Wikipedia Streaming Project

Stetsyshyn Victoria, Savkiv Yaryna

Project Overview

The main task of the project was to process streaming data from Wikipedia using stream and batch processing so that the user could make API requests and receive analytical answers based on the collected data.

The system implements two types of API requests:

- ❖ **Category A** - requests to pre-computed aggregated reports, which are updated hourly based on batch processing.
- ❖ **Category B** - ad-hoc queries to the stored data, which allow you to get results in real time.

Project Architecture

The architecture of our project consists of 11 components.

- ★ **Wikipedia Stream -** data of real-time page creation events (JSON format)
- ★ **Kafka Producer** - reads the Wikipedia stream and pushes raw messages into the Kafka Input-Topic.
- ★ **Input-Topic** - stores the JSON format raw data from Wikipedia Stream, which means that the data isn't preprocessed yet at this stage.
- ★ **Spark Streaming Job** - reads from Kafka Input-Topic, transforms data (parsing and filtering), and writes to Processed-Topic
- ★ **Processed-Topic** - stores clean, structured messages, which are ready for the further processing

We decided to use **two Kafka topics** instead of writing directly to the database **in order to separate concerns and improve system flexibility.** This approach enables easy **replay of raw data**, **protects against invalid records**, and allows **asynchronous**, fault-tolerant processing — all of which contribute to a more scalable and resilient architecture.

Category A:

- **PostgreSQL -** intermediate relational database to store non-aggregated events coming from Kafka Processed-Topic after processing in Spark Streaming.

We chose PostgreSQL as the database for intermediate storage because it is a powerful relational database that is well suited for further analytics and aggregation computing, unlike Cassandra.

- **Pandas Aggregator -** scheduled batch processor that reads data from PostgreSQL, computes aggregates, and stores results into Cassandra.

Initially, we planned to implement full data processing and aggregation within a Spark Streaming Job, using periodic calls to PostgreSQL for further analysis using Cron jobs.

However, in the implementation process, we faced computing resource limitations, which made it impossible to implement this approach in full.

In this regard, it was decided to use a less resource-intensive approach - Python script (**Pandas Aggregator**) that is periodically executed and processes data from PostgreSQL.

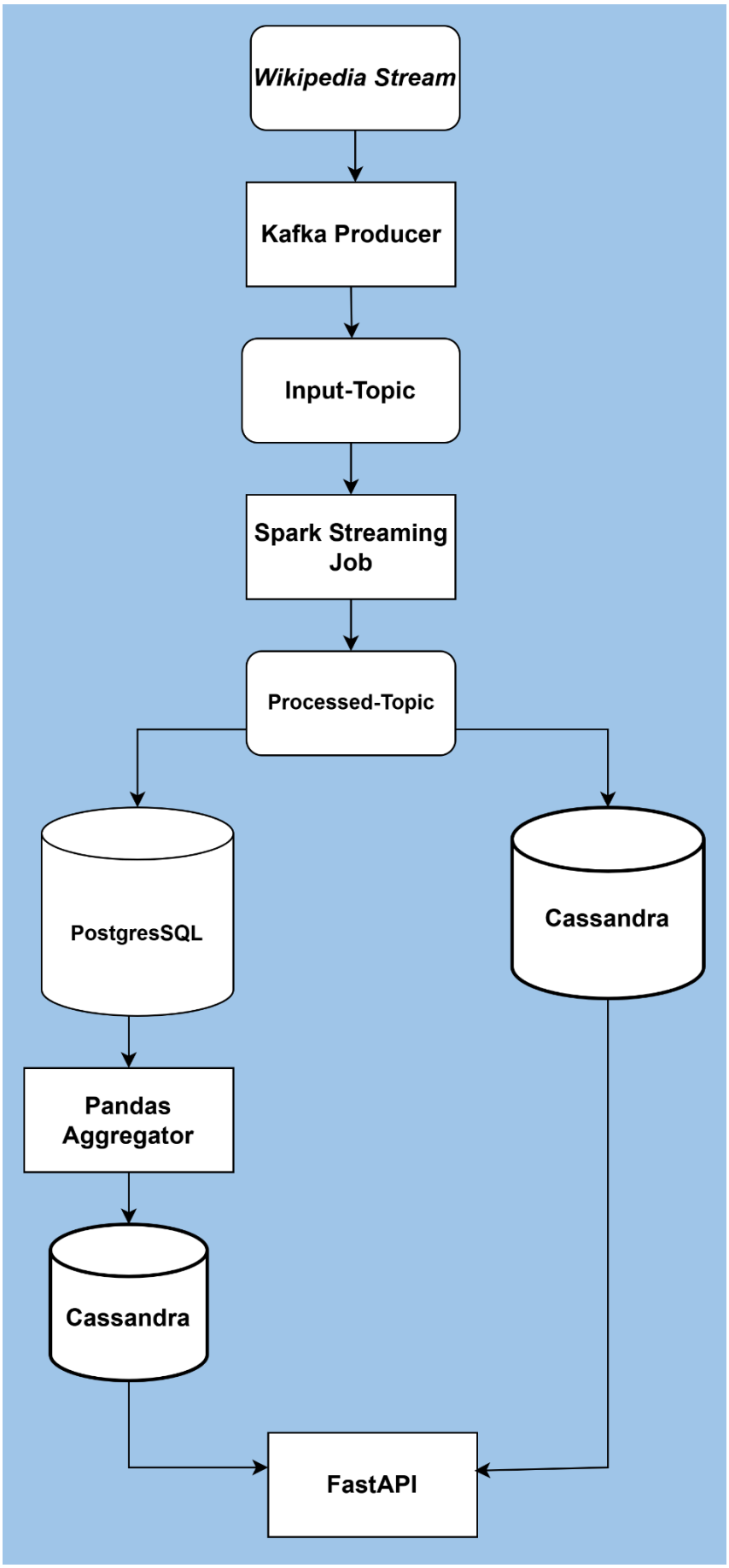
- **Cassandra** - database, which has 3 tables (***domain_stats_by_hour, bot_stats_last_6h, top_users_last_6h***), which are designed to store the final aggregated results required to serve the API requests from Category A, and are optimized for direct lookup based on time intervals.

