Binary Data Stream Processing and Visualization

# Prompt

## **Overview**

Your task is to design and implement a system that processes a binary data stream, extracts meaningful parameters, stores them, and provides a web-based interface for users to visualize the data. The project involves the following components:

1. **Binary Data Stream Processing:**
   * You’ll receive a continuous binary data stream (e.g., 11110101011100110) from a producer system (source).
   * The stream contains a header (e.g., 1111) followed by data bits.
   * Your system must:
     + Identify the header and ignore any bits before it.
     + Extract the subsequent bits for each parameter (e.g., A, B).
     + Convert those bits to decimal values.
     + Store the parameter values (e.g., in a database).
2. **Data Storage:**
   * Use MongoDB (or any other suitable database) to store the parameter values.
   * Design a schema to represent the data (e.g., separate collections for A, B, etc.).
   * Insert the converted values into the appropriate collections.
3. **Web Interface:**
   * Develop a web-based interface using HTML, CSS, and JavaScript (JS):
     + Implement user authentication (login functionality).
     + Allow users to select parameters (e.g., A, B, C) they want to visualize.
     + Provide a form or UI elements for user input.
     + Communicate with the backend (Python) to retrieve data.
     + Display the selected data points on the web page.
4. **Data Visualization:**
   * Use JavaScript (JS) to:
     + Dynamically update the plot based on user selections.
     + Retrieve data from MongoDB (via the Python backend) for the selected parameters.
   * Consider using charting libraries (e.g., Chart.js, D3.js) to create line plots.
   * Plot the data points corresponding to the selected parameters.
5. **Real-Time Updates:**
   * Since this is a real-time streaming application:
     + Use WebSocket connections to receive updates from the Kafka topic (if applicable).
     + Periodically fetch new data from the Kafka topic (if using Kafka) and update the database.
     + Update the web interface in real time as new data arrives.

## **Requirements**

* Set up an Apache Kafka topic for data streaming (if not already done).
* Write Python code to parse the binary stream and extract parameter values.
* Implement user authentication in the web interface.
* Create a visually appealing line plot for the selected parameters.
* Ensure error handling for invalid data and missing headers.

## **Deliverables**

1. **Backend (Python):**
   * Binary stream parser.
   * Connection to MongoDB.
   * Real-time data updates (if applicable).
2. **Frontend (HTML, CSS, JS):**
   * Login page.
   * Parameter selection form.
   * Dynamic data visualization.
3. **Documentation:**
   * Detailed instructions for setting up and running the system.
   * Explanation of design choices and trade-offs.

## **Bonus Features (Optional)**

* Explore Kafka Streams or Kafka Connect for additional stream processing capabilities.
* Enhance the visualization with interactive features (zoom, tooltips, etc.).
* Optimize the data storage and retrieval process.

## **Notes**

* Keep the system modular and well-documented.
* Test thoroughly to ensure correctness and robustness.
* Consider scalability and performance aspects.

# Procedure

### **Stage 1: Setup and Data Stream Processing**

#### **Objective 1.1: Kafka Topic Setup**

* **Tools Required**:
  + Apache Kafka
  + Kafka command-line tools (e.g., kafka-topics, kafka-console-producer)
* **Process**:
  1. Install and configure Apache Kafka.
  2. Create a Kafka topic for data streaming.
  3. Set up the producer system to publish binary data to the topic.
  4. Verify that data is being produced and available in the topic.

#### **Objective 1.2: Binary Stream Parsing**

* **Tools Required**:
  + Python
  + Kafka Python library (e.g., confluent-kafka-python)
* **Process**:
  1. Write Python code to consume data from the Kafka topic.
  2. Parse the binary stream:
     + Identify the header (e.g., 1111).
     + Extract subsequent bits for each parameter (e.g., A, B).
     + Convert bits to decimal values.
  3. Store the parameter values (e.g., in MongoDB).

### **Stage 2: Web Interface Development**

#### **Objective 2.1: User Authentication**

* **Tools Required**:
  + HTML, CSS
  + JavaScript (for form validation)
* **Process**:
  1. Create a login page with input fields for username and password.
  2. Implement basic authentication (e.g., compare credentials with hardcoded values for now).

#### **Objective 2.2: Parameter Selection Form**

* **Tools Required**:
  + HTML (form elements)
  + JavaScript (for form interaction)
* **Process**:
  1. Design a form where users can select parameters (A, B, C).
  2. Use JavaScript to handle form submission and user selections.

### **Stage 3: Data Visualization**

#### **Objective 3.1: Data Retrieval**

* **Tools Required**:
  + Python (backend)
  + MongoDB (database)
* **Process**:
  1. Retrieve parameter values from the database based on user selections.
  2. Prepare the data for visualization.

#### **Objective 3.2: Line Plot Creation**

* **Tools Required**:
  + JavaScript (frontend)
  + Charting library (e.g., Chart.js)
* **Process**:
  1. Dynamically create a line plot based on the selected parameters.
  2. Display the plot on the web page.

