Jean-Marin Chaintron

Théo Risy

The Nutritional Weather Dashboard : Nutriweather

Big data project

II.2314 / II.2414 - Advanced Databases and Big Data

Table des matières

Introduction

This project is a big data project that combines weather data and meal recommendations from API using technologies like Apache Airflow, Apache Spark, Elasticsearch, HDFS and Kibana. The pipeline fetches real-time weather data and meal information, processes them through multiple stages and provides meal recommendations and advices based on the weather conditions.

Usage

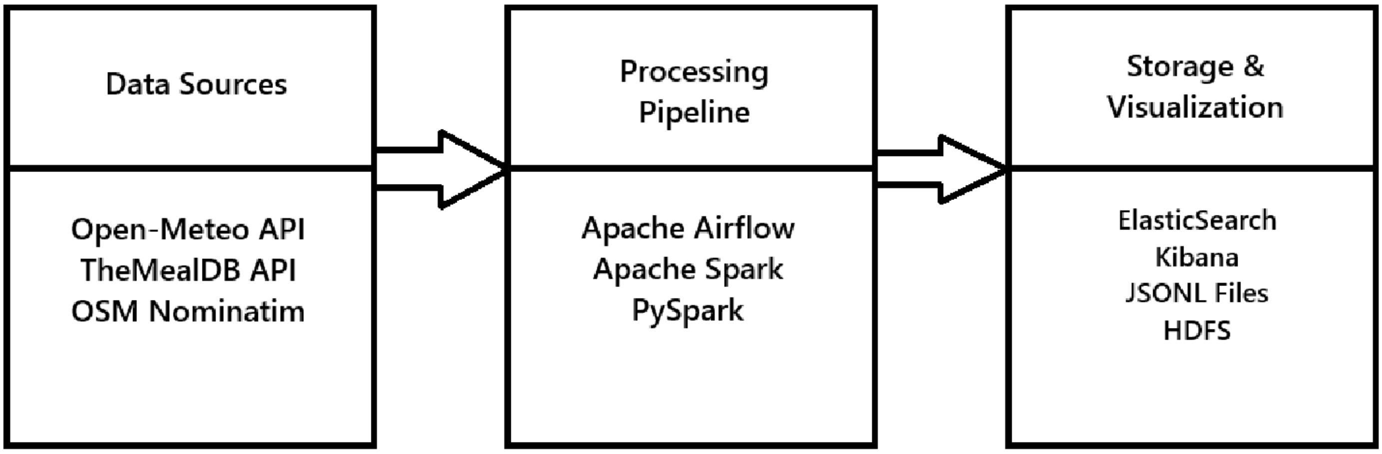
To use our project you need OpenJDK 17, Python 3.11+, Docker 4.0+ and Docker Compose V2 and Astronomer 3.0-2 :

- Install Astronomer CLI: https://www.astronomer.io/docs/astro/cli/install-cli/
- Clone the GitHub repository on your local disk: git clone https://github.com/Melaeline/nutriweather-datalake
- Open Docker and in a terminal, write this: cd nutriweather-datalake
- Then: astro dev start

And you're good to go ! You will just need to open airflow port 8080:8080, run the pipeline dag and open the ELK port 5601:5601.

On Kibana, create a new data view with the two nutriweather-indexes and import the dashboard as a saved object (dashboard.ndjson available in the zip file of the project) and select nutriweather*.

Architecture Overview



Technology Used

Core Technology	• Apache Airflow 2.9.2 - Workflow orchestration and scheduling • Apache Spark 3.5.6 - Distributed data processing • PySpark - Python API for Spark • Elasticsearch 8.15.0 - Search and analytics engine • Kibana 8.15.0 - Data visualization and exploration • Docker & Docker Compose - Containerization and orchestration

	• HDFS : Distributed file system to store our indexes
Development & Runtime	• Astronomer Runtime 3.0-2 - Airflow distribution • Python 3.11+ - Primary programming language • OpenJDK 17 - Java runtime for Spark • Bitnami Spark Images - Pre-configured Spark containers
Data Processing Libraries	• pandas 1.5.0+ - Data manipulation and analysis • pyarrow 10.0.0+ - Columnar data format support • numpy 1.26.0+ - Numerical computing • requests 2.31.0+ (< 2.33.0) - HTTP client library (version-pinned for urllib3 compatibility) • urllib3 1.26.0+ (< 2.3.0) - HTTP library (compatible with requests)
API Integration	• openmeteo-requests 1.1.0+ - Weather API client • requests-cache 1.1.0+ - HTTP request caching • retry-requests 2.0.0+ - Request retry mechanism

