Assignment 03 Report

University of Bologna

Master Degree in Computer Science and Engineering - Software Architecture and Platforms

Luca Tonelli	0001126379	luca.tonelli11@studio.unibo.it

Table of Contents

Chapter 1. Obiettivi e Descrizione

Il presente assignment ha l'obiettivo di estendere e trasformare il sistema di bike sharing sviluppato negli assignment precedenti, applicando i principi dell'architettura a microservizi basata su event-driven design. L'elaborato si concentra sull'evoluzione di un progetto già modellato secondo Domain Driven Design (DDD), adottando un'architettura distribuita in grado di favorire scalabilità, resilienza e indipendenza dei singoli componenti.

A partire dall'analisi di dominio e dal modello già sviluppato nell'Assignment 2 – di cui si rimanda al report ufficiale disponibile all'indirizzo https://github.com/TonelliLuca/SAP-ass-02/blob/main/doc/asciidoc/doc/assets/docs/report.pdf – il lavoro si focalizza sulle attività di refactoring architetturale e sulle scelte progettuali che abilitano la comunicazione asincrona tra servizi, la suddivisione delle responsabilità, e la gestione delle interazioni tramite eventi di dominio.

Per rispondere in modo strutturato ai requisiti dell'Assignment 3, il progetto è stato suddiviso in tre package principali:

- `p-1`: contiene la trasformazione del sistema in una soluzione a microservizi event-driven, mantenendo la coerenza con il dominio originale.
- `p-2`: Definisce un deployment del sistema sviluppato nel punto precedente utilizzando Kubernetes.
- `p-3`: Estende le funzionalità sviluppate in p-1 introducendo la gestione delle a-bike, delle stazioni e una prima integrazione di digital twin, in linea con le direttive dell'assignment.

Questa organizzazione riflette la volontà di separare logicamente le diverse fasi e aree di sviluppo, facilitando la leggibilità, la manutenzione e la tracciabilità degli avanzamenti progettuali.

Nel seguito del report verranno dettagliate le scelte architetturali, le principali differenze rispetto alla soluzione originaria, e le motivazioni a supporto delle strategie adottate.

Chapter 2. Package P-1: Microservizi Event-Driven

Il package P-1 rappresenta il nucleo centrale del progetto, fornendo un'implementazione completa dell'applicazione E-Bike tramite un'architettura basata su microservizi event-driven. La soluzione si basa sulla soluzione proposta dell'Assignment 02.

2.1. Visione Architetturale

L'architettura si basa sulla comunicazione asincrona tra microservizi tramite eventi, con Apache Kafka utilizzato come broker centrale. Ogni microservizio è stato progettato secondo il principio del **single writer**: ciascun servizio produce gli eventi relativi al proprio dominio, diventando così la fonte di verità per quell'ambito e semplificando la gestione della consistenza e della tracciabilità degli stati nel sistema.

Per la definizione e l'evoluzione degli eventi, sono stati adottati Apache Avro e Confluent Schema Registry, così che ogni evento scambiato su Kafka rispetti uno schema esplicito e condiviso, mantenendo compatibilità tra servizi anche in caso di evoluzione futura dei dati. L'approccio garantisce che producer e consumer siano sempre allineati sul significato e la struttura delle informazioni trasmesse.

Dal punto di vista della struttura interna, ogni microservizio segue un ciclo preciso: riceve eventi da Kafka, li elabora applicando la propria logica di business, aggiorna lo stato locale e, quando necessario, pubblica nuovi eventi verso gli altri servizi. Questo pattern permette a ciascun componente di essere autonomo ma coordinato con il resto del sistema, mantenendo basso l'accoppiamento tra le parti.

Vert.x è stato mantenuto come framework principale sia per la gestione delle API REST (sincrone), sia per health check, caricamento configurazioni tramite variabili d'ambiente, interazione con MongoDB e gestione dell'event bus interno di alcuni servizi. In questo modo si differenzia chiaramente il canale delle chiamate API (esterne/sincrone) dalla comunicazione asincrona (interna/eventi).

Per il monitoraggio del sistema è stata mantenuta l'integrazione con Prometheus, che consente di raccogliere metriche in tempo reale da tutti i microservizi e agevola l'analisi delle performance o l'individuazione di anomalie operative. Completano l'architettura l'API Gateway (punto di accesso centralizzato) e il Service Discovery (Eureka), che garantisce la registrazione e scoperta dinamica dei servizi all'interno del sistema distribuito.

2.2. Dettaglio dei microservizi

Ogni microservizio è stato adattato a una logica event driven, definendo una serie di eventi di dominio gestiti nell'application layer.

2.2.1. EBike Microservice

Il microservizio EBike rappresenta un componente fondamentale nell'architettura event-driven complessiva, responsabile della gestione del ciclo di vita delle biciclette elettriche nel sistema.

Approccio Event-Driven

Il microservizio implementa un approccio event-driven puro:

- **Stato persistente**: Mantiene lo stato attuale delle e-bike nel repository MongoDB
- **Comunicazione asincrona**: Utilizza eventi per notificare altri servizi di cambiamenti di stato
- **Reazioni agli eventi**: Aggiorna lo stato delle e-bike in base agli eventi ricevuti dai noleggi

Questo approccio permette al servizio di mantenere una fonte di verità per lo stato delle e-bike, comunicando efficacemente ogni cambiamento agli altri microservizi interessati.

Modello a Eventi

Il microservizio utilizza diversi tipi di eventi immutabili:

- `EBikeCreateEvent`: Segnala l'aggiunta di una nuova e-bike al sistema
- `EBikeUpdateEvent`: Notifica cambiamenti nello stato di un'e-bike esistente
- `EBikeRechargeEvent`: Comunica la richiesta di ricarica di una specifica e-bike
- `RequestEBikeUpdateEvent`: Trasporta richieste di aggiornamento dei dati di un'e-bike

Tutti gli eventi implementano l'interfaccia `Event` e sono definiti come `record` Java immutabili, garantendo integrità e consistenza nei dati trasmessi.

Adapter di Comunicazione

Il microservizio espone due modalità di interazione:

1. **API REST (sincrone)** tramite `RESTEBikeAdapter` che espone endpoint per:
   - Creazione di e-bike (`POST /api/ebikes/create`)
   - Ricarica di e-bike (`PUT /api/ebikes/:id/recharge`)
2. **Comunicazione asincrona con Kafka**:
   - **Consumer**: Tramite `MapEventConsumer` che ascolta gli eventi `RideUpdateEventAvro` dal topic "ride-events"
   - **Producer**: Pubblica eventi di aggiornamento attraverso `EbikeUpdatesProducer` sul topic "ebike-events"

Flusso Operativo

Per ogni operazione, il microservizio:

1. **Ricezione input**: Tramite REST o eventi Kafka
2. **Elaborazione**: `EBikeServiceImpl` processa le richieste applicando la logica di business
3. **Persistenza**: Lo stato attuale delle e-bike viene mantenuto nel repository
4. **Pubblicazione eventi**: Dopo ogni cambiamento di stato, viene pubblicato un `EBikeUpdateEvent` su Kafka per informare gli altri servizi

All'avvio, il servizio carica tutte le e-bike dal repository e pubblica eventi di aggiornamento per sincronizzare gli altri componenti del sistema.

2.2.2. User Microservice

Il microservizio User rappresenta un'implementazione completa del pattern Event Sourcing all'interno dell'architettura, gestendo tutte le informazioni e le operazioni relative agli utenti del sistema.

Event Sourcing Implementation

A differenza del microservizio EBike, che utilizza un approccio event-driven ma mantiene lo stato corrente, il microservizio User adotta pienamente il pattern Event Sourcing:

- **Persistenza eventi**: Ogni modifica allo stato di un utente viene salvata come evento immutabile nell'event store (MongoDB)
- **Ricostruzione stato**: Lo stato corrente di un utente non è mai salvato direttamente, ma viene sempre ricostruito dalla sequenza degli eventi storici
- **Completa tracciabilità**: La cronologia completa di ogni utente è disponibile per audit, debugging e analisi

Il metodo `rebuildUserState` in `UserServiceImpl` è il cuore dell'implementazione, ricostruendo lo stato attuale dell'utente processando in sequenza temporale tutti gli eventi associati. Nello specifico è stato scelto di integrare event-sourcing in questo microservizio pechè, in un contesto reale, gestisce il credito utente, è quindi utile tenere una traccia temporale delle modifiche di credito del singolo utente.

Modello a Eventi

Il microservizio utilizza diversi tipi di eventi immutabili:

- `UserCreatedEvent`: Registra la creazione di un nuovo utente nel sistema
- `UserSignInEvent`: Traccia gli accessi degli utenti
- `UserUpdateEvent`: Cattura modifiche allo stato dell'utente (tipo, credito)
- `RechargeCreditEvent`: Registra le operazioni di ricarica del credito

- `RequestUserUpdateEvent`: Veicola richieste di aggiornamento provenienti da altri servizi

Tutti gli eventi implementano l'interfaccia `Event` e sono definiti come `record` Java, garantendo immutabilità e consistenza.

Adapter di Comunicazione

Il microservizio espone due modalità di interazione:

1. **API REST (sincrone)** tramite `RESTUserAdapter`:
   - Registrazione utente (`POST /api/users/signup`)
   - Accesso utente (`POST /api/users/signin`)
   - Ricarica credito (`PATCH /api/users/:username/recharge`)
   - Osservazione utenti tramite WebSocket (`/observeAllUsers`, `/observeUser/:username`)
2. **Comunicazione asincrona con Kafka**:
   - **Consumer**: `RideUpdateConsumer` ascolta il topic "ride-events" per aggiornare il credito degli utenti
   - **Producer**: Pubblica eventi di aggiornamento attraverso `UserUpdatesProducer` sul topic "user-events"

Flusso operativo

Per ogni operazione, il microservizio:

1. **Ricezione input**: Da API REST o eventi Kafka
2. **Recupero eventi**: Ottiene tutti gli eventi storici relativi all'utente
3. **Ricostruzione stato**: Applica sequenzialmente gli eventi per ottenere lo stato corrente
4. **Valutazione comando**: Verifica la fattibilità dell'operazione richiesta
5. **Generazione evento**: Crea un nuovo evento che rappresenta l'operazione
6. **Persistenza evento**: Salva l'evento nell'event store
7. **Pubblicazione**: Notifica altri servizi tramite Kafka e WebSocket

Questo approccio garantisce consistenza, tracciabilità completa e la possibilità di "time travel" attraverso la storia di ogni utente.

