`

IOT PLATFORM -DESIGN DOCUMENT & RUN INSTRUCTIONS

Abstract

This document presents the architecture and implementation details of a scalable IoT data processing prototype designed to ingest, process, and analyze data from over 100,000 simulated IoT devices. The system demonstrates a distributed event-driven architecture utilizing Apache Kafka as a message broker to decouple data ingestion from processing, ensuring high throughput and reliability.

The Ingest Service, built with Spring Boot, receives JSON payloads from simulated IoT devices via REST API and publishes them to a Kafka topic. The Processor Service, also developed in Spring Boot, consumes these messages asynchronously, processes the sensor data, and persists it in MongoDB for further analysis.

This prototype showcases how modern microservice-based architectures can efficiently handle large-scale IoT data streams in real time without bottlenecks. It provides a foundation for extending into analytics, monitoring, and alerting systems for smart IoT infrastructures.

Contents

List of figures

1- Overview & Goals

Objective

Design a scalable, fault-tolerant pipeline that can ingest, process and analyze telemetry from >100,000 IoT devices, implemented as a prototype in Java (Spring Boot) with Kafka, MongoDB and Docker.

High-level goals

- Decouple ingestion from processing (Kafka).
- Handle very high concurrency via non-blocking client (WebFlux simulator) and partitioned consumers.
- Persist messages for analytics and querying (MongoDB).
- Provide reproducible local setup using Docker Compose; support running services locally or connecting to MongoDB Atlas.

2- System Architecture (Components)

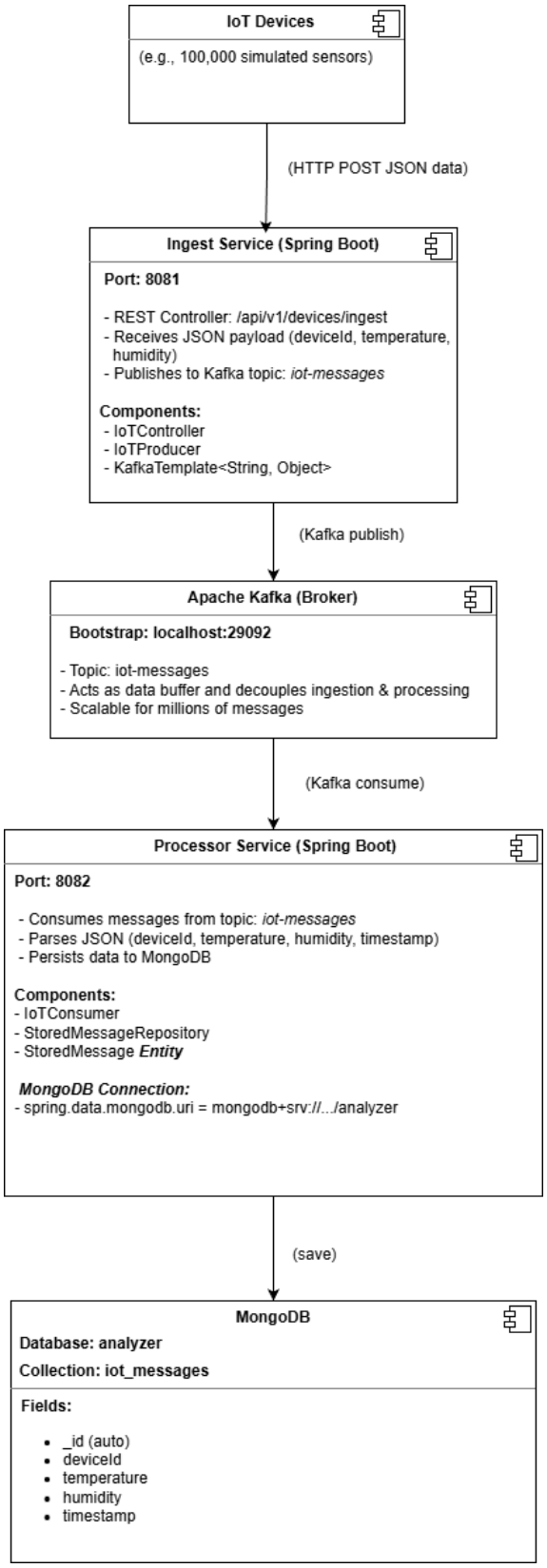

Figure 2. 1: System Architecture

Sequence:

- Device → HTTP POST → Ingest service.
- Ingest service serializes message and publishes to Kafka (iot-messages topic).
- Processor service consumes messages (consumer group), processes them and saves to MongoDB.
- Analytics layer queries MongoDB

3- Technology choices & reasoning

Java 17 + Spring Boot (3.x)

- Mature ecosystem, Spring Kafka and Spring Data MongoDB integrations.

Apache Kafka

- Durable, partitioned, highly available message bus that decouples producers from consumers and scales horizontally (partitions + consumer groups).

MongoDB (Atlas or local)

- Flexible schema suited to time-series-like IoT records; easy to query for analytics dashboards.