Category B:

- **Cassandra** - database, which has 6 tables (***domains_created, pages_by_user, domain_page_counts, pages_by_id, user_page_counts_by_date, users***), which are designed to store the necessary data to serve the API requests from Category B.

For Both Categories:

- **FastAPI -** endpoints rely entirely on the precomputed data for category A and ad hoc queries for category B. Since the aggregations are already stored in Cassandra databases(for category A), API responses are extremely fast and efficient, as no real-time processing is required during request time. Category B requires a little more time to respond, but not significantly. This design ensures consistent low-latency performance even under high query loads. You can see the documentation and interact with the available API endpoints using **Swagger UI**


Wikipedia Stream
Kafka Producer
Input-Topic
Spark Streaming Job
Processed-Topic
PostgresSQL
Cassandra
Pandas Aggregator
Cassandra
FastAPI

Data Models

1. **PostgreSQL** *(Relational database)*

Role: stores raw cleaned data for further batch analytics in Category A.
Why we used:

- simple structure writing from Kafka.
- supports JOIN, SQL queries, aggregation
- Gives opportunity to easily process necessary data for further analysis

```
▷Run on active connection | ≡Select block
CREATE TABLE IF NOT EXISTS wiki_events (
    id SERIAL PRIMARY KEY,
    domain TEXT,
    created_at TIMESTAMP,
    page_id BIGINT,
    page_title TEXT,
    user_id BIGINT,
    user_name TEXT,
    comment TEXT,
    user_is_bot BOOLEAN
);
```

2. **Cassandra** *(NoSQL database)*

Role: stores both pre-aggregated data (for Category A) and 'fresh' non-aggregated events for queries (Category B).
Why we used: provides very fast REST API reading

3. Kafka *(Message Broker)*

Role: handles real-time ingestion of events from Wikipedia Stream into the data pipeline.

Why we used: enables real-time, fault-tolerant data delivery, integrates seamlessly with Spark Streaming

4. Spark Streaming *(Stream Processing)*
Role: consumes messages from Kafka Input-Topic, parses and filters them, and outputs cleaned data into Processed-Topic, and then writes the structured results to both PostgreSQL (for Category A) and Cassandra (for Category B).

Why we used: integrates seamlessly with Kafka, and allows us to validate schemas and transform incoming Wikipedia event data before storing it in the topics and databases.

Created tables in Cassandra for Category A and Category B

- **"Return the list of existing domains" -** ***domains_created***

Purpose: stores domains for which at least one page has been created.

```
CREATE TABLE IF NOT EXISTS domains_created (
    domain TEXT,
    created_at TIMESTAMP,
    PRIMARY KEY (domain)
);
```

- **"Return all the pages created by the user with a specified user_id" - pages_by_user**

Purpose:stores a list of pages created by a particular user

```
CREATE TABLE IF NOT EXISTS pages_by_user (
    user_id BIGINT,
    page_id BIGINT,
    dt TIMESTAMP,
    PRIMARY KEY (user_id, page_id)
);
```

- **"Return the number of articles created for a specified domain" - domain_page_counts**

