Documentation

High overview

This refactor could be found on best_practices branch. It wasn't finished so some parts from the original codebase are not transferred here yet.

The main idea was to make Sigint a web API and do a global refactor. We also introduced celery lib to distribute jobs to workers. The ETL (Extract Transform Load) architecture is still used here.

Way to run the app is at the end.

Packages overview

The api package was supposed to be web api, but we didn't get to that part.

config package

This package contains json with configurations like Mongodb uri info, s3 keys, api keys, info about selectors for scraping websites, etc.

Also, there is model data_config that abstracts this config data.

dataset package

For now, it only contains csv datasets for data from LinkedIn sales navigator.

extract package

Contains classes for extracting companies' data from external sources, both for extracting base information of a company (company external data) and crawling companies' website. For extracting base information about companies in this refactor csv extract is only implemented. Other options are getting data from apis, or by scraping from some website that has directory of companies that could be interesting for our use cases. All classes that should be used for extracting base information about companies have to implement Extract class from extract/extract.py.

Csv package:

- ***CsvFilesDataExtractor*** - extract company external data (external_profile_dict) from csv files of a given directory
- ***AsyncCsvFilesDataExtractor*** – async version of above class (not implemented)

Scraping package:

- ***scrape_data*** – only has companies_crawler class which main functionality is to take in a list of CompanyInternalProfile objects and for each company run crawling of a company website to scrape content from it. This crawling is done using CustomSiteSpider (inherits scrapy.Spider class)

from extract/scraping/scrapy/spiders/custom_site_spider.py. After scraping each website page, the content is stored on s3 and content's path on s3 is stored in db.

- ***scrapy*** – this package has all the code needed for running scrapy spiders (standard scrapy stuff). Only thing that is added is CustomSiteCrawlerSpiderMiddleware class as a downloading middleware (which uses Selenium browser).
- ***selenium_local*** – selenium browser encapsulation.
- ***models*** – CsvConfig models some csv config information.

transform package

Code for transforming different data representations to internal representation (project's internal model - CompanyInternalProfile) and other way around. For example, from data extracted from external sources to internal representation and from internal representation to representation for csv export.

- **sources package** – classes that transform data for corresponding external source. This is needed because different sources have different data fields. (Only LinkedIn transferred in this refactor).
- **Transform class** – each method in this class calls appropriate method of SourcesTransformBase type of objects based on an external source. For example, Transform instance method *get_company_internal_profile_from_external* calls *get_company_internal_profile_from_external* from appropriate SourcesTransformBase type of object based on external_data_source, and this method basically takes external_profile_dict and creates CompanyInternalProfile. Example of other way around, we provide CompanyInternalProfile and external_data_source and Transform's method *company_internal_profile_to_csv* calls method of appropriate type of object based on external_data_source and gives us a dictionary that is suitable for csv export for a given external source.

load package

This package is in charge of data persistence. There are two parts, one for persisting domain model – classes that inherit abstract class Load, and the other part for storing scraped data – s3 package. Load class is acting as a repository (repository pattern), so we could change database easily just implement Load in new database. For data storage we didn't introduce interface to abstract any other type of data storage (we only have aws s3 service). But it would be easy to replace it with, for example, azure's equivalent just introduce a common interface that both s3 and azure would implement, similarly like it would be done with Load for database.

- **models package** – domain model. Will be discussed in more details in following heading.
- **mongodb package** – implementation of Mongo database, standard stuff, connection to db, queries, etc. The only specific is we are storing data in mongodb in different collections based on the first letter of the company's web domain (base web address) and saving to index table info about which collection company is stored in (there are only base_web_address and internal_collection_name fileds in index collection). Because of this we always need to query the index table first to take the collection name in which company data resides. There are 3 classes: the sync and async versions and abstract base class that these two inherit from. The base class inherits Load class.

- **store_company_iternal_profiles** – also 3 classes, one sync, one async and the base one. The need for this class emerged because a lot of pipelines need the functionality of storing a list of company internal profiles. So, its only functionality is to store a list of CompanyInternalProfiles.
- **s3 package** – interface toward aws s3 data storage buckets. There are sync and async versions that share the same base abstract class too.

