STORI Annual Report Downloader

Building a Reusable Web-Scraping Pipeline for Financial Reports

Gold Nwobu

November 24, 2025

Abstract

This report describes the design and implementation of a small but complete data pipeline that downloads annual financial report PDFs from the Belgian Financial Services and Markets Authority (FSMA) STORI system. Instead of manually clicking through the web interface, the solution interacts directly with the STORI backend API, filters for "Annual financial report" documents, and stores the resulting files in a structured folder with consistent filenames.

The pipeline is implemented in Python and split into two modules: `api_client.py`, which is responsible for all HTTP communication, and `main.py`, which orchestrates the workflow, handles logging, and manages local storage. The project emphasises clean separation of concerns, robust logging, and an easily extensible structure that can be reused for other web-scraping or data collection tasks. The report also outlines several concrete extensions, including CSV-based summaries, parallel downloads and a simple user interface.

1 Introduction

Regulatory filings such as annual financial reports are an important data source for academic research and financial analysis. In practice, however, these documents are often accessed via web portals that are designed primarily for human users. Manually downloading dozens or hundreds of reports is both time-consuming and error-prone.

The FSMA STORI platform is the official Belgian mechanism for the storage of regulated information. It exposes a web interface where users can search for issuers, filter by document type (for example, "Annual financial report"), and download associated files. Behind this interface, the browser communicates with a JSON-based backend API, which can be reused programmatically once the endpoints and payload formats are understood.

The goal of this mini-project is to build a small, reproducible pipeline that:

- automatically queries the STORI backend for annual reports,
- filters the results for English or Dutch PDF documents, and
- downloads a limited number of documents into a local folder using a clear naming convention.

Although the code base is relatively small, the project covers several core ideas in data engineering: reverse-engineering an API using developer tools, encapsulating HTTP logic, controlling side effects (files and logs), and implementing defensive checks such as rate limits and file name sanitisation.

2 System Overview

The project is organised around a simple but explicit architecture. The core components are:

- **api_client.py**: low-level client responsible for network communication with the STORI API.
- **main.py**: high-level orchestration layer that loads issuers, calls the API, filters results and saves files.
- **issuers.json.txt**: local, offline JSON file containing issuer information exported from STORI.
- **downloads/**: destination folder for saved PDFs.
- **logs/**: log directory containing `stori_downloader.log`.

Figure 1 conceptually illustrates the data flow.

Offline issuer list (`issuers.json.txt`) → `main.py` → `api_client.py` → STORI API (JSON) → filter EN/NL PDFs → download via `/download` → `downloads/`

Figure 1: High-level pipeline flow from local issuer list to downloaded PDFs.

The pipeline is intentionally capped to a small number of files (currently five). This makes it safe to run while testing and demonstrates how to enforce a global limit across multiple issuers.

3 Reverse Engineering the STORI API

The STORI website itself does not document its backend API. To call the endpoints programmatically, I used the browser's Developer Tools (Network tab) and observed the traffic while manually performing searches.

3.1 Discovering the Result Endpoint

When searching for an issuer and a specific document type in the web interface, the browser sends a POST request similar to:

```
POST https://webapi.fsma.be/api/v1/en/stori/result
```

The request body is a JSON object with fields such as:

```
{
  "startRowIndex": 0,
  "pageSize": 10,
  "sortDirection": "Ascending",
  "documentTypeId": "9813c451-9fd4-41ba-ba7d-4e0dda0d3051",
  "isDocumentTypeGroup": false,
  "publicationStart": "2021-01-01",
  "companyId": "02285e3c-71e9-496a-8d47-d1408131c44b"
}
```

The `documentTypeId` above corresponds to "Annual financial report" and was obtained directly from the payload generated by the web interface. Changing the issuer in the dropdown changes the `companyId` field.

The response is JSON with two main top-level keys: `resultCount` and `storiResultItems`. The latter is a list of filings, and each item contains metadata such as the company name, LEI, and a list of documents and attachments.

