all given Documents, which results in a single batch. Passing asynchronous=False means that the “process” step (call to PdfThumbnailResouce) and “merge” step (write results to Document) happen synchronously without using the dispatch queue to split up these processing steps. Pushing “process” and “merge” to the queue separately is how the old harvester used to achieve some level of parallelism.
- All process results are written to a relevant key in the Document.derivatives field. This happens by convention and isn’t enforced anywhere, but tasks shouldn’t write data to other schemaless fields generally. However it does sometimes write to specific Document columns, which allows for better filtering in the Django admin out-of-box.
- Batch and ProcessResult models are shared for tasks within the same Django app (like products and files). Batch is probably redundant with the new harvester, because there is always one batch per task invocation, but because the new dispatch loop is itself asynchronous the lock mechanism that ProcessResult provides is still useful and still kicks in occasionally (as seen through Sentry warnings).

check_set_integrity a state management task

A special kind of task is executed at the Set level (when all new Documents for a source have been added and all Document tasks are completed). Not all sources are as reliable as we hope them to be and some sources need a complete download every night, because their endpoints don’t support a delta mechanism. This causes a problem. If a source fails (sometimes with a 200 status code), but it didn’t give all the data, then we run the risk of overwriting yesterdays data with the faulty data of today. This would result in hundreds or thousands of missing products/materials. To prevent this we check the integrity of the incoming data by comparing it to yesterday. If the quantity of the data is significantly lower than we disregard the entire incoming Set and use the Set of yesterday instead. To simplify the code as much as possible the check_set_integrity provides a data fallback and then continues the harvester process as normal with the Set fallback in place. So subsequent tasks and steps will execute using the old data even though the same processing happened yesterday.

Another variant of using state inside the harvester process is that the harvester stores a hash of the incoming data and compares new incoming data by comparing previously created hashes. If the hash of data changes we know that data has updated even though the source often doesn’t indicate which data has or hasn’t changed. This makes it possible to skip certain tasks where otherwise we would have to assume that the incoming data needs processing. We can also output less data to our consumers, because we can withhold data if the consumer indicates it wants “data changes since

X”. If the consumers would connect to the sources directly it wouldn’t be able to filter in this way in many cases. Of course it would be better if the sources would provide a decent delta mechanism and use standard error codes, but beggers can’t be choosers.

See next page for a full task overview in table format.

Tasks overview

Above I’ve given some examples of how tasks are executed and what their function is. Here I want to give an overview of all tasks together with some information to indicate what kind of task it is. I’ll be giving the following information:

- The task name
- A short description of the task
- For which entity the task executes (like products, files, projects or persons)
- At which level the task operates (document, set or dataset_version level)
- Which Resource the task uses to extract data from a HTTP, shell or Python library source
- Which Document.property changes will trigger a re-run of the task
- Which tasks or boolean logic a task depends on to check if it should run

Task	**Description**	**Entity**	**Level**	**Resource**	**Data dependency**	**Task dependency**
set_current_dataset_version	Promotes the new dataset version to the latest dataset version. Relevant for API output that builds Edusources’s sitemap and Publinova’s data copy. The API’s always returns “current” dataset versions which happens after all tasks completed.	All	Dataset version	N.A.	N.A.	All Set tasks
create_opensearch_index	Creates a OpenSearch index with a field mapping coming from the search-client package. That package is a way for the Edusources team to contribute to the indices schema as well as the predefined search queries.	All	Dataset version	N.A.	N.A.	All Set tasks
check_set_integrity	Checks integrity of incoming data and stubs further processing with historic data as a fallback, when the check fails.	All	Set	N.A.	N.A.	All Document tasks from a specific source
check_url	Checks whether the URL from a source is actually a URL and not some string. It also checks if the URL redirects somehow and returns a 200 eventually. Checking the URL is important before giving it to other services that often	File	Document	CheckURL Resource	$.hash	No

	crash on invalid URL’s (like Tika).					
tika	Passes the URL to a Tika instance with the enableUnsecureFeatures configuration set to true. This returns the text from the file at that URL. Note that we’re considering to use the XML mode of Tika, because it allows for better text filtering.	File	Document	HttpTika Resource	$.hash	check_url, valid access policy or copyright
pdf_preview	Creates a preview and thumbnails for a PDF file	File	Document	PDFThumbnail Resource	$.hash	check_url, valid access policy or copyright, only is_pdf Documents
image_preview	Creates thumbnails for an image file	File	Document	Image Thumbnail Resource	$.hash	check_url, valid access policy or copyright, only is_image Documents
video_preview	Creates a single thumbnail for Youtube or Vimeo videos through the youtubedl command line tool	File	Document	Youtube Thumbnail Resource	Nothing explicit, but URL changes will trigger the task	Nothing explicit, mostly to prevent Youtube ratelimiting. We need to migrate to using Youtube API, but GDPR concerns block us.
youtube_api	Fetches length and embed URL for a video from Youtube API	File	Document	YoutubeAPI Resource	$.hash	YouTube videos only
lookup_study_vocabulary_parents	Edusources only. Edurep and Sharekit return partial information about study vocabularies. We use preloaded Eduterm data to complement Sharekit and Edurep and store it on the Document	Product	Document	N.A.	$.learning_material.study_vocabulary	Relevant Eduterm data needs to be present in Postgres
normalize_disciplines	Uses data coming from a previously completed harvest to categorize	Product	Document	N.A.	$.learning_material.disciplines	Relevant metadata needs to be present

	“studies” information into “discipline” information. Only relevant for Edurep, but runs on all Documents					in Postgres
normalize_publisher_year	Uses data coming from a previously completed harvest to categorize “publisher year” information into X recent years and a “older” category	Product	Document	N.A.	$.publisher_year	Relevant metadata needs to be present in Postgres