Docker Compose

- Quick reproducible environment for Kafka + Zookeeper + MongoDB for local testing.

Spring WebFlux (WebClient) for simulator

- Non-blocking, highly concurrent client for simulating many devices without thread explosion.

JSON over HTTP

- Simplicity for device-to-ingest prototype.

Spring Kafka JSON SerDe

- Use Spring's JsonSerializer/JsonDeserializer to send POJOs via Kafka.

4- Data model

StoredMessage (Mongo document)

```java
@Document(collection = "iot_messages")  5 usages  lakmalasela
public class StoredMessage {
    @Id  2 usages
    private String id;
    private String deviceId;  2 usages
    private long timestamp;  2 usages
    private double temperature;  2 usages
    private double humidity;  2 usages

    // getters and setters
    public String getId() { return id; }  no usages  lakmalasela
    public void setId(String id) { this.id = id; }  no usages  lakmalasela
    public String getDeviceId() { return deviceId; }  1 usage  lakmalasela
    public void setDeviceId(String deviceId) { this.deviceId = deviceId; }  1 usage  lakmalasela
    public long getTimestamp() { return timestamp; }  no usages  lakmalasela
    public void setTimestamp(long timestamp) { this.timestamp = timestamp; }  1 usage  lakmalasela
    public double getTemperature() { return temperature; }  no usages  lakmalasela
    public void setTemperature(double temperature) { this.temperature = temperature; }  1 usage  lakmalasela
    public double getHumidity() { return humidity; }  no usages  lakmalasela
    public void setHumidity(double humidity) { this.humidity = humidity; }  1 usage  lakmalasela
}
```

Figure 4. 1: Mongo document

JSON example:

```json
{
  "deviceId": "sensor-123",
  "timestamp": 1730000000000,
  "temperature": 27.34,
  "humidity": 55
}
```

Figure 4. 2: JSON Payload

5- Important design details & decisions

Decoupling and elasticity

- Kafka topic partitioning is the primary scaling knob for consumers: more partitions -> more parallelism.
- In production, use multiple processor instances (same consumer group) to process partitions in parallel.

Producer acknowledgement and reliability

- Configure producer acks=all in production to avoid data loss; for prototype default settings are acceptable.

Consumer semantics

- Consumers should be idempotent or de-duplicate if devices can resend messages (use unique IDs or deviceId+timestamp).

Backpressure / bursting

- Use client-side throttling or concurrency limits in simulator.
- Use Kafka retention and topic sizes to absorb bursts.

Persistence & schema

- Store raw message and optionally derived metrics (e.g., moving average) to speed analytics.
- Index fields frequently queried: deviceId, timestamp.

Security

- For production, enable TLS for Kafka and MongoDB, secure credentials via secrets manager (not plain properties).
- For MongoDB Atlas, ensure IP whitelist and proper user credentials.

Observability

- Add metrics (Micrometer → Prometheus), logging and tracing (OpenTelemetry) to track throughput, lags, and errors.
- Monitor Kafka consumer lag to ensure processing keeps up.

Error handling & DLQ

- Configure consumer error handling and Dead-Letter Queue (DLQ) for messages that repeatedly fail processing.

6- Kafka tuning recommendations (prototype → production)

- Topic partitions: set partitions >= expected parallel consumers. For 100k+ devices you may start with 12–50 partitions and adjust.
- Replication factor: ≥2 (or 3) in production.
- Producer batching: tune linger.ms and batch.size.
- Throughput: use partition keys (e.g., deviceId) to control ordering and distribution.
- Use Kafka Connect or Kafka Streams for advanced stream processing if needed.

7- Local prototype: File layout (example)

```
iot-platform/                          (root, multi-module maven)
  pom.xml                              (parent)
  ingest-service/
    pom.xml
    src/main/java/com/example/iot/ingest/
      controller/DeviceController.java
      kafka/IoTProducer.java
      model/IoTMessage.java
      simulator/IoTSimulatorWebFlux.java  (optional)
    src/main/resources/application.properties  (ingest config)
    Dockerfile
  processor-service/
    pom.xml
    src/main/java/com/example/iot/processor/
      kafka/IoTConsumer.java
      model/StoredMessage.java
      repository/StoredMessageRepository.java
    src/main/resources/application.properties  (processor config)
    Dockerfile
  docker-compose.yml
  README.md
```

Figure 7. 1: Local prototype: File layout

8- Build & run instructions (prototype)

Below are reproducible commands to build and run locally. Replace values where needed (Windows vs Unix notes included).