3.2 Discovering the Download Endpoint

Clicking the download icon for a document triggers a GET request of the form:

```
GET https://webapi.fsma.be/api/v1/en/stori/download?fileDataId=...
```

Here `fileDataId` is an opaque identifier found inside each `mainDocuments` or `attachments` entry in the result JSON. The response is raw binary data; the `Content-Type` header is typically `application/pdf`.

Together, these two endpoints (`/result` and `/download`) are sufficient to search for and retrieve documents without needing to automate the browser itself.

4 Implementation

4.1 Technology Stack

The project is implemented in Python using only the standard library plus the `requests` package for HTTP communication. No browser automation tools such as Selenium are required, because the pipeline works directly against the JSON API.

The main dependencies are:

- Python 3.8+ (tested with Python 3.10)
- `requests` for HTTP requests
- `logging`, `pathlib`, and `json` from the standard library

4.2 HTTP Client: `api_client.py`

The `api_client.py` module provides a minimal wrapper around the FSMA endpoints.

4.2.1 Session Management

The function `get_http_session()` creates a `requests.Session` with a custom User-Agent:

```
def get_http_session() -> requests.Session:
    session = requests.Session()
    session.headers.update({
        "User-Agent": "STORI-Downloader/1.0 (academic project)",
    })
    return session
```

Using a session allows connection reuse and centralises header configuration.

4.2.2 POST Helper

To avoid repeating boilerplate code, the function `post_json` takes a session, a relative path and a JSON body, and returns the decoded JSON response. It performs basic error handling:

- raises on HTTP errors,
- logs timeouts and request exceptions,
- logs the first part of the raw response if JSON decoding fails.

4.2.3 Result and Download Functions

Two higher-level helpers are built on top of `post_json` and `session.get`:

- `fetch_stori_results(session, payload)` calls `/api/v1/en/stori/result` and returns a dictionary whose structure matches the STORI response.
- `download_file(session, file_data_id)` calls `/api/v1/en/stori/download` with the given `fileDataId` and returns the raw bytes of the downloaded file.

By isolating all network-specific code in one file, the rest of the pipeline can focus on business logic and becomes easier to test and maintain.

4.3 Orchestration: `main.py`

The `main.py` module is responsible for:

- configuring logging,
- loading the issuer list,
- coordinating search requests and downloads,
- enforcing the global download limit.

4.3.1 Logging Setup

The function `setup_logging()` configures logging to write both to the console and to `logs/stori_downloader`. Throughout the pipeline, `logging.info` and `logging.error` are used instead of plain `print`, which makes it easier to diagnose issues after the fact.

4.3.2 Issuer Management

To decouple the scraper from the live dropdown API, the issuer list is stored locally in `issuers.json.txt`. Each entry contains at least a `companyId` and an `abbreviation`. The function `ensure_issuer_file` loads this file and normalises it into a simpler list of dictionaries with keys `id` and `name`.

If the file is missing, a default list with a single issuer (ACACIA PHARMA GROUP) is created and written to disk. This makes the script self-contained for testing but in practice the full issuer list is prepared in advance.

4.3.3 File Naming

To keep the downloaded reports organised, each file name follows the pattern:

`IssuerName_LEI_AnnualReport_YYYY-MM-DD.pdf`

Before constructing the file name, the helper function `sanitize_for_filename` removes problematic characters and replaces spaces with underscores. If a file with the same name already exists, versioned suffixes (`_v2`, `_v3`, ...) are appended to avoid overwriting.

4.3.4 Per-Issuer Download Logic

The core of the pipeline is implemented in `download_for_issuer(session, company_id, ...)`. For a given issuer, it:

1. builds a search payload with the issuer's `companyId`, the annual report document type ID, and a lower bound on `publicationStart`,
2. calls `fetch_stori_results` to retrieve all relevant filings,
3. iterates through `storiResultItems`, collecting both `mainDocuments` and `attachments`,
4. filters documents to those that:
   - have language `"en"` or `"nl"`,
   - are PDFs (either via `fileType` or the file extension),
5. downloads each selected document via `download_file` and saves it to the `downloads/` folder.