Purpose: stores the total number of pages created in a particular domain.

```
CREATE TABLE IF NOT EXISTS domain_page_counts (
    domain TEXT,
    page_count COUNTER,
    PRIMARY KEY (domain)
);
```

- **"Return the page with the specified page_id" - pages_by_id**

Purpose: allows you to find a page by its page_id

```
CREATE TABLE IF NOT EXISTS pages_by_id (
    page_id BIGINT,
    page_title TEXT,
    domain TEXT,
    dt TIMESTAMP,
    PRIMARY KEY (page_id)
);
```

- **"Return id, name, and number of created pages for all users in a specified time range" - user_page_counts_by_date**

Purpose: allows you to get information about user activity in a certain time range.

```
CREATE TABLE IF NOT EXISTS user_page_counts_by_date (
    dt DATE,
    user_id BIGINT,
    page_count COUNTER,
    PRIMARY KEY ((dt), user_id)
);
```

- This table is used as helper table for previous task, because in the table **user_page_counts_by_date** we used **COUNTER a**nd it is impossible to store **TEXT** values and **COUNTER** values at the same table, but we need to give the **user_name -** so we created table **users**

Purpose: stores the **mapping** user_id and **user_name**.

```
CREATE TABLE IF NOT EXISTS users (
    user_id BIGINT PRIMARY KEY,
    user_name TEXT);
```

- **"Return aggregated statistics per domain per hour" - domain_stats_by_hour**

Purpose: aggregate statistics - the number of pages created for each domain per hour.

```
CREATE TABLE IF NOT EXISTS domain_stats_by_hour (
    time_start TIMESTAMP,
    domain TEXT,
    page_count INT,
    PRIMARY KEY ((time_start), domain)
);
```

- **"Return statistics about pages created by bots" - bot_stats_last_6h**

Purpose: stores the number of pages created by bots in the last 6 hours.

```
CREATE TABLE IF NOT EXISTS bot_stats_last_6h (
    time_start TIMESTAMP,
    domain TEXT,
    created_by_bots INT,
    PRIMARY KEY ((time_start), domain)
);
```

- **"Return Top 20 users that created the most pages" - top_users_last_6h**

Purpose: stores the top 20 users who have created the most pages in the last 6 hours.

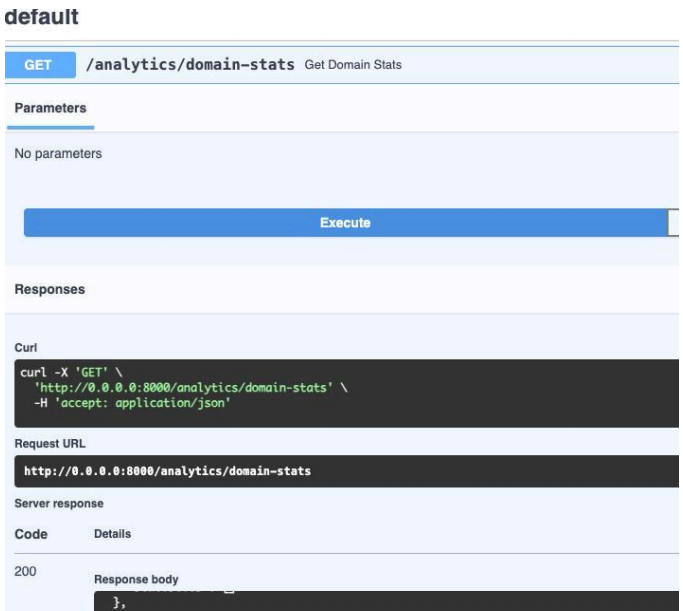
```
CREATE TABLE IF NOT EXISTS top_users_last_6h (
    time_start TIMESTAMP,
    user_id BIGINT,
    user_name TEXT,
    page_count INT,
    page_titles LIST<TEXT>,
    PRIMARY KEY ((time_start), page_count, user_id)
) WITH CLUSTERING ORDER BY (page_count DESC);
```

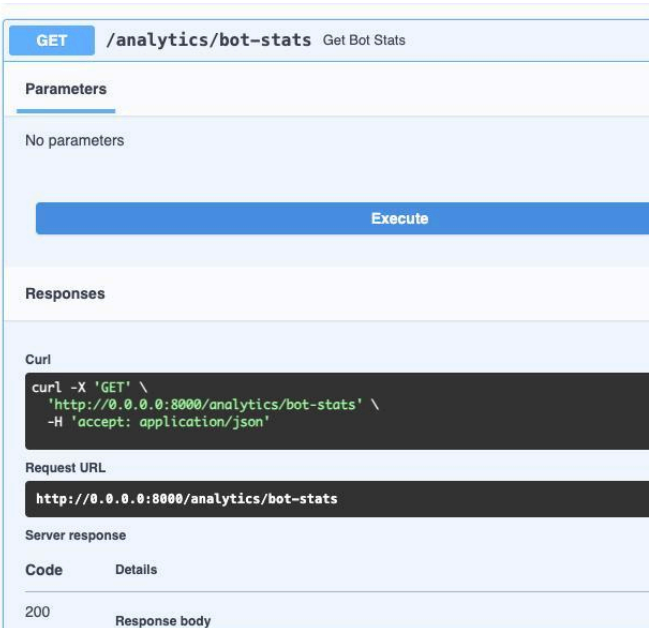
Results

Category A:

