kafka-api-wrapper — Comprehensive Project Documentation

1. Project summary

Name: kafka-api-wrapper

Description:
A Spring Boot–based microservice that exposes REST APIs for secure transaction processing using AES encryption, JWT authentication, and Kafka message streaming. The project is designed to be easily integrated into other applications — either as a standalone service, a Maven dependency, or a Dockerized component. **Core Features:** - REST endpoints for submitting and processing transaction payloads. - AES-256 encryption/decryption for data confidentiality. - JWT validation for authentication. - MD5 checksum verification for payload integrity. - Kafka producer and consumer for reliable message delivery. - Docker Compose for local Kafka + ZooKeeper setup.

---

2. Prerequisites

Before using orintegrating this project, ensure you have: - **Java:** 17 or higher - **Maven:** 3.x or higher - **Docker & Docker Compose:** for local Kafka/ZooKeeper setup - **Kafka knowledge:** basic understanding of producers, consumers, and topics - **Postman or curl:** for testing APIs

---

3. Quick Start (Local Setup)

Step 1: Start Kafka andZooKeeper

```
docker compose up -d
```

If port 9092 is already used, stop the process occupying it or modify `docker-compose.yml` to use a different host port (e.g., `9093:9092`).

Step 2: Build the Project

```
mvn clean package -DskipTests
```

Step 3: Run the Application

```
java -jar target/kafka-api-wrapper-1.0.0.jar
```

The service will start at `http://localhost:8086` .

Step 4: Test API Endpoint

Use Postman or curl to POST a sample transaction payload:

```
payload='{"Mobile":"8999999999","UserId":"123","Amount":"5000"}'
checksum=$(echo -n "$payload" | md5sum | awk '{print $1}')
curl -X POST http://localhost:8086/api/tx/send
  -H "Authorization: Bearer <JWT_TOKEN>"
  -H "Checksum: $checksum"
  -H "Content-Type: application/json"
  -d "$payload"
```

---

4. Project Structure Overview

```
kafka-api-wrapp/
├── pom.xml                                  # Maven configuration
├── docker-compose.yml                       # Kafka + ZooKeeper setup
├── src/main/java/com/example/kafkaap/wrapper/
│   ├── KafkaApiWrapperApplication.java      # Main Spring Boot entry point
│   ├── config/                              # Kafka producer/consumer
│   │   ├── KafkaConfig.java                 # Web CORS configuration
configuration
│   │   └── WebConfig.java
│   ├── controller/
│   │   ├── TransactionController.java       # Main REST controller
│   │   └── TransactionListenerController.java # Secondary controller for
message viewing/testing
│   ├── kafka/                               ├──
│   KafkaProducerService.java                │ └──  # Produces encrypted messages to Kafka
│   KafkaConsumerService.java                       # Consumes & decrypts messages from
Kafka
│   └── util/
│       ├── AESUtil.java                     # AES encryption/decryption utility #
│       ├── ChecksumUtil.java                MD5 checksum calculator/validator #
│       └── JwtUtil.java                     JWT validation helper
└── src/main/resources/application.properties
```

---

5. Configuration (application.properties)

```
server.port=8086
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=group_json
```

```
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-
serializer=org.apache.kafka.common.serialization.StringSerializer
app.encryption.secret=12345678901234567890123456789012
app.jwt.secret=ReplaceWithAStrongSecretKey
app.kafka.topic=transaction_topic
```

Note: Replace demo secrets before deploying.

---

6. System Workflow

1. **Client** sends transaction payload via REST API.
2. **Server** validates JWT token and MD5 checksum.
3. Payload is **encrypted using AES**.
4. Encrypted data is sent to **Kafka** using `KafkaProducerService`.
5. **KafkaConsumerService** receives messages, decrypts them, and logs/handles the result.