The function also keeps track of the cumulative number of downloaded files and returns the updated count, so that the global limit is respected.

4.3.5 Global Control Flow

The `main()` function ties everything together:

1. initialise logging and create an HTTP session,
2. load issuers from `issuers.json.txt`,
3. set:
   - `DOCUMENT_TYPE_ANNUAL` to the ID of "Annual financial report",
   - `MAX_DOWNLOADS` to 5,
4. iterate over issuers, calling `download_for_issuer` until the maximum number of downloads is reached.

In the current configuration, the pipeline stops after downloading five PDF files, even if more are available.

5 Experimental Run

To validate the pipeline, I used an issuer with several years of annual financial reports available. Running:

```
python main.py
```

produced log output indicating successful calls to the STORI API, filtering of documents and a final summary such as:

```
Finished. Total PDFs downloaded: 5
```

The `downloads/` folder then contained five PDF files with names like:

```
2VALORISE_5493002PV2OV5MP41Y57_AnnualReport_2014-04-22.pdf
2VALORISE_5493002PV2OV5MP41Y57_AnnualReport_2015-04-21.pdf
...
```

This confirms that the filter conditions (annual reports, EN/NL, PDF) and the naming logic are working as intended.

6 Limitations

Although the current implementation works reliably for the test cases used, there are several limitations:

- **Hard-coded document type ID.** The ID for "Annual financial report" is currently hard-coded. If the FSMA changes this ID, the script must be updated manually.
- **Issuer list maintenance.** The offline issuer file `issuers.json.txt` is created outside the script. Keeping this file up to date is a separate step.
- **Simple stopping condition.** The global limit is based only on the number of files downloaded, not on date coverage or issuer count.
- **Single-threaded downloads.** Documents are downloaded sequentially, which is simpler to reason about but slower than parallel approaches.

7 Future Work and Extensions

During development I identified several improvements that would make this pipeline more powerful and researcher-friendly. These are left as future work but the current design already anticipates them.

7.1 CSV Summary of Downloads

One natural extension is to generate a CSV file (for example, `download_summary.csv`) that records, for every downloaded document:

- issuer name and LEI,
- publication date,
- original file name and language,
- final local file path and size.

This would turn the pipeline into a ready-to-use dataset builder and make it easy to inspect coverage or join the reports with other financial data.

7.2 Parallel Downloading

Currently, all files are retrieved one after another. Since each download is I/O bound, it would be straightforward to parallelise this using Python's `concurrent.futures.ThreadPoolExecutor`. The main challenges would be:

- preventing the server from being overwhelmed (e.g., by limiting the number of concurrent threads), and
- keeping the global download counter consistent in a multi-threaded environment.

7.3 Basic User Interface

For non-technical users, a small user interface would be helpful. One option would be a simple command-line interface where the user can specify:

- maximum number of downloads,

- date range,
- a subset of issuers to target.

Another option would be a lightweight web UI using a framework like Streamlit, which could visualise progress, show the list of downloaded reports and allow users to export metadata.

7.4 Additional Robustness

Further improvements could include:

- automatic retry strategy for transient network errors,
- configuration via a separate `config.json` instead of hard-coded constants,
- optional integration with cloud storage (e.g., uploading PDFs to an S3 bucket after download).

8 Conclusion

This project demonstrates how a relatively small amount of Python code can replace a manual and repetitive downloading task with a reproducible pipeline. By reverse engineering the FSMA STORI backend API and wrapping it in a clean `api_client`, the solution avoids browser automation and focuses on structured HTTP requests and well-defined JSON responses.

The final system reads issuers from a local JSON file, queries the STORI API for annual financial reports, filters documents to EN/NL PDFs, and stores them in an organised folder with meaningful file names. Comprehensive logging makes it easier to understand what the script is doing and to troubleshoot issues.

Although there are many opportunities for extensions—such as CSV summaries, parallel downloads and user interfaces—the current version already provides a solid foundation for large-scale text-based research using annual reports from the FSMA STORI platform.