HARVESTER TECHNICAL DESIGN

This document tries to explain some of the design choices that were made when creating v1.39.x of the harvester. Also known as the “New Harvester”. Please note that at the end of 2023 production harvesting is still performed by the “Legacy Harvester” and the principles described in this document are in the code somewhat compromised due to backward compatability reasons or because we just haven’t come around to fully implement them. All try to flag where caveats exist between v1.39.47 and what is described here. There is also an issue that documents which code is deprecated and could be removed in future and this document won’t digress to much into which code should be removed when we stop using the legacy harvester.

Modules / Django apps

One important difference between the legacy harvester and the new harvester is how we use modules. Django comes with its own modules system called Django apps. This modules system is used by Django to store configuration, but more importantly provides a namespace for Django models, the principal storage for data within Django. This means that with Django apps it’s possible to have diffent versions of identical models, but who store their data within their own namespace (called app labels) and are therefor separated. Through inheritence the logic for these models can differ and therefor operating on the data can differ.

There are four types of modules that we’ll shortly describe here and further explain below:

- Package modules like core. Core holds abstract models that don’t store data themselves as well as other generic code used across modules. This module could become a standalone package in the future.
- Entity modules like products and files. They use models from core to store data and leverages other classes from core as well as Datagrowth to retrieve that data.
- Implementation modules like sources. Sources holds models that specify which sources the harvester should harvest from. The app also holds shared logic about how to connect to external systems. Each entity module has its own sources Python package that hold source and entity specific implementation details. These packages import shared logic from the sources app.
- Output modules like search and testing. These apps use all the modules above to manage the search engine (which holds all data in one place) or for testing (which often transcends module boundaries in some way). The coupling between search and other modules is fairly loose due to Django’s abstraction of modules/apps. Search uses a protocol and expects all modules it interacts with to respect that protocol. For simplicity the testing app is more tightly coupled.

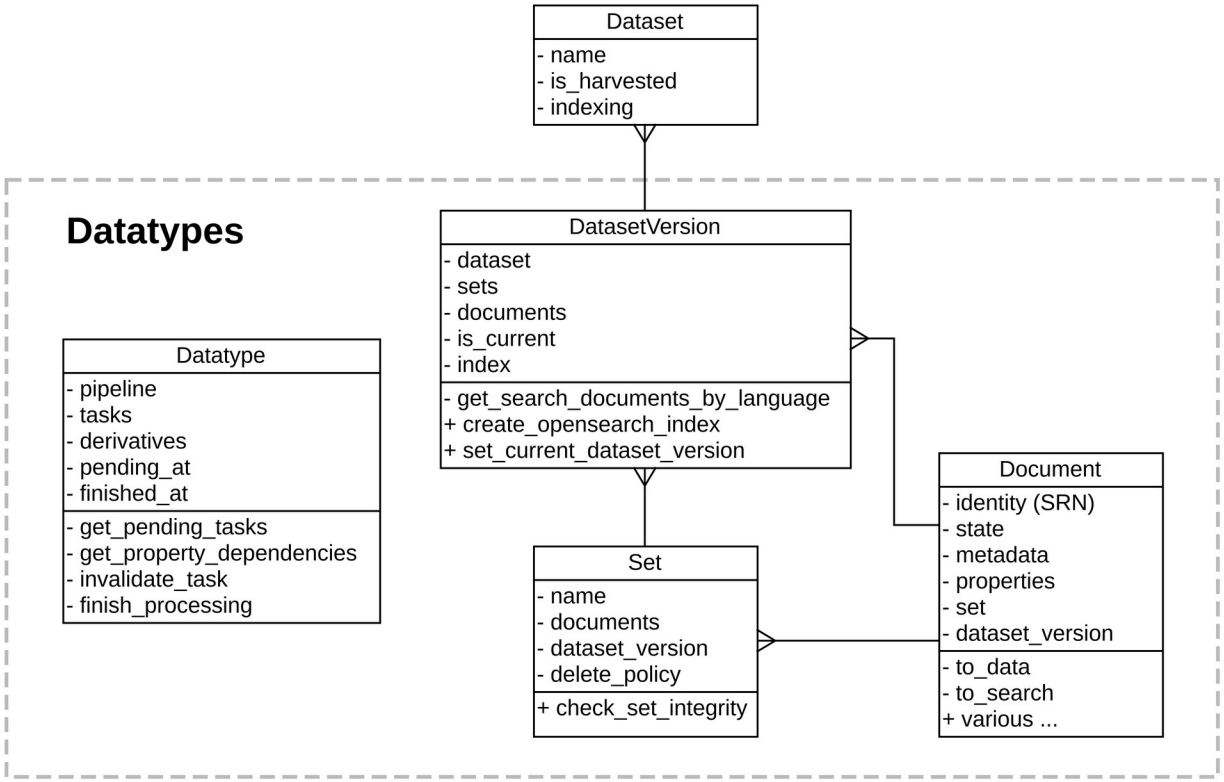
Core [Publinova, Edusources, Github]