- **/analytics/domain-stats**

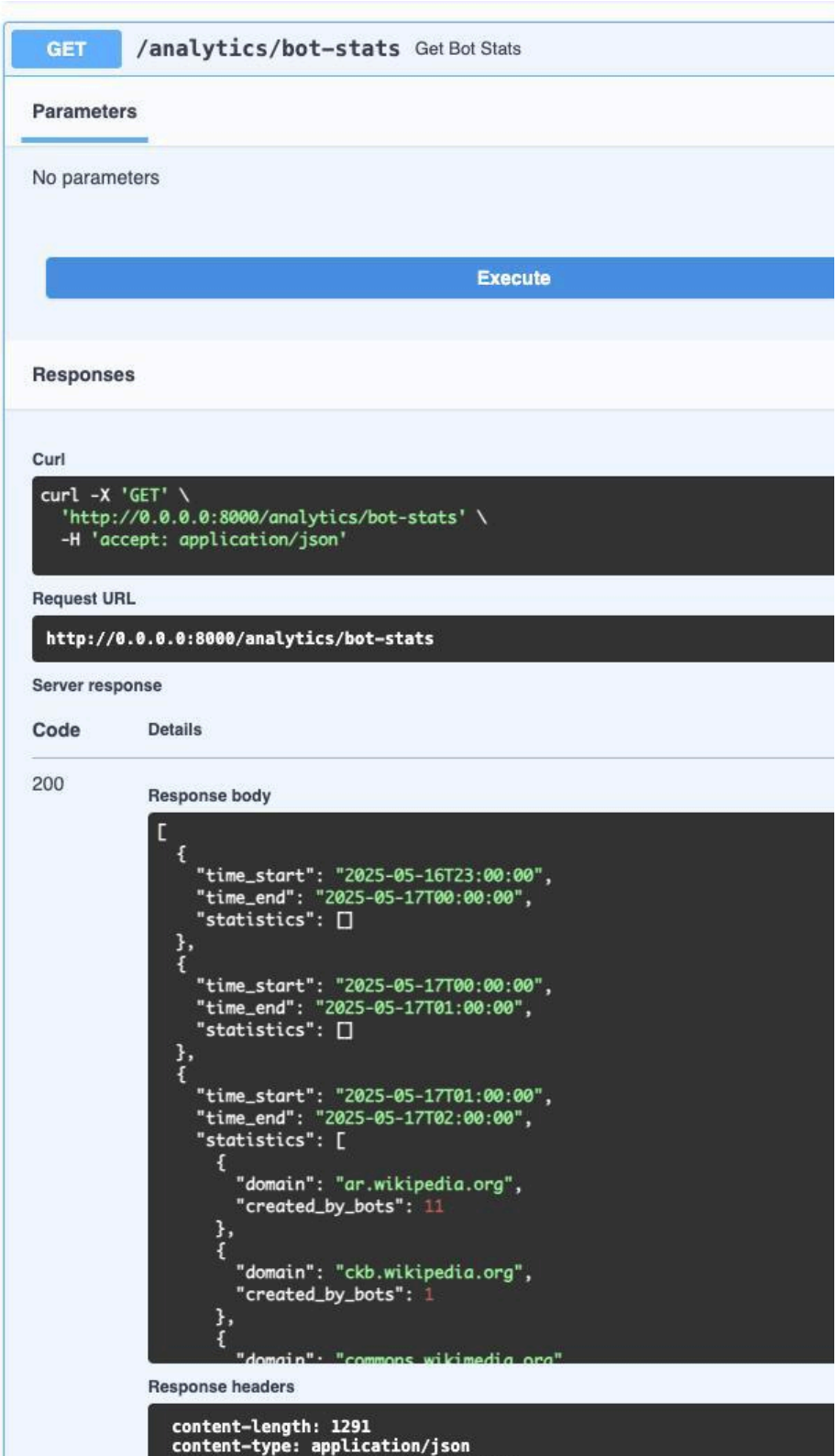
  Returns a list of domain-level page creation statistics for each of the last 6 hours (excluding the current hour).