External APIs

Open-Meteo Weather API	• Endpoint: `https://api.open-meteo.com/v1/forecast` • Purpose: Real-time weather data collection • Data Retrieved: • Current temperature, humidity, wind speed • Hourly temperature forecasts • Daily UV index maximum • Location metadata (coordinates, timezone) • Rate Limits: Free tier, no authentication required • Coverage: Global weather data with high accuracy
TheMealDB API	• Endpoint: `https://www.themealdb.com/api/json/v1/1` • Purpose: Comprehensive meal database access • Data Retrieved: • Meal names, categories, and regions • Detailed cooking instructions • Ingredient lists with measurements • Meal thumbnails and metadata • Coverage: 1000+ international recipes • Search Method: Alphabetical iteration (a-z) for complete dataset
OpenStreetMap Nomitatim	• Endpoint: `https://nominatim.openstreetmap.org/reverse` • Purpose: Reverse geocoding for location names • Usage: Convert coordinates to human-readable locations • Rate Limits: 1 request per second (implemented with delays)

Data Pipeline Flow

For the steps of the project, we use dag files and scripts launched by Airflow. All the files created during the different steps are stored locally on hard disk but also on HDFS as double storage.

Stage 1: Data Ingestion (Raw Layer)

The first step is the call of the APIs in order to get the raw data we need to make our dashboard.

Raw Data Structures:

- Meals: Complete TheMealDB API response with 50+ fields per meal
- Weather: Structured JSON with metadata, current, hourly, and daily sections
- Retention: All raw files preserved for reprocessing

Stage 2: Data Transformation (Formatted Layer)

The second step consists in transforming the data and formatting it in parquet format and json format using PySpark so that it is usable.

Transformations Applied:

- Meals Processing:
  - Ingredient consolidation (20 ingredient fields → single array)
  - Preparation time estimation algorithm
  - Instruction cleaning and formatting
  - Category and region standardization
  - Clean single-file Parquet output (no Spark artifacts)
- Weather Processing:
  - Location name enrichment via reverse geocoding
  - Timestamp standardization (ISO 8601)
  - Data validation and type conversion
- File Architecture:
  - Clean Output: Single parquet files without `_SUCCESS` or partition artifacts
  - Temporary Processing: Spark artifacts handled internally and cleaned up
  - Consistent Naming: `formatted_meals_YYYYMMDD_HHMMSS.parquet` format

Stage 3: Data Integration (Usage Layer)

The third step consists in merging both formatted files to produce a usable unique file to index.

```
merge_formatted.py → /include/usage/temperature_timeseries_YYYYMMDD_HHMMSS.jsonl

                   → /include/usage/enhanced_recommendations_YYYYMMDD_HHMMSS.jsonl
```

Stage 4: Data Indexing (Elasticsearch)

We then use a dag and a script to index data so that we can make our data views and our dashboard.

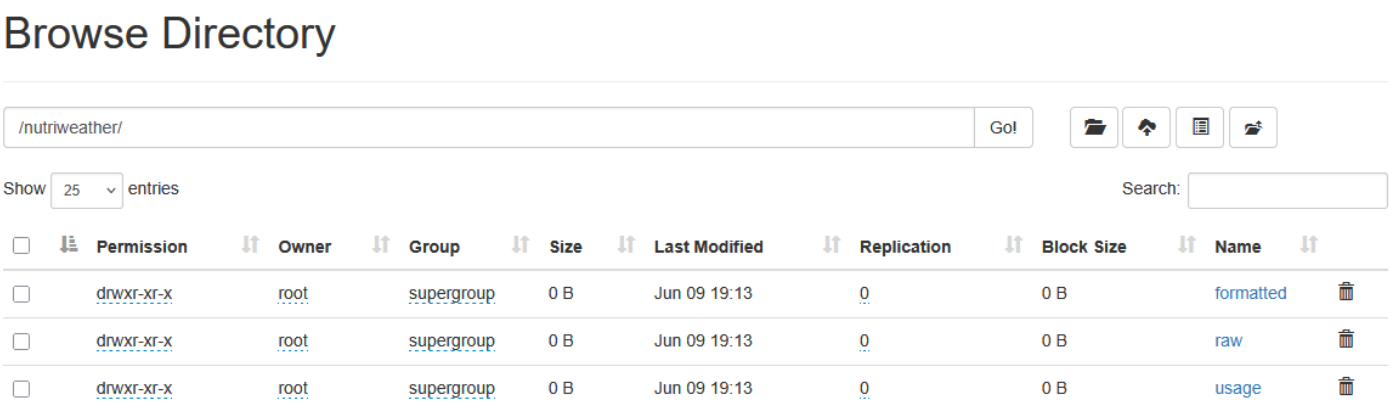
```
index_elasticsearch.py → Elasticsearch indices:

                       → nutriweather_temperature

                       → nutriweather_enhanced
```

