Software Architecture and Platforms
Assignment 3 Report

Università di Bologna · Campus di Cesena

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Giacomo Ragni	000518425218	jack

Table of Contents

Chapter 1. Introduzione

Il segreto di una buona introduzione è catturare l’attenzione del lettore.

La storia di Mario e Giovanni inizia con un viaggio in Italia. Io catturo l’attenzione del lettore con una frase d’effetto. Eccola: Ciao a tutti sono un sasso omega e sono qui per raccontarvi la mia storia.

Chapter 2. EBike Event Sourcing

2.1. Kafka Topics

This part describes the Kafka topics used in the EBike event sourcing implementation.

Topics Used:

- **ebike-updates**: This topic carries status and position updates from ebikes to the map service (for visualization purposes) and to the ride service (for local state synchronization). When an ebike changes position or updates its status (available, in use, maintenance), these events are published here.
- **ebike-ride-update**: This topic handles ride-related updates that flow from the ride service to the ebike service. It includes events like ride start/end notifications that trigger state changes in the bicycle's operational status.
- **ride-map-update**: This topic transmits ride events (start/end of rides) from the ride service to the map service. These updates ensure that the map visualization remains current with active rides and completed journeys.
- **ride-user-update**: This topic carries updates from the ride service to the user service, including credit charges for completed rides and ride status changes that affect the user experience.
- **user-update**: This topic handles general user data updates (such as profile information) from the user service to the ride service. These events are used to maintain synchronized local state across services when user information changes.

2.2. Event Flow

At the purpose of explaining the event flow, i'll detail how events are produced and consumed across the microservices involved in the EBike system. The following sections outline the communication patterns between services, including producers, consumers, and the topics they interact with.

2.2.1. Detailed Communication Patterns

1. EBike State Update

- **Producer:** ebike-microservice (*MapCommunicationAdapter*)
- **Topic:** ebike-updates
- **Consumers:** map-microservice (*BikeUpdateAdapter*), ride-microservice (*BikeConsumerAdapter*)
- **Flow:** When an EBike's state or position changes, the ebike-microservice publishes this update to the ebike-updates topic. The map service consumes this message to update the bike's position on the map, while the ride service updates its local repository of available e-bikes.

2. Ride Events Affecting EBike

- **Producer:** ride-microservice (*EBikeCommunicationAdapter*)
- **Topic:** ebike-ride-update
- **Consumer:** ebike-microservice (*RideCommunicationAdapter*)
- **Flow:** When the ride service processes a ride event (e.g., start/end ride), it publishes an update to the ebike-ride-update topic. The ebike service consumes this message and updates the EBike's state accordingly (e.g., from AVAILABLE to IN_USE).

3. Ride Events Affecting Map

- **Producer:** ride-microservice (*MapCommunicationAdapter*)
- **Topic:** ride-map-update
- **Consumer:** map-microservice (*RideUpdateAdapter*)
- **Flow:** When the ride service processes a ride event, it sends a message to the ride-map-update topic. The map service consumes this message to associate or disassociate the user with the bike on the map.

4. Ride Events Affecting User

- **Producer:** ride-microservice (*UserCommunicationAdapter*)
- **Topic:** ride-user-update
- **Consumer:** user-microservice (*RideConsumerAdapter*)
- **Flow:** When the ride service handles user-related events (e.g., charging for a completed ride), it sends an update to the ride-user-update topic. The user service consumes this message to update user data (e.g., remaining credit).

5. User Data Update

- **Producer:** user-microservice (*RideProducerAdapter*)
- **Topic:** user-update
- **Consumer:** ride-microservice (*UserConsumerAdapter*)
- **Flow:** When user data changes in the user service, it sends an update to the user-update topic. The ride service consumes this message to keep its local user repository synchronized.

2.2.2. Typical Event Flow Scenario

When a user starts a ride through the system, a complex sequence of events propagates through the microservices:

1. The ride service receives the API call to start a ride and becomes the initial event producer
2. The ride service publishes events to multiple topics:
    - To ebike-ride-update to inform the ebike service to mark the bike as in use
    - To ride-map-update to update the map visualization
    - To ride-user-update to charge the user's credit for the ride
3. The ebike service responds to these events by:
    - Consuming the ebike-ride-update message and changing the bike's status
    - Publishing its own update to ebike-updates to notify all interested services of the bike's new state
4. The map service maintains an up-to-date view by:
    - Consuming ebike-updates to have current bike positions and statuses
    - Consuming ride-map-update to visualize user-bike associations
5. The user service consumes ride-user-update messages to manage user credit and ride history

This event-driven approach allows the system to maintain consistency while avoiding tight coupling between services. Each service can evolve independently as long as it maintains compatibility with the event formats it produces and consumes.

2.3. Adapter Configuration

Every adapter uses a shared Kafka configuration to connect to the Kafka Cluster.