### **Stage 4: Real-Time Updates**

#### **Objective 4.1: WebSocket Integration**

* **Tools Required**:
  + JavaScript (frontend)
  + WebSocket library (e.g., Socket.io)
* **Process**:
  1. Set up WebSocket connections to receive real-time updates from Kafka (if applicable).
  2. Update the plot as new data arrives.

## Iteration 1

### UML Diagrams

#### Class Diagram

Certainly! Let’s create a textual representation of the UML class diagram for **Stage 1** of your project. I’ll include the relationships, their respective types, attributes, and operations.

### **UML Class Diagram for Stage 1**

**Classes**:

* KafkaTopic:
  + Attributes:
    - topicName: String
    - partitions: int
    - replicationFactor: int
  + Operations:
    - createTopic()
    - getTopicName(): String
    - getPartitions(): int
    - getReplicationFactor(): int
* BinaryStreamProcessor:
  + Attributes:
    - inputTopic: KafkaTopic
    - outputDatabase: MongoDB
  + Operations:
    - processBinaryStream()
    - extractHeader()
    - extractParameterBits()
    - convertToDecimal()
    - storeParameterValues()

**Relationships and Types**:

a. **Dependency**:

* Between BinaryStreamProcessor and KafkaTopic.
* Indicates that BinaryStreamProcessor depends on KafkaTopic (i.e., it uses the Kafka topic for data input).

b. **Association**:

* Between BinaryStreamProcessor and KafkaTopic.
* Represents a general relationship between the two classes.
* Can be further specified as:
  + **Aggregation**:
    - Indicates that BinaryStreamProcessor has an aggregation relationship with KafkaTopic.
    - The whole (BinaryStreamProcessor) contains parts (KafkaTopic).
    - The lifetime of the whole and parts is consistent.
  + **Composition**:
    - Indicates that BinaryStreamProcessor has a composition relationship with KafkaTopic.
    - The whole (BinaryStreamProcessor) contains parts (KafkaTopic).
    - The parts cannot exist independently from the whole.

c. **Generalization**:

* Between BinaryStreamProcessor and an abstract class (e.g., StreamProcessor).
* Represents an inheritance relationship.
* BinaryStreamProcessor inherits from the abstract class.
* The abstract class defines common methods or attributes related to stream processing.

d. **Realization**:

* Between BinaryStreamProcessor and an interface (e.g., StreamInterface).
* Indicates that BinaryStreamProcessor realizes the methods defined in the interface.
* The interface provides a contract for stream processing methods.

**Summary**:

* The UML class diagram captures the relationships between KafkaTopic and BinaryStreamProcessor, including dependency, association (aggregation or composition), generalization, and realization.
* Attributes and operations are included to provide a more detailed view of the classes.

#### Sequence Diagram

**Scenario**: Setting up the Kafka topic and processing binary data.

**Participants (Lifelines)**:

* User: Represents the user or external actor.
* KafkaTopic: Represents the Kafka topic for data streaming.
* BinaryStreamProcessor: Represents the component responsible for processing binary data.

**Interactions**:

* User requests the creation of a Kafka topic and KafkaTopic creates the topic (message: “Create topic”).
  + **Function Name**: createTopic()
  + **Description**: The KafkaTopic class creates the specified topic.
* User provides binary data (message: “Provide binary data”).
  + **Function Name**: provideBinaryData()
  + **Description**: The user provides binary data to be processed.
* BinaryStreamProcessor consumes data from KafkaTopic (message: “Consume data”).
  + **Function Name**: consumeData()
  + **Description**: The BinaryStreamProcessor class consumes data from the Kafka topic.
* BinaryStreamProcessor parses the binary stream (messages: “Parse header,” “Extract parameter bits,” “Convert to decimal”).
  + **Function Name**: parseBinaryStream()
  + **Description**: The BinaryStreamProcessor class identifies the header, extracts parameter bits, and converts them to decimal values.
  + **Interactions**:
    - extractHeader()
    - extractParameterBits()
    - convertToDecimal()
* BinaryStreamProcessor stores parameter values in MongoDB (message: “Store parameter values”).
  + **Function Name**: storeParameterValues()
  + **Description**: The BinaryStreamProcessor class stores the parameter values (e.g., A, B) in a suitable data store (e.g., MongoDB).

**Sequence Diagram Notation**:

* Actors: User (external), KafkaTopic (internal), BinaryStreamProcessor (internal).
* Lifelines: User, KafkaTopic, BinaryStreamProcessor.
* Messages: Create message, Delete message, Self message, Reply message, Found message, Lost message.
* Lifeline head (rectangle) with instance name and type.