File Formats & Data Organization

Directory Structure

```
/usr/local/airflow/include/
├── raw/
│   ├── meals/          # JSON files from TheMealDB API
│   └── weather/        # JSON files from Open-Meteo API
├── formatted/
│   ├── meals/          # Parquet files (optimized for Spark)
│   └── weather/        # JSON files (enriched with location data)
├── usage/              # JSONL files (Elasticsearch-ready)
└── scripts/            # Python processing modules
```

For HDFS, we have /nutriweather/ directory containing the /raw, /formatted and /usage folders.

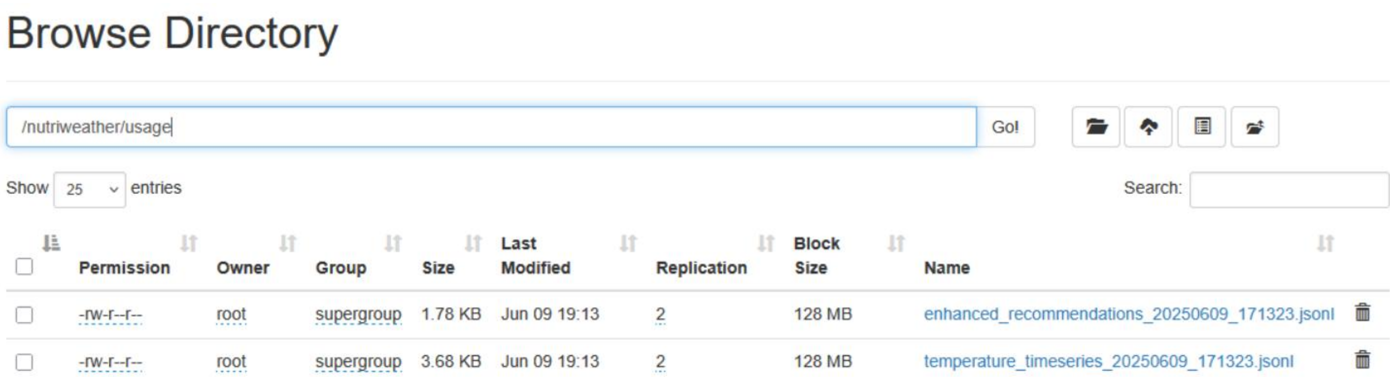


Browse Directory

/nutriweather/ Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	root	supergroup	0 B	Jun 09 19:13	0	0 B	formatted
drwxr-xr-x	root	supergroup	0 B	Jun 09 19:13	0	0 B	raw
drwxr-xr-x	root	supergroup	0 B	Jun 09 19:13	0	0 B	usage



Browse Directory

/nutriweather/usage Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	1.78 KB	Jun 09 19:13	2	128 MB	enhanced_recommendations_20250609_171323.jsonl
-rw-r--r--	root	supergroup	3.68 KB	Jun 09 19:13	2	128 MB	temperature_timeseries_20250609_171323.jsonl