All'avvio, il servizio ricostruisce lo stato di tutti gli utenti dai loro eventi e pubblica eventi di aggiornamento per sincronizzare gli altri microservizi.

2.2.3. Ride Microservice

Il microservizio Ride rappresenta un componente fondamentale nell'architettura event-driven complessiva, responsabile della gestione del noleggio di e-bike da parte degli utenti e della simulazione in tempo reale delle corse.

Approccio a Proiezioni Locali

Il microservizio implementa un approccio che combina elementi event-driven con proiezioni locali:

- **Event-driven**: Utilizza eventi per comunicare con altri servizi e gestire il ciclo di vita delle ride
- **Proiezioni locali**: Mantiene copie locali degli stati di utenti ed e-bike attraverso il `LocalProjectionRepository`
- **Simulazione eventi**: Genera eventi in tempo reale durante il noleggio tramite `RideSimulation` e `EventPublisher`

Questo approccio permette al servizio di reagire rapidamente alle richieste degli utenti senza dover interrogare continuamente gli altri microservizi, mantenendo una cache locale degli stati più recenti ricostruiti dagli eventi ricevuti.

Modello a Eventi

Il microservizio utilizza diversi tipi di eventi immutabili:

- `RideStartEvent`: Registra l'inizio di un noleggio, associando un utente a una e-bike
- `RideStopEvent`: Segnala la conclusione di un noleggio, liberando l'e-bike
- `RideUpdateEvent`: Comunica aggiornamenti continui durante la simulazione (posizione, batteria, credito)
- `RequestRideEndEvent`: Veicola richieste di terminazione anticipata di un noleggio

Parallelamente, il servizio consuma e memorizza eventi provenienti da altri domini: * `UserUpdateEvent`: Eventi relativi agli utenti ricevuti dal topic "user-events" * `EBikeUpdateEvent`: Eventi relativi alle e-bike ricevuti dal topic "ebike-events"

Tutti gli eventi sono implementati come `record` Java immutabili, garantendo integrità e tracciabilità delle operazioni.

Adapter di Comunicazione

Il microservizio espone due modalità di interazione:

1. **API REST (sincrone)** tramite `RideServiceVerticle`:
   - Avvio noleggio (`POST /startRide`)
   - Terminazione noleggio (`POST /stopRide`)
   - Health check (`GET /health`) e metriche (`GET /metrics`)
2. **Comunicazione asincrona con Kafka**:
   - **Consumer**: Riceve aggiornamenti sugli utenti e le e-bike tramite `ProjectionUpdatesConsumer`
   - **Producer**: Pubblica eventi di noleggio attraverso `RideEventsProducer`

Flusso Operativo

Il flusso di lavoro del servizio segue questo pattern:

1. **Ricezione richiesta**: Tramite REST o eventi
2. **Consultazione proiezioni**: Verifica dello stato attuale di utenti ed e-bike dalle proiezioni locali
3. **Validazione business rules**: Controllo di disponibilità dell'e-bike, credito sufficiente e livello batteria
4. **Creazione noleggio**: Inizializzazione di un oggetto `Ride` e avvio della simulazione
5. **Pubblicazione eventi**: Notifica agli altri servizi tramite eventi Kafka

La simulazione di una corsa, come nell'implementazione base: * Aggiorna continuamente la posizione dell'e-bike secondo un algoritmo di movimento * Riduce progressivamente il credito dell'utente e la batteria dell'e-bike * Pubblica eventi `RideUpdateEvent` ad ogni aggiornamento tramite `EventPublisher` * Termina automaticamente quando la batteria o il credito raggiungono zero

All'avvio, il servizio si registra con Eureka per il service discovery e inizia ad ascoltare gli eventi da Kafka, costruendo progressivamente le sue proiezioni locali per utenti ed e-bike.

2.2.4. Map Microservice

Il microservizio Map, già presente nella precedente versione dell'architettura, ha richiesto solo adattamenti minimali per integrarsi nel nuovo sistema event-driven. In particolare, è stato implementato il componente `MapEventConsumer` che permette al servizio di reagire in tempo reale agli eventi generati dagli altri microservizi.

Questo consumer si sottoscrive a due topic Kafka fondamentali:

- ride-events: per tracciare inizio e fine dei noleggi
- ebike-events: per aggiornare la posizione e lo stato delle biciclette

Quando riceve un evento, il consumer lo processa e direziona l'update alla web socket specifica.

2.3. Scelte architetturali e organizzazione dei topic Kafka

Nel disegno dei topic Kafka, si è seguito un criterio di coerenza con i bounded context del dominio, evitando topic troppo generici/specifici. Ogni topic è stato progettato per raccogliere eventi di una singola tipologia logica, in modo che i consumatori possano sottoscriversi solo ai flussi rilevanti e interpretare con chiarezza la semantica degli eventi ricevuti.

Nello specifico:

- Il topic `ride-events` veicola esclusivamente eventi relativi al ciclo di vita dei noleggi (inizio, aggiornamento, fine corsa).
- Il topic `ebike-events` raccoglie eventi di aggiornamento delle biciclette elettriche.
- Il topic `user-events` è dedicato agli aggiornamenti di stato degli utenti.

Questa organizzazione dei topic facilita la **gestione della single source of truth** (ogni servizio produce eventi solo sul proprio topic di riferimento), consente la scalabilità indipendente dei consumer per ogni flusso e rende semplice evolvere gli schemi degli eventi con Avro senza impattare altri domini.

2.3.1. Flusso di propagazione e validazione degli eventi

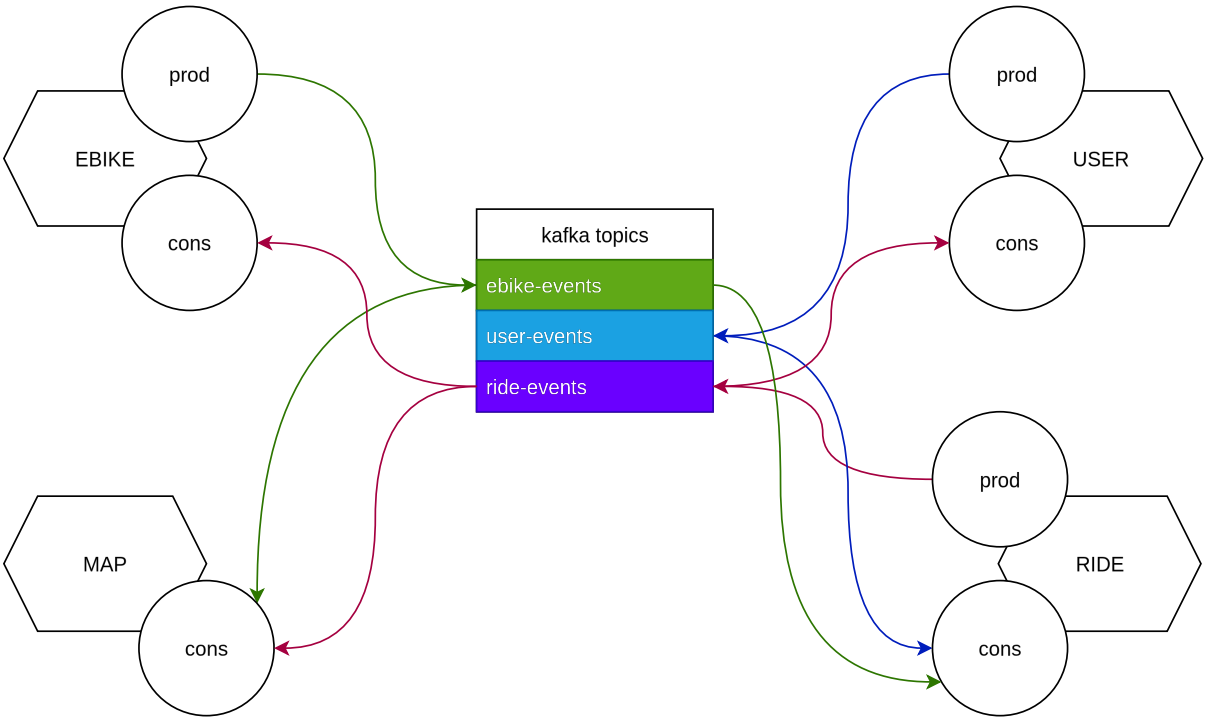


Figure 1. Lista dei topic e definizione dei consumer/producer

Nel ciclo di vita di una corsa, i microservizi non comunicano tramite invocazioni dirette, ma esclusivamente attraverso eventi Kafka, mantenendo un disaccoppiamento logico tra le componenti.

Il microservizio Ride ha il compito di gestire la logica di simulazione della corsa: al momento dell'avvio (`startRide`) o durante l'esecuzione, esso produce eventi (`RideStartedEvent`, `RideUpdateEvent`) che contengono informazioni come la bici utilizzata, l'utente coinvolto, la posizione, il tempo e i consumi stimati.

Tuttavia, questi eventi non modificano direttamente lo stato effettivo delle entità coinvolte (come la disponibilità di una bici o il credito di un utente). Invece, tali eventi vengono **consumati dai microservizi EBike e User**, che:

- Valutano il contenuto dell'evento secondo le proprie regole di business (es. controllo sul credito residuo o sul livello di batteria),
- Decidono autonomamente se accettare l'azione proposta,
- E, solo in caso di esito positivo, producono un proprio evento (`EBikeUpdatedEvent`, `UserUpdatedEvent`) che rappresenta la modifica effettiva dello stato interno.

Questo meccanismo garantisce che ogni microservizio resti l'unico responsabile dello stato che gestisce, evitando conflitti e mantenendo la coerenza dei dati. Ad esempio:

- È **EBikeService**, e non Ride, a decidere se una bici può essere assegnata e a comunicarne la nuova disponibilità.
- È **UserService**, e non Ride, a calcolare e registrare la diminuzione del credito utente.