Domain model

The domain model will be mentioned here too because some fields that weren't used that much are removed in this refactor. Some maybe should be removed from here too, because are not used that much, but it required some additional efforts in cleaning the db so it remained in this version for the time being.

There are 3 classes:

- CompanyInternalProfile
- CompanyInternalItem
- CompanyInternalItemPrediction

CompanyInternalProfile:

```python
class CompanyInternalProfile:
    id: str
    company_name: str
    full_web_address: str
    base_web_address: str
    company_internal_items: List[CompanyInternalItem]
    company_external_items_dict: dict
    language: Optional[str]
    company_internal_item_labels_dict: Dict[str, CompanyInternalItemPrediction]
    company_internal_item_predictions_dict: Dict[str, CompanyInternalItemPrediction]
    date: datetime
    latest_date_published: datetime
    source_data_created_date: datetime
    scraping_error_list: List[str]
```

The main class of the project is CompanyInternalProfile. It models the data of the real-world companies we are investigating using the sigint software. General rule is that things that have word external in them are something raw taken from some external source, and internal stuff is project's internal representation.

We'll not explain fields that are self-explanatory. The fields are:

- full_web_address – web address we got from external source.
- base_web_address – domain of the company's website. It's used as a unique field for querying the company and checking if it already exists in our database.
- company_internal_items – list of CompanyInternalItems (will be explained bellow).
- company_external_items_dict - since we have a lot of external sources we use it to store some additional data points that specific source is providing and might be useful.

- language – language of scraped content.
- company_internal_item_predictions_dict – for persisting info about criteria predictions
- company_internal_item_labels_dict – for info about the labels when doing the supervised learning
- latest_date_published – date when the company has been exported for the last time. At the same time, if the company profile has this field set it means it was processed.
- scraping_error_list – since we had a lot of issues with scraping we stored the errors so we could investigate further what was the cause.

CompanyInternalItem:

```
class CompanyInternalItem:
    full_web_address: Optional[str]
    base_web_address: Optional[str]
    bucket_name: Optional[str]
    bucket_path_list: list
    company_name: Optional[str]
    language: Optional[str]
    company_internal_item_predictions_dict: Optional[dict]
```

Models the scraped content of each web page of a company's website. We are processing these items when making predictions and training the model.

Fileds description:

- full_web_address – url of a scraped web page
- base_web_address – domain of a compnie's website
- bucket_name – name of a s3 bucket in which the content is stored
- bucket_path_list – list of paths where the scraped content is stored (can't remeber what was the idea with list since it's always one element in it). In mongo it is saved as a string, so we must do a string eval to get a list when reading data from db, not sure why it was stored that way, it's a thing we planned to resolve in some point of time.
- language – language of the scraped content
- company_internal_item_predictions_dict – storing information about predicted criteria of the item

CompanyInternalItemPrediction

```
class CompanyInternalItemPrediction:
    criterion_name: Optional[str]
    labeled_probabilities: Optional[dict]
    criterion_value_label: Optional[str]
    raw_criterion_value_number: Optional[int]
    raw_probability_values: Optional[list]
```

Model for predictions and labeling of the data for training. Fields:

- criterion_name – name of criterion (B2B_B2C, SOFTWARE_NON_SOFTWARE, STARTUP_CORPORATE)
- criterion_value_label – value of a label (predicted label when doing predictions)
- raw_criterion_value_number – integer value of a label
- raw_probability_values – probability of belonging to a criteria class

export package

This package is for exporting data that was processed by the models. The class DataExport abstracts any type of export (json, csv, etc.). In csv sub-package is implementation of DataExport for exporting to csv files, and that's the only type of export we used in the project. In the package export_data we have 3 classes: sync version, async version and the abstract base class. These classes have the whole pipeline for export. We provide the list of profiles to export_company_internal_profiles method and it exports the profiles to a file, then it stores file to s3 and updates the latest_date_published of exported companies. Dependency injection is used in constructor of these classes (sync and async versions) to provide the concrete DataExport object for exporting company profiles to a wanted file type. We used dependency injection also for db and data storage.

engines package

This package is for engines that run pipelines. It has two pipelines implemented (both have async and sync version). Engines don't share a common interface but follow duck typing concept. Every engine has a **run** method that does the whole job.