The core app is a “package module” and is most likely to become a separate package if other projects require similar functionality. It sits at the top of the dependency tree and it shouldn’t import from other modules. Its models are mostly abstract which means it doesn’t add database tables on its own, but apps that use it will add the tables definined in this module. We’ll go over some of the components inside the core module. Beware that this module is most contaminated with legacy harvester code!

Datatypes

Datatypes define how data should be storaged. They are abstract Django models, which means that the tables that datatypes define are created in other modules. There are for instance “product” datatypes in the products module, which are stored completely separated from the “file” datatypes in the files module.

This is the generic data schema for datatypes:



Datatype is an abstract base class that is inherited by Document, Set and DatasetVersion. It adds functionality for administrating background tasks. The tasks are defined in the tasks field. A definition specifies:

1. which function should run in the background under which conditions
2. which tasks should run first
3. which data the function will use (meaning that if the underlying data changes the task will re-run)
4. which types of services the function will call (used to manage caching for these services)

Results from background tasks gets stored in the derivatives field. Information about success, failure and Resources used by the background task are stored in the pipeline field. Other fields and methods are listed for integrality, but aren’t essential to understand functioning of the system.

Document is the most important datatype, which gets most customized by the inheriting module. In general the Document will store information coming from an external source in the field properties. It takes on an identity which is a Surf Resource Name (SRN). A unique name within the SURF namespace. A Document can have the following states:

- *active* – the default
- *deleted* – source marks document as deleted
- *inactive* – system marks document as invalid

- *skipped* – system marks document as a test document that is not to be included on production

Apart from these Django model fields the Document has a number of methods that are part of the harvester protocol:

- to_data – combines properties and derivates data into a single output dictionary
- to_search – same as to_data, but will yield data that can be piped to OpenSearch directly

The most interesting part of the Document are its background tasks. In the diagram these are labeled as “various …”. A FileDocument will have very different background tasks than a ProductDocument and we’ll touch on these tasks when discussing their modules. Document tasks will run as soon as the Document is created during harvesting or when a webhook is invoked that updates the data.

Set is a collection of Documents. It is not a mathematical set, but refers to how the old outdated OAI-PMH protocol specifies sets. Within that protocol a source can have multiple sets to group its content and harvesters can choose which sets they want to harvest. The delete_policy is also definined in OAI-PMH.

Currently sets have only one background task to execute, which is the check_set_integrity task. This task will compare the Documents within Set to historic data if available. If too many Documents have been deleted the Set will be rejected and old Documents are used until the harvester is run with the “reset” flag. check_set_integrity tasks will do nothing when resetting the data. The check_set_integrity is useful if something goes wrong and Sharekit for example accidently deletes a lot of publications. In that case these changes won’t propagate to the Edusources sitemap, which Google uses to see which learning materials are available on Edusources. Likewise a deformation of metadata will not accidently remove a lot of data from SURF’s search engine. Due to their nature the set background tasks only run after all Document tasks have completed, because only then can we determine the definite integrity of the data.

DatasetVersion is a collection of Sets. Every source can have multiple sets and a complete Dataset will consist of data from multiple sources. So one DatasetVersion will manage a bunch of Sets.

There is only one truly important background task for the DatasetVersion which is set_current_dataset_version. The is_current field on DatasetVersion indicates which DatasetVersion is the current and accurate version of data for the outside world. When Edusources generates its Sitemap or when the search module generates a search index it will look for the one and only is_current DatasetVersion. This prevents accidents where Sitemaps or search indices get exposed to the outside world where a lot of data is missing or inacurate.

Another background task is create_opensearch_index, which will quite literally only create an index if indexing within the module is enabled (currently only for products). Filling the index is a responsibility of the search module, but creating the index here will signal to the search module that work needs to be done.