Gli eventi `RideUpdateEvent` prodotti da Ride fungono quindi da **richiesta**, ma la modifica reale dello stato avviene solo in risposta alla validazione interna da parte del microservizio destinatario. In questo modo, la sincronizzazione dello stato tra i vari microservizi avviene in maniera affidabile, tracciabile e coerente.

I servizi di supporto come Map consumano infine gli eventi di aggiornamento (`ride-events`, `ebike-events`) per aggiornare le proiezioni grafiche o informative, senza avere alcuna autorità sui dati stessi, gli aggiornamenti relativi all'utente invece, vengono notificati alla gui tramite le web socket interne al microservizio user come da previa implementazione.

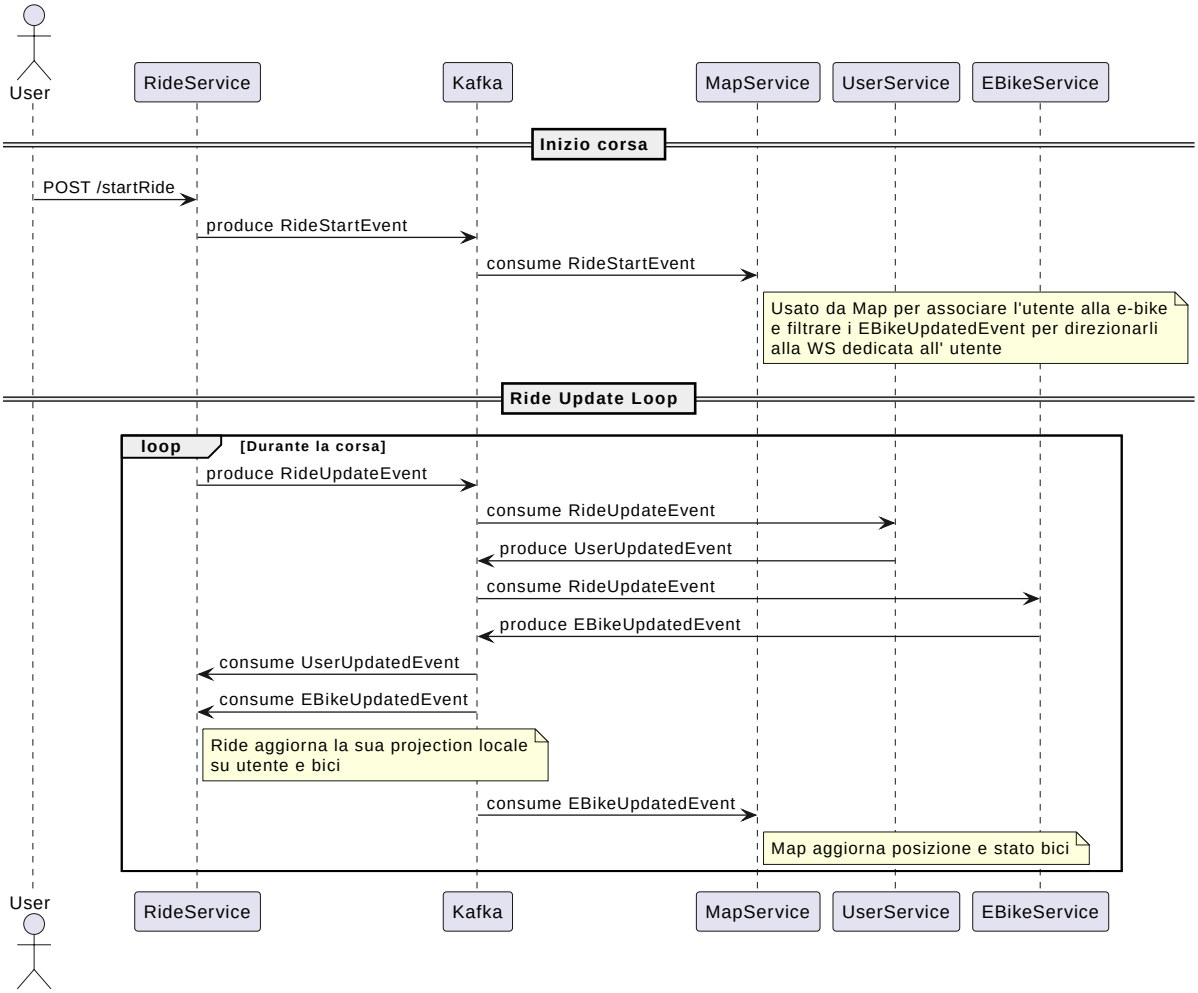


Figure 2. Diagramma di sequenza che rappresenta il ciclo di eventi prodotti da una ride

Serializzazione e validazione degli eventi con Apache Avro e Schema Registry

All'interno del sistema, la serializzazione e la validazione degli eventi tra microservizi si basano sull'utilizzo congiunto di **Apache Avro** e **Confluent Schema Registry**. Questa combinazione consente di garantire messaggi binari compatti, schema-based, validati automaticamente sia lato producer che consumer.

Confluent Schema Registry è stato integrato nel sistema tramite container Docker dedicato, configurato all'interno del `docker-compose.yml` per comunicare con Kafka:

```
schema-registry:
  image: confluentinc/cp-schema-registry:7.9.1
  ports:
    - "${SCHEMA_REGISTRY_HOST_PORT}:8081"
  environment:
    - SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS=PLAINTEXT://kafka:29092
    - SCHEMA_REGISTRY_HOST_NAME=schema-registry
    - SCHEMA_REGISTRY_LISTENERS=http://0.0.0.0:8081
```

Lo Schema Registry mantiene un registro centralizzato di tutti gli schemi `.avsc` utilizzati nel sistema, assegnando a ciascuno un identificatore univoco. In questo modo, i messaggi Kafka trasportano solo l'ID dello schema e non l'intera definizione, ottimizzando le performance e garantendo la validazione automatica e la compatibilità evolutiva.

Gli eventi sono modellati come record Avro, che possono contenere campi primitivi oppure record annidati. Ogni producer riceve una serie di eventi di dominio, che successivamente vengono mappati in uno schema avro specifico e inviati. La definizione di uno schema tipico prevede:

- Il tipo principale `record`
- Un nome simbolico identificativo
- Un namespace coerente con il dominio applicativo
- Timestamp e UUID dell'evento
- Un array di `fields`, ciascuno con nome e tipo

Esempio generico di schema Avro

```
{
  "type": "record",
  "name": "SomeDomainEvent",
  "namespace": "events.avro",
  "fields": [
    {"name": "id", "type": "string"},
    {"name": "timestamp", "type": "string"},
    {"name": "payload", "type": {
      "type": "record",
      "name": "Entity",
      "fields": [
        {"name": "id", "type": "string"},
        {"name": "status", "type": "string"},
        {"name": "value", "type": "int"}
      ]
    }}
  ]
}
```

Questa struttura permette di definire eventi ricchi e auto-descrittivi, mantenendo un contratto formale tra i servizi che li scambiano.

In alcuni topic Kafka vengono trasmessi più tipi di eventi appartenenti a uno stesso dominio. Per gestirli con un unico schema, è stato adottato il pattern **Union Type** di Avro.

Si definisce un record contenitore che include un campo `payload`, il cui tipo è una unione di più tipi concreti.

Esempio generico di schema Union

```
{
  "type": "record",
  "name": "DomainEventUnion",
  "namespace": "events.avro",
  "fields": [
    {
      "name": "payload",
      "type": [
        "events.avro.EventTypeA",
        "events.avro.EventTypeB",
        "events.avro.EventTypeC"
      ]
    }
  ]
}
```

Questo approccio consente: - di trasmettere più eventi logicamente affini in un singolo topic; - di evitare la proliferazione di topic distinti; - di mantenere coerenza e validazione schema-based.

Lato consumer, il record Avro ricevuto viene deserializzato come `GenericRecord`. Poiché il campo `payload` è una union, è necessario **identificare il tipo specifico dell'evento ricevuto** tramite il nome dello schema.

Il consumer può quindi applicare un filtro e processare solo i tipi rilevanti per il proprio dominio.

Esempio generico di codice di consumo filtrato

```
private void processEvent(String key, GenericRecord event) {
    String schemaName = event.getSchema().getName();
    if (!"EventTypeB".equals(schemaName)) {
        logger.debug("Evento ignorato: {}", schemaName);
        return;
    }

    // Elaborazione dell'evento di interesse
    String id = event.get("id").toString();
    GenericRecord payload = (GenericRecord) event.get("payload");
    String status = payload.get("status").toString();

    // Logica applicativa...
}
```

Questa tecnica permette di mantenere consumer generici, ma configurabili per gestire **solo gli eventi significativi**.

Chapter 3. Package P-2: Deployment su Kubernetes

3.1. Obiettivo

Il secondo package dell'elaborato si concentra sul deployment dell'intera architettura progettata nel package P-1 su una infrastruttura distribuita basata su Kubernetes.

3.2. Strategia di Deployment

L'intero sistema è stato containerizzato e distribuito tramite oggetti Deployment, Service, ConfigMap e PersistentVolumeClaim definiti in manifesti YAML e orchestrati all'interno del namespace dedicato sap-assignment. La piattaforma di riferimento è Minikube, configurata con 4 CPU e 4 GB di RAM, che consente di testare l'infrastruttura in locale in un contesto realistico.

3.3. Componenti Infrastrutturali Distribuiti

3.3.1. Kafka & Zookeeper

Il broker Kafka è gestito da un Deployment dedicato, accompagnato dal coordinatore Zookeeper. Entrambi i componenti utilizzano init container per attendere l'avvio dell'altro e assicurare la corretta sequenza di bootstrap.

Kafka espone sia porte interne (29092) che esterne (9092) e ha abilitato la creazione automatica dei topic

Zookeeper viene reso disponibile sulla porta 2181 tramite service ClusterIP

3.3.2. Schema Registry

Schema Registry di Confluent è deployato come microservizio separato e interfacciato con Kafka. La configurazione ne garantisce l'inizializzazione solo dopo la disponibilità del broker. Espone la porta 8081 ed è configurato per l'accesso in lettura/scrittura degli schemi Avro.