Prerequisites

- Java 21 JDK
- Maven 4.0
- Docker & Docker Compose
- (Optional) MongoDB Atlas account with username/password and network access allowed

8.1- Prepare config

ingest application.properties

(Place in ingest-service/src/main/resources/application.properties)

```
server.port=8081
spring.kafka.bootstrap-servers=localhost:29092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.properties.spring.json.trusted.packages=*
iot.topic=iot-messages
```

Figure 8. 1:ingest application.properties

processor application.properties

(Place in processor-service/src/main/resources/application.properties)

```
server.port=8082
spring.kafka.bootstrap-servers=localhost:29092
spring.kafka.consumer.group-id=processor-group
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.ErrorHandlingDeserializer
spring.kafka.consumer.properties.spring.deserializer.value.delegate.class=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.properties.spring.json.trusted.packages=*
spring.kafka.properties.spring.json.use.type.headers=false
spring.kafka.properties.spring.json.value.default.type=com.fasterxml.jackson.databind.JsonNode
spring.data.mongodb.uri=mongodb+srv://lakmalasela:9vnlOD16mXvTqrgx@cluster0.xstks4a.mongodb.net/iot-analyzer?retryWrites=true&w=majority
iot.topic=iot-messages
```

Figure 8. 2: processor application.properties

- If using local Docker Mongo, use spring.data.mongodb.uri=mongodb://localhost:27018/analyzer or host and port properties.
- If username contains special chars, URL-encode them (e.g., + → %2B).

8.2 -Start infrastructure (Docker Compose)

Create docker-compose.yml (project root) with Kafka + Zookeeper + local Mongo (example):

```yaml
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    ports:
      - "22181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181

  kafka:
    image: confluentinc/cp-kafka:7.4.1
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    ports:
      - "9092:9092"
      - "29092:29092"


  mongodb:
    image: mongo:6.0
    container_name: mongodb
    command: [ "mongod", "--port", "27018" ]
    ports:
      - "27018:27018"
    volumes:
      - mongo-data:/data/db


volumes:
  mongo-data:
```

Figure 8. 3: docker-compose.yml

Start containers: *docker-compose up -d*

Check containers: *docker ps*

8.3- Build services

From project root: mvn -T 1C clean package

or build per-module:

- cd ingest-service
- mvn clean package
- cd ../processor-service
- mvn clean package

8.4- Run services locally (IDE or CLI)

Run from IDE:

- Import modules into IntelliJ / Eclipse as Maven projects.
- Run IngestApplication and ProcessorApplication.

Or run with Maven:

Ingest:

- cd ingest-service
- mvn spring-boot:run

Processor:

- cd ../processor-service
- mvn spring-boot:run

Logs should show services starting and Kafka consumer registering.

8.5- Run reactive Java simulator (WebFlux)

Place simulator class in ingest-service (or separate module) under com.example.iot.ingest.simulator:

```
Flux.range( start: 1, NUM_DEVICES)
        .flatMap(deviceId -> {
            double temperature = ThreadLocalRandom.current().nextDouble( origin: 20,  bound: 40);
            int humidity = ThreadLocalRandom.current().nextInt( origin: 30,  bound: 90);
            long timestamp = System.currentTimeMillis();
            String json = String.format(
                    "{\"deviceId\":\"sensor-%d\",\"temperature\":%.2f,\"humidity\":%d,\"timestamp\":%d}",
                    deviceId, temperature, humidity, timestamp
            );

            return webClient.post() RequestBodyUriSpec
                    .header( headerName: "Content-Type",  ...headerValues: "application/json") RequestBodySpec
                    .bodyValue(json) RequestHeadersSpec<capture of ?>
                    .retrieve() ResponseSpec
                    .bodyToMono(String.class) Mono<String>
                    .doOnNext(response -> System.out.println("Device " + deviceId + " sent: " + response))
                    .doOnError(error -> System.err.println("Error sending device " + deviceId + ": " + error.getMessage()));
        }, concurrency: 100) // concurrency level, 100 requests at a time
        .blockLast(); // Wait until all requests are finished
```

Figure 8. 4: - Reactive Java simulator

Set NUM_DEVICES or concurrency modestly first (e.g., 5k) and ramp up to 100k to avoid saturating local resources.

8.6- Verify MongoDB saves

MongoDB Atlas: connect with MongoDB Compass or mongosh:

mongosh "mongodb+srv://cluster0.xstks4a.mongodb.net/analyzer" -u <USER> -p <PASS>

> use analyzer

> db.iot_messages.find().limit(5).pretty()

9- Troubleshooting checklist

- Consumer shows UnknownHostException: kafka → If running app locally, use localhost:29092 as bootstrap. Only use kafka:9092 when app runs inside Docker compose network.
- Kafka client errors about bootstrap.servers placeholders → Ensure application.properties keys are correct (no ${spring.kafka.bootstrap-servers} unresolved references).
- Mongo connect errors:
    - If Atlas: IP whitelist, credentials and URL encoding for special chars.
    - If local Docker: ensure container internal port (27017) mapped correctly (e.g., 27018:27017) and use localhost:27018 or mongodb://localhost:27018/analyzer.
- Messages not saving:
    - Confirm consumer receives messages (look at processor logs).
    - Confirm JSON includes timestamp and field names match StoredMessage.
    - Use POJO deserialization (map message to StoredMessage) instead of JsonNode where possible.
    - Check repository bean available and no exceptions printed when saving.
- Port already in use → Adjust ports or stop local service using same port.