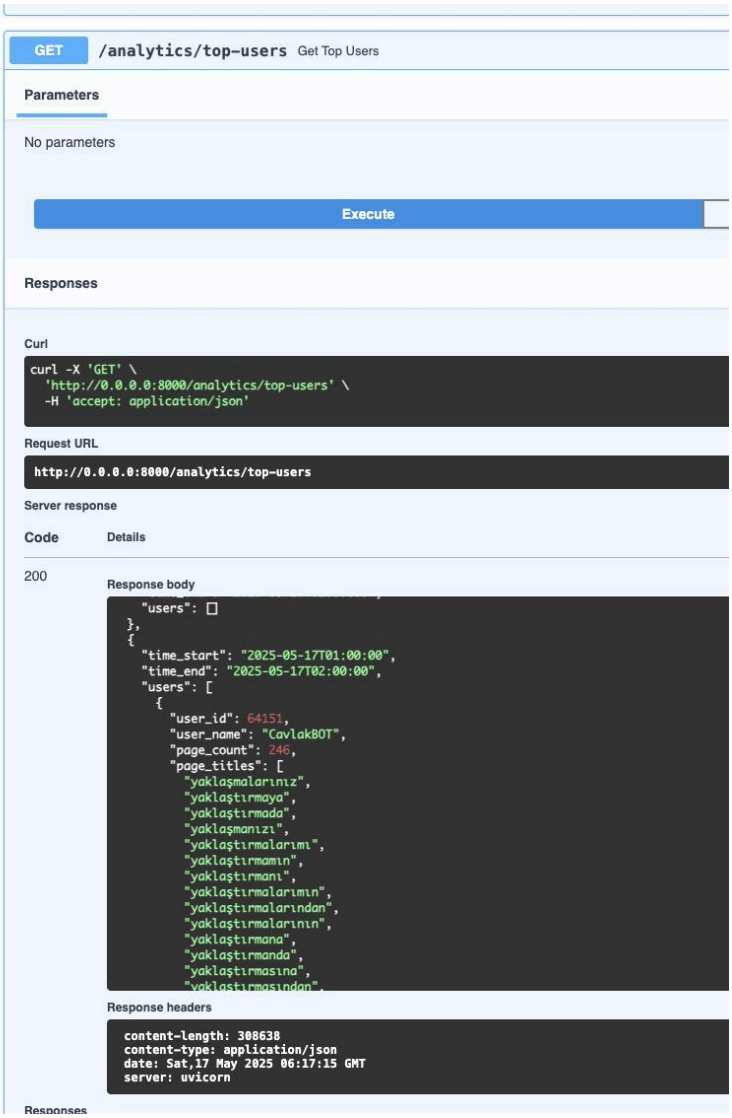
- **/analytics/bot-stats**

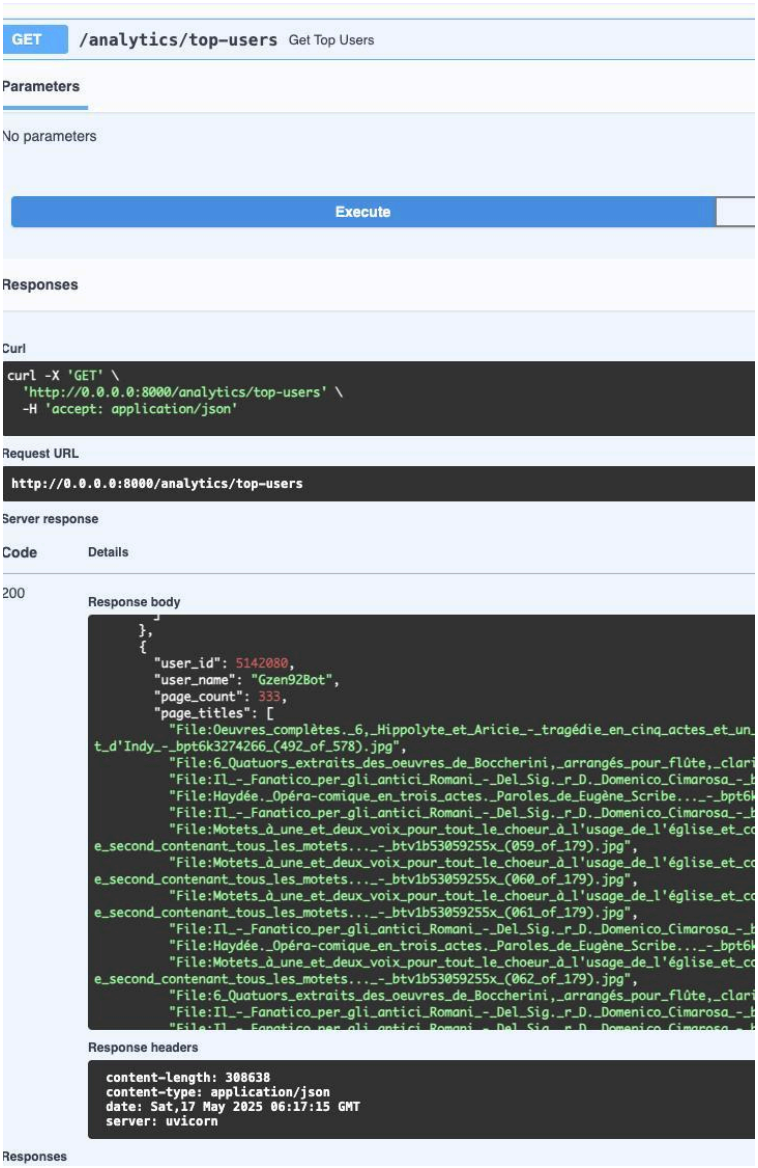
  Returns hourly statistics on the number of pages created by bots per domain for the last 6 hours.