---

7. API Reference

Endpoint: `/api/tx/send`

Method: POST

Headers: - `Authorization: Bearer <JWT_TOKEN>` `Checksum: <MD5 of payload>` - `Content-Type: application/json`

Example Request:

```
{
  "Mobile":"8999999999",
  "UserId":"123",
  "Amount":"5000"
}
```

Response:

```
{
  "status":"success",
  "message":"Transaction sent successfully."
}
```

⚙️ Step 1: Go inside your jar-test/lib folder

Run:

cd lib

⚙️ Step 2: Extract your existing fat JAR
mkdir extracted
cd extracted
jar xf ../kafka-api-wrapper-1.0.0.jar

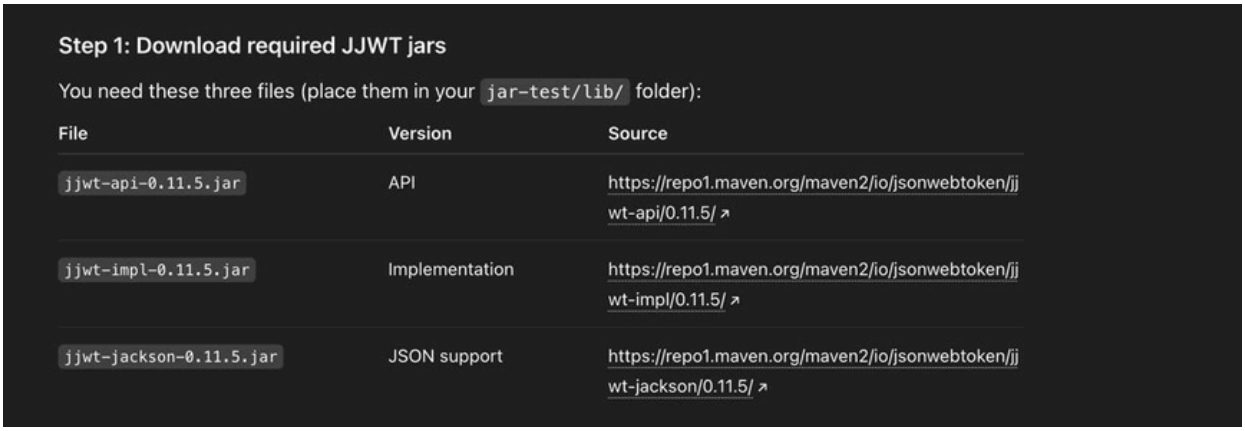
Now your folder will look like this:

extracted/
└── BOOT-INF/
    └── classes/
        └── com/example/kafkaap/wrapper/util/...

⚙️ Step 3: Repackage only the real classes into a new flat JAR
cd BOOT-INF/classes
jar cf ../../../kafka-api-wrapper-flat.jar com
cd ../../../

✅ You now have a new usable file:

lib/kafka-api-wrapper-flat.jart



Step 1: Download required JJWT jars

You need these three files (place them in your `jar-test/lib/` folder):

File	Version	Source
`jjwt-api-0.11.5.jar`	API	https://repo1.maven.org/maven2/io/jsonwebtoken/jjwt-api/0.11.5/ ↗
`jjwt-impl-0.11.5.jar`	Implementation	https://repo1.maven.org/maven2/io/jsonwebtoken/jjwt-impl/0.11.5/ ↗
`jjwt-jackson-0.11.5.jar`	JSON support	https://repo1.maven.org/maven2/io/jsonwebtoken/jjwt-jackson/0.11.5/ ↗

curl -O https://repo1.maven.org/maven2/io/jsonwebtoken/jjwt-api/0.11.5/jjwt-api-0.11.5.jar
curl -O https://repo1.maven.org/maven2/io/jsonwebtoken/jjwt-impl/0.11.5/jjwt-impl-0.11.5.jar
curl -O https://repo1.maven.org/maven2/io/jsonwebtoken/jjwt-jackson/0.11.5/jjwt-jackson-0.11.5.jar

javac -cp "lib/*" -d out src/TestJarMain.java
java -cp "lib/*:out" TestJarMain

8. Integrating kafka-api-wrapper into Other Projects

Thisprojectisdesignedforreusabilityandcanbeintegratedinmultiplewaysdepending on the target environment.