3.3.3. Kafka UI

Per l'osservabilità e il debugging è stato incluso Kafka UI, configurato per comunicare con Kafka e Schema Registry. Espone la porta 8082 tramite un Service di tipo LoadBalancer per facilitarne l'accesso.

3.3.4. MongoDB

Il database MongoDB viene eseguito in un Deployment associato ad una PVC da 1Gi. Il volume persistente è utilizzato per conservare i dati utente, noleggi ed e-bike anche in caso di riavvio dei pod.

3.3.5. Prometheus

Per il monitoraggio dei microservizi, Prometheus è stato incluso e configurato con initContainers per attendere la disponibilità del gateway. Il servizio Prometheus espone la porta 9090 e può essere raggiunto esternamente tramite NodePort.

3.3.6. Service Discovery (Eureka)

Il registro dei servizi Eureka viene eseguito in un Deployment separato e reso disponibile tramite service sulla porta 8761. I microservizi si registrano dinamicamente tramite configurazione centralizzata in ConfigMap.

3.4. Microservizi Applicativi

Tutti i microservizi del package P-1 (API Gateway, EBike, Ride, User, Map) sono deployati come Deployment Kubernetes con una replica ciascuno. Ogni servizio dispone di:

initContainers per assicurare il corretto ordine di avvio rispetto a dipendenze (Eureka, Kafka, MongoDB, ecc.)

readinessProbe e livenessProbe per supportare auto-riparazione e rolling update

Service di tipo ClusterIP per comunicazioni interne e per il load balancing interno tra pod

Tutti i microservizi sono accessibili unicamente tramite l'API Gateway, che espone un Service di tipo NodePort. In questo modo, il bilanciamento del carico tra le repliche dei microservizi avviene internamente grazie alla natura dei servizi ClusterIP, mentre l'accesso esterno avviene attraverso un singolo punto di ingresso centralizzato (API Gateway).

3.5. Configurazione Centralizzata

La configurazione di sistema è centralizzata in una ConfigMap denominata app-config, che contiene:

Informazioni su host e porte dei servizi

URL di connessione a Kafka, MongoDB, Eureka, Prometheus

Parametri runtime per microservizi

Tutti i container referenziano la ConfigMap per caricare dinamicamente le variabili d'ambiente.

3.6. Esecuzione del Deployment

L'intero processo di deploy viene automatizzato tramite uno script Bash (rebuild-minikube.sh), che:

1. Ferma ed elimina eventuali istanze Minikube precedenti
2. Avvia una nuova istanza Minikube configurata con risorse adeguate
3. Costruisce localmente tutte le immagini Docker necessarie a partire dai Dockerfile nei microservizi
4. Crea il namespace sap-assignment
5. Applica in sequenza tutti i manifest Kubernetes
6. Espone gli indirizzi IP e monitora lo stato dei pod

```
#!/bin/bash
set -e

echo "==== Destroying existing Minikube ===="
minikube stop
minikube delete

echo "==== Creating fresh Minikube instance ===="
minikube start --driver=docker --cpus=4 --memory=4096

echo "==== Setting up Docker environment for Minikube ===="
eval $(minikube docker-env)

echo "==== Building Docker images ===="
docker build -t eureka-server:latest -f ../p-1/microservices/service-discovery/Dockerfile ../p-1/microservices/service-discovery
docker build -t api-gateway:latest -f ../p-1/microservices/api-gateway/Dockerfile ../p-1/microservices/api-gateway
docker build -t user-microservice:latest -f ../p-1/microservices/user-microservice/Dockerfile ../p-1/microservices/user-microservice
docker build -t ebike-microservice:latest -f ../p-1/microservices/ebike-microservice/Dockerfile ../p-1/microservices/ebike-microservice
docker build -t map-microservice:latest -f ../p-1/microservices/map-microservice/Dockerfile ../p-1/microservices/map-microservice
docker build -t ride-microservice:latest -f ../p-1/microservices/ride-microservice/Dockerfile ../p-1/microservices/ride-microservice

echo "==== Deploying to Kubernetes ===="
echo "Creating ConfigMap..."
kubectl create ns sap-assignment
kubectl apply -f ./k8s/configMap.yaml
echo "Deploying infrastructure..."
kubectl apply -f ./k8s/kafka/zookeeper.yaml
kubectl apply -f ./k8s/kafka/kafka-broker-1.yaml
kubectl apply -f ./k8s/kafka/schema-registry.yaml
kubectl apply -f ./k8s/mongodb.yaml
kubectl apply -f ./k8s/prometheus.yaml
kubectl apply -f ./k8s/api-gateway.yaml
kubectl apply -f ./k8s/eureka-server.yml

echo "Deploying services..."
kubectl apply -f ./k8s/user-microservice.yaml
```

```
kubectl apply -f ./k8s/ebike-microservice.yaml
kubectl apply -f ./k8s/ride-microservice.yaml
kubectl apply -f ./k8s/map-microservice.yaml

echo "Deploying Kafka UI"
kubectl apply -f ./k8s/kafka/kafka-ui.yaml

echo "==== Minikube IP address ===="
minikube ip

echo "==== Deployment complete. Checking status... ===="
watch kubectl get pods -n sap-assignment
```

3.7. Accesso al Sistema

Una volta completato il deployment, l'accesso all'API Gateway avviene tramite il comando:

```
kubectl port-forward --address 0.0.0.0 api-gateway- 8080:8080 -n sap-assignment
```

Questo consente di esporre localmente l'applicazione per testare l'interazione tra microservizi e verificare i flussi di eventi.

All'interno del package è presente una copia del presentation layer del package p-1 per testare il funzionamento.

3.8. Pipeline di Lancio Completa

Di seguito viene descritta la sequenza operativa per eseguire il deploy e l'avvio completo dell'applicazione, a partire dall'infrastruttura fino all'interfaccia utente.

1. **Eseguire il deployment dell'infrastruttura**

   ```
   cd src/p-2/
   ./rebuild-minikube.sh
   ```

   Attendere che tutti i pod risultino in stato `Running` con tutti i container pronti (es. `1/1`, `2/2`, ecc.).

2. **Impostare il port forwarding per l'API Gateway**

   ```
   kubectl port-forward --address 0.0.0.0 deployment/api-gateway 8080:8080 -n sap-assignment
   ```

   Lasciare aperto questo terminale durante l'utilizzo dell'applicazione.

3. **Avviare il presentation layer**

   ```
   cd ../presentation/
   ./gradlew run
   ```

Chapter 4. Package P-3: Introduzione di A-Bike, Station e Digital Twin

4.1. Estensione del Domain-Driven Design: A-Bike, Station e Digital Twin

Per rispondere alle nuove richieste progettuali introdotte nella terza parte dell'elaborato, si è reso necessario estendere il Domain-Driven Design sviluppato nel progetto originale. In particolare, vengono introdotti i concetti di bicicletta autonoma (a-bike), stazione (station) e digital twin, al fine di supportare scenari avanzati di mobilità autonoma all'interno di un contesto smart city.

4.1.1. Obiettivi e requisiti dell'estensione

Gli obiettivi definiti dall'elaborato sono i seguenti:

- Introdurre una nuova tipologia sperimentale di bicicletta, la **a-bike** (bicicletta autonoma), in grado di:
  - Raggiungere autonomamente la stazione più vicina dopo essere stata utilizzata;
  - Raggiungere autonomamente la posizione dell'utente che ne richiede il servizio.
- Integrare le a-bike in un ambiente di smart city, sfruttando una componente di **digital twin** che:
  - Mantenga una rappresentazione virtuale aggiornata dello stato delle biciclette e delle stazioni;

4.1.2. Estensione dell'Ubiquitous Language

Termine	Descrizione
A-Bike	Bicicletta elettrica autonoma, dotata di capacità di movimento automatico, in grado di raggiungere la stazione più vicina dopo l'utilizzo e di muoversi verso l'utente che ne richiede il servizio.
Stazione	Punto fisico all'interno della città dove le a-bike possono essere parcheggiate e ricaricate. Ogni stazione gestisce la propria capacità disponibile.
Digital Twin	Gemello digitale della città: una rappresentazione virtuale aggiornata in tempo reale di biciclette e stazioni che fornisce servizi di supporto e monitoraggio.

4.1.3. Requisiti Funzionali Estesi per A-Bike, Station e Digital Twin

1. Il sistema deve permettere agli utenti di richiedere una a-bike tramite l'interfaccia grafica, così che una bicicletta autonoma si diriga automaticamente verso la loro posizione.
2. Il sistema deve permettere agli utenti di annullare una richiesta di a-bike già inviata; in tal caso, la a-bike deve tornare autonomamente alla stazione più vicina disponibile.
3. Il sistema deve permettere agli utenti di visualizzare tramite l'interfaccia grafica tutte le stazioni e monitorarne la disponibilità in tempo reale.
4. Il sistema deve permettere agli utenti di seguire, sempre tramite interfaccia grafica, l'intero ciclo della a-bike richiesta, dalla partenza verso l'utente, durante la corsa e fino al ritorno in stazione.
5. Il sistema deve permettere alle a-bike di raggiungere autonomamente la stazione più vicina al termine di ogni corsa.
6. Il sistema deve permettere agli utenti amministratori di creare nuove stazioni all'interno dell'ambiente.
7. Il sistema deve permettere agli utenti amministratori di aggiungere nuove a-bike e di assegnarle a una stazione specifica.
8. Il sistema deve permettere alla componente Digital Twin di mantenere una vista aggiornata e coerente della posizione e dello stato di tutte le a-bike e delle stazioni.