- **/analytics/top-users**

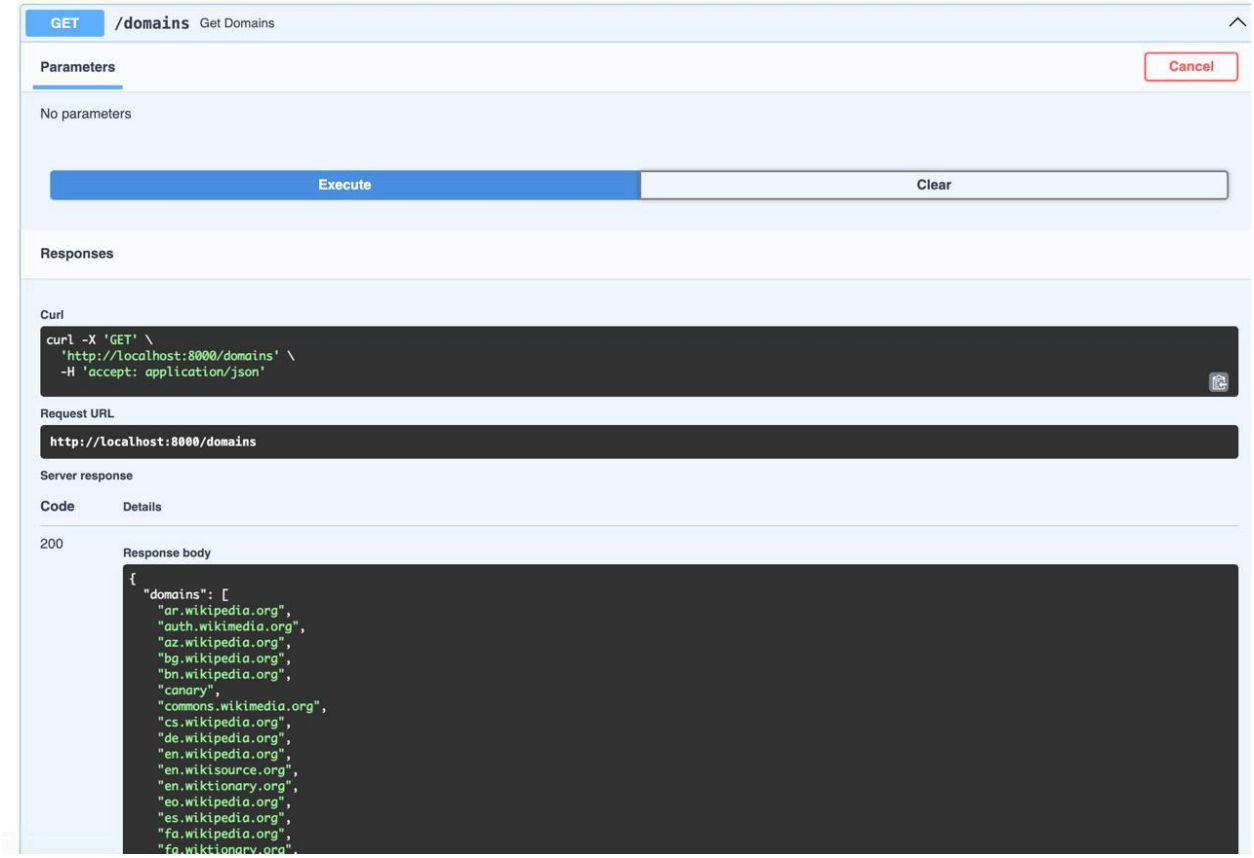
- **Returns the top 20 users by page creation count for each of the last 6 hours, along with page titles and user info.**