Other generic models

There are also other abstract models inside the core module that other apps use for their own concrete models. I won’t go into too much depth here, but this list gives an idea:

- *Dataset* – determines what should be done with the current DatasetVersion. For instance whether indexing is necessary. When the schema of Documents within a Dataset changes from version to version the idea is to start a new Dataset instead.
- *HarvestState* – tracks whether a harvest is in progress and acts as a lock to prevent multiple harvests that may race each other.
- *Batch* and *ProcessResult* – together act as a lock on Document groups to prevent race conditions as background tasks complete.

Seeding processors

Documents are the data structures where the system stores data at rest. Seeding is the process of gathering this data in-memory before committing to the database. In most cases the seeding process is as simple as calling an API endpoint and extract the relevant data. However there are a few (futuristic) scenario's where the data gathering is more involved and a special HttpSeedingProcessor class is respinsible for handling these cases:

- SIA projects can only be listed with their ids. Another API call is needed to fetch the details that actually contains the data we want to store in Document.properties
- Pure persons never expose e-mail addresses, only Pure users have these. To get the e-mail addresses from Pure authors for Publinova login system we need to fetch email addresses separately from information like name and identifiers.
- Files don't usually have a separate endpoint, but are nested within publications/records endpoints. Files also don't have a "active" or "deleted" state. To propagate a delete signal for files properly the HttpSeedingProcessor has a special back fill option where it tries to fill out the gaps that the endpoints leave behind with historic data.

The alternative to a seeding processor would be to store partial data. For instance for SIA you could store only the ids and store the details of projects separately. A reader would then have to merge these streams if it needs to. We didn't implement this alternative, because it adds a lot of complexity for reading data. All readers would have to know about different remote systems and how streams need to be combined or not. At the same time storing the partial data doesn't have any value, so this alternative only adds complexity.

Entities

Only having abstract classes is useless. The apps that implement the abstract classes from core are entity apps. Every app handles its own type.

Products [Publinova, Edusources, Github]

Products are research products (Publinova) or learning materials (Edusources). There are only a handful of fields that differ between research products and learning materials. In the product defaults you can see exactly which fields differ.

Background tasks for products are only normalizing learning material disciplines and filling out abstract study vocabulary terms. These tasks are pure functions that use the metadata app (outside of the scope of this document) to post-process some metadata values.

Files [Publinova, Edusources, Github]

Another name for files would be urls. It basically stores a URL with some metadata as you can see in the file defaults.

However unlike the products the files do trigger a lot of background tasks, which I won't explain in great detail, but I'll give a quick rundown below:

- *check_url* – will check if the URL exists and whether it redirects
- *tika* – will extract the text from the given URL by passing the URL directly to a separate Tika server that runs with the -enableUnsecureFeatures and -enableFileUrl flags. We think this is a responsible risk, because nothing but background workers can reach these Tika instances. These instances also run in their own Docker container, so a breach won't jeapordise the system. Lastly the URL's that are unsafe will standout to SURF quickly, because it means that a lot of jumble content will be visible in the frontends.
- *pdf_preview* – generates a preview by calling the pdf2image library.
- *video_preview* – generates a preview by calling YoutubeDL (for Vimeo URLs only)
- *youtube_api* – gathers data about Youtube videos (including thumbnails) for Youtube URLs.

Sources [Publinova, Edusources, Github]

This Django app specifies which entities should be harvested for which sources. You can do this from the admin screen by creating a "harvest source", but remember that code that is needed to harvest that source will need to exist as modules in the system. Apart from a "harvest source" it's also necessary to create a "harvest entity". This model specifies how an entity for a particular should be harvested and which sets should be included. It's also possible to pause harvesting by putting an entity for a source into manual mode. That way it becomes possible to inject test data into the dataset without sources overwriting that test data during harvests. Conceptually the sources module works similar to how the sources module works for the system that proxies persons and projects for Publinova without harvesting them. API endpoints exposed by that system should be possible to move to the harvester sources module.

Apart from specifying which sources to harvest in what capacity this module also contains code that is specific for certain sources. For instance, Sharekit has a particular way of parsing publications in API endpoints and instead of duplicating that code for products and files there are some generic functions inside sources that handle this Sharekit input. Because currently all source Resources are shared between Products and Files you find these in the sources app.

Search [Publinova, Edusources, Github]

This app is currently the only real output module. It takes the data from the current DatasetVersion and publishes it inside an OpenSearch index. Changes made to Documents are also synced to the index through tasks in this app.