- csv_file_scraper_engine – this engine imports data from csv file(s) and runs scrapers of companies' webistes (both sync and async versions). First it is extracting external data from csv and creating CompanyInternalProfile-s from extracted data by using method get_company_internal_profiles of CompanyInternalProfilesCreator object, stores CompanyInternalProfile-s in db and runs the scraping of companies' websites. This is one of the most common pipeline that was used in the project to predict new companies.
- read_csv_files_export_engine – this one is for reading csv data of companies that we already run predictions on so we could export results. First it is extracting external data from csv and creating CompanyInternalProfile-s from extracted data by using method get_company_internal_profiles of CompanyInternalProfilesCreator object, queries created profiles to grab just those that are stored (since only those could have been processed by the

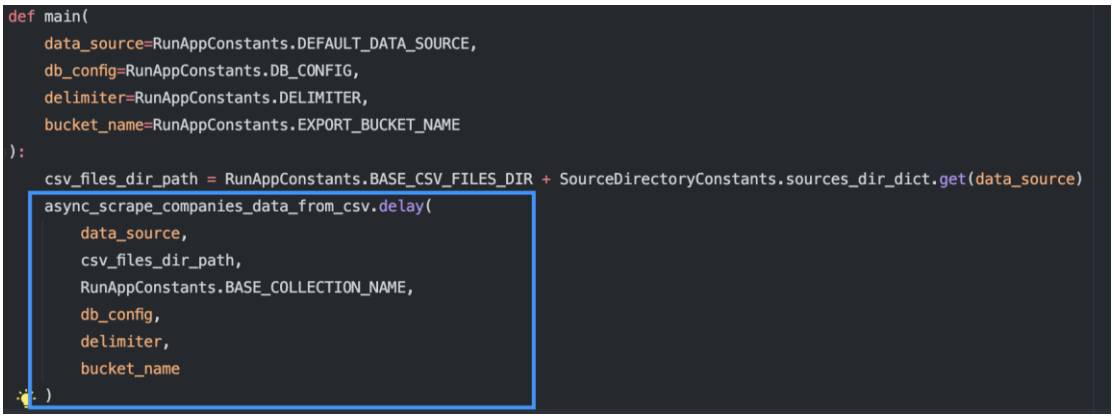
models), then it exports those companies to csv file and at the end adds latest_date_published to those companies that are exported.

Package company_internal_profiles_creator doesn't contain engines but a util classes (async and sync version and common base abstract class) for extracting data from external source and creating CompanyInternalProfiles from extracted data. It's located here because it doesn't fit in any other packages and here is the only place that it is used.

Running the app

For running the app first you need to start Redis server on port 32768. Redis is used as a broker for celery jobs. When Redis is set up you run celery with: *celery -A tasks worker --loglevel=INFO* command in terminal. To add new jobs to celery you write them in tasks.py file at the root of the project. The tasks have sync and async versions implemented.

You run specific task by importing it from tasks.py to main.py and running it in the following way:



```
def main(
    data_source=RunAppConstants.DEFAULT_DATA_SOURCE,
    db_config=RunAppConstants.DB_CONFIG,
    delimiter=RunAppConstants.DELIMITER,
    bucket_name=RunAppConstants.EXPORT_BUCKET_NAME
):
    csv_files_dir_path = RunAppConstants.BASE_CSV_FILES_DIR + SourceDirectoryConstants.sources_dir_dict.get(data_source)
    async_scrape_companies_data_from_csv.delay(
        data_source,
        csv_files_dir_path,
        RunAppConstants.BASE_COLLECTION_NAME,
        db_config,
        delimiter,
        bucket_name
    )
```

We use fire library for automatically generating command line interface. You can run the program by calling following command from terminal: python main.py --data_source=linkedin_sales_navigator --db_config=mongo_db_dev_local.