4.1.4. User Story

User Story	Descrizione
US12: Richiesta A-Bike	**Given** un utente autenticato **When** l'utente richiede una a-bike tramite l'interfaccia grafica **Then** il sistema seleziona la a-bike disponibile più vicina e la fa muovere automaticamente verso la posizione dell'utente

User Story	Descrizione
US13: Annullamento richiesta A-Bike	**Given** un utente che ha richiesto una a-bike **When** l'utente annulla la richiesta prima che la corsa inizi **Then** la a-bike torna autonomamente alla stazione più vicina disponibile
US14: Monitoraggio A-Bike e Stazioni	**Given** un utente che ha richiesto una a-bike **When** la a-bike è in movimento verso l'utente, durante la corsa, e al termine della stessa **Then** l'utente può seguire graficamente su mappa lo spostamento della bici, la corsa e il ritorno in stazione, e può visualizzare lo stato e la disponibilità delle stazioni in tempo reale
US15: Docking Automatico	**Given** una a-bike che termina una corsa **When** la corsa viene conclusa **Then** la a-bike si dirige e parcheggia autonomamente presso la stazione più vicina con slot disponibile
US16: Creazione Stazione (Admin)	**Given** un utente amministratore **When** l'amministratore accede alla sezione di gestione delle stazioni **Then** può creare una nuova stazione indicando posizione e capacità
US17: Aggiunta A-Bike (Admin)	**Given** un utente amministratore **When** l'amministratore accede alla sezione di gestione delle a-bike **Then** può aggiungere una nuova a-bike e assegnarla a una stazione specifica
US18: Monitoraggio tramite Digital Twin (Admin)	**Given** un amministratore **When** accede all'interfaccia di Digital Twin **Then** può visualizzare in tempo reale lo stato, la posizione e la disponibilità di tutte le a-bike e delle stazioni

4.2. Use-Case diagrams

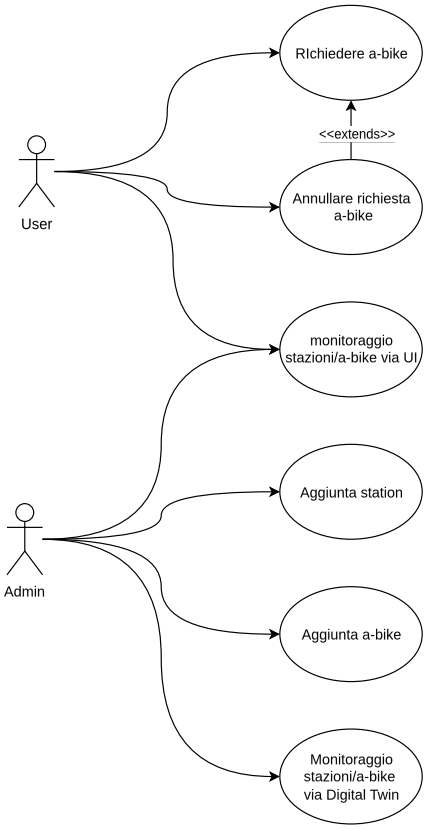


Figure 3. Use-case diagram relativo agli utenti

Campo	Descrizione
Use Case	Richiedere a-bike
Attori	Utente

Campo	Descrizione
Scope	Applicazione Smart City
Livello	Alto
Precondizioni	L’utente è autenticato e ha credito sufficiente; esiste almeno una a-bike disponibile
Postcondizioni	Una a-bike viene inviata verso la posizione dell’utente; la richiesta è tracciata nel sistema
Trigger	L’utente seleziona “Richiedi a-bike” nell’interfaccia grafica
Flusso	. L’utente seleziona l’opzione per richiedere una a-bike . Il sistema individua la a-bike disponibile più vicina . Il sistema comanda la a-bike a muoversi verso l’utente . L’utente può monitorare lo stato della richiesta tramite la GUI

Campo	Descrizione
Use Case	Annullare richiesta a-bike
Attori	Utente
Scope	Applicazione Smart City
Livello	Alto
Precondizioni	Esiste una richiesta di a-bike in corso non ancora conclusa
Postcondizioni	La richiesta viene annullata; la a-bike torna autonomamente alla stazione più vicina disponibile
Trigger	L’utente seleziona “Annulla richiesta a-bike” nell’interfaccia grafica
Flusso	. L’utente seleziona l’opzione per annullare la richiesta di a-bike . Il sistema riceve l’annullamento . La a-bike interrompe il movimento verso l’utente (se non ancora arrivata) . Il sistema comanda la a-bike a tornare alla stazione disponibile più vicina

Campo	Descrizione
Use Case	Monitoraggio stazioni/a-bike via UI
Attori	Utente
Scope	Applicazione Smart City
Livello	Alto
Precondizioni	L’utente è autenticato
Postcondizioni	L’utente può visualizzare in tempo reale la posizione e la disponibilità delle stazioni e delle a-bike, e seguire lo stato della richiesta
Trigger	L’utente effettua il login al sistema
Flusso	. L’utente effettua il login al sistema . Il sistema mostra le stazioni, la disponibilità di slot e la posizione delle a-bike . L’utente può seguire l’intero ciclo della a-bike richiesta (avvicinamento, corsa, ritorno in stazione)

Campo	Descrizione
Use Case	Aggiunta stazione
Attori	Admin
Scope	Applicazione Smart City
Livello	Medio
Precondizioni	L’admin è autenticato
Postcondizioni	Una nuova stazione è registrata nel sistema, visibile nella mappa e disponibile per l’assegnazione di a-bike
Trigger	L’admin seleziona “Add stazione” nell’interfaccia di amministrazione
Flusso	. L’admin preme il pulsatne "Add stazione" . Inserisce posizione, capacità e altri dettagli della stazione . Il sistema salva la nuova stazione e la rende disponibile

Campo	Descrizione
Use Case	Aggiunta a-bike

Campo	Descrizione
Attori	Admin
Scope	Applicazione Smart City
Livello	Medio
Precondizioni	L’admin è autenticato; esiste almeno una stazione a cui assegnare la nuova a-bike
Postcondizioni	Una nuova a-bike è aggiunta alla flotta ed è assegnata a una stazione
Trigger	L’admin seleziona “Add a-bike” nell’interfaccia di amministrazione
Flusso	. L’admin preme il pulsante "add a-bike" . Inserisce i dati della nuova a-bike . Seleziona la stazione di assegnazione . Il sistema salva la nuova a-bike associandola alla stazione scelta

Campo	Descrizione
Use Case	Monitoraggio stazioni/a-bike via Digital Twin
Attori	Admin
Scope	Applicazione Smart City
Livello	Alto
Precondizioni	L’admin è autenticato
Postcondizioni	L’admin può visualizzare in tempo reale la posizione, lo stato e la disponibilità di tutte le a-bike e delle stazioni tramite la piattaforma Digital Twin
Trigger	L’admin accede alla dashboard Digital Twin
Flusso	. L’admin apre la dashboard Digital Twin . Il sistema mostra una vista aggiornata di tutte le stazioni e delle a-bike . L’admin può filtrare, ricercare o analizzare lo stato delle risorse

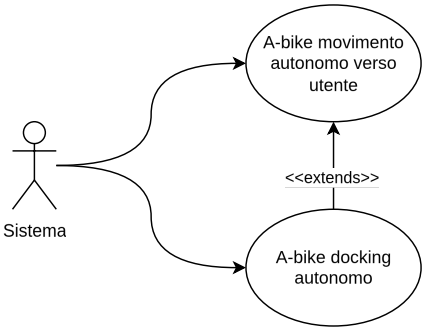


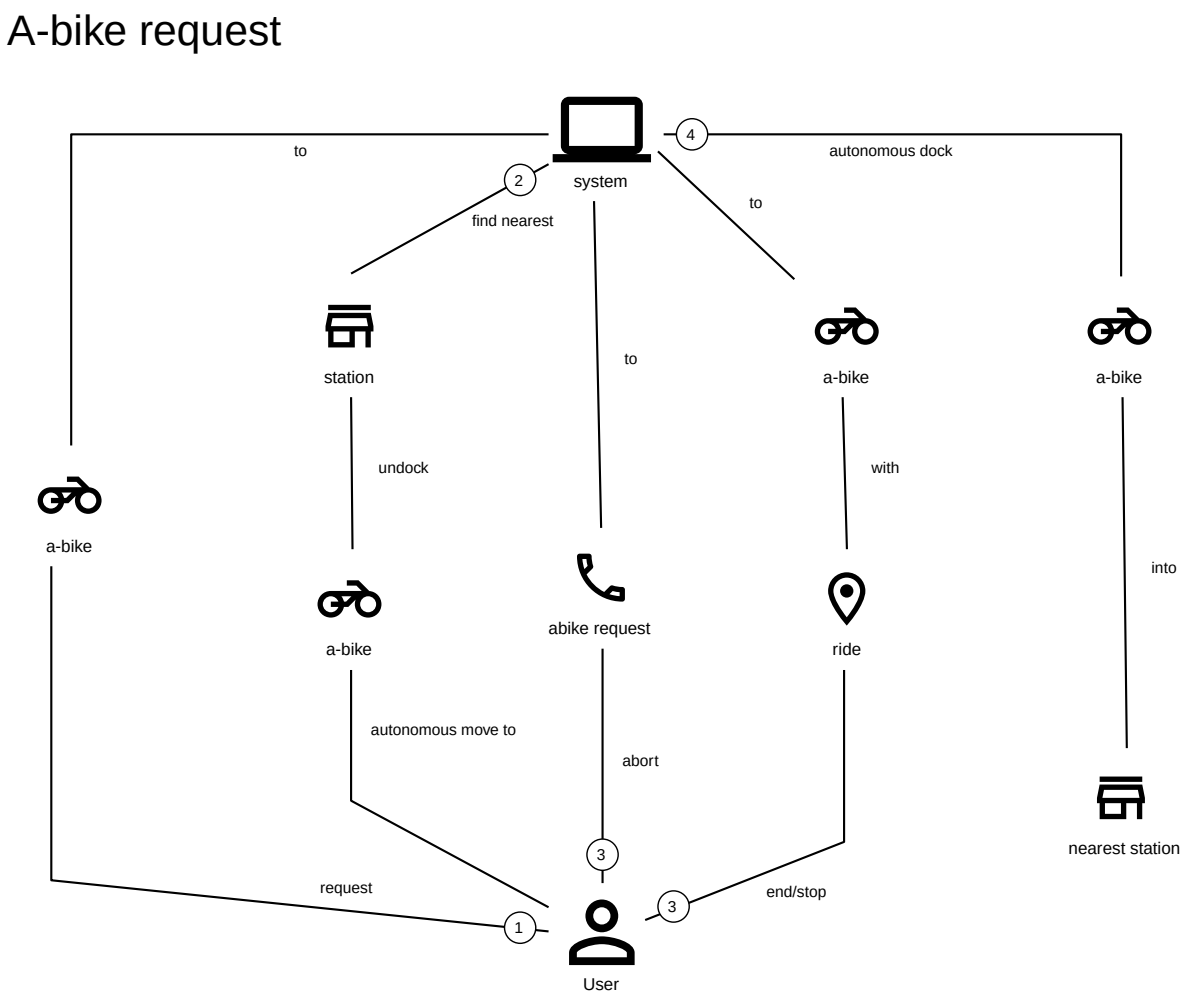
Figure 4. Use-case diagram relativo alle automazioni di sistema