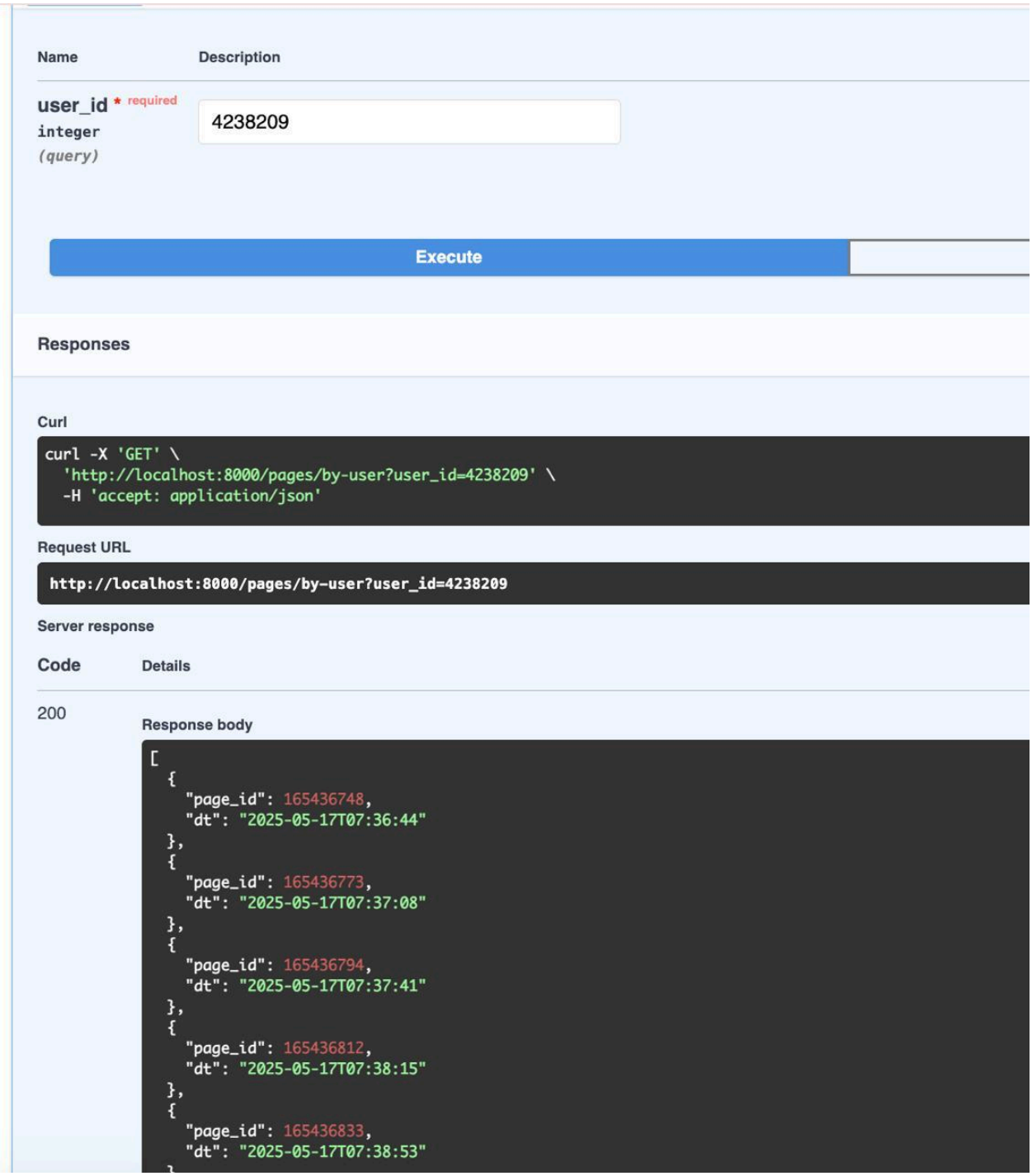
Category B:

- **/domains**



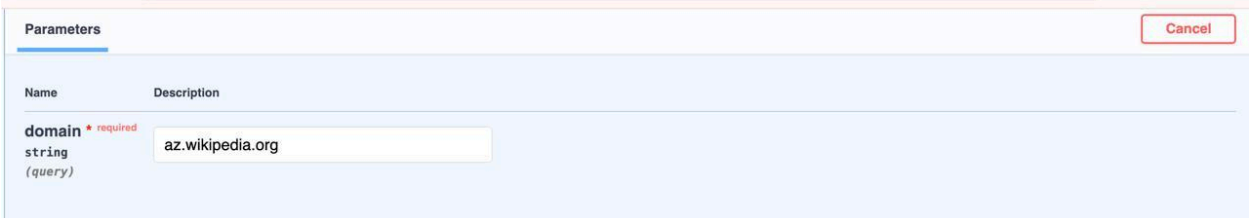
Returns list of unique text domains!

