1. `config.py`:

- Contains all configuration settings for the project, including API keys, database credentials, and other constants.

2. `bot\_main.py`:

- The core of your Telegram bot. It handles user interactions, processes commands, and coordinates between weather and event APIs.

3. `weather\_api.py`:

- Interfaces with the weather API to fetch forecast data for specified locations and date ranges.

4. `events\_api.py`:

- Communicates with the events API to retrieve event information for given locations and dates.

5. `kafka\_producer.py`:

- Sets up the Kafka producer and provides a function to send messages to Kafka topics.

6. `mariadb\_consumer.py`:

- Consumes messages from the Kafka 'mariadb-messages' topic and inserts them into the MariaDB database.

7. `minio\_consumer.py`:

- Listens to the Kafka 'minio-messages' topic and stores received messages as objects in MinIO.

8. `database\_setup.py`:

- Creates the necessary table in the MariaDB database for storing bot messages.

9. `minio\_setup.py`:

- Sets up the required bucket in MinIO for storing bot message objects.

10. `kafka\_setup.py`:

- Creates the required Kafka topics for the bot's message handling system.

11. `init\_all.py`:

- Orchestrates the entire setup process and launches all necessary components of the bot system.

12. `top\_cities.py`:

- Contains a pandas DataFrame (`df\_cities`) that stores information about all the cities supported by the bot. This includes details such as city names, countries, and potentially other relevant information like coordinates or population. The bot uses this data to validate user inputs and provide a list of supported cities.

These files collectively create a Telegram bot that provides weather forecasts and event information for a predefined list of cities. The bot logs all interactions across different storage systems (MariaDB and MinIO) using Kafka as a message broker. The `top\_cities.py` file ensures that the bot only processes requests for supported cities, enhancing the user experience and data consistency.

1. The Kafka producer is not started in a separate file like the consumers are. Instead, it's integrated directly into the main bot code.

2. In the `bot\_main.py` file, the Kafka producer is initialized and used. Specifically:

3. The reason for this difference in approach between producers and consumers is due to how they're typically used:

- Producers are often tightly integrated with the application logic, sending messages as part of the main program flow.

- Consumers usually run as separate processes, continuously listening for new messages.

4. the main bot process acts as the producer, sending messages to Kafka topics whenever it needs to log something. The separate consumer processes (for MariaDB and MinIO) then pick up these messages and process them accordingly.

the producer is "started" (or more accurately, initialized and used) within the main bot script (`bot\_main.py`). It doesn't need a separate process like the consumers do, because it's part of the main application flow.

Producers:

1. There is one producer, which is part of the main bot application (`bot\_main.py`).

Consumers:

1. MariaDB Consumer (`mariadb\_consumer.py`)

2. MinIO Consumer (`minio\_consumer.py`)

Topics:

1. "bot-messages"

2. "mariadb-messages"

3. "minio-messages"

Logic:

1. The single producer in `bot\_main.py` sends messages to all three topics every time a message needs to be logged.

2. The MariaDB consumer listens only to the "mariadb-messages" topic and writes the messages to the MariaDB database.

3. The MinIO consumer listens only to the "minio-messages" topic and stores the messages as objects in MinIO.

4. The "bot-messages" topic is currently not being consumed, but it could be used for general logging or future extensions.

Flow:

1. When the bot sends or receives a message, it calls the `log\_message` function.

2. This function sends the message data to all three Kafka topics using the single producer.

3. The MariaDB consumer picks up messages from its topic and stores them in the database.

4. The MinIO consumer picks up messages from its topic and stores them as objects.

This setup allows for decoupled, scalable processing of bot messages. Each consumer can work independently, and you can add more consumers or topics in the future without changing the core bot logic.