Campo	Descrizione
Use Case	A-bike movimento autonomo verso utente
Attori	Sistema
Scope	Applicazione Smart City – Automazione
Livello	Medio
Precondizioni	Il sistema ha ricevuto una richiesta valida di a-bike da parte di un utente
Postcondizioni	La a-bike si muove autonomamente verso la posizione dell’utente e aggiorna il suo stato durante il percorso
Trigger	Il sistema riceve il comando di muovere la a-bike verso l’utente
Flusso	. Il sistema individua la posizione dell’utente . Calcola il percorso ottimale . Comanda la a-bike a spostarsi autonomamente verso l’utente . Aggiorna in tempo reale la posizione della a-bike

Campo	Descrizione
Use Case	A-bike docking autonomo
Attori	Sistema

Campo	Descrizione
Scope	Applicazione Smart City – Automazione
Livello	Medio
Precondizioni	La corsa con la a-bike è terminata, oppure la richiesta è stata annullata
Postcondizioni	La a-bike si dirige autonomamente verso la stazione più vicina con slot disponibile e aggiorna il proprio stato
Trigger	Termine corsa o annullamento richiesta
Flusso	. Il sistema rileva la necessità di docking automatico . Calcola la stazione più vicina disponibile . Comanda la a-bike a dirigersi autonomamente verso la stazione . Aggiorna la disponibilità della stazione e la posizione della a-bike

4.3. Domain storytelling



Domain Storytelling – Flusso utente per l'utilizzo di una a-bike autonoma: Il diagramma rappresenta il percorso tipico di un utente che richiede una a-bike, dal momento della richiesta fino alla conclusione della corsa. Mostra come il sistema individua la stazione più vicina, sgancia la bici, gestisce il movimento autonomo verso l'utente e il ritorno automatico della bici alla stazione al termine della ride o in caso di annullamento.

4.4. Bounded Context

In seguito all'estensione del sistema per supportare le biciclette autonome e la componente Digital Twin, sono stati individuati i seguenti nuovi bounded context:

Bounded Context	Descrizione
Station Management	Gestisce le stazioni di parcheggio e ricarica per abike. Mantiene lo stato, la capacità e la disponibilità delle stazioni. Gestisce tutte le operazioni di docking e undocking delle bici e pubblica eventi di aggiornamento ufficiali relativi alle stazioni.
ABike Management	Rappresenta il dominio delle biciclette autonome (a-bike). Gestisce il ciclo di vita, il movimento automatico (verso stazione o utente). Mantiene una proiezione locale delle stazioni per ottimizzare le decisioni di mobilità, ma delega sempre la validazione delle operazioni critiche (ad es. undocking) a Station Management.
Digital Twin Integration	Adatta e traduce gli eventi di aggiornamento provenienti da ABike Management e Station Management verso la piattaforma Digital Twin (es. Eclipse Ditto), mantenendo una vista digitale e aggiornata della situazione reale del sistema. Questo context funge da adattatore e sincronizzatore tra il sistema event-driven e l'ambiente smart city virtuale.

4.5. Domain model

In seguito verranno definiti i domain model dei tre context individuati

4.5.1. Station Management

Aggregate Root:

- *Station*: Rappresenta una stazione fisica dove le a-bike possono essere parcheggiate e ricaricate. Gestisce la propria capacità e mantiene traccia delle biciclette attualmente ancorate.

Value Objects:

- *P2d*: Rappresenta un punto bidimensionale (x,y) che definisce la posizione geografica di una stazione.

Domain Events:

- *BikeDockedEvent*: Generato quando una a-bike viene ancorata con successo a una stazione.
- *BikeReleasedEvent*: Generato quando una a-bike viene rilasciata da una stazione.
- *CreateStationEvent*: Generato quando una nuova stazione viene creata nel sistema.

Repository:

- *StationRepository*: Gestisce la persistenza delle stazioni

4.5.2. ABike Management

Aggregate Root:

- *ABike*: Rappresenta una bicicletta autonoma con capacità di movimento senza conducente. Mantiene informazioni sulla posizione, livello batteria e stato operativo corrente.

Value Objects:

- *P2d*: Rappresenta un punto bidimensionale (x,y) utilizzato per la posizione della bicicletta.
- *ABikeState*: Enumeration che definisce gli stati possibili di una a-bike: AVAILABLE (disponibile), AUTONOMOUS_MOVING (in movimento autonomo), IN_USE (in utilizzo), MAINTENANCE (in manutenzione).
- *Purpose*: Enumeration che definisce lo scopo di un movimento autonomo: TO_USER (verso l'utente) o TO_STATION (verso una stazione).

Entities:

- *Destination*: Rappresenta una destinazione con posizione e identificativo, può essere un utente o una stazione.

Domain Services:

- *Simulation*: Servizio di dominio che gestisce il movimento autonomo di una a-bike verso una destinazione. Calcola il percorso, aggiorna la posizione e notifica eventi di posizione fino al raggiungimento della meta.

Domain Events:

- *ABikeCreateEvent*: Generato quando una nuova a-bike viene creata nel sistema.
- *ABikeUpdate*: Generato quando le proprietà di una a-bike vengono aggiornate (posizione, stato, ecc.).
- *ABikeArrivedToUser*: Generato quando una a-bike raggiunge con successo un utente.
- *ABikeArrivedToStation*: Generato quando una a-bike raggiunge con successo una stazione.
- *CallAbikeEvent*: Generato quando un utente richiede una a-bike.
- *CancellCallRequest*: Generato quando un utente annulla una richiesta di a-bike.
- *ABikeCallComplete*: Generato quando una richiesta di a-bike viene completata.

Repository:

- *SimulationRepository*: Gestisce la persistenza delle simulazioni di movimento autonomo
- *ABikeRepository*: Gestisce la persistenza delle abike

Aggregate Projections:

- *ABike*: Rappresentazione locale dell'aggregato ABike, contenente identificativo, posizione, livello batteria e stato operativo.
- *Station*: Rappresentazione locale dell'aggregato Station, con informazioni su posizione, capacità e biciclette attualmente ancorate.

Value Objects:

- *Location*: Rappresenta un punto bidimensionale (x,y) utilizzato per definire la posizione di a-bike e stazioni.

Domain Events:

- *ABikeUpdateEvent*: Evento che trasporta aggiornamenti relativi a una a-bike, incluso il suo stato completo.
- *StationUpdateEvent*: Evento che trasporta aggiornamenti relativi a una stazione, incluso il suo stato completo.

Domain Services:

- *DittoTranslatorService*: Servizio core che gestisce la traduzione degli eventi di dominio in messaggi compatibili con Eclipse Ditto. Mantiene uno stato dei "thingId" già noti per ottimizzare le comunicazioni.

Factory:

- *DittoEventFactory*: Factory specializzata nella creazione di messaggi nel formato richiesto da Ditto, sia per la creazione che per l'aggiornamento di entità.

4.6. Implementazione dei microservizi

4.6.1. Station Microservice

Approccio Event-Driven e gestione del dominio

- **Event-Driven**: La comunicazione con altri microservizi avviene esclusivamente tramite eventi Kafka e, per le operazioni amministrative, tramite API REST.
- **Gestione dello stato**: Lo stato di ciascuna stazione (posizione, capacità, biciclette attraccate) è mantenuto in modo persistente e aggiornato tramite eventi, con la validazione locale di tutte le operazioni critiche (docking/undocking).
- **Integrazione proiezioni**: Il servizio mantiene la fonte di verità riguardo la disponibilità delle stazioni, pubblicando update ufficiali verso altri context e microservizi che mantengono proiezioni locali.

Modello a Eventi

Il microservizio produce e consuma diversi eventi di dominio:

- `StationRegisteredEvent`: Emesso quando viene creata una nuova stazione.
- `StationUpdateEvent`: Pubblicato dopo ogni aggiornamento di stato (docking/undocking di una bici).
- `BikeDockedEvent`: Consumato per gestire l'attracco di una bici a una stazione.
- `BikeReleasedEvent`: Consumato per gestire il rilascio (undocking) di una bici da una stazione.
- `CreateStationEvent`: Consumato quando si richiede la creazione di una nuova stazione (es. da API REST).

Tutti gli eventi sono implementati come `record` Java immutabili, garantendo tracciabilità, integrità e auditabilità delle operazioni.

Adapter di Comunicazione

1. **API REST (sincrone)** tramite `RESTStationAdapter`:
   - Creazione di nuove stazioni (`POST /api/stations`)
   - Health check (`GET /health`)
   - Esposizione metriche (`GET /metrics`)
2. **Comunicazione asincrona con Kafka**:
   - **Consumer**: Ascolta il topic `"abike-events"` per ricevere richieste di docking/undocking da a-bike.
   - **Producer**: Pubblica eventi di registrazione e aggiornamento su `"station-events"` per sincronizzare le proiezioni negli altri microservizi.

Flusso Operativo

1. **Gestione creazione stazione**:
   - Riceve richiesta REST o evento di creazione.
   - Valida e salva la nuova stazione.
   - Pubblica `StationRegisteredEvent`.
2. **Gestione docking/undocking**:
   - Consuma eventi di docking/undocking da a-bike.
   - Valida l'operazione sullo stato locale.
   - Aggiorna lo stato della stazione e pubblica `StationUpdateEvent`.
3. **Inizializzazione**:
   - All'avvio, il servizio carica tutte le stazioni dal database e pubblica un evento per ciascuna, permettendo la sincronizzazione delle proiezioni negli altri context.

Persistenza e Repository

La persistenza delle stazioni è affidata a un repository MongoDB, implementato tramite `MongoRepository` che fornisce: - Salvataggio e aggiornamento di entità Station. - Recupero di tutte le stazioni. - Mappatura tra documenti MongoDB e oggetti dominio (Station, P2d).