#### Deployment Diagram

**Nodes**:

* User Workstation:
  + Represents the user’s local machine (e.g., laptop, desktop).
  + Deployed on the user’s workstation.
* Kafka Server:
  + Represents the server hosting the Kafka broker.
  + Deployed on a dedicated server or cloud instance.
* Database Server:
  + Represents the server hosting the MongoDB database.
  + Deployed on a separate server or cloud instance.

**Components**:

* BinaryStreamProcessor:
  + Represents the software module responsible for processing binary data.
  + Deployed on the User Workstation.
* KafkaTopic:
  + Represents the Kafka topic for data streaming.
  + Deployed on the Kafka Server.
* MongoDB:
  + Represents the MongoDB database.
  + Deployed on the Database Server.

**Dependencies and Associations**:

* BinaryStreamProcessor depends on the KafkaTopic for data input.
* BinaryStreamProcessor communicates with the MongoDB to store parameter values.

**Communication Paths**:

* Communication channels (e.g., network connections, protocols) connect the components to their respective nodes.

**Deployment Specification**:

* Specify hardware configurations, communication protocols, and other properties for each node and component.

**Deployment Locations:**

* User Workstation: Deploy the BinaryStreamProcessor here.
* Kafka Server: Deploy the KafkaTopic.

### Folder Structure

**Folder Structure**:

main/  
├── kafka/  
│ ├── kafka\_config.py  
│ └── kafka\_producer.py  
├── binary\_stream\_processor/  
│ ├── binary\_stream\_parser.py  
│ └── parameter\_storage.py  
├── database/  
│ ├── mongodb\_config.py  
│ └── mongodb\_connector.py  
└── README.md

**File Descriptions**:

* kafka/:
  + This folder contains files related to Kafka setup and data streaming.
  + kafka\_config.py:
    - Configuration settings for connecting to the Kafka broker (e.g., broker address, topic name).
    - Example content:KAFKA\_BROKER = "localhost:9092"  
      KAFKA\_TOPIC = "my\_data\_stream"
  + kafka\_producer.py:
    - Python script that produces binary data to the Kafka topic.
    - Example content (simplified):from kafka import KafkaProducer  
      from kafka\_config import KAFKA\_BROKER, KAFKA\_TOPIC  
        
      def produce\_binary\_data(data):  
       producer = KafkaProducer(bootstrap\_servers=KAFKA\_BROKER)  
       producer.send(KAFKA\_TOPIC, value=data.encode())  
       producer.close()  
        
      if \_\_name\_\_ == "\_\_main\_\_":  
       binary\_data = "11110101011100110" # Example binary data  
       produce\_binary\_data(binary\_data)
* binary\_stream\_processor/:
  + This folder contains files related to processing the binary stream.
  + binary\_stream\_parser.py:
    - Python script that parses the binary stream (extracts header, parameter bits, and converts to decimal).
    - Example content (simplified):def extract\_header(binary\_data):  
       # Extract header logic  
       pass  
        
      def extract\_parameter\_bits(binary\_data):  
       # Extract parameter bits logic  
       pass  
        
      def convert\_to\_decimal(bits):  
       # Conversion logic  
       pass  
        
      if \_\_name\_\_ == "\_\_main\_\_":  
       binary\_data = "11110101011100110" # Example binary data  
       header = extract\_header(binary\_data)  
       parameter\_bits = extract\_parameter\_bits(binary\_data)  
       decimal\_values = [convert\_to\_decimal(bits) for bits in parameter\_bits]  
       print("Decimal values:", decimal\_values)
  + parameter\_storage.py:
    - Python script responsible for storing parameter values (e.g., in MongoDB).
    - Example content (simplified):from mongodb\_connector import connect\_to\_mongodb  
        
      def store\_parameter\_values(parameter\_values):  
       # Store parameter values in MongoDB  
       pass  
        
      if \_\_name\_\_ == "\_\_main\_\_":  
       parameter\_values = [42, 73, 125] # Example parameter values  
       db = connect\_to\_mongodb()  
       store\_parameter\_values(parameter\_values)
* database/:
  + This folder contains files related to MongoDB setup and interaction.
  + mongodb\_config.py:
    - Configuration settings for connecting to MongoDB (e.g., database URL, credentials).
    - Example content:MONGODB\_URL = "mongodb://localhost:27017/"  
      DATABASE\_NAME = "my\_data\_db"
  + mongodb\_connector.py:
    - Python script that establishes a connection to MongoDB.
    - Example content (simplified):from pymongo import MongoClient  
      from mongodb\_config import MONGODB\_URL, DATABASE\_NAME  
        
      def connect\_to\_mongodb():  
       client = MongoClient(MONGODB\_URL)  
       db = client[DATABASE\_NAME]  
       return db  
        
      if \_\_name\_\_ == "\_\_main\_\_":  
       db\_connection = connect\_to\_mongodb()  
       print("Connected to MongoDB:", db\_connection)
* README.md:
  + A README file with instructions on how to set up and run your Stage 1 components.
  + Include details about installing dependencies, configuring Kafka, MongoDB, and executing the scripts.