- **/pages/by-user**



Returns a list of pages (with timestamps) created by a specific user.

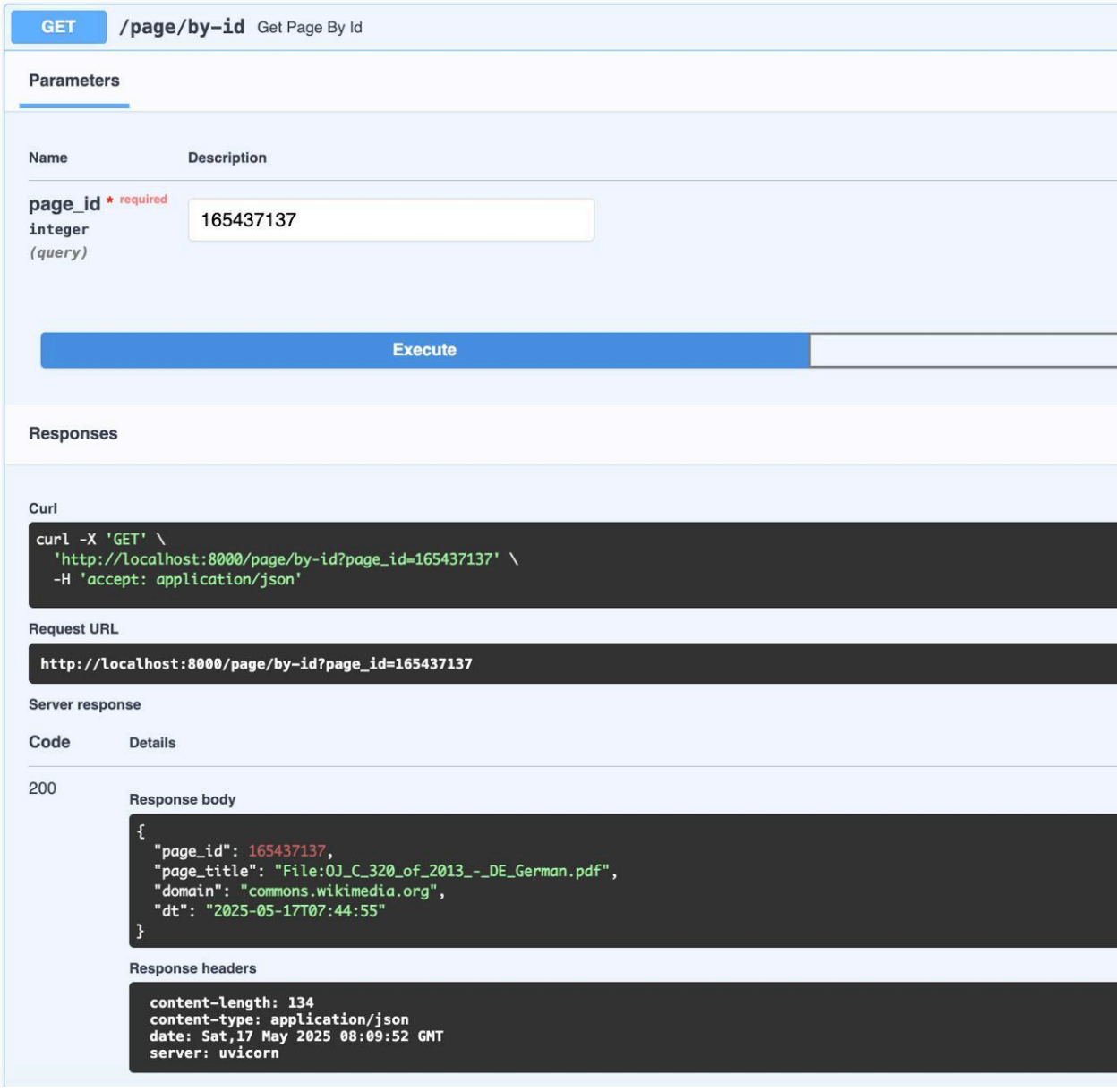
- **/domain/count**



Returns the total number of pages created for a specified domain.

- **/page/by-id**

Returns detailed information about a page with the specified ID, including title, domain, and creation time.



- **/users/activity**

Returns the total number of pages created by each user within a given date range, sorted by activity.