Il microservizio Station non accede direttamente ad altri bounded context, ma interagisce solo tramite eventi, mantenendo basso l'accoppiamento e assicurando la coerenza tramite la validazione locale delle operazioni di docking.

4.6.2. ABike Microservice

Il microservizio ABike costituisce il bounded context dedicato alla gestione delle biciclette autonome, orchestrando tutte le logiche di movimento automatico, assegnazione e stato delle a-bike nel sistema smart city.

Approccio Event-Driven e gestione del dominio

- **Event-Driven**: Tutta la logica di comunicazione con altri microservizi (Station, Ride, Map, ecc.) è basata su eventi pubblicati e consumati tramite Kafka, in particolare su topic come `"abike-events"`, `"station-events"` e `"ride-events"`.
- **Proiezioni locali**: Il servizio mantiene una proiezione locale delle stazioni per selezionare sempre la più vicina quando necessario

(ad esempio, per il ritorno della bici in stazione o per la chiamata da parte di un utente).

- **Movimento autonomo**: Il movimento automatico delle a-bike (verso utente o stazione) è simulato da un servizio di dominio (`Simulation`) che aggiorna posizione e stato ogni tick, pubblicando gli eventi di avanzamento.

Modello a Eventi

Il microservizio produce e consuma diversi eventi di dominio:

- `ABikeCreateEvent`: Segnala la creazione di una nuova a-bike nel sistema.
- `CallAbikeEvent`: Segnala la richiesta di una a-bike da parte di un utente (con destinazione utente).
- `ABikeRequested`: Evento di inizio movimento verso utente.
- `CancellCallRequest`: Segnala la cancellazione di una chiamata da parte dell'utente.
- `ABikeUpdate`: Aggiornamenti continui sullo stato e posizione della bici.
- `ABikeArrivedToUser`: Raggiungimento della destinazione utente.
- `ABikeArrivedToStation`: Raggiungimento di una stazione.
- `ABikeCallComplete`: Completamento del ciclo di chiamata.
- `RequestStationUpdate`: Sincronizzazione proiezioni di stazione.

Tutti gli eventi sono implementati come `record` Java, garantendo immutabilità e tracciabilità.

Adapter di Comunicazione

1. **API REST (sincrone)** tramite `RESTABikeAdapter`:
   - Creazione di a-bike (`POST /api/abikes/create`)
   - Richiesta a-bike (`POST /api/callAbike`)
   - Annullamento richiesta (`POST /api/cancelCall`)
   - Health check e metriche
2. **Comunicazione asincrona con Kafka**:
   - **Producer**: pubblica tutti gli eventi del ciclo di vita della a-bike su `"abike-events"`.
   - **Consumer**: ascolta eventi da `"station-events"` (per mantenere aggiornata la proiezione locale delle stazioni) e da `"ride-events"` (per intercettare le richeiste di update dello stato delle a-bike durante una ride).

Flusso Operativo

1. **Creazione a-bike**:
   - Riceve richiesta REST e genera l'evento di dominio `ABikeCreateEvent`.
   - Recupera la posizione iniziale dalla stazione selezionata.
   - Crea la nuova a-bike (stato AVAILABLE), la salva e pubblica evento di creazione e update. Una nuova a-bike diventa a tutti glie effetti disponibile se Station microservice approva il docking della a-bike nella stazione definita e rilascia uno `StationUpdateEvent`.
2. **Chiamata a-bike (callAbike)**:
   - Riceve richiesta (REST/evento) con destinazione utente `CallAbikeEvent`.
   - Seleziona la stazione più vicina con bici disponibile.
   - Pubblica gli eventi `AbikeRequested` e `CallAbikeEvent` per segnalare la richiesta di undock della a-bike a station e la richiesta di pagamento per la chiamata all' utente.
   - Assegna una a-bike e crea una nuova simulazione con `Purpose.TO_USER`, iniziando a generare update.
   - Aggiorna ciclicamente posizione e stato, pubblicando eventi di update `ABikeUpdate`.
   - Al termine, pubblica evento di arrivo (`ABikeArrivedToUser`).

3. **Fine corsa e ritorno in stazione**:
    - Al termine di una ride viene intercettato l'evento di stop.
    - Identifica la stazione più vicina con spazio disponibile.
    - Simula il movimento della bici verso la stazione (creando una nuova simulazione con `Purpose.TO_STATION`) e pubblica eventi di update e arrivo (`ABikeUpdate`, `ABikeArrivedToStation`).
    - Pubblica l'evento di completamento della call `ABikeCallComplete`.
4. **Annullamento richiesta**:
    - Riceve richiesta di cancellazione (REST/evento `CancellCallRequest`).
    - Interrompe la simulazione in corso (se presente).
    - Avvia il ritorno automatico della bici alla stazione più vicina (nuova simulazione con `Purpose.TO_STATION`).
    - Al termine esegue la procedura standard di termine chiamata descritta al punto 3
5. **Proiezioni locali di stazione**:
    - Sincronizza lo stato delle stazioni ascoltando eventi `"StationRegisteredEventAvro"` e `"StationUpdateEventAvro"` da Kafka, mantenendo aggiornata la lista interna.

Persistenza e Repository

- **ABikeRepository**: persistenza delle a-bike (posizione, stato, batteria).
- **StationProjectionRepository**: persistenza e aggiornamento delle proiezioni delle stazioni.
- **SimulationRepository**: gestione delle simulazioni attive per ogni a-bike.

4.6.3. Ride Microservice

Il microservizio Ride mantiene l'approccio event-driven descritto nella prima versione dell'architettura, ma è stato esteso e adattato per supportare la nuova tipologia di bicicletta autonoma (ABike) accanto alle EBike tradizionali.

Estensione del Domain Model

Per astrarre la gestione di EBike e ABike, sono stati introdotti i seguenti elementi:

- **Interfaccia Bike**: rappresenta un contratto comune che consente al microservizio di gestire in modo generico le istanze di EBike e ABike nelle proiezioni locali.
- **Classe ABike**: aggiunta come implementazione di Bike, con i propri attributi specifici (in particolare, lo stato di movimento autonomo).
- **Enum BikeState**: ora contiene gli stati di EBike e ABike, consentendo di uniformare la gestione dello stato durante la simulazione e le ride.

Queste astrazioni permettono di utilizzare la stessa logica di proiezione, simulazione e verifica sia per ebike che per abike, mantenendo separata la gestione degli eventi specifici.

Eventi di Dominio Aggiornati

Gli eventi di dominio sono stati estesi come segue:

- **Eventi comuni**: `RideStartEvent`, `RideStopEvent`, `RequestRideEndEvent`, `UserUpdateEvent`
- **Eventi specifici EBike**: `EBikeUpdateEvent`, `RideUpdateEBikeEvent`
- **Eventi specifici ABike**: `ABikeUpdateEvent`, `RideUpdateABikeEvent`

Per quanto riguarda gli eventi pubblicati sul topic Kafka, questi vengono ora completamente separati in base al tipo di bicicletta, come evidenziato dagli schemi Avro. Eventi di dominio generici come `RideStartEvent` e `RideStopEvent` sono stati differenziati a livello di adapter gestendoli con schemi Avro specifici per tipo di bike.

Proiezione Locale

Il microservizio mantiene proiezioni locali tramite il `LocalProjectionRepository`, che implementa il pattern Event Sourcing per:

- **Utenti**: Tracciati in una mappa con chiave username, dove ogni elemento contiene una lista cronologica di `UserUpdateEvent`. Lo stato corrente dell'utente viene ricostruito recuperando l'evento più recente.
- **Biciclette**: Gestite attraverso l'interfaccia `Bike` che unifica il comportamento, ma memorizzate in una mappa con chiave composta (id + tipo) che distingue tra:
  - `EBike`: Popolata tramite eventi `EBikeUpdateEvent`
  - `ABike`: Popolata tramite eventi `ABikeUpdateEvent`

Ogni richiesta di informazioni su un'entità viene soddisfatta asincronamente (`CompletableFuture`), recuperando l'evento più recente dalla cronologia pertinente e ricostruendo lo stato corrente dell'oggetto di dominio.

4.6.4. Map microservice

Questo microservizio è stato adattato a processare anche i vari eventi generati da Station e ABike. La gestione dedlle E-Bike e A-Bike è stata resa generica come nel microservizio Ride.

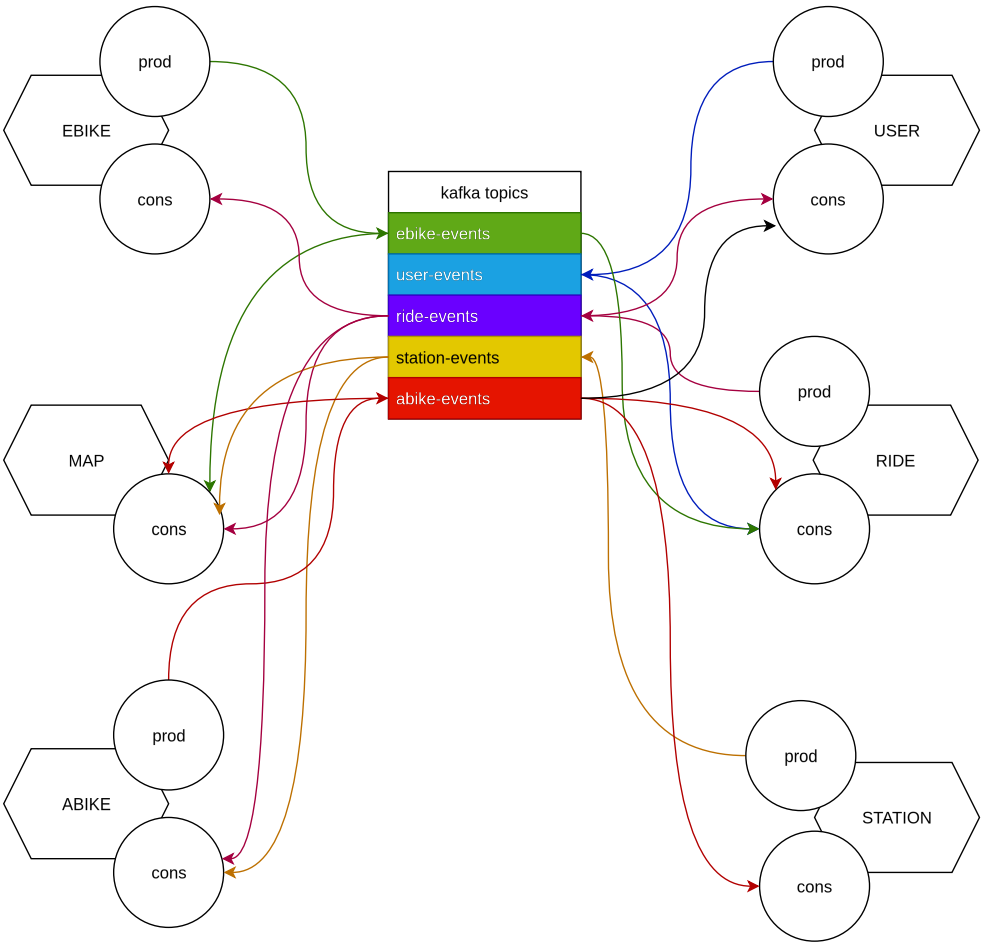
4.6.5. User microservice

Questo microservizio è stato adattato a processare `CallAbikeEvent` per decrementare il credito dell'utente che ha effettuato la chiamata.

