Voith Test API

Author: Rui Carlos Lorenzetti da Silva LinkedIn: www.linkedin.com/in/ruilorenzetti Phone (WhatsApp): +55 19 9 8304 0440

Table of Contents

Introduction

The **Voith Test API** is a distributed microservices system designed to handle telemetry, errors, failures, maintenance, and machine data efficiently. It follows a **lambda architecture**, ensuring scalability and resilience.

The system is composed of three core microservices: - `microservice-api` - Exposes RESTful APIs for querying data. - `microservice-consumer` - Processes messages from Kafka. - `microservice-ingestion` - Handles data ingestion from various sources into InfluxDB.

Project Structure

```
voith_test_api/
│── microservice-api/         # Exposes REST API
│── microservice-consumer/    # Kafka consumer processing
│── microservice-ingestion/   # Data ingestion layer
│── DOCKER/                   # Docker Compose for services
│── README.md                 # Project documentation
│── pom.xml                   # Parent Maven configuration
```

Each microservice is built with: - **Java 21** - **Spring Boot 3.2.0** - **Maven 4.0 RC**

Installation and Setup

Prerequisites

- **Java 21**
- **Maven 4.0 RC**

- **Docker & Docker Compose**
- **Kafka**
- **InfluxDB**

Running the Project

1. Clone the repository: `bash git clone https://github.com/your-repo/voith_test_api.git cd voith_test_api`

2. Start the required services with Docker Compose: `bash cd DOCKER docker-compose up -d`

3. Build all microservices: `bash mvn clean install`

4. Run the microservices: `bash cd microservice-api mvn spring-boot:run` Repeat for `microservice-consumer` and `microservice-ingestion`.

Datasets and Their Role

The system processes different datasets, each serving a distinct purpose:

- **Telemetry Data**: Machine sensor readings (voltage, pressure, rotation, etc.).
- **Error Data**: Records machine errors.
- **Failure Data**: Logs failures for later analysis.
- **Maintenance Data**: Tracks scheduled and unscheduled maintenance activities.
- **Machine Data**: Static information about machines (model, age, status).

API Endpoints

Endpoint	Description	Example Request
`/api/telemetry`	Fetch telemetry data	`/api/telemetry?start=2025-02-03T14:35:00Z&end=2025-02-03T14:45:00Z`
`/api/errors`	Retrieve logged errors	`/api/errors?start=2025-02-03T14:35:00Z&end=2025-02-03T14:45:00Z`
`/api/failures`	Fetch failure records	`/api/failures?start=2025-02-03T14:35:00Z&end=2025-02-03T14:45:00Z`
`/api/maintenance`	Get maintenance events	`/api/maintenance?start=2025-02-03T14:35:00Z&end=2025-02-03T14:45:00Z`
`/api/machines`	Retrieve machine details	`/api/machines`

Architectural Decisions

- **Lambda Architecture**: Combines batch and real-time data processing.
- **Kafka Event-Driven Processing**: Ensures reliable and scalable data ingestion.
- **Spring Boot 3.2.0**: Modern, reactive, and cloud-native framework.
- **InfluxDB for Time-Series Data**: Optimized storage for telemetry readings.
- **Docker & Sidecar Pattern**: Kafka and InfluxDB run as sidecar services.

ADR 0001 - Adoption of Lambda Architecture

- **Date:** February 3, 2025
- **Author:** Rui Carlos Lorenzetti da Silva

Context

As I designed this system, I needed a robust architecture capable of handling high volumes of data while balancing real-time insights with historical analysis. The goal was to ensure low-latency data processing while still being able to run complex analytical queries on historical datasets. Given these requirements, I needed a solution that provided both **real-time stream processing** and **batch processing**.

Decision

I decided to adopt **Lambda Architecture**, which allows me to integrate both real-time and batch processing into the system. This approach ensures that I can process immediate data while also performing extensive computations on historical records.

Rationale

The choice of Lambda Architecture is based on its advantages in handling big data efficiently. Some of the key benefits I considered include:

- **Scalability –** The architecture is designed to process large-scale data efficiently.
- **Fault Tolerance –** By maintaining both batch and real-time processing paths, I can ensure system resilience even in case of failures.
- **Flexibility –** Lambda Architecture supports diverse processing needs, allowing real-time stream processing while maintaining the integrity of batch analytics.

These benefits align with my objective of delivering **fast insights from real-time data** while ensuring **comprehensive and accurate analysis** through batch processing.

Implementation in This Test

For this proof of concept, I have **implemented only the real-time and data ingestion layers**:

✅ **Datasource Layer –** Implemented in `microservice-ingestion`, which handles data ingestion from external sources.
✅ **Real-Time Processing Layer –** Implemented in `microservice-consumer`, responsible for processing incoming data streams in real time.
✅ **Serving Layer (API) –** Implemented in `microservice-api`, which provides a RESTful interface to expose real-time data to consumers.

Batch Layer (Future Improvement) – The batch processing layer is not implemented in this version. It is planned as an improvement to allow **more complex historical analysis and data reprocessing**.

Consequences

Positive Outcomes:

✅ **Real-Time Insights –** The system is capable of processing and analyzing data as it arrives.
✅ **Scalability –** The real-time and ingestion layers are built with Kafka and InfluxDB to handle large amounts of data efficiently.
✅ **Extensibility –** The batch processing layer can be integrated in the future without disrupting the existing architecture.

Challenges:

⚠ **Increased Complexity –** Managing two parallel data pipelines (batch and real-time) adds to the system's complexity and requires additional maintenance.
⚠ **Missing Historical Analysis –** Since the batch layer is not yet implemented, the system does not support large-scale historical queries at the moment.

Visual Representation

Infrastructure and Docker

The **DOCKER** directory contains a `docker-compose.yml` file with: - **Kafka** (message broker) - **InfluxDB** (time-series database) - **Zookeeper** (Kafka dependency)

Running the Full Stack

```
cd DOCKER
docker-compose up -d
```

This will spin up Kafka, InfluxDB, and all necessary services.

Areas for Improvement

1. **Logging and Monitoring**:
2. Integrate **Prometheus + Grafana** for better observability.
3. **Security Enhancements**:
4. Add **OAuth2 authentication** for secured endpoints.
5. **Scalability**:
6. Deploy microservices in **Kubernetes** for horizontal scaling.
7. **Testing Coverage**:
8. Increase **unit and integration test coverage** across all services.
9. Implement **contract testing** to ensure API consistency between services.
10. **Performance Optimization**:

11. Use the **official InfluxDB Java client** instead of manually parsing **CSV/JSON** for better performance and reliability.

12. Code Organization and Reusability:

13. Extract the **shared model classes** into a **separate module** to improve maintainability and avoid redundancy across microservices.

Next Steps

- **Performance Optimization**: Profile API responses and optimize database queries.
- **CI/CD Pipeline**: Automate builds and deployments using **GitHub Actions**.
- **Expand Dataset Coverage**: Incorporate predictive maintenance analytics.

Final Thoughts

> *"Perform your duty and abandon all attachment to success or failure. Such evenness of mind is called yoga." — Bhagavad Gita 2.48*

This project has been an incredible journey in developing this test. I am truly excited about the opportunity to work with the **Voith team** and bring this project to new heights.