8.1. As a REST Microservice (Recommended)

Use Case: When your main project is in a different language or framework.

1. Deploy this project as a separate microservice.
2. Use HTTP requests from your main application to communicate.
3. Handle security by sharing the same JWT signing secret.
4. Configure network/firewall to allow communication between services.

Example Workflow: - Your main project (e.g., Node.js, .NET, or another Spring Boot service) calls `POST /api/tx/send`. - The wrapper handles encryption, validation, and Kafka message transmission.

Advantages: - Decoupled architecture. - Language-independent. - Can be scaled separately.

Disadvantages: - Requires a separate deployment and network calls.

8.2. As a Maven Dependency (Embed Directly)

Use Case: When both projects are Java-based.

1. Install kafka-api-wrapper locally:

```
mvn clean install -DskipTests
```

2. Add it as a dependency in your main project:

```
<dependency>
  <groupId>com.example.kafka</groupId>
  <artifactId>kafka-api-wrapper</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

3. Import and use the required classes directly:

```
import com.example.kafkaap.wrapper.util.AESUtil;
import com.example.kafkaap.wrapper.kafka.KafkaProducerService;

AESUtil aes = new AESUtil("12345678901234567890123456789012");
String  encrypted =  aes.encrypt("Confidential Data");

kafkaProducerService.sendMessage("transaction_topic", encrypted);
```

Advantages: - Direct class-level integration. - Simplified function calls.

Disadvantages: - Tight coupling. - Requires Java-based consumers.

8.3. As a Dockerized Service

Use Case: Ideal for containerized microservices.

1. Build Docker image:

```
docker build -t kafka-api-wrapper:1.0 .
```

2. Reference it in your main project's Docker Compose file:

```
services:
  kafka-api-wrapper:
    image: kafka-api-wrapper:1.0
    ports:
      - "8086:8086"
    environment:
      - SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka:9092
      - APP_ENCRYPTION_SECRET=${APP_ENCRYPTION_SECRET}
      - APP_JWT_SECRET=${APP_JWT_SECRET}
    depends_on:
      - kafka
```

3. Your main app communicates via REST with `kafka-api-wrapper` on `http://kafka-api-wrapper:8086` .

8.4. As a Git Submodule or Multi-Module Project

Use Case: When maintaining multiple related projects in a shared repository.

1. Add this project as a submodule:

```
git submodule add <repo-url> libs/kafka-api-wrapper
```

2. Update your root `pom.xml`:

```
<modules>
  <module>libs/kafka-api-wrapper</module>
  <module>main-project</module>
</modules>
```

3. Build both modules together:

```
mvn clean install
```

Advantages: Easy debugging and development synchronization.

Disadvantages: Slightly more complex repository management.

---

9. Configuration and Secrets Management

- Replace demo AES and JWT secrets before production use.
- Store secrets securely (e.g., environment variables, Vault, Docker secrets).
- Example environment variable usage:

```
export APP_ENCRYPTION_SECRET="your-32-byte-key"
export APP_JWT_SECRET="your-secure-jwt-key"
java -jar kafka-api-wrapper.jar
```

---

10. Security Recommendations

- Always use HTTPS in production.
- Prefer AES-GCM mode for authenticated encryption.
- Rotate encryption and JWT secrets regularly.
- Validate all incoming payloads against a schema.
- Use proper authentication between microservices.

---

11. Common Commands Recap

```
# Start Kafka/ZooKeeper
docker compose up -d

# Stop Docker services
docker compose down

# Build the project
mvn clean package

# Run application
java -jar target/kafka-api-wrapper-1.0.0.jar

# Run tests
mvn test
```

---

12. Summary

The **kafka-api-wrapper** is a secure, modular, and extensible Kafka-based API service that developers can integrate into their systems for encrypted and authenticated message streaming. Whether used as a RESTful microservice, Maven library, Docker container, or submodule, it provides flexibility and robust message handling for enterprise-grade applications.