Kafka Producer Configuration

```
public Properties getProducerProperties() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerAddress);
    props.put(ProducerConfig.ACKS_CONFIG, "all");
    props.put(ProducerConfig.RETRIES_CONFIG, 5);
    props.put(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG, 1000);
    props.put(ProducerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG, 5000);
    props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 500);
    props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
    props.put(ProducerConfig.LINGER_MS_CONFIG, 1);
    props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
    props.put(
        ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringSerializer");
```

```
    props.put(
        ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringSerializer");
    return props;
}
```

Kafka Consumer Configuration

```
public Properties getConsumerProperties() {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerAddress);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "ebike-user-group");
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
    props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "30000");
    props.put(
            ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");
    props.put(
            ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");
    return props;
  }
```

The *Consumer* adapters execute on a separate thread, managed through a single-thread `ExecutorService`. This approach allows for continuous background polling of Kafka messages without blocking the main thread. The polling cycle processes incoming messages by transforming them into JSON objects and updating the appropriate repository (e.g., user, bike, or ride repository depending on the adapter).

Kafka Consumer Execution

```
private void startKafkaConsumer() {
    consumerExecutor = Executors.newSingleThreadExecutor();
    running.set(true);
    consumerExecutor.submit(this::runKafkaConsumer);
  }
```

2.4. Deployment Configuration

The EBike system uses Docker Compose to orchestrate its services, including the Kafka event streaming platform. The Kafka infrastructure consists of Zookeeper for coordination and a Kafka broker for message handling, both integrated into the application's network.

2.4.1. Kafka Infrastructure in Docker Compose

The following services are added to the Docker Compose configuration to support the event sourcing architecture:

- **Zookeeper**: Manages the Kafka cluster coordination
- **Kafka Broker**: Handles the message queuing and delivery
- **Redpanda Console**: Provides a web UI for monitoring Kafka topics and messages

Docker Compose Configuration for Kafka

```
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:5.5.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    networks:
      - eureka-network

  kafka-broker:
```

```
    image: confluentinc/cp-kafka:5.5.0
    hostname: ${KAFKA_BROKER_HOSTNAME}
    depends_on:
      - zookeeper
    ports:
      - "${KAFKA_BROKER_EXTERNAL_PORT}:${KAFKA_BROKER_EXTERNAL_PORT}"
    networks:
      - eureka-network
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://${KAFKA_BROKER_HOSTNAME}:${KAFKA_BROKER_PORT},PLAINTEXT_HOST://localhost:${KAFKA_BROKER_EXTERNAL_PORT}
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    healthcheck:
      test: [ "CMD-SHELL", "kafka-topics --bootstrap-server localhost:${KAFKA_BROKER_EXTERNAL_PORT} --list || exit 1" ]
      interval: 15s
      timeout: 10s
      retries: 5
      start_period: 45s

  redpanda-console:
    image: docker.redpanda.com/redpandadata/console:latest
    ports:
      - "8087:8080"
    networks:
      - eureka-network
    environment:
      KAFKA_BROKERS: "kafka-broker:9092"
    depends_on:
      kafka-broker:
        condition: service_healthy
```

2.4.2. Environment Variables

The following environment variables are set in the `.env` file to configure the Kafka broker:

```
#kafka configuration
KAFKA_BROKER_HOSTNAME=kafka-broker
KAFKA_BROKER_PORT=9092
KAFKA_BROKER_EXTERNAL_PORT=29092
```

These variables are referenced in the Docker Compose file and passed to each microservice to ensure consistent Kafka broker configuration across the system. The internal port (9092) is used for service-to-service communication within the Docker network, while the external port (29092) is mapped to the host for access from outside the container environment.

Each microservice container receives these Kafka connection parameters through environment variables, which are then used in their respective adapter configurations to establish producer and consumer connections to the Kafka broker.

Chapter 3. Kubernetes Deployment

3.1. Containerization and Orchestration

Kubernetes provides a robust platform for deploying, scaling, and managing containerized applications. Container orchestration is crucial for maintaining system reliability and scalability. Kubernetes automates deployment, scaling, and operations of application containers.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kafka
```

3.2. Helm Charts and Operators

NOTE	Helm is the package manager for Kubernetes that simplifies deployment.

> Kubernetes provides the ability to orchestrate complex applications, ensuring they run exactly as intended.
>
> — Kubernetes Documentation, kubernetes.io

StatefulSets are particularly important for *stateful applications* like Kafka. This deployment approach ensures `proper scaling and data persistence`.

Kubernetes Documentation

This architecture provides high availability.[1]

[1] High availability refers to the ability of a system to operate continuously without failure.

Chapter 4. ABike Study Case and Digital Twin

4.1. Digital Twin Concept

4.1.1. Overview and Benefits

A digital twin is a virtual representation of a physical object or system. In the context of ABike, digital twins enable real-time monitoring, analysis, and optimization of bike-sharing operations.

4.2. Implementation Architecture

Digital twins for bike-sharing systems capture various dimensions:

- Physical state (location, condition)
- Usage patterns
- Maintenance requirements
    - Predictive maintenance
    - Service history
- Physical state (location, condition)
- Usage patterns
- Maintenance requirements
    - Predictive maintenance
    - Service history

Component	Function	Description
IoT Sensors	Data Collection	Collect real-time data from bikes and stations
Event Bus	Data Transport	Kafka-based event streaming for real-time updates
Twin Models	State Management	Digital representation of each physical bike
Analytics Engine	Insight Generation	Process data to generate operational insights

4.3. Benefits and Outcomes

The ABike digital twin implementation resulted in:

- 24% increase in bike availability
- 18% reduction in maintenance costs
- Enhanced user experience through predictive station rebalancing