File Naming Convention

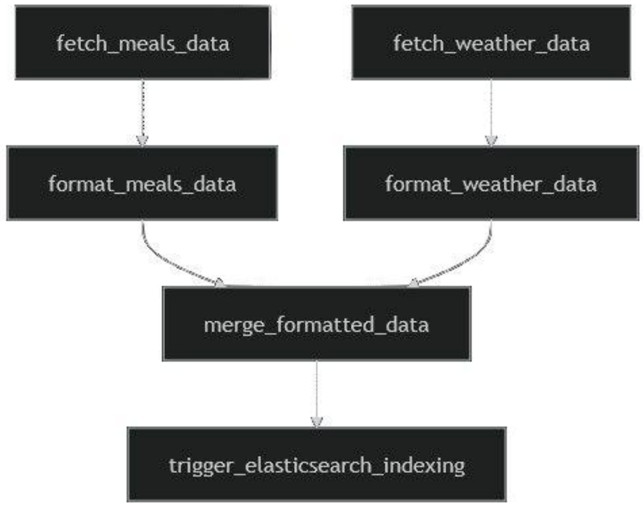
- Pattern: `{type}_{category}_{YYYYMMDD_HHMMSS}.{extension}`
- Example: `formatted_meals_20241219_153045.parquet`
- Benefits: Chronological sorting, easy latest file identification

Format Justifications

- Raw → JSON: Preserves original API response structure
- Formatted Meals → Parquet: Columnar format optimized for Spark operations
- Formatted Weather → JSON: Maintains nested structure for complex weather data
- Usage → JSONL: Elasticsearch bulk indexing compatibility

DAG Execution & Orchestration

All the dags are launched by the first dag pipeline dag as shown on the schema.



Primary DAG: start_pipeline_dag

Trigger: Manual execution or API call

Schedule: On-demand (no automatic scheduling)

Max Active Runs: 1 (prevents concurrent executions)

Secondary DAG: index_elasticsearch_dag`

Purpose: Elasticsearch data indexing

Trigger: Called by primary DAG

Features:

- Index creation with optimized mappings
- Bulk document indexing
- Error handling and retry logic

Two indexes are created to prevent the default aggregation of data from Kibana, especially for the hourly data part (temperature and timestamp).

Docker Environment

Container Architecture

```
Services:
├── Airflow Scheduler     # DAG scheduling and monitoring
├── Airflow Webserver     # Web UI (localhost:8080)
├── Airflow Worker        # Task execution
├── PostgreSQL            # Airflow metadata storage
├── Elasticsearch         # Search and analytics (localhost:9200)
├── Kibana                # Visualization dashboard (localhost:5601)
├── Namenode              # Belongs to HDFS
├── Datanode1             # Belongs to HDFS
├── Datanode2             # Belongs to HDFS
├── Spark Master          # Cluster coordination (localhost:8082)
└── Spark Worker          # Distributed computing (2GB memory)
```

Volume mapping and Network configuration are available in the docker-compose.override.yaml or on the github readme.

Prerequisites

- Docker Desktop 4.0+
- Docker Compose V2
- 8GB+ available RAM
- 10GB+ free disk space

Monitoring & Observability

Airflow Monitoring

- Web UI: Task status, logs, execution history
- Metrics: Task duration, success rate, resource usage
- Alerting: Email notifications on failure (configurable)

Spark Monitoring

- Spark UI: Job execution, stage details, executor status
- Metrics: Memory usage, task distribution, shuffle operations
- Logs: Driver and executor logs via Docker

Elasticsearch Health

- Cluster Health: `GET /_cluster/health`
- Index Statistics: `GET /_stats`
- Document Counts: Real-time via Kibana dashboards

Data Analysis & Visualization

Kibana Dashboards

1. Temperature Trends: Time-series visualization of temperature data
2. Meal Recommendations: Distribution of suggested meals by weather
3. Location Analytics: Geographic distribution of weather data
4. System Health: Pipeline execution metrics and error rates

We chose to make a mockup for the dashboard. The idea is to show the weather metrics of the day, including current temperature, wind speed and humidity rate, but also the temperature during the entire day. The dashboard then shows the recommended meal with metrics about the recipe and all the ingredients and cooking steps.

We created a dataview in order to use the indexes. The problem we encountered was that the graph was using aggregation, as the data is indexed by Elasticsearch. To prevent that from happening, we separated the indexed data in two different indexes.

Nutriweather Dashboard

On the left side we have all the meal recommendation and information and on the right we can see the different metrics of the dashboard, with the hourly temperatures graph.

Conclusion

This project has been an interesting project, having to understand and explore different technologies, building a coherent pipeline to receive data, format it and index it so that we could make a valuable dashboard.

We encountered some problems, for example, HDFS is only a backup storage, as we store our data on both HDFS and the local disk. The dashboard was not easy to make and we have some troubles with the visualization of the latest data information, but we still managed to get a dashboard that can updates itself.

We are still happy of what we accomplished considering the difficulty to discover technologies that we never used before.