4.7. Organizzazione topic Kafka e nuovo flusso di comunicazione

La logica di creazione dei topic di kafka rispetta la medesima del p-1 realizzando quindi due nuovi topic:

- Il topic `abike-events` che veicola gli update relativi all'intero ciclo di vita di una abike.
- Il topic `station-evetns` che veicola gli update relativi alle stazioni.



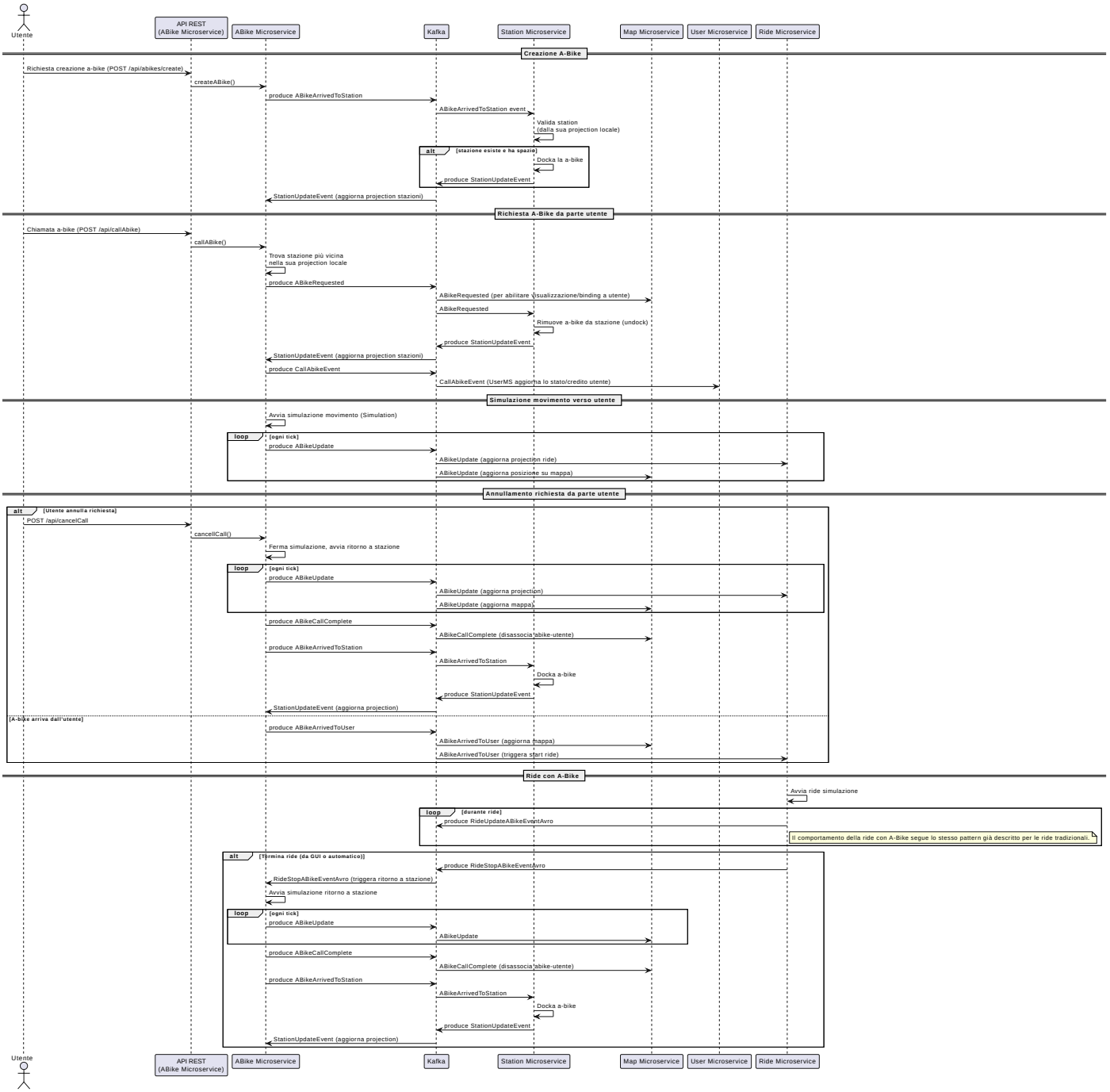


Figure 5. Diagramma di sequenza che rappresenta il ciclo di eventi introdotti da a-bike

Il flusso di eventi non descritto nel diagramma è da considerarsi identico al punto p-1, visto che ride-microservice è stato reso generico dal punto di vista di una bike ed è stato adattato il suo adapter consumer a processare nel modo corretto i nuovi eventi prodotti.

4.8. Integrazione Eclipse Ditto & Digital Twin Adapter

Per abilitare la sincronizzazione tra la flotta di a-bike/stazioni fisiche e la loro rappresentazione digitale in ambiente smart city, è stata integrata la piattaforma **Eclipse Ditto** come Digital Twin framework.

4.8.1. Architettura e orchestrazione

L'infrastruttura include i servizi principali di Ditto (gateway, things, policies, connectivity, search, UI) deployati tramite Docker Compose insieme ai microservizi core. Tutte le componenti condividono una rete bridge dedicata per garantire isolamento e connettività diretta ai servizi Kafka e MongoDB condivisi.

Il Digital Twin si interfaccia con l'ecosistema event-driven tramite un microservizio denominato **ditto-kafka-adapter**, che svolge un ruolo di **translator** e **bridge** tra i topic Kafka (dove i microservizi pubblicano eventi di dominio serializzati con Avro) e le API di Ditto.

4.8.2. Flusso dati e ruolo del microservizio adapter

ditto-kafka-adapter svolge due ruoli principali:

1. **Ricezione eventi da Kafka**
   - Sottoscrive i topic Kafka `abike-events` e `station-events`.
   - Consuma in particolare eventi di tipo `ABikeUpdate` e `StationUpdateEvent` prodotti dai rispettivi microservizi.
   - Alla ricezione di questi eventi, converte i dati di dominio in **comandi Ditto** (formato JSON), rispettando le specifiche delle **Ditto Things** (gemelli digitali di a-bike e stazioni).
   - Gestisce in modo trasparente sia la creazione (comando "create") che l'aggiornamento periodico ("modify") dei gemelli digitali, pubblicando i messaggi su Kafka nel topic dedicato a Ditto `ditto-messages`.
2. **Gestione eventi di ritorno da Ditto**
   - Implementata (per scopi di test/estendibilità) la ricezione di eventi/risposte provenienti dal topic Kafka `ditto-commands`, normalmente prodotti da Ditto in seguito a modifiche o richieste verso i suoi gemelli digitali.
   - Il microservizio elabora questi messaggi, estrae identificativi, correlation-id e invia (tramite le API di Ditto o di nuovo via Kafka) eventuali risposte o conferme di avvenuta esecuzione, secondo lo standard Ditto.

4.8.3. Pipeline operativa

1. **Aggiornamento Digital Twin**
   - Ogni volta che un evento di aggiornamento (ad esempio una nuova posizione di una a-bike o lo stato di una stazione) viene emesso su Kafka, il **ditto-kafka-adapter** lo intercetta, lo converte e lo inoltra verso Ditto.
   - Se il gemello digitale della a-bike o della stazione non esiste ancora, il microservizio provvede automaticamente alla creazione, altrimenti effettua un update.
2. **Ciclo completo di Digital Twin**
   - Tutte le modifiche di stato, posizione, batteria, capacità delle stazioni sono sincronizzate in near real-time tra la parte fisica (microservizi) e la rappresentazione digitale (Ditto Things).
   - È stata validata la possibilità di ricevere comandi/risposte da Ditto tramite Kafka, aprendo a scenari di controllo bidirezionale.

Chapter 5. Conclusioni

Questo assignment si è rivelato particolarmente interessante e formativo. Ho avuto la possibilità di mettere in pratica molti concetti affrontati durante il corso, ma soprattutto di approfondire tecnologie che non conoscevo o che avevo solo incontrato superficialmente, come Kafka, Avro ed Eclipse Ditto.

L'approccio event-driven mi ha permesso di comprendere in modo molto più chiaro e concreto i vantaggi dell'architettura a microservizi basata su eventi, sia dal punto di vista della scalabilità che della resilienza. La gestione delle comunicazioni tramite Kafka e la serializzazione con Avro sono state sfide inizialmente complesse, ma estremamente utili per rafforzare le mie competenze pratiche.

Inoltre, la parte di deployment su Kubernetes mi ha permesso di acquisire maggiore familiarità con l'orchestrazione di applicazioni containerizzate in ambienti simili al mondo reale.

Infine, è stato stimolante poter integrare la tematica dei Digital Twin tramite Eclipse Ditto.

In sintesi, considero questa esperienza una tappa importante per la mia crescita tecnica e una base solida per affrontare progetti di maggiore complessità in futuro.