```
{
  "bikeId": "B-1234",
  "status": "in_use",
  "location": {
    "latitude": 44.494887,
    "longitude": 11.342616
  },
  "batteryLevel": 78,
  "lastMaintenance": "2023-10-15"
}
```

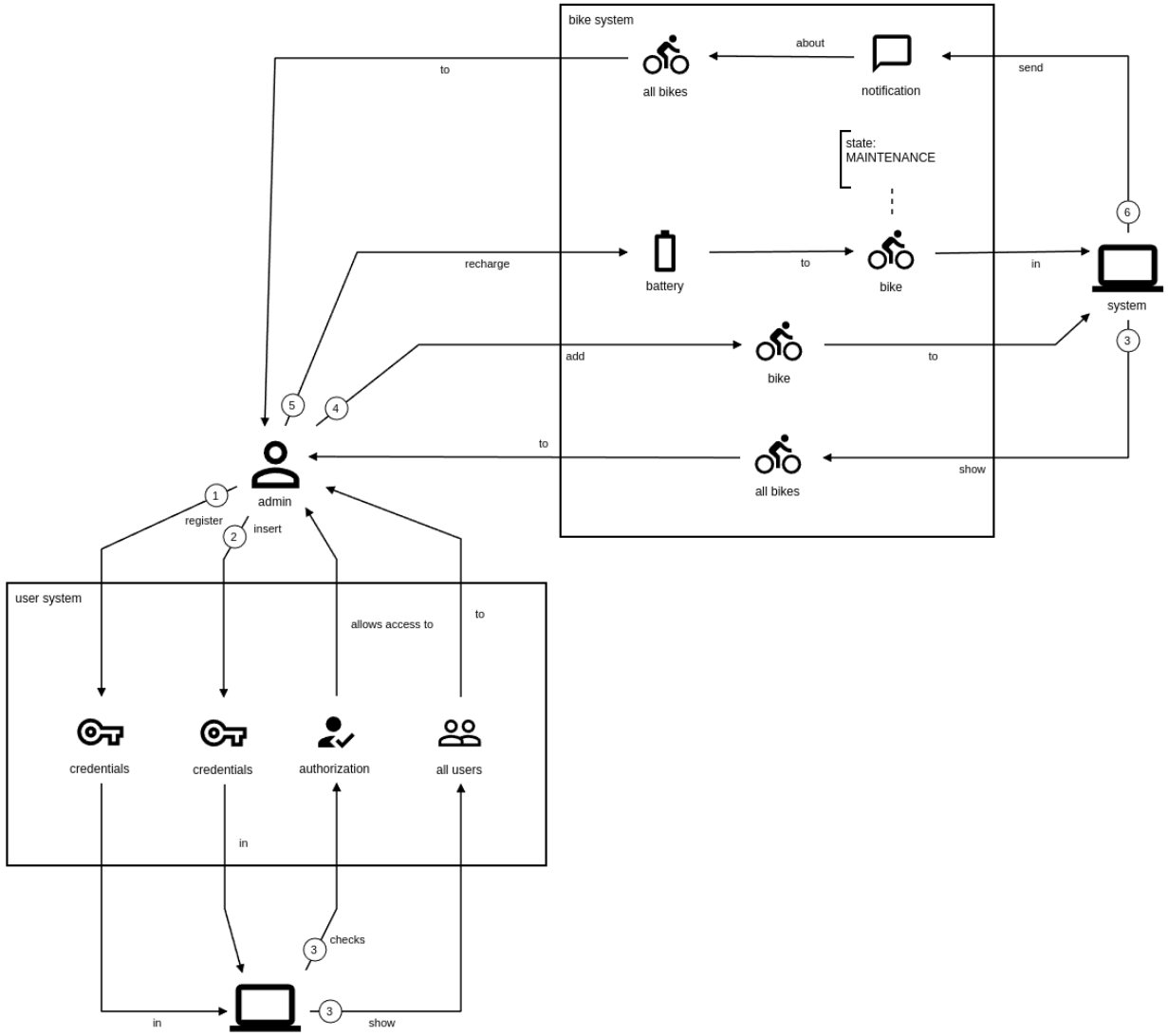
This case study demonstrates how event sourcing and digital twins can transform urban mobility services.

Chapter 5. Conclusion

This is the conclusion of the document. La storia di Mario e Giovanni è finita. Ora Mario è felice e Giovanni è triste. Cosa succederà ora? Quale sarà il prossimo capitolo? Il dinosauro e Mario sono amici?

Admin storytelling

undefined



Admin storytelling

undefined