www.servion.com | © Servion Global Solutions

Table of Contents

VERSION HISTORY

Version No	Date	Change Initiated By	Updated By	Summary of Changes
1.1	04/04/2025	*Servion*		Baselined

1. General Information

1.1. Overview

This manual is intended to help developers understand and implement the ApiDataFetcher class, which is designed to fetch and process API data for multiple domains from specific API endpoints. The class supports asynchronous operations, multiple data types, and different platforms. It handles the entire workflow from loading domain data from input files to making API requests and saving the responses in a structured format.

ApiDataFetcher is a comprehensive solution for batch processing domain data through the Semrush API. It organizes the retrieved data in a structured manner, ensuring efficient data organization and retrieval for further analysis.

2. System Summary

This section provides a general overview of the system components and requirements.

2.1. Key features

- Supports loading data from CSV and Excel files (XLSX, XLS)
- Handles asynchronous API requests with concurrency control
- Processes multiple domains, platforms, and data types
- Organizes responses by site ID, platform, and data type
- Implements comprehensive logging
- Saves responses in a structured JSON format with organized folder hierarchy

2.2. Dependencies

The ApiDataFetcher class relies on several Python libraries to function properly:

- **pandas:** For data manipulation and CSV/Excel file handling
- **logging:** For event logging
- **io:** For file stream handling

- **json:** For JSON parsing and serialization
- **os:** For filesystem operations
- **asyncio:** For asynchronous processing
- **aiohttp:** For asynchronous HTTP requests
- **urllib.parse:** For URL encoding
- **datetime:** For date/time handling
- **typing:** For type hints
- **glob:** For file pattern matching

3. Getting Started

This section provides a detailed overview of the ApiDataFetcher class implementation, including initialization, core methods, and helper functions.

Class initialization

If The constructor initializes the ApiDataFetcher with a file path or file stream containing domain data. It sets up the necessary attributes for API connections, data processing, and result storage.

def __init__(self, file_path: Union[str, io.TextIOWrapper]):

Parameters:

- file_path: Path to the input file or a file stream. Supports CSV, XLSX, and XLS formats.

Attributes:

- file_path: Input file path or stream
- df: Pandas DataFrame to store loaded domain information
- base_url: Base API endpoint URL
- platforms: List of platforms to fetch data for (desktop, mobile)
- data_types: Types of data to retrieve (subdomain_organic, subdomain_rank)
- all_responses: Dictionary to store all API responses
- fetch_datetime: Datetime of data fetch from the input file

- run_datetime: Current run datetime
- processed_domains: Set of domains already processed (to avoid duplicates)
- site_responses: Dictionary to store responses organized by site ID, platform, and data type

Core Methods

3.2.1 Load Data

Loads domain data from the input file into a pandas DataFrame.

def load_data(self) -> None:

- Supports Excel (.xlsx, .xls) and CSV files
- Works with file paths or file streams
- Validates the presence of required 'domain' and 'site_id' columns
- Sets fetch datetime from the DataFrame when available
- Handles potential loading errors with appropriate logging

3.2.2 Get Export Columns

Determines the export columns based on the specified data type.

def get_export_columns(self, data_type: str) -> str:

- Returns different sets of columns based on the data type
- For 'subdomain_organic', returns a comprehensive set of columns
- For 'subdomain_rank', returns a specific subset of columns

3.2.3 Make API Request Async

Makes an asynchronous API request for a specific domain, platform, and data type.

async def make_api_request_async(self, domain: str, platform: str, data_type: str, site_id: str, semaphore: asyncio.Semaphore) -> Optional[Dict[str, Any]]:

- Constructs the API URL with appropriate query parameters
- Uses a semaphore to limit concurrent requests

- Handles request timeouts and connection errors
- Parses responses based on their format (CSV or JSON)

3.2.4 Parse CSV to JSON Async

Parses CSV content from API responses into a structured JSON format.

async def parse_csv_to_json_async(self, content: str, domain: str, platform: str, data_type: str, site_id: str) -> Optional[Dict[str, Any]]:

- Converts CSV content to a pandas DataFrame
- Organizes the data in a standardized format
- Stores the processed data in the site_responses dictionary

3.2.5 . Parse JSON Response Async

Parses JSON content from API responses into a structured format.

async def parse_json_response_async(self, content: str, domain: str, platform: str, data_type: str, site_id: str) -> Optional[Dict[str, Any]]:

- Handles different types of JSON responses (lists or dictionaries)
- Organizes the data in a standardized format
- Stores the processed data in the site_responses dictionary.

3.2.6 Fetch Data Async

Asynchronously fetches API data for all domains in the DataFrame.

async def fetch_data_async(self) -> None:

- Manages concurrent API requests with a semaphore (limiting to 10 simultaneous tasks)
- Creates tasks for each domain, platform, and data type combination
- Prevents processing duplicate domains
- Handles and logs potential errors during data fetching

3.2.7 Save Site Specific Responses

Saves API responses as site-specific JSON files with an organized folder structure.

def save_site_specific_responses(self) -> None:

- Creates a folder for each site ID
- Creates a YYYY-MM subfolder for date organization
- Generates separate JSON files for each data type
- Includes data for both platforms (desktop and mobile)
- Constructs filenames with site ID, data type, and run datetime

3.3. Helper Methods

Set Run Datetime

Sets the run datetime to the 15th of the current month.

@staticmethod

def set_run_datetime() -> str:

- Returns a formatted datetime string used for file naming and record-keeping

Run Methods

Run the entire data fetching and processing workflow.

async def run_async(self) -> None:

def run(self) -> None:

- run_async(): Asynchronous version of the process
- run(): Synchronous wrapper that calls the async version (for backward compatibility)
- Loads data, fetches API data, and saves responses

4. Usage Examples

Here are examples of how to use the ApiDataFetcher class:

Single file processing:

```
# Single file processing
fetcher = ApiDataFetcher("path/to/domains.csv")
fetcher.run()
```

Processing multiple files from a directory:

```
# Processing multiple files from a directory
directory_path = "path/to/csv/files"
csv_files = glob.glob(os.path.join(directory_path, "*.csv"))
for csv_file in csv_files:
    with open(csv_file, 'r', encoding='utf-8') as file_stream:
        fetcher = ApiDataFetcher(file_stream)
        fetcher.run()
```

5. Error Handling and Logging

The code implements comprehensive error handling and logging:

- All exceptions are caught and logged with appropriate context
- A log file (api_requests.log) records all operations with timestamps
- Console messages provide real-time feedback during execution
- Specific error messages for different types of failures (file loading, API requests, parsing)

6. APPENDIX A - ACRONYMS AND ABBREVIATIONS

Acronym	Abbreviation
API	Application Programming Interface
CSV	Comma-Separated Values
JSON	JavaScript Object Notation
XLSX	Excel Open XML Spreadsheet
XLS	